

Physics 430 — Digital Electronics and Computer Interfacing  
Instructor: Neil Alberding  
Fall 2001 Lecture Notes Table of Contents

Logic .....	4
Electronic Logic.....	5
Types of Logic Gates.....	7
Transmission Gate Logic (TGL) .....	7
Mickey Mouse Logic (M2L).....	8
Diode Transistor Logic (DTL).....	8
Resistor Transistor Logic (RTL) .....	8
Transistor Transistor Logic (TTL) .....	9
Complementary Metal Oxide Semiconductor (CMOS) .....	10
Electronic Arithmetic.....	12
Circuits for Arbitrary Functions.....	14
Flip-Flops and Memory .....	14
More on Flip-Flops .....	17
Microprocessor Architecture.....	30
Tips for Constructing Wire-Wrap Circuits .....	32
Hardware Debugging Hints .....	32
Communication of the Microprocessor with the Outside World.....	33
Decoding the Address .....	36
Latching and Buffering the Data.....	38
The 68000 Series of Microprocessors.....	39
The PDP-11 unibus.....	43
The 8255 Programmable Peripheral Interface.....	45
computer-side lines.....	45
peripheral-side lines.....	45
Mode 0.....	45
Mode 1 of the 8255.....	47
The 8253 Programmable Interval Timer.....	51
Programming the control register.....	51
The six modes of the 8253.....	51
Mode 0 .....	51
Mode 1 .....	51
Mode 2 .....	51
Mode 3 .....	51
Mode 4 .....	51
Mode 5 .....	51

## Table of Contents Continued

The Real World.....	53
Digital -to-Analog Conversion .....	53
Analog-to-Digital Conversion.....	56
Other A/D Conversion Methods .....	57
Stepping Motors.....	63
Unifilar and Bifilar Stepping Motor and their Drives .....	63
Ramping, Damping and Vibration.....	67
Transients and Their Suppression.....	68
Hardware Interrupts .....	71
Interrupt Circuit.....	75
Direct Memory Access .....	77
DMA, the Software Side.....	80
The IEEE-488 Instrumentation Bus.....	83
The Talker, The Listeners and the Controller.....	84
Choosing Talkers and Listeners .....	85
The Three Wire Handshake.....	86
Other Control Lines.....	87
You Didn't Really Have to Know.....	88
But If You're Really Stuck.....	90

## Physics 430–Digital Electronics and Computer Interfacing Fall 2000 Lecture Notes



### Computers, Physics and Philosophy

Physics is a branch of philosophy, or at least it used to be. “Natural Philosophy” is an old name for physics and it implies the philosophical exploration of the natural world. These days there are still some who go into physics out of the love of studying and understanding what goes on in the world. This undertaking will never reach its ultimate goal. We’ll probably never really understand everything and its most fundamental nature. We persist because it’s not the destination, but the journey that counts. In this journey of exploration we do turn up things that are of interest in a mundane and practical way—things like better ways of sending messages, faster and cheaper transportation, improved building methods, more efficient calculating machines, more destructive weapons. These spin-offs of our philosophical journey benefit people who can exploit these discoveries for profit. These technological spin-offs belong to the realm of technology, not science. They can help finance our philosophy, but they shouldn’t be confused with it.

So here we are in a physics course learning about electronics and computer interfacing. Is this Technology or Science? Because this is a physics lecture we can start a little more philosophically than might be expected in a more engineering-oriented lecture. Don’t worry though. We won’t stray too long on this path.

\*           \*           \*

Computers. Computer Science. These words are symbols. It is sometimes useful to examine our symbols before we start to use them. People go astray when they confuse the symbol with the thing. The English verbal symbol (word) “computer” for this machine implies something that takes a bunch of numbers, adds them, subtracts them multiplies, divides, integrates, differentiates... Computers do all these things, but they also do much more. The question arises, “is ‘computer’ a good symbol.” If not, well, it’s too late to change it, but how does this choice of symbol influence how we deal with it? Similarly, how does attaching the word “science” to “computer” influence our dealings with them?

The English-speaking world is careless about how it uses words. We tend to pick up words from anyplace, without thinking, and start to use them right away. The French are far more careful with their language. They have an Academy that scrutinizes every new word added to the vocabulary. In French they use the word “*ordinateur*” instead of computer. This has the connotation of ordering or organizing things. That is much more general than the meaning of “computer” which points out only a small fraction of what these machines are used for.

Similarly, the study of computers, “computer science,” in English carries with it a certain sense that we’re studying the fundamental laws of these machines which we humans have made. This seems a little ridiculous because if we’ve made them, then we surely don’t have to take them apart to find out how they work, do we? In French this discipline is called more aptly, “*l’informatique*.” This implies a study of information, how to get it, how to organize it, how to use it, etc. I think one should keep in mind that computer science is a study of information and its interaction with ourselves and the world, and that computers are machines for dealing with information.

\*           \*           \*

Information: what’s that? Ask yourself what is the smallest piece of information that is possible to convey. This is what we usually call a “bit.” \* Imagine one of those isolation experiments where somebody puts himself in a room alone with no contact to the outside world. Now let’s imagine that we will allow only the smallest quantum of information to be passed to this person. First of all there must be an agreement as to what this information will be about. For example, it might be the time or, perhaps, information about the weather. What would be the smallest imaginable amount of time information? What about telling him whether it’s day or night? You can’t be much less informative than that and still give information. Here we’re distinguishing between two possible conditions, day and night, which can’t happen at the same time. If we were to give information about the weather, it might be to say if it’s cool or warm. We might arbitrarily define cool as below 20° C and warm as above 20° C or something like that. Again we first pick what our information will be about, then we divide it into two, mutually exclusive conditions it

---

\* In English we use the word “information” in the singular to designate a measurable quantity like milk. In French, “*informations*” is plural: a countable quantity like beads.

can be in. The information will be to specify which of these two conditions is true.

There are many possible mechanisms for passing this information. We could have a rod through the wall into the isolation chamber. When the rod is pushed in it's day. Pulling the rod out would mean night. Or we could have a wire with one voltage on it for day, and another for night. In both cases we are using something with two states: in/out, high/low, on/off. Whatever we use will be a *symbol* for the day/nightness of the time. As long as the persons inside and outside the room agree to the meaning of this symbol, information can be transmitted.

It's possible to extend this experiment to pass information about many more things. In each case, the starting point is dividing the subject into two polar states, then symbolizing these states by a device. The means of conveying the information, the symbolic mechanism, can be the same no matter what it is we're informing about. You might say, "This bit tells me the time, this one the weather, and this bit over here tells me whether our side won their last ball game or not." Not only can we have bits for information about different things, but we can add bits to be more precise about something. For example, an extra bit about time might tell us whether or not we are in the first half of the day or the night. By adding more and more bits, we can be as precise as we like.

There seems to be an almost natural division of things into polar opposites. This binary information scheme was recognized in ancient Chinese philosophy by the idea of *yin* and *yang*, the male and female principles.\* Originally *yin* and *yang* stood for the sunny and shady sides of a hill. In philosophy though, these words came to designate the two polarities in which almost all things can be classified. For example bland food is *yin* and hot, spicy food is *yang*. Similar dualities are female/male, soft/hard, negative/positive, shady/sunny, nothing/something, 0/1. It seems that every aspect of life and the world can be categorized as either *yin* or *yang*. Using a binary symbolism to fit the duality of nature has proved very powerful in its application to computers. Various computer designs have proved inferior to the binary digital computer. For example, analog computers represent different numbers by different voltage levels. Today analog computers are an interesting curiosity, whereas digital computers are everywhere.



Chinese characters for *yin* and *yang*.

Computer memories can be viewed then as massive arrays of *yin/yang* symbols. Each one represents the smallest, indivisible quantity of information: a bit. These bits are carried on electrical wires or pc board traces by voltages. A common convention is 0V = *yin*, and 5V = *yang*. On paper we usually write 0=*yin*, and 1=*yang*. As always we must keep in mind that for these symbols to have meaning, there must be some convention as to what each and every symbol refers to: time, weather, colours, tastes, etc.

The first great bureaucracies of history were established after the invention of pen, ink and paper for the symbolic recording of information. These implements allowed recording large amounts of information and storing them for long periods of time. Computers give another dimension to the recording and storage of information: motion or interactions. The symbols can not only be stored, they can be brought together to interact and produce new symbols. Such transformations of symbols can be made to simulate the transformations of things and events in the world. Now, not only can a static state be symbolized, but also the processes of evolution and change of these states from one form to another can be symbolized. In fact we use the word "simulation" to represent a dynamic symbol of a process. Computers, through their ability to record *processes*, i.e., dynamic information, as well as *states*, i.e., static information, are bringing the world, and bureaucracies, through another turning point, another revolution, such as was brought about by pen and paper.

The laws of combination of things with two possible states were first stated in terms of logical propositions. A proposition is statement that can be either true or false. George Boole set forth his "Boolean" algebra in 1847 in a treatise called "*The mathematical analysis of logic*." Boolean algebra is only one instance of things with two possible states and their combinations. When discussing digital and computer design, it's conventional to speak in terms of the algebra of propositions for concreteness. However, our bits of zeros and ones, *yins* and *yangs* don't have to represent propositions which are true or false. They often represent the myriad other things which can be expressed in a binary, 0/1 way—things

---

\* Joseph Needham, *Science and Civilisation in China*

like the existence or nonexistence of the universe, God and the devil.

Let's try to imagine now the most primitive sort of interaction that can occur between bits of information. The simplest transformation is one bit goes in, and one bit comes out. There are four ways this can happen: 1. The same bit comes out that came in, the identity operation, 2. the bit gets changed to its opposite, the inversion 3. the bit is always transformed to 0 or 4. the bit is transformed to 1. Let's let A represent something that can be either 0 or 1 (*yin* or *yang*). Then if A is transformed to B, the four operations are written algebraically as

$$\begin{array}{ll} B = A & B = 0 \\ B = -A & B = 1 \end{array}$$

Now let's investigate the transformation of *two* bits of information. To keep it simple let's say two bits combine to form just one other. Two bits go in, one comes out. There are four ways that two *yin/yang*, 0/1, bits can combine: (1) first bit=0, second bit=0, (2) first bit=0, second bit=1, (3) first bit=1, second bit=0, and (4) first bit=1, second bit=1. The results of each combination can each be either 0 or 1. One type of interaction process would result in a definite outcome for each of the four starting combinations of the two interacting bits. Thus the number of different processes symbolized by the combining of two bits is  $2^4$  or 16. One of them can be illustrated in a table showing the two original bits and the result

bit 1	bit 2	result
0	0	0
0	1	0
1	0	0
1	1	1

There are 16 different ways of writing the result column before we start to repeat.

To Boole, 1 represented True and 0, False. So the table above represented the process of writing an *and* between two propositions to form a third proposition. This new proposition is true only if both the original ones are also True. In writing we could say

let A= bit 1, let B= bit 2, let C=result  
then  $C = A \bullet B$ .

Here the big  $\bullet$  represents the *and*. Similarly the *or* operation would have the "truth table" as follows:

bit 1	bit 2	result
0	0	0
0	1	1
1	0	1
1	1	1

This *or* operation is represented algebraically as  $C=A+B$ .

By combining *ands*, *ors*, and the four one-bit operations you can get all the other 14 two-bit operations.

## Logic

A simple decision about whether to go to lunch can illustrate principles of propositional logic. Let's say that two conditions must be met: (1) you're hungry and (2) you have money, i.e., you are solvent. If both are true at the same time then you can go to lunch. This is the logical AND operation:

(hungry) AND (solvent)    go to lunch

A truth table is used to solve this kind of dilemma. It's just a case of writing down all possible combinations of the conditions and their result.

hungry	solvent	hungry AND solvent
false	false	false
false	true	false
true	false	false
true	true	true

It would also be possible to state the problem in terms of the logical inverses of "hungry" and "solvent," that is "full" and "broke." The truth table can be extended to cope with this.

hungry	solvent	full	broke	full OR broke	NOT (full OR broke)
false	false	true	true	true	false
false	true	true	false	true	false
true	false	false	true	true	false
true	true	false	false	false	true

Being hungry and solvent is the same as not being full or broke. Note that the OR is an inclusive OR, which means that the result is true if either one or both of the conditions is true. Also one must be careful to place the brackets around "full OR broke" to show that the OR is done before the NOT.

Stating the problem in terms of "hungry" and "solvent" could be called positive logic and in terms of "full" and "broke," negative logic. The observation that

$$(\text{hungry AND solvent}) = \text{NOT} (\text{full OR broke})$$

is called DeMorgan's Theorem. DeMorgan's theorem is about the highest level of mathematics we shall use in this course and it can be stated as follows:

A positive logic AND is a negative logic OR and  
a positive logic OR is a negative logic AND.

Symbolically we can use the letters A and B to denote logical conditions. Sometimes the logical AND operation is written like multiplication:  $A \cdot B$  or just  $AB$ . Likewise the OR operation is written like addition:  $A + B$ . The inverse, NOT, is written with a bar over the letter:  $\bar{A}$ , or with a minus sign:  $-A$ . Then DeMorgan's theorem is

$$A \cdot B = -(( -A) + ( -B) ) \text{ and} \\ A + B = -(( -A) \cdot ( -B) ).$$

(Another symbolism uses    for AND,    for OR and ~ for NOT:  $A \cdot B = \sim((\sim A) \cdot (\sim B))$ .)

The set of rules for manipulating logical expressions is called Boolean algebra. Most of the rules are fairly obvious. Here is a list.

$$ABC = (AB)C = A(BC)$$

$$AB = BA$$

$$AA = A$$

$$A \bullet \text{true} = A$$

$$A \bullet \text{false} = \text{false}$$

$$A(B + C) = AB + AC$$

$$A + AB = A$$

$$A + BC = (A + B)(A + C)$$

$$A + B + C = (A + B) + C = A + (B + C)$$

$$A + B = B + A$$

$$A + A = A$$

$$A + \text{true} = \text{true}$$

$$A + \text{false} = A$$

$$\neg \text{true} = \text{false}$$

$$\neg \text{false} = \text{true}$$

$$A + (\neg A) = \text{true}$$

$$A(\neg A) = \text{false}$$

$$\neg(\neg A) = A$$

$$A + (\neg A)B = A + B$$

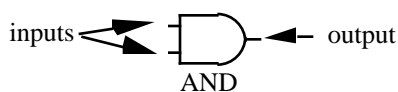
$$\neg(A+B) = (\neg A)(\neg B)$$

$$\neg(AB) = (\neg A) + (\neg B)$$

## Electronic Logic

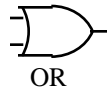
An electronic device can be constructed to make these kind of decisions. Voltage levels with respect to a common ground can be used to represent the logical true or false states, e.g., +5V for true and 0V for false. An AND operation could be done with a circuit with three terminals. Two terminals are for input and one for output. A voltmeter between the output and ground would measure 5V if both inputs were connected to 5V. If either input were at 0V, the output would drop to zero. The "user interface" of such a device would entail connecting each input to a switch so that the inputs could be switched between 0 and 5V at whim. The output could be connected to light a lamp when it is 5V. This interface could be called "user friendly" if the switches were labelled with words like "Hungry", "Full", "Solvent" and "Broke" and the output lamp were labelled, "Go to Lunch."

Without worrying about the details of how to build this circuit, it can be represented by a symbol showing the inputs and output. A little truth table beside it shows its function.



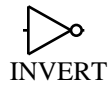
input 1	input 2	output
0V	0V	0V
0V	5V	0V
5V	0V	0V
5V	5V	5V

An OR gate can be diagrammed as follows:



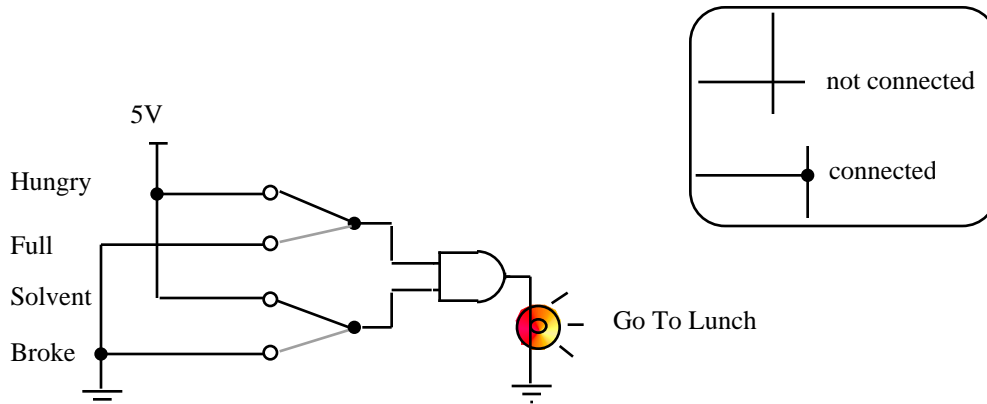
input 1	input 2	output
0V	0V	0V
0V	5V	5V
5V	0V	5V
5V	5V	5V

and a NOT device, or "inverter", is drawn as

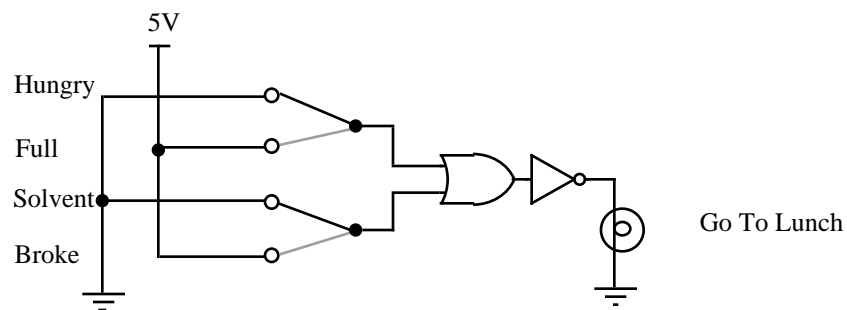


input	output
0V	5V
5V	0V

Our Lunch Decision box could be wired up as below.



It could also be built with negative logic.



The combination of an OR gate with its output connected to an inverter is so common that it is given a special name, the NOR gate. Its symbol is



NOR

Similarly NOT AND is a NAND gate with the symbol

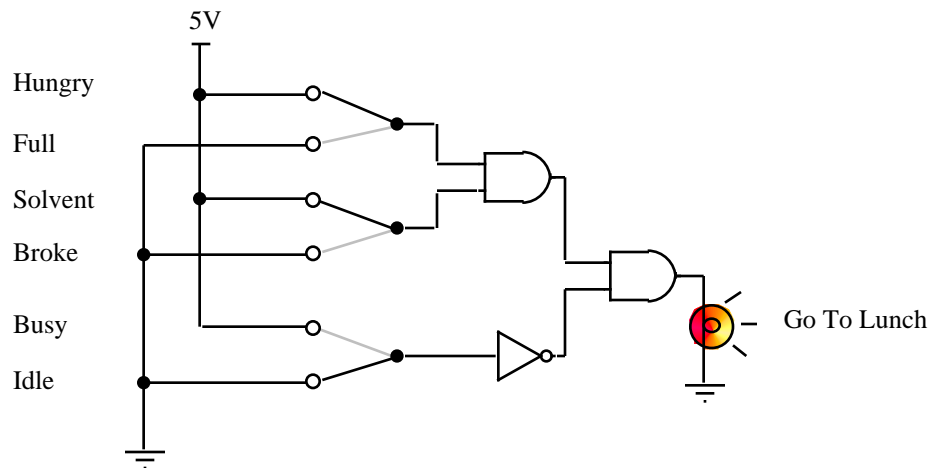


NAND

The ability to feed the output of one gate to the input of another allows constructing circuits for even more complicated decisions. For example

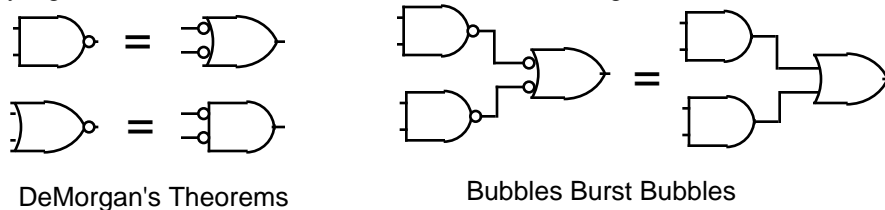


if hungry and solvent and not busy then go to lunch.



Assignment problem: Sketch two other circuits to accomplish the same logic with the same switch connections.

The electronic symbols provide another notation for Boolean expressions. For example, DeMorgan's theorem can be expressed by changing a NAND gate to an OR gate with inverters on both inputs. (Notice that a little circle on a gate's input indicates inversion of the input signal.) NAND gates are often represented by the "bubbled OR" symbol. You can now see that either an AND gate or an OR gate can be made from a NAND and one or two inverters. Also it is very simple to make inverters out of NAND gates. Thus any logic circuit can be made from a collection of NAND gates.



DeMorgan's Theorems

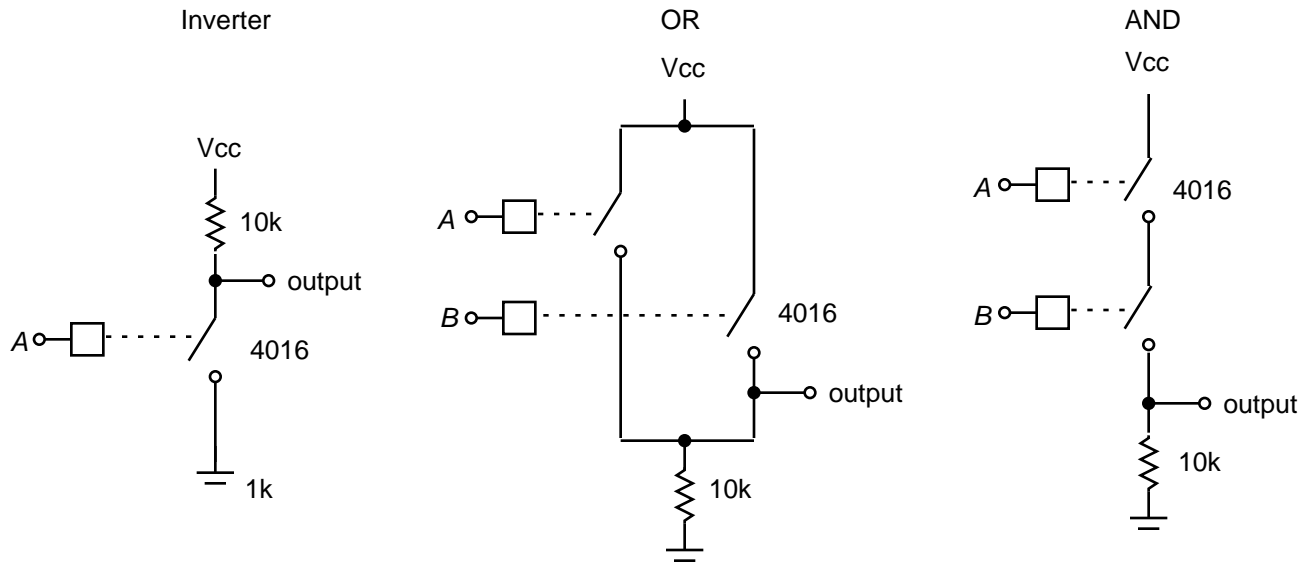
Bubbles Burst Bubbles



NANDs and NORs as NOTs

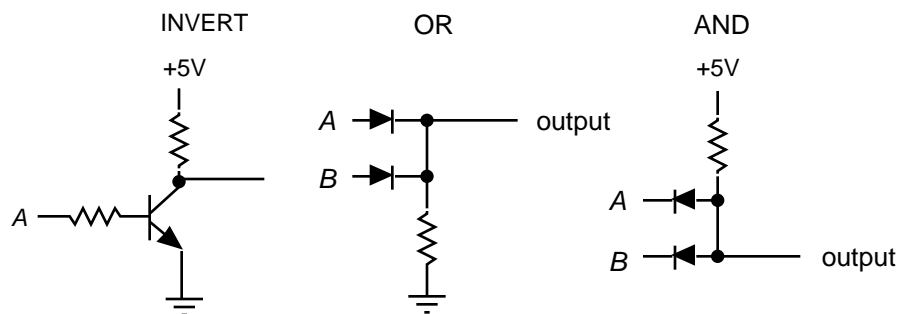
### Types of Logic Gates

*Transmission Gate Logic (TGL)* is the most straightforward. It is just a connection of electrical switches, relays or field effect transistors. A CMOS analog switch chip such as 4016 could be used. With TGL, some input to output connections can be used in reverse: they are bidirectional.



g

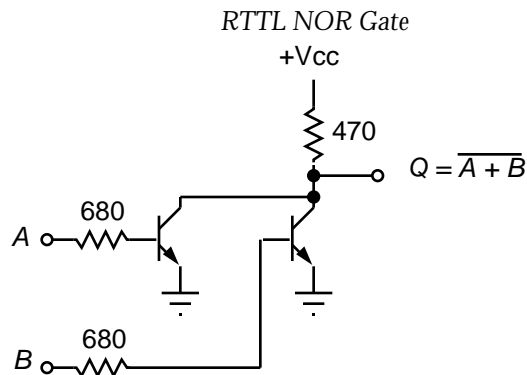
*Mickey Mouse Logic ( $M^2L$ )* uses discrete diodes and transistors.



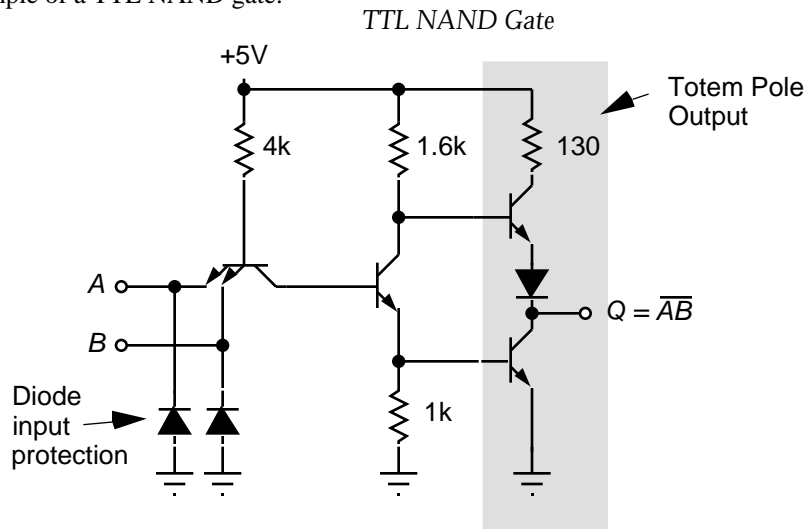
These circuits are not often useful because the 0.6V drop across the diodes degrades the output voltage levels making them not strictly TTL compatible. Furthermore the passive pullup resistors on the outputs slow the response time. These circuits draw a fair amount of current and are limited to relatively slow switching rates. An occasional  $M^2L$  gate is ok, if input levels are not close to the TTL limit, and can save an additional chip.

*Diode Transistor Logic (DTL)* is an old family of logic chips that uses these diode input circuits followed by transistor output circuits. Although DTL is no longer used as such, the newer LS-TTL series uses Schottky diodes and transistors with very similar diode inputs for its NAND and NOR gates.

*Resistor Transistor Logic (RTL)* is an obsolete system that was used before TTL and CMOS logic gates became common. For example here is an RTL OR gate. Notice the similarity to the simple inverter above.



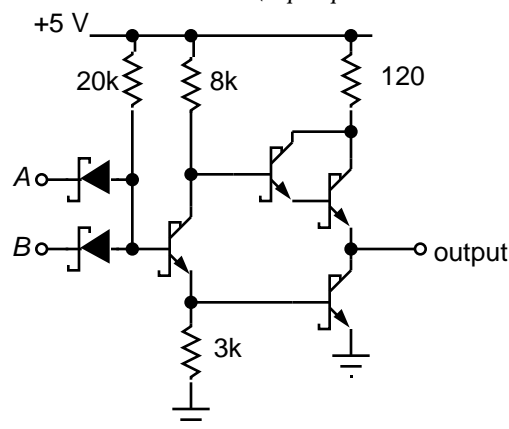
*Transistor Transistor Logic (TTL)* is the basis of Texas Instruments famous 7400 series chips. Here is an example of a TTL NAND gate.



The two-emitter transistor functions somewhat like the diode AND gate. The rest of the circuit acts like an inverter. If one emitter is low the output is high. Both inputs must be high for the output to be low. The input is usually protected from negative voltages by a diode. This also helps to dampen ringing of pulses.

The original 7400 family of TTL chips required 1.6 mA for each input and could provide 16 mA at the output. It is now common to use the low power Schottky family of chips which requires only 0.4 mA for each input, whereas an output can supply 8 mA. These low power Schottky chips are marked with numbers 74LSxx instead of 74xx and use M<sup>2</sup>L-like Schottky diode inputs. In general one should not connect an output to more than ten inputs of chips in the same family—the maximum "fan out" is ten.

*Simplified LS TTL NAND GATE (Input protection diodes not shown.)*



TTL and LS TTL outputs are HIGH or TRUE between 2.4V to 5V and LOW or FALSE from 0V to 0.4V. TTL and LS TTL inputs from 2 V to 5 V are HIGH and from 0.8 and 0 V are LOW. There is a 0.4 V margin for error between the lowest TTL high output and the lowest TTL high input. Similarly there is a 0.4 V difference between the highest TTL low output and the highest TTL low input. This is necessary to allow for voltage drops or noise pickup between the output of one gate and the input of another gate to which it is connected. This margin is sometimes called "noise immunity."

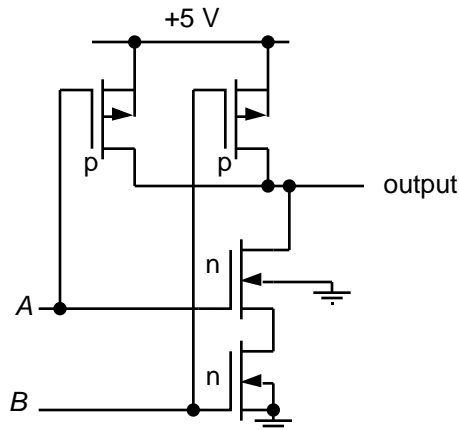
The standard two-transistor output is called a "totem-pole" output. In order for the output to pull an input low, it must be able to pass current, 1.6 mA for regular TTL or 0.4 mA for LS-TTL. On the other hand, a high level does not sink any current. In fact, an unconnected TTL input is high.

The lower transistor of the totem pole is designed to sink more current than the upper transistor

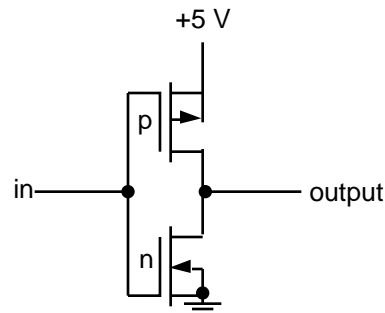
can source. If a light emitting diode is put on the output it is preferable to connect it between the output and a resistor to 5 V rather than between the output and ground. Another practical consideration is that when the gate is switching between high and low levels, both transistors momentarily conduct. This draws more current and can load the voltage supply and ground lines abnormally. To avoid consequent voltage drops, a "despiking" capacitor can be connected between the +5V pin and the ground pin to serve as a reservoir for the chip's momentary current needs. One such despiking capacitor for every three or four chips is usually enough.

*Complementary Metal Oxide Semiconductor (CMOS) Logic* uses field effect transistors (FETs) as switches. n-channel and p-channel FETs are used in complementary pairs. The standard 7400 series of chips is available in CMOS and are labelled 74HCxx. There is an older series of CMOS chips originally manufactured by RCA with numbers 4xxx. These chips are also acceptable but do not follow the 7400 series pin assignments.

**CMOS NAND gate (unbuffered version)**  
Buffered gates are followed by two CMOS inverters.



**CMOS Inverter**

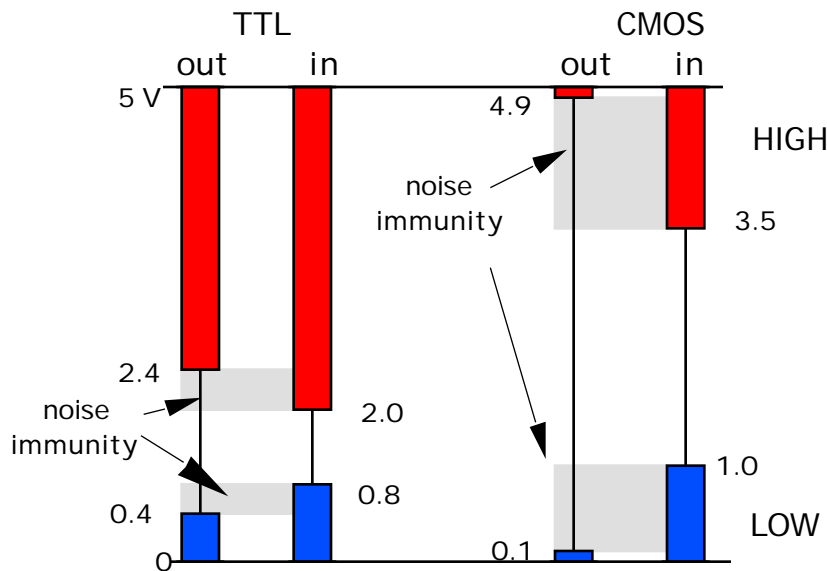


Because of the high impedance input to FETs, CMOS uses almost no current unless it is switching. But the capacitance of the input restricts the high frequency limit to about 10 MHz. TTL can be used for higher frequency up to about 50 MHz. At the high frequency end of the CMOS range, TTL and CMOS require about the same current.

The range of acceptable logic levels for CMOS is different from TTL levels. A CMOS HIGH output is from 4.9 V to 5 V. A CMOS LOW output is between 0.1 and 0 V. An input voltage from 3.5 to 5 V is accepted as HIGH and input levels from 1.0 V to 0 V are accepted as low.

NMOS and PMOS are also used sometimes. They have only n-channel or p-channel FETs respectively. Higher component packing densities are possible at the expense of higher current usage.

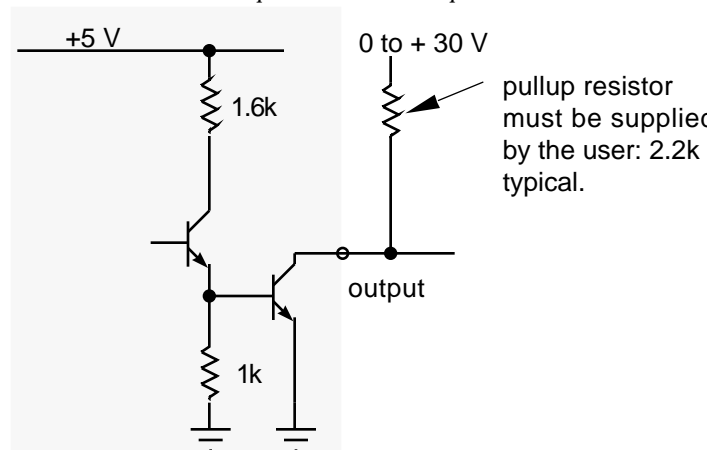
CMOS and TTL logic levels for output and input.



CMOS is sensitive to static discharge. A static spark can ruin a chip. Unused CMOS inputs need to be connected to ground or 5 V. Unconnected CMOS inputs do not default to high like TTL inputs. Unconnected inputs could assume an indeterminate voltage allowing both transistors in the inverter circuits to conduct and to consume a lot of current. CMOS  $V_{CC}$  can be set to any voltage between 3 to 15 V and runs best around 9 V. However, if TTL chips are to be used in the same circuit, a 5 V supply is convenient. If  $V_{CC} = 5$  V then CMOS output levels are compatible with TTL input criteria but a TTL HIGH output level may not fall within the required CMOS input range. Therefore TTL outputs should not be fed directly to CMOS inputs. A special CMOS series labelled 74HCTxx will accommodate TTL input levels.

The totem pole output of TTL gates and the inverter output of buffered CMOS gates are sometimes replaced by open collector or open drain outputs.

Open Collector Output

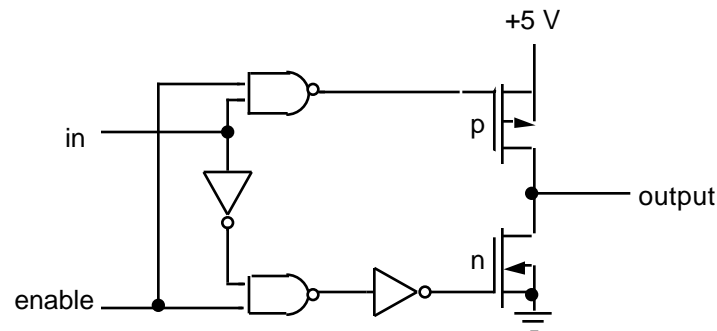


The open collector output is connected through an external resistor to a voltage line. This is useful if an output voltage other than 5 V is required. The resistor can be connected to any voltage level up to 30 V. Several outputs can be connected to the same resistor. If all the outputs are high then the output is high. But if any of the gates connected to that resistor goes low, the output is low. This arrangement is called a "wired or" or sometimes "cheap or." Open collector outputs are commonly used on bussed lines where several devices share the same bus. The quiescent level is high, but any device can take over the line by pulling it low while the other devices are still high. If a totem pole output were so connected, a conflict would arise between the high outputs and low outputs on the bus which could damage the circuit. The CMOS version of open collector is called "open drain" output.

Another scheme for allowing bussed output is so-called "three-state" logic. Here there is a third

input to the gate which is called the "enable" though it is often the inverse–enable. If enable is high, the output is disconnected from the circuit. Only when enable is low is the output actively engaged and acts like a normal TTL output. The trick here is to enable the right gate at the right time.

*Three-State Gate using CMOS transistors and logic gates.*



### Electronic Arithmetic

In order to make logic circuits do arithmetic, we can assign a numerical value to True and False states.

True = 1

False = 0

A sequence of logic levels can represent a number using binary notation. E.g., True,False = 10 (binary) = 2 (decimal). In general let  $A_0, A_1, A_2, \dots, A_n$  be  $n+1$  binary digits which corresponds to  $n+1$  wires at 0V or 5V. Each  $A_i$  is 0 or 1. This sequence of digits or wires represents the number  $X$  given by

$$X = A_0 + A_1 \cdot 2 + A_2 \cdot 2^2 + \dots + A_n \cdot 2^n.$$

Any integer can be represented this way up to a maximum determined by the number of wires available. Each of the numbers from 0 to 15 decimal are represented by four binary numbers as shown in the table.

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

Dealing with long binary numbers is confusing; therefore, we usually group binary digits in fours. Each group of four digits can be represented by one hexadecimal digit which represents a number between 0 and 15 decimal. Thus a 16 digit binary number would be represented by a 4-digit hexadecimal number. If  $H_i$  are the digits of a hexadecimal number then its value is given by

$$X = H_0 + H_1 \cdot 16 + H_2 \cdot 16^2 + \dots + H_h \cdot 16^h.$$

As an example, the binary number

0111 1100 0001 1111

would be represented in hexadecimal as      7      C      1      F .

Its decimal value would be calculated as follows

$$15 + 1 \cdot 16 + 12 \cdot 16^2 + 7 \cdot 16^3 = 15 + 1 \cdot 16 + 12 \cdot 256 + 9 \cdot 4096 = 31775.$$

Negative numbers can be represented by a system called “two’s complement.” One bit must be sacrificed to represent the sign. Therefore, if negative numbers are to be represented with a given number of bits, only half the range of positive numbers can be represented. The recipe to obtain the two’s complement of a positive number is (1) reverse all bits from 1 to 0 or from 0 to 1, and (2) add 1. For example

	0111 1100 0001 1111	= 31775	number to be inverted
(1)	1000 0011 1110 0000		reverse all bits
(2)	1000 0011 1110 0001	= -31775	add one to get the negative.

If you add the binary representation of -31775 to that of +31775 then all bits are zeroed.

A collection of 16 bits could represent a signed number or an unsigned number depending on the established convention. The bits themselves do not carry that information. Thus a person or program might interpret 100001111100001 as 33761 decimal if uninformed or misinformed about how to interpret the bits.

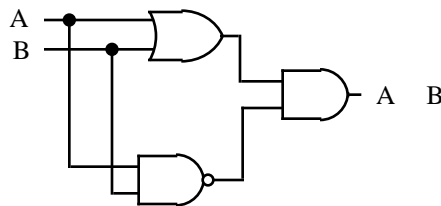
A circuit to add two one-digit (or one bit) binary numbers would have two inputs and two outputs with the following truth table

input	output
0 0	0 0
0 1	0 1
1 0	0 1
1 1	1 0

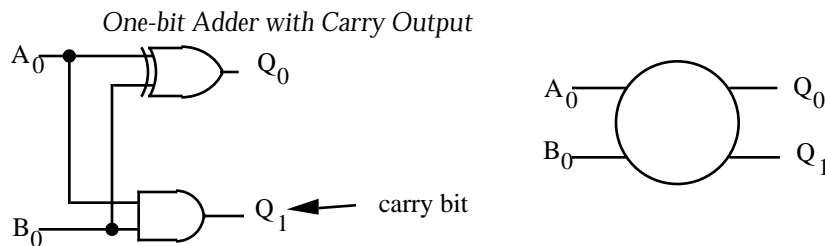
The low order bit of the output can be got by ORing the two inputs, except if both inputs are 1, then the result must be zero. This is the exclusive OR, XOR. The exclusive or is written  $A \oplus B$  and its symbol is



We can make a circuit to do XOR with the building blocks we've used so far.

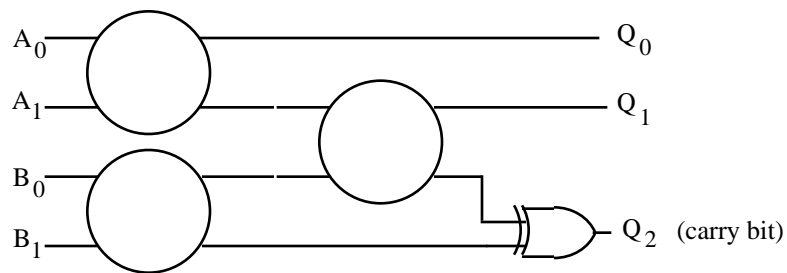


A one-bit adder can be assembled as follows



$Q_1$  is often referred to as the carry bit. To make a two-bit adder, just cascade the two one-bit adders.

### Two-bit Adder without Carry Output



Problem: Design a two-bit adder with carry input and output.

### Circuits for Arbitrary Functions

Any arbitrary function may be built with the simple ANDs, ORs, and NOTs. For example, three bits of input can be mapped to one bit of output. First make a truth table showing the result desired.

input	output	
ABC	Q	
000	1	$(-A)(-B)(-C)$
001	0	
010	0	
011	1	$(-A)BC$
100	1	$A(-B)(-C)$
101	0	
110	0	
111	0	

OR the logic implied by every "1" that appears in the output column:

$$(-A)(-B)(-C) + (-A)BC + A(-B)(-C) = Q$$

Now use the Boolean algebra rules given under the Logic section to simplify this expression.

$$\begin{aligned} ((-A)+A)((-B)(-C)) + (-A)BC &= Q \\ (-B)(-C) + (-A)BC &= Q \\ -(B+C) + (-A)BC &= Q \end{aligned}$$

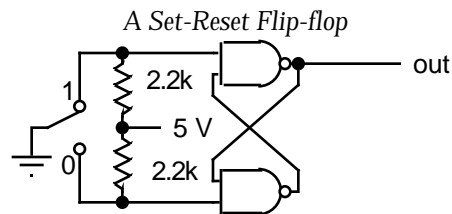
Then translate it to logic gates. The most practical circuit may not be the one with the simplest algebraic expression. It will depend on the gates available. Computer programs exist to do these manipulations.

Another method of reducing logical expressions is the Karnaugh map. Karnaugh maps only work for up to four inputs. You can find an example of Karnaugh maps in Horowitz and Hill (p 333). Complex combinatorial logic devices are most easily achieved using programmable logic devices such as PALs (Programmed Array Logic). These are fundamentally read only memories which can be programmed for any possible output for any possible input.

### Flip-Flops and Memory

The flip-flop circuit illustrated has two outputs but can exist in only two states which are stable. The outputs must be in opposite states in order to be stable so the two states are 0 1 and 1 0. For this reason the outputs are usually labelled Q and  $\neg Q$ . When this switch is thrown to one pole or another, one of the two states is chosen. If the switch is not on either of its poles, the state previously chosen stays. Therefore the flip-flop can be used as a memory register. Eight flip-flops can store an eight-bit number.





If several such registers are connected to an eight line bus, three-state logic gates could be used to connect the inputs of just one of the registers to the bus in order to store the number on the bus. That number could be read later by connecting the outputs to the same bus.

With an adding circuit and a bunch of registers together with some switching circuitry, we have the basic components of a computer.

### References

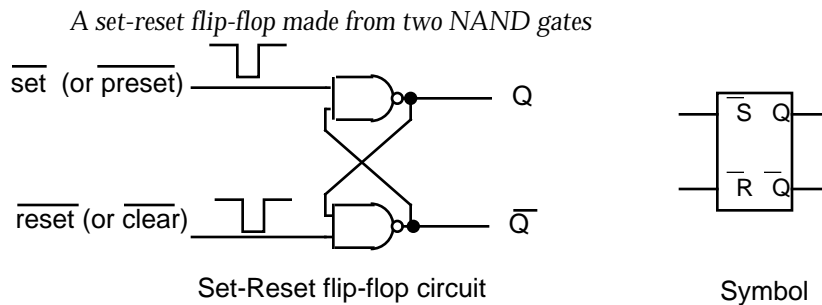
Albert Malvino, *Digital Computer Electronics*, 2nd ed. McGraw Hill, 1977. (or 3rd ed., 1993) Chap 1-6  
Horowitz and Hill, *The Art of Electronics*, 2nd ed, Cambridge, 1989. Chap 8 and 9.  
Joseph Needham, *Science and Civilisation in China*, V2, Cambridge, 1956. Chap 13, esp. sec. g.



## Lecture 2

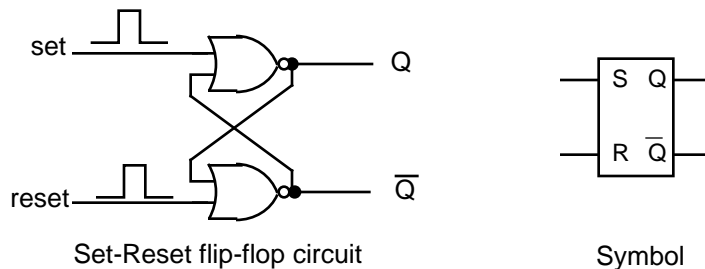
## More on Flip-Flops

The simplest type of flip-flop is the RS flip-flop where RS stands for Set-Reset. It can be made with two NAND gates or two NOR gates as shown below. In the NAND gate version the two inputs are really  $\overline{\text{set}}$  and  $\overline{\text{reset}}$ , the inverse of set and reset. Such a circuit is useful for switch debouncing. Normally a switch may jump open and closed momentarily after being changed. A flip-flop can stop any bounce of the signal from the switch because the output does not change when the switch opens after having been closed on either the high level or the low level.



Truth table for the NAND set-reset flip-flop

$\overline{S}$	$\overline{R}$	Q	$\overline{Q}$
1	1	no	change
0	1	1	0
1	0	0	1
0	0	not	used

*A set-reset flip-flop made from two NOR gates*

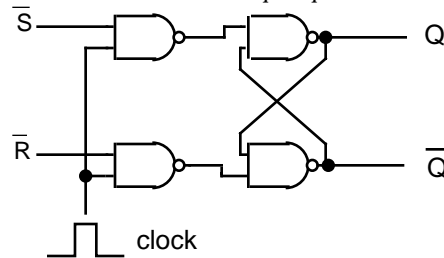
Truth table for the NOR set-reset flip-flop

S	R	Q	$\overline{Q}$
0	0	no	change
1	0	1	0
0	1	0	1
1	1	not	used

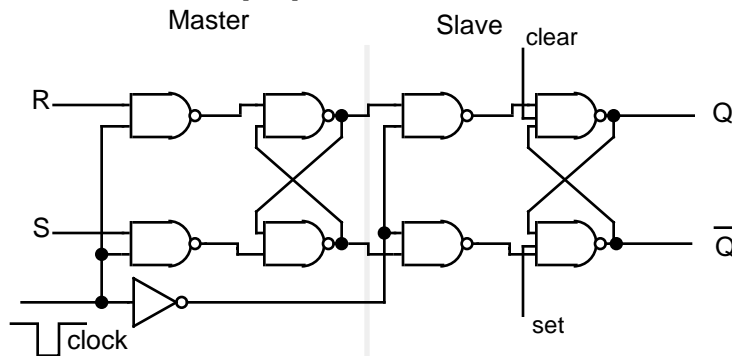
The “not used” state is not useful because the outputs are unpredictable. Going from the “not used” state to the quiescent state involves changing both inputs. Because it is unlikely that both inputs will change at exactly the same time, the flip-flop will momentarily pass through either the set or the reset state. Thus the outputs depend unpredictably on which input changes first.

Besides being used as switch debouncers, the SR flip-flop is useful for storing data and it forms the basis for static ram devices. They have serious limitations for other uses: 1. there is no mechanism for clocking the input, the output changes the moment the input changes, 2. the inputs are level sensitive instead of edge sensitive and 3. there is a disallowed state possible. To overcome these problems two other types of pre-packaged flip-flops are available—the JK flip-flop and the D flip-flop.

A clocked input to the RS flip-flop insures that the output changes only when the clock input level indicates that the signals are valid.

*Clocking added to the basic set-reset flip-flop*

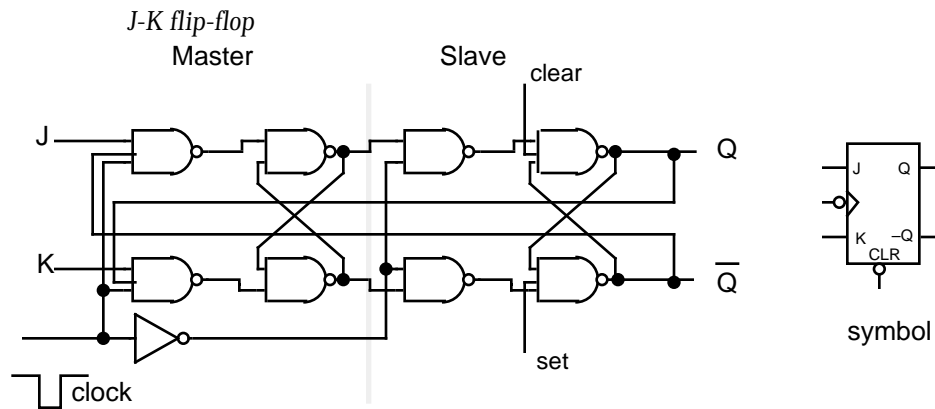
Sometimes you want to transfer data consecutively from one flip-flop to the next by hooking the output of one RS flip-flop to the input of another. With the simple or clocked RS flip-flop the data will race through the chain. The Master Slave flip-flop combines two clocked RS flip-flops to control the transfer of data from one stage to the next in step with the clock pulses. When clock is high the signal is loaded into the master's inputs while the slave is disabled. Then the clock goes low and the output of the master is transferred to the slave and appears at the slave's output. These can be chained to allow an orderly flow of data. Often the output AND gates have a third input to allow initializing the slave to a definite state.

*The Master-Slave flip-flop*

Feeding the Q output of the slave to the S input and the  $\bar{Q}$  output to R creates a T flip-flop. This toggles on every second down tick of the clock and divides the clock frequency in half.

A common type of flip-flop used in circuits is the JK flip-flop. (Does anybody know what JK stands for?) Functionally, it is like the T flip-flop where the feedback lines are NANDed with the J and K inputs. Its truth table shows that it operates like a Master-Slave flip-flop triggered on the down beat of the clock except when J and K are both high when it becomes a T(oggle) flip-flop.

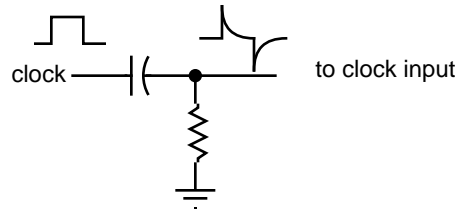
The Master-Slave system is used because the datum on the master is transferred to the slave on the falling edge of the clock. As long as the clock is low, changes on the master's inputs have no effect on the output. When the clock rises the slave is locked. Still no change is possible on the output until the clock falls again at which point the datum on the input is again given to the slave's inputs. But this datum doesn't get to the output until the clock rises again. Data are captured only at the instant of clock transition from high to low but the delay from input to output is the clock pulse length.



Chips in common usage based on the Master-Slave principle are the 7473, 7476 and 74107, 74109 and 74112. The clock signal is level sensitive. Therefore, the input data should not change except immediately after the clocking happens. Then it can only be changed once. If continuously changing data is being sampled, the output may not represent the data on the input at the time of the clocking. Master-slave designs are difficult to use and have been superseded by newer designs such as that of the D flip-flop.

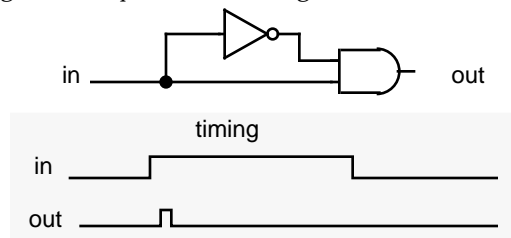
It is desirable to have the input-to-output transition time independent of clock pulse length. A way one might get such edge triggering is by putting a “half-monostable,” or RC high-pass filter, on the clock so that a high clock pulse decays rapidly. Now the transition time depends only on the RC time constant. (R must include the input-impedance of the logic gates.)

*Cheap way to get edge triggering from a level trigger*



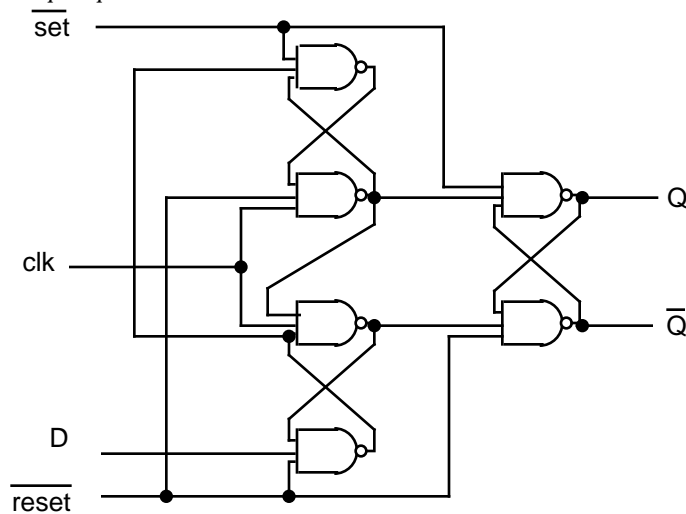
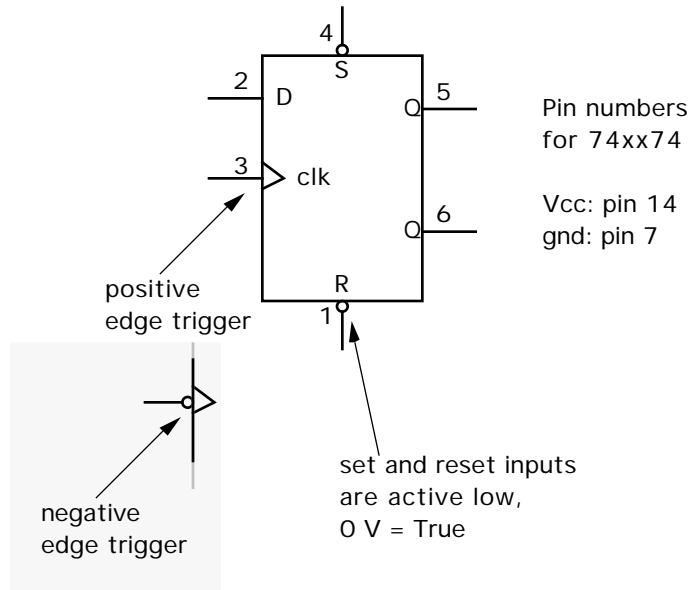
Capacitors and resistors are usually hard to make on an integrated circuit so most edge-triggering schemes use delays to create a short pulse from a clock transition. The figure below illustrates the principle.

*How to get a short pulse from an edge*



When the input goes from low to high both inputs to the AND gate are high for about 10 ns. The direct input on the bottom rises immediately but the top input stays high until the change has passed through the inverter. For this delay period, the output is high. The transition from high to low doesn't cause any pulse on the output.

The D flip-flop is functionally very similar to a J-K flip-flop in which the K input is wired to be the inverse of the J input. Most packaged D flip-flops are truly edge-triggered. Data is captured only during the clock transition. Common TTL D flip-flop packages are the 7474, 74175 and 74174. They are triggered on the positive clock transition. The D-flip-flops use a delay mechanism for their triggering.

*D flip-flop insides**Symbol for D flip-flop**D flip-flop truth table*

S	R	clk	D	Q
0	0	x	x	*
0	1	x	x	1
1	0	x	x	0
1	1	0	x	NC
1	1	x	x	NC
1	1	0	0	0
1	1	1	1	1

\* = not defined

x = doesn't matter, could be either 0 or 1

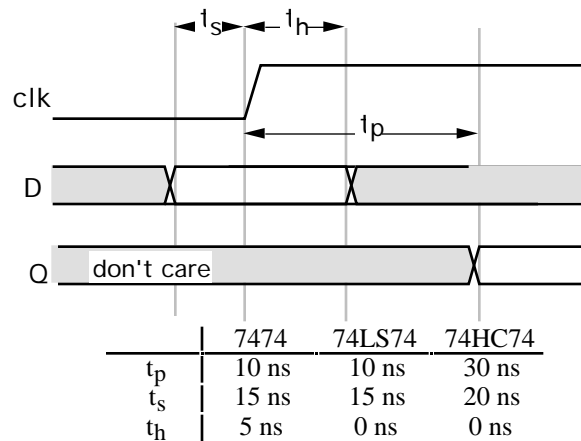
NC = no change

The 74xx74 is the most commonly used D flip-flop chip with two per chip. Important timing parameters for the D flip-flop are as follows:

the setup time,	$t_s$	time the data must be stable before the clock edge
the propagation delay time	$t_p$ ,	time for the input data to transfer to output after clock edge
the hold time.	$t_h$	time data must be stable after the clock edge.

These are illustrated on the following sketch.

*Timing for the D flip-flop*

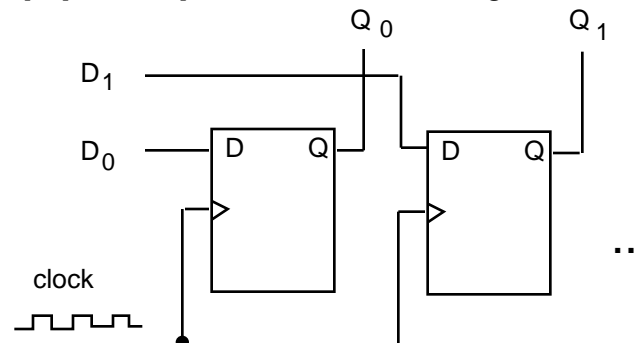


The D flip-flop can be configured to act like a toggle-flip flop or even a simple set-reset flip-flop if necessary.

### Flip-Flop Applications

A common use of flip-flops is for data storage. The simplest parallel-in parallel-out (PIPO) register is made by stringing together several D flip-flops on the same clock signal. As many bits as needed can be made by continuing the sequence.

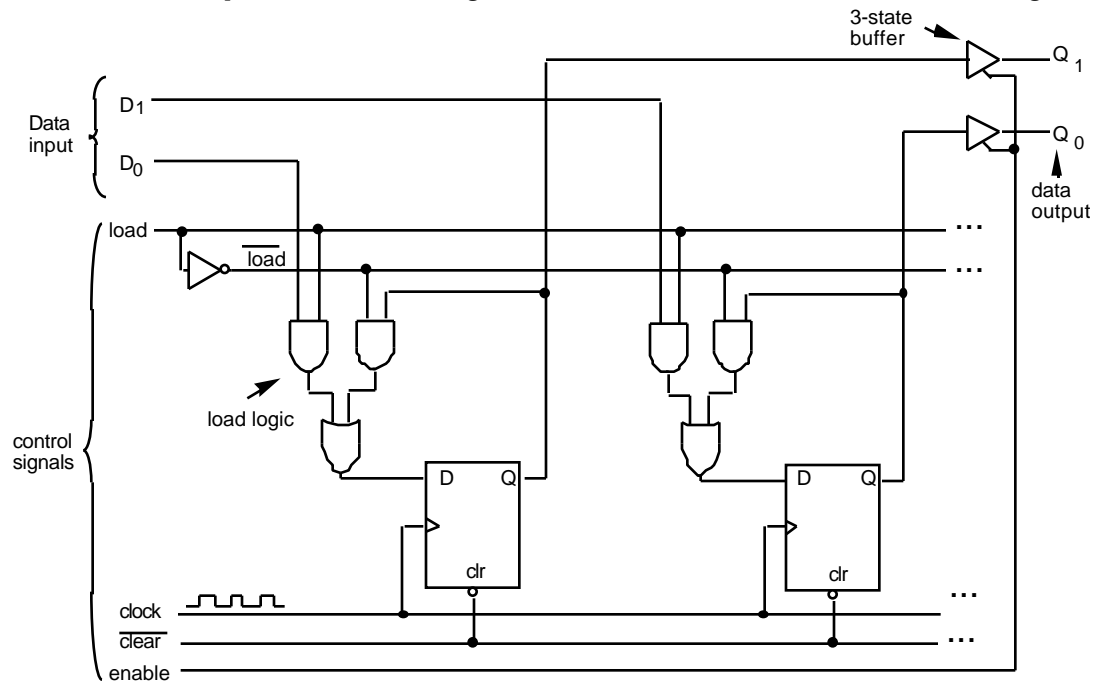
*Simple parallel-in parallel-out (PIPO) buffer register*



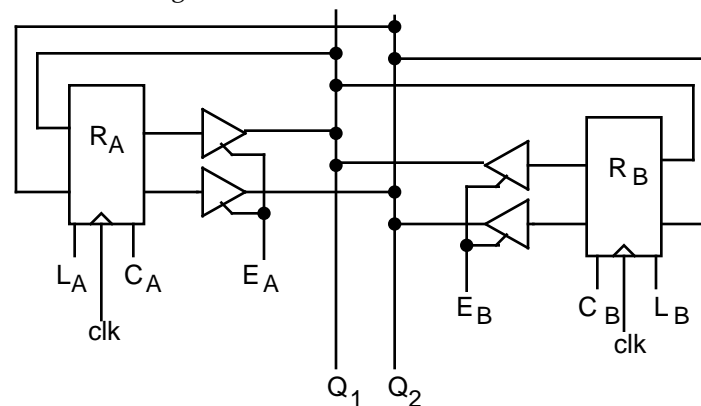
This simple register transfers the input data to the outputs on every clock pulse. This may present a problem when the register is used in a situation where the data do not need to be changed at every clock cycle. Another signal, LOAD, needs to be provided so that the register can be informed as to when the input data are valid and when they are to be ignored. The simplest way to accomplish LOAD would be to AND the LOAD signal with the clock before the clock signal is sent to the flip-flops' inputs. The practical problem with this design is that the actual loading of the data is delayed by the AND operation. In many situations it is important that the actions of all devices such as registers and counters be exactly synchronized with the clock. Another way to accomplish the load function is illustrated below. When LOAD is high the load logic circuitry transfers the input data to the D input of the flip flops. If LOAD is low then the D input of each flip-flop is presented with its own Q output, thus leaving the state unchanged. There is a delay between assertion of the LOAD signal and the time when it is in effect. Thus the LOAD must be asserted about 40 or 50 ns before that clock pulse which is to load the data.

Two more functions are shown on the circuit. A clear signal enables all outputs to be zeroed and ENABLE which allows the outputs to be disconnected from other circuitry. This allows several buffer registers to share the same data bus.

A practical data buffer register with *LOAD*, *CLEAR* and *ENABLE* control signals.



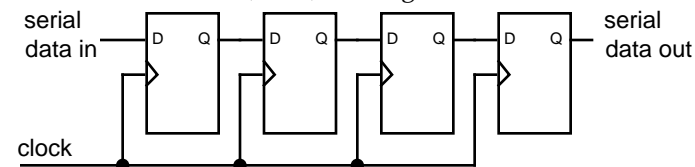
Two two-bit registers connected to a bus with three-state buffers



Many registers can be connected to the same bus. All share the same clock signal and the inputs and outputs of each register are isolated from the bus unless the appropriate signals are invoked. For example, to transfer data from register A to register B signals  $E_A$  and  $L_B$  must be raised. The next clock pulse causes the data to flow.

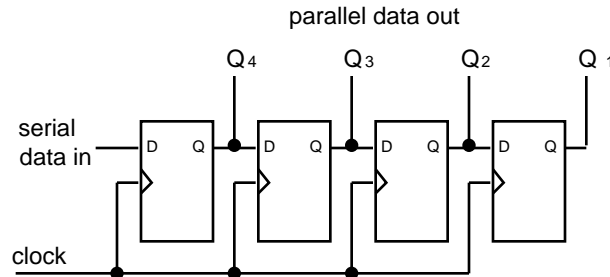
Shift registers are composed of a sequence of flip-flops. Serial-in serial-out shift registers accept data bit-by-bit on the input and then send them out in the same manner. Serial-in Parallel-out shift registers will accept bits clocked in serially and shift them into consecutive parallel output locations.

A serial-in serial-out (SISO) shift register



A serial-in parallel-out (SIPO) shift register

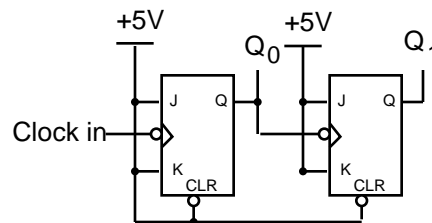




## Counters

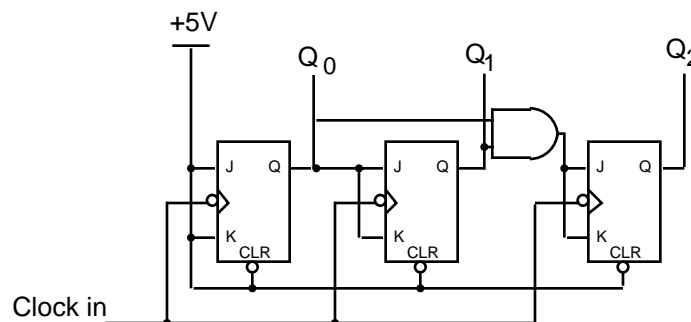
Flip-flops are also the basis for counters. Because J-K flip-flops act as toggle flip-flops when both J and K are high, they are natural for counter applications. The simple two-bit ripple counter cycles one step through the sequence 00–01–10–11–... on every clock pulse. The clock is fed only into the lowest order bit. The flip-flops for the higher order bits are clocked from the output of the next lower flip-flop. Thus there is a delay which gets larger as the flip-flop gets farther from the low order bit. Ripple counters are an easy way to divide down the frequency of clocks, but synchronous counters are better for real counter application.

### *Ripple counter*



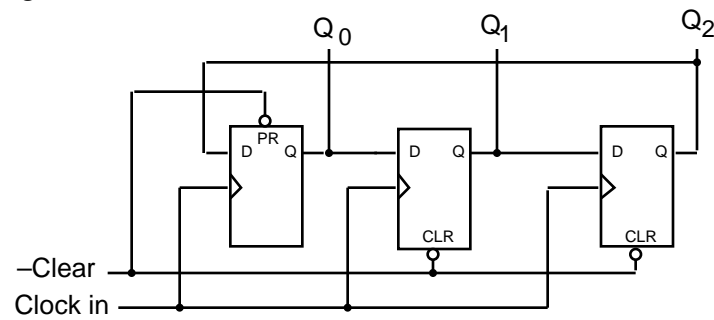
The synchronous counter shown below allows all bits to respond to the clock at the same time. Most prepackaged, multi-bit synchronous counters have a clear and some have a parallel load feature. You must be careful when using clear and load because sometimes these functions occur immediately when activated (asynchronously) and sometimes they occur only on the clock edge (synchronously). The 74LS161 used in the lab has an asynchronous clear and a synchronous load.

### *Synchronous counter*



Another type of counter which is used in computer design is the ring counter. In its basic version, only one of the output bits,  $Q_1 \dots Q_n$ , of the ring counter is active at one time. Each successive clock pulse moves the active bit to the next successive bit. Thus if  $Q_i$  is active,  $Q_{i+1}$  is active after the next clock pulse and all others are off. When the active bit reaches the end, it reverts to the first bit. Variations on the ring counter may move the active bit lower instead of higher at each clock pulse or leave the previous bit on until all are on and then start over with only the low bit on—the “walking ring counter.”

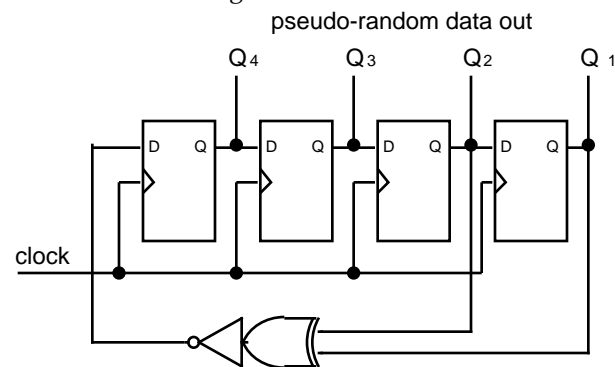
## Ring Counter



## Other Flip-Flop Applications

See the TTL Cookbook for ideas. For fun you might want a pseudo-random number generator. Although the binary sequence repeats after  $2^n - 1$  binary digits, where  $n$  is the number of registers, any short portion of the sequence appears random.

### Pseudo-random number generator



## References

Don Lancaster, *The TTL Cookbook*, Sams, 1974

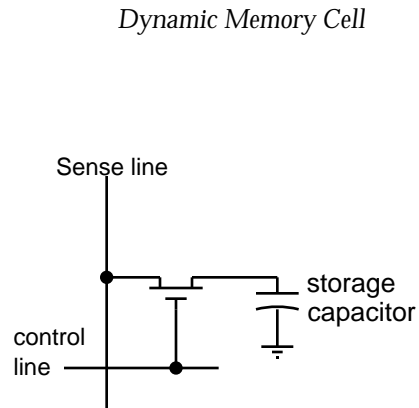
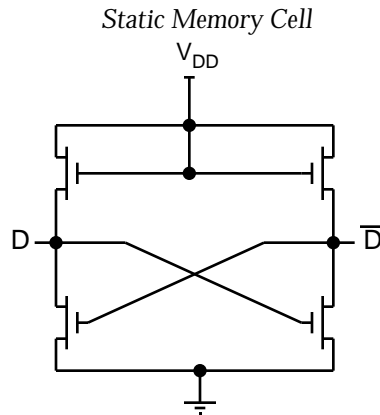
Don Lancaster, *The CMOS Cookbook*, Sams, 1977

## Lecture 3

**Memory**

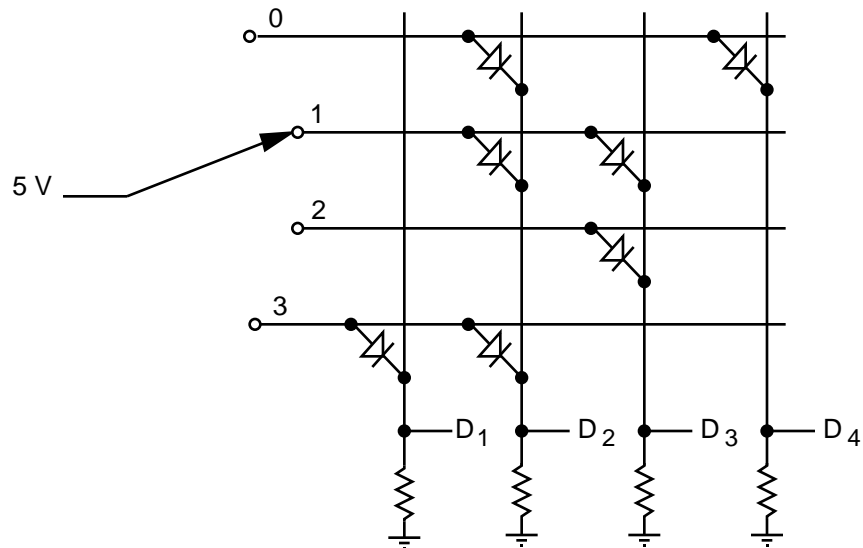
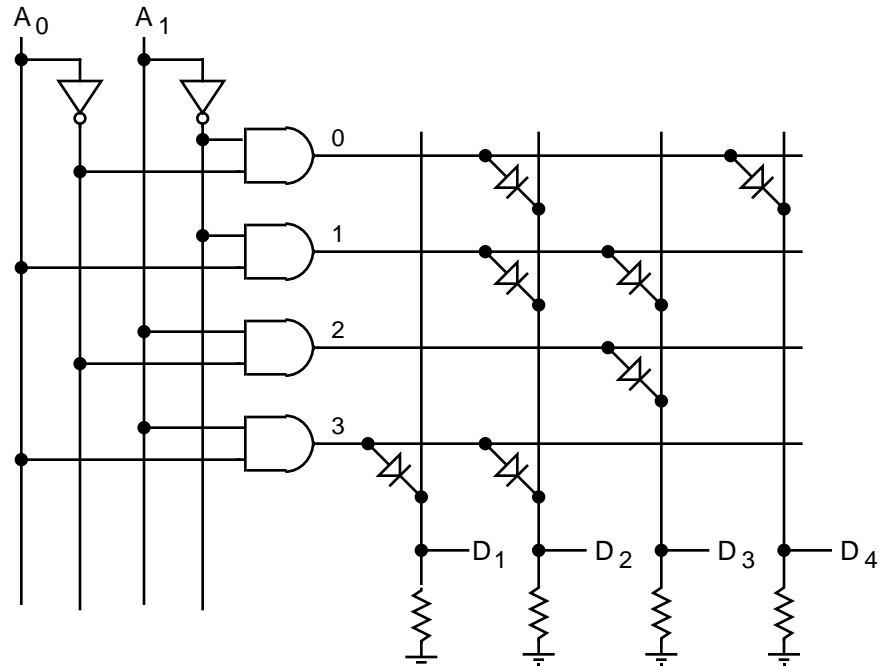
Buffer registers made from flip-flops offer an obvious way to design computer memory. However, for large amounts of memory a simpler method is needed to save overhead. Memory which can be easily changed and read is usually called Random Access Memory (RAM) to distinguish it from memory which cannot be easily changed which is called Read Only Memory. Random access means that any memory location can be read without having first read previous locations. In fact, read only memory is also random access but it has become conventional to call Read-Write memory RAM and Read-only memory ROM.

Read-Write memory (RAM) is available in two forms: Static and Dynamic. A static memory cell is a simple two-transistor flip-flop. In addition to the two transistors of the flip-flop, two more transistors are used as load resistances. (Transistors are easier to make on integrated circuits than resistors.) As long as the power is on, the flip-flop maintains its memory.



Dynamic memory is essentially a capacitor which can be charged and discharged through a transistor switch. When a high memory bit is read, the capacitor discharges and must be restored. Furthermore, the capacitor discharges spontaneously over time. Most dynamic memory must be refreshed at least every 2 milliseconds. In most computers the memory refresh cycle is controlled by the Direct Memory Access controller which takes over the computer bus every  $15.12\mu\text{s}$  in order to refresh  $1/128$ th of the memory simultaneously.

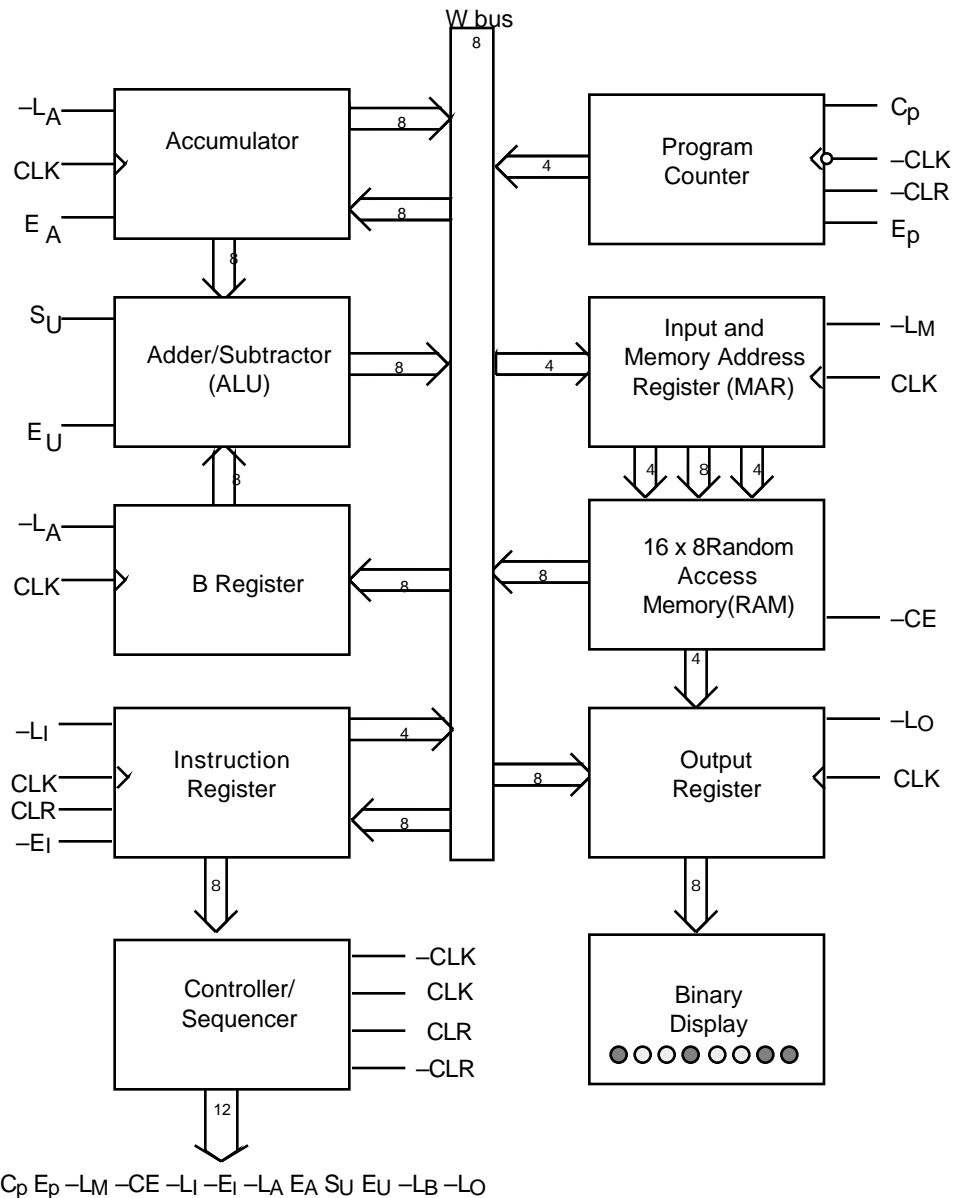
Read-only memory is useful for routines such as boot-up sequences, basic input-output systems, graphics and numerical routines. The memory holding these machine instructions need never change. The basic design of read-only memory utilizes an array of diodes. Each row of the array is one memory location. The columns are bits of the output. When a diode connects a column and row that bit of the memory location is high. The absence of a diode results in a low value.

*Read-only memory using a diode array**Read-only memory with on-chip decoding*

### The Simple as Possible Computer: SAP-1

We now have the components needed to construct a real working computer: the SAP-1. This “Simple as Possible Computer” was invented by Albert Malvino to illustrate the basics of computer design with a model system capable of being built using available TTL or CMOS logic chips. The components used are a counter, several registers made of flip-flops, a 8x16 static random access memory, an arithmetic logic unit which does additions and a controller/sequencer which translates binary machine instructions into the control signals for the components. These components interconnect via an 8-bit bus called the W bus.

### Architecture of the Simple-as-possible Computer



The program counter (PC) keeps track of which machine instruction is to be executed next. It starts at zero when the machine is turned on and increments during each instruction. The program in the form of 8-bit binary numbers is stored in the RAM starting at location 0000. The number in the PC indicates the memory location containing the next instruction.

The instruction set is also as simple as possible consisting of the following:

mnemonic	op code	explanation
LDA	0000 nnnn	load accumulator with number in memory location nnnn
ADD	0001 nnnn	add number in memory location nnnn to accumulator
SUB	0010 nnnn	subtract number in memory location nnnn from accumulator
OUT	1110 xxxx	output accumulator to the lights (xxxx = anything)
HLT	1111 xxxx	stop everything

All instructions are coded with four bits at the high end of the byte. LDA, ADD and SUB have an additional four bits in the low half holding the address in RAM where the operand resides. OUT and HLT don't care about their low four bits.

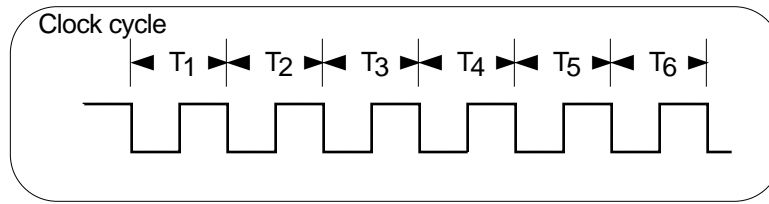
The Memory Address Register (MAR) holds the the address of the memory location

currently being used. This location contains either an instruction or data. Addresses of instructions originate from the PC in the SAP-1 and addresses of data originate from the low four bits of an instruction.

The instruction register has an eight-bit input for loading an entire instruction. The four-bit output to the W bus sends the MAR the data address contained in some instructions. The other four-bit output goes to the controller/sequencer which translates the instruction operation code into control signals for the devices on the bus.

Accumulator A can load data from the bus when  $\neg LA$  is active. It outputs its contents to the bus when EA is active. Register B can only load data. This data is output to the adder / subtractor. The adder / subtractor also receives data from A. The result of the addition or subtraction is output to the bus when EU is high. The signal SU causes the unit to subtract when high; otherwise it adds.

Execution of each instruction takes place in up to six steps. Each step starts with a downward clock transition. Midway through each step the clock rises. The next falling edge marks the beginning of the next step. A six-bit ring counter keeps track of which step is being executed. The ring counter signals, labelled T1 through T6, together with the instruction op code determine the control word which actuates the devices on the bus.



A control word with all signals inactive can be considered as a “no op” which does nothing: NOP control word = 0011 1110 0011. A complete instruction cycle is illustrated by the ADD instruction. The first three steps are common to all instructions and is called the Fetch sequence.

Control words for the Fetch sequence:

Load instruction: 0101 1110 0011

Increment PC 1011 1110 0011

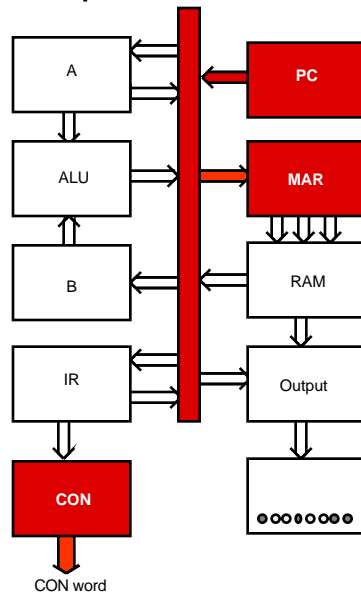
Fetch instruction 0010 0110 0011

EP  $\neg LM$  active

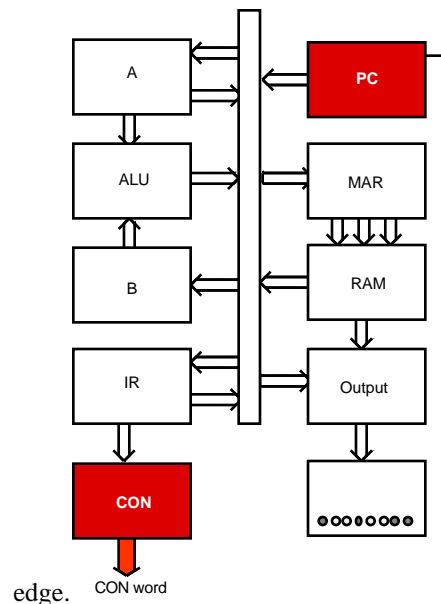
CP active

$\neg CE$  and  $\neg LM$  active

### Fetch Sequence:

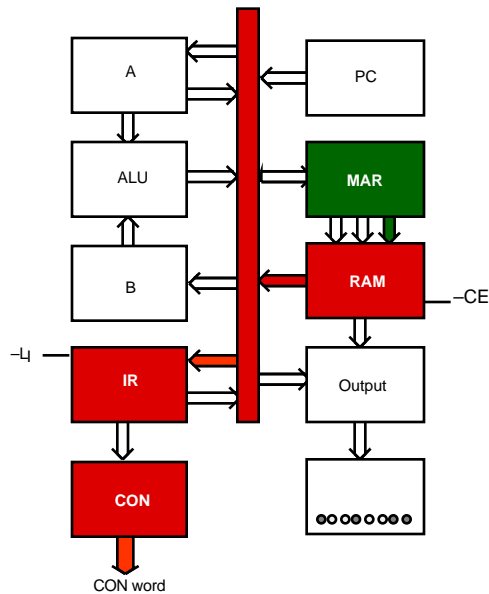


**T1:** The contents of the program counter are loaded into the MAR on the positive clock



edge. CON word

**T2:** Positive clock edge advances program counter by one.



**T3:** The MAR sends an address to the RAM on the negative clock edge then the contents of the addressed RAM location are loaded into the instruction register via the bus on the positive clk edge.

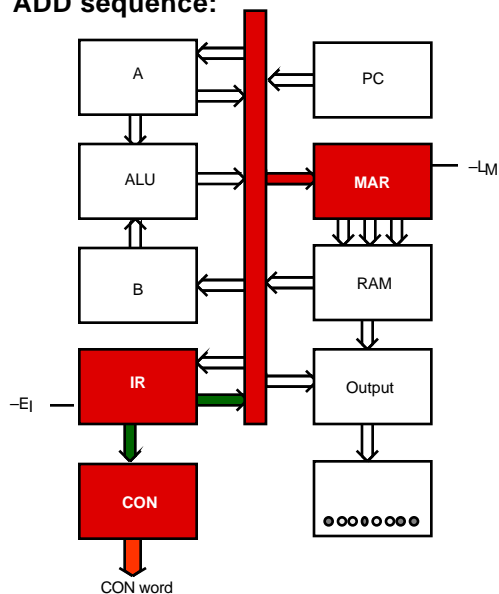
The next three steps are unique to the ADD instruction.

Control words for the Add sequence:

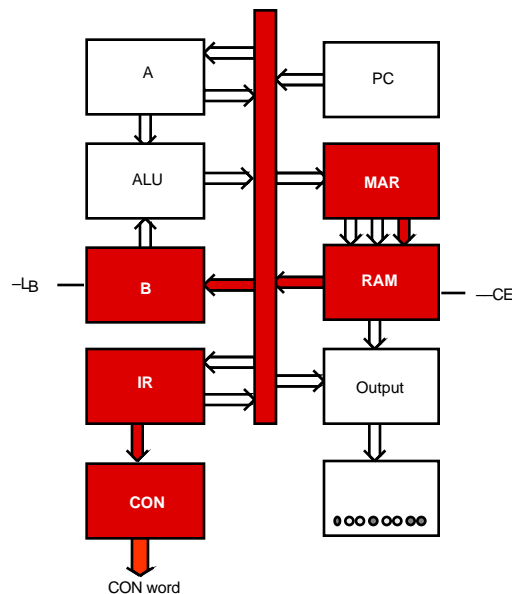
Get data address	0001 1010 0011
Load B register	0010 1110 0001
Load sum into A	0011 1100 0111

–EI and –LM active  
–CE and –LB active  
–LA and EU active

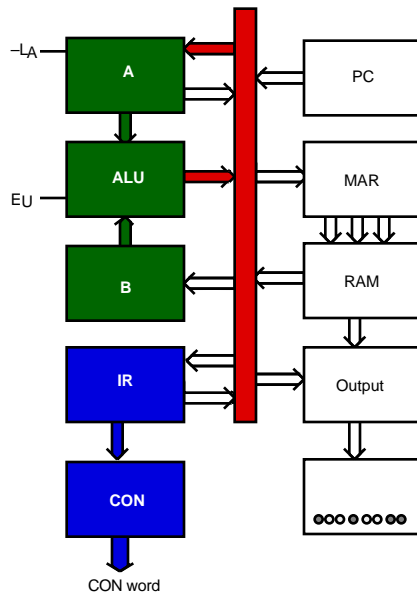
**ADD sequence:**



**T4:** The address field of the Add instruction is loaded into the MAR via the bus



**T5:** The contents of the addressed RAM location are loaded into the B register on the positive clock edge.



**T6:** On the negative clock edge the ALU sums A and B registers. The accumulator is loaded with the sum on the positive clock edge.

Other instructions have similar control word sequences. These control words can be considered to be microinstructions for a six-instruction microprogram of the ADD process.

**Reference:** Albert Paul Malvino, *Digital Computer Electronics* (2nd ed. or 3rd ed. ), McGraw Hill (1983 or 1993)

### The Intel 8088 and the IBM PC

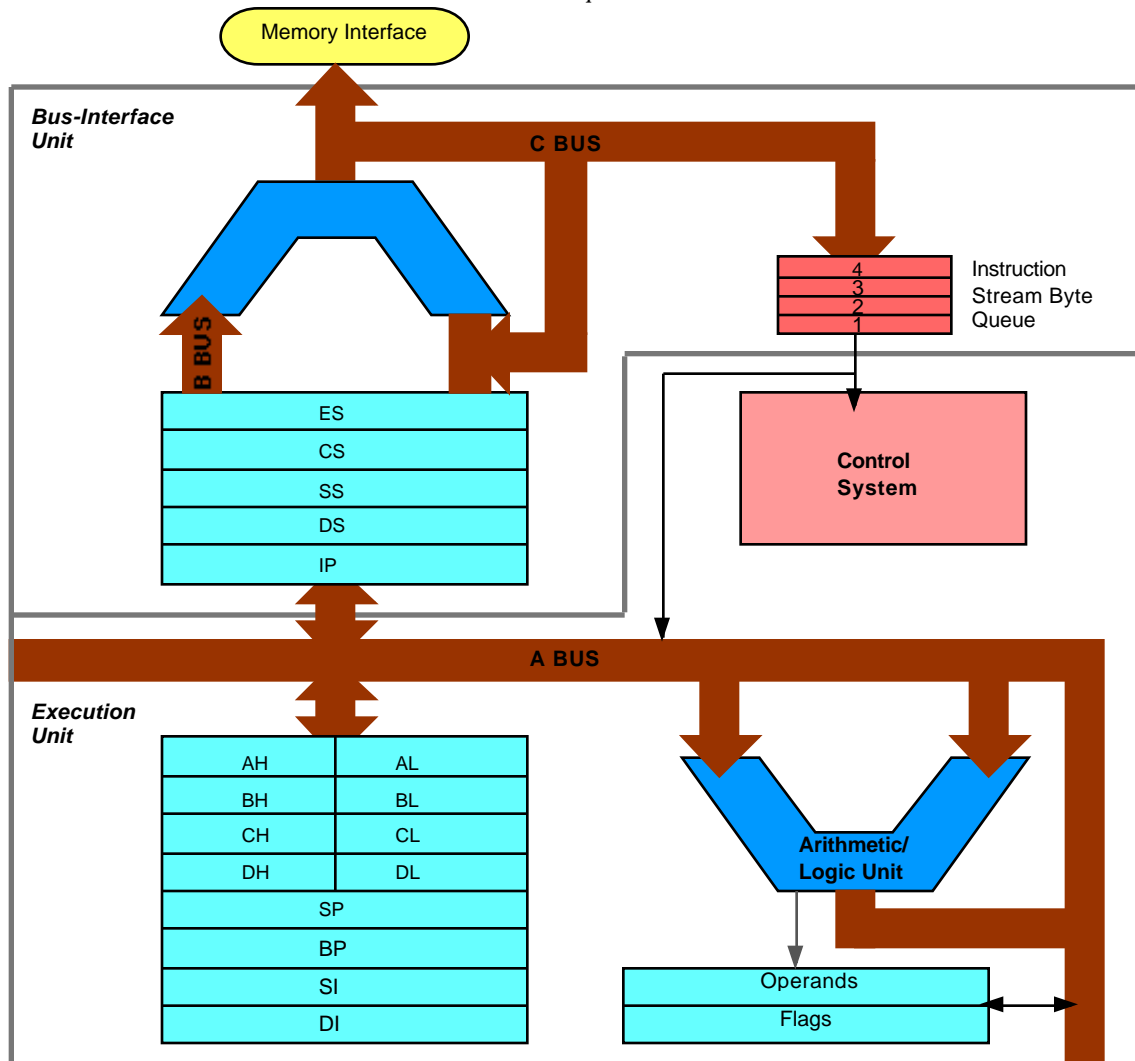
The first Intel microprocessor with 16-bit internal registers was the 8088 though in order to minimize external connections, the 8088 still used an 8-bit data bus. The 8086 has the same internal architecture as the 8088 but has a 16-bit data bus. Understanding the design of the 8088 serves as a basis for understanding more advanced Intel microprocessors from the 8086 through the Pentium.

The architecture of the Intel 8088 microprocessor used in the original IBM pc computer is shown below. There are three internal busses for the flow of data and instructions. AH,HL, BH,BL, CH,CL and DH,DL are eight-bit data register pairs. SP, BP, SI, DI are the stack pointer, base pointer, source index and destination index registers. There is a four-byte queue for incoming instructions which speeds up the processing by preloading the next sequential instructions to be executed.

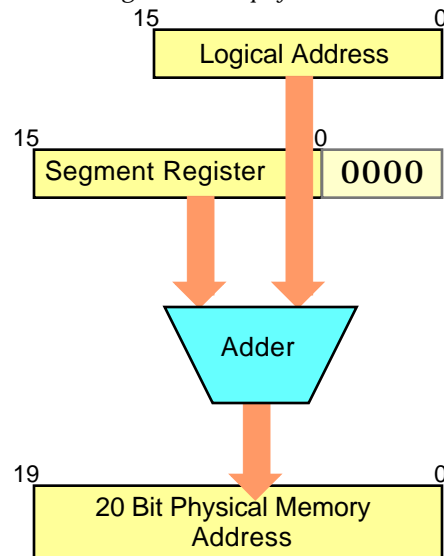
The specific purposes of the pointer, index and segment registers can be found in any good book such as Eggebrecht or Sargent and Shoemaker. The segment registers need some comment. Normally the 16 bit address field of the 8088 instruction set would allow addressing only 64K (65536 bytes) of data. The 20-bit address line, however, allows addressing up to one megabyte. The 64K limitation is circumvented by using the segment register to specify a specific region in the one Mbyte. The contents of the segment register are shifted left four bits and added to the 16-bit instruction address to form the 20-bit physical address. Thus an address of 1 would refer to the physical address 1 if the segment register were 0. If the segment register were 1 it would refer to 17. If the segment register were 2, then the physical address would be 33. And so on ad nauseum. Physical addresses on the pc are written as two numbers separated by a colon (ssss:aaaa) where ssss is the segment register and aaaa is the address.

It's easy to write programs that use 64K of contiguous memory. But if you want to go outside this region you have to manipulate the segment register. That's why many pc programs don't let you have arrays or structures that take up more than 64 K. The 80286 processor in the pc AT also uses this scheme of things. Real computers have 32-bit words. Starting with Intel's 80386 microprocessor, 32 bit address and data registers became standard. Segment registers remain for backwards compatibility.



*Internal structure of the 8088 microprocessor*

*How segment registers are used to get a 20-bit physical address from a 16-bit logical address*



**References:**

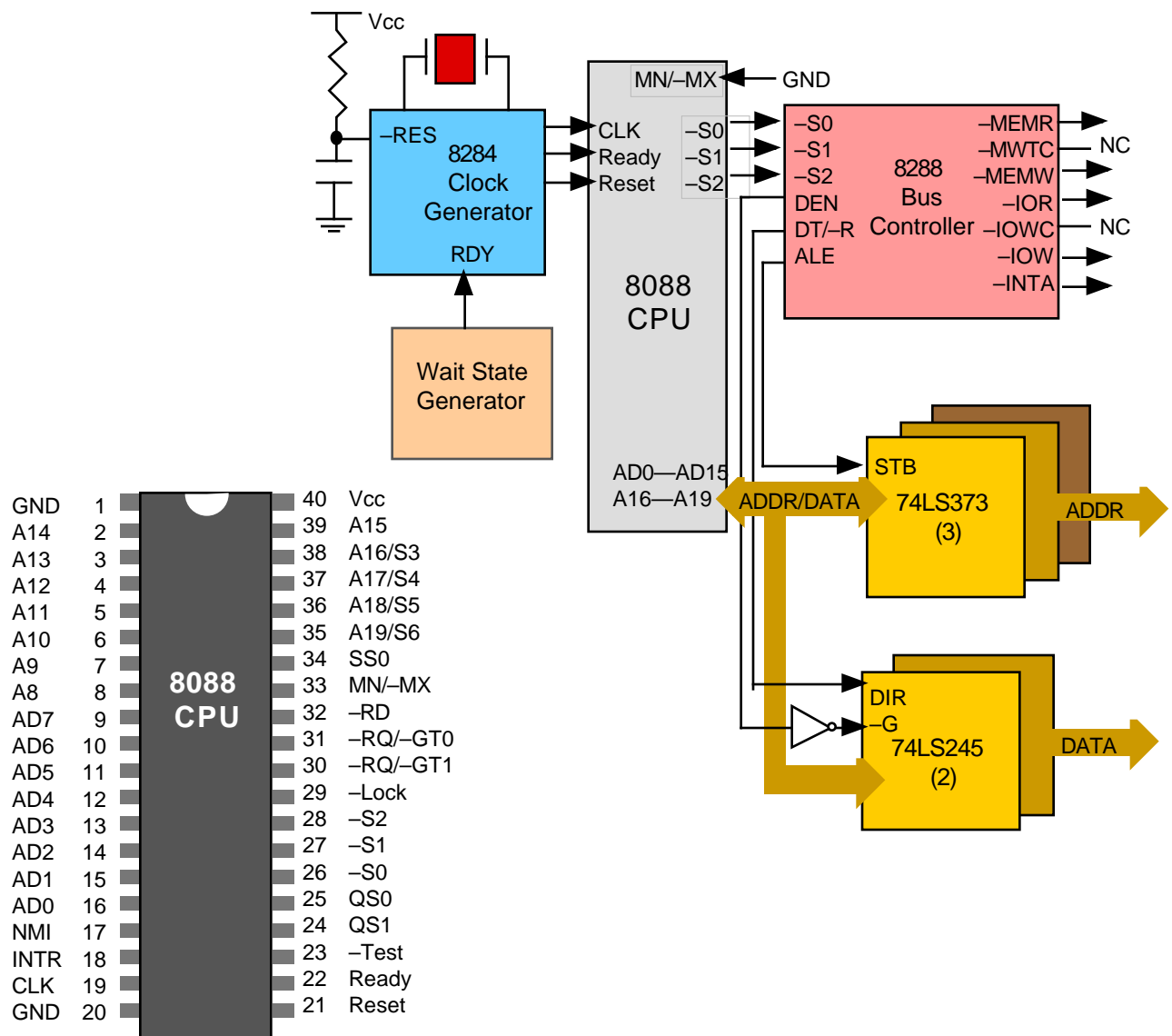
- Lewis C. Eggebrecht, *Interfacing to the IBM Personal Computer*, 2nd edition, 1990 (1st ed also ok.)  
Murray Sargent III, Richard L. Shoemaker, *The IBM PC from the Inside Out.*, 1986  
Paul Horowitz and Winfield Hill, *The Art of Electronics*, 2nd ed, Chap. 10, 1989.

## Lecture 4

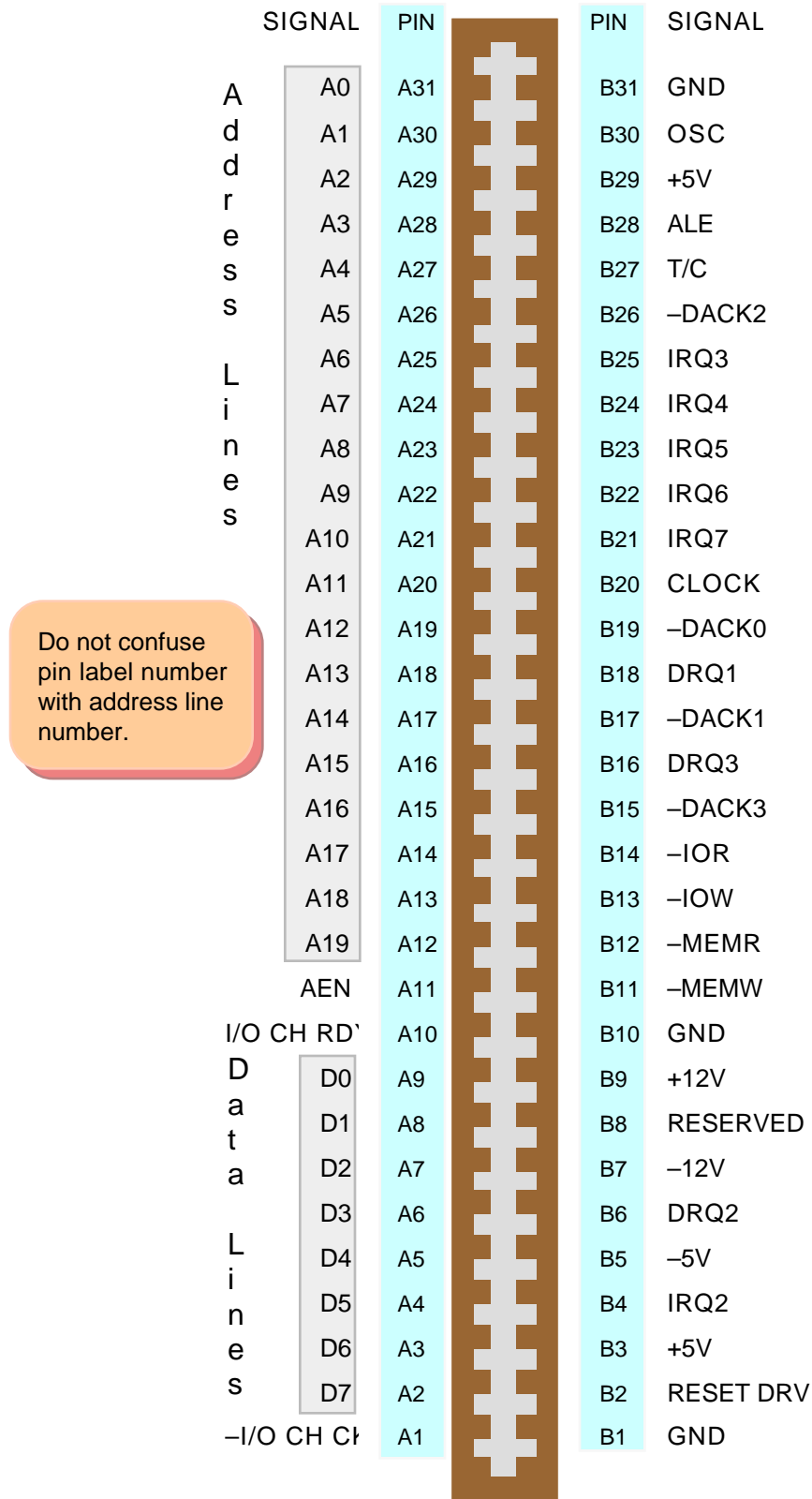
**Communication of the Microprocessor with the Outside World**

The microprocessor uses an external bus to interface to memory and peripheral devices. The bus has three parts: Address lines, Data lines and Control lines. For the most part, the lines on the bus correspond to pins on the 8088 chip. However, the bus control lines are decoded from the chip status pins, S0, S1 and S2.

*The 40 Pins of the 8088 Microprocessor and how they Interface to the Bus*



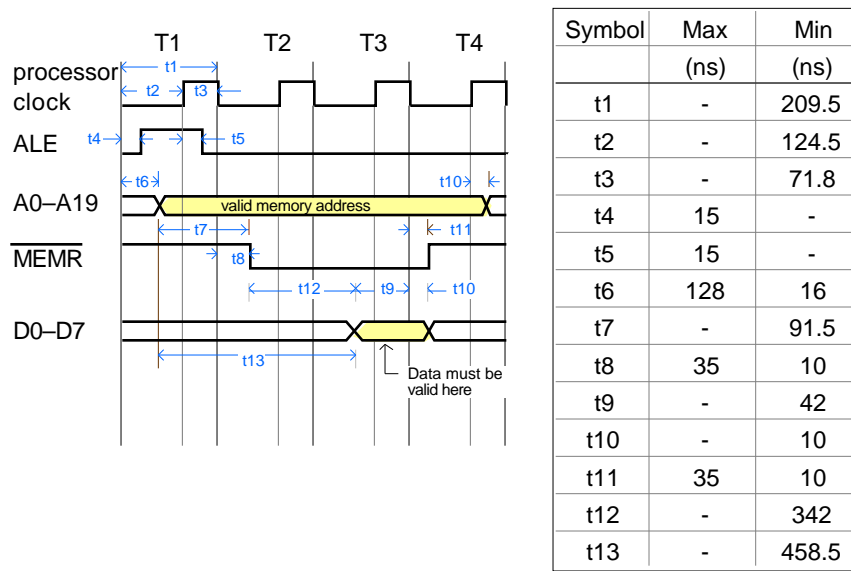
*The Signals on the 8-bit portion of the ISA bus (IBM PC-XT)*



The basic method of interaction is illustrated by the cycle to read a byte of data from external memory. At the first step the microprocessor puts the address of the desired byte on the address lines. This happens during the first clock tick, T1. It then expects the external circuitry to

take care of finding the data and putting it on the data lines. The 8088 shares some address lines with data lines, so the address must be latched externally. The Address Latch Enable (ALE) goes low to signal when the address should be captured by the external latch. At this time the Data Transmit/Receive goes low to show that the bidirectional data lines are to be used to receive data. At the beginning of tick T2  $\overline{\text{MEMR}}$  drops low to request the addressed data.  $\overline{\text{DEN}}$  rises to connect the data bus buffer to the 8088 data lines. Before T3 is over, or 342 ns after  $\overline{\text{MEMR}}$  drops, data should be available. Memory used in these pc's usually take 200ns to respond to a read request. That leaves plenty of time for the other circuitry to work. The cycle is finished at the end of T4.

Memory Read Bus Cycle Timings for 4.77 MHz Bus Clock



Timings vary for systems that operate at higher clock speeds. The ISA bus seldom runs faster than 10 MHz, even if the microprocessor is much faster. A good rule is to scale the timings to the clock speed of the ISA bus system. However, due to custom integrated circuits used in some systems, it is nearly impossible to give absolutely reliable timings for all systems. For example, in a 10 MHz system  $t_1=100$  ns,  $t_2=67$  ns and  $t_3=33$  ns. Timings which are measured from a clock edge are generally valid independent of clock speed, e.g.,  $t_5=15$  ns.

The memory write operation works in similar manner except it's  $\overline{\text{MEMW}}$  that signals data is available on the bus. (See Eggebrecht for complete timing charts of memory and I/O read and write cycles.)

One way to send or receive data from a peripheral device is to wire it up to respond to memory read and write instructions. The memory map of the pc shows that locations 640K to 784K are used by the video display. To display a character, simply write to a memory location in this range. This is called memory mapped I/O. The 8088 also provides specific instructions for I/O to peripherals, the IN and OUT instructions. These use address lines A0-A15 allowing 65536 additional ports. Cards on the ISA bus usually decode only A0-A9, giving 2048 addresses. By convention, addresses with A9 low are on the motherboard and A9 high is for devices off the mother board. If a device were designed to respond to address 911 hex, it would likely activate a device at 111 as well, if it existed, causing a real emergency. IBM has reserved many I/O address for specific devices. I/O addresses 300h to 31Fh are reserved for prototype boards. Nevertheless, many ethernet cards use 300h by default, though this can usually be changed either by software, or by fiddling with dip switches.

The control lines  $\overline{\text{IOR}}$  and  $\overline{\text{IOW}}$  signal data flow for input/output operations like the  $\overline{\text{MEMR}}$  and  $\overline{\text{MEMW}}$  lines for data read and write. The instruction set for these I/O is not as varied as for data read and write, but you don't have to reserve empty memory addresses to use them. .

If the device cannot respond fast enough to complete the data transfer within four clock ticks, it can pull the I/O Channel Ready line low. This is an open collector line which suspends operation of the cpu. When the device has data ready, it allows the line to rise again and the processor continues. The READY line should not be held low for more than 2 ms, or the memory won't be refreshed in time and all will be lost.

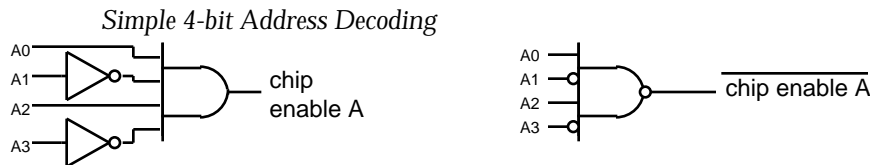
### Decoding the Address

The first circuitry that needs to be made on a peripheral interface is the address decoding circuit. The software will access the device by calling its "port address." For example the BASIC statement `X=INP(&H300)` causes the address lines on the bus to have the following levels:

adr:	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
bin:	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0
hex:	0				0				3				0				0			

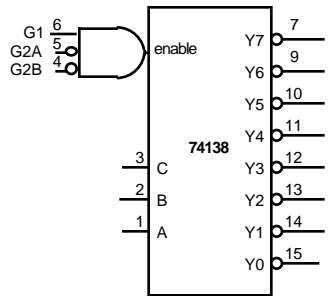
Besides the address lines there are two other control lines which must be monitored. The AEN being low signifies that the address is coming from the microprocessor. Sometimes the Direct Memory Access controller can take over the use of the bus temporarily to transfer data directly from a device to memory. This situation is signalled by AEN going high which means that peripheral devices expecting output from the microprocessor should not heed the address lines. Besides AEN the `–IOW` and `–IOR` signals will go low if an `OUT` or `IN` instruction respectively has been used to address the lines. If memory write or read instructions are addressing the bus, the `–MEMW` or `–MEMR` will drop. Only address lines `A0–A15` can be activated by an `IN` or `OUT` instruction, but a memory read or write can use all 20 of them.

Decoding is just doing a big AND on all the relevant signal lines. Those lines where a 1 is expected go directly to the input of the AND gate and those lines where 0 is desired must first pass through an inverter. The 7430 TTL chip is an eight input NAND gate that is commonly used for this purpose. The 7420 chip is a four input NAND which is also useful. The figure below shows this simple-minded decoding arrangement for a four-bit address to decode address A hexadecimal. The right-hand diagram illustrates a conventional realization.



In the left-hand diagram the signal "Chip Enable A" is high only when A (1010) appears on the address lines `A3..A0`. In the right-hand diagram `–(Chip Enable A)` is low only when A is on the address lines. The output of the NAND can be wired directly to an active-low enable input of another device, such as a latch or three-state buffer. To decode a 16-bit address, the outputs of two 7430s can be ORed.

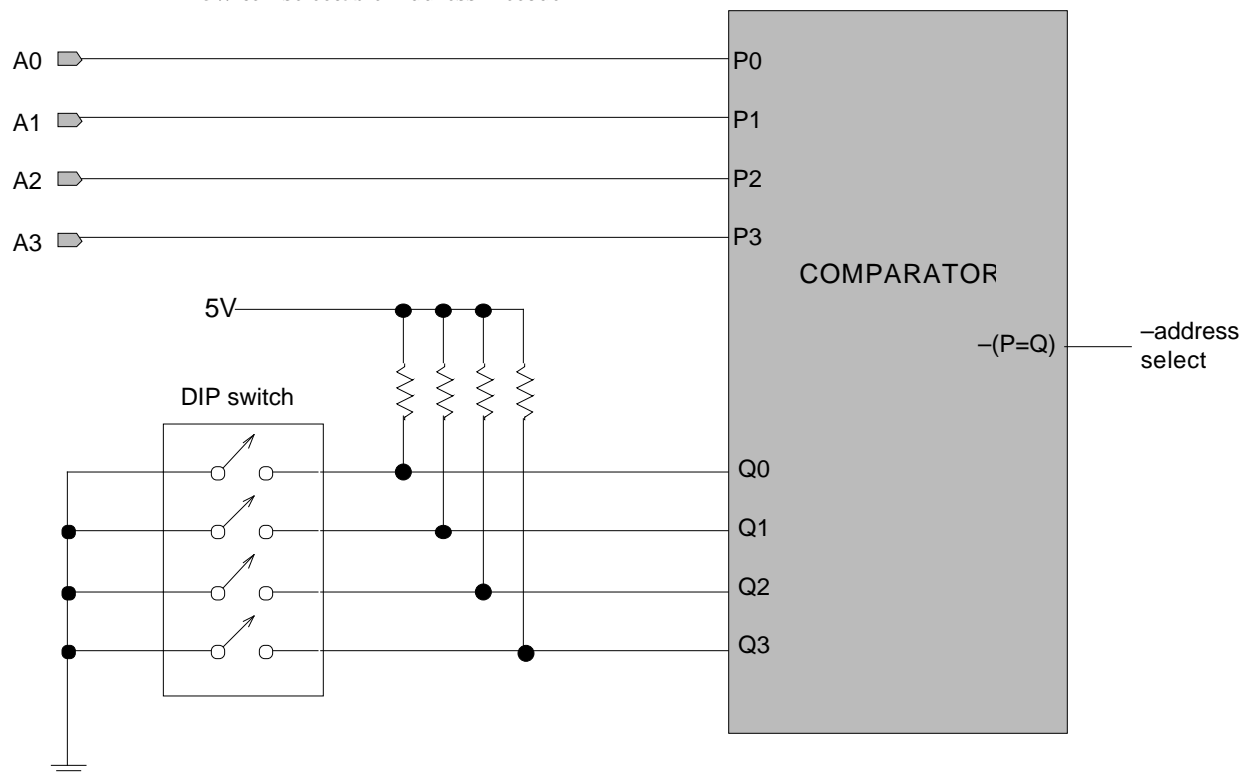
Other methods of decoding are more versatile. For example the 74138 takes three address inputs and activates one of the eight outputs by pulling it low and leaving the others high. There are also three inputs to enable the chip: two of them must be low and the other high for address decoding to work. Six lines can be decoded with one 74138 chip to select eight different addresses. Similarly the 74139 decodes two address bits to four outputs. The outputs of the decoder are sometimes referred to as Chip Select lines (CS). The 74138 chip can be combined with NAND gates, other 74138s or 74139s by cascading them through one of the enable inputs.

*Pin Diagram and Truth Table of the 74 138 integrated circuit*


G1	G2A	G2B	C	B	A	Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0
0	x	x	x	x	x	1	1	1	1	1	1	1	1
1	1	1	x	x	x	1	1	1	1	1	1	1	1
1	1	0	x	x	x	1	1	1	1	1	1	1	1
1	0	1	x	x	x	1	1	1	1	1	1	1	1
1	0	0	0	0	0	1	1	1	1	1	1	1	0
1	0	0	0	0	1	1	1	1	1	1	1	0	1
1	0	0	0	1	0	1	1	1	1	1	0	1	1
1	0	0	0	1	1	1	1	1	1	0	1	1	1
1	0	0	1	0	0	1	1	1	0	1	1	1	1
1	0	0	1	0	1	1	1	0	1	1	1	1	1
1	0	0	1	1	0	1	0	1	1	1	1	1	1
1	0	0	1	1	1	0	1	1	1	1	1	1	1

x = doesn't matter

Some devices may need to have addresses which can be changed by a bank of little switches on the board. The 74688 comparator chip is useful for this purpose. As configured in the circuit below, the output is low only when all bits of both four-bit input ports are equal. The address select signal from the comparator could be routed to an enable of a 74'183 which decodes the other address lines and would allow switching among 15 consecutive I/O addresses. An eight-bit comparator would allow switching among 128 I/O addresses.

*Switch-selectable Address Decode*

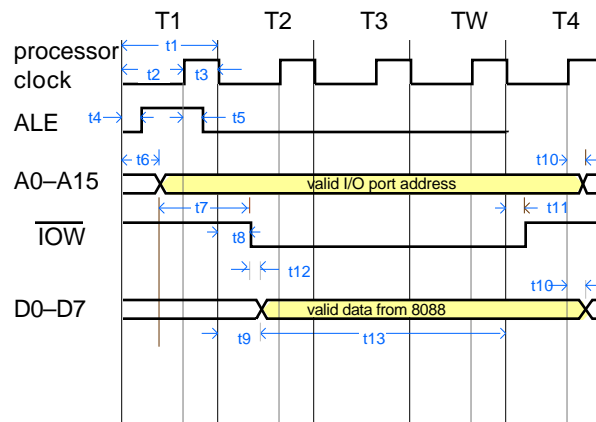
In rare cases it might be necessary to have several devices with widely varying addresses decoded on one board. The best solution for this could be PROM (programmable read-only memory) which can be programmed to output anything for all possible inputs. For example all memory bits could be 1 except those locations addressed when the desired port address appears on the PROM's address lines.

### Latching and Buffering the Data

Data transfer from the CPU to the peripheral device must be buffered by a latch. This is just eight flip-flops which are clocked by the result of the address decoder circuit. The latch is necessary because the desired data is on the data bus only when the address, AEN and  $\text{IOW}$  signals are active. At other times the data bus may contain anything or nothing. The latch stores the data until the next write to its port. The 74273 latch chip is commonly used.

The timing for the I/O port write operation is shown below. For I/O operations, the processor inserts an extra clock tick, TW, which is not in the memory write cycle. This wait state can be extended if the peripheral device needs more time to carry out its task. The extra wait states are requested by pulling  $\text{IOChanRdy}$  line to ground during T2. It is an open collector line so any device can request a wait state at any time. When the  $\text{IOChanRdy}$  line is low on the rising edge in T2, the CPU inserts wait states until the line is high on the rising of the clock. It will then proceed normally after the end of the current tick.

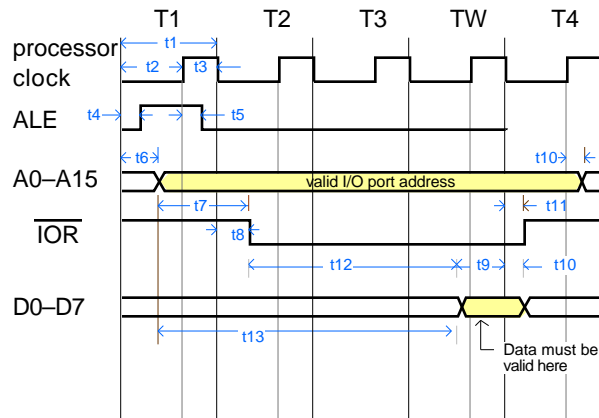
*Input/Output Write Cycle for the ISA BUS (Timings for 4.77MHz Bus Clock)*



Symbol	Max (ns)	Min (ns)
t1	-	209.5
t2	-	124.5
t3	-	71.8
t4	15	-
t5	15	-
t6	128	16
t7	-	91.5
t8	35	10
t9	122	14
t10	-	10
t11	35	10
t12	112	-
t13	-	506.5

The I/O read cycle presents the inverse problem. The data must be present on the bus only when it is requested. If data are there at other times there will be conflict with other devices using the bus. Gating the data to the bus at the precise time is accomplished with the 74241 tri-state buffer. This has two groups of four-bit buffers. One group is enabled by an active-low signal (pin 1), the other by an active-high signal (pin 19). Thus both the inverted and the non-inverted address decode signals are used.

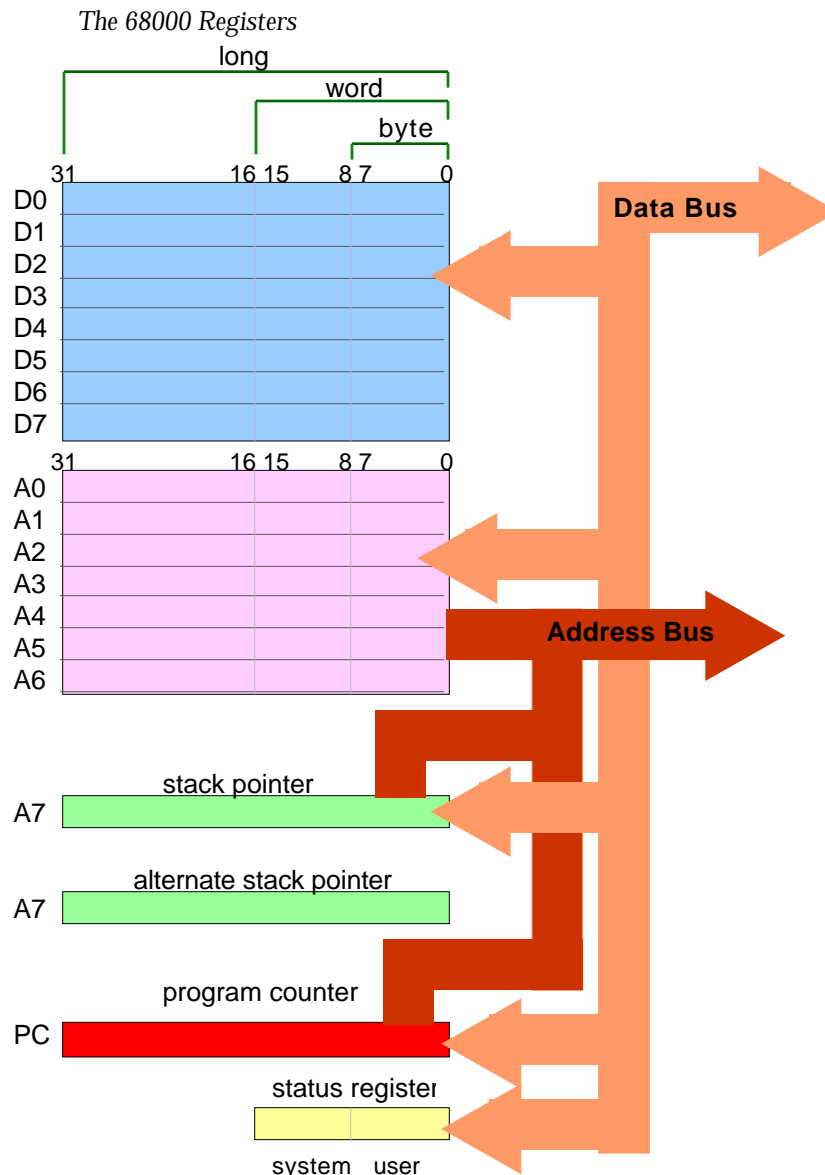


*I/O Read Cycle Timing*

Symbol	Max (ns)	Min (ns)
t1	-	209.5
t2	-	124.5
t3	-	71.8
t4	15	-
t5	15	-
t6	128	16
t7	-	91.5
t8	35	10
t9	-	42
t10	-	10
t11	35	10
t12	-	551.5
t13	-	668

**The 68000 Series of Microprocessors**

The Motorola 68000 series of microprocessors is used in many small computers like the Macintosh, Amiga, Atari and NeXT. These microprocessors are derived from the older 6800 chip, but Motorola, unlike Intel, decided to completely redesign the architecture and instruction set when upgrading from its first-generation products. All members of the 68000 series have internal 32-bit registers. There are eight data registers, seven address registers, two stack pointers a program counter and a 16-bit status register. Notice the refreshing lack of segment registers. The lowest member of the series, the 68008, interfaces to an eight-bit data bus and 20-bit address bus labelled A0 through A19. The standard-bearer, the 68000 has a 16-bit data bus and a 23-bit address bus with lines labelled A1 through A23. Because of the 16-bit data bus, the lowest bit of the address is unnecessary. Only 16-bit words beginning with an even-numbered byte can be addressed.



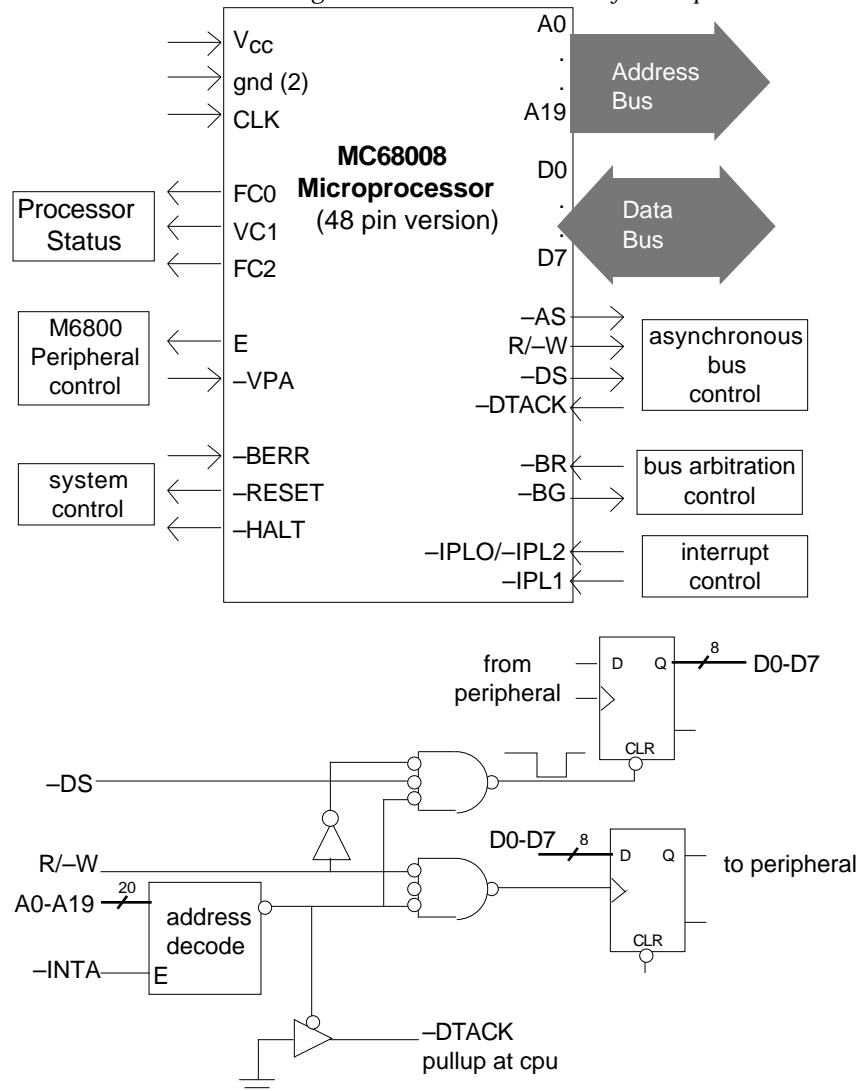
Data are moved between registers or between a register and a memory location using the MOVE instruction. One may request a byte move, a 16-bit word move or a full 32-bit "long" word move by suffixing the MOVE with either a B, W or L. For example MOVE.W D0,D1 moves the 16 low-order bits of register D0 to register D1. MOVE.B D2,\$C0000 moves the low-order byte of register D2 to memory location \$C0000. The full 32-bit memory address was included in the instruction. It is also possible to use the contents of one of the address registers to refer to a memory location. MOVE.L D3,(A0) would move all 32 bits of data register D3 to the four bytes of memory starting at the location whose address is in A0. The contents of A0 would have been previously loaded with a MOVEA instruction. Other addressing modes are possible such as post-incrementing, pre-decrementing, and immediate. The versatility of the MOVE instruction illustrates the convenient way the 68000 instruction set has been designed. All members of the 68000 series use the same instructions. Further information on the instruction set can be found in Horowitz and Hill (2nd ed.) or other good books on this microprocessor.

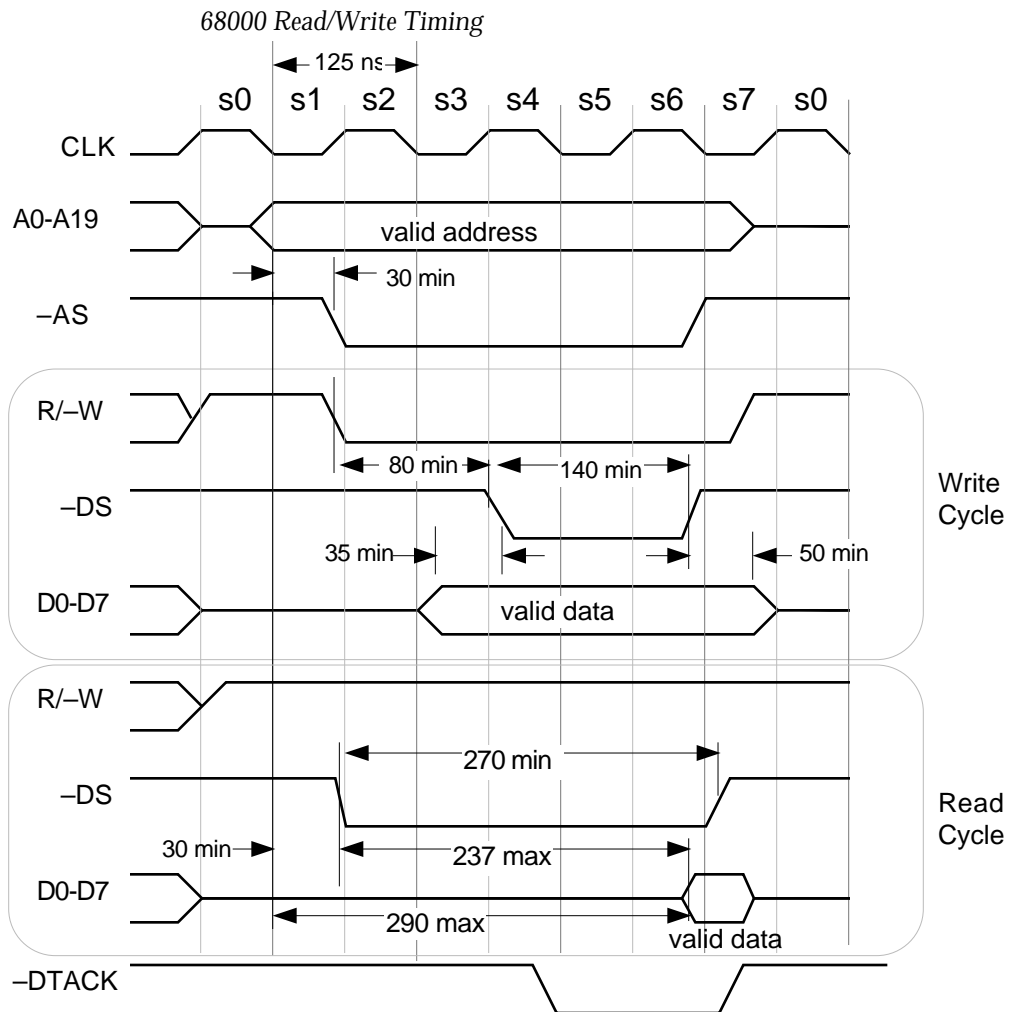
There are no separate instructions for peripheral input and output. The 68000 uses memory-mapped I/O. This is really no problem because the extra one or two signal lines needed for port I/O such as Intel uses can provide for addressing more memory. In addition, the full power of the instruction set can be used on peripheral data as well as on memory data. The IN and OUT instructions in the 8088 require all data to pass through the accumulator (register A). A simple bit-test of an I/O port requires about five instructions. One instruction suffices if you are using the 68000

instruction set. The only inconvenience with memory-mapped I/O is that all address bits must be decoded on peripheral interface boards instead of just those consecrated for an I/O port.

The bus interfacing signals also reveal a slightly different philosophy at Motorola as opposed to Intel. There is one R/-W signal which indicates whether the next data transfer will be input or output. The "data strobe" line, -DS, initiates the data transfer after valid data are on the bus and the R/-W line has been set up. There is an analogous -AS, "address strobe," which indicates a valid address on the address bus, but it isn't usually needed. The -DTACK signal allows peripherals to slow down the read process until the data are ready. -DTACK is by default high and can be pulled low by any peripheral using a "wired or," open collector arrangement. After asserting -DS the cpu expects an acknowledgment in the form of -DTACK going low. If this acknowledgment doesn't occur before the end of clock cycle S4, extra wait states are inserted. Reasonably competent (fast) peripheral devices will assert -DTACK upon decoding its address. If one is sure all devices are fast enough, -DTACK can be permanently tied low for so-called "DTACK-grounded" operation. (Other similar terms are "zero-wait-state" and "pedal-to-the-metal.")

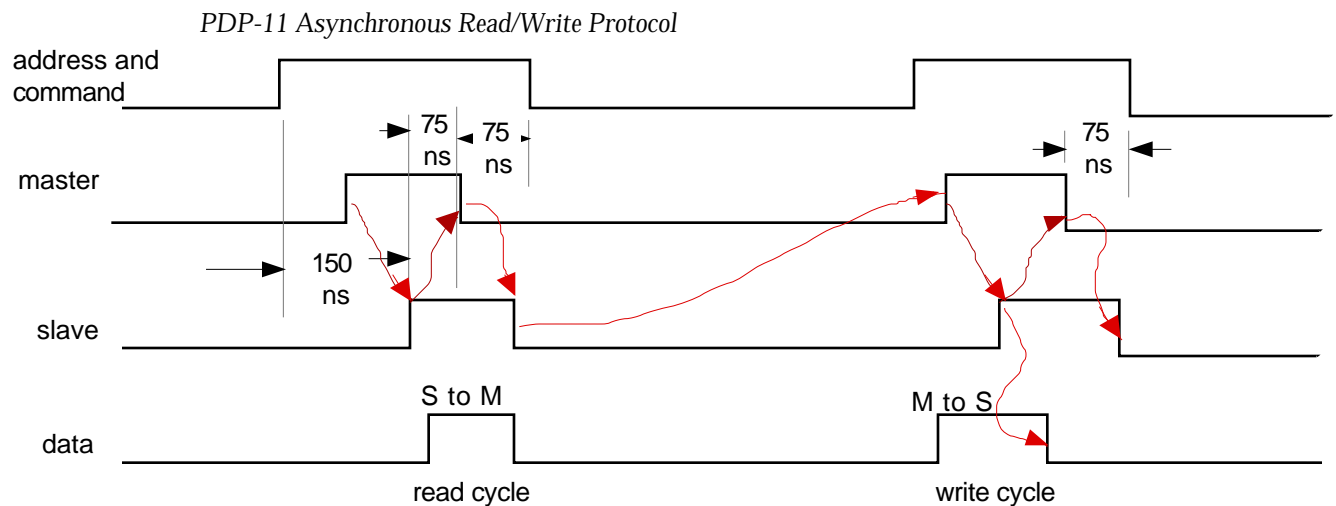
*Motorola 68000 Bus Signals and Interface Circuitry Example*





### Fully Asynchronous Bus Protocols

The PDP-11 unibus is an example of asynchronous handshaking protocol. When the CPU wants to convey data to a peripheral or memory, it puts the address on the bus and activates the MASTER line. The peripheral reads the data and then raises the SLAVE line to say, "I've got it." The MASTER line is then dropped to say, "I see you got it" and then the SLAVE drops signifying "I see you see I got it." Similarly the read cycle starts with the CPU putting the address on the line and then raises MASTER. The peripheral then puts the data on the line and raises the SLAVE line. When the CPU gets the data it lowers MASTER and the peripheral acknowledges by lowering SLAVE and removing the data from the bus. IBM's microchannel bus uses asynchronous handshaking.



**Simplified 8086/8088 Instruction Set**arithmetic

MOV	<i>b,a</i>	move	<i>a b</i> ; <i>a</i> unchanged
ADD	<i>b,a</i>	add	<i>a + b b</i> ; <i>a</i> unchanged
SUB	<i>b,a</i>	subtract	<i>a - b b</i> ; <i>a</i> unchanged
AND	<i>b,a</i>	and	<i>a AND b b</i> bitwise; <i>a</i> unchanged
OR	<i>b,a</i>	or	<i>a OR b b</i> bitwise; <i>a</i> unchanged
CMP	<i>b,a</i>	compare	set flags as if <i>a - b</i> ; <i>a, b</i> unchanged
INC	<i>rm</i>	increment	<i>rm + 1 rm</i>
DEC	<i>rm</i>	decrement	<i>rm - 1 rm</i>
NOT	<i>rm</i>	not	1's complement of <i>rm rm</i>
NEG	<i>rm</i>	negate	2's complement of <i>rm rm</i>

stack

PUSH	<i>rm</i>	push	push <i>rm</i> onto stack (2 bytes)
POP	<i>rm</i>	pop	pop 2 bytes from stack into <i>rm</i>

control

JMP	<i>label</i>	jump	jump to instruction <i>label</i>
Jcc	<i>label</i>	jump	jump to instruction <i>label</i> if <i>cc</i> is true
CALL	<i>label</i>	call	push next address, jump to instruction <i>label</i>
RET		return	pop stack and jump to that address
IRET		return from interrupt	pop stack, restore flags and return
STI		set interrupt	enable interrupts
CLI		clear interrupt	disable interrupts

input/output

IN	<i>AX(AL),port</i>	input	<i>port AX</i> (or <i>AL</i> )
OUT	<i>port,AX(AL)</i>	output	<i>AX</i> (or <i>AL</i> ) <i>port</i>

notes

*b,a*: any of *memory,register register,memory memory,immediate immediate,register*

*rm* any of *register* or *memory* via various addressing modes

*cc* any of *Z NZ G GE LE L C NC*  
(zero, not zero, greater than, greater than or equal, less than or equal, less than, carry set, carry not set)

*label* via various addressing modes

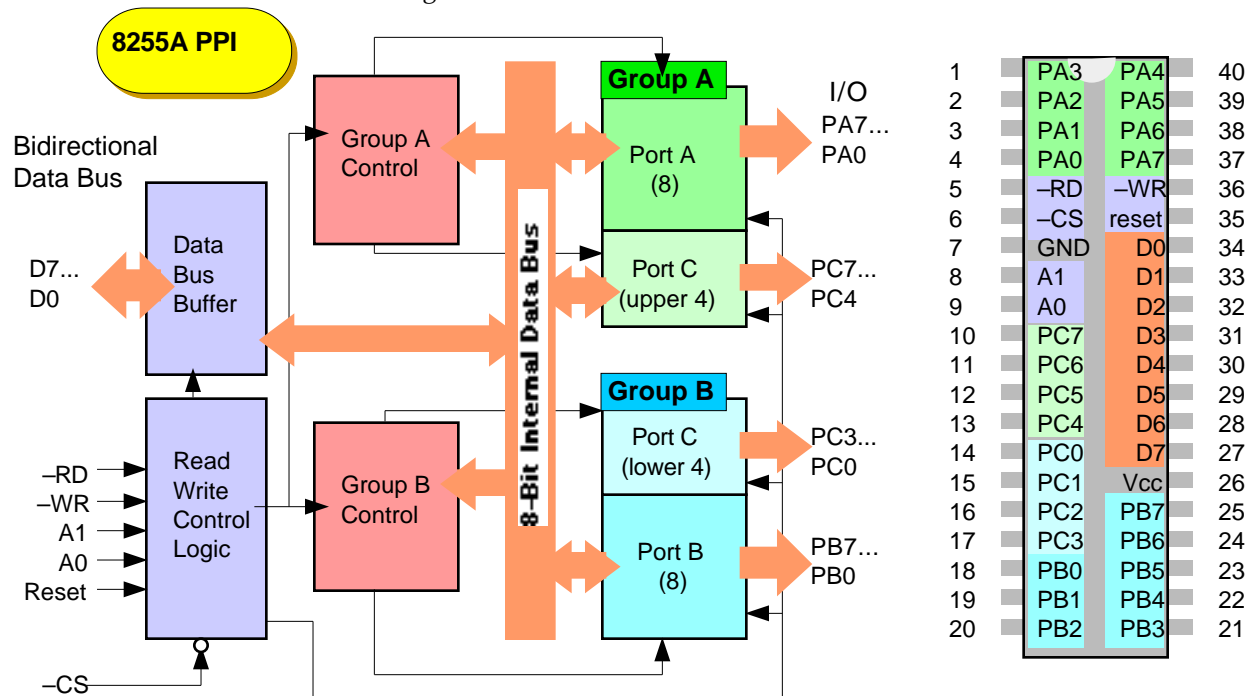
*port* byte (via *imm*) or word (via *DX*)

## Lecture 5

**The 8255 Programmable Peripheral Interface**

The combination of an output data latch and a tristate buffer is very common on peripheral interfaces. A number of specialized integrated circuits have been developed which combine these two functions on one chip. Most of these ICs also provide for handshaking, strobing of input data and hardware interrupt interfacing. One of the earliest of these multi-purpose interfacing chips is Intel's 8255 Programmable Peripheral Interface, PPI. The 8255 was originally developed for the 8008 microprocessor and is useful for interfacing to almost any eight-bit data bus, although it fits Intel's protocol the best. A block diagram of the 8255 below illustrates its functions.

Functional Block Diagram and Pin-out of the P255A PPI



The *computer-side lines* D0-D7 connect directly to the eight-bit data bus for either input or output to the peripheral device. The -CS (chip select) pin is driven by the address decoding circuitry. -CS must be low to enable the PPI's operation. In addition there are two address lines, A0 and A1. These should be connected to two of the buss address lines that are not decoded by the decode logic. This will allow the 8255 to respond to three separate address for three distinct uses. It is convenient to use the lowest address bits from the bus, A0 and A1, to connect to the chip's A0 and A1 pins thus making the chip's three addresses consecutive. The -I/ORD and -I/OWR bus lines connect to the correspondingly labelled pins on the chip.

The *peripheral-side lines* are grouped in to three eight-bit ports, A, B and C. The C port is further subdivided into two groups of four-bit nibbles, called upper C (PC4-PC7) and lower C (PC0-PC3). At any time, only one of the eight-bit ports is active. The active port is chosen by the chip's address lines: A0=0, A1=0 chooses port A; A0=0, A1=1 chooses port B; and A0=1, A1=0 selects port C. The functioning of the chip can be controlled by writing to its control register which is addressed by A0=1, A1=1.

**Mode 0 of the 8255 interface chip**

There are three modes of operating the 8255 PPI. Mode 0 is the simplest. It latches output data and buffers input data but provides no handshaking, strobing or interrupt signals. An eight-bit control word written to the control register determines whether a port is in mode 0, 1 or 2 and whether the port is to be used for input or output. The bits of the control word are described in the illustration below.

*8255 Mode Set Control Word (See 8255 Data sheet fig. 6 for another representation)*

CW Bit	0	1		
D7	bit set/reset	mode set	Determines function of control word	
			Port	Group
D6	mode 0 or 1	mode 2	A,C-up	A
D5	mode 0 or 2	mode 1 or 2	A,C-up	A
D4	output	input	A	A
D3	output	input	C-up	A
D2	mode 0	mode 1	B,C-low	B
D1	output	input	C-low	B

(If D7,D6 = 00: mode 0;  
01: mode 1; 1x: mode 2)

When used for mode definition, bit 7 of the control word is always set to 1. Port A and the lower nibble of C are controlled together and are called group A. Bits 6 and 5 select the mode for both port A and lower C (C0-C3). D6=0, D5=0 chooses mode 0 for group A. D4 determines whether group A is used for input to the computer, or output to the peripheral. 1 is for input and 0 for output. (The mnemonic 1=I, 0=O fails if you forget whether it refers to input to the computer or input to the chip! All references to input or output here are with respect to the computer.) D3 set the input/output state of all four lower C lines. D2 determines whether the mode of port B and upper C is 0 or 1 and, finally, bits D1 and D0 are the I/O bits of port B and upper C respectively.

All 16 control words for mode 0 operation are illustrated in the 8255 data sheet. As an example suppose you want to select mode 0 for all ports with port A and upper C as output and port B and lower C for input. The control word would be determined as follows

Mode select	D7 = 1	
Group A in mode 0	D6 = 0	8 (hex)
	D5 = 0	
Port A is output	D4 = 0	
Port C upper is input	D3 = 1	
Group B in mode 0	D2 = 0	A (hex)
Port B is input	D1 = 1	
Port C lower is output	D0 = 0	

In lab 3, the Q0 output of the 74138 decoder selects the separate latch and data buffer chips when addressed by 300 hex. In Lab 4 we direct Q1 to the –CS line of the 8255. Port A is addressed by 304 hex, port B by 305, port C by 306 and the control register by 307. Every port has its own address even though for mode selection, lower C is grouped with port A and port B is grouped with upper C. In this example Port C may be used either for input or output but a different nibble is involved in each case. An OUT instruction to address 307 hex with the data 8A will set up the operation as specified above.

The output ports latch the data on the bus when the PPI's –WR line goes low. The output port chosen by the address lines will hold the data until the next write operation. The mode 0 data output timing is illustrated below. The port address must be on A0 and A1 before the –WR drops to signal a write request. The data on D0-D7 should be stable at least 100 ns before and 30 ns after the write request signal rises again. This data will appear on the addressed output port no later than 350 ns after –WR rises. The address lines must remain stable 20 ns after –WR rises.

Please refer to the MODE 0 (Basic Output) timing diagram in the 8255 data sheet, p. 3-106

The input ports do not latch the data. When the –RD line drops, whatever is on the addressed input port at that time will appear on D0–D7 and be read by the CPU. The timing diagram shows the protocol. First the address lines select the port. Then the input data should be present before the –RD requests input. The data lines will contain valid data a maximum of 250 ns after the request. The important point is that the data should be stable on the port during the time that –RD is low, a minimum of 300 ns. (If you are using the newer CMOS version, 82C55A, timings are more



lax, e.g. –RD can be low 150 ns minimum.)

More elaborate interfacing techniques of mode 1 or 2 can be used if the simple mode 0 operation is insufficient.

Please refer to the MODE 0 (Basic Input) timing diagram in the 8255 data sheet, p. 3-105.

### Mode 1 of the 8255

Mode 1 allows the input data to be strobed. That is to say, the data are latched into the buffer when the –STB line goes low. In mode 0, the data were captured at read time. Thus mode 1 allows the peripheral, not the CPU, to determine when valid data will be entered onto the port. It then sets the IBF (input buffer full) flip-flop high to indicate that data are ready for reading. The computer can poll IBF to see if the data are ready. After reading them, it resets the IBF flip-flop to low on the rising edge of the –RD signal. Ports A and B use different pins of port C for these handshaking signals, so they may be hand-shaken separately. The pins of port C that are not used for handshaking may be used for other input or output if needed. Whether they are used for input or output is determined, as before, by the control word bits for upper and lower port C.

Mode 1 output uses a –OBF (output buffer full) flip-flop to signal that data have been written to the output port. –OBF is set by the rising edge of –WR. The peripheral can then acknowledge reception of this data on the –ACK line. A low on –ACK acknowledges data reception and resets the –OBF flip-flop.

### Interrupt processing in Mode 1

Both mode 1 input and output provide an interrupt request signal, INTR. INTR will not be operative unless its interrupt-enable flip-flop is set. Interrupts are enabled by writing a special word to the control register which sets this flip-flop. Likewise interrupt processing for a port is disabled by resetting its flip-flop.

#### Mode 1 Signals and Pins

I/O port	INTE flip-flop bit <i>pin</i>	control word		–STB bit <i>pin</i>	IBF bit <i>pin</i>	–OBF bit <i>pin</i>	–ACK bit <i>pin</i>	INTR bit <i>pin</i>
		set	reset					
in A	C4 13	0xxx1001	0xxx1000	C4 13	C5 12			C3 17
in B	C2 16	0xxx0101	0xxx0100	C2 16	C1 15			C0 14
out A	C6 11	0xxx1101	0xxx1100			C7 10	C6 11	C3 17
out B	C2 16	0xxx0101	0xxx0100			C1 15	C2 16	C0 14

In contrast to the mode definition control word, the control word to set the port C flip-flops has bit 7 = 0. The set/reset status is indicated by bit 0 being 1 or 0 respectively. Bits 1, 2 and 3 is the binary number of the port C flip-flop to be set or reset.

The input INTR line goes high when –STB is 1 and IBF is 1 signalling that data have been loaded. For input, INTR is reset by the rising edge of –RD. The output interrupt request is high in order to signal that output data have been received by the peripheral. It is set when –ACK is 1 and –OBF is 1, and is reset by the falling edge of –WR.

The diagrams in the data sheet illustrate the operation of mode 1.

Please refer to the MODE 1 (Strobed Input) timing diagram in the 8255 data sheet, fig. 9, p. 3-110.  
Please refer to the MODE 1 (Strobed Output) timing diagram in the 8255 data sheet, fig. 11, p. 3-111.

### Mode 2 of the 8255

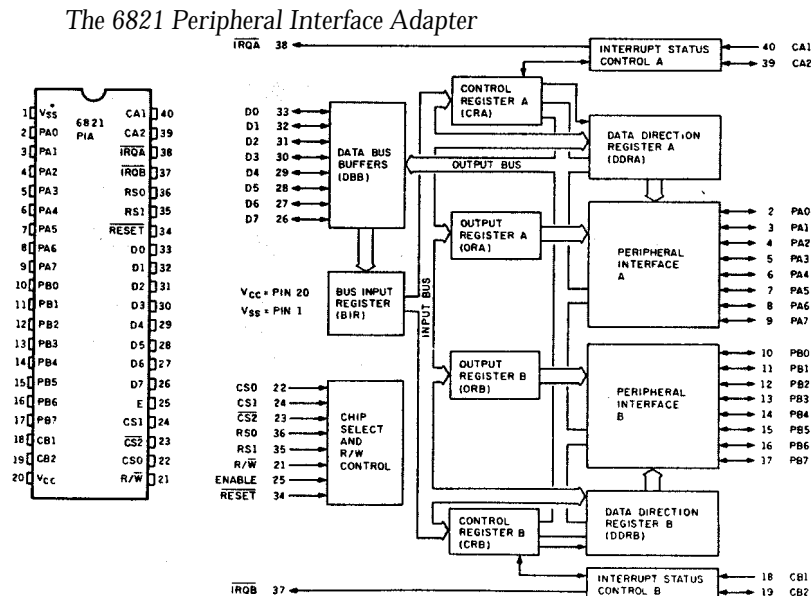
Bidirectional input/output can be handled with mode 2. Only group A is used and the handshaking signals are congruent to the mode 1 signals, except that both the input and output signals

are available. Interrupt enable, INTE, is controlled by the same flip-flops as for mode 1, port A: C6 for output and C4 for input. If  $\text{--STB}$  is 0, input is indicated and the bus can transfer data to the CPU. Similarly, a low  $\text{--ACK}$  enables the three-state output buffer which is otherwise in a high-impedance state. If you understand mode 1, then the following diagrams should be enough to give a general idea of mode 2 operation.

Please refer the MODE 2 (Bidirectional) timing diagram in the 8255 data sheet, fig. 15, p. 3-113.

### Other Interface Chips

There are various other parallel interface chips. The 6821 is a very versatile member of the "68" family which is called PIA for Peripheral Interface Adapter. Its functioning is similar to the 6820 and 6521. The 6821 has six registers occupying four addresses. Each port has one data register, one control register and one data direction register. This last feature distinguishes it from the 8255 because each bit of a port can be defined as either input or output. Handshaking and interrupt signals are also available. The 6822 operates like the 6821 except its outputs are open drain and are capable of interfacing to non-TTL levels up to about 15 V. This is useful in laboratory or industrial applications where many instruments use BCD outputs with voltage levels of 12 to 14 V. This 6822 is called an Industrial Interface Adapter, IIA, for this reason. The diagram and discussion of the 6821 PIA and its relations can be found in Ciarcia's Circuit Cellar of August 1986 *BYTE* magazine, "Parallel Interfacing: A tutorial Discussion."







## Lecture 6

### The 8253 Programmable Interval Timer

The 8253 Programmable Interval Timer can be used for a variety of counting and timing applications. Its interface to the computer bus is almost identical to the 8255 programmable peripheral interface with eight data lines, read and write signal inputs and a chip select and two address pins. Only the reset input is missing. The circuit contains three independent counters and a control register corresponding to the four addresses of a chip. Each counter provides one output and accepts a clock input and a gate input. All timing applications are controlled by the clock inputs which may be any frequency up to 2.6 MHz. If faster clock rates are required, the 8254 or 8254-2 IC will operate up to 10 MHz. Each of the three counters can be programmed to operate in one of six modes by writing to the control register.

#### Programming the control register

In order to use any of the counters, a control word must be sent to the control register to define the mode of that counter and then the counter register must be loaded. The state of a counter is undefined before it is programmed. There is no initial or default start-up state.

The control register is programmed by making  $A0, A1 = 11$  and lowering the  $\text{--WR}$  input. After programming the control register, the counter must receive its data, either one or two bytes depending on bits D4 and D5 of the control word. If a two-byte count has been requested, the least significant byte is sent first and then the most significant. These bytes are written by addressing the counter in question. These count bytes can be sent out any time after the control word for that counter has been sent. For example, the control words for the three counters can be sent consecutively, then all the count bytes for all the three counters. The only thing that matters is that any particular counter gets its own bytes in order: 1. Control word, 2. least significant byte, 3. most significant byte. The format of the first control word is described by the following diagram.

#### The six modes of the 8253

The function of each mode is described very nicely in the data sheet. Here we will just glance at the general function of each mode and leave the details to the official data sheet. In general the counter will count down from the number it has received and then do something with the output line. The gate input can affect the counting in several ways depending on the mode. Sometimes it delays, sometimes it triggers. Counting is always down from the count register value and can be done in binary or decimal depending on the BCD bit, D0, of the control word: 1=binary coded decimal, 0=binary. A control register value of zero specifies the largest possible count:  $2^{16}$  for binary or  $10^4$  for BCD.

Mode 0 is the basic timing operation. The output line starts low after the mode is programmed. Immediately after the count register is loaded, it starts counting down and raises the output when the count reaches zero. The output remains high until the mode is set again. Counting stops temporarily if the gate goes low during the count.

Mode 1 is a programmable one-shot. It is similar to mode 0 except that the output waits until the rising edge of the gate before it goes low. The counting starts at that time and the output rises when finished. If the gate drops and rises again any time during the count the counter is retriggered, i.e., the count restarts from the beginning and the output will remain low for the full count period.

Mode 2 is called a rate generator. It will start with the output low for one clock tick. It then rises for the number of counts programmed and drops for one count again. This repeats. Thus a one-tick low pulse is produced every so many counts. A low gate input forces the output high. When the gate goes low the process starts over again. Thus the gate can be used to synchronize the pulses.

Mode 3 is a square wave generator. Like mode 2, it switches from high to low every so many counts, but unlike mode 2 it spends half the time high and half low if the initial count is even. If the count register is set to an odd number the output spends one more count high than low.

Mode 4 is a software triggered strobe. The output is high when the count is loaded and remains high until the count terminates. Then the output goes low for one tick and rises again. It doesn't repeat. Counting is inhibited when the gate is low.

Mode 5 is a hardware triggered strobe. Here the count starts when the gate rises. The output is low

for only one clock tick after the countdown from the count register value.

#### Reading the counter

The value of the count may be read by the computer. There are two situations: (1) The counting is stopped, or (2) the count is progressing.

In the first case, the gate or a stopped clock may be inhibiting the count. Then a simple IN instruction, addressed to the counter's own port will read the current count. Two bytes must be read if a two-byte load has been specified by the control word written out before. The least significant byte is read first, followed by the most significant byte.

If the count is progressing, the counter value should be latched by writing a special word to the control register. If you try to read the count while it is going on, funny things might happen. The latching command word is just the counter number followed by two zeros. The two high bits, D7,D6 of control word that latches the count indicate the counter you want to read. The next two bits, D5,D4 must be zero and the other bits can be anything. The next read of the counter specified by D5,D4 will return the count value at the time of the latching command.

## Lecture 7

**The Real World**

The pristine world of computing is all too often shattered by the need to deal with the real world. This unforgiving world almost never prepares its data in binary or BCD form. Some control and data monitoring may be done by opening or closing a single switch or monitoring an electrical contact. But in general, if a computer is to control the world, or even a small part of it, internal binary data must be converted to an analog quantity like a voltage or current that is variable. This analog voltage can then be translated to movement, pressure, heat or other physical quantity. On the other hand, if the computer is to measure the world, the analog measures of physical significance have to be converted to digital form. For digital to analog conversion several lines of binary digital data, e.g, 0 or 5 V each, are translated to a one analog voltage. For analog to digital conversion a single variable signal is translated into, say, eight, twelve, or sixteen binary signals.

**Digital -to-Analog Conversion**

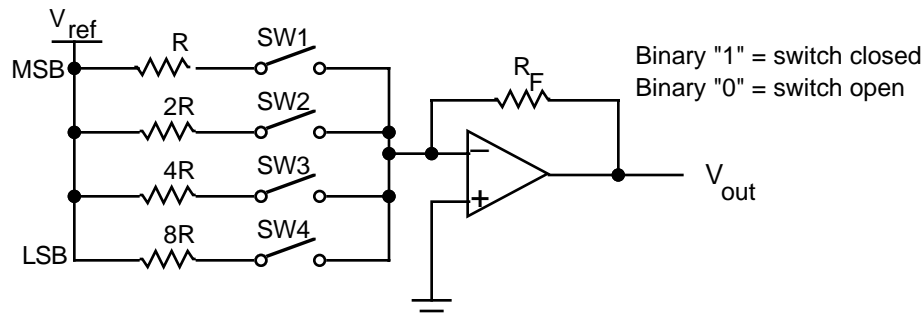
A straight-forward way to convert several lines of binary signals into an analog voltage, is to have each line control a switch which connects a resistor in series with a voltage source. A current proportional to the binary number is obtained if the high order bit switches in a resistance  $R$ , the next lower bit switches in  $2R$ , the next  $4R$ , etc. If eight bits are to be converted, the lowest bit should connect a resistor  $2^7$  or 128 times the value of the resistor controlled by the highest bit. The eight bits,  $D0\dots D7$ , can be thought of as forming binary fraction between 0 and  $1-2^{-8} = 0.9961$ . The current pulled from a voltage source,  $V$ , is given by

$$I_{\text{out}} = \frac{V}{R} \left( D7 + \frac{D6}{2} + \frac{D5}{4} + \frac{D4}{8} + \frac{D3}{16} + \frac{D2}{32} + \frac{D1}{64} + \frac{D0}{128} \right)$$

The negative input of an operational amplifier is a virtual ground, so the circuit below (shown for four bits) will convert this current to a voltage determined by the feedback resistor  $R_f$ .

$$V_{\text{out}} = I_{\text{out}} R_f$$

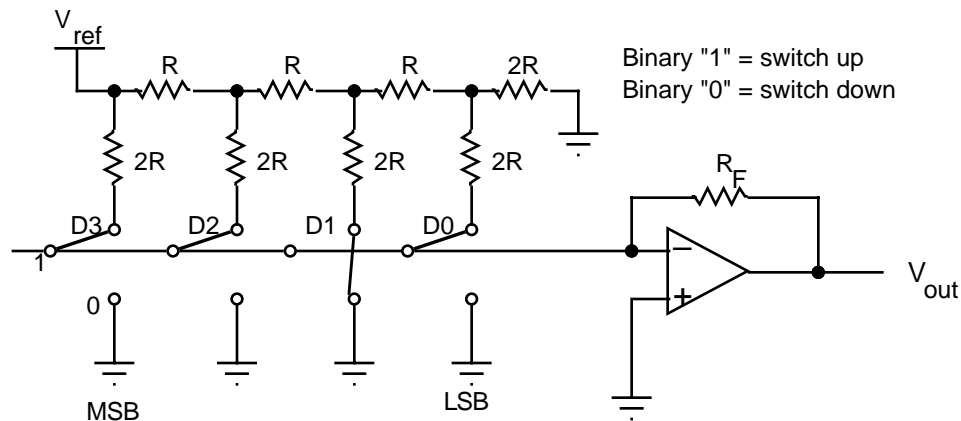
*Simple 4-bit digital-analog converter using a binary resistance ladder*



$$V_{\text{out}} = \left( \frac{-V_{\text{ref}}}{R} + \frac{-V_{\text{ref}}}{2R} + \frac{-V_{\text{ref}}}{4R} + \frac{-V_{\text{ref}}}{8R} \right) R_F$$

This sort of bit-weighted resistor network is convenient for converting a small number of bits, but soon gets out of hand if many more than eight bits are in the number. For example, a sixteen-bit number would require that the smallest resistor be  $2^{-15}$  times smaller than the largest. So if  $R$  were  $100\ \Omega$ , the lowest bit would switch in a resistor of over  $3\ \text{M}\Omega$ . It's difficult to get matched resistors over such a large range of values. Their temperature characteristics are likely much different and accuracy will suffer. Therefore another method can be used which is extendible to as many bits as necessary, the R-2R ladder.

Digital-analog converter using the R-2R Ladder



At each successive node down the ladder, the impedance of each branch is equal so the current flowing into the node is divided equally between the two other branches. Therefore the current in the ladder is halved at each node. The total current flowing from  $V_{ref}$  is  $V_{ref}/R$ . The current flowing to switch D3 is  $V_{ref}/2R$ . The current in switch D2 is one-half that in D3.; the current in D1 is one-quarter that in D3 and so forth. If a switch is in the "1" position, the current is routed through  $R_F$  and contributes to  $V_{out}$ . Otherwise the current flows directly to ground. Thus the current flowing through  $R_F$ , and hence the voltage at the output, is proportional to the binary number selected by the switches D3–D0.

It's very easy to build a ladder from a matched set of resistors. In fact you can use only one resistance value if a series combination is used for the  $2R$  values. This R-2R ladder is found in most packaged A/D converters. In Lab 6 we connect the MC1408 current-output, multiplying A/D converter. It puts out a current which is proportional to a reference current. It is called multiplying because the output current equals the binary input number multiplied by one-half of the reference current:

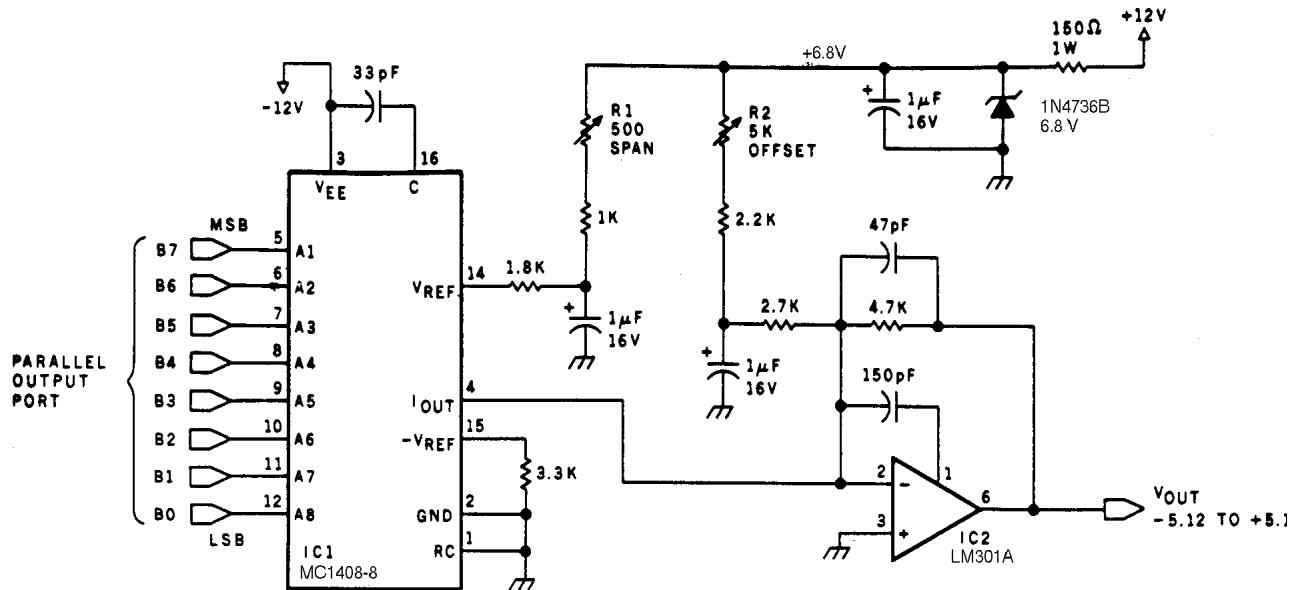
$$I_{out} = I_{ref} \left( \frac{D7}{2} + \frac{D6}{4} + \frac{D5}{8} + \frac{D4}{16} + \frac{D3}{32} + \frac{D2}{64} + \frac{D1}{128} + \frac{D0}{256} \right).$$

D7 is the most significant bit and D0, the least significant bit. The Motorola data sheet refers to the most significant bit as A1 and the least as A8. The settling time of the 1408's output is about 300 ns.

The current output is usually converted to a voltage by sending it to the input of an operational amplifier. The op amp keeps the input at ground potential but, because of the amplifier's high input impedance, almost all the current must pass through the feedback resistor. Therefore, the voltage at the output is  $-I_{out}R_F$ . Usually  $I_{ref}$  and  $I_{out}$  both flow into the circuit. They are negative and the output voltage from the op amp is positive. The following circuit shows the 1408 connected to an op amp and provides both a span and offset adjustment.

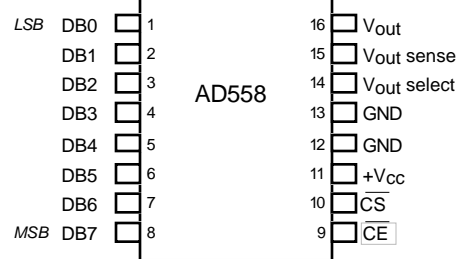


Schematic diagram of the 8-bit MC1408-8-based multiplying digital-to-analog converter with span and offset adjustments

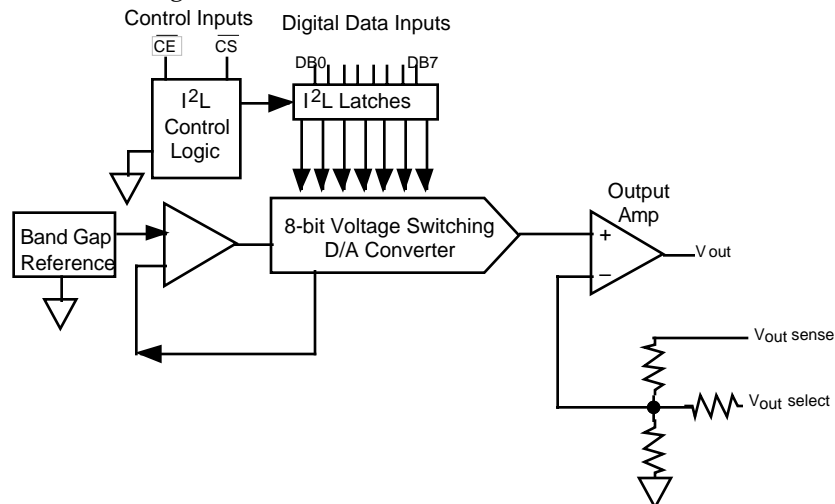


There exist pre-packaged D/A converters containing address decoding logic along with the multiplying D/A and current to voltage converter. The Analog Devices AD558 takes write enable ( $-\overline{WR}$  or  $-\overline{CE}$ ) and chip select ( $-\overline{CS}$ ) inputs as well as eight data bits and it outputs a voltage between 0 to 2.56 V. The AD558 doesn't need a separate reference signal and requires fewer external components. Its settling time is less than 1  $\mu$ s.

The Analog Devices AD558 digital-to-analog converter



AD558 Functional Block Diagram

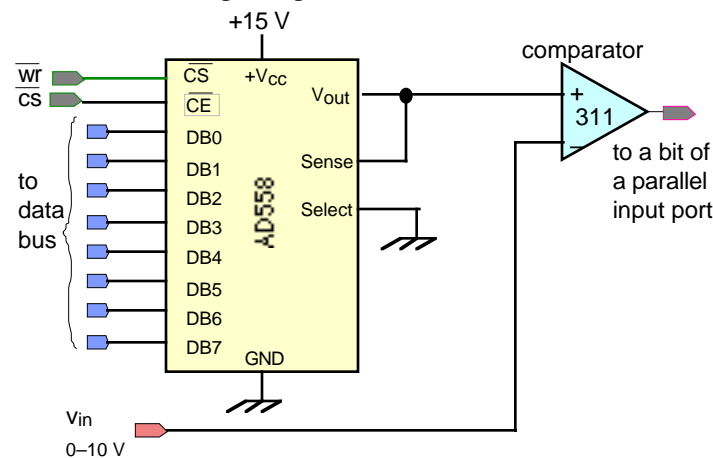


## Analog-to-Digital Conversion

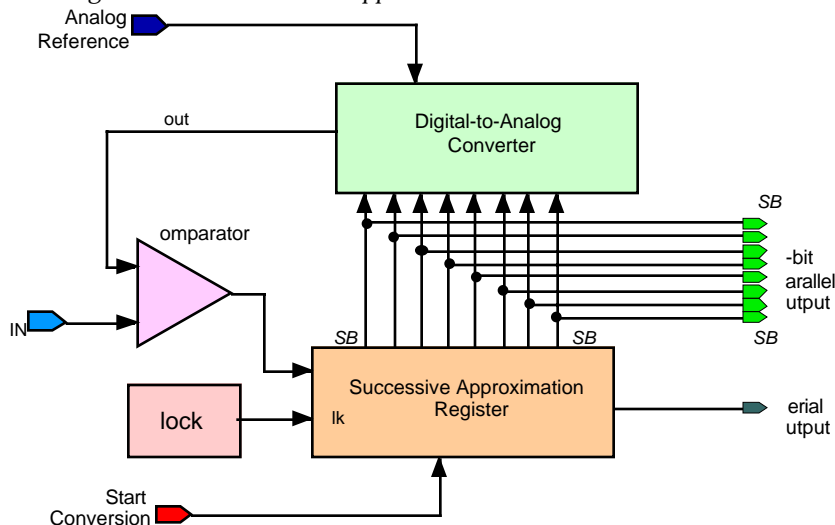
Conversion of an analog voltage to a binary value is fairly easy once a Digital to Analog converter is available. Though there are several methods of doing D/A conversion a common method is just a guessing game. The analog voltage to be digitized is compared to the output of a D/A where an initial "guess" is used to drive the D/A. The comparator circuit is just an op amp which saturates at about +5 V or –5 V depending on whether the guess is too high or too low. The output of this comparator could be monitored by an input port of the computer and a software algorithm could be used to home in on the right value. Alternatively, if faster conversions are necessary, a hardware circuit could search for the right answer.

The easiest search method consists of starting from zero and increasing the guess until it gets too high. A faster method starts in the middle of the range and continues to divide the range by halves until its finished. This is done by sequentially turning on the bits from high to low and leaving them on or off according to the result of the comparison. Software and hardware implementations of this successive approximation method are illustrated below.

*Software controlled 8-bit analog-to-digital converter*



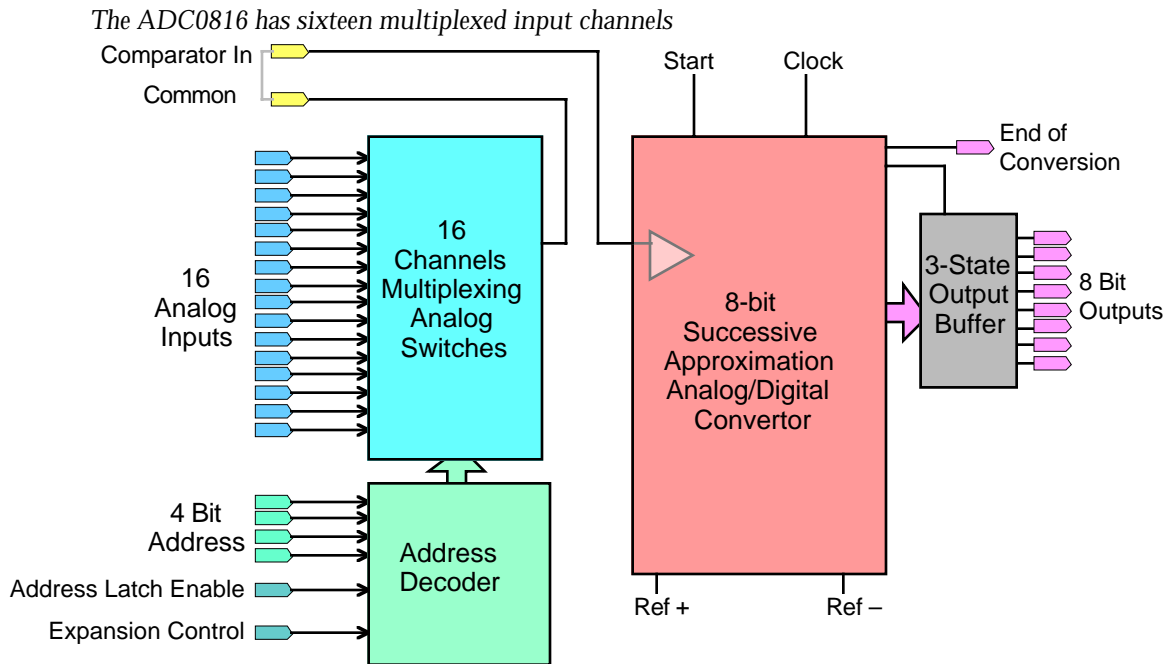
*Block diagram of 8-bit successive approximation A/D conversion*



In Lab 6 we will play with the ADC0801 A/D converter. Besides an input for the voltage to be converted, there are several control lines for interfacing to the computer bus.  $\overline{CS}$  is the chip select signal which can come from the address decoding circuit or it can be grounded permanently.  $\overline{WR}$  is the write input which is used for resetting and initiating the data conversion. A 0 to 1 transition on this line resets the converter. The  $\overline{INTR}$  output goes low when a conversion has finished and rises again after either  $\overline{RD}$  and

–WR drops because of either a read or a reset. The ADC0801 contains an internal clock with nominal frequency of 640 kHz. A conversion takes 66 to 73 of its clock cycles so a typical conversion could take up to 12 ms. The ADC0801 data sheet shows various configurations for using the converter.

If several analog inputs are to be monitored, the ADC0816 (National Semiconductor) has up to sixteen multiplexed inputs which are addressed with four address bits and can be read into a single tri-state latch buffer.

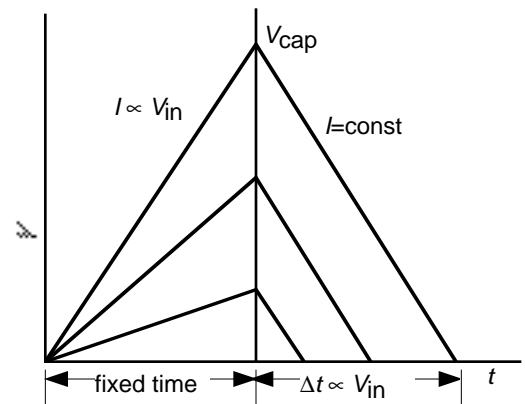
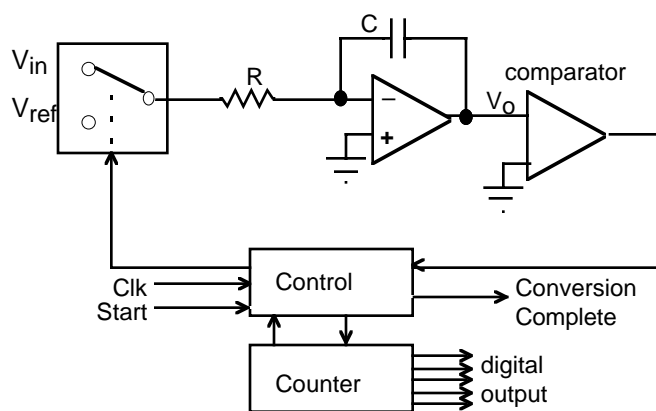


### Other A/D Conversion Methods

If greater accuracy is required and speed is not so important, a voltage-to-frequency converter produces pulses at a rate which is proportional to the input voltage. The digital value is obtained by counting the number of pulses in a known time period. Long integration times provide very accurate, average values of the quantity measured.

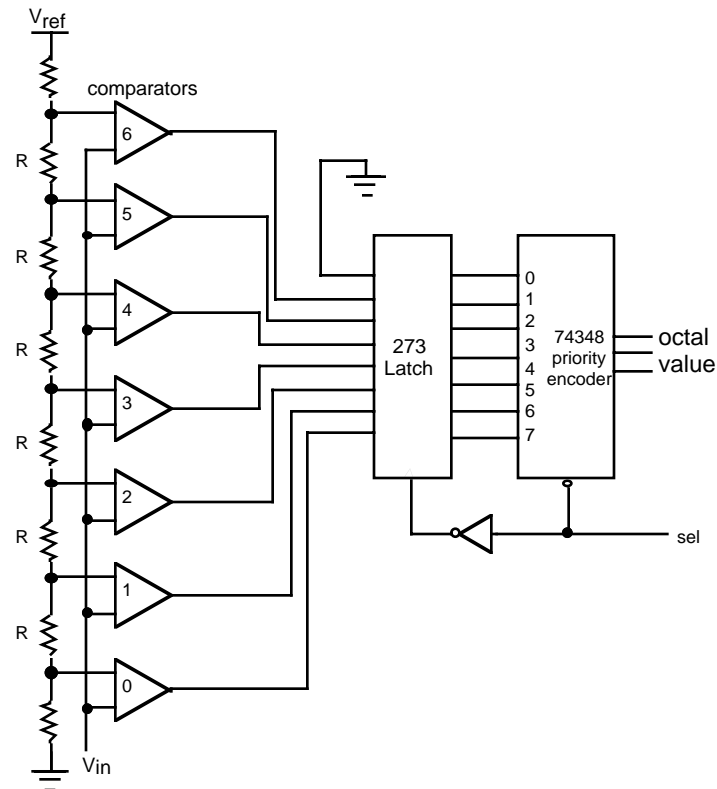
Another method for slow but accurate results is the dual-slope integrating A/D converter. It integrates a positive input voltage  $V_{in}$  for a fixed time  $t_1$  and then integrates a fixed negative reference voltage  $V_{ref}$  until the op amp output voltage is driven back to zero. The time  $t_2$  required to get the output back to zero is a measure of  $V_{in}$ .

### A Dual-Slope Integrating A/D Converter



For very fast conversions, the flash A/D converter simultaneously compares the input voltage to a series of reference voltages through several comparators. In the diagram below, equally spaced voltage levels are compared. The 74348 priority encoder converts the outputs of the comparators to a three-bit octal value. A typical conversion time is about 20 ns.

### A Flash A/D Converter



### A-D Conversion Problems

In principle an analog quantity can have an uncountable infinity of possible values whereas a digital quantity can have only a countable infinity of possible values. This fundamental difference becomes irrelevant in real world measurements because of measurement inaccuracies in the instrumentation or the operator. If we confine ourselves to a fixed interval, 0–1 V for example, any measured analog quantity can have only a finite number of *distinguishable* values. The number of bits that must be used so that the digital representation does not suffer a loss in accuracy depends on the accuracy of the measurement method. Common A/D converters used 8, 12 or 16 bits. Exceptional cases may require more bits. No conversion can represent a value with greater accuracy than  $\pm 1/2$  the value of the least significant bit. For eight bits this accuracy is  $1/256$  or  $\pm 0.2\%$  of the conversion range. In practice it is preferable for the conversion to be much more accurate than the measurement itself.

The number of bits in the conversion sets a limit on the analog-digital conversion accuracy. Other errors may creep into the conversion process as illustrated in the following figure.

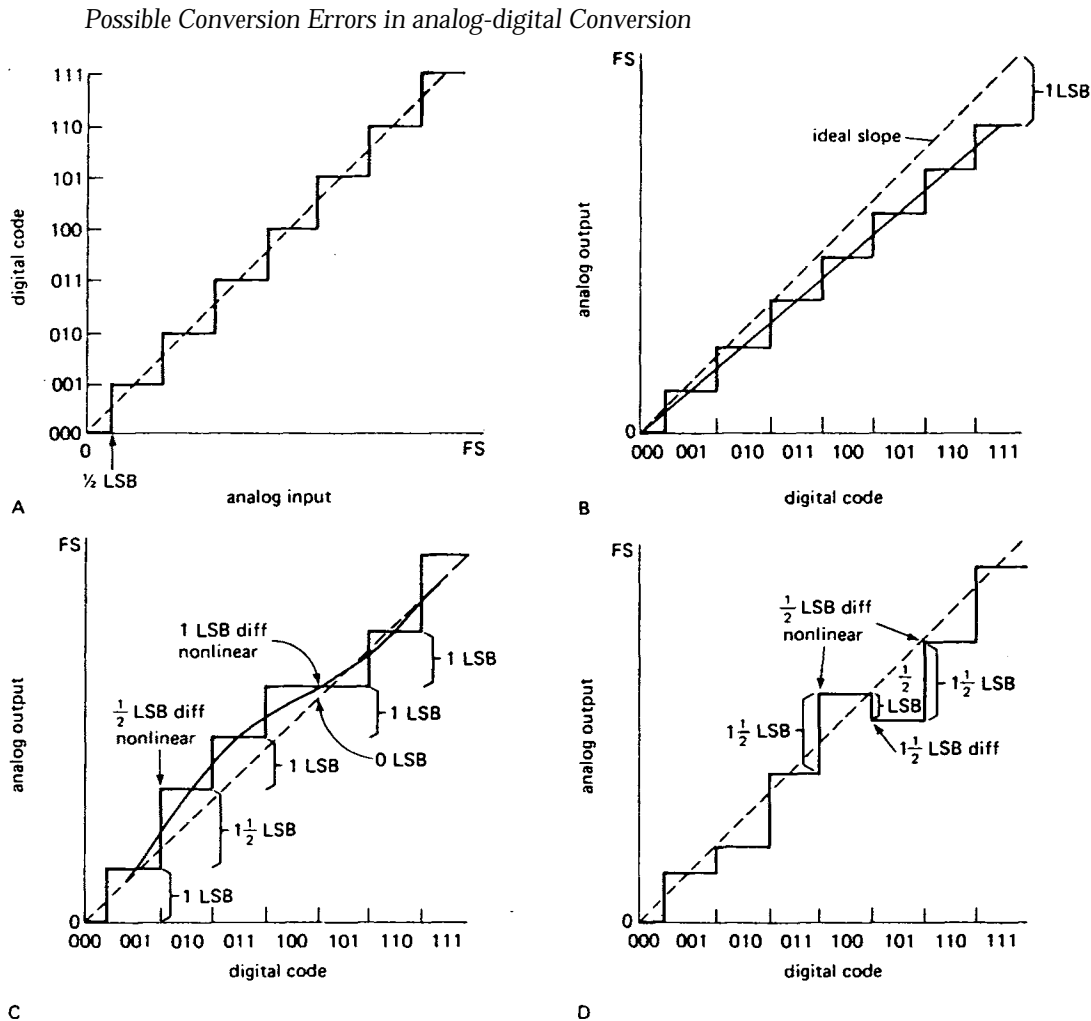


Figure 9.44. Graphs illustrating the definitions of four common digital conversion errors. (Courtesy of National Semiconductor Corp.).

A. ADC transfer curve,  $\frac{1}{2}$  LBS offset at zero.

B. Linear, 1LSB scale error.

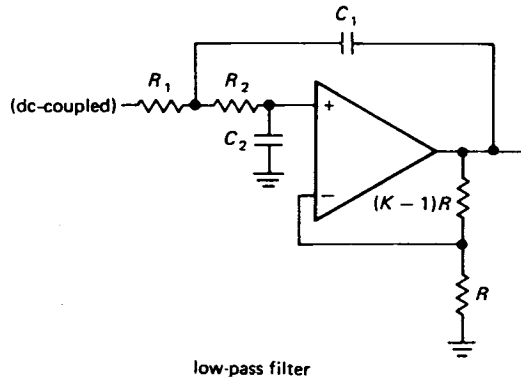
C.  $\pm \frac{1}{2}$  LSB nonlinearity (implies 1 LSB possible error); 1 LSB differential nonlinearity (implies monotonicity).

D. Nonmonotonic (must be  $> \pm \frac{1}{2}$  LSB nonlinear).

The time scale of conversion is another consideration. Any input signal may contain a range of frequency components. The upper limit of the conversion frequency is limited by Nyquist's theorem which states that the sampling rate should be at least twice the highest frequency present in the input. Higher frequencies will be distorted after digitization. High frequencies sampled at too low a rate are "aliased" to lower frequencies, *i.e.*, they appear to be a lower frequency in the reconstructed waveform. Thus the sampling rate sets an upper limit to the frequency which can be digitized and higher frequencies must be removed before digitization by using a preconditioning filter.

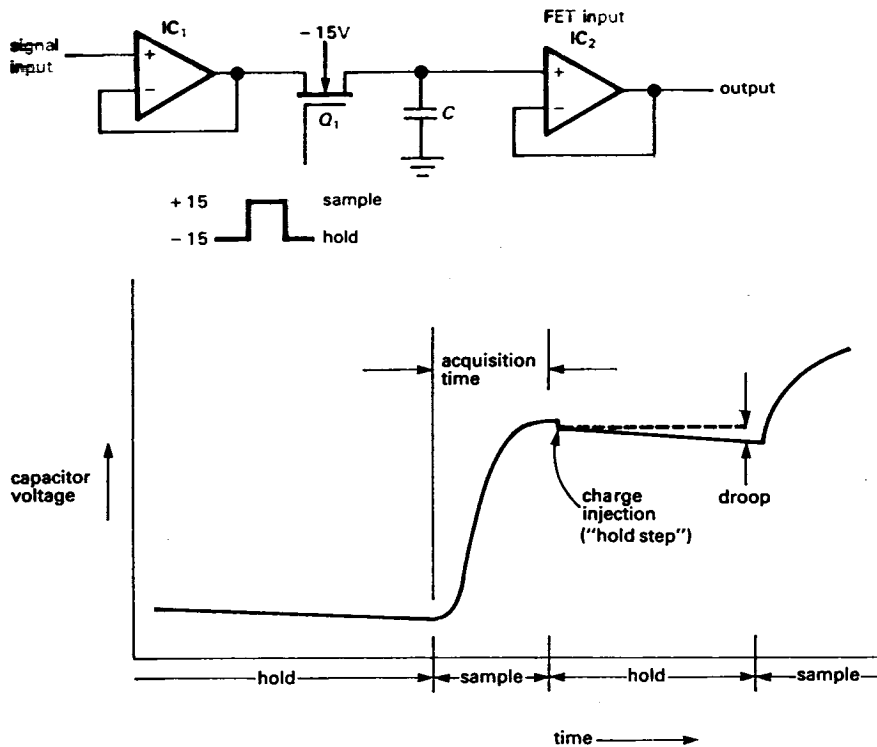
An active low-pass filter works better than a passive RC filter to eliminate unwanted high frequencies. Remember, if the critical frequency of a passive low-pass filter is set at 10,000 Hz, for example, there still remains after filtering about 10% of any component around 100,000 Hz and 50% at 20,000 Hz. An active filter, such as the Butterworth filter, has much better stop-band attenuation. The two-pole Butterworth design attenuates 40dB (99%) at  $f = 10f_c$  and 12 dB (75%) at  $f = 2f_c$ . Better stop-band response can be achieved with more poles and other designs, such as the Chebyshev. Better stop-band attenuation is usually achieved at the expense of flat frequency response in the pass band.

*Design of a two-pole low-pass filter. For the Butterworth design choose  $R_1 = R_2$  and  $C_1 = C_2$ ,  $F_c = 1/2\pi RC$  and  $K=1.586$ .*

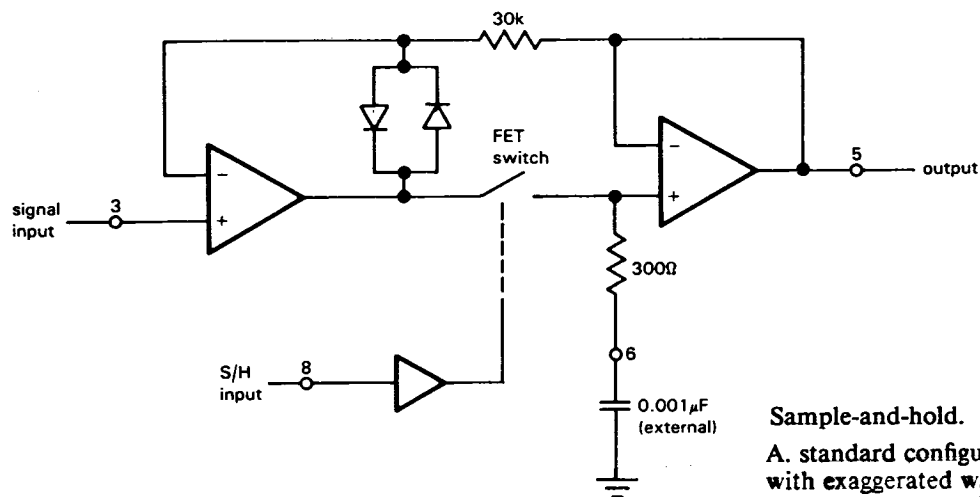


Another issue of concern is variation of the input signal during the conversion process. A rapidly varying input can confound a successive-approximation converter. Other methods, such as dual-slope or voltage-frequency conversion may not be bothered by rapidly varying input voltages. A sample-and-hold circuit keeps the voltage on the A/D input constant until conversion is finished. The sample-and-hold circuit relies on charging a capacitor up to the sample voltage and discharging after digitization is complete.

## Sample and Hold circuits



A



Sample-and-hold.

A. standard configuration,  
with exaggerated waveform.

B. LF398 single-chip S/H.

Horowitz and Hill, *The Art of Electronics*, sections 9.15-9.26, p 612 ff

For a good time, read Ciarcia's Circuit Cellar articles on A/D and D/A converters:

BYTE, September 1977, p30, (reprinted in Ciarcia's Circuit Cellar, Vol. I)

BYTE, January 1982, p. 72, (reprinted in Ciarcia's Circuit Cellar Vol VI)

BYTE, January 1986, p. 105.





## Lecture 8

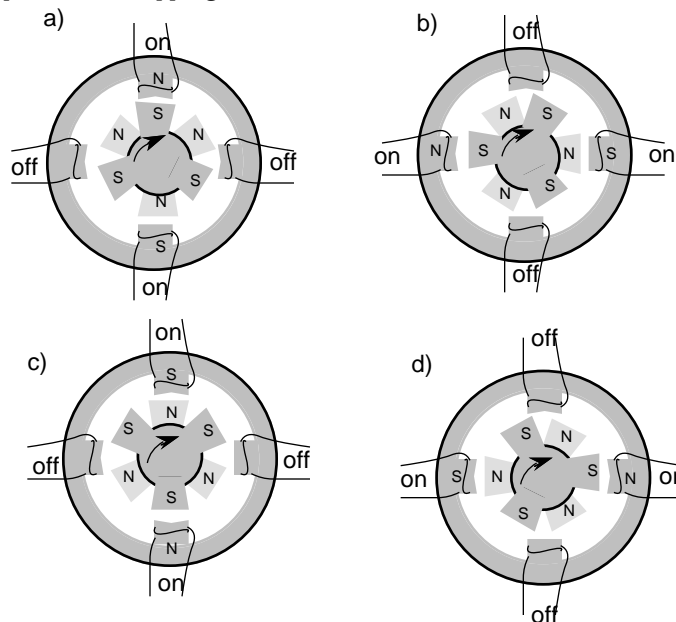
**Motion**

Two common devices used to move things with computers are servo-motors and stepping motors. Servo-motors are called closed-loop devices. There must be some method to measure the position of the motor and feed this position back to the computer. The computer can quickly control the position by reading the current position back to the computer. The computer can quickly control the motor by reading its current position and then sending the motor signals necessary to bring its position to that desired. The other method, stepping motors, move in discrete increments. These can be used in an open-loop fashion where one relies on the motor stepping according to the computer's command.

**Stepping Motors**

Stepping motors advance a fixed amount for each impulse sent from the computer. Motors are available which have step sizes of  $1.8^\circ$ ,  $2^\circ$ ,  $2.5^\circ$ ,  $5^\circ$ ,  $7.5^\circ$ ,  $15^\circ$ ,  $18^\circ$ , and  $30^\circ$ . The smallest step size,  $1.8^\circ$ , gives 200 steps per revolution. The mechanism of stepping motors is illustrated for a simplified case below

*Stepping Sequence of a Stepping Motor*



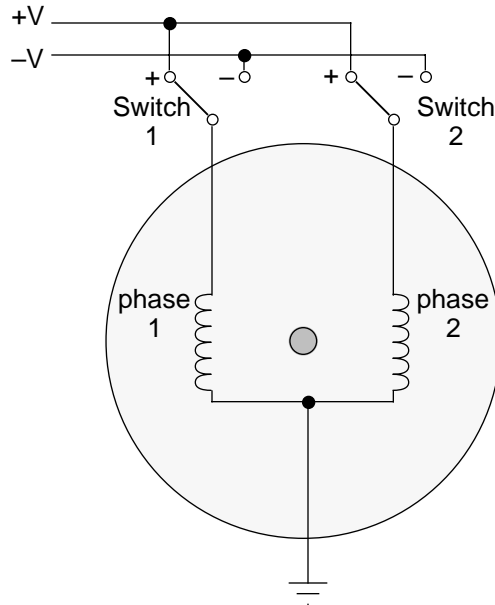
The rotor is magnetized axially with two gear-like hubs at each end. In the case shown the north end has three teeth which are staggered with respect to those of the south end. The rotor is surrounded by a stationary hoop, the stator. The stator has a different number of teeth from the rotor and is not permanently magnetized. The teeth of the stator are magnetized by current-carrying windings so that their polarity can be switched or they can be turned off altogether. By energizing the windings of the stator sequentially, the rotor can be pulled around step by step. a) Two poles of the stator are first magnetized with opposite polarity and the other two are not energized. The rotor aligns itself with the magnetized stator teeth: north rotor aligned with south stator and south rotor aligned with north stator. b) To advance to the next position, the other two stator teeth are magnetized while the originally energized windings are turned off. The polarity of these teeth determine whether the motor advances clockwise or counter-clockwise. c) Now the originally magnetized teeth are reenergized with polarity opposite that in (a) and the other two teeth are not magnetized. Thus the rotor has moved  $90^\circ$  in two steps. The smaller step sizes usually available are achieved by putting more teeth on the rotor hubs and on the stator.

**Unifilar and Bifilar Stepping Motor and their Drives**

The unifilar motor has one winding on each stator tooth as illustrated above. The bifilar motor puts two windings on each tooth, wound in opposite directions. The main advantage of the bifilar design is

that there is no need to reverse the current direction through the windings as in the unifilar design. In the unifilar design either a dual polarity power supply is necessary, or four switching transistors must be connected in an H-bridge. The unifilar motor is sometimes called bipolar and the bifilar is called unipolar. The drive circuit for unifilar and bifilar motors is shown below with switches.

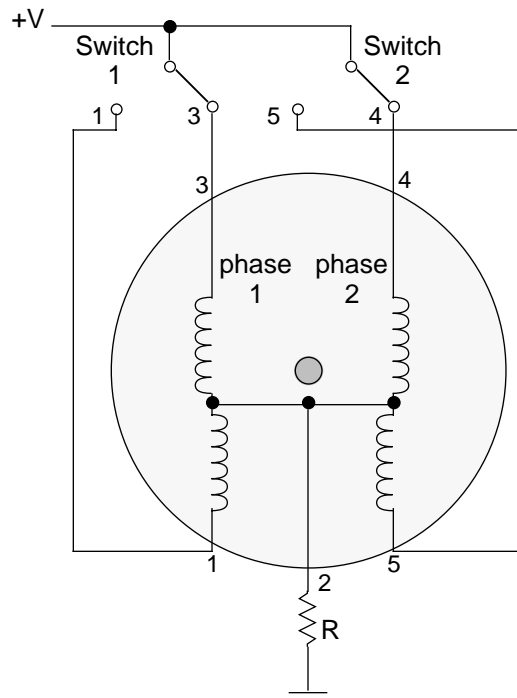
*Stepping Sequence for a Unifilar Stepping Motor*



**Bipolar (unifilar)  
Stepping Sequence**

step	clockwise		counterclockwise	
	Switch 1	Switch 2	Switch 1	Switch 2
1	off	–	off	–
2	+	off	–	off
3	off	+	off	+
4	–	off	+	off

*Stepping Sequence for a Bifilar Stepping Motor*

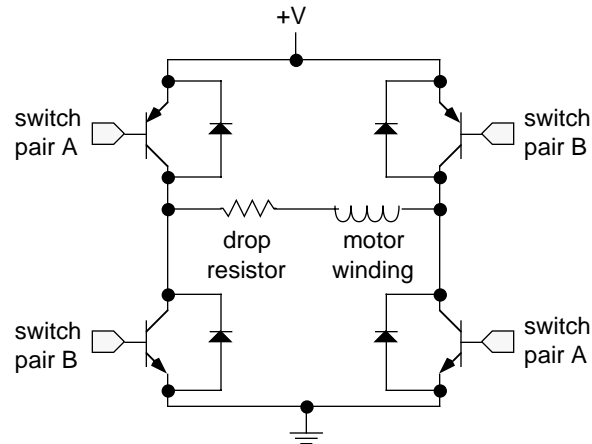


**Unipolar (bifilar)  
Stepping Sequence**

step	clockwise		counterclockwise	
	Switch 1	Switch 2	Switch 1	Switch 2
1	1	5	1	5
2	1	4	3	5
3	3	4	3	4
4	3	5	1	4

Of course nobody really drives stepping motors with little switches. Instead transistor circuits can close the contacts. In order to avoid a dual polarity power supply, the H-bridge can be used with a unipolar power supply to drive a bipolar, unifilar motor.

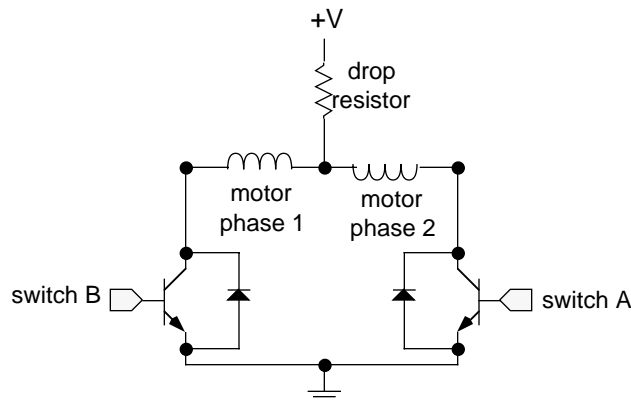
### *H-Drive Circuit for One Winding of a Unifilar Motor*



When using the H-drive circuit, care must be taken not to short circuit the power supply by turning on the wrong two transistors at once. The drop resistor may be necessary to limit the maximum current and increase the response time for energizing the winding. If you want to avoid the power loss through the resistor, there are other methods of increasing the response time using more complicated switching circuits (see Giacomo, BYTE, Feb. 1979, p90).

A bifilar motor requires a much simpler drive circuit.

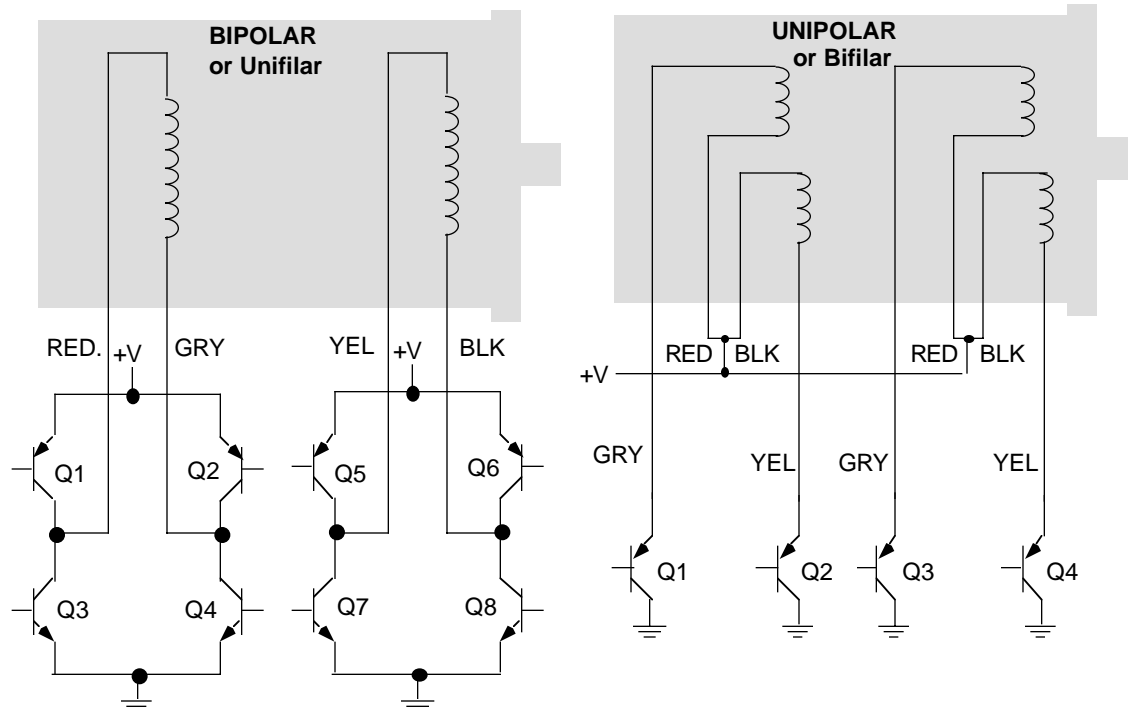
### *Simplified Drive for One-Half of a Bifilar Motor*



The on-off sequence of the transistors for each of these drive systems is illustrated in the diagram on the next page. Beside the normal sequence, a one-half step sequence is shown. This half-step mode gives greater position resolution, but my experience has been that the torques for alternate steps are not equal and can result in uneven steps. The wave-drive sequence only requires one winding to be powered at once, but delivers less torque and less step position accuracy.

The four signals needed for sequencing the transistors can be derived from four bits of an output port. If one uses the normal full step sequence, only two bits need be put out, with the other two being inverses of the first two. The rotation can be controlled in software by successive OUT instructions. An eight-bit byte can control up to four stepping motors. This software approach has the disadvantage of occupying the computer while moving the motors. Furthermore the speed is limited by the loop time of the program which can be very slow in interpreted Basic. However, if a small dedicated microprocessor is being used for the task, and the program compiled or speed isn't important, this is a reasonable method.

*Schematic Bipolar (Unifilar) and Unipolar (Bifilar) Switching Sequence.*



Normal 4-step sequence

Unipolar	Q1	Q2.	Q3	Q4
Bipolar	Q1-Q4	Q2-Q3	Q5-Q8	Q6-Q7
step				
1	ON	OFF	ON	OFF
2	ON	OFF	OFF	ON
3	OFF	ON	OFF	ON
4	OFF	ON	ON	OFF
1	ON	OFF	ON	OFF

Wave Drive 4 Step Sequence

Unipolar	Q1	Q2.	Q3	Q4
Bipolar	Q1-Q4	Q2-Q3	Q5-Q8	Q6-Q7
step				
1	ON	OFF	OFF	OFF
2	OFF	OFF	OFF	ON
3	OFF	ON	OFF	OFF
4	OFF	OFF	ON	OFF
1	ON	OFF	OFF	OFF

1/2 Step 8 Step Sequence

Unipolar	Q1	Q2.	Q3	Q4
Bipolar	Q1-Q4	Q2-Q3	Q5-Q8	Q6-Q7
step				
1	ON	OFF	ON	OFF
2	ON	OFF	OFF	OFF
3	ON	OFF	OFF	ON
4	OFF	OFF	OFF	ON
5	OFF	ON	OFF	ON
6	OFF	ON	OFF	OFF
7	OFF	ON	ON	OFF
8	OFF	OFF	ON	OFF
1	ON	OFF	ON	OFF

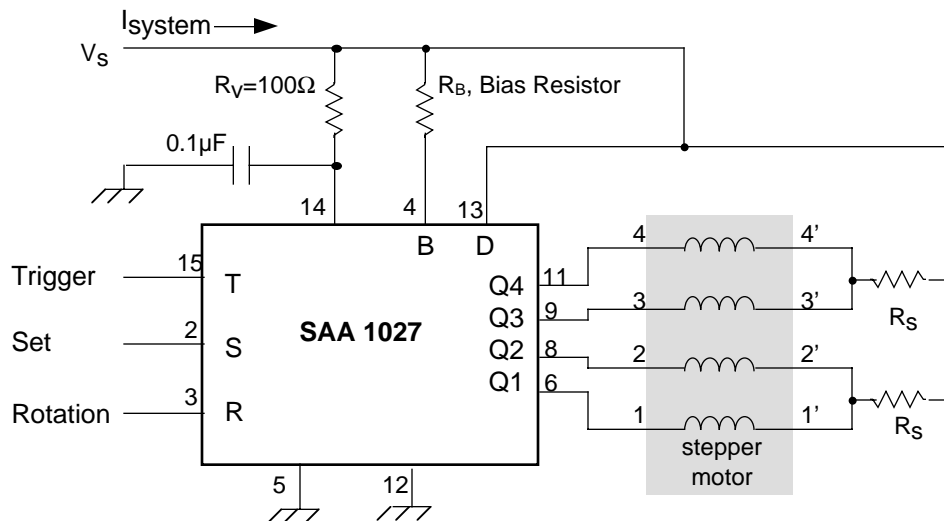
Counter Clockwise  
Rotation (Read Up)

Clockwise Rotation  
(Read Down)

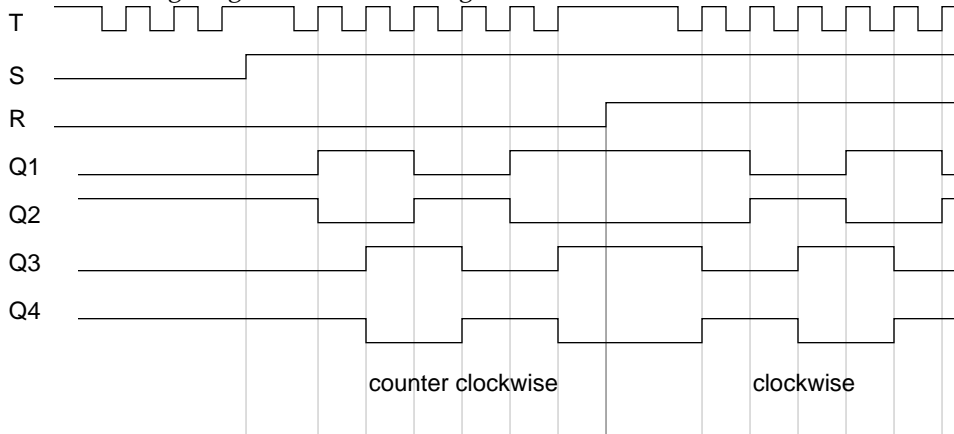
Direction of  
rotation viewed  
from shaft end.

Intersil supplies a chip, originally made by North American Philips (Airpax), to simplify (a little) the control of stepping motors. The SAA1027 takes a trigger input which steps the motor on a high-low-high transition. The direction input determines the direction: high gives counter-clockwise and low, clockwise. The four outputs can directly drive low power stepping motor windings or can be fed to high power transistors, or Darlington pairs, for heavier applications. Of course there is no need to have the computer put out the trigger pulses. A counter could be programmed to pulse the motor and one bit of an output latch could set the direction. This way the computer could be free to do other tasks until the motor is moved into position.

*The SAA1027 Stepping Motor Controller Chip*



*Timing Diagram of SAA 1027 Signals*



### Ramping, Damping and Vibration

Depending on the motor, stepping rates of several thousand steps per second are possible. However, if one tries to immediately drive the motor at a high speed, it is likely just to sit there and buzz. For high stepping rates, it is necessary to begin the motor at a relatively slow speed and then increase its speed gradually. This "ramping" is probably best done by trial and error to determine how fast the motor can be accelerated in each particular application. It should also be noted what stepping rates produce resonance vibrations and these stepping rates avoided.

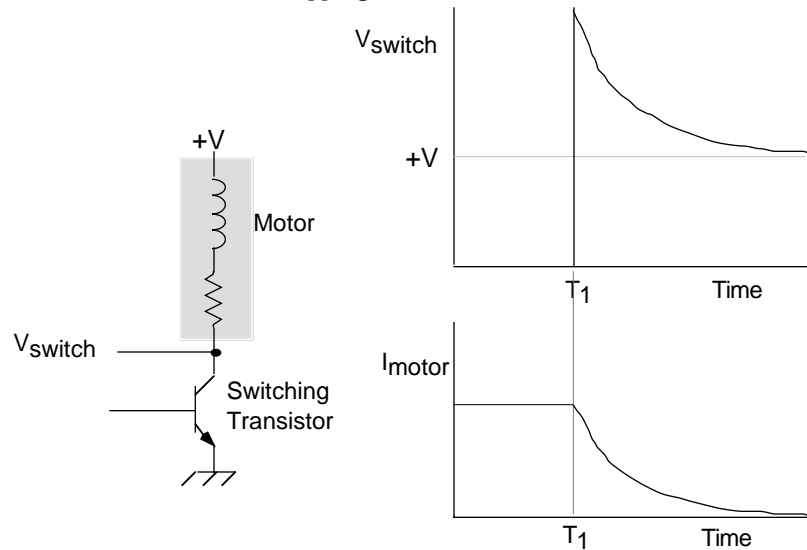
Each single step of the motor may vibrate depending on the power, inertia and friction. If this vibration causes problems, mechanical dampers such as slip pads or a fluid-coupled flywheel can be

inserted but these result in loss of torque. Another method involves applying a time delay, a reverse pulse and then a second time delay and a forward pulse to each step, or to the last step of each movement. Just delaying the final pulse of a movement can also lessen vibrations. Applying more gradually varying wave forms than square waves to the windings is also purported to smooth the motor's operation.

### Transients and Their Suppression

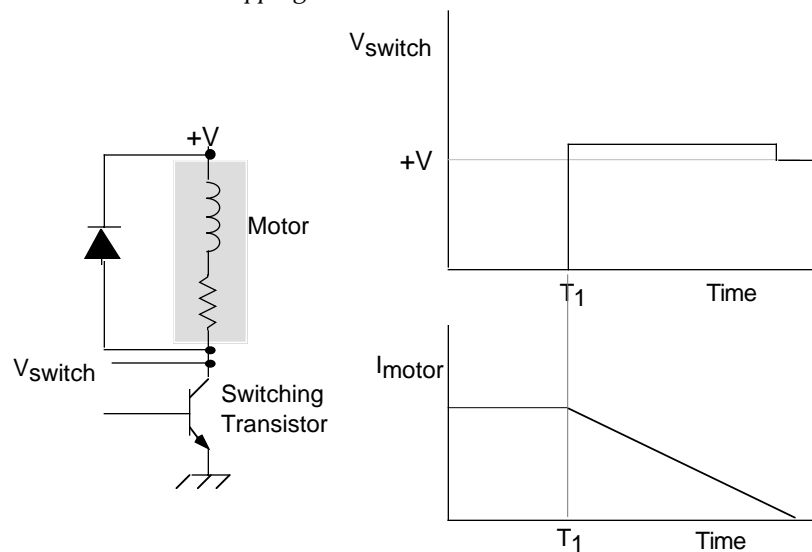
The inductance of the motor windings can cause large transient voltages when the current is abruptly switched on or off. The diagrams below illustrate the problem and suggest several remedies. If current is flowing through the winding when the switching transistor is turned off, the inductance of the winding will maintain this current by inducing a large positive voltage on the collector.

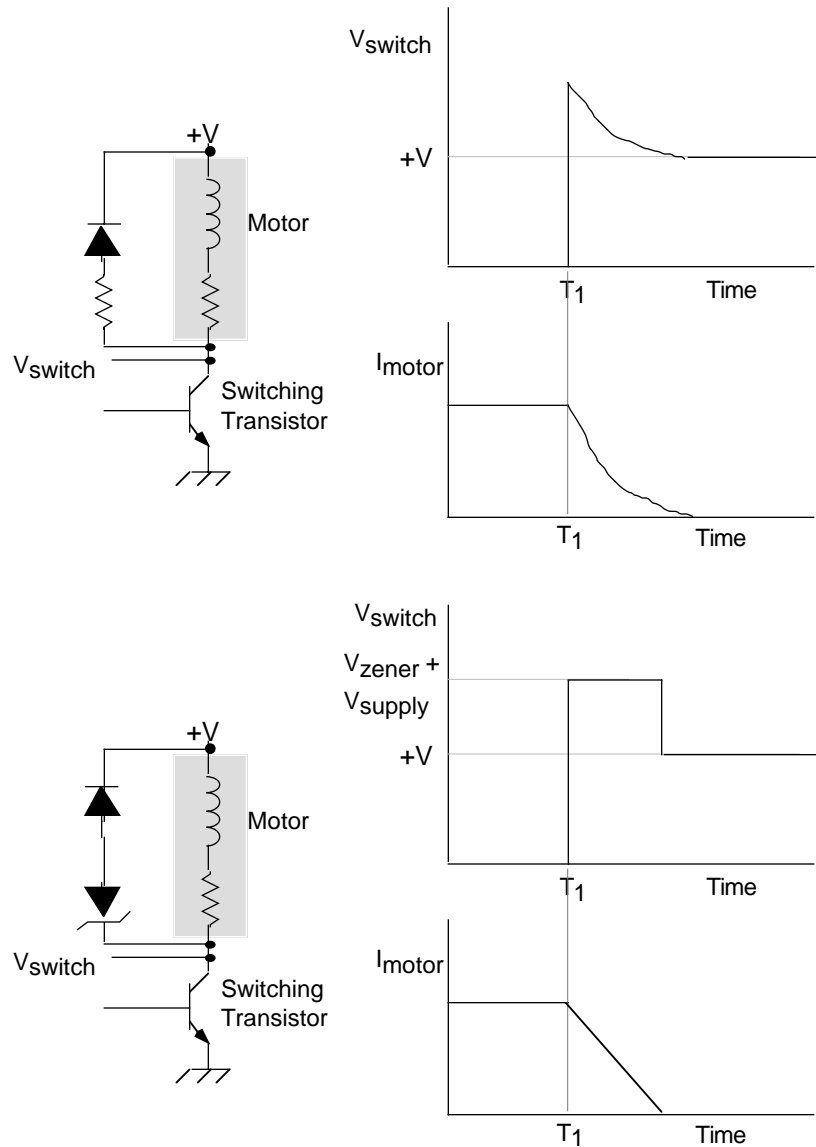
*Stepping Motor Transient Problem*



One solution is to place a diode across the winding. This diode will dump any voltage which builds up on the collector that is over 0.6 V above the supply voltage. Putting a resistor or Zener diode in series with the diode will accelerate the current decay which may be needed for faster stepping rates.

*Suppression Circuits for Stepping Motor Transients*





## References

For Stepping Motors

“A Stepping Motor Primer,” Paul Giacomo, BYTE, February 1979, p 90 and March 1979, p 142  
Stepper Motor Handbook, Airpax, North American Phillips.

For D.C. Motors

“DC Motor Controls: Build a Motorized Platform,” S. Ciarcia, BYTE, May 1981, p 66.  
“Controlling DC Motors,” R.L. Walton, BYTE, July 1978, p 72.





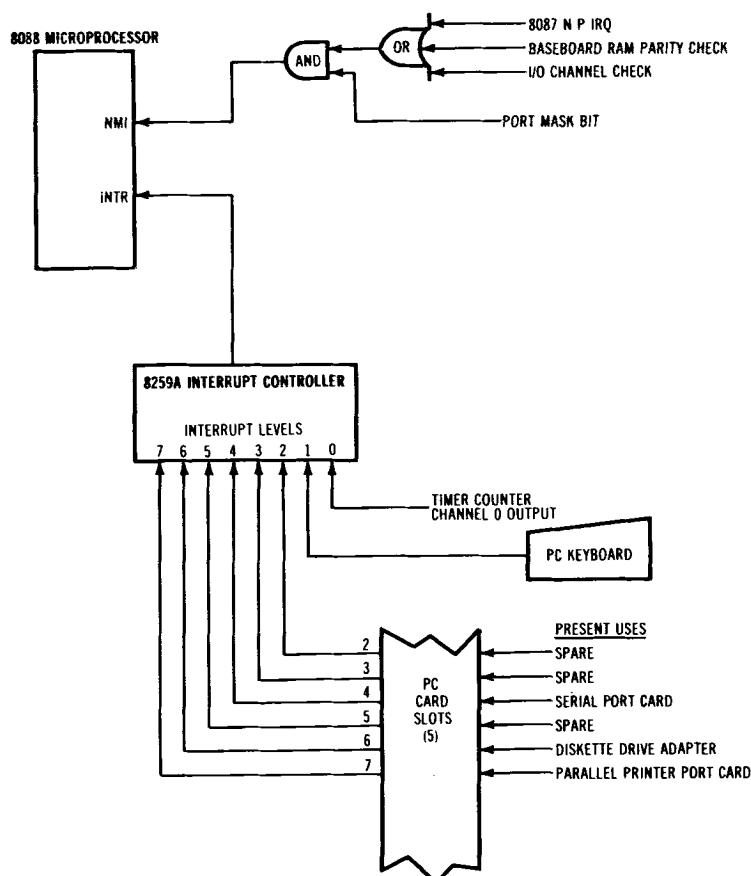
## Lecture 9

**Hardware Interrupts**

A computer is a terrible thing to waste. It's a shame to spend your whole working day watching a simple signal, waiting for just one chance to act. Similarly, to have a computer continually reading from a port just to see if there is some new data needlessly ties up the CPU. If several devices are sending data to or requesting data from the computer, the program must loop around, polling each port in turn: "Can I help you, device 1?" ..., "Anything new this time around, device 5?" ... "Hi, device 17, how's it going, any thing I can do for you?" Obviously, a waste of talent.

A better way is to let the computer do some other tasks (sorting paper clips, filing its fingernails, etc.) and arrange to interrupt these tasks temporarily only when a device really needs attention. This method is called hardware interrupt processing. The 8088 microprocessor has input pins for two interrupt signals. The most urgent interrupt signals shouldn't be ignored. Such interrupts use the Non-Maskable Interrupt (NMI). The NMI cannot be disabled in software, whereas the other interrupt, the Maskable Interrupt, can be turned on and off by machine instructions. In the IBM PC, the NMI is used for RAM parity check, I/O channel check request or auxiliary processor (8087) interrupt request. These NMI interrupts can, in fact, be turned off because all these signals are ANDed with a bit of one of the PC's output ports. To set the NMI mask bit from software requires writing 80h to port A0h. To clear the mask, write 00 to A0h. This is usually done only at power-up time.

*Interrupt System Block Diagram*



Noncatastrophic interrupts can be handled by the maskable interrupt input. Any nontrivial computer system needs more than one maskable interrupt line. Furthermore, a priority system must be established to determine which device gets serviced first in case two devices make a simultaneous request.

The single maskable interrupt input of the 8088 is connected to the output of the 8259A interrupt controller chip which allows eight interrupt inputs. The degree of equality of the interrupt inputs is programmable, and the established order in the PC is from 0 to 7 with 0 having the highest priority.

*Interrupt Priority Table*

Interrupt Level		Usage
Highest Level	NMI	Baseboard RAM parity, I/O channel check, numeric processor.
	IRQ 0	System timer output 8253-5 Channel 0.
	IRQ 1	Keyboard scan code interrupt.
Available in System Bus	IRQ 2	Not used at present.
	IRQ 3	Not used at present.
	IRQ 4	RS-232-C serial port.
	IRQ 5	Not used at present.
	IRQ 6	Diskette DRV status.
	IRQ 7	Parallel PRT port (not used in BIOS).

The interrupt request is processed by a six-step procedure:

1. The peripheral device activates an interrupt request on the system bus, IRQ0...IRQ7.
2. The 8259A interrupt controller prioritizes this request with others which may be coming in or pending.
3. If the request is the only one, or is the next highest level pending at the end of a higher-level service, an interrupt request is sent to the 8088 processor.
4. The 8088 next sends to INTA response pulses to the 8259A interrupt controller. The first freezes the priority and sets the levels in the service latch, The second INTA requests an eight-bit pointer value.
5. The 8259A then sends the pointer value to the 8088. It is used to index into a low memory table, which contains the IP (offset) and the code segment values of the interrupt-service routine for the specific level that is being serviced.
6. Next, the 8088 microprocessor fetches the IP and code segment value and pushes its present IP, code segment and flags onto the system stack and then branches to the interrupt service routine. The interrupt service routine now begins execution.

The pointer values which the 8259A sends out on the second INTA bus cycle are programmed by the system when it boots up. In the IBM PC, this “interrupt controller vector table” starts at 20 hex. Each pointer takes two 16-bit words. Because memory locations 0 to 1F are used for other system pointer values, the first hardware interrupt pointer is the eighth in the memory. Thus the highest priority hardware interrupt, from the system timer counter, is called a “Type 8” interrupt. The complete interrupt vector table is shown below.

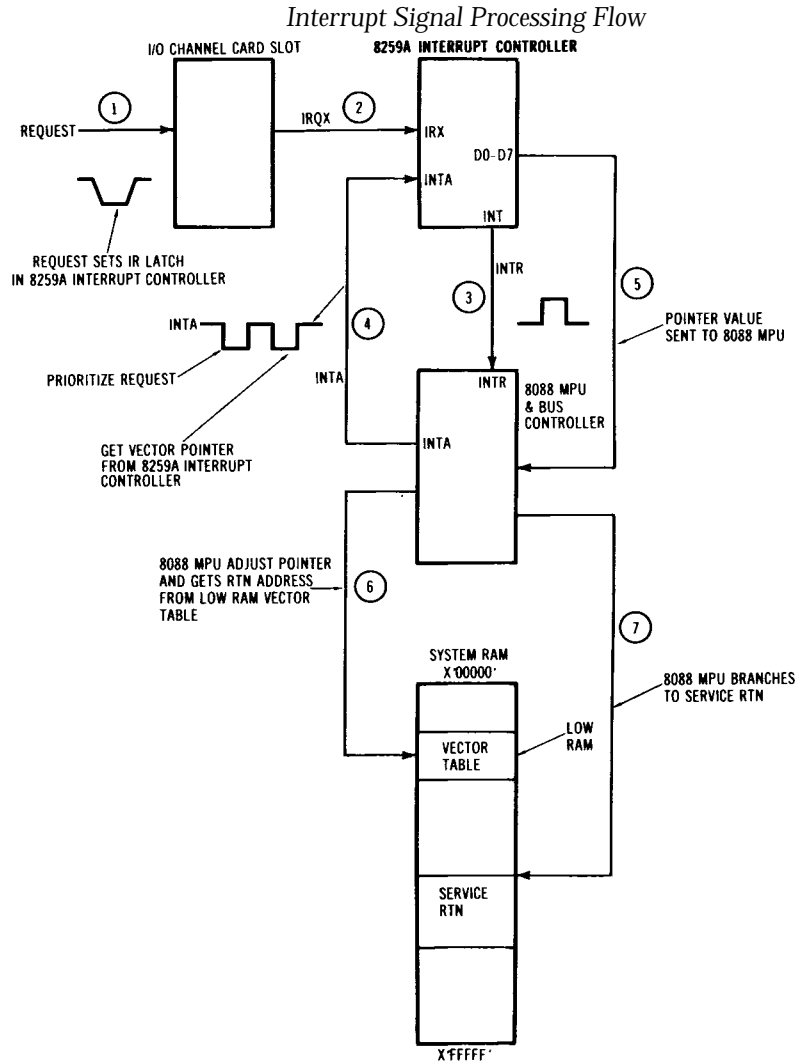
The main part of a user’s program that is to use hardware interrupts must first set the pointer values of the interrupt service routine for the interrupt levels that are going to be needed. The following Pascal code illustrates a procedure for doing this:

```

Procedure Set-IVT(entry : integer)

var
  offset, segment, first_word, second_word : integer;

begin
  offset := Ofs(Interrupt_Service_Routine) + 7
  segment := Cseg;
  first_word := (entry + 8)*4;
  second_word := first_word + 2;
  memW[$0000:first_word] := offset;
  memW[$0000:second_word] := segment;
end;
```



*The Interrupt Vector Table used in the IBM PC*

#### HEX ADDRESS

X'0003F'	IRQ7	TYPE 15 PARALLEL PRINTER PORT CARD
X'0003C'	IRQ6	TYPE 14 DISKETTE ADAPTER CARD
X'00038'	IRQ5	TYPE 13 NOT USED
X'00034'	IRQ4	TYPE 12 SERIAL PORT CARD
X'00030'	IRQ3	TYPE 11 NOT USED
X'0002C'	IRQ2	TYPE 10 NOT USED
X'00028'	IRQ1	TYPE 9 KEYBOARD
X'00024'	IRQ0	TYPE 8 TIMER COUNTER CHANNEL 0
X'00020'		

NOTE: TO GET ACTUAL TABLE VALUE, USE 'DEBUG' AND DISPLAY THIS AREA OF MEMORY

The Interrupt Service Routine, which in this case is called “Interrupt\_Service\_Routine,” must be compiled first, in a simple one-pass compiler, if the `Ofs(Interrupt_Service_Routine)` statement is to return the correct offset address. Seven bytes are added to the offset to account for overhead of the compiler when using Turbo Pascal.

Another initialization step is enabling the interrupt, the procedure `Enable_IRQx` enables the 8259 to pass on an interrupt of a specific level, IRQ, to the 8088’s INTR pin if no other higher interrupt is pending.

```

Procedure Enable_IRQx(IRQ : byte );

var
  imr, mask : byte

begin
  mask := not (1 shl IRQ);
  imr := port($21);      { get Interrupt Mask Register from 8259
  }
  imr := imr and mask;    { clear mask bit
  }
  port[$21] := imr;      { and return to controller
  }
end;
```

The main Interrupt Service Routine will be called everytime there is an interrupt. It must first save all the microprocessor’s registers because the routine being interrupted will want to continue unaltered after the ISR has finished. The easiest way to save them is to PUSH each one on the system stack. Then at the end of the program, they can be restored by POPing them in the exact reverse order. One other option, is to enable further interrupts of equal or higher priority while your ISR is operating. Interrupts are usually masked inside the ISR unless an STI (SeT Interrupts) instruction is given. This is useful if you want, for example, the keyboard to operate normally. But if you’re unsure of how further interrupts will affect you routine, then it’s usually safer to leave out STI at least until version beta 0 is working.

At then end of the ISR you must restore the registers, send and End of Interrupt (EOI) to the 8259 interrupt controller, and return using the special IRET (Interrupt RETurn) instruction. The following shell illustrates an ISR.

```

Procedure Interrupt_Service_Routine;

begin
  inline(  $FB/      { STI      enable further interrupts      }
           $1E/      { PUSH DS                                }
           $50/      { PUSH AX                                }
           $53/      { PUSH BX                                }
           $51/      { PUSH CX                                }
           $52/      { PUSH DX                                }
           $57/      { PUSH DI                                }
           $56/      { PUSH SI                                }
           $06);      { PUSH ES                                }

  INLINE(  $8C/$C8/   { MOV AX,CS  restore data segment      }
           $8E/$D8/   { MOV DS,AX                                }
           $A1/dsave/ { MOV AX,dsave                          }
           $8E/$D8);  { MOV DS,AX                                }

  .
  .
  { much nice Pascal Code for Servicing the Interrupt      }

  .
```

```

      .
port[$0020] := $20    { nonspecific EOI to 8259 PIC }
inline(  $07/         { POP ES }
        $5E/         { POP SI }
        $5F/         { POP DI }
        $5A/         { POP DX }
        $59/         { POP CX }
        $5B/         { POP BX }
        $58/         { POP AX }
        $1F/         { POP DS }
        $CF);        { IRET }
end

```

In Turbo pascal, the machine language instructions for PUSHing and POPing the registers, etc, can be inserted using the "inline" instruction.

Another piece of housekeeping is to insure that the Data Segment Register is correct. When the interrupt occurs, the CPU may be processing routines in the BIOS or anywhere and the Data Segment Register may be incorrect for the ISR. The ISR must restore this value from the variable dsave, which has been carefully initialized at the beginning of the user program. Note that dsave is given an absolute address in the Code Segment which is known to be free in Turbo Pascal.

```

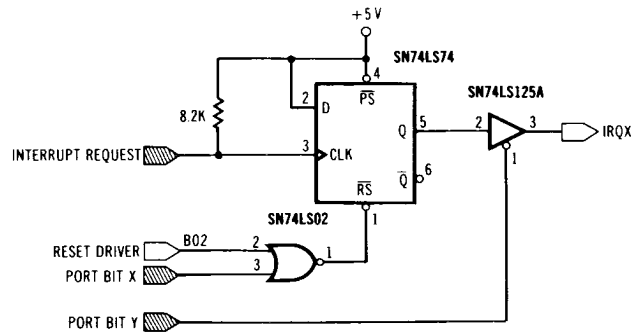
Program Test(input,output)

var
  dsave : integer absolute Cseg:$0006;      { $0006 is ok to use }
  .
  .
  { Procedure definitions etc. }
  .
  .
begin
  dsave := Dseg;      { saves the data segment register }
  .
  .
end.

```

### Interrupt Circuit

The circuit shown below can be used to drive the I/O interrupt request line. A low-to-high transition from a device sets the 7474 flip-flop latch. Ther interrupt lines are *not* open collector. The output of this latch is buffered by the 74125 tri-state buffer whose output goes directly to the system bus. The clear on the latch and the tristate buffer enable are controlled by bits on an output port. The latch holds the request active so that is available for the second INTA pulse from the 8088 processor. This latch can then be reset by an OUT instruction from the interrupt service routine. The programmable I/O port bit on the D-latch "clear" can also be used to inhibit an interrupt request from occurring without using the mask register in the 8259A interrupt controller. The tri-state buffer and its I/O port bit allow the source to be enabled or disabled from the bus interface. Then the interrupt request line on the bus can be used for other purposes.



*Interrupt Request Circuit*

### References

- Eggebrecht, Interfacing to the IBM Personal Computer, Chapter 9  
 Sargent and Shoemaker, The IBM PC from the Inside Out, Chapter 7  
 “IBM PC Interrupt Service Routines,” Paul M Dunphy, BYTE, Fall 1985 (special IBM PC issue),  
 p223  
 “What Is an Interrupt?” R.T. Atkins, BYTE, March 1979, p 230

## Lecture 10

**Direct Memory Access**

Sometimes you have to get data from a device to memory or vice-versa in a hurry. For example, digitized audio signals or data from a mass storage device may require massive transfers of data which would take a long time using the microprocessor to address the port and transfer data to and from memory using its internal registers. The fastest transfer rate under control of the 8088 microprocessor is obtained by assembly language. Transferring one byte of data takes more than 26 clock cycles per byte under program control. The following assembly language code will transfer data at 11.5  $\mu$ s per byte in the 4.77 MHz PC.

```

START:  MOV DX,PORT      ; Load DX register with port address
        MOV BX,BUFFER    ; Load BX register with buffer address
        MOV CX,COUNT     ; Load CX register with loop count
LOOP:   MOV AL,[BX]      ; Load AL register with data from buffer
        OUTB DX          ; Write AL register byte to port
        INC BX           ; Increment buffer address
        DEC CX           ; Decrement loop count
        JNZ LOOP        ; Loop if count is not equal to zero
        .
        .

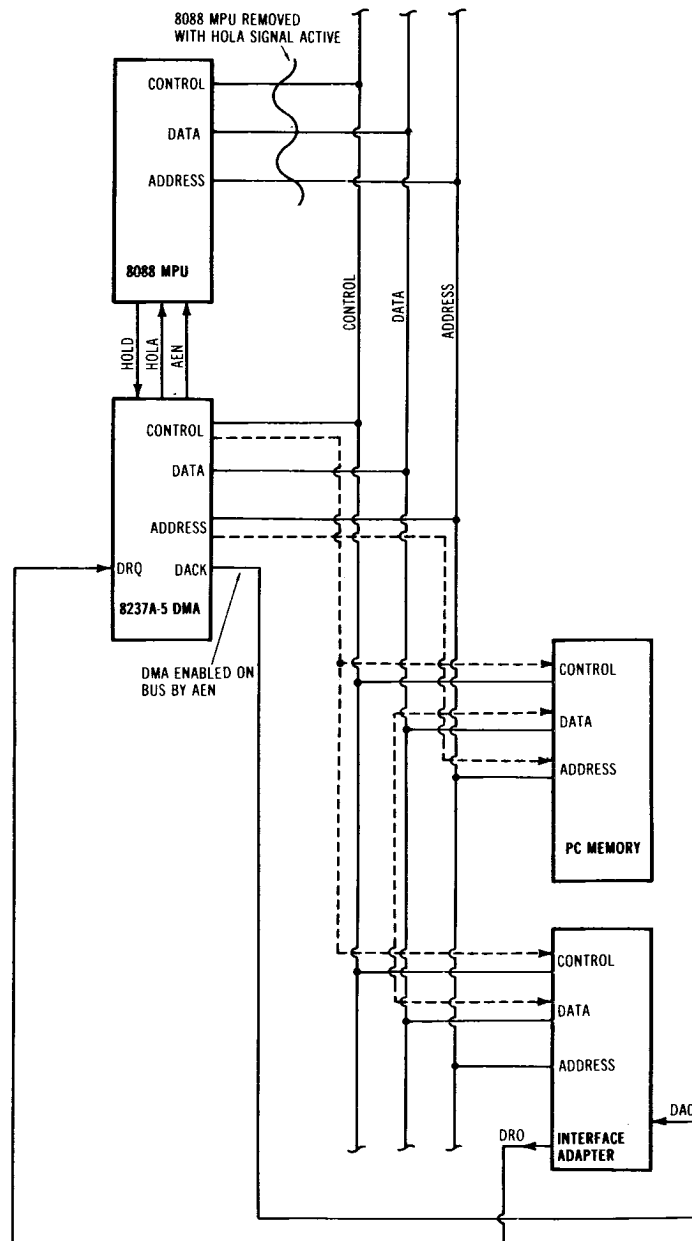
```

Direct memory access (DMA) can transfer one byte in five clock cycles—more than five times faster. In the special block transfer mode, the DMA can transfer each byte in only two clock cycles. During the DMA process the 8088 microprocessor is replaced with another special-purpose chip, the 8237 DMA controller chip. This is essentially a microprocessor that is optimized for transferring data. It completely takes over the data, address and control lines from the 8088 during the transfer. The relationship among the 8088 MPU, the 8237 DMA controller, PC memory, I/O interface and the bus lines is shown in the next figure.

The 8237 has four channels for four different I/O devices, but on the IBM PC design, one of them is used to refresh the memory and only three channels are accessible from the bus. There are three Direct memory access ReQuest lines on the bus: DRQ1, DRQ2 and DRQ3. These are inputs to the 8237 which the peripheral device must activate when it needs to use DMA. The Direct Memory Access follows the following steps:

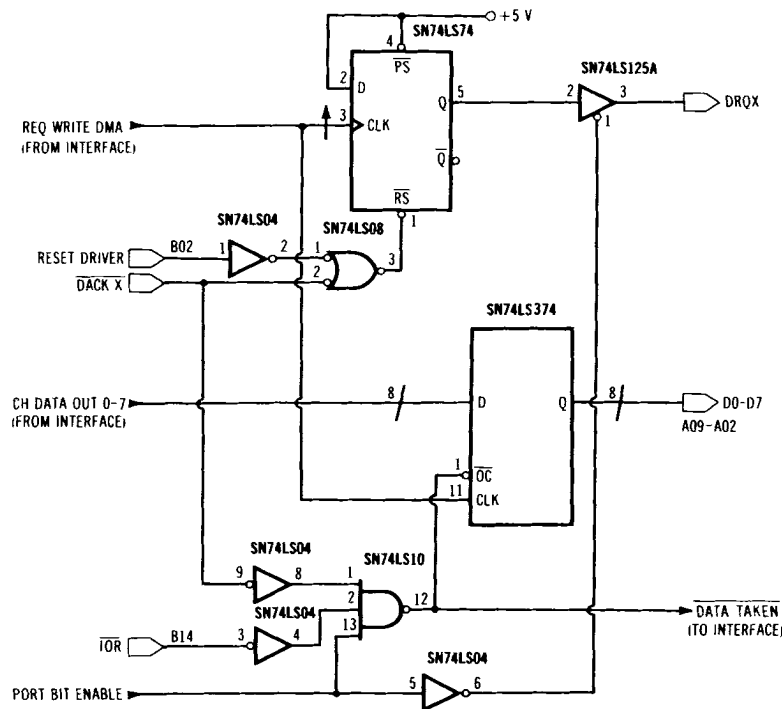
1. A device activates one of the DRQ<sub>n</sub> lines.
2. Once the 8237 gets a DRQ, it will send out a HRQ (Hold ReQuest) to the 8088 wait state generator causing the MPU to wait and detach itself from the bus.
3. When the bus is free, the 8088 sends a HOLDA (HOLD Acknowledge) back to the 8237 DMA controller.
4. The 8237 lowers the  $\overline{\text{DACK}}_n$  line corresponding to the DRQ<sub>n</sub> that it received in step 2. This activates the chip select of the I/O device and allows its data onto the system bus. The interface device drops its DRQ<sub>n</sub> line.
5. The 8237 puts the correct memory address onto the bus and activates the corresponding control lines. If the transfer is from memory to the device  $\overline{\text{MEMR}}$  and  $\overline{\text{IOW}}$  are active, *i.e.*, low. Similarly, if the transfer is from the I/O device to memory,  $\overline{\text{MEMW}}$  and  $\overline{\text{IOR}}$  are active.
6. The HRQ and HOLDA signals also revert to their inactive states.

This process is repeated each time the DRQ<sub>n</sub> line makes a positive transition until the 8237's terminal count has been reached. The terminal count is the total number of bytes to be transferred and is sent out to the 8237 by the user program.

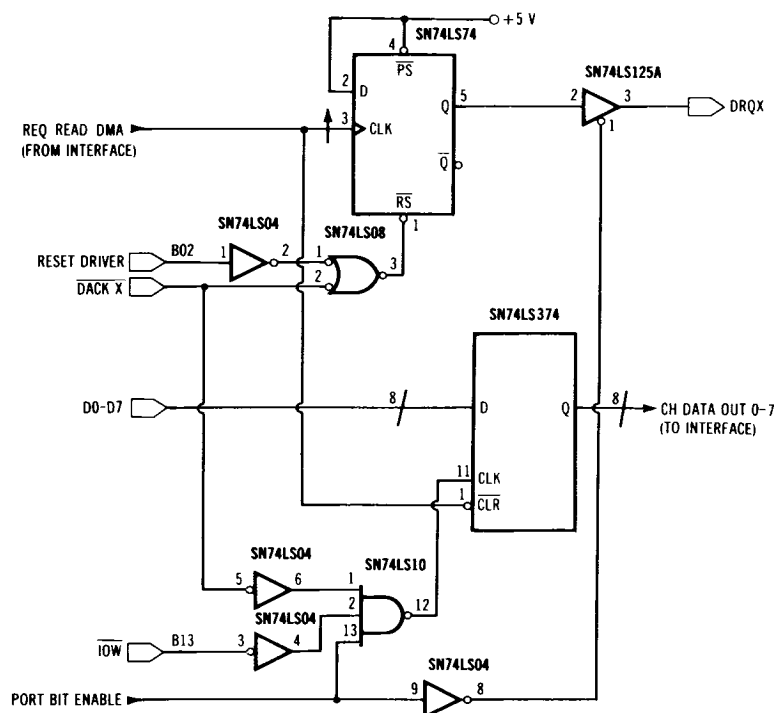
*Data Flow During Direct Memory Access*

The interface circuit shown next can be used for transfers from the device to the system's memory. The D-type flip-flop latches a DRQ onto the bus when a write request is received. In this design, an output port is used to mask the DRQ signal from the peripheral. This mask allows one DRQ line to be shared by several devices. When the  $\neg$ DACK reply is received from the DMA controller it resets this latch. The device mask bit,  $\neg$ DACK, and  $\neg$ IOR all have to be active in order for the data to be transmitted to the data bus. Because the controller has set the address lines for the memory, these data are immediately transferred to memory when they hit the bus.



*DMA Circuit for Transfer from a Device to Memory*

For transfer from memory to a device, a very similar circuit is used. The main difference between this circuit and the previous device-to-memory circuit is that the tri-state buffer (74374) is directed from the data lines to the interface device and is selected by  $\text{-IOW}$  instead of  $\text{-IOR}$ .

*DMA Circuit for Transfer from Memory to an Output Device*

You will notice that because –IOW and –IOR are used in the DMA operation as well as the 8088 port I/O instructions, conventional I/O ports must monitor AEN in order to determine if the read or write is coming from the microprocessor's IN or OUT instruction, or from a direct memory access.

### DMA, the Software Side

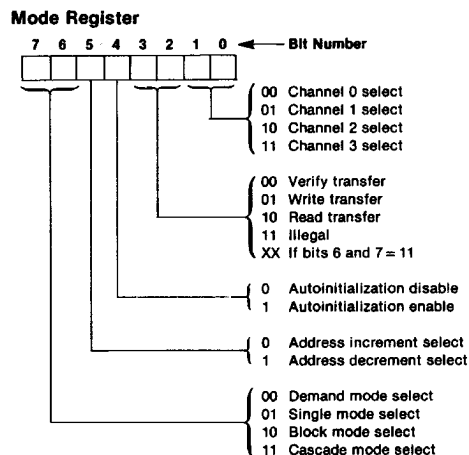
The 8237 has eight control registers which can be written to by the microprocessor when direct memory access is not in progress. There are also two registers which can be read to obtain status information. The following table shows the addresses these occupy on the IBM PC.

*8237 DMA Controller Control and Status Register Ports*

I/O Port	Read Function	Write Function
0008	Status register	Command register
0009	not used	Request register
000A	not used	Single-mask bit register
000B	not used	Mode Register
000C	not used	Clear byte flip-flop
000D	Temporary Register	Master clear
000E	not used	Clear mask register
000F	not used	Write all mask registers

Many of these registers are preprogrammed by BIOS at boot-up time and you don't have to worry about them in your own program. The mode register is important. Bits 3,2 of the mode register specify read (10), or write (01) transfers and bits 7,6 specify the transfer mode: Demand=00, Single=01, Block=10 or Cascade=11. Usually only single mode transfers are used in the PC design because the others may interfere with the memory refresh which is done through DMA channel 0. Bit 5 determines whether the data are to be loaded into successively higher or lower addresses. Bits 1,0 choose the DMA channel.

### Mode Register Bit Definitions



In addition to the control and status registers, each DMA channel has four sixteen-bit registers: the current memory address and count, and base address and base count. At the start of a DMA transfer, the current memory address has the same value as the base register and the current count register is the same as the base count. During the transfer the memory address is either incremented or decremented and the count is decremented. The corresponding base registers maintain the initial values of the memory address and count so they can be reloaded at completion. The automatic reload occurs if bit 5 of the Mode register is 1. The addresses of the current and base registers are given below.

*8237 DMA Controller Address and Count Register Ports*

<u>Address</u>	<u>Channel</u>	<u>Read Register</u>	<u>Write Register</u>
0000	0	Base Address	Current Address
0001	0	Base Count	Current Count
0002	1	Base Address	Current Address
0003	1	Base Count	Current Count
0004	2	Base Address	Current Address
0005	2	Base Count	Current Count
0006	3	Base Address	Current Address
0007	3	Base Count	Current Count

The 8237 really only has a 64 k address space. To allow it to address 1 megabyte of memory, the PC maintains a DMA page register which holds the high four bits of the address. A 74670 is used for a 4-by-4 register for each of the four DMA channels. The four bits are appended externally to the address the 8237 sends out to provide a 20-bit address. The DMA controller really doesn't know there are any more than 64 k in the computer's memory. Therefore transfers cannot be done continuously across a 64 k page boundary. The page registers' I/O port addresses are as follows:

*DMA Page Register I/O Ports*

<u>Channel</u>	<u>Address</u>
1	83h
2	81h
3	82h

The 8237 has sixteen-bit registers but only eight bits are transferred at a time. There is a byte-pointer flip-flop which indicates whether the next read or write goes to the high or low byte of the register. Writing to port 000Ch clears this flip-flop so it points to the low-order byte for the next transfer. It doesn't matter what data are written; any write operation clears the flip-flop.

Carefully reading the Intel data sheet for the 8237 may clear up some confusion about this DMA controller. Its practical use doesn't require knowing everything about it. Here is a simple assembly language program which shows how to transfer data to a 12-bit D/A converter directly from memory. Such a process might be needed for sound synthesis. Here DACK3 has been connected to the DAC's chip select. A clock is wired up to DREQ3 so that a new byte is requested at every clock tick. A table of 16-bit values that define the waveform is stored in the memory area called WAVE in the program. The program below sets up DMA channel 3 to put out the contents of WAVE to the D/A converter.

```

DMA      EQU 0                ; 0 is the DMA port origin on the PC
DMAPAGE  EQU 80H              ; the DMA page register is at 80 hex
DWAVCNT  EQU 2000H            ; there 4096 words in the waveform

DMAWAV:  MOV AL,5BH           ; set DMA channel 3 to single mode,
read                                           ;
        OUT DMA+11,AL         ; and autoinitialize
        OUT DMA+12,AL         ; reset first/last flip-flop
        MOV AX,CX             ; calculate high order 4 bits of buffer
        MOV CL,4              ; put 4 in lower byte of C register
        ROL AX,CL             ; rotate AX left by four bits
        OUT DMAPAGE+2,AL      ; set up the 64k DMA page for channel 3
        AND AL,0F0H
        ADD AX,OFFSET WAVE    ; get page offset
        OUT DMA+6,AL          ; output the start addr of the waveform
        MOV AL,AH
        OUT DMA+6,AL
        MOV AX,DWAVCNT
        MOV WAVCNT,AX         ; output DMA byte count

```

```
        OUT DMA+7,AL
        MOV AL,AH
        OUT DMA+7,AL
        MOV AL,3
        OUT DMA+10,AL      ; unmask DMA channel 3 (let 'er rip)
        RET

WAVCNT  DW  DWAVCNT        ; waveform byte count
BFRPTR  DW  0              ; waveform buffer pointer
        EVEN              ; start at an even address
WAVE    DW  ?              ; values for waveform stored starting
here
```

**References**

Eggebrecht, Interfacing to the IBM Personal Computer, pp188-191 and Ch 10.  
Sargent and Shoemaker, The IBM PC from the Inside Out, pp242-247.

## Lecture 11

**The IEEE-488 Instrumentation Bus**

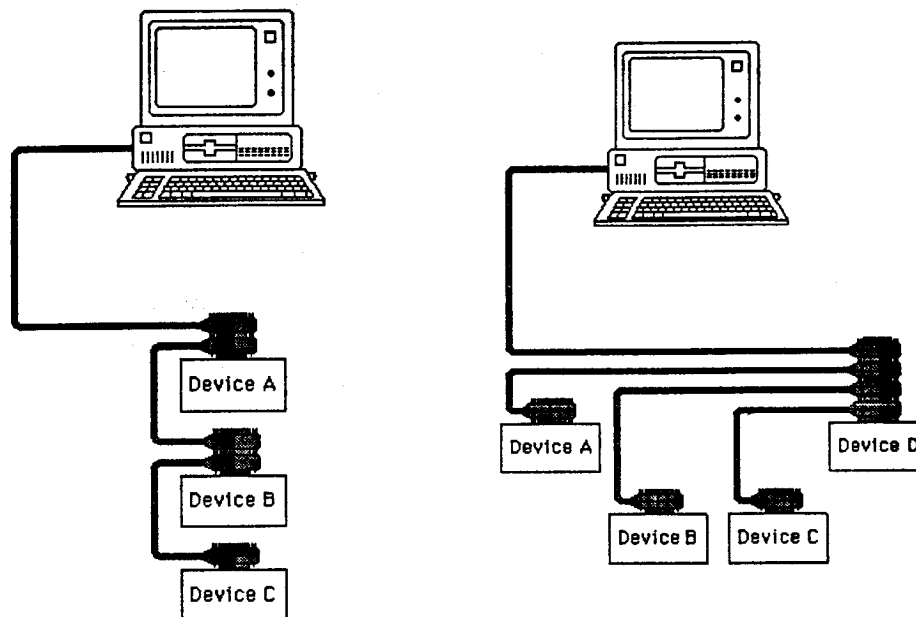
The IBM PC bus has several disadvantages for general-purpose interfacing to laboratory equipment. The most obvious problem is that the interface must be completely redone if another type of computer is to be used. Also, it is difficult to extend this bus more than a meter from the computer, and even then, the lines must be well buffered.

Hewlett-Packard developed a bus for laboratory instruments in the 1960's which could be generally used no matter what computer system was controlling it. Originally this bus was called the HP-IB (Hewlett-Packard Interface Bus). It proved to be such a hit that it was developed into a standard called IEEE-488. Other common names for it are GPIB (General Purpose Interface Bus) or IEC Standard 625-1. With such a standard bus, any computer system can be connected to a network of instruments. All that is necessary is an interface between the specific computer's bus and the IEEE-488 bus. Some of the features of the 488 system are as follows.

- Up to 15 instruments or instrument clusters
- 20 m maximum bus length, 4 m maximum device separation, 2 m average device separation
- 1 Mbyte/s data transmission rate, up to 4 Mbyte/s in synchronous mode
- Open-collector signal lines
- TTL negative logic: True is  $\leq 0.8$  V, False is  $\geq 2.0$  V
- HP patented three-wire data handshake
- 16 signal lines (8 data, 3 handshake and 5 control), 7 ground lines, and 1 shield.
- Stackable connectors for either linear or star configurations

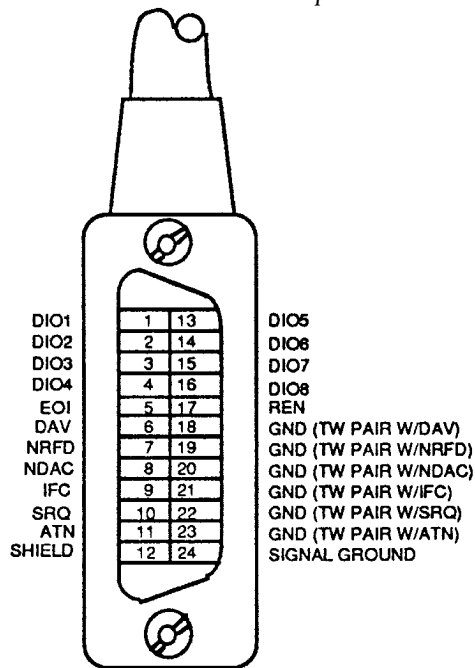
The stackable connectors together with the open collector signal lines allow various configurations for connecting several instruments together. Two configurations, the linear (or daisy chain) and star are illustrated below.

*The Linear Configuration and the Star Configurations for the IEEE-488 Bus*



The signals of the IEEE-488 24-pin connector are shown in the following figure. Note that pin 1 is on the cable side of the connector. One caveat is that the IEC-625 standard is slightly different. Although the same signals are present only the data lines are on the same pins and the IEC connector has 25 pins. Siemens makes an adapter between IEEE-488 and IEC-625.

The IEEE-488 24-pin Connector



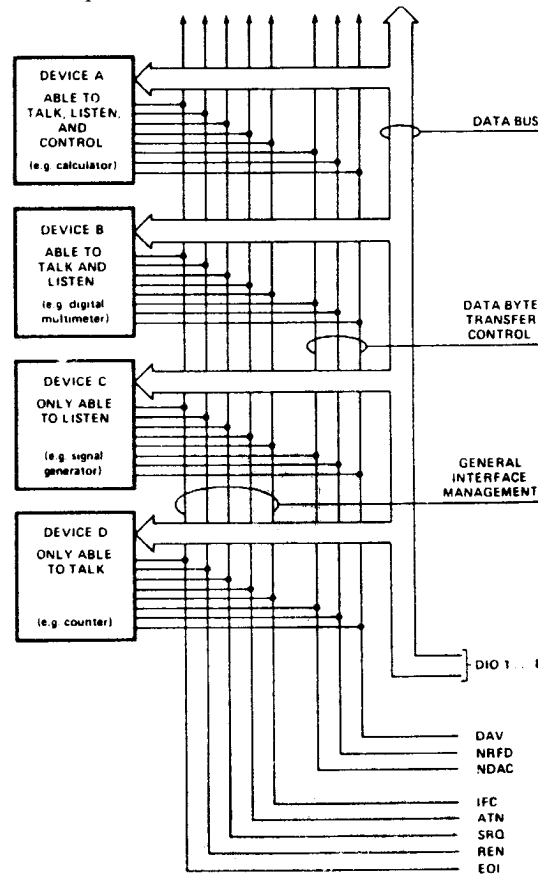
It is a peculiarity of the labelling that the lowest data line is labelled DIO1 instead of DIO0 and the highest DIO8 instead of DIO7. Furthermore, even though all signals are negative logic, no indication is given in the mnemonics, *e.g.*, DAV would be more conventionally labelled  $\overline{\text{DAV}}$  *etc.*

### The Talker, The Listeners and the Controller

The instruments connected to a 488 bus are classed as talkers, listeners or controllers. At any one time there may be only one controller and one talker on the bus, but there may be any number of listeners. A controller can simultaneously be a talker (controller/talker) or a listener (controller/listener). Any particular device could potentially be programmable as either a talker or listener, but it may not be a talker if there is another talker already designated. Similarly, if two devices are on the bus which could be controllers, only one can act as a controller. A digital multimeter, for example, can be a talker when it is sending out measurements, but it can also listen when it is receiving its setup information such as function (Voltmeter/Ammeter/Ohmmeter) and scale.

Each device is identified by a number between 0 and 30 and these numbers should be unique. Usually an instrument which is IEEE-488 compatible has some switches on it for setting up its device number. The diagram below shows four instruments and their connections to the bus.

Example of Instruments Connected to an IEEE-488 Bus



### Choosing Talkers and Listeners

The first task of the controller upon power-up is to designate which device is a talker, and which are listeners. The controller does this by sending control bytes down the data bus. The fact that the bus is carrying control information and not data signified by the ATN line being active, *i.e.*, grounded. The instruments are designed to be on the look-out for control information as long as ATN is low. Each byte of control information contains bits indicating for which device it is destined. The control information only tells the device whether it should be a talker, listener or inactive, it doesn't set up the specific settings for its operation such as voltage, time or current scale settings. These settings are conferred latter as data while the device is a listener.

The use of the bits in the control byte is illustrated in the table below.

Information type	DIO 8	7	6	5	4	3	2	1
Bus Command	x	0	0	C	C	C	C	C
Listen Address	x	0	1	L	L	L	L	L
Talk Address	x	1	0	T	T	T	T	T
Secondary Address	x	1	1	S	S	S	S	S

Notes: x = don't care

CCCCC = Bus Command

LLLLLL = Listener Address

TTTTTT = Talker Address

SSSSSS = Secondary Address

The highest order bit is not used for control commands. When bits DIO7,DIO6 = 01, then low order bits contain the address of a device which is to be a listener. If DIO7,DIO6 = 10, the low five bits

designate a talker. To guarantee only one talker on the bus, any talker ceases to be a talker if it sees another device being chosen as talker.

Provision is made for a group of devices to be attached to a single ID number through the secondary device designation when DIO7,DIO6 = 11. Thus subunits within an instrument cluster may be addressed.

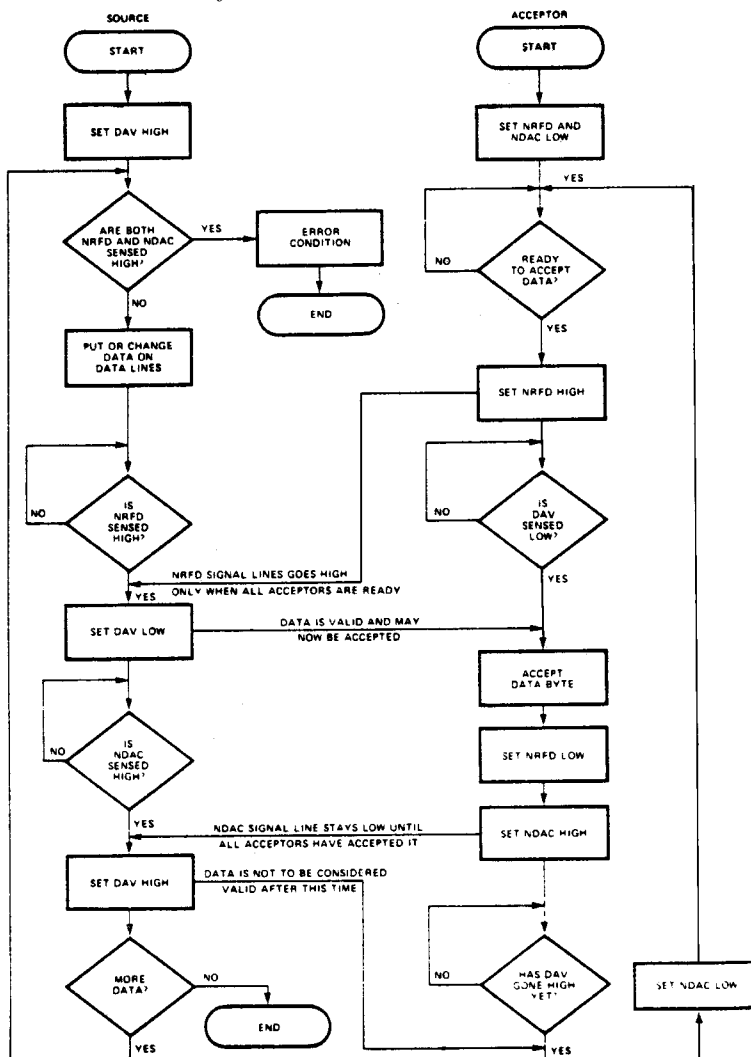
Various bus commands such as “Trigger Talker” or “Serial Poll” can be sent when DIO7,DIO6 = 00.

The special address 11111 (31 decimal) is reserved to tell all devices to cease being listeners or talkers. That is, x0111111 is the global UNLISTEN command and x1011111 is the UNTALK command.

### The Three Wire Handshake

In order to transfer data reliably to a wide variety of instruments, a robust handshaking protocol has been developed and patented (still?) by Hewlett-Packard. This involves three signal lines DAV, NRFD and NDAC. The DAV (DATA Available) line is controlled by the talker and the NRFD (Not Ready For Data) and NDAC (Not Data ACcepted) lines are controlled by the listeners. The handshake for either control bytes or data proceeds as follows:

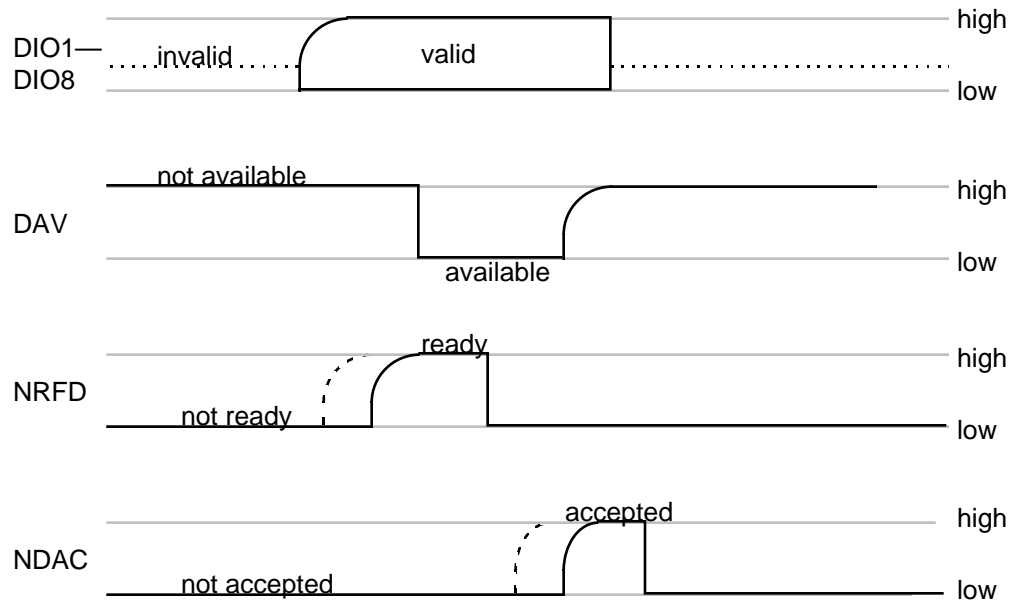
*Flowchart of the IEEE-488 three-wire handshake*





1. Initially DAV is high signifying no data are on the lines. When the talker has data available it first checks that NRFD is high signifying that the that *all* listeners are ready. Because this is an open collector line, it stays low if any listener is Not Ready. After NRFD goes high the talker puts the data on the DIO lines (if it isn't already) and lowers the DAV signal indicating that data are available.
2. The listeners accept the data as each one raises the NDAC line and lowers its NRFD. When all listeners have accepted the data, NDAC goes high.
3. Upon receiving confirmation of data acceptance from the NDAC line the talker raises DAV.
4. Upon sensing DAV high, the listeners pull NDAC low in preparation for more data.
5. The handshaking repeats until all data have been transferred. If the talker has been transferring control information it raises the ATN line again. On the other hand, if it has been transferring data it lowers the EOI (End Of Information) signal.

*Timing Diagram of the Three-wire Handshake*



The transfer is effected by the active talker and listeners. The controller takes no part in the transfer (unless the controller is itself also a talker or listener).

### Other Control Lines

The IFC (Interface Clear) signal can be asserted by the controller to abort all data transfers or other bus activity and put the bus in a known state. It's usually used when something has gone wrong.

The REN (Remote ENable) is sent to a listener to indicate whether it should heed the setup information sent to it by a talker. Otherwise it uses the setup on the front panel controls.

The SRQ (Service ReQuest) allows a device to get the attention of the controller. If and when the controller acknowledges the SRQ it must determine which device sent it out. It can do this by either a *serial poll* or a *parallel poll*.

The *serial poll* is done by sending each device in turn a serial-poll enable command. This is one of the bus control commands the controller can send when ATN is active. When the device gets the serial poll command it replies with 8 bits of status information one of which indicates whether that device was requesting service. The controller then sends a serial-poll disable command and the device reverts to data mode.

The *parallel poll* is faster but cannot get as much information from the devices. A parallel poll is requested by the controller by articulating both ATN and EOI lines. Up to eight devices can respond, each having one bit on the data line. If a device has requested service, it drops its data line after perceiving the ATN and EOI signals from the controller.

### You Didn't Really Have to Know...

Most of the time laboratory instruments will be supplied with IEEE-488 interfaces and an interface card will be purchased for the particular computer in use. An exact knowledge of the protocol is not really necessary for use. What you need to know are the commands to set up the instruments and the format in which these instruments will send the data out while it is a talker. For example, the setup data for an HP 3455A multimeter for “high resolution”, “auto calibration off”, “10-volt range”, and “DC volts function” is “F1R3A0H1.” These codes are particular to each instrument and are not part of the 488 standard. As an illustration the table below shows the setup codes for the various functions of this multimeter

*IEEE-488 Program Codes for an HP 3455A Multimeter*

Function Group	Control	Program Code
Function	DC Volts	F1
	AC Volts	F2
	Fast AC Volts	F3
	2 Wire k $\Omega$	F4
	4 Wire k $\Omega$	F5
	Test	F6
Range	0.1	R1
	1	R2
	10	R3
	100	R4
	1 K	R5
	10 K	R6
	AUTO	R7
Trigger	Internal	T1
	External	T2
	Hold/Manual	T3
Math	Scale	M1
	Error	M2
	Off	M3
Enter	Y	EY
	Z	EZ
Store	Y	SY
	Z	SZ
Auto Cal	Off	A0
	On	A1
High Resolution	Off	H0
	On	H1
Data Ready RQS	Off	D0
	On	D1
Binary Program		B

The syntax used to communicate with instruments have been standardized to a certain extend with the IEEE 488.2 standard. For example the command

\*RST;:Freq 100,10,0;:Measure:Voltage?<NL>

Illustrates the use of the asterisk (\*) for command flag, the semicolon (;) for a compound message separator, the comma (,) for a data separator, the colon (:) for a command separator the question mark (?) for a query and <NL> for the terminator. Further standardization of the commands used to control specific

types of instruments is being attempted with SCPI (Standard Instrument Control Command Set). For example to set a multimeter to DC volts the SCPI command would be "MEAS:VOLTS" for all types of multimeters, whereas before SCPI would use different commands for different brands such as F0 or FUNC DC or MD VOLTS for the same function.

On the computer side, the company that sells you the interface board usually gives or sells a set of subroutines for using the bus. Hewlett Packard's original method of interfacing to existing programming languages was a "Universal Language Interface" (ULI) which consisted of software loaded with the operating system at startup. These allowed any programming language to access the GPIB card through the language's file input or output routines. The ULI software would associate two file names with GPIB input and GPIB output respectively and reads and writes to these files would interact with the GPIB system instead of the file system. This system has the advantage of being applicable to any programming language and being easy to use, but it is rather slow compared to direct subroutine calls. Because of its ease of use we use the ULI system in our Physics 430 GPIB lab.

Most manufacturers of cards supply language interace routines for the most popular programming languages and interpreters. These must be integrated with the language programming system being used and then accessed by calling the subroutines or functions from within the language. Here is an example of the NI-488.2 subroutines from National Instruments as used with the C language.

Call Syntax	Description
AllSpoll (board,addresslist,resultlist)	Serial poll all devices
DevClear (board,address)	Clear a single device
DevClearList (board,addresslist)	Clear multiple devices
EnableLocal (board,addresslist)	Enable operations from the front of a device
EnableRemote (board,addresslist)	Enable remote GPIB programming of devices
FindLstn (board,addresslist,resultlist,limit)	Find all Listeners
FindRQS (board,addresslist,result)	Determine which device is requesting service
GenerateREQF (board, addr)	Cancel service request
GenerateREQT (board, addr)	Request service
GotoMultAddr (board, type,addrfunc,spollfunc)	Enable multiple primary or secondary address support
PassControl (board,address)	Pass control to another device with Controller capability
PPoll (board,result)	Perform a parallel poll
PPollConfig (board,address,dataline,sense)	Configure a device for parallel polls
PPollUnconfig (board,addresslist)	Unconfigure devices for parallel polls
RcvRespMsg(board,data,termination)	Read data bytes from already addressed device
ReadStatusByte (board,address,result)	Serial poll a single device to get its status byte
Receive (board,address,count,termination)	Read data bytes from a GPIB device
ReceiveSetup (board,address)	Prepare a particular device to send data bytes and prepare the GPIB board to read them
ResetSys (board,addresslist)	Initialize a GPIB system on three levels
Send (board,address,data,eotmode)	Send data bytes to a single GPIB device
SendCmds (board,commands,count)	Send GPIB command bytes
SendDataBytes (board,data,count,eotmode)	Send data bytes to already addressed devices
SendIFC (board)	Clear the GPIB interface functions with IFC
SendList (board,addresslist,data,count,eotmode)	Send data bytes to multiple GPIB devices
SendLLO (board)	Send the local lockout message to all devices
SendSetUp (board,addresslist)	Prepare particular devices to receive data bytes
SetRWLS (board,addresslist)	Place particular devices in the Remote with Lockout state
TestSRQ (board,result)	Determine the current state of the SRQ line
TestSys (board,addresslist,resultlist)	Cause devices to conduct self-tests

Trigger (board,address)	Trigger a single device
Triggerlist (board,addresslist)	Trigger multiple devices
WaitSRQ (board,result)	Wait until a device asserts service Request

The hc.c program is used in our lab to download screen images from our Tektronix digital oscilloscopes to the computer. It was written by a former Physics 430 student (Cameron Dale, 1999) for his project. By reading it you can get a flavour of programming using this system. Here is one subroutine from that program which checks that the scope is available at the desired address.

```
void check_scope(void) {
    /* Initialize the GPIB interface for IEEE488.2 communication */
    SendIFC(0);
    if (ibsta & ERR) report_error("Could not send IFC");

    /* Send the identification query to the specified address. */
    Send(0, scope, "**IDN?", 5L, NLEnd);
    if (ibsta & ERR) report_error("Could not find anything at specified address");

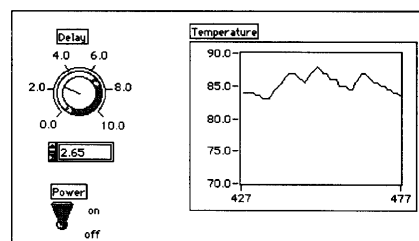
    /* Read the name identification response returned from the device. */
    Receive(0, scope, buffer, 150L, STOPend);
    if (ibsta & ERR) report_error("Could not read anything from specified address");
    for(j=0; (j<ibcntl) && (buffer[j] != '\n'); j++); /* find the end */
    buffer[j] = '\0'; /* terminate the string */

    /* Check to see if this address is that of a scope. */
    if (strncmp(buffer, "TEKTRONIX,TDS", 13) == 0)
        fprintf(stderr, "Found the %s at address %d\n", buffer, scope);
    else report_error("Could not find the Tektronix TDS oscilloscope at specified address");

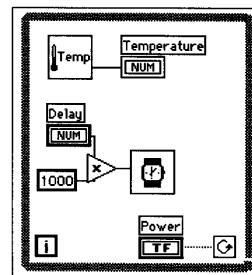
    /* Clear the GPIB interface from IEEE488.2 mode so that IEEE488 commands can be used */
    ibonl(0,0);
}
```

Another technique in common use is a graphical system originally developed by National Instruments for use with the Macintosh and now available for Windows, Unix and Linux systems. This system, called LabView, allows the user to design a using interface containing dials, graphs readouts, buttons, controllers which imitates an actual instrument's front panel. Then these readouts and controllers are programmed using a graphical programming system. The advantages are that it is fairly easy to create a convenient user interface and the system can be reconfigured fairly easily, once the graphical programming system has been mastered. The disadvantages are the cost and the necessity of obtaining drivers for all the devices which one wishes to use. Hewlett Packard also has a similar graphical system available.

#### *LabView used for a Simple Temperature Monitor*



User Interface



Graphical Code

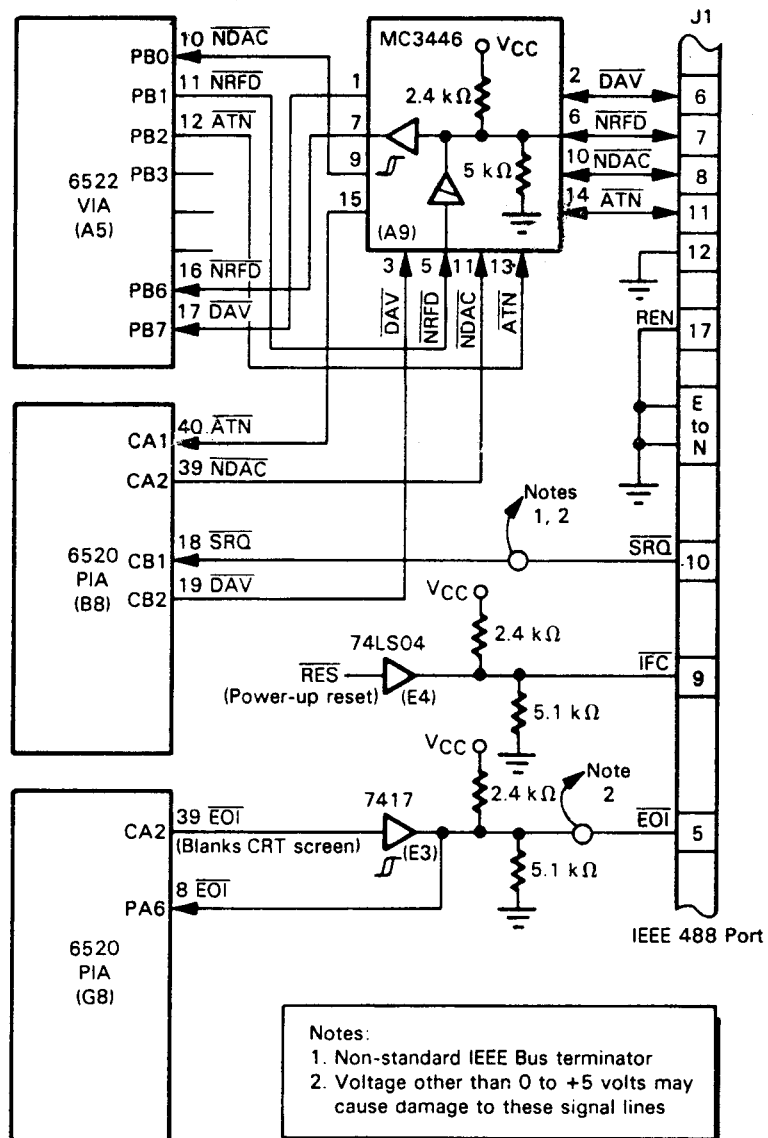
#### **But If You're Really Stuck**

It's useful to know that you can run a 488 bus through the 8255 PPI chip using a couple of smart open-collector buffers 75160 and 75161. The correct protocol can then be supplied in software.

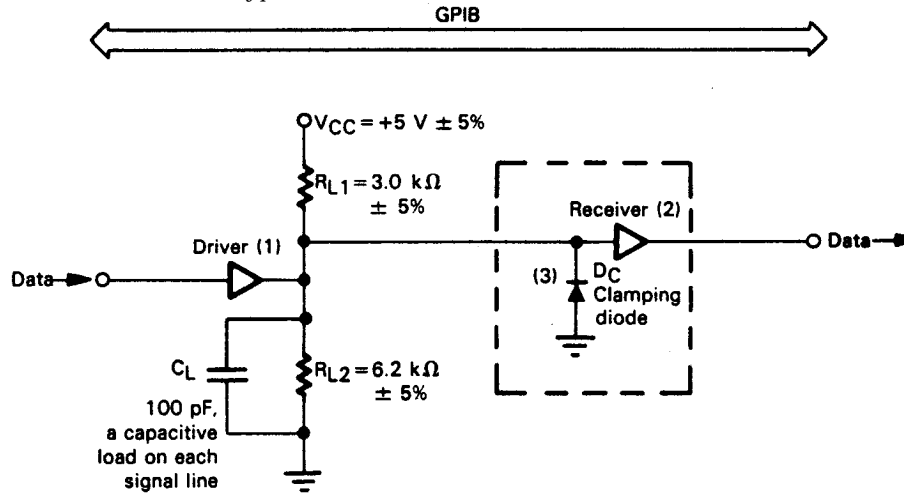
- Use the 75160 bidirectional buffer to buffer the DIO1–DIO8
- Use the 75161 to buffer the control lines
- Feed the data lines through Port A of the 8255
- Monitor EOI and SRQ through Port B in mode 0
- Use a nybble of port C to output ATN, REN, IFC and to enable the bus buffers for listening purposes.

The old Commodore Pet computer used 6520 PIAs and a 6522 VIA along with an MC3446 buffer/terminator chip to implement a 488 bus as shown below.

## IEEE-488 Interface Using 65xx Stuff



*Typical Interface Between the 488 bus and a Device*



**Notes:**

1. Driver output current leakage:  
 Open collector: +0.25 mA max where  $V_O = +5.25\text{ V}$   
 Tristate:  $\pm 40\text{ }\mu\text{A}$  max where  $V_O = +2.4\text{ V}$
2. Receiver input current:  
 1.6 mA max at  $V_O = 0.4\text{ V}$
3. Typically, clamping diode is located within the receiver component.

**References**

- Sargent and Shoemaker, *The IBM PC from the Inside Out*, pp 389–395
- S. Leibson, "The Input/Output Primer, Part 3: The Parallel and GPIB (IEEE 488) Interfaces,"  
 BYTE April 1982, p186
- E. Fisher and C.W. Jensen, *PET and the IEEE 488 Interface*, Osborn/Mcgraw Hill (1980).
- H.S. Stone, *Microcomputer Interfacing*, pp 215–227

**Appendix: Listing of hc.c**

```

/* Program for reading hardcopy output from oscilloscopes */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include "decl.h"

/* Function prototypes */
void report_error(char *errmsg);
void initialize_inputs(int argc, char** argv);
void initialize_output(void);
void check_scope(void);
void find_scope(void);
void setup_scope(void);
void hardcopy(void);
void terminate(void);

char          filename[13];      /* The name of the output file          */
FILE*         ofp;              /* The output file pointer             */
int           usage,            /* Binary: to display the proper usage */
             extension,        /* Binary: to force the extension      */
             layout,          /* Binary: output in portrait (1) or landscape (0) */
/*
             screen;          /* Binary: to print to screen (stdout) */
char          format;           /* The format of the output           */
char          buffer[1001];     /* Data received queries              */
int           i, j, n,          /* Loop counter variables              */
             num_listeners;     /* Number of listeners on GPIB        */
unsigned long bytes;            /* The number of bytes transferred    */
unsigned int  instruments[32],  /* Array of primary addresses          */
             result[31],       /* Array of listen addresses          */
             ascope[31],       /* Array of oscilloscope addresses    */
             scope;            /* The oscilloscope address to use    */

/* Prints errors to standard error output and exits the program */
void report_error(char *errmsg) {
    fprintf(stderr, "Error %d,%d: %s\n", ibsta, iberr, errmsg);
    ibonl(0,0);          /* take all GPIB devices offline */
    exit(1);             /* abort the program */
}

void initialize_inputs(int argc, char** argv) {
    /* Set the defaults */
    usage = 0;           /* do not display the usage          */
    extension = 0;       /* don't force the file extension    */
    layout = 1;          /* layout in portrait                */
    format = 'x';        /* output in pcx file format        */
    screen = 1;          /* output to screen                  */
    scope = 0;           /* the GPIB board address (invalid) */
    strcpy(filename, ""); /* no file unless specified          */

    /* Test proper usage of inputs */
    if (argc == 1) usage = 1;
    for (i=1; i<argc; i++) {
        if (argv[i][0] == '-') {
            /* if this is an option          */
            if (strlen(argv[i]) > 2) usage = 1; /* it should have a length of 2 */
            if (isdigit(argv[i][1])) scope = argv[i][1] - '0'; /* read the specified
address */
            else switch(tolower(argv[i][1])) {
                case 'p': layout = 1; break; /* set the layout to portrait
*/

```

```

        case 'l': layout = 0; break;          /* set the layout to landscape
*/
        case 'x': format = 'x'; break;       /* set the format to pcx
*/
        case 'b': format = 'b'; break;       /* set the format to bitmap
*/
        case 'd': format = 'd'; break;       /* set the format to deskjet
printer */
        case 'j': format = 'j'; break;       /* set the format to laserjet
printer */
        case 'e': extension = 1; break;      /* force the file extension
*/
        case 'n': extension = 0; break;      /* don't force the file extension
*/
        default : usage = 1;                 /* else display the proper usage
*/
    }
}
else { /* if not an option, then it should be the filename */
    if ((i == argc-1) && (strlen(argv[i]) < 13)) {
        /* the filename should always come last and be 12 characters or less */
        strcpy(filename, argv[i]);
        screen = 0; /* don't print to screen */
    }
    else usage = 1;
}
}

/* if the input was improperly used, display the proper usage */
if (usage) {
    fprintf(stderr, "Oscilloscope Hardcopy Program, Version 1.1, Written By Cameron
Dale\n");
    fprintf(stderr, "    for use with the IEEE488 GPIB interface\n");
    fprintf(stderr, "\n");
    fprintf(stderr, " Usage: %s [options] [filename]\n", argv[0]);
    fprintf(stderr, "           where filename is an acceptable DOS filename:
????????[.???]\n");
    fprintf(stderr, "           (if one is not specified, the default is to use the
standard output)\n");
    fprintf(stderr, "           and options are some of the choices below (* indicates the
default).\n");
    fprintf(stderr, "\n");
    fprintf(stderr, "    * -p   set layout to portrait\n");
    fprintf(stderr, "    -l   set layout to landscape\n");
    fprintf(stderr, "\n");
    fprintf(stderr, "    * -x   output in pcx format (.pcx)\n");
    fprintf(stderr, "    -b   output in bitmap format (.bmp)\n");
    fprintf(stderr, "    -d   output in deskjet printer format (.dkj)\n");
    fprintf(stderr, "    -j   output in laserjet printer format (.lsj)\n");
    fprintf(stderr, "\n");
    fprintf(stderr, "    * -n   don't force the extension\n");
    fprintf(stderr, "    -e   force the correct extension\n");
    fprintf(stderr, "\n");
    fprintf(stderr, "    * -0   search for connected scopes to output to\n");
    fprintf(stderr, "    -#   force the output to the scope at address # (integer
from 1 to 9)\n");
    exit(0); /* and then exit the program */
}
}

void initialize_output(void) {
    /* Initialize the output file */
    if (extension) { /* if the extension is to be forced */
        /* find the start of the extension */

```



```

    for (i=0; i < strlen(filename); i++) if (filename[i] == '.') break;
    filename[i] = '\\0'; /* end the string there */
    switch(format) { /* append the proper extension */
        case 'x': strcat(filename, ".pcx"); break;
        case 'b': strcat(filename, ".bmp"); break;
        case 'd': strcat(filename, ".dkj"); break;
        case 'j': strcat(filename, ".lsj"); break;
        default : report_error("Invalid format type.");
    }
}
ofp = fopen(filename, "wb"); /* open the file for binary write and assign the
file pointer */
if (ofp == NULL) { /* check to make sure the file was opened */
    fprintf(stderr, "Make sure the filename is a valid DOS name ???????.[.???]\n");
    report_error("Could not open output file.");
}
}

void check_scope(void) {
    /* Initialize the GPIB interface for IEEE488.2 communication */
    SendIFC(0);
    if (ibsta & ERR) report_error("Could not send IFC");

    /* Send the identification query to the specified address. */
    Send(0, scope, "**IDN?", 5L, NLEnd);
    if (ibsta & ERR) report_error("Could not find anything at specified address");

    /* Read the name identification response returned from the device. */
    Receive(0, scope, buffer, 150L, STOPend);
    if (ibsta & ERR) report_error("Could not read anything from specified address");
    for(j=0; (j<ibcntl) && (buffer[j] != '\\n'); j++); /* find the end */
    buffer[j] = '\\0'; /* terminate the string */

    /* Check to see if this address is that of a scope. */
    if (strncmp(buffer, "TEKTRONIX,TDS", 13) == 0)
        fprintf(stderr, "Found the %s at address %d\\n", buffer, scope);
    else report_error("Could not find the Tektronix TDS oscilloscope at specified
address");

    /* Clear the GPIB interface from IEEE488.2 mode so that IEEE488 commands can be
used */
    ibonl(0,0);
}

void find_scope(void) {
    /* Initialize the GPIB interface for IEEE488.2 communication */
    SendIFC(0);
    if (ibsta & ERR) report_error("Could not send IFC");

    /* Create an array containing all valid GPIB primary addresses except 0.
The constant NOADDR, defined in DECL.H, signifies the end of the array. */
    for (i = 1; i <= 30; i++) instruments[i] = i;
    instruments[31] = NOADDR;

    /* Find all of the listeners on the bus. Store the listen addresses in the array
RESULT. */
    fprintf(stderr, "Finding all listeners on the bus...\\n");
    FindLstn(0, instruments, result, 30);
    if (ibsta & ERR) report_error("Could not find listeners");

    /* Loop for each address found except the GPIB interface (0) */
    num_listeners = ibcnt - 1;
    n = 0; /* initialize the number of scopes found to 0 */
    for (i = 1; i <= num_listeners; i++) {

```

```

/* Send the identification query to all addresses in the array RESULT. */
Send(0, result[i], "IDN?", 5L, NLEnd);
if (ibsta & ERR) continue; /* on error go to next device */

/* Read the name identification response returned from each device. */
Receive(0, result[i], buffer, 150L, STOPend);
if (ibsta & ERR) continue; /* on error go to next device */
for(j=0; (j<ibcntl) && (buffer[j] != '\n'); j++); /* find the end */
buffer[j] = '\0'; /* terminate the string */

/* Check to see if this address is that of the scope. */
if (strcmp(buffer, "TEKTRONIX,TDS", 13) == 0) {
    fprintf(stderr, "%d Found the %s at address %d\n", n+1, buffer, result[i]);
    ascope[n] = result[i]; /* save the scope address in the array */
    n = n + 1; /* increment the number of scopes found */
}
}
if (n == 0) report_error("Did not find any Tektronix Oscilloscopes");
if (n == 1) scope = ascope[0]; /* then the scope is the only one found */
else { /* if multiple scopes were found */
    /* ask the user which scope to use */
    do { fprintf(stderr, "Enter the number of the scope you would like to use: "); }
    while ((scanf("%d", &scope) != 1) || (scope > n) || (scope <= 0));
    scope = ascope[scope-1]; /* translate number to scope address */
}

/* Clear the GPIB interface from IEEE488.2 mode so that IEEE488 commands can be
used */
ibonl(0,0);
}

void setup_scope(void) {
    /* Initialize the GPIB device for IEEE488 commands */
    scope = ibdev(0, scope, 0, T10s, 1, 0);
    if (ibsta & ERR) report_error("Could not open oscilloscope.");

    /* Set the device to assert the SRQ line when ready */
    ibwrt(scope, "SRE 16", 7L);
    if (ibsta & ERR) report_error("Could not set SRQ.");

    /* Set the scope's hardcopy port to GPIB */
    ibwrt(scope, "HARDCOPY:PORT GPIB", 17L);
    if (ibsta & ERR) report_error("Could not set hardcopy port.");

    /* Set the scope's output layout */
    if (layout) ibwrt(scope, "HARDCOPY:LAYOUT PORTRAIT", 24L);
    else ibwrt(scope, "HARDCOPY:LAYOUT LANDSCAPE", 25L);
    if (ibsta & ERR) report_error("Could not set hardcopy layout.");

    /* Set the scope's output format type */
    switch(format) {
        case 'x': ibwrt(scope, "HARDCOPY:FORMAT PCX", 19L); break;
        case 'b': ibwrt(scope, "HARDCOPY:FORMAT BMP", 19L); break;
        case 'd': ibwrt(scope, "HARDCOPY:FORMAT DESKJET", 23L); break;
        case 'j': ibwrt(scope, "HARDCOPY:FORMAT LASERJET", 24L); break;
        default : report_error("Invalid format type.");
    }
    if (ibsta & ERR) report_error("Could not set hardcopy format.");
}

void hardcopy(void) {
    /* Start the hardcopy output */
    ibwrt(scope, "HARDCOPY START", 14L);
    if (ibsta & ERR) report_error("Could not start hardcopy.");
}

```

```

/* Wait for the scope to be ready */
ibwait(scope, TIMO | RQS);
if (ibsta & (ERR | TIMO)) report_error("Waiting for SRQ");

bytes = 0;
if (screen) { /* if the output is to the standard output */
    fprintf(stderr, "Beginning output ...\n");

    /* read each character from the device and print it to standard output */
    do {
        /* read 1000 bytes from scope into the character array buffer */
        ibrd(scope, buffer, 1000L);
        if (ibsta & ERR) report_error("Input of bytes failed");

        /* output each byte to the standard output */
        for (i=0; i<ibcntl; i++) putc(buffer[i], stdout);
        bytes = bytes + ibcntl;
    } while (!(ibsta & END)); /* until there are no more to read */
}
else { /* if the output is to a file */
    fprintf(stderr, "Beginning output to file %s ...\n", filename);

    /* ibrdf sends the output from scope to the file specified by filename */
    ibrdf(scope, filename);
    if (ibsta & ERR) report_error("Output to file failed");

    bytes = bytes + ibcntl;
}

/* indicate successful completion */
fprintf(stderr, "Output completed, %lu bytes transferred.\n", bytes);
}

void terminate(void) {
    if (!screen) fclose(ofp); /* close the file */
    ibonl(scope, 0);          /* close the GPIB device connection */
}

void main(int argc, char *argv[]) {

    initialize_inputs(argc, argv);

    /* If the output is not to go to the standard output */
    if (!screen) initialize_output();

    /* If the scope address was specified, check to make sure it's right */
    if (scope) check_scope();
    else find_scope(); /* else find the addresses of the scopes */

    setup_scope();

    hardcopy();

    terminate();
}

```

## Assignment 1

1. Draw two alternative circuits for the three-condition lunch decision box on p. 4 of the lecture notes.
2. Design a one-bit adder with carry input and carry output that can be cascaded to make a multiple-bit adder. (Lecture notes p. 9)
3. Verify the logic identity on p. 9 of the lecture notes using a truth table:

$$(-A)(-B)(-C) + (-A)BC + A(-B)(-C) = -(B+C) + (-A)BC = Q$$

Draw a circuit of logic gates with three inputs, A, B, C and one output, Q, to perform this operation.

4. Draw up a circuit using NANDs and NORs for a four-input function which gives a 1 as output if the sum of the inputs is odd, and gives 0 if the sum is even. This is a parity generator. (Lab 1 assignment question)
5. Using four JK flip-flops and whatever ANDs, ORs, NANDs and NORs you need, design a four-bit up-down counter with both clock and control inputs which counts up when a control input is HIGH and counts down when the control input is LOW. (Lab 1 Assignment question)
6. What addresses do outputs Y1 through Y7 decode in the circuit of Lab 2?

## Assignment 2

1. The three-nand gate circuit used to provide the load signal of the input buffer is sometimes called a “steering network”. Show how one could use steering networks to construct a four-bit up/down counter. When UP is high the counter counts 1-2-3-.... When UP goes low it counts down 3-2-1-0...

2. Extend the synchronous counter in Fig. 2.2 of the lab script to five bits. Draw the circuit diagram and the timing diagrams.

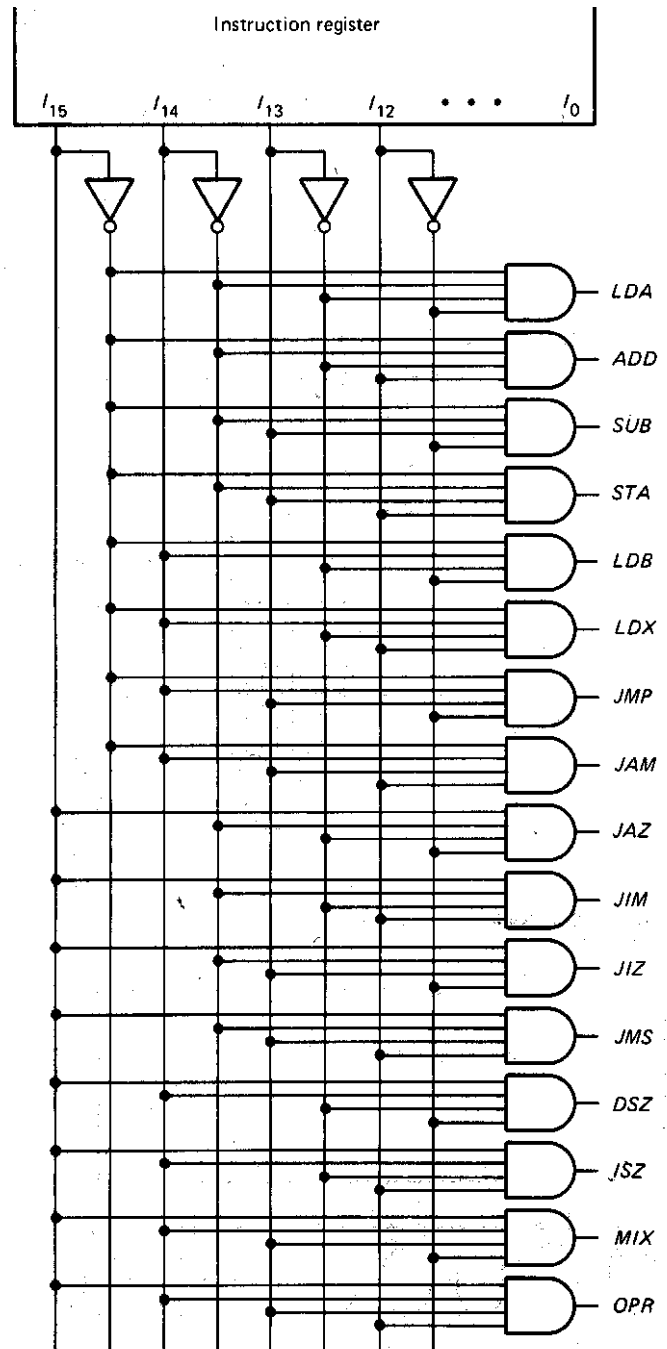
3. The figure shows a 1-of-16 decoder. The signals coming out of the decoder are labelled LDA, ADD, SUB and so on. The word formed by the 4 leftmost register bits is called the “OP CODE.” As an equation  $OPCODE = I_{15}I_{14}I_{13}I_{12}$

a) If OPR is high what does OPCODE equal?

b) If JIM is high what does OPCODE equal?

c) Which output signal is high if OPCODE = 0000?

d) Which output signal is high if OPCODE = 0001?



3. Fig. 3.35 below shows a control matrix. The inputs  $T_1$  through  $T_6$  are timing signals. They are

usually driven by one ring counter and only one is high at a time.  $T_1$  goes high first, then  $T_2$  and so forth. The signals control the rate and sequence of computer instructions. The lower set of inputs, LDA, ADD, SUB and OUT are computer instructions; only one of them is high at a time. The outputs  $C_p$ ,  $E_p$ ,  $L_m$ , ...,  $L_o$  control different registers. Which are the high outputs for each of the following conditions?

- |                       |                       |
|-----------------------|-----------------------|
| a) $T_1$ high         | g) $T_5$ and ADD high |
| b) $T_2$ high         | h) $T_6$ and ADD high |
| c) $T_3$ high         | i) $T_4$ and SUB high |
| d) $T_4$ and LDA high | j) $T_5$ and SUB high |
| e) $T_5$ and LDA high | k) $T_6$ and SUB high |
| f) $T_4$ and ADD high | l) $T_6$ and OUT high |

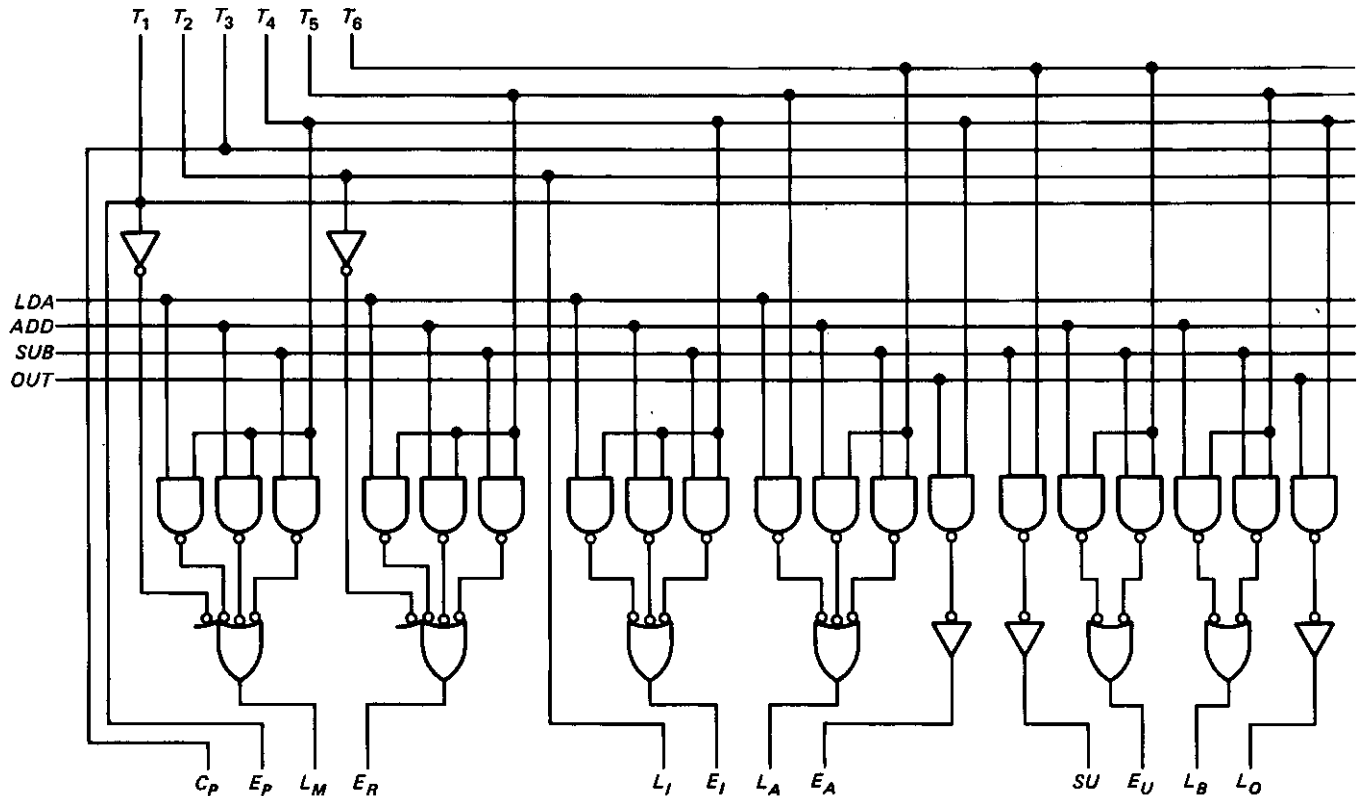


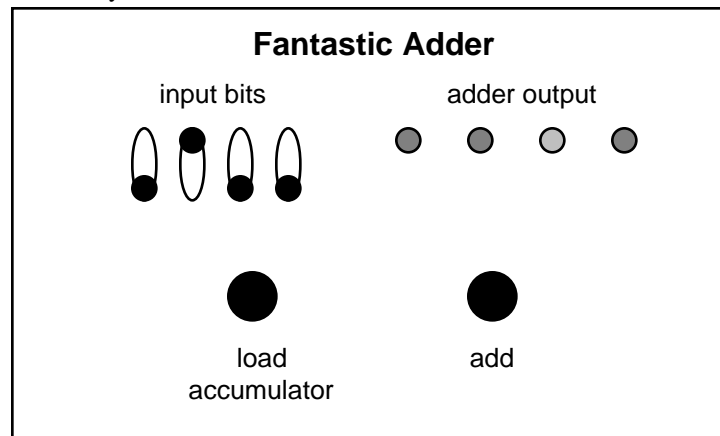
Fig. 3-35 Control matrix.

**Design problems:**

- Include a brief description of the operation.
- Label chips with number and function if it's not obvious from the symbol.
- Provide enough detail that somebody unfamiliar with your thought processes could build it.

**Assignment 3**

1. Design a 4-bit binary adding machine with 4 toggle switches for input and 4 leds for output. The 4 toggle switches are the binary number input. One pushbutton loads the accumulator register with the value set by the switches and the second pushbutton adds the current switch settings to the accumulator. The output of the addition is displayed on the leds. Use bus architecture with the registers connected to the bus by three-state buffers. You may design for either TTL or CMOS but try not to mix families if that's not necessary.



2. You have a device which was designed to interface to a 68008 microprocessor with signals including  $\overline{AS}$ ,  $\overline{DS}$ ,  $R/\overline{W}$  and  $\overline{DTACK}$  as well as the data bits and a chip select input. Describe and draw the circuit needed to allow these signals to interface to the IBM PC bus.

### Assignment 4

1. Design a one-channel I/O port to reproduce some of the features of the 8255 chip in Mode 1. The computer-side signals should include eight data bits, chip select, and I/O select signals. Use a high on the I/O line for input (to CPU) and 0 for output. The peripheral-side signals should include eight data bits. In addition, provide for the appropriate handshaking signals, but don't include the interrupt option. Use chips described in the Lab Manual.

2 a) Use an 8253 and as few extra parts as possible to build a frequency meter. Use one counter, counting the 1 kHz clock, to set up a gate of variable width, controllable from a program. Let the maximum gate width be about 65 seconds. The other two counters should be cascaded to allow a 32-bit count of an external signal during the gate period.

b) Write a program which asks for the gate width in seconds, and then types out the unknown frequency in Hz.