Embedded Controller Hardware Design

Designing *Reliable* Microcontroller Hardware
for Real World Embedded Applications

Ken Arnold

Arnold, Ken
Embedded Controller Design / Ken Arnold
1. Embedded Computers/Controllers. 2. 8051 Microcontroller. 3. Digital Control Systems. 4. Automatic Control. I. Title.
ISBN tba
QA76.8.I27 1999

THIRD PRINTING

Embedded Computer Design
Arnold, Ken

Published by
HiTech Publications
9672-101 Via Excelencia
San Diego, CA 92126
USA

Phone (619) 566-1892
Fax (619) 530-1458
info@hte.com
http://www.hte.com

Contents

Preface

During the early years of microprocessor technology, there were few engineers with education and experience in the application of microprocessor technology. Now that microprocessors and microcontrollers have become pervasive in many types of equipment, it has become almost a requirement that many technical people have the ability to use them. Today the microprocessor and the microcontroller have become two of the most powerful tools available to the scientist and engineer. Microprocessors have been embedded in so many products that it is easy to overlook the fact that they greatly outnumber personal computers. While a great deal of attention is given to personal computers, the vast majority of new designs are for embedded applications. For every PC designer there are thousands of designers using microcontrollers in an embedded application. The number of embedded designs is growing quickly. The purpose of this book is to give the reader the basic design and analysis skills to design reliable microcontroller or microprocessor based systems. The emphasis in this book is on the practical aspects of interfacing the processor to memory and I/O devices, and the basics of interfacing such a device to the outside world.

A major goal of this book is to show how to make devices inherently reliable by design. While a lot of attention has been given to "quality improvement," the majority of the emphasis has been placed on the processes occurring *after* the design of a product is complete. Design deficiencies are a significant problem, and can be exceedingly difficult to identify in the field. These types of quality problems can be addressed in the design phase with relatively little effort, and with far less expense than will be incurred later in the process. Unfortunately there are many hardware designers and organizations that, for various reasons, do not understand the significance and expense of an unreliable design. The design methodology presented in this text is intended to address this problem.

Learning to design and develop a microcontroller system without any practical hands-on experience is a bit like trying to learn to ride a bike from reading book. Thus, another goal is to provide a practical example of a complete working product. What appears easy on paper may prove extremely difficult without some real world experience and some potentially painful crashes. In order to do it right, it's best to examine and use a real design. On the other hand, the current state of the technology (surface mounted packaging, etc.) can make the practical side problematic. In order to address this problem, a special educational System Development Kit is available to accompany this book (8031SDK). All the documentation to construct an SDK is available on the companion CD-ROM. This info, along with updated information and application examples, is available on the web site for this book: http://www.hte.com/echdbook. All information needed to build the SDK is available there, as well as information on how to order the SDK assembled and tested.

While searching for an appropriate text for one of the courses I teach in embedded computer engineering, I was unable to locate a book that covered the topic adequately. An earlier version of this book was written to accompany that course and has since evolved into what you see here. The course is offered at the University of California, San Diego Extended Studies, and is titled "Embedded Controller Hardware Design." The same courses may also be taken on-line using the Internet, and can be found at: www.hte.com/uconline. The goal of the course and the book are very much the same: to describe the *right way* to design embedded systems.

While no prior knowledge of microcontrollers or microprocessors is required, the reader should already be familiar with basic electronics, logic, and basic computer organization. Chapter one is intended as a review of those basic concepts. Next there is a general overview of microcontroller architecture, and a specific microcontroller chip architecture, the 8051 family, is introduced and detailed. The concepts of worst case design and analysis is described, along with techniques for hardware interfacing. A good embedded design requires familiarity with the underlying memory technology, including ROM, SRAM, EPROM, Flash EPROM, EEPROM storage mechanisms and devices. The processor bus interface is then covered in general form, along with an introduction to the 8051's bus interface. Most embedded designs can also benefit from the use of user Programmable Logic Devices (PLD). This subject is too complex for in-depth coverage here, so PLD technology is covered from a relatively high level. The central theme of designing an embedded system that can be proven to be reliable is illustrated with a simple embedded controller. The iterative nature of the design process is shown by example, and several design alternatives are evaluated. With the central part of the design completed, the remaining chapters cover the various types of I/O interfaces, bus operations, and a collection of information that is seldom included in the usual sources, but is often handed down from one engineer to another.

I hope that you will find this book to be useful, and welcome any observations and contributions you may have. If you should find any errors in the text, or if you know of some good embedded design resources that are not included on the CD-ROM, please feel free to contact me directly by e-mail: ken.arnold@ieee.org

Acknowledgment

This book is the direct result of contributions from many of the students I have been fortunate enough to have in my Embedded Computer Engineering courses at UCSD extension. They have provided a valuable form of feedback by sharing their notes and pointing out weaknesses in the text and in-class presentations. Some sections of this text were provided by David Fern and Steven Tietsworth.

Finally, I would also like to thank my family for supporting me and, Mary, Nikki, Kenny, Daniel, Amy and Annie for being patient and helping out when I needed it!

Dedication

This book is dedicated in memory of my father, Kenneth Owen Arnold, who always encouraged me to follow my dreams. When other adults discouraged me from entering the engineering field, he told me "If you really like what you're doing and you're good at it, you will be successful." Nowadays I get paid to have fun doing things I'd do for free anyway, so that meets my definition of success! Thanks, Dad.

1 Introduction

Why are microprocessors and microcontrollers designed into so many different devices? While there are many dry and practical reasons set forth, I suspect one of the strongest motivations for using a microprocessor is simply that it is a lot more fun. Over the past few decades of the so-called "computer revolution" I have seen many products and projects that could have been handled without resorting to a computer. Yet there is always a tendency to rationalize the choice of a micro-based solution by economic or technical arguments to support the decision. In fact, most of the really excellent products were successful to a great extent because they were fun to develop. Many of the best product ideas have occurred when someone was "playing" with something they were interested in. In my own experience, I have found learning something new is much easier and more effective when I am "just playing around" rather than trying to learn in a structured way or against a deadline. Studies of various educational methods also indicate "coached exploration" is more effective than the traditional methods. These and other observations lead me to the conclusion that the best way to learn about a microcontroller is by "playing" with one. No book, no matter how well written, can possibly motivate and educate the student as well as building and playing with a microcontroller. The best way to learn the concepts in this book is to build a simple microcontroller. Even if it is capable of nothing more than blinking a light, it will provide a concrete example of the microcontroller as a tool that can be fun to use. To ease this effort, a companion System Development Kit (SDK), is available to accompany this text. It incorporates the functions of a stand-alone single board computer (SBC), and an In-Circuit Emulator (ICE). It also serves as a sample embedded controller design. The design is included on the CD-ROM and web site for this book, so anyone can reproduce and use it as a learning tool. By applying the guidelines set forth in this book to real world hardware, the reader can learn to design reliable embedded hardware into other products.

Objectives

The reader, in conjunction with an instructor or advisor, should know how to do the following things after reaching the end of this book:

- Interpret design requirements for the design of an embedded controller
- Read and understand the manufacturer's specification sheets
- Select appropriate ICs for the design
- Interface the CPU, memory, and I/O devices to a common bus
- Design simple I/O (Input/Output) interfaces
- Define the decoding and interconnection of the major components
- Perform a worst case analysis of the timing and loading of all signals
- Understand the software development cycle for a microcontroller
- Debug and test the hardware and software designs

These tasks represent the major skills required in the successful application of an embedded micro. In addition, other abilities such as the design and implementation of simple user programmable logic will be covered as required to support the proficient application of the technology.

Embedded Microcomputer Applications

There is an incredible diversity of applications for embedded processors. Most people are aware of the highly visible applications, but there are many less apparent uses. Many of the projects my students have chosen turned out to be of practical use in their work. However, they have covered the entire range from the economically practical to the blatantly absurd. One practical example was the use of a microprocessor to monitor and control the ratio of ingredients used in mixing concrete. About a year after the student implemented the system he wrote to inform me that the system had saved his company between two and three million dollars a year by reducing the number of "bad batches" of concrete that had to be jack hammered out and replaced. Another example was that of a student who suspended a ball by airflow generated by a fan and provided closed loop control of the ball's position with the microprocessor. The only thing that many of the student projects really had in common was the use of a microcontroller as a tool.

Some of the actual commercial applications of embedded computer controls that the author has been directly involved with include:

A belt measures a person's heart rate and respiration that signals an alarm when safe limits are exceeded. A radio signal is then transmitted to a microcontroller in a pocket pager to display the type of problem and the identity of the belt.

An environmental system controls the heating ventilating and air conditioning in one or more large buildings to minimize peak energy demands.

A system that measures and controls the process of etching away the unwanted portions of material from the surface of an Integrated Circuit being manufactured.

The fare collection system used to monitor and control entry to a rapid transit system based on the account balance stored on the magnetic stripe on a card.

Determination of exact geographic position on the earth by measuring the time of arrival of radio signals received from navigational beacons.

An intelligent phone that receives radio signals from smoke alarms, intrusion sensors, and panic switches to alert a central monitoring station to potential emergency situations.

A fuel control system that monitors and controls the flow of fuel to a turbine jet engine.

Selecting a particular processor for a given application is usually a function of the designer's familiarity with a particular architecture. While there are many variations in the details and specific features, there are two general categories of devices: microprocessors and microcontrollers. Microcontrollers are generally used for dedicated tasks and include other functions in addition to the central processor.

Microcomputer and Microcontroller Architectures

The key difference is that a **microprocessor** contains the Central Processing Unit (CPU) only, whereas a **microcontroller** has memory and I/O on the chip in addition to the CPU. **Microcomputer** is a general term, which applies to complete computer systems implemented with either a microprocessor or microcontroller. Microprocessors are generally utilized for relatively high performance applications where cost and size are not the most critical selection criteria. This is because microprocessor chips that have their entire function dedicated to the CPU have room for more circuitry to increase execution speed, but require external memory and I/O hardware. Microprocessor chips are used in desktop PCs and workstations where software compatibility, performance, generality, and flexibility are important. Microcontroller chips on the other hand, are usually designed to minimize the total chip count and cost by incorporating memory and I/O, with application specialization at the expense of flexibility. In some cases the microcontroller has enough resources on-chip that it is the only IC required for a product. Examples of a single chip application include the key fob used to arm a security system, a toaster or hand held games. The hardware interfaces of both devices have a lot in common, and those of the microcontrollers are generally a simplified subset of the microprocessor. If the primary design goals for each type of chip were summarized in a phrase, microprocessors are most flexible and microcontrollers are most compact.

There are also differences in the basic CPU architectures used, which tend to reflect the application. Microprocessor based machines usually have a **von Neumann architecture** with a single memory for both programs and data to allow maximum flexibility in allocation of memory. Microcontroller chips on the other hand frequently embody the **Harvard architecture**, which have separate memories for programs and data.

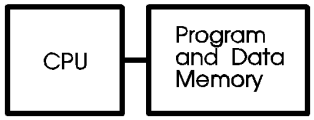


Figure 1-1) von Neumann Architecture

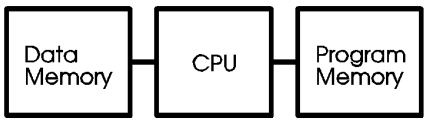


Figure 1-2) Harvard Architecture

One advantage this has for embedded applications is due to the two types of memory used in embedded systems. They are the fixed program and constants stored in non-volatile ROM memory, and the working variable data storage residing in volatile RAM. Volatile memory loses its contents when power is removed, but non-volatile ROM memory always maintains its contents, even after power is removed.

The Harvard architecture also has the potential advantage of a separate interface allowing twice the memory transfer rate by allowing instruction fetches to occur in parallel with data transfers. Unfortunately, in most Harvard architecture machines, the memory is connected to the CPU using a bus that limits the parallelism to a single bus. A typical embedded computer consists of the CPU, memory, and I/O. They are most often connected by means of a shared bus for communication, as shown in the figure.

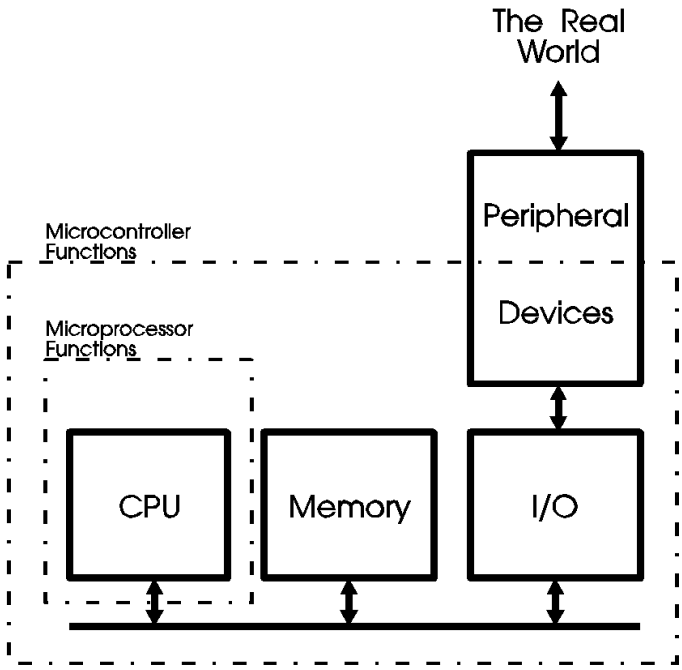


Figure 1-3) Typical Bus Oriented Microcomputer

The peripherals on a microcontroller chip are typically timers, counters, serial or parallel data ports, analog-to-digital and digital-to-analog converters that are integrated directly on the chip. The performance of these peripherals is generally less than that of dedicated peripheral chips, which are frequently used with microprocessor chips. Having the bus connections, CPU, memory, and I/O functions on one chip has several advantages:

- Fewer chips are required since most functions are already present on the processor chip.
- Lower cost and smaller size result from a simpler design.

- Lower power requirements because on-chip power requirements are much smaller than external loads.
- Fewer external connections are required because most are made on-chip, and most of the chip connections can be used for I/O.
- More pins on the chip are available for user I/O since they aren't needed for the bus.
- Overall reliability is higher since there are fewer components and interconnections.

Of course there are disadvantages too, including:

- Reduced flexibility since you can't easily change the functions designed into the chip
- Expansion of memory or I/O is limited or impossible
- Limited data transfer rates due to practical size and speed limits for a single chip
- Lower performance I/O because of design compromises to fit everything on one chip

Digital Hardware Concepts

In addition to the CPU, Memory, and I/O building blocks, other logic circuits may also be required. Such logic circuits are frequently referred to as "**glue logic**" because they are used to connect the various building blocks together. The most difficult and important task the hardware designer faces is the proper selection and specification of this "glue logic." Devices such as registers, buffers, drivers and decoders are frequently used to adapt the control signals provided by the CPU to those of the other devices. While TTL gate level logic is still in use for this purpose, the Programmable Logic Device (PLD) has become an important device in connecting the building blocks. Contemporary microcontroller designers need to acquire the following skills:

- Interpretation of manufacturers specifications
- Detailed, worst case timing analysis and design
- Worst case signal loading analysis
- Design of appropriate signal and level conversion circuits
- Component evaluation and selection
- Programmable logic device selection and design

REVIEW OF BASIC ELECTRONICS

Concepts And Notation Conventions

Fluid Flow Analogy: The glue logic used to join the processor, memories, and I/O is ultimately composed of logic gates, which are themselves composed almost entirely of the following component parts: transistors, diodes, resistors, and

interconnecting wires. In order to review the basic operation of the glue logic, we are going to begin at the component level with basic electronics concepts, presented as fluid flow analogies. In the diagram below, a battery provides a voltage source, which is equivalent to a pump, which provides a pressure source. Voltage, or pressure, is required to promote current flow in the circuit.

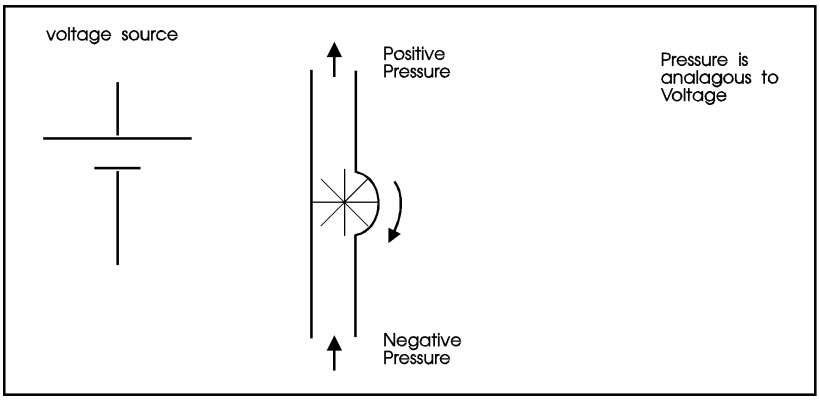


Figure 1-4) Circuit Analogies - Voltage is like Pressure

The voltage source provides the pressure "motivation" if you will, for current flow, and the resistance provides a limiting constraint on the amount of current actually flowing. The resistor will allow a current to flow through it that is proportional to the voltage across it, and inversely proportional to the resistance value. Higher resistance is like a smaller aperture for the fluid to flow through. The resistance results in a voltage, or pressure drop, across the resistance as long as current is flowing in the resistor.

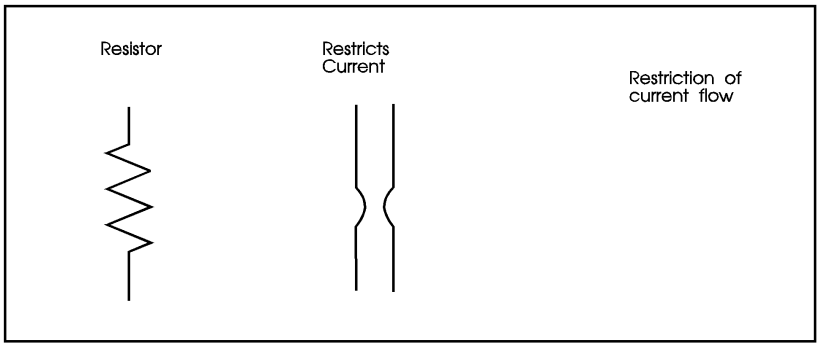


Figure 1-5) Circuit Analogies - Resistance is a Flow Restriction Aperture

Wire is like the piping connecting the components in a circuit. The flow of current in the circuit is controlled by the magnitude of the voltage (pressure) and the resistance (pressure drop) in the circuit. In the circuit below, the battery provides a voltage to force current through the resistor. The magnitude of the voltage (V) generated by the battery is developed across the resistor, and the magnitude of the resistance (R), determine the current (I). Note the "return" current path is often shown as "ground," which is the reference voltage used as the "zero volts" point. In this case, current flows from the positive battery terminal, through the wire,

then the resistor, then through the "ground" connection to the minus terminal of the battery. This is usually not the same as earth ground, which provides a connection to a stake or pipe literally stuck in the ground. The magnitude of the current in this case is I = V / R by re-arranging the equation V = I * R in the diagram below. (Ohm's law.) Another way to look at it is that whenever current flows through a resistor, there is a drop in voltage across the resistor due to the restriction in current.

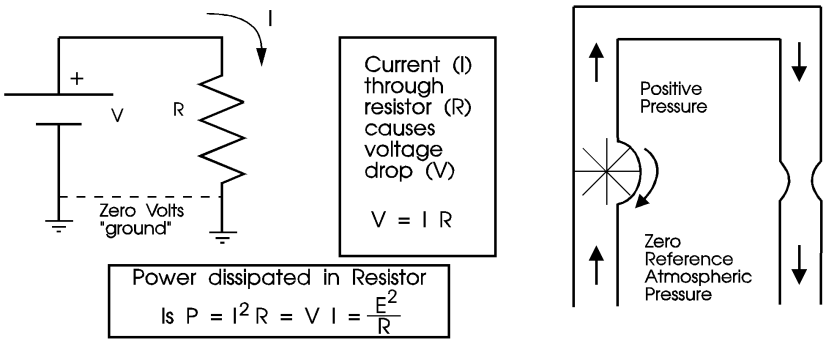


Figure 1-6) Circuit Analogies -- Voltage across R = Current times Resistance

Real components are not perfect voltage sources, resistances, etc. as we see here. They have parasitic values which limit their performance in the real world, and are subject to other limitations, such as operating temperature, power limits, and so on. Current flows only through a complete circuit, and in most cases (for a positive power supply) current flows from the power source through the circuitry, returning to the power supply through the common "ground" connection. Current flowing through any resistance results in the dissipation of power as heat. The power dissipated is $P = I^2R = V*I = V^2/R$. Note that voltage is sometimes denoted by the variable V and by E, for "electromotive force." All practical components have some resistance. Real batteries have an internal resistance, for example, which provides an upper limit to the current the battery can supply to an external circuit. Real wires have resistance as well, so the actual performance of a circuit will deviate somewhat from the ideal. These effects are obvious in some cases, but not in others. In an automobile starting circuit, it's not surprising that the battery, supplying 12 volts to a starter with internal resistance on the order of 0.01 to 0.1 Ohms, will result in currents of hundreds of amperes in order to start the engine. On the other hand, while consulting with a prominent notebook computer manufacturer, I uncovered a design error resulting in an internal current of hundreds of amperes flowing in the circuit for a few nanoseconds. Obviously, this wreaked havoc on the operation of the computer, and generated a great deal of electromagnetic noise!

One of the things you will learn in this book is how to avoid those kinds of mistakes. It's also important to remember that power is dissipated in *any* resistance present in the circuit. The power is proportional to the voltage times the current across the resistance, which is dissipating the power. In the last two examples, the amount of power dissipated instantaneously is quite high while the

current is flowing. When the current pulse is only a few nanoseconds long, however, it may not be obvious, since there won't be much heat generated.

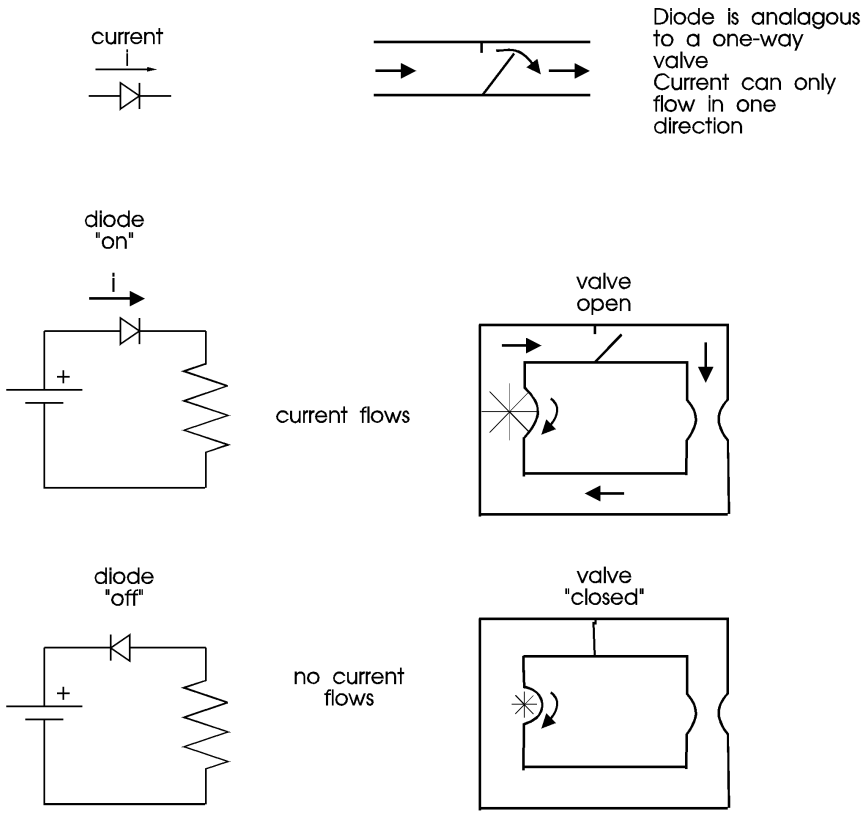


Figure 1-7) Diode Analogy

Circuit Analogies - Diode

The diode is a simple semiconductor device acting as a "one way" current valve. It only lets current flow in one direction. The figure illustrates how the diode operates like a “one-way” fluid valve.

Purists please note: This book does not use electron current flow. All electrical current flow will be "positive" or "conventional" current flow, meaning current always flows from the most positive terminal to the most negative terminal of a component. The use of positive current flow follows the intuitive direction of the arrows inherent in the component drawings for diodes, transistors, etc.

Transistor Analogy

The flow analogy can also be used to model how a transistor operates in a logic circuit. The transistor is an amplifier. It takes a small amount of energy to control a larger energy source just as a valve controls a high-pressure water source. There are two kinds of transistors: Bipolar and Field Effect Transistors. We will look at bipolar transistors first, which amplify current. A small amount of current flows in

the control circuit (the transistor base-emitter circuit) to turn the transistor on. This control current is amplified (multiplied by the “gain” or "Beta" of the transistor), allowing a larger current to flow in the output circuit (the collector-emitter circuit). Once again, the device is not perfect, because of the resistance, current, gain, and leakage limitations of real transistors. Bipolar transistors come in two polarities, NPN and PNP, with the difference being the direction in which current flows for normal operation. A bipolar PNP transistor is shown in the figure below.

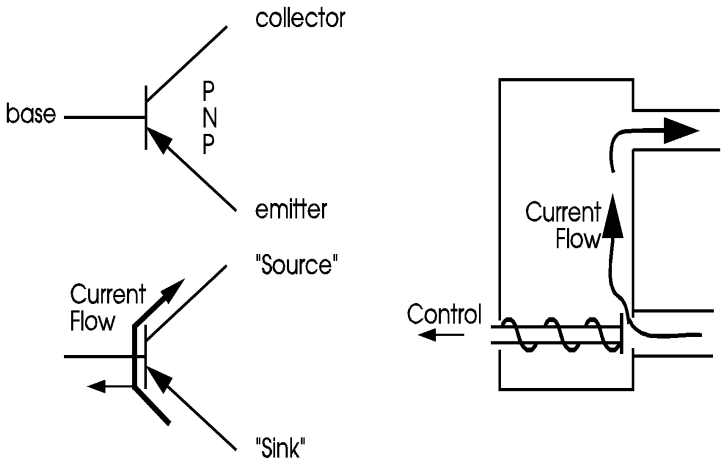


Figure 1-8) Transistor Analogy -- Bipolar PNP Transistor

For most of the illustrative circuit examples in this book, we will be using NPN transistors, as shown in the figure below.

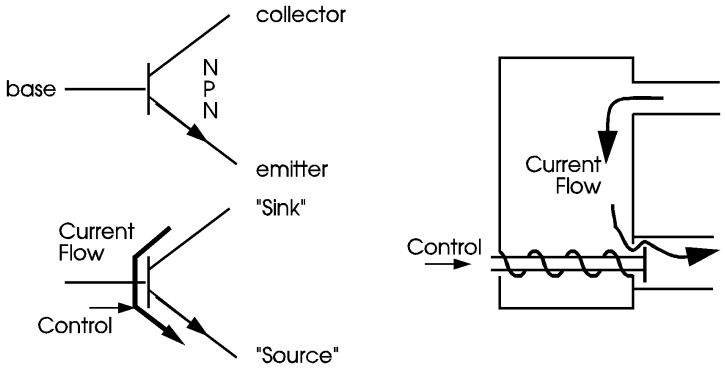


Figure 1-9) Transistor Analogy - Bipolar NPN Transistor

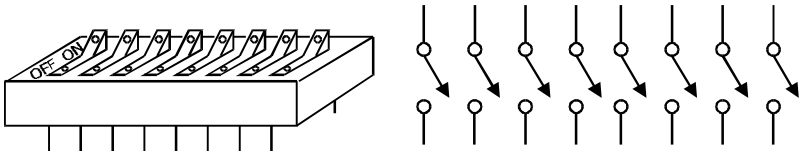
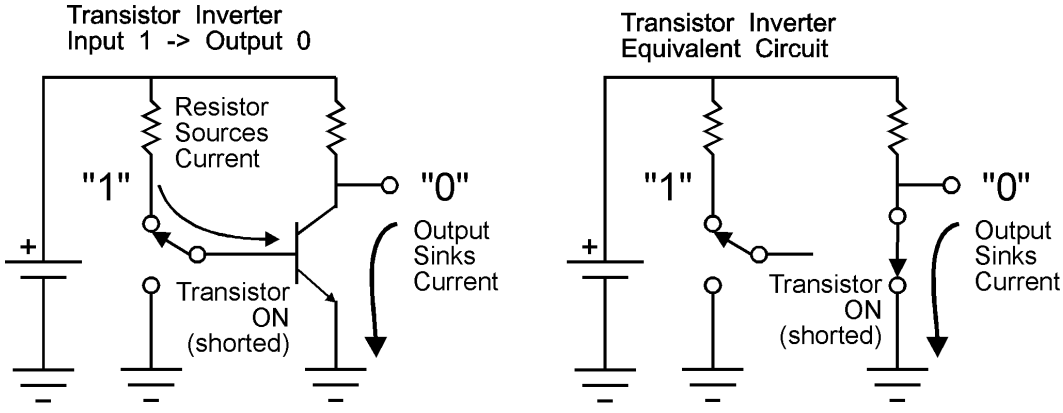


Figure 1-10) 8 Position DIP Switch Picture and Schematic

Mechanical switches are useful for direct input to digital circuits. One of the more convenient versions is a bank of rocker switches packaged into a module that can fit into the same location as a standard chip. The "Dual In-line Package", or DIP switch, is one of the easiest ways to add multiple switches to a microcontroller design. The mechanical switch has extremely low "on" resistance and high "off" resistance, unlike most semiconductor switches.

Transistor Switch ON

As can be seen in the next figure, an NPN transistor operating as a current controlled switch can be used to build a simple inverter. It changes a logic one on its input to a logic zero at its output, and vice versa. In this case, logic one is represented as a positive voltage, and a logic zero is represented by zero volts. The logic one input (positive input voltage) is supplied through a resistor from the power supply voltage to the transistor base terminal, resulting in a small base control current into the base.



Transistor Switch "ON" **Equivalent Circuit**

Figure 1-11) The Transistor Inverter: Input = 1, Transistor ON

The transistor is used because it has gain allowing a larger output current to flow as controlled by a weaker input. When the transistor is turned on as much as it can be, the collector emitter circuit looks almost like a short circuit, effectively connecting the output to ground or zero volts, giving a logic zero on the collector output. When the transistor collector is shorted to ground, current flows from the supply through the resistor and into the transistor collector to ground. The transistor is said to **sink** the resistor current into ground. If there is an external load, such as another inverter or gate, connected to the collector output, the transistor can also sink current from the load. This is also referred to as pulling down the output voltage. The current sinking capacity of the transistor limits the number of devices this inverter can drive.

Transistor Switch OFF

When the input is connected to logic zero (ground voltage), no current flows into the base of the transistor, since its base and emitter terminals are at the same voltage. When there is no current flowing in the base, the transistor will not allow current to flow in the collector emitter circuit either, so the circuit behaves like the transistor was removed from the circuit. The output resistor will source current to any potential load. The output is pulled up to the supply voltage, resulting in logic one at the output. Once again, there is a limit to the resistor's ability to source current, resulting in a limit to the number of loads being attached to this circuits output. Notice these two limits are defined by the ability of the transistor to pull down the output, and the resistor's ability to pull up the output become the main limits to its ability to drive other devices. Gates can be constructed by adding diodes or transistors to the inverter circuit above.

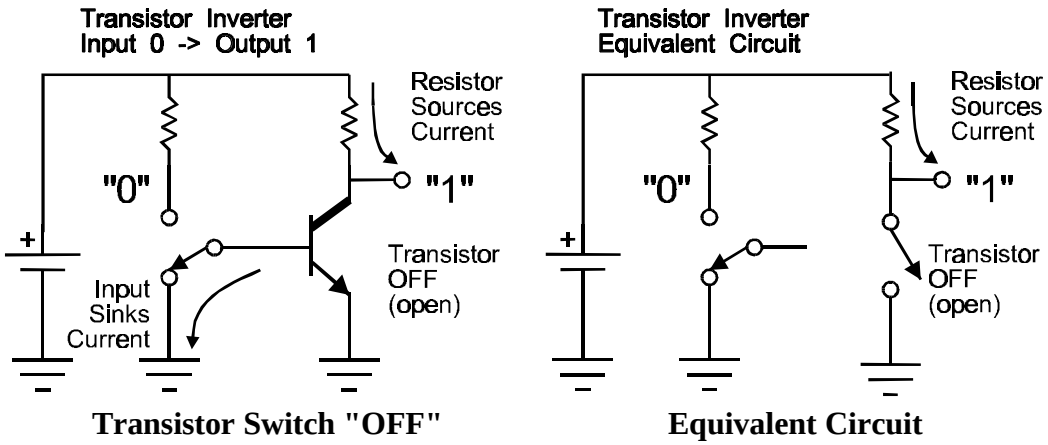


Figure 1-12) The Transistor Inverter: Input = 0, Transistor OFF

The FET (Field Effect Transistor) as a Logic Switch

Most of the logic devices used in highly integrated circuits use a different transistor technology, referred to as a field effect transistor. They perform a similar function to the bipolar transistors discussed earlier, but they are voltage-controlled. While the current flowing in the base controls bipolar transistors, the voltage between the gate and source controls field effects transistors. The gate voltage of a field effect transistor controls the current flowing in the drain-source circuit. The symbol for the FET shows the gate to be insulated from the source-drain circuit.

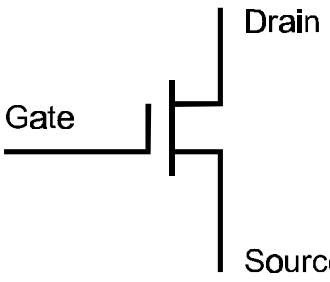


Figure 1-13) Field Effect Transistor Switch Symbol

This type of FET is referred to as a MOSFET (Metal Oxide Semiconductor FET), since the insulating material is Silicon Dioxide (SiO_2) commonly known as glass, and for the early devices, the gate was made of metal. Similar to the bipolar NPN and PNP transistors with opposite polarity, FETs come in N and P channel varieties, referring to the polarity of the source drain element of the device.

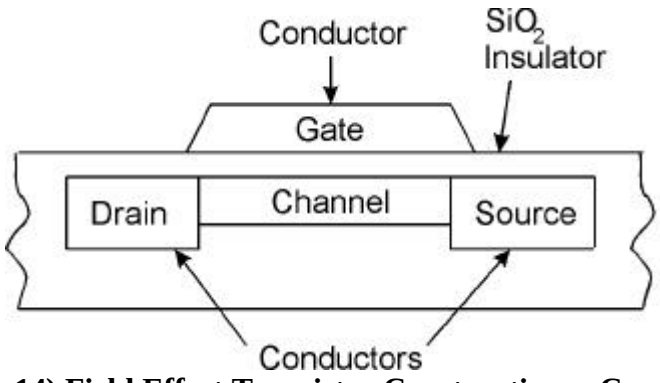


Figure 1-14) Field Effect Transistor Construction -- Cross Section

NMOS Logic

The conductive state of the FET's channel is what allows or prevents current from flowing in the device. For a typical logic N-Channel MOSFET, the channel becomes conductive when the gate has a positive voltage with respect to the source, allowing current to flow between the drain and source terminals. When the gate is at the same voltage as the source, no current flows. The design of MOS logic circuits can be almost exactly equivalent to the bipolar inverter we saw earlier, substituting an N-channel MOSFET for the bipolar NPN transistor. In fact, the most of the early microcontroller integrated circuits were manufactured using variations of this method, and are referred to as NMOS logic. As can be seen from the following figure, the NMOS FET circuit behaves in an equivalent way to the NPN transistor inverter. When the gate (control input) of the NMOS FET is at a positive voltage, the FET is ON, effectively shorting the source and drain pins. When the gate is at 0 volts, the FET is OFF, opening the circuit between the source and drain. Older NMOS logic ICs use this type of circuit. The original 8051 chip was an NMOS processor.

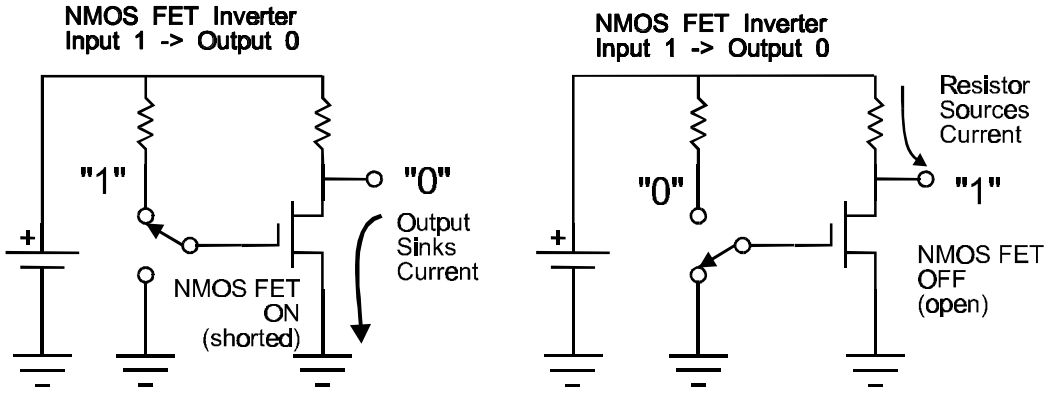


Figure 1-15) NMOS Inverter Circuit

CMOS logic

CMOS logic (Complementary symmetry MOS), another form of MOS logic, has the advantage over NMOS logic for low power circuitry and for very complex integrated circuits. NMOS logic is relatively simple, but it has one serious drawback: it consumes a significant amount of power. In fact, it would be impossible to manufacture the largest ICs using NMOS logic, as the power dissipated by the chip would cause it to overheat. This is the main reason CMOS logic has become the dominant form of logic used for large, complex ICs. Instead of using a resistor to source current when the output is high, a CMOS device uses a P channel MOSFET to pull the output high. CMOS logic is based on the use of two complementary FETs that switch the output between the power supply and ground. A simple CMOS inverter is shown in the figure below.

CMOS logic uses two switches, one P-channel pull-up transistor, and one N-channel pull-down device to pull the output low or high, one at a time. CMOS logic is designed with an N-channel device that turns on and conducts when the gate voltage is at logic one (positive voltage), and the P-channel device turns on when the gate is at ground voltage. A CMOS inverter is comprised of a pair of FETs, one device of each type, as shown in the figure.

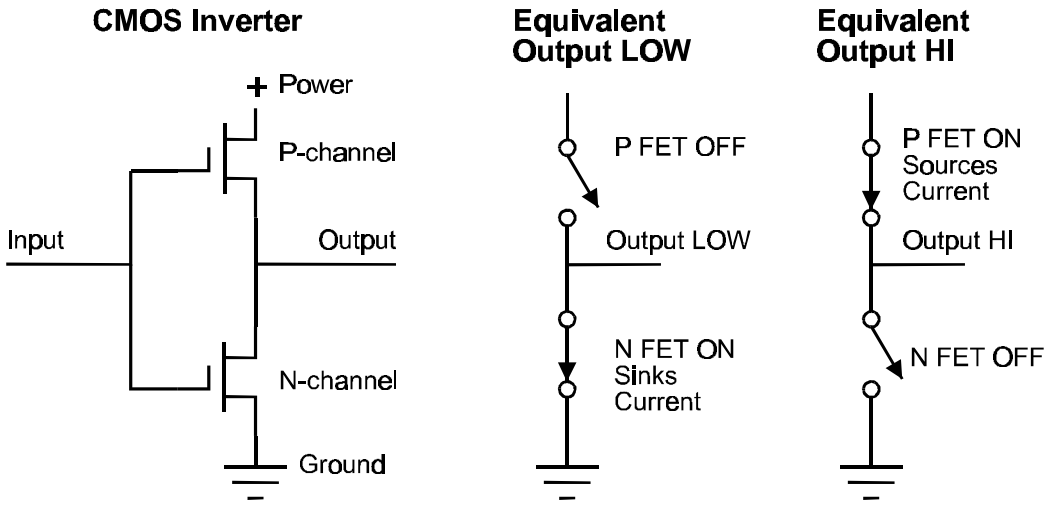


Figure 1-16) CMOS Inverter Circuit and Equivalent Output

When the transistor gate inputs are at logic one (positive voltage), the P-channel device is off, and the N-channel device is on, effectively connecting the output to ground, or logic zero. Likewise, when the input is grounded, the P-channel device turns on and the N-channel device turns off, effectively connecting the output to the positive supply voltage, or logic one. Gates and more complex logic functions can be constructed by using series and parallel connected MOSFETs in circuits similar to the one above. The gate of a MOSFET, as implied by the symbol, is essentially an open circuit. In fact, the gate of a MOSFET does have an **extremely** high resistance. The operation of the MOSFET's channel is controlled by the voltage of the gate, unlike the bipolar NPN transistor we examined in the inverter, which is controlled by input (base) current. Bipolar transistors are current amplifiers, with their output current being controlled by their base current. FET outputs, on the other hand, are dependent on the gate voltage.

Since almost no current flows in a CMOS output when it is driving a CMOS gate input in the steady state condition, these logic devices consume much less power than the other types. MOS logic has some other advantages over bipolar logic, since there is almost no input current (<1 nano-Amp, or 10^{-9} Amp), so it does not need to exact a DC current load on the device driving it. This is good news, because it means that the input current of a CMOS device does not limit the number of gates that can be connected to the output of the driving gate. The number of gate inputs that a single gate output can drive is the gate **fan-out**. Fan-out applies between gates of the same logic family, as different families of logic have different output capabilities and their inputs present different loads.

Now for the bad news about the high input resistance of MOS devices: the insulation separating the input from the channel is very thin (measured in Angstroms). This thin layer can easily be punctured by an electrostatic discharge (ESD), such as occurs regularly when dissimilar materials rub against one another. Just walking across the room can generate 10's of kilovolts, which is more than

enough to destroy a MOS device. As a result, special precautions must be taken to prevent damage to MOS devices. When handling these devices, it is important to ground your body BEFORE touching the device, and keep the device at or near ground. Special wrist straps and workspace mats are available to assist in keeping static voltages from building up, and for dissipating them when they do occur. Special, conductive bags and containers should be used when possible to contain sensitive devices.

CMOS power consumption is usually dominated by the power consumed during the transition of a logic device from one state to another. As a result, pure CMOS devices consume only a few microamps of current when they are not switching, and the bulk of the current drawn is a function of clock frequency. The higher the clock frequency, the greater the current consumption. For pure CMOS, the power supply current is linearly proportional to the clock rate.

Mixed MOS

Many logic devices labeled as CMOS are actually a mixture of NMOS and CMOS, because the manufacturer needs to compromise the extremely low power of CMOS with the performance of NMOS logic. This can be a problem for designers of battery powered systems, since the current requirement (and the resulting battery life) of a pure CMOS circuit is orders of magnitude better than an NMOS circuit. Many CMOS memories are actually mixed MOS, and are not appropriate for battery powered systems. True CMOS chips can retain their contents for years using only a single coin cell to maintain power to the memory.

Real Transistors Don't Eat Q

So far we have described the various types of transistors as perfect switches which have zero resistance when they're on and infinite resistance when they're off. When we examine the actual behavior, we will find that real transistors do not exhibit these characteristics. A transistor switch may have tens or hundreds of Ohms of resistance when it is on, and hundreds or even tens of thousands of Ohms of "leakage" resistance when it's off. As a result, the logic outputs aren't perfect either. When the transistor is on, the output voltage is a function of the output current, due to the voltage drop across the resistance. As the diagram shows, the output voltage of a logic device will depend upon how much current is flowing in the output and the resistance of the switch.

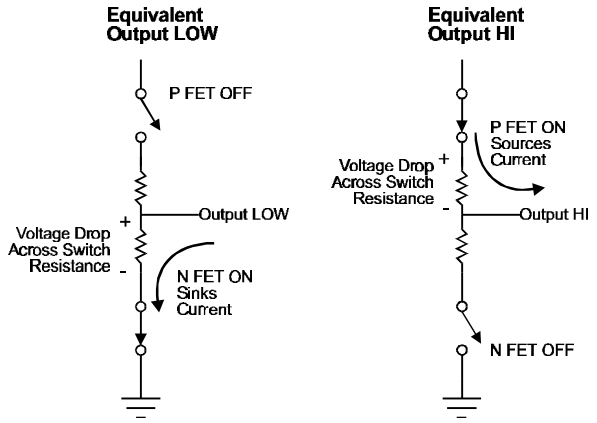


Figure 1-17) Logic Outputs Voltage is Current Dependent

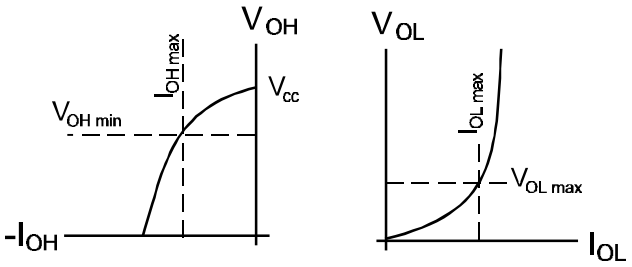


Figure 1-18) Output Voltage, Vo vs. Current, Io

Unfortunately, the switch resistance is also non-linear so that the switch resistance changes as the voltage across the switch changes. This makes it difficult to picture the output behavior under different operating conditions. The behavior will also differ from one device to another, over temperature, and so on. Manufacturers only specify the output characteristic at one point on the curve, Vo at Io max. As a result, the best we can do is to look at the output characteristics graphically.

Logic Symbols

Logic symbols are used to represent the logic functions in a more abstract way allowing the designer to specify the logical function of a circuit without getting into the details of the underlying components, such as the transistors and resistors. The logic symbols used in this text represent those most commonly used in commercial documentation. There are other standards, such as the ANSI/IEEE standard gate level symbols, but they are not encountered as frequently in practice.

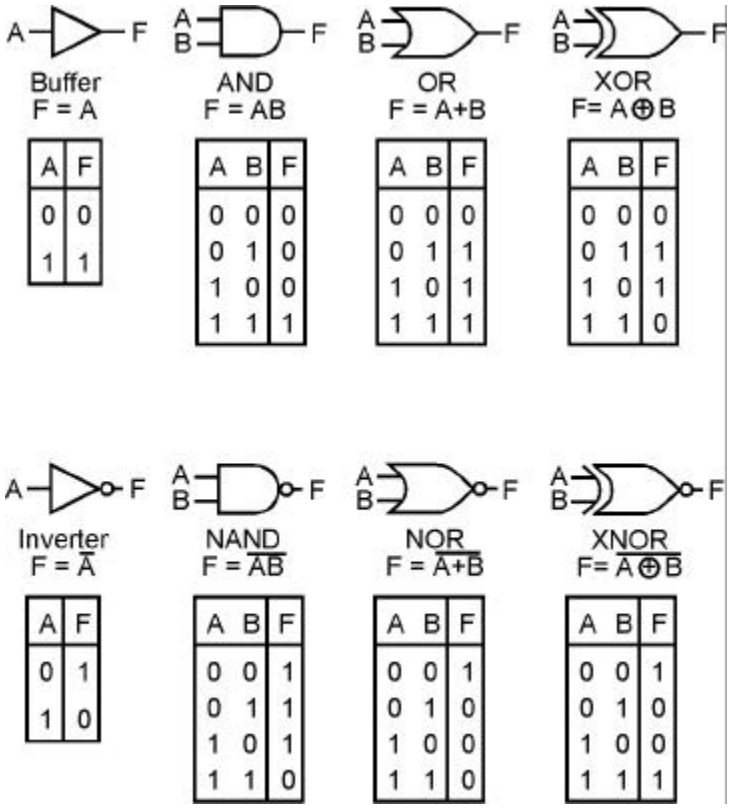


Figure 1-19) Logic symbols, Symbolic notation and Truth tables

The logic symbols in the figure show the shapes and Boolean logic functions for the most common gate configurations. The buffer device is a triangle, the symbol for an amplifier, because it amplifies the input signal allowing an increase in the number of loads that can be driven. Note that a small circle, often referred to as a "bubble," on an input or output terminal designates a logical inversion. Thus the inverter is shown as a triangle (amplifier) with a bubble on the output to signify the logic level inversion on the output. The logic voltage levels for TTL logic are:

Positive Logic	**Corresponding TTL Logic Voltages**
0=False=Lowest Voltage Level	0=input voltages 0 to 0.8 volts (low)
1=True=Highest Voltage Level	1=input voltages 2 to 5 volts (high)

This means that a TTL compatible logic input is guaranteed to respond to an input signal between 0 and 0.8 volts as a logic zero, and input voltages from 2 to 5 volts as a logic one. Note that Voltages between 0.8 and 2 volts are not valid logic levels.

Logic voltage levels are different for different types of logic, but the most common logic levels correspond to the original TTL (Transistor Transistor Logic), using a 5 Volt power supply. CMOS levels, using 3 or 5 Volt power, are also common. TTL and CMOS logic, like almost every other type of logic in common use, are called positive logic because the most positive voltage corresponds to the logic one value.

Tri-state Logic

Tri-state Logic does not refer to orderly thinking in a three state geographic region... When we speak of binary (base two number) values, we mean that a given bit or logic signal can take on either one of two valid states (zero or one) at any instant in time. A logic gate not forcing its output to be either one or zero, is said to be "Tri-stated." Tri-state logic does not refer to base three numbers, but rather to a third invalid logic state, when the output of a logic device is neither sinking nor sourcing current. The so-called third state is really an undefined condition, because the device output is not forcing a logic level on its output. It is said to be in a floating, high impedance, passive, or Hi-Z state, since the output circuits are effectively disconnected. A tri-state driver connected to one signal wire of the bus is shown in the following figure.

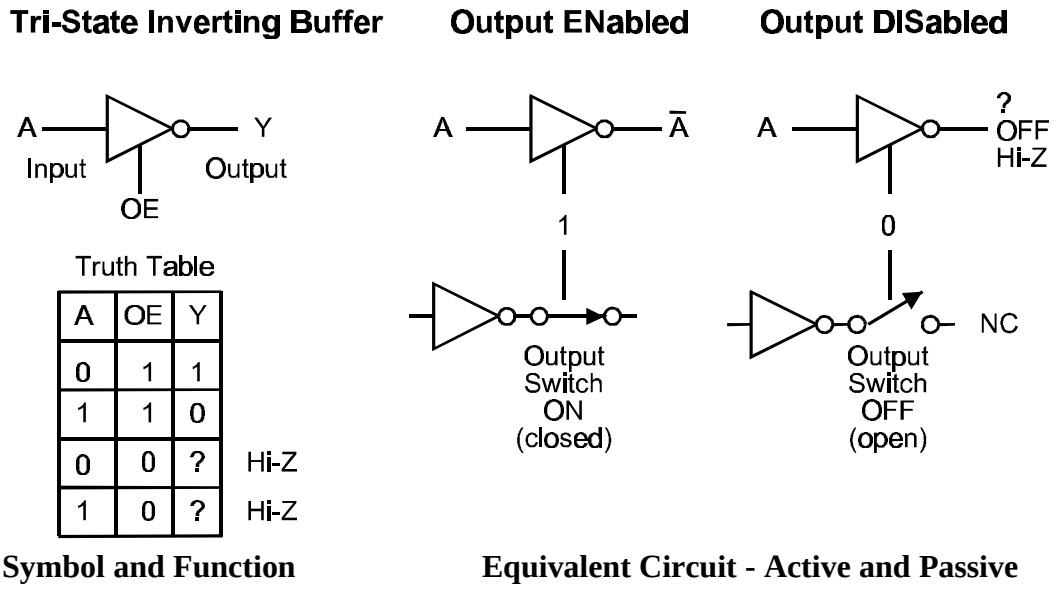


Figure 1-20) Tri-state Buffer - Active and Passive States

On the left is an inverting buffer with an enabled tri-state output. On the right side is an example showing two of the same type of buffers, with the top device in the disabled or passive state, and the lower device is enabled or actively driving the data bus to a logic one level. The control signal determines whether the output is passive or active, and is called the output enable or OE signal. The device shown above is actively driving the bus whenever the OE control line is at a logic one level, and is passive when the OE line is at a logic zero level. Most of the time, output enable signals are **active low,** meaning that the output is enabled when the /OE signal is low, and passive when the /OE signal is high. This is shown on the logic symbol with an inversion bubble where the enable signal enters the logic device.

As computer circuits become more dense and complex, the connecting wires have become increasingly difficult to route and interconnect. This is especially true on a densely packed integrated circuit, where it turns out that the wiring is more valuable than the logic gates! On one common CPU chip, 68% of the chip area is used for interconnect wiring. Even on a circuit board, it is important to use the board wiring in an efficient way. Since there are many parallel address and data lines that must go to multiple chips, the multiplexing approach makes it practical to connect many devices. The purpose for using Tri-state logic is to allow multiple devices to share wires by taking turns one at a time. This may sound a bit silly, but it is just one form of multiplexing, or sharing a resource that needs to be allocated among multiple devices. When the resource is a collection of parallel data wires, referred to as a data bus, and the bus is shared by multiple microcomputer CPU and peripheral devices transferring information one at a time in sequence, it is referred to as a multiplexed data bus.

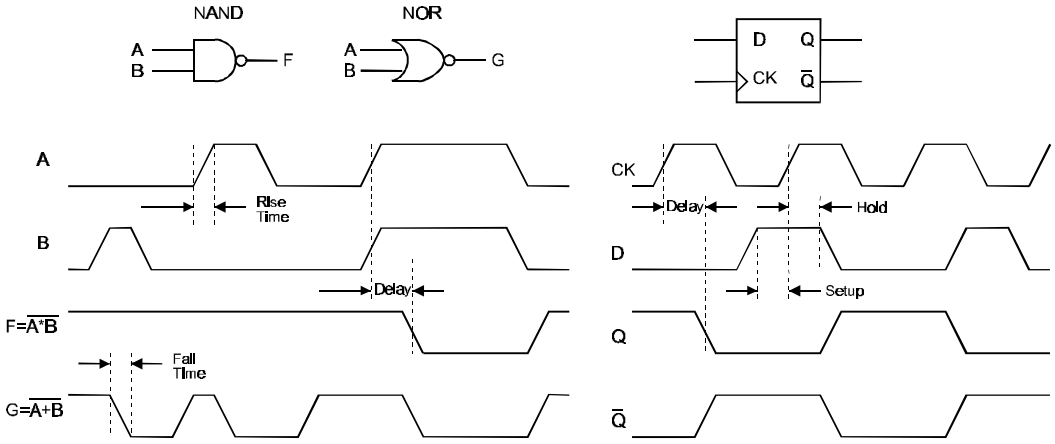


Figure 1-21) Timing Diagram Notation Examples

Timing Diagrams

The timing diagram is the standard "language" of illustrating timing relationships between different parts of a design. In order to understand the relationship of different signal with respect to time, it is necessary to learn how to read and interpret timing diagrams. The diagram above shows examples of asynchronous (un-clocked or combinatorial gates) and synchronous (clocked flip-flop) logic. The notation used in this book is representative of that used in most component specifications. Timing specifications, such as delay, setup, and hold times, specify the limits under which the device is guaranteed to operate as intended. If those specifications are violated, the device may very well operate correctly most of the time. However, a change in temperature, voltage, or variations from unit to unit may make the circuit unreliable. The most undesirable result of timing violations is that the circuit makes very infrequent errors, perhaps one error in hundreds of hours of operation. If you have ever wondered why your PC crashes mysteriously for no apparent reason, timing specification violations may well be the cause!

Timing relationships are particularly important for signals that are “time shared” on a single wire. A group of these wires which carries different information at different times is called a bus.

Multiplexed Bus

In order to describe the timing of such a shared data bus, it is necessary to define some notation for timing diagrams. The notation used in this book is below.

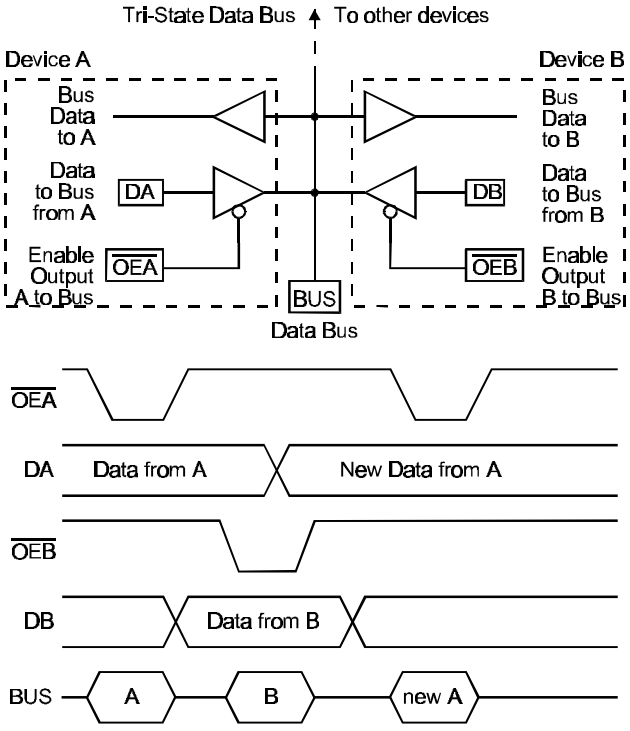


Figure 1-22) Time Multiplexed Data Bus and Timing

The terminology for timing parameters is covered in another chapter, but the basic concept for time multiplexed data on a bus is shown in the above figure. The two devices are alternately enabled to drive the data bus wire, allowing each to drive the bus in turn. Only one device is allowed to drive the bus at a time when it is operating correctly.

Timing diagrams are a critical method to allow accurate and unambiguous representation of the time related operations of digital circuits, which we will be using to understand and document the correct sequence of operations for microcomputer systems. Timing analysis, using these diagrams, allows the designer to determine safe and reliable limits to proper operation of the various circuits in the system. It is better to take a little more time to design a circuit correctly from the start, rather than having to find and fix bugs when they appear

during testing. This is especially of concern because of the increasing cost of fixing a bug as a product progresses through production and into the field.

Loading and Noise Margin Analysis

In addition to timing, the designer must consider the voltages and loads at the logic inputs and outputs. If the output of one gate is connected to the input of another, the designer must assure that the logic voltages are compatible. Once again, just as for the timing, violations of these specifications often results in infrequent errors that are very tricky to reproduce. Again, prevention is much simpler than tracking down bugs as they appear in production units.

The Design and Development Process

Structured design of a microcomputer requires the ability to do the system design and partitioning from the top down while implementing the system from the bottom up. The hardware design and development process should consist of the following steps:

1) Defining the requirements
2) Collecting information on potential components
3) Evaluate the components with respect to the requirements
4) Do a block diagram preliminary design and component selection
5) Perform a preliminary timing and loading analysis
6) Define the functions of the "glue logic"
7) Schematic entry using CAD (Computer Aided Design) software
8) Programmable logic device design and simulation
9) Detailed timing analysis & simulation, adjusting the design as required
10) Check the signal loading, buffering signals as needed
11) Document the design and generate a net list and bill of materials
12) Begin the design and layout of a printed circuit board
13) Implement the design in breadboard or prototype form
14) Program the memories and programmable logic as required for testing
15) Debug and verify operation using oscilloscope, logic analyzer, and in-circuit emulator
16) Update and complete documentation as the design changes

The order of tasks shown is variable, and some of the tasks may be performed in parallel. Software design is also frequently done in parallel with hardware design, and sometimes even before the hardware design. This is frequently a result of the fact that the cost and time required to develop the software exceeds that of the hardware development. In some cases the cost of modifying existing programs may be so high as to be impractical. In these cases, it is the designer's responsibility to maintain software compatibility with previous hardware designs.

Chapter 1 Problems

1. If an open-drain N-channel FET transistor is used as a logic output, is it possible to connect more than one open-drain transistor output to the same signal? What would the effect of doing so be on the resulting combined signal?

2. If a logic output sinks I_{OL} = 10 mA with an output voltage, V_{OL} = 0.5 volts, how much power is dissipated by a 450 Ohm resistor between the output an the 5V power supply?

3. How much current must a logic output source, in order to maintain an output voltage of 2.5V when driving a 5K resistor connected to ground?

4. In a CMOS inverter, there is a short period of time when both the N- and P-channel transistors are partially turned on when the input is changing from low to high or high to low. What effect will this have on power consumption? What characteristic in the input signal would reduce this effect?

2 Overview

Hierarchical Computer Organization

Another way of looking at a computer system is to look at the successive translations that occur from the high level code to the electrical signals that are really the only means of communication with the hardware. A computer system can be broken down into multiple levels or layers to show the translation of a specific instruction into a form that can be directly processed by the computer hardware. The hierarchical levels are discussed in detail in "Structured Computer Organization," by A.S. Tanenbaum. This hierarchy is shown in the diagram below.

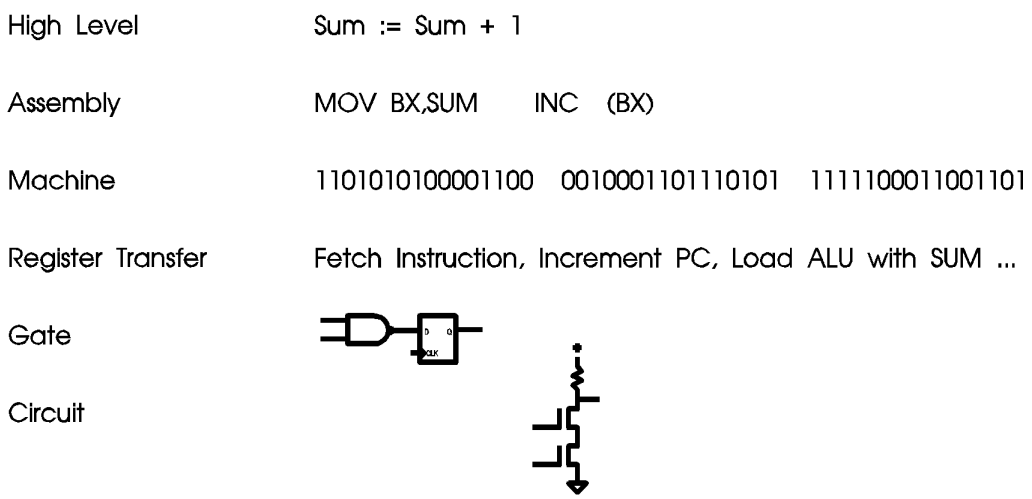


Figure 2-1) Layers of a Computer System

Language translators such as compilers and assemblers translate high level code into machine code that can be executed by the processor. The primary focus of this book will be from the assembly and machine language level downward.

Organization: von Neumann vs. Harvard

The von Neumann machine, with only one memory, requires all instruction and data transfers to occur on the same interface. This is sometimes referred to as the **"von Neumann bottleneck."** In common computer architectures, this is the primary upper limit to processor throughput. The Harvard architecture has the potential advantage of a separate interface allowing twice the memory transfer rate by allowing instruction fetches to occur in parallel with data transfers. Unfortunately, in most Harvard architecture machines, the memory is connected to the CPU using a bus that limits the parallelism to a single bus. The memory separation is still used to advantage in microcontrollers, as the program is usually stored in **non-volatile memory** (program is **not** lost when power is removed), and the temporary data storage is in **volatile memory**. Non-volatile memories, such as

ROM (Read Only Memory) are used in both types of systems to store permanent programs. In a desktop PC, ROMs are used to store just the start-up or bootstrap programs and hardware specific programs. Volatile **RAM** can be read and written easily and loses its contents when power is removed. RAM is used to store both application programs and data in PCs that need to be able to run many different programs. In a dedicated embedded computer however, the programs are stored permanently in ROM where they will always be available. Microcontroller chips that are used in dedicated applications generally use ROM for program storage and RAM for data storage. Memory technology is crucial to the design and understanding of embedded computers, and chapter 4 is dedicated to this important technology.

Microprocessor/Microcontroller Basics

There are three groups of signals, or buses, that connect the CPU to the other major components. The buses are:

- Data Bus
- Address Bus
- Control Bus

The **data bus width** is defined as the number of bits that can be transferred on the bus at one time. This defines the processor's "**word size**." Many chip vendors define the word size based on the width of an internal data bus. For the purposes of this book however, a processor with eight data bus pins is an 8-bit CPU. Both instructions and data are transferred on the **data bus** one "word" at a time. This allows the re-use of the same connections for many different types of information. Due to packaging limitations, the number of connections or pins on a chip is limited. By sharing the pins in this way the number of pins required is reduced at the expense of increased complexity in the external circuits. Many processors also take this a step further and share some or all of the data bus pins to carry address information as well. This is referred to as a **multiplexed address/data bus**. Processors that have multiplexed address/data buses require an external address latch to separate and hold the address information stable for the duration of a data transfer. The processor controls the direction of data transfer on the data bus.

The **address bus** is a set of wires that are used to point to the memory or I/O location that is to be read from or written to. The address signals must generally be held at a constant value for some period of time before, during and after the data is transferred. In most cases, the processor actively drives the address bus with either instruction or data addresses.

The **control bus** is an assortment of signals that determine what kind of information is on the data bus and determines where the data will go, in conjunction with the address bus. Most of the design process is concerned with the logic and timing of the control signals. The timing analysis is primarily

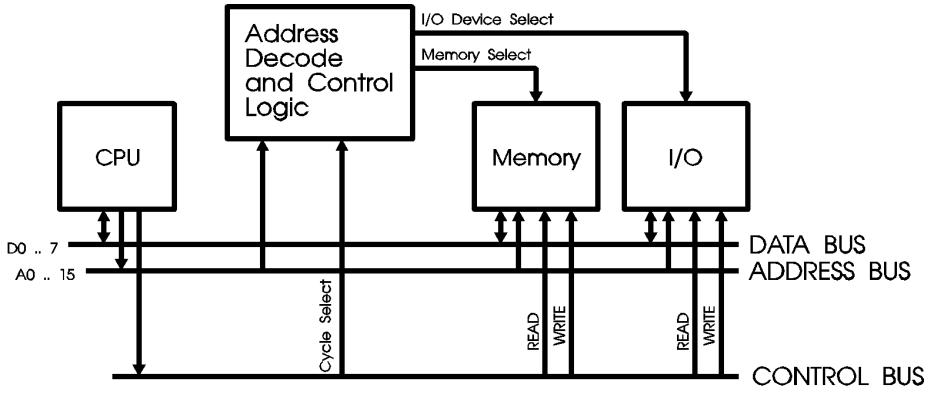
involved with the relative timing between these control signals and the appearance and disappearance of data and addresses on their respective buses.

Microcontroller: CPU, Memory, I/O

The interconnection between the CPU, memory, and I/O of the address and data buses is generally a one-to-one connection in most cases. The hard part is designing the appropriate circuitry to adapt the control signals present on each device to be compatible with that of the other devices. The most basic control signals are generated by the CPU to control the data transfers between the CPU and memory, and between the CPU and I/O devices. The four most common types of CPU controlled data transfers are:

1) CPU reads data/instructions from memory **(memory read)**
2) CPU writes data to memory **(memory write)**
3) CPU reads data from an input device **(I/O read)**
4) CPU writes data to an output device **(I/O write)**

In this book, **read** and **input** will be used interchangeably. These terms refer to the transfer of information from an external source into the CPU. **Write** and **output** will be used to denote the transfer of data from the CPU to an external destination. The data direction is defined with respect to the CPU.



Microcontroller, control logic, memory and I/O

Figure 2-2) Microcomputer Buses

Design Methodology, simple design example

The address decode and control logic shown in the diagram above is the key part of the design, which requires attention to timing analysis to guarantee signal logic and timing compatibility between the other blocks. The simplified timing diagram for such a system is shown below for a simple memory read and write cycle.

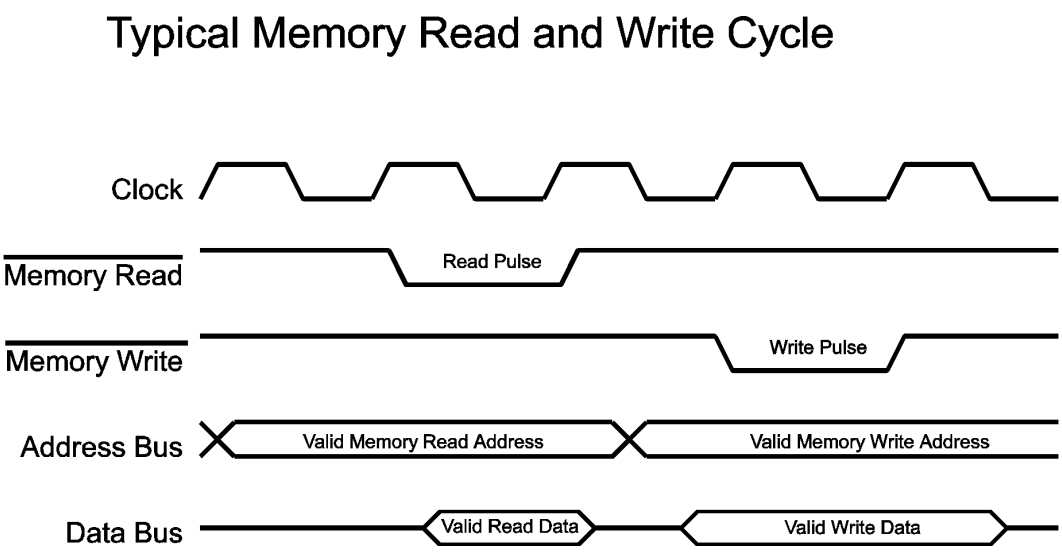


Figure 2-3) Generic Bus Timing Example

The figure above is a generic diagram and represents a typical example of a bus cycle for a typical CPU.

We see that there are two cycles:

> Memory Read: The processor places an address on the address bus, and activates the memory read signal by pulling it low, which causes the selected memory location to be placed on the data bus.
>
> Memory Write: The processor places an address on the address bus, data to be written on the data bus, and activates the memory read signal by pulling it low, which causes the selected memory location to be loaded with the data the CPU placed on the data bus.

Up to this point, we have discussed microcontroller architecture in a very general form, as it applies to most common devices. In order to go deeper into the operation of a microcontroller, it is appropriate to present one specific processor as an example. In order to really understand and apply this information to a real hardware and software design, it is necessary to cover one specific machine architecture in detail.

The 8051 Family Microcontroller Processor Architecture

You might wonder why the 8051 family of processors was chosen for this purpose, as it is a relatively old processor. If you read current technical journal articles, you might get the impression that all the action is in 32 bit micros. That is primarily due to the fact that the companies that sell the high-end devices are working very hard to put their newest technology in front of their customers, and they are the ones who write most of the trade articles.

It is important to note that the trade press is always emphasizing the high end 16, 32 and larger processors due to their dependence on the advertising revenue from chip vendors. Though you would never guess it from reading these publications, it is only recently that shipment of 4 bit microcontrollers has exceeded 8 bit units. It will be quite some time before the 16 bit micros will approach the sales volume that the 8-bit units have reached, and the 8-bit units are still growing in volume. According to one of the leading industry publications, there are more 8051 derivative CPU chips being produced than any other 8-bit micro. From this point forward, the 8051 family architecture will be used. Later on, other architectures and generic features not implemented in the 8051 will be discussed for completeness. Much like learning a foreign language, once you have learned the concepts, you will find that the next architecture you need to use will be much easier to learn.

The 8051 microcontroller was chosen as the example processor in this book for several reasons:

- The timing specifications are simple and allow a complete detailed timing analysis within the limited scope of this book.
- Interfacing to the processor's multiplexed address/data bus provides valuable design experience.
- Development tools, including assemblers, simulators and compilers are readily available as freeware shareware and demo versions.
- It is available at a low cost, allowing low cost versions of In-Circuit Emulators, peripheral components, and single board computers to be purchased by the student.
- The 8051 is the most popular microcontroller family, with many derivatives available, and it is manufactured by multiple vendors.
- The 8051 architecture is available in a wide range of cost, size, and performance. For example, one version is available in a 20-pin small outline surface mount package for less than a dollar in volume, and another one is about 8 -10X the speed of the original 8051.
- The 8051 CPU is also available as a building block for custom chip designs, and is the most popular CPU for "system on a chip" designs. It is also the only readily available, non-proprietary building block CPU architecture available for chip design.

Software tools for the 8051 family, such as assemblers, compilers and simulators are available at no cost on the Internet. Hardware tools, such as the combination software development kit and In-Circuit Emulator (the SDK which can be used in conjunction with this book), are available for under $100, and complete design documentation is available on the web to allow anyone to build their own.

In addition, the 8051 has the simplest timing specifications of a device which can address external memory, making it practical to go into the details of the design which are necessary to understand. With less than two dozen timing specifications, compared to several times as many for most other equivalent processors, it is possible to cover the timing specifications in detail. Once this process is understood, it is a straightforward jump to understanding and using the larger number of equivalent specifications characteristic of other devices.

Introduction to the 8051 Architecture

This section is intended to provide a broad overview of the 8051 microcontroller architecture. References to "8051" or "'51" in this book generally indicate the entire family of 8051 CPU instruction set compatible devices. Since the original 8051 had an internal read-only memory for programs, which was defined at the time the chips were fabricated, that device is not appropriate for our study. For flexibility and simplicity, we will be discussing the 8031, which does *not* have any internal program memory, but rather, fetches its program from an external memory device. Otherwise, almost all the versions of the processor family share the same features. If one were to do a practical commercial embedded computer design using an 8051 derivative, one could take advantage of the additional features that are commonly included in the more recent parts. For example, the NMOS versions of this family (e.g. 8031) described here have mostly been displaced by their CMOS counterparts, such as the 80C31. The 8032 and 80C32 with 256 bytes of internal data RAM and an additional timer, at about the same cost, have replaced the '31 versions. Most of the new versions of these devices have been built upon the features of the '32 version. Higher speed versions of the device, such as the Dallas Semiconductor 80C320, provide throughput equivalent to almost 100 MHz, compared to the original parts 12 MHz clock. The 8051 CPU element is even available as a standard building block for use in designing other chips. There are also 16 bit superset versions of the 8051 architecture! A simple 8051 system is shown in the next figure.

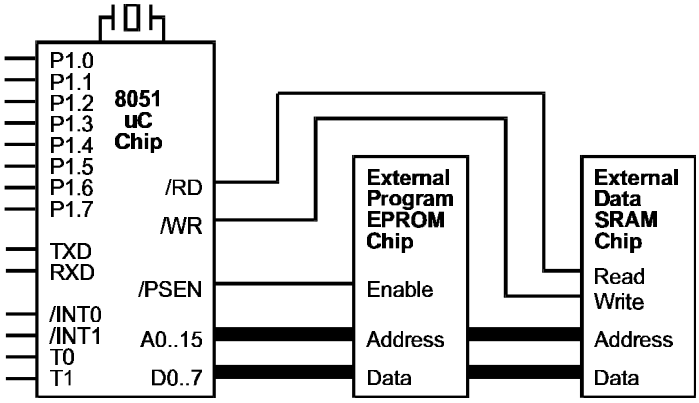


Figure 2-4) A Simple 8051 System Using External Memories

This block diagram shows a highly simplified version of the CPU with external program and data memory. (An address latch is also required, but not shown in this figure.) The program is stored in non-volatile ROM memory, such as an EPROM (Erasable and Programmable Read Only Memory), and the data is stored in a volatile RAM. In this configuration with external memory, the amount of useable I/O is limited by the number of pins which are used for the address, data, and control lines. Only port 1 and part of port 3 is available for user I/O in this case. This is a simplified version of the software development kit (SDK) used in this series of classes. The complete design documentation for the SDK, including schematics, PCB board layout, bill of materials and source code is available free on the Internet. In its simplest configuration, only the processor's internal memory is needed for the application, so most of the pins are available for I/O. In that case the microcontroller is the only required chip, which is also the lowest cost configuration. There are versions of this device that have internal program memory that can be programmed with an inexpensive programmer connected to a PC.

Now that we've introduced the 8051 architecture, we need to get into the "low level details" in order to really understand it. Up to this point we've had a view from 50,000 feet, where all the landscaping looks perfectly manicured. Now we need to get down to ground level, where we can see all the bits of trash and imperfections of reality. Every processor has its own idiosyncrasies, and the 8051 is no exception. While it may seem quite odd at first, it does have some very useful features, which make it fairly adept at handling the sorts of things that are often required in an embedded application. Having said that, let's get down to looking at the innards of the processor. The figure shows a top view of the processor with pin numbers, starting with pin 1 in the upper left corner.

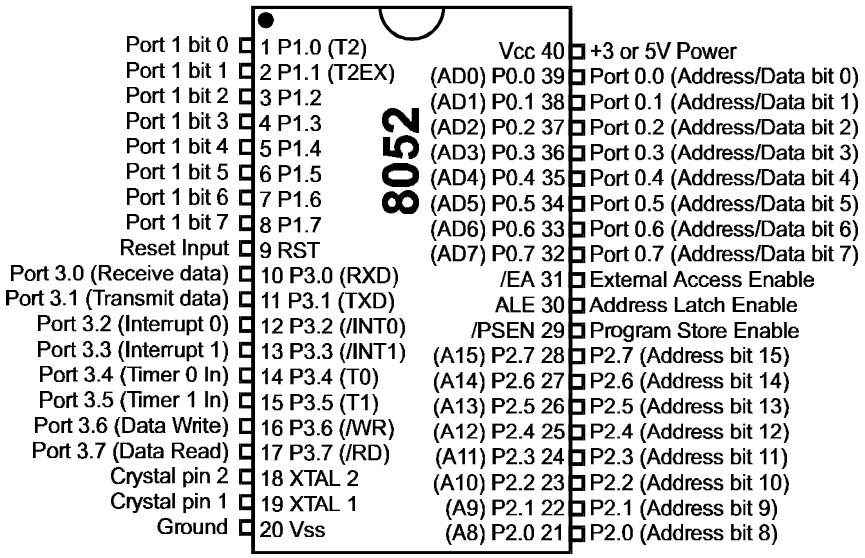


Figure 2-5) Top View of 8052 40-Pin DIP Package

The diagram shows the pin numbers, names and functional description of the pin functions for the 8052 CPU in a Dual In-line Plastic (DIP) package. The 80x1 and 80x2 pin definitions are identical, except for the fact that the 80x1 does not have Timer 2, so those pins are different on the 80x1 parts.

8051 Memory Organization

In order to understand the processor, it is necessary to see how the various memory spaces are organized. The memory organization of the 8051 family of processors may seem complex at first, however it as not as random as it might seem. There are separate memories for program storage, internal memory and registers, internal I/O functions, and external data memory. The program and external data memories are relatively simple. They each hold up to 64K bytes of instructions and data respectively. Program instructions are always fetched from program memory, and are indicated by the CPU activating the /PSEN pin. External data is transferred when the CPU executes a MOVX (MOV eXternal memory) instruction, and the CPU indicates this by activating the /RD or /WR line. The 8051 family chips only have three types of external memory cycles:

- Program read when /PSEN goes low
- External data read when /RD goes low
- External data write when/WR goes low

This makes interfacing other bus-oriented devices to the processor relatively easy. (Some general purpose or PC CPUs have many different types of bus cycles.)

The internal data address space of the 8051 family is not quite as simple as the external memories. It includes 4 banks of 8 registers, memory that can be accessed

one byte or one bit at a time, a stack, and the Special Function Registers (SFRs) which hold the data and control information for the serial port, timers, and other I/O. This internal memory address space can be accessed in several different ways. The internal data space of the CPU can be rather confusing at first, but it is one of the characteristics of the 8051 family, which allows so much to be done with such limited resources.

The 8051 CPU manipulates operands in three memory address spaces.

- **64K-byte Program Memory** (external program memory on the 8031) which is enabled when the processor is fetching an instruction to be executed, **signaled by activating the CPU's /PSEN control line**. The MOVC instruction also activates /PSEN to enable reading the code memory into the accumulator for accessing lookup tables and other unchanging data stored in the program memory space.

- **64K-byte External Data Memory** which is enabled when the processor reads or writes data from the **External** data memory, **signaled by activating the /RD and /WR control lines**. **This occurs only when a MOVX instruction is used** to read or write from external memory.

- **Internal Data RAM** (128 bytes for the '31, 256 bytes for the '32) and **Special Function Registers (SFR)**. Four Register Banks (each bank has eight registers), 128 individually addressable memory bits, and the stack reside in the Internal Data RAM. The stack depth is limited only by the available Internal Data RAM. Its location is determined by the 8-bit Stack Pointer. The 128-byte Special Function Register address spaces are shown in the figure below.

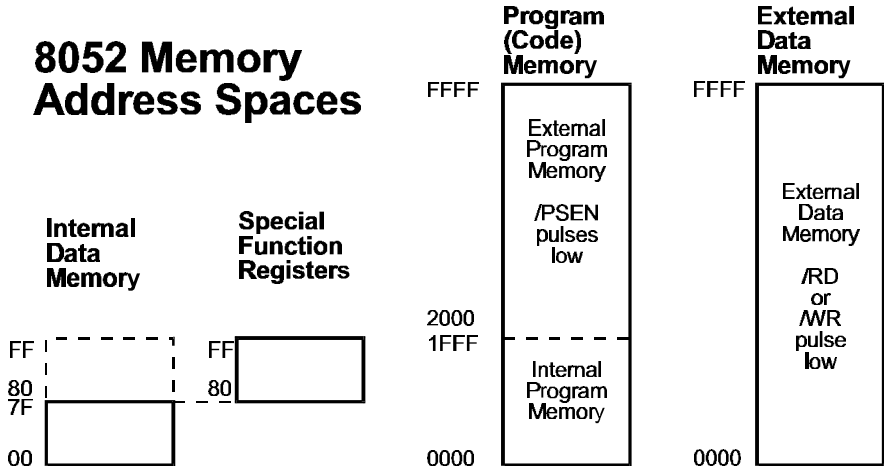


Figure 2-6) 8052 Memory Address Spaces

The lower 128 byte half of the 256 byte internal data memory address space contains four blocks of eight CPU registers, R0-7. In the 8032 CPU, the upper 128 bytes of the internal data memory address space are shared between data memory and the SFRs, depending upon the address mode. The upper 128 bytes of data memory must be accessed using the indirect register 0/1 (@R0 or @R1 operands) or stack accesses, and all other references to addresses of 128 or higher will access the SFRs. All registers except the Program Counter and the four 8-Register Banks reside in the Special Function Register address space. These memory mapped registers include arithmetic registers, pointers, I/O ports, and registers for the interrupt system, timers and serial channel. There are 128 bit locations in the SFR address space that are addressable as bits. The 8031 contains 128 bytes of Internal Data RAM and 20 Special Function Registers (SFRs), while most other processor family variants include an additional 128 bytes of internal data memory overlapped with the SFR addresses.

8051 CPU Hardware

The 8051 is classified as an 8-bit machine, since the internal ROM, RAM, Special Function Registers, Arithmetic/Logic Unit and external data bus are each 8-bits wide. The 8031 is identical to the 8051, except that it does not have any internal program ROM. The 8051 performs operations on bit, nibble, byte and double-byte data types. It excels at bit handling since data transfer, logic and conditional branch operations can be performed directly on the bit addressable SFRs.

This section describes the hardware architecture of the 805l CPU. A detailed 8051 Functional Block Diagram is displayed in the figure below.

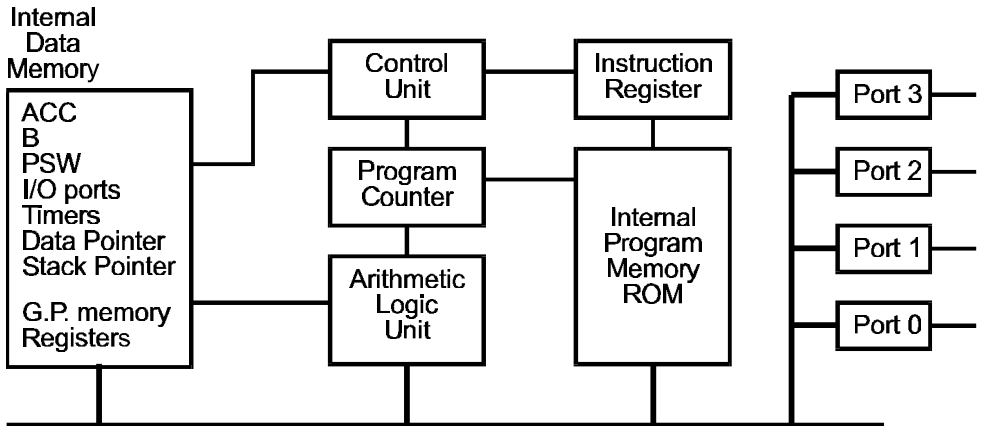


Figure 2-7) 8051 CPU Block Diagram

Control Unit

Each program instruction is decoded by the control unit, which is also called the instruction decoder. This unit generates the internal signals that control the functions of all the other units within the CPU section. All instructions are fetched from the program memory ONLY. Instructions can be fetched from either the internal program memory (for those devices which possess one) or from external program memory. Instruction fetch operations are indicated when the CPU activates (lowers) the /PSEN line (NOT Program Strobe Enable). A program memory fetch cycle lasts as long as /PSEN stays low. External program memory must only drive the data bus with the addressed instruction while /PSEN is low.

Program Counter

This is the pointer to the next instruction to be executed. The 16-bit Program Counter (PC) controls the sequence in which the instructions stored in program memory are executed.

Instruction Register

This is the register that contains the instruction, which is currently being executed.

Internal Program Memory

The 8051 family has 16 address lines, and can directly address 2^{16} = 64K bytes of program memory. The original 8051 has 4K bytes of program memory resident on-chip, the 8031 has no on-chip program memory, the 8052 has 8K bytes of program memory. Other variants of the family are available with 1K to 64K bytes of various types of non-volatile program memory built in. The 64K-byte Program Memory address space is composed of a combination of internal and external program memory (external program memory only on the 8031 and 8032). When external program memory is accessed, and the processor is fetching an instruction to be executed, the external program read cycle is signaled by activating the CPU's /PSEN control line. The MOVC instruction also activates /PSEN to enable reading the code memory into the accumulator for accessing lookup tables and other unchanging data stored in the program memory space.

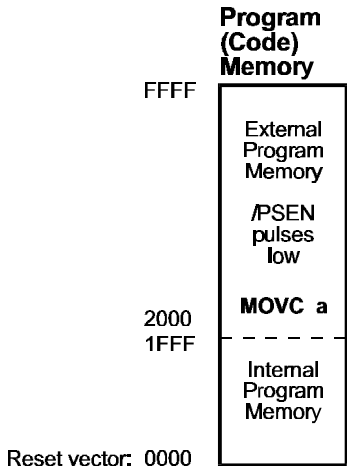


Figure 2-8) Program Memory Map

The processor can fetch instructions from internal or external program memory. There is a control input pin, /EA (External Access), which forces all instructions to be fetched from the external program memory when the pin is pulled low. If the /EA pin is pulled high, then the processor will fetch instructions from any available internal program memory. When the processor first powers up and receives a Reset signal, it begins by executing the instruction at location 0000 in program memory. When the processor fetches instructions from external program memory, it puts the instruction address out on the address bus, pulses the /PSEN (Program Strobe ENable) pin low to enable the external program memory to place the instruction on the data bus to the processor.

The generic part numbering scheme is as follows:

8xxx: NMOS logic
8xCxx: CMOS logic
803x: No internal program memory
805x: Factory programmed internal ROM program memory
87xx: Internal user programmable EPROM program memory
89xx: Internal Flash EPROM program memory
8xx1: 4K internal program memory, 128 byte internal RAM
8xx2: 8K internal program memory, 256 byte internal RAM

For example, the 80C32 used as the standard processor in the SDK board is a CMOS part with no internal program ROM, and 256 bytes of internal data RAM.

Internal Data Memory

The Internal Data RAM provides a convenient 128-byte scratch pad memory which includes the register banks, SFRs, and general-purpose data storage. The programmer (or compiler) may also use this scratch pad memory for storing intermediate calculations on a temporary basis. The 8031 contains a 128-byte Internal Data RAM (addresses 0-7Fh, which includes registers R0-R7 in each of four Banks), in addition to the memory-mapped Special Function Register

(locations 80-FFh). The 8032 has an additional 128 bytes of internal data RAM also at locations 80-FFh, which can only be accessed by using indirect register addressing (@R0, @R1) and the stack. The lower 128 byte half of the 256-byte internal data memory address space contains four blocks of eight CPU registers, R0-7. In the 80x2 CPU, the upper 128 bytes of the internal data memory address space are shared between data memory and the SFRs, depending upon the address mode. The upper 128 bytes of data memory must be accessed using the indirect register 0/1 (@R0 or @R1 operands) or stack accesses, and all other references to addresses of 128 or higher will access the SFRs. All registers except the Program Counter and the four 8-Register Banks reside in the Special Function Register address space. These memory mapped registers include arithmetic registers, pointers, I/O ports, and registers for the interrupt system, timers and serial channel. There are 128 bit locations in the SFR address space that are addressable as bits. The 8031 contains 128 bytes of Internal Data RAM and 20 Special Function Registers (SFRs), while most other processor family variants include an additional 128 bytes of internal data memory overlapped with the SFR addresses.

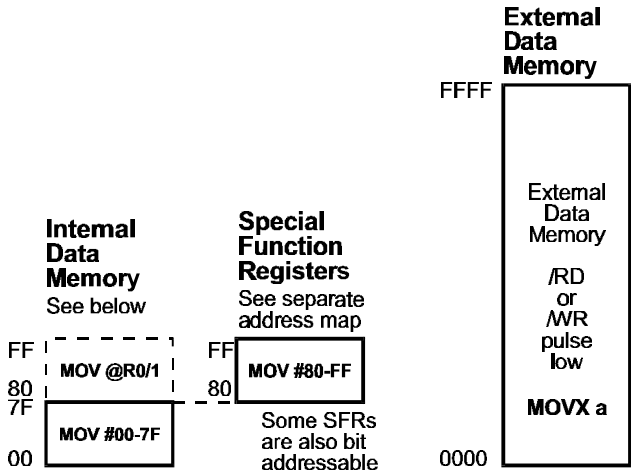


Figure 2-9) Data Memory Address Spaces in the 8051

Data Memory

The 8051 family parts have two data memories, internal and external. With 16 address bits, there is a maximum of 64K-bytes of External Data Memory, which is useful for storing large blocks of variable information which will not fit in the internal data RAM. It is enabled when the processor reads or writes data from the external data memory, signaled by activating the /RD and /WR control lines. This occurs only when a MOVX instruction is used to read or write from external memory.

The Internal Data Address space has two different parts. One part contains the general-purpose registers and general-purpose data storage RAM, and the other part contains all the special registers and I/O devices, such as the parallel and serial ports, and timers. These registers are called Special Function Registers. There is a maximum of 256 bytes of Internal RAM (128 bytes for the '31/'51, 256 bytes for the '32/'52) and Special Function Registers (SFR). Four Register Banks (each bank has eight registers), 128 individually addressable memory bits, and the stack reside in the Internal Data RAM. The stack depth is limited only by the available Internal Data RAM. The 8-bit stack pointer determines the stack's location.

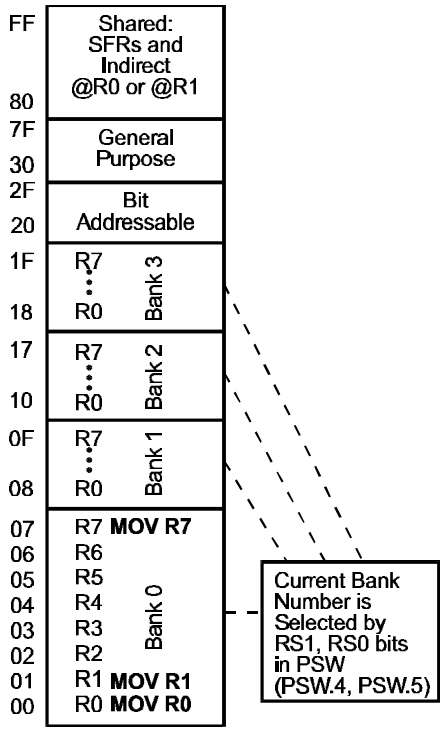


Figure 2-10) The Internal Data Memory

The Internal Data RAM provides a convenient 128-byte scratch pad memory which includes the register banks, SFRs, and general-purpose data storage.

RAM locations 00-7F hex

Register Banks - There are four Register Banks within the Internal Data RAM. Each Register Bank contains registers R7-R0,

128 Addressable RAM Bits - In the 8031, there are 128 addressable software flags in the Internal Data RAM. They are located in the 16 byte locations starting at byte address 20h and ending with byte location 2Fh of the RAM address space.

Special Function Register (SFR) locations 80-FF hex

General Registers A, B, and other registers are mapped here.

Parallel I/O Ports - The 8031 has four eight bit ports.

Serial I/O Port - The serial I/O port built into the 8031.

Timer/Counters - There are counters which can count external events, or count processor clock cycles to operate as timers.

Many of the SFRs are also bit addressable.

Bit Addressable Memory

The bit address space has a total of 256 possible bit addresses. The first 128 bits, 00 to 7F hex, are used to access individual bits of the internal memory from location 20 to 2F hex. The second 128 bits, from 80 to FF hex, allow selected bits in the special function registers to be accessed at the bit level. Not all SFRs are bit addressable, and not all bit addresses are used in most processors.

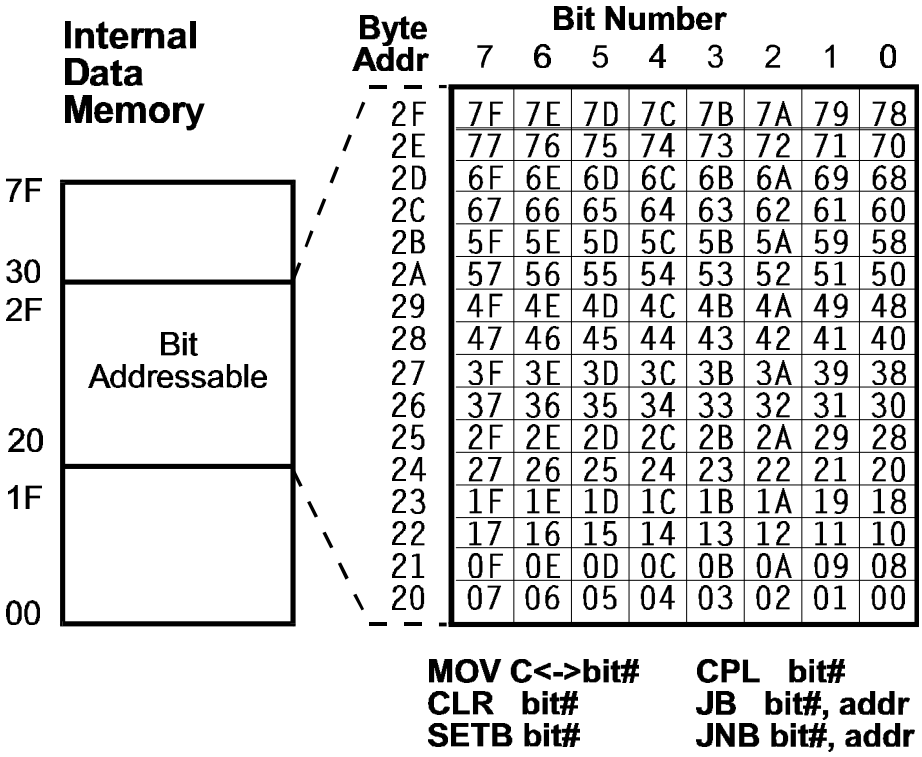


Figure 2-11) Bit Addressable Space in the Internal Data Memory

Bit addressable memory allows the manipulation and test of individual bits, which is a very common operation in embedded systems. Almost every application requires that some output bits be used to control an On/Off device, such as an indicator or relay. Likewise input bits are used to sense the status of some external device, such as a switch or sensor. The bit addressable address space allows the programmer to operate on information at the bit level just as easily as at the byte level. This is contrasted by most other processors, in which the programmer must

write multiple instructions to select the appropriate bit in a byte before processing or testing it.

Internal memory locations from 20 to 2F hex, are accessible either one byte at a time, or one bit at a time. That makes it easy to convert inherently serial information to parallel and vice versa, and to perform Boolean logic functions. This bit-level processing is one of the most unique and powerful features of the 8051 family architecture, and is one of the features that differentiates it from other microcontrollers.

Register Banks
The four Register Banks within the Internal Data RAM each contain eight registers named R0-R7.

128 Addressable Bits
There are 128 addressable software flags in the Internal Data RAM. They are located in the 16 byte locations starting at byte address 20h and ending with byte location 2Fh of the RAM address space.

I/O Ports
There are four eight bit ports. When using external program or data memory, only Port 1 (P1) is available for general purpose I/O. External memory uses Port 0 (P0) for the multiplexed data bus and address bits 0-7, and Port 2 (P2) for address bits 8-15, while Port 3 (P3) contains special control signals, such as the read and write strobe pins. In addition to the basic parallel I/O bits on the four ports, some of the port bits have alternate functions. The alternate functions include the serial I/O port signals, timer and interrupt inputs.

Timer/Counter
The 8031 has two timer/counters and the 8032 has three.

Serial I/O
The serial I/O port that is built into the 8031 can be used to transmit and receive asynchronous (un-clocked) serial data, as is used on a PC's serial port. It can also be used for synchronous (clocked) serial data transfers.

Reset Circuitry
The reset input pin should be connected to an external resistor and capacitor, so that the processor will be properly initialized upon initial application of power. There is a capacitor between the reset pin and the power supply, and a resistor from the reset pin to ground. When power is first applied, the capacitor has no

voltage across it, forcing the processor to reset. After resistor R1 charges the capacitor C, the reset signal goes low (inactive), and the processor begins executing the program beginning at location 0 in program memory. The recommended reset circuit is shown below.

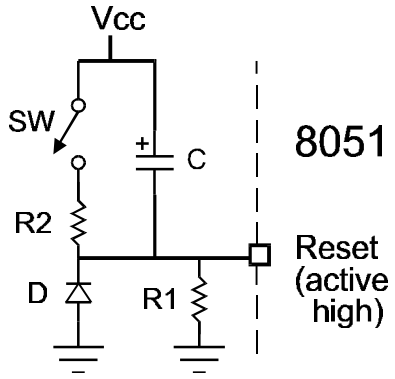


Figure 2-12) Recommended Reset Circuit for the 8051

When power is first applied, capacitor C has zero voltage across it, and reset is held high until the current that flows through R1 charges C. Once the capacitor is charged, the reset pin is at zero volts and inactive. The diode allows the capacitor to discharge when Vcc goes to zero, even for a short period. If there was no diode, and there was a brief power loss, the CPU state would be indeterminate, and would not be reset. Optionally, closing switch SW through a series resistor R2, which limits the current through the switch, can reset the processor. The current flowing through the switch discharges the capacitor. If resistor R2 was not present, very high currents could flow through the switch. These high currents which flow very briefly while the capacitor is shorted, can cause the switch contacts to fail or even weld the contacts together.

The R1*C time constant must be long enough to guarantee that the processor will be completely reset to a known state upon power up. The delay must allow the oscillator to start up and stabilize, as well as the time it takes the processor to reset after the oscillator is stable. Different processors require different numbers of clock cycles to reset themselves, and the oscillator start-up time can vary widely depending on the frequency reference, voltage, capacitive loads, and other factors. If the processor reset is not long enough, the processor may behave in unpredictable ways, and it may not be apparent that the problem is due to an incomplete reset operation. In most cases, it's better to have a relatively long reset time constant, on the order of 100s of milliseconds, to be sure that the processor has been completely reset. External peripherals can also exhibit this problem. During the initial development of the SDK, we experienced occasional problems with the external serial port chip used on the board. The problem turn out to be related to the length of the reset pulse and the period of time *after* the reset when the chip must be left alone to pull itself together! This sort of problem can be very difficult to trace down, since it is difficult if not impossible to determine when a chip has not been completely reset.

The 8051 is unique in that its reset signal is active high. Other processors use active low reset signals, so the reset circuit must be adjusted to perform the equivalent function with the reset pulse going low at power up and when the capacitor is charged, the reset goes high. The circuit configuration except R and C1 are swapped, as are D1 and the SW/R2 pair.

The circuit above is good enough for most applications, however it is not foolproof. Even with the above precautions, it is possible that power transients which are too short to cause a reset can jumble the processor state. When a processor is used in a critical or long term unattended application, that probably won't be good enough to meet the need for reliable operation. To deal with this, processor supervisory chips are available to monitor the power supply voltage for out of tolerance fluctuations and automatically reset the processor when the power supply falls out of tolerance. Some of these supervisory chips also have a special "watchdog" timer circuit that expects to be "fed" by a pulse which resets the watchdog counter periodically by a correctly functioning program running on the processor. If the watchdog timer is not "fed" with a pulse periodically, the counter will overflow and it will "bark" by pulling the reset pin active. That way if the processor goes off in the weeds, due to a hardware glitch or a program bug, the CPU will be reset. This is a simple method of obtaining tolerance to fault conditions, but it also requires careful design to avoid undesired reset conditions. It is also the designer's responsibility to assure that the processor can't get stuck in a loop while feeding the watchdog timer.

When designing a microcontroller which must operate in high noise environments, or where correct operation is safety critical, special care must be taken to ensure that electromagnetic noise does not cause problems. This noise can come from other parts of the system and environmental conditions such as electromagnetic fields from other devices such as wireless communication devices. With the rapid increase in the number of electronic and wireless devices, this problem is becoming more and more serious. The field of Electromagnetic Compatibility (EMC) covers this noise, as well as others such as Electrostatic Discharge (ESD). A good summary of EMC concepts as they relate to microcontrollers can be found in the Intel application note AP-125, "Designing Microcontroller Systems for Electrically Noisy Environments."

Oscillator and Timing Circuitry

Timing generation is completely self-contained on the 8051, except for the frequency reference, which can be a crystal or external clock source. The on-board oscillator is a parallel anti-resonant circuit with a frequency range of 1.2 MHz to 12 MHz for the original 8051. There is a divide-by-12 internal clock counter that gives the standard 8051 an instruction cycle of 1 uS with a 12 MHz crystal. Higher speed versions of the processor are also available, which use fewer than twelve clocks per instruction cycle. The Dallas 80C320 uses only four clock cycles for most instruction cycles, so it is three times faster than the original CPU

using the same clock frequency. The XTAL2 pin is the output of a high-gain amplifier while XTAL1 is its input. A crystal connected between XTAL1 and XTAL2 provides the feedback and phase shift required for oscillation. For stability and consistent oscillator start-up, two capacitors in the range of 10 to 20 picofarads should be connected from the XTAL pins to ground. If XTAL1 is being driven by an external frequency source, XTAL2 should not be connected. An external clock can also be applied to XTAL1 to allow the use of a separate clock frequency source, such as an oscillator module.

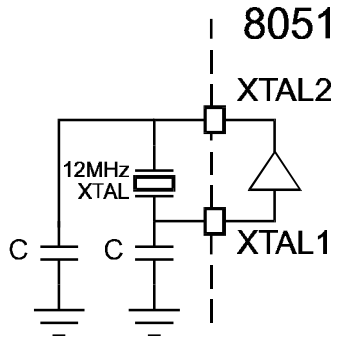


Figure 2-13) Standard Oscillator Configuration

The oscillator circuit consists of a crystal connected between the XTAL1 and XTAL 2 pins of the processor, along with two capacitors, one from each XTAL pin to ground to improve stability and start-up characteristics of the oscillator. The internal amplifier and quartz crystal form a series resonant oscillator which operates at the at the crystal's resonance frequency. The amplifier in the original 8051 was an inverting amplifier, but other variants and other processor families make use of non-inverting amplifiers in some cases. All of the processor's timing is derived from this oscillator. For the standard 8051 compatible parts, each instruction cycle requires a multiple of 12 clock cycles. For the Dallas high speed CPU versions, 4 clock cycles are used for most instruction cycles.

In most 8051 designs, the capacitors connected to the crystal should be in the 10 to 50 pF range 30 pF being a typical value. The crystal should be an "AT cut" series resonant device. The "AT" designation refers to the way the quartz crystal is cut from the blank with an orientation relative to the crystal lattice that reduces the crystal's frequency dependence on temperature changes. The crystal is manufactured so that it is series resonant at the specified frequency. A given crystal will resonate in a series or parallel mode. A parallel resonant crystal will still operate in the circuit, but it will operate at a slightly different frequency. Actual operating frequency depends on the load capacitance, and is subject to temperature, and will drift over time.

Selection of the capacitors is a trade-off between oscillator start-up time and stability. Specification of a crystal depends upon the specific design requirements and the processor being used. Even parts with the same number may have different requirements, especially for parts from different manufacturers.

There's a lot more information available from the crystal and processor manufacturers on the proper design and operation of crystal oscillators. Other frequency references, such as ceramic resonators and even simple R-C circuits can be used for many processors. Some microcontrollers even include on-chip oscillators that can be calibrated to operate at a specific frequency, albeit with less accuracy and greater drift. Application note AP-155, "Oscillators for Microcontrollers" from Intel Corporation, is a very useful reference and describes the characteristics of the crystal and ceramic resonator's operation, as well as the processor's oscillator amplifier.

The 8051 Microcontroller Instruction Set Summary

The following description of the instruction set is not a complete list, but serves to introduce the general character of the standard 8051 instructions. The instruction set utilized by the 8051 microcontroller consists of a total of 111 instructions, which may be divided up into several different categories. These are:

1. Arithmetic (24)
2. Logical (25)
3. Data transfer (28)
4. Bit (Boolean) variable manipulation (17)
5. Program branching and control (17)

Each of these categories is comprised of instructions that utilize **mnemonics** as shown below:

Arithmetic

ADD, ADDC, SUBB, INC, DEC, MUL, DIV, DA

Logical

ANL, ORL, XRL, CLR, CPL, RL, RLC, RR, RRC, SWAP

Data Transfer

MOV, MOVX, MOVC, PUSH, POP, XCH, XCHD

Bit (Boolean) Variable Manipulation

CLR, SETB, CPL, ANL, ORL, MOV, JC, JNC, JB, JNB, JBC

Program Branching and Control
ACALL, LCALL, RET, RETI, AJMP, LJMP, SJMP, JMP, JZ, JNZ, CJNE, DJNZ, NOP

Direct and Register Addressing
While the number of mnemonics is clearly smaller in number than the total of 111 instructions, a given mnemonic may be used in several different ways to make up a valid 8051 instruction. These different ways of forming instructions are classified by the types of **arguments** that a given mnemonic takes. A mnemonic can refer to data in a number of ways. One can refer to data located in particular address in the **Data Memory** space either by specifying its address directly, or indirectly by using a **data pointer register.** In this case, the data pointer register contains the address of the memory location we seek. The 8051 looks in the data pointer register, and then retrieves the information located in the location referred to (or **pointed to**) by the data pointer. Additionally, the 8051 has 32 bytes of internal memory divided up into four **Register Banks** of eight bytes each. These register banks may be referred to in an 8051 instruction by either their direct address (which ranges between 00h and 1Fh), or by their **Register Name**, which is denoted by **R0** through **R7**. When these memory locations are addressed by their register name, it is important to remember which register bank is currently in use. These register banks, numbered 0 through 3, are selected through two bits located in a special register called the **Program Status Word**, or **PSW.** The PSW contains a number of very important bits, which are used to indicate the current status of the processor. Note that because the registers R0 through R7 are located in the data memory space, they may be addressed either by the register name or by their direct address location. Consider the instruction:

```
MOV A,R3
```

This instruction takes the contents of register R3 and moves it (actually, the data is copied) to a register denoted by the letter "A", called the **Accumulator**. The accumulator is the "working" register of the 8051, and is the register that is used in most all arithmetic and logical operations performed by the processor.

Assuming we are using register bank 0, the following instruction is identical to the instruction just shown:

```
MOV A,03h
```

Since register R3 is at internal RAM location 03h, the above instruction takes the data stored in RAM location 03h and moves it to the accumulator.

What is the difference between these two forms of saying the same thing? The first instruction is called **Register Addressing**, while the second instruction is called **Direct Addressing**. The reason for the difference in nomenclature is

obvious, and while it may seem a bit pointless to dwell on the difference between these two modes, there is a significant difference in the way the 8051 deals with each type of addressing.

Looking in the *80C51-Based 8-Bit Microcontrollers* Data Book (publication number IC-20) published by Philips, the instruction MOV A,R3 takes up only one byte of program memory space, while the instruction MOV A,03h requires two bytes of program memory space. The reason the register mode instruction requires less program memory to store is that **a reference to a register requires three bits to represent its address, and a reference to an arbitrary location in internal data memory requires 8 bits.** Once a particular register bank is selected by setting the proper bits in the PSW, any register in that bank may be completely determined by only 3 bits (3 bits are required to distinguish 8 possible locations). If we use direct mode to perform the very same operation, we now require 7 bits to completely determine the exact location out of 128 possible locations – thus, direct addressing instructions generally occupy more program memory space than register addressing instructions.

There are two other memory locations in the 8051 that may be addressed through register mode. These are the **Accumulator**, which we have already seen is denoted by the letter "A", and the **Data Pointer**, which is actually two registers. The letters DPTR denotes the data pointer, and is a 16-bit quantity used for addressing locations in data memory external to the microcontroller itself. Since the DPTR is a 16-bit quantity, a total of 64Kbytes of data may be addressed. This is, of course, the maximum data that may be accessed at any one time by the 8051.

The following instructions are examples of data movement instructions which utilize **direct addressing**:

```
MOV 24h,A    ;move accumulator contents to internal RAM location 24h

MOV 7Ch,0Fh ;move location 0Fh contents to internal RAM location 7Ch

PUSH 22h     ;PUSH location 22h contents onto the stack

POP 4Eh      ;POP the top of the stack into location 4Eh
```

The following instructions are examples of data movement instructions, which utilize **register addressing**:

```
MOV R0,49h  ;move location 49h to register R0

MOV R2,A    ;move accumulator contents to register R2
```

Note that in all instructions, the order of the memory locations in all instructions is always **destination, source**. The destination address appears first, followed by the source address.

The instructions PUSH and POP perform operations on a portion of memory called the **stack**. While not a separate memory space, the stack is located in the internal data memory portion of the 8051/52, and is structured as a **LIFO** (Last In, First Out) data structure.

The instruction:

```
PUSH 49h
```

Takes the data stored in internal RAM location 49h and puts it onto the top (that is, the first available location) of the stack. Exactly where the top of the stack is situated is determined by the value contained in the **SP** (Stack Pointer) Special Function Register. When the processor executes a PUSH instruction like the one above, it first increments the SP register by 1, and then copies the internal RAM register specified in the PUSH instruction to the address pointed to by the SP register. In other words, the value contained by the SP register is a pointer to the memory location **one byte below the top of the stack**.

The POP instruction takes the data at the top of the stack and copies it to the internal RAM location specified in the POP instruction. After copying the data, the SP is decremented by 1. The SP register in the 8051/52 is therefore a **pre-increment, post-decrement** register. In the 8051, which contains 128 bytes of internal data RAM, the maximum legal value that the SP register may contain is 07Fh. The 8052 has an additional 128 bytes of internal RAM, separate from the Special Function Registers. This section of RAM is accessible through the stack, and so the 8052 permits a maximum value of the SP register of 0FFh.

The SP register can be set by the programmer to any value that is convenient for the particular application. When the processor comes out of RESET, the SP register is loaded with 07h, thus placing the top of the stack at internal RAM location 08h. This is just above Register Bank 0. The stack always grows **upwards** through internal RAM. Care must be taken that the stack does not collide with other registers in internal RAM which have other uses. Additionally, if the SP register reaches its maximum value, 0FFh, and then overflows, the stack will continue to grow through the Bank 0 registers. As no stack overflow or underflow features are present on the 8051, this becomes the responsibility of the programmer.

Indirect Addressing

In many applications, it is inconvenient or impossible to always refer to data directly or as a register. When large amounts of data are being manipulated, either in internal or external data memory, very often it is required to address such data

through the use of a **data pointer**. Use of a data pointer to address data memory is known as **indirect addressing**. The 8051 has four different methods by which data may be addressed indirectly:

1. The Indirect registers R0 and R1, located in each of the 4 register banks
2. The Data Pointer DPTR and the Accumulator
3. The Program Counter and the Accumulator
4. The XCHD instruction

Indirect addressing of data is used frequently. Many embedded applications require calculation of one form or another, and frequently the most efficient means of doing this is through the use of a "look-up table." As an example, an 8051 microcontroller such as the 80C552 has an eight channel, 10-bit analog to digital converter (ADC). The ADC takes an analog voltage as its input, and converts it to a 10-bit binary number between 000h and 3FFh. If this ADC is used, for example, to convert the analog output voltage of a pressure transducer to a digital value, it is necessary to relate each of the 1024 possible counts of the ADC to a pressure value. If the computer in use is very fast, or has a great deal of floating point mathematical ability, it would be possible to directly calculate the pressure value from the ADC count – one would need the characteristics of the transducer to accomplish this. However, an 8-bit embedded controller such as the 8051 does not have such capability, or at least the ability to do complex mathematical calculations quickly. In this case, it is far more efficient to simply generate the 1024 numbers that correspond to the pressure output of the transducer and store these numbers in a table. The processor then takes the output of the ADC and uses this 10-bit number as an **offset** into the table stored in RAM. This offset, when added to the **base address** of the lookup table (the base address is the address of the first record in the table), constitutes the physical address of the data record which corresponds to the actual pressure sensed by the transducer. Since this lookup table could be located literally anywhere in either the code or data memory spaces, and since each record could be more than a single byte, it is in general not possible to store the actual location of each entry in the table. Rather, the ADC output is used to **indirectly address** the data through the use of a data pointer.

Registers R0 and R1 in each of the four register banks may be used to indirectly access any of the internal data memory space of the 8051. To illustrate by example, consider the instruction:

```
MOV A,@R1
```

Here, the "@" symbol is used to denote indirection, similar to the asterisk "*" in C. This instruction takes the data located in the location **pointed to** by register R1 and copies it to the accumulator. Note that the value copied to the accumulator is **not** the contents of R1, but the value in the memory location equal to the contents of R1. This is why register R0 is said to be a **data pointer,** pointing to another internal RAM location. Notice that only data located in the internal data memory

space of the 8051 may be accessed through @r0 or @R1 instructions. As these registers are only 8 bits wide, a total of 256 bytes may be specified. The 8051 microcontroller contains a total of 128 bytes of internal RAM located between addresses 00h and 7Fh, while the 8052 contains an additional 128 bytes of internal RAM between 80h and 0FFh. These upper 128 bytes of internal RAM can **only** be accessed by indirect addressing. It is important to distinguish these upper 128 bytes of internal RAM in the 8052 microcontroller from the Special Function Registers. The SFRs **are not** part of the upper 128 bytes of internal RAM – they are a separate memory space.

Very often, an embedded system will require a much larger amount of RAM than is present on an 8051 or 8052 microcontroller. When this is the case, one generally uses external RAM chips that are interfaced to the processor over the address, data, and control bus structure. Since the address bus of the 8051/52 microcontroller family is 16 bits wide, a total of 64Kbytes of either program memory or data memory may be accessed. Restricting our attention to the data memory space and RAM for the moment, we need some way of accessing the (at most) 64Kbytes of RAM external to the microcontroller. The **MOVX** instruction (X denotes "External") is used to move data into and out of RAM located external to the microcontroller. The **only** way the 8051/52 microcontroller can access external RAM is through indirect addressing.

The MOVX instruction can be used in two different ways. If the external RAM space is small (small meaning 256 bytes or less in this case), it may be accessed with an 8-bit address. The R0 and R1 registers may be used in this manner just as they are used for indirect addressing of internal RAM. Consider the instruction:

```
MOVX @R0,A
```

This instruction takes the byte in the accumulator and copies it to the location at the address in external RAM pointed to by R0.

The instruction

```
MOVX @R1,A
```

performs the opposite operation. It takes the value held in the external RAM location pointed to by R1 and copies it to the accumulator. There is an important difference between this type of external data addressing and internal data addressing – whenever data is being read from or written to external RAM, either the source or the destination register **must** be the accumulator.

What if our external RAM array contains more than 256 bytes? Recall that the 8051/52 family of microcontrollers have a 16-bit address bus, permitting up to 64Kbytes of external program and/or data memory. The Data Pointer (DPTR) is

used to store a 16-bit address for indirect addressing of external RAM. DPTR is loaded with the address of interest, and the instruction

MOVX A,@DPTR

Copies the data at the external RAM location pointed to by the 16 bit address pointer, called DPTR into the accumulator. The instruction

MOVX @DPTR,A

performs the opposite operation. The contents of the accumulator A is copied to external RAM at the location pointed to by DPTR. Again it is important to notice that either the source or the destination register in the instruction **must** be the accumulator.

Sometimes it is necessary to store information other than actual program instructions in a nonvolatile memory. Critical configuration data, lookup tables, or serial number information for unit identification oftentimes must be stored and available at system power-up without having to be regenerated by the system itself. While there are external nonvolatile memory technologies available (EEPROM and FLASH, for example), it is possible to use the program memory space of the 8051/52 for this same purpose. While it is not possible to **write** to the program memory space during normal operation (that could have potentially disastrous results!), it **is** possible to read data from it. The **MOVC** instruction ("C" denotes "Code") copies a byte in the program memory space to the accumulator. In order to accomplish this, the instruction requires the use of a base address and an offset. It is best to illustrate this with some examples. The two allowable forms of the MOVC instruction are:

MOVC A,@A+DPTR

MOVC A,@A+PC

In each of these instructions, the contents of the accumulator and either the DPTR or the PC (the Program Counter register) are added together, generating a 16-bit address. The contents of the address in the program memory space pointed to by this 16-bit sum is copied to the accumulator. In this way, either the PC or the DPTR can be used as a **base address** into a data table in the program memory space. The accumulator then becomes an **offset** into the data table, with a maximum offset value of 256.

The last method of indirect addressing available in the 8051 is the **XCHD** (Exchange Digit) instruction. The XCHD instruction is frequently used when BCD (binary coded decimal) arithmetic is being performed, or when a BCD lookup table is stored in internal RAM (a common use of a BCD lookup table

would be for driving a 7-segment LED display). The XCHD instruction has the following syntax:

```
XCHD A,@R0

XCHD A,@R1
```

This instruction exchanges the low nibble (that is, the low 4 bits) of the accumulator with the low nibble of the internal RAM location pointed to by R0 or R1 registers. Recalling that BCD uses 4 bits to represent the decimal numbers 0 through 9, this instruction offers a quick way to indirectly address a BCD (or any other 4-bit coding scheme) lookup table in internal RAM. To illustrate this with an example: suppose the accumulator contains A6h, register R1 contains 43h, and internal RAM location 43h contains 0BBh.

The instruction:

```
XCHD A,@R1
```

Will result in the accumulator containing 0ABh, and internal RAM location 43h containing 0B6h.

Immediate Addressing

Sometimes it is necessary to place a fixed constant into a memory location. This may be performed through the use of the **immediate** operator "#". As an example:

```
MOV A,#09h
```

Places the **number** 09h into the accumulator. Likewise:

```
MOV 52h,#3Ah
```

Places the constant 3Ah into internal RAM location 52h. The immediate operator indicates that the number, which follows, is to be interpreted as an **immediate constant**, rather than a memory location. Notice that, had we issued the instruction:

```
MOV 52h,3Ah
```

This would have copied the contents of internal RAM location 3Ah to internal RAM location 52h. Since this is a perfectly valid 8051 instruction, the assembler will not flag this as an error if we had actually meant to prefix the 3Ah with the immediate operator. The code will not function as we might expect it to operate. **Watch out for this – it is a VERY common error!**

Immediate data, by its very nature, **must** only occur as the source operand of an 8051 instruction.

The instruction:

```
MOV #52h,44h
```

Makes no sense, and will be flagged as an error by the assembler. ON the other hand, below is a valid instruction that will put the number 44h into internal data RAM location 52h.

```
MOV 52h,#44h
```

A detailed list of all of the instructions and their operations is contained in the 8051 programmer's reference handbook.

Generic Address Modes and Instruction Formats

Regardless of the type of processor, certain address modes are usually available in one form or another. This section describes some of the generic address mechanisms and instruction encoding formats, using the 8051 instructions and address modes as an example.

The number of operand addresses that are explicitly specified in an instruction can classify instructions. For example, "CPL A -- complement accumulator" is an instruction that does not contain an explicit address, so it is a **zero-address instruction**. The accumulator is called an **implied operand** because the instruction op code does not have an address field, since this instruction always refers to the accumulator. An **explicit operand** has an address field embedded in the instruction op code or follows the op code, usually as a pointer to the data that is to be used. Other examples of zero-address instructions include PUSH, POP and RETurn because the operand is implied to be on the stack. The instruction "MOV A,address" -- load accumulator with the content of internal memory location address" is a **one-address instruction** because the accumulator is an implied address, but the memory location is specified explicitly by its address. A **two-address instruction,** such as "MOV addr,@R0" - move the data at address pointed to by R0 to "addr," has two address fields. Some processors have **three-address instructions**, which allow the processor to perform an operation on two operands and store the result in a third operand, all of which may be referred to explicitly.

Instructions for a typical 8 bit CPU might consist of one or more op code bytes followed by optional operand fields. The first (op code) byte would identify the type of instruction, and the optional byte(s) following it would be the operand(s) or addresses of the operand(s).

8051 ADDRESS MODES

Implied addressing, as described above, always references the same location and does not have an explicit address field in the instruction. The instruction shown would take only one byte, and would result in only one memory cycle to fetch the op code byte.

Implied addressing

Instruction	Operand
CPL complement	A accumulator
E4 (op code)	A (implied)

Immediate addressing is used when the operand is a constant value, and is part of the instruction, usually immediately following the op code in program memory. An example would be an instruction that loads a constant into the accumulator, as shown below.

Immediate addressing

Instruction	Operand
MOV A, Load accumulator	#35H with 35 hex
74 (op code)	35 (constant)

The instruction would be stored in an eight bit processor's memory as follows:

Address	Value(hex)
1000	74 op code
1001	35 operand

Execution of this instruction would result in two memory cycles, one to fetch the op code and one to fetch the constant.

Direct addressing includes the address of the operand as part of the instruction rather than the operand itself. The address part of the instruction acts as a pointer to the data to be accessed. An instruction that loads the byte of data stored in memory location 1234 into the accumulator would consist of the op code followed by the address 1234.

Direct addressing

Instruction	Operand
MOV A, load accumulator	34H with the contents of location 34
E5 (op code)	34 (constant)

The instruction could be stored in an eight bit processor's memory as follows:

Address	Value(hex)
1000	E5 op code
1001	34 operand address

Execution of this instruction would result in three memory cycles, one to fetch the op code and one to fetch the address and one to fetch the byte at location 1234.

When dealing with values of more than 8 bits, different microprocessor vendors use different methods of storing data in memory. Of course, Intel and Motorola chose opposite methods. The 16 bit address stored high byte first followed by the

low byte as it is done in the Motorola 68000 family. Other processors, such as the Intel CPUs, reverse the order. For machines that support two-byte or four-byte values, there are two different ways of storing the bit operands: low byte first (Intel), and high byte first (Motorola).

Indirect addressing specifies a memory address that contains the address of the data to be transferred. An instruction that loads the byte of data that is pointed to by the address stored in memory location whose address (1234h) resides in the 16 bit register DPTR into the accumulator is shown below.

Indirect addressing

Instruction	**Operand**
MOVX A, load accumulator	@DPTR contains the address of the byte to be accessed
E0 (op code)	DPTR=1234h (address of the operand)

The instruction could be stored in an eight bit processor's memory as follows, assuming that DPTR contains 1234h:

		External Memory	
Code Address	Value(hex)	Data address	Value
1000	E0 op code	1234	57 operand

After completion of this instruction, the value 57 would be left in the accumulator. Execution of this instruction would result in two memory cycles, one to fetch the op code (E0), one to fetch the content (57) of the address (1234).

The 8051 does not support true Indirect Addressing. In processors that do, the address of the operand is stored at the location contained in the instruction op code.

Register indirect addressing (e.g. MOV A,@R1) uses the contents of a register as an address, so the instruction would consist of only an op code byte. A register points to the operand in memory, so there is no need for an address field in the instruction. Two memory cycles are needed, one for instruction fetch and one for fetching the data.

Indexed addressing (e.g. MOVC A,@A+DPTR) is a combination of direct and register indirect addressing, because the instruction includes an offset address (DPTR), which is added to an index register (A register) to determine the address of the data to transfer.

It should be noted that the nomenclature for the various address modes varies, and the 8051 family address modes used for the examples above are not necessarily the

best examples, as other processors support more extensive and flexible address modes.

The Software Development Cycle

The standard software development process consists of the following steps:

1. Create or edit an ASCII text file containing the human readable source code, also known as the program instructions.
2. Translate the source code to machine-readable binary instruction code using a language translator. This is accomplished using an assembler or compiler.
3. Load the program memory with the binary instruction code into the processor's program memory chip. For the SDK, the program is downloaded into program memory on the SDK.
4. Execute the program to test it and find program errors. For the SDK, this "debugging" process is facilitated using a special program (debugger or monitor) resident on the SDK.
5. Once the problem is located, the source code is corrected by repeating this process until all errors are corrected.

Software Development Tools

Software tools include translators, like assemblers and compilers, and debugging tools. Active debugging tools include: In-Circuit Emulators (ICE) for HW/SW integration are plugged into the application circuit (the “target” system) in place of the CPU, allowing the designer to “see inside” the microcontroller, download, and execute programs selectively. ROM Emulators (ROM ICE) allow the designer to reduce the time it takes to edit-compile-load-debug programs by replacing the program EPROM with a RAM that can be loaded quickly and easily from the host computer. Simple tools, such as an LED or speaker can also be useful in debugging.

Hardware Development Tools

There are two general classes of hardware development tools available to the embedded developer: passive analysis tools which allow looking at the operation of the system, and active tools. Active tools allow the designer to intrude on the operation of the system while it’s running, even making changes to the system’s configuration and software while it is under test. The system under test is usually referred to as the “target” system, and the computer that is used to develop, edit, compile, assemble, and download the code to the target system is called the “host” system.

Hardware tools include: Logic probe displays static logic levels and detect pulses, Oscilloscope to look at signal waveform amplitude vs. time, Logic analyzer, with processor specific probes, a PROM programmer.

Chapter 2 Problems

1. Processors such as the 8031 use multiplexed address/data buses. They require more than one clock cycle to do a memory transfer because some or all of the bus lines are shared. 16 bit addresses alternate with 8 bit data. The ALE (Address Latch Enable) signal indicates when address information (A0-7) is present on the multiplexed address/data bus. The ALE signal is used to latch the least significant 8 bits of the address in an 8-bit register. A minimum of two clock cycles is required to transfer data: one for latching the address when ALE is high, and one for the actual data transfer. How many clock cycles (minimum) would be required if the processor was a 16 bit machine doing a 16 bit transfer? Would the address latch have to be different?

2. How many unique locations could be referenced as "address zero" in the 8031 CPU architecture? (Remember to consider *all* the address spaces!)

3. Most processor control lines are active low. Comment on the reasons for this.

3 Worst Case Timing, Loading, Analysis and Design

Introduction to Timing Analysis

Just as it is in comedy, timing is essential to the success of a microcomputer design. Often it is quite possible to get *one* system functioning by just interconnecting the various components. It is *significantly* more difficult to be able to guarantee that many systems will work under the entire range of possible conditions that they may be exposed to. There are many designs in production right now that have a number of unidentified failures due to the lack of a worst case analysis of the design. When timing or loading problems show up in a design, they usually appear as intermittent failures or as sensitivity to power supply fluctuations, temperature changes, and so on.

A **worst case design** takes into account all available information regarding the components to be used with respect to variations in performance. Even when all parameters are at their most adverse values, the worst case design can still be proved to meet the specifications. These variants may be due to changing manufacturing conditions, temperature, voltage, and other variables. Without performing a detailed analysis, there is no way of knowing if the design will work reliably under all operating conditions. It is much better to design reliability and simplicity of manufacturing a product using worst case design rules rather than to attempt to correct a problem after the design has been implemented. With the emphasis that must be given to the quality of the final product, a designer is obligated to perform a detailed examination of the timing in a system. As is the case in most quality improvements, these efforts result in direct cost and time savings. *This is clearly one of the places where the designer can have the greatest impact on overall product quality.*

Timing Diagram Notation Conventions

Timing notation is documented in the following figure. The timing notation used in manufacturers' data sheets may vary from this, but is usually very similar. It is also important to notice that while the diagrams are reasonably standard, there is a wide variation in the selection of symbols for each timing parameter.

The purpose of timing analysis is to determine the sequence of events in each of the bus cycles so that we can delimit, among other things, the time available for each of the components to respond to changes. This time is compared to the requirements as specified in the manufacturers' data sheets to determine if they are compatible, and by what margin.

The most important timing specifications for interfacing components to a bus oriented design are:

- **Rise/Fall time**
- **Propagation Delay time**
- **Setup time**
- **Hold time**
- **Tri-state Enable and Disable delays**
- **Pulse Width**
- **Clock Frequency**

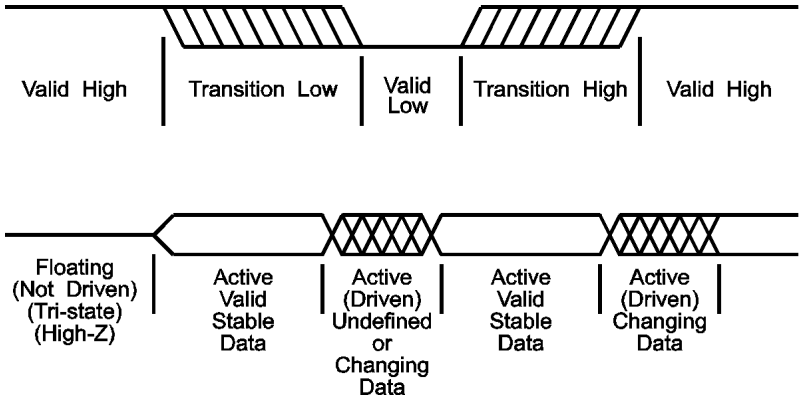


Figure 3-1) Timing diagram notation as used in this book

There are two general classes of logic: **combinatorial logic** and **sequential logic**. Combinatorial logic has no memory and its output is some logical function of its current inputs, after some delay. Examples of combinatorial logic include gates, buffers, inverters, multiplexers, and decoders. Sequential logic has memory, which means that its outputs are a function of both current and past inputs. Examples of sequential logic are flip-flops, registers, microprocessors, and counters. There are two types of sequential logic. **Synchronous logic** is synchronized to change only when there is a clock transition. **Asynchronous logic** does not use a clock signal. Almost all of the logic used in a microcomputer design will either be un-clocked asynchronous logic (gates, decoders), or clocked synchronous logic (counter, latch or microprocessor). Some types of devices are available in either form. Each of the timing specs is described below using simple logic devices as they are typically used in embedded computer designs.

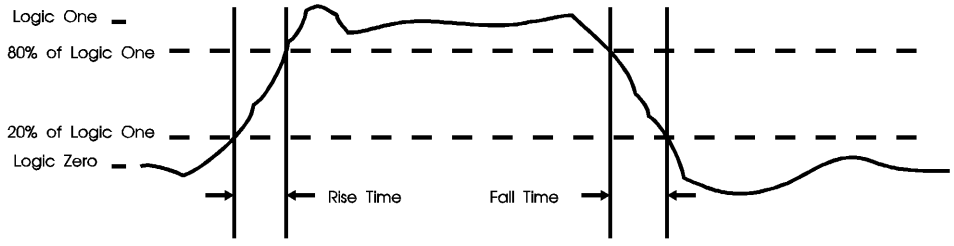


Figure 3-2) Rise and Fall Time of a Signal

Rise and Fall Time

The **rise time** of a signal is usually defined as the time required for a logic signal voltage to change from 20% to 80% of its final value. The **fall time** is from 80% to 20%, as shown in the figure below. These times are also commonly defined as the transitions between the 10% and 90% levels by some manufacturers.

Propagation Delays

The **propagation delay** is the time taken for a change at the input of a device to effect a change at the output. All devices, *even wires*, exhibit some propagation delay. Some devices do not have symmetrical delays for positive and negative transitions. In the diagram below, the propagation times for a high to low transition are shorter than for a low to high transition. This **asymmetrical delay** is common for TTL and open collector and open drain outputs because they are better at sinking current than sourcing it. So, the load capacitance is charged more slowly when the current is being supplied from the weaker "high side" or pull-up device. Most of the time, propagation delays are measured from the 50% amplitude points, as shown in the diagram.

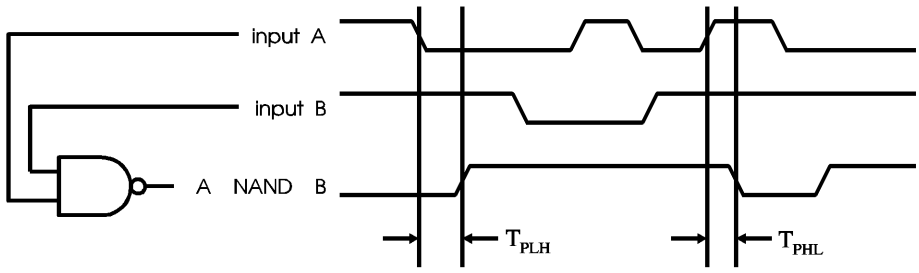


Figure 3-3) Propagation Delay

Setup and Hold Time

In the figure above, a standard D type flip-flop (e.g. a 74xx74 device) is shown along with a sample timing diagram, which illustrates the operation and key timing parameters of a flip-flop. This type of flip-flop samples the D input whenever the clock (CK) line goes high, and after a delay, the output remains in the same state until the next rising edge on the clock line. The triangle on the clock input indicates that it is a rising edge sensitive input, meaning that it will only have an effect when there is a rising edge on the clock pin. A falling edge sensitive input would have a bubble outside the block where the clock enters the flip-flop. In order to be able to guarantee that the flip-flop will operate correctly, the D input must be stable during the setup and hold time.

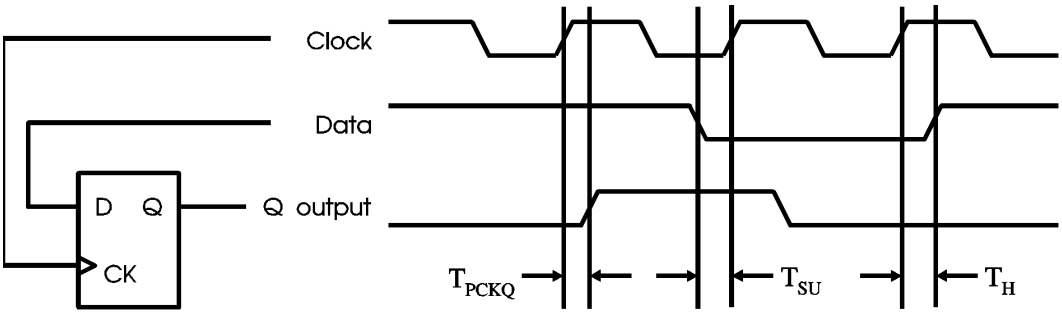


Figure 3-4) Setup and Hold Time

The figure above shows the propagation delay from clock to Q out (T_{PCKQ}), the Setup time (T_{SU}), and the Hold time (T_H). **Setup time** is the amount of time a sampled input signal must be valid and stable prior to a clock signal transition. **Hold time** is the amount of time that a sampled signal must be held valid and stable after a clock signal transition occurs. If these conditions are not met, the Q output may become invalid or even oscillate. This condition is referred to as **metastability**. The times of these and most other signals are frequently measured with respect to the 50% amplitude points of the clock signal rather than the valid logic one and zero levels. An analogy for the flip-flop as a sampling device is that of an instant camera: the clock is the shutter, the D input is the lens, and the output is the film image. The input is sampled when the shutter is open, and if the subject moves with the shutter open the picture will be blurred. For the flip-flop, the "shutter open" time, referred to as the window of uncertainty, and is shown in the diagram below along with some possible results. Metastability of a storage device such as a flip-flop or register is caused by the change of an input signal too close to the edge of the clock signal. In other words, if the setup or hold time requirements are not met, the output of the device is unpredictable and may even be unstable! The output may operate normally, take an invalid level, or oscillate, which may also explain why indecisive people take bad photos.

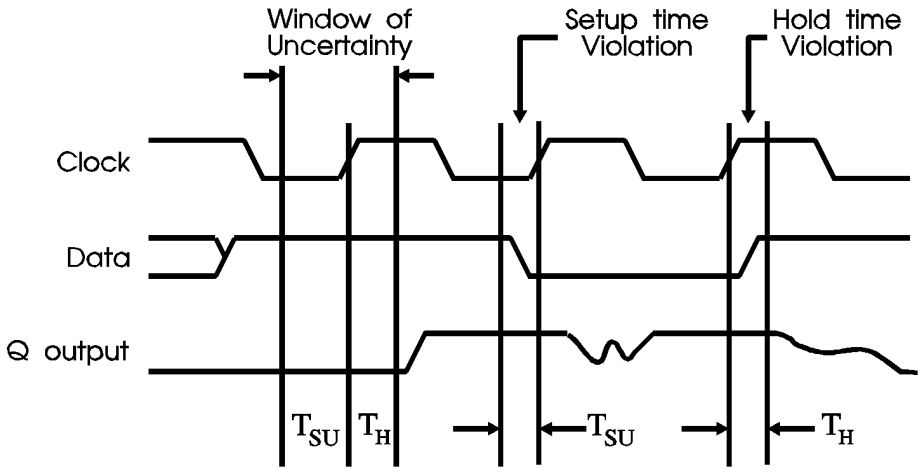


Figure 3-5) Metastability of a Flip flop

Tri-state bus interfacing

When multiple devices are capable of driving the same line, the possibility exists that two or more of them will try to drive it in opposite directions at the same time. When tri-state devices fight like this it is called **bus contention**. While the data is unpredictable during this period, there are far worse things that can happen as a result of this condition. Since most tri-state devices have the ability to drive many loads, they are also capable of sourcing and sinking large currents. When two of these devices are in contention, very large currents with peaks in the tens or hundreds of Amperes can flow for times on the order of nanoseconds.

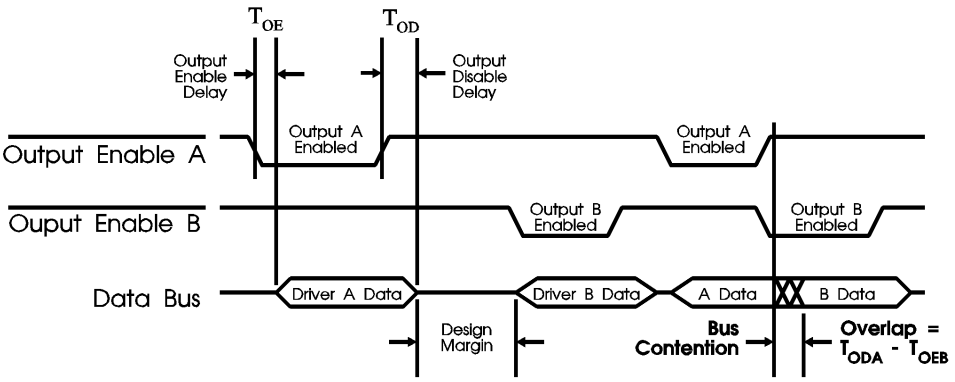


Figure 3-6) Tri state Bus Timing and Contention

The large current spikes that occur during contention may stress the devices and significantly reduce their reliability. A far more frequent problem, however, is the temporary drop or glitch in the local power supply wires that can cause any other nearby devices to change state. As you can imagine, this can create havoc in sequential logic, particularly for micros. Based on past experience with Murphy's law, these glitches generally seem to change the current instruction to "jump immediate to format hard disk routine," thereby erasing all your data. In a properly designed system, there is a "dead time" when no device is driving the bus to act as a safety margin between the time that two devices are enabled to drive their outputs. The problems arise when the output enable time of a device which is just turning on is less than the output disable time of a device which is turning off.

Pulse Width and Clock Frequency

The width of a positive going pulse is the period beginning from its positive transition (rising edge or leading edge) to its negative transition (falling or trailing edge). Pulse widths are important in defining the operation of control signals such as the memory read or write signals and clocks. Clock signals used for modern microprocessors usually, but do not always have equal high and low pulse width requirements. The period (T) of a signal is the sum of the rise time, high time, fall time, and low time. The frequency of a processor clock (f = 1/T), may have a lower limit as well as an upper limit. The standard NMOS 8051 family of parts has a lower frequency limit of 1.2 MHz. That means that the processor cannot be operated at a lower frequency. The reason is the processor’s internal design

requires a constant clock to correctly maintain its state. Other processors (such as the 80C51 series CMOS devices) can tolerate having their clock stopped completely, as they have been designed to maintain their internal states indefinitely, as long as power is applied.

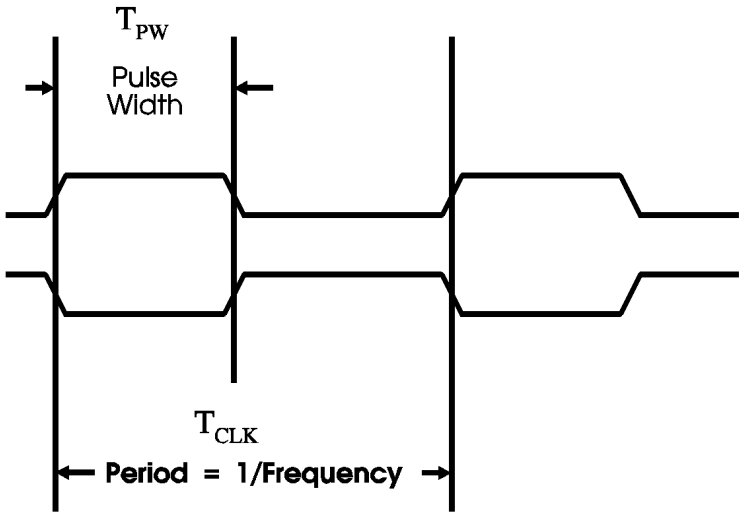


Figure 3-7) Pulse Width, Period, and Clock Frequency

Fanout, Loading analysis - DC and AC

Another important part of worst case design is a realistic model of the signal loading for each of the circuit's outputs. If insufficient drive is available, buffer circuits must be added or the number of loads must be reduced to guarantee correct operation. **Fanout** is the number of equivalent inputs that can be safely driven by one output. A fanout of 10 indicates that one device output can drive ten inputs. The fanout is determined from:

1. The source, type and number of loads
2. DC characteristics sources and load
3. AC characteristics of the loads vs. the source test conditions

DC characteristics of the output and inputs consist of:

- The maximum current that can be produced by an output
- Maximum currents required to drive an input

The maximum output currents are specified as:

I_{OLmin} Minimum output low (sink) current for a valid zero output voltage
I_{OHmin} Minimum output high (source) current for a valid one output voltage

Note that a LOW output is sinking currents that are coming out of the inputs that are being driven. Likewise, a HIGH output is sourcing current that goes into the inputs that are being driven.

Maximum currents required to drive input are specified as:

I_{ILmax} Maximum input low current for a valid zero input voltage
I_{IHmax} Maximum input high current for a valid one input voltage

Another important convention has to do with the sign of the current flowing in or out of a device pin. In most cases, current flowing **IN** to a device pin is given a **positive** sign, and current flowing **OUT** of a pin is given a negative sign. The conventions for the standard current direction and sign are illustrated in the following figures. The device on the left is the driving device, which tries to force its output to the desired logic state. In the logic one state, the output sources current (-50 uA), and the receiving device absorbs that current (+50 uA). In the example below, the available output current is exactly equal to the input current used by the load, resulting in a DC fanout of 1.

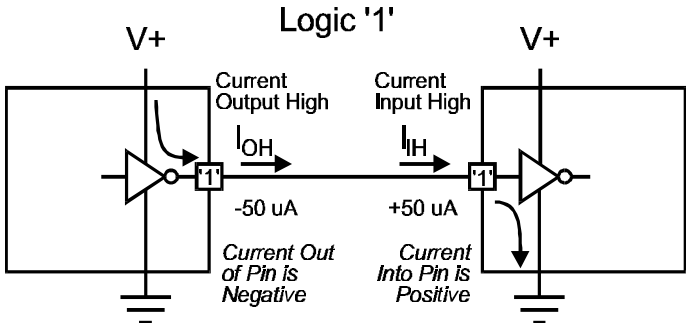


Figure 3-8) Current Sign for Logic High

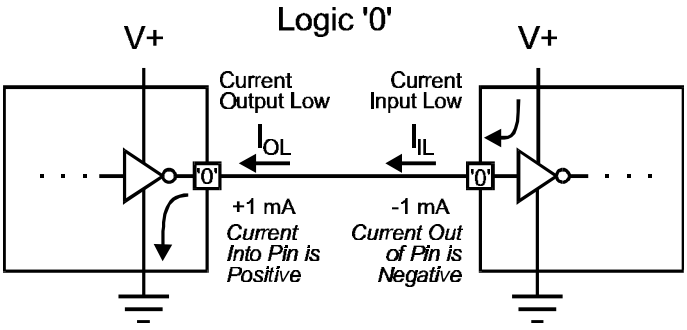


Figure 3-9) Current Sign for Logic Low

Unfortunately, ***this convention is not always followed consistently***, so it is up to you to recognize the current direction from the context of the situation in which it appears. Generally, the current direction can be determined by keeping these

images in mind, especially since many data sheets do not specify the sign for the input and output currents.

The other type of fanout limitation is the ability of an output to drive the capacitance of the loads and stray wiring capacitance, also known as **AC fanout**. The AC fanout is determined by the specified test load for the driving chip, and the load presented by the actual load capacitance. The capacitive load is the parallel combination of all the input capacitance of the gate inputs attached to the signal, plus the wiring capacitance. Since the capacitors in parallel are equivalent to a single capacitor equal to the sum of the individual capacitance, we just add up all the load capacitor values and compare this to the output's specified test load. The driving device's specified load capacitance, $\mathbf{C_L}$, the test load capacitance used by the manufacturer for specifying the AC or timing characteristics of the device. Most often, this specification is listed in the test conditions or notes for the timing specifications of the chip. As long as the sum of the load capacitance, including the stray wiring capacitance, is less than the specified test load for the driving device, all the timing specifications will be valid as specified in the timing section of the data sheet. If the driving device is overloaded (actual $\mathbf{C_L}$ > specified $\mathbf{C_L}$), then the timing specifications of the device need to be derated (slowed down), since additional capacitance will increase the rise and fall times of the signal line in question. Methods for estimating the amount that an overloaded output are described later.

AC characteristics:

- $\mathbf{C_L}$ The load capacitance that an output is specified to drive, is listed in the timing specifications for the driving device under the name "test conditions" which is usually in the notes at the bottom of the specification sheet.
- $\mathbf{C_{in}}$ Maximum input capacitance of a driven input load.
- $\mathbf{C_{stray}}$ Wiring and stray capacitance can be approximated to be in the range of 1 to 2 picofarads per inch of wiring on a typical PC board.

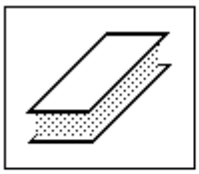
As long as the inequality below is satisfied, the signal will meet the timing specifications for the driving device. If the actual load is greater, it will delay

$$\text{Driving device spec } \mathbf{C_L} > \text{actual Cload} = \mathbf{C_{in1}} + \mathbf{C_{in2}} + \ldots + \mathbf{C_{wiring}}$$

The AC fanout is limited by the parallel combination of the logic inputs' capacitance, C_{in}, and the stray or wiring capacitance. Capacitors in parallel are additive, so the load presented to an output is the sum of the input capacitances of the logic inputs plus the wiring capacitance. Logic input capacitance is often difficult to find, as it may not be listed in the component data sheet, but rather in another section of the data book describing the characteristics common to all members of a given logic family. Typical logic input capacitance ranges from 1 to 5 pF (picofarads or 10^{-12} F), but may be outside this range. The maximum load capacitance which a device is specified to drive (C_L), is usually defined in the test

conditions for the timing specifications of an integrated circuit, as it is the timing which is most affected by capacitance. Load capacitance is usually specified in the range of 50 to 150 pF. Wiring capacitance is often in the range of 1 to 2 pF per inch of wire for a nominal printed circuit trace. Actual values can vary quite a bit, depending upon the physical dimensions of the trace, proximity to surrounding signals and distance from a ground plane, as well as the dielectric constant of the circuit board material.

Calculating Wiring Capacitance

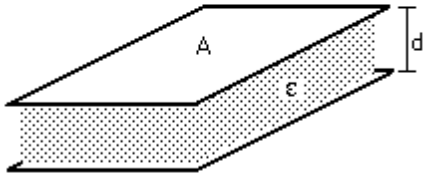


Capacitance:

$$C = \frac{\varepsilon \cdot A}{d}$$

Wiring capacitance is determined as a capacitance per unit length for a given trace width and distance from the ground or power plane.

The capacitance between two closely spaced parallel plates, given the area of the plates, A, the distance between the plates, d, and the permittivity of the material, ε.
The area, in this case, is the trace length times the trace width.



Standard units: $mils \equiv 10^{-3} \cdot in$ $pF \equiv 10^{-12} \cdot farad$

Example:
For an eight layer PC board with 8 mil traces, and innermost layer ground/power planes, what is the capacitance per inch of trace on each of the signal layers?

For:

$w := 8 \cdot mils$	Trace width	$T := 0.062 \cdot in$	Total Board thickness
$l := 1000 \cdot mils$	Trace length	$N := 8$	Number of layers
$A := w \cdot l$	Area	$n := 1$	Number of layers separation from power/ground plane

$t := \frac{T}{N - 1}$ $t = 8.857 \cdot mils$ Thickness of each dielectric layer

$d := n \cdot t$ Distance between trace and ground/power plane

$\varepsilon := 8.859 \cdot 10^{-12} \cdot \frac{coul^2}{(newton \cdot m^2)}$ permittivity of air

$\varepsilon r := 6$ Relative permittivity of glass-epoxy dielectric. This is a typical number for FR-4 material.

$f := 1.7$ Fringe effect and inter-trace stray capacitance adjustment factor

$C(d) := \frac{\varepsilon \cdot \varepsilon r \cdot A \cdot f}{d}$ Capacitance as a function of number of layers distance.

$C(1 \cdot d) = 2.073 \cdot pF$ Layer closest to ground/power plane

$C(2 \cdot d) = 1.037 \cdot pF$ Layer next closest to ground/power plane

$C(3 \cdot d) = 0.691 \cdot pF$ Layer farthest from ground/power plane

$Cavg := \frac{C(1 \cdot d) + C(2 \cdot d) + C(3 \cdot d)}{3}$ $Cavg = 1.267 \cdot pF$ Average per inch

From the sample calculations above, it is apparent that the stray wiring capacitance can vary significantly depending upon which layer of a multi-layer PC board a particular trace is located. Since a signal may travel on different layers between source and destination, exact values may be difficult to determine.

When performing a worst case analysis of a given design, it is most effective to calculate the total load capacitance based on the sum of the loads' input capacitances plus an estimate of the nominal wiring capacitance using 1 or 2 pF per inch of wiring using a rough guess for the length of the trace. In a typical design, we might pick the diagonal distance from one corner of the board to the other, and multiply by 1 or 2 pF. If the total load capacitance is less than the driving device's specified test load capacitance, then the device will perform as specified. If not, or if it's very close, we might want to make a more accurate estimate, or avoid the problem by using a driving device that has a larger specified test load capacitance. Other alternatives include using two outputs ***from the same chip*** in parallel to double the drive capacity, or splitting the loads into two separate groups and driving them independently from two different sources.

As digital IC technology has improved, allowing signals to be processed at ever-increasing rates, the other non-ideal effects of the devices that could be ignored at lower speeds become more important. At very high speeds, these secondary effects become much more important. A wire ceases to be equivalent to a zero Ohm connection with zero time delay. For the newer high-speed logic devices, the speed of the signal travelling down the wire, distributed resistance and inductance, as well as capacitance, may become very important. When the time it takes a signal to propagate down a wire are of the same order as the rise and fall time of the signal, it behaves as a ***transmission line***, rather than an ideal wire. Transmission line effects are briefly described in the next section.

Example 3-1 – Fanout, LS Output Driving CMOS Input

Use the parameters below to determine the fanout of CMOS driving LSTTL

3-1 a) Using the DC specifications, how many LSTTL loads can a CMOS gate drive?

LSTTL gate DC Parameters:

Symbol	Parameter	min	typ	max	Units	Conditions
VIL	Input Low voltage	-0.3		0.8	V	
VIH	Input High voltage	2.4		Vcc+0.3	V	
IIL	Input Low current		-120	-360	uA	
IIH	Input High current		30	50	uA	
Cin	Input Capacitance			10	pF	

Absolute Maximum Operating Conditions:

Symbol	Parameter	min	typ	max	Units	Conditions
VOL	Output Low voltage		0.2	0.4	V	@ IOL max
VOH	Output High voltage	2.8	3.5		V	@ IOH max
IOL	Output Low current	3.2	8		mA	@ VOL max
IOH	Output High current	-600	-1000		uA	@ VOH min

Note: Test conditions R_L = 1K, C_L = 100 pF

CMOS gate DC Parameters:

Symbol	Parameter	min	typ	max	Units	Conditions
VIL	Input Low voltage			2.0	V	
VIH	Input High voltage	3.0			V	
II	Input leakage current			~ 0	uA	
Cin	Input Capacitance			25	pF	

Absolute Maximum Operating Conditions:

Symbol	Parameter	min	typ	max	Units	Conditions
VOL	Output Low voltage			0.4	V	@ IOL max
VOH	Output High voltage	4.5			V	@ IOH max
IOL	Output Low current	3.6			mA	@ VOL max
IOH	Output High current	600			uA	@ VOH min

Note: Test conditions R_L = 5K, C_L = 150 pF

For Logic One:

CMOS Ioh = 600 uA

LSTTL Iih = 50 uA so 600uA/50uA = 12 loads

For Logic zero:

CMOS Iol = 3.6 mA

LSTTL Iil = 360 uA so 3.6mA/360uA = 10 loads

3-1 a) Answer: Considering the DC specifications only, the maximum number of loads driven is 10, since the zero state is the worst case condition. The AC parameters would not be the limiting factor in this case, since the CMOS output is specified with a C_L of 150pF, and each LS input is only 10pF. Thus, 10 loads would present 100pF

plus stray wiring capacitance of less than 50pF would present an AC load less than the 150pF CMOS output load handling capability.

3-1 b) How many additional CMOS loads could be added?

There are two levels of answer for this problem. First, from a DC point of view all the CMOS Iol output sink current is used up, so from this point of view, no loads could be added. However, there is negligible current in a CMOS input, so it is not the practical limit. In fact, the errors in the DC computations above are in excess of the amount required to drive a CMOS input, so in reality the DC current is not a problem. The real limitation is the capacitive loading. Even if you assume the loading from the TTL inputs and wiring can be ignored, the CMOS input capacitance will limit the loading. For the output to conform to the specs, the test load was specified as 150 pF (C_L). With ten LSTTL loads of 10 pF each, the C_L on the CMOS gate output would be 10*10=100pF. Since the CMOS gate timing is specified at C_L =150pF, there is only 150-100=50pF left over to drive the additional CMOS loads. Since the CMOS Cin is 25 pF, the number of additional gates that can be driven is:

50pF/25pF = (remaining C_L) / (Cin of additional CMOS inputs) = 2

3-1 b) Answer: Practically speaking, the wiring capacitance on a PC board will generally be in the 2-3 pF per inch range, so allowing 25 pF for wiring capacitance would permit 1 CMOS load in addition to the 10 LSTTL loads from above.

What if the CMOS output were to drive only CMOS loads?
The input capacitance of the CMOS gate is 25 pF, so even if *all* loads were CMOS, it can only drive C_L/Cin = 150pF / 25pF = 6 CMOS loads, and still meet its test condition limits. Since we must also allow for the wiring capacitance, we should limit this device to 5 loads, leaving 25 pF for the wiring capacitance. The additional load capacitance from more than 5 devices would likely result in timing performance that would be poorer than that specified in the data sheet. Excessive capacitance can also make ground bounce worse, which is the change in on-chip ground voltage due to rapid current spikes caused by charging load capacitance, developing a voltage across the lead inductance of the driving IC.

Transmission Line Effects
When using high-speed logic and the rise and fall times are of the same order as the propagation of the signal, transmission line effects become significant. When a signal transition propagates down a wire, it will be reflected back if the signal is not absorbed at the destination end. At lower speeds, the effect can be ignored, but with the fastest processors now in use, most designers will need to consider whether the effects will have a negative impact on their designs, and take appropriate action if necessary.

There are several characteristics of digital transmission lines, which must be addressed, including the following:

signal transition time vs. clock rate
mutual inductance and capacitance (crosstalk)
physical layout effects
impedance estimates
strip line vs. micro strip
effects of unmatched impedances
termination and other alternatives
series termination vs. parallel termination
DC vs. AC termination techniques

The techniques for high speed design are beyond the scope of this text, and are covered in detail in an excellent text on the subject, *High Speed Digital Design, a Handbook of Black Magic*, by Howard W. Johnson and Martin Graham. In contrast with the subtitle, this subject is easily understood by applying some very basic physics.

A transmission line is a conductor long enough so that the signal at the far end of the line is significantly different from the near end, due to the time it takes the signal to propagate from one end to the other.

In this book, we will assume that the interconnections between the devices are ***not*** long enough to require transmission line analysis. In order to verify that this is the case we can use a simple estimate. The rough estimate we will make is based on the idea that a wire does not have to be analyzed as a transmission line if the signal takes longer to rise or fall than it takes to get from one end of the wire to another. In other words, if the signal doesn't have to travel too far, both ends of the wire are at approximately the same voltage. In order to come up with a numerical value to determine if a signal must be treated as a transmission line, there is a simple calculation that can be used, shown below.

$l = T_r / D$ where
l = length of rising or falling edge in inches (in)
T_r = rise time in picoseconds (pS)
D = delay in picoseconds per inch (pS/in)

For traces on a standard printed circuit board, the value for D will be in the range of 100 to 200 pS/in. Depending upon how much distortion you're willing to live with, the critical trace length will be between $1/6^{th}$ and$1/4^{th}$ of the length of a trace corresponding to the signal's transition. For a trace that is shorter than $l/6^{th}$ the length of the signal's rising or falling edge, the circuit seldom needs to be considered to be a transmission line. Traces that are much longer than $1/4^{th}$ the length of the fastest edge will start to behave as transmission lines, exhibiting reflections of the signal when the transition gets to the far end of the trace and is

reflected back to the near end. Once the trace is about half of the length it takes for a logic transition to propagate, the problems become quite pronounced.

Example:
A logic device on a standard glass-epoxy printed circuit board has a 2 nS rise time.

This signal has a rising edge that is:

(2nS)/(150pS/in) = ~13 inches long.

That means a trace that is $1/6^{th}$ that length, or about two inches or less, does not have to be considered as a transmission line. If the trace is much longer than two inches, it will begin to show significant distortions on the rising and falling edges due to the fact that there is a different signal voltage at each end of the trace at the same instant, resulting in reflections of the signal from the ends of the trace.

This is one of the most important reasons for using logic, which is fast enough, and not too much faster than required to meet the timing requirements. While it might seem tempting to buy the fastest device available to reduce the delays in a device which does not meet the timing requirements, doing so can result in a lot more difficult problems to solve!

Ground Bounce
Another effect of high-speed signal transitions is called ***ground bounce***. Ground bounce occurs when a large peak current flows through the ground pin of a chip when one or more logic outputs change state and discharge their load capacitances through the chip's ground pin. While the parasitic inductance of the ground pin may not seem very significant, in the nanohenry (10^{-9}H) range, fast transients can cause large voltages to appear across the ground pin. This occurs most often when multiple bus signal outputs from one chip change state at the same time. The rapid, parallel current pulses which result from charging or discharging stray bus capacitance must be carried through the ground or power pins, which have inductance.

The voltage across an inductor is equal to the inductance times the rate of change of current through the inductor, or:

$V = L * di/dt$ where
- V = instantaneous voltage across the inductor (volts)
- L = Inductance (Henry)
- di/dt = Rate of change of current (Amps/sec)

and current $i = Q/t$ (Amps = Coulombs per second)
The charge on a capacitor is $Q = CV$ (Coulombs = Farads * Volts)
$V = L * C * (\text{delta } V) / (\text{delta } t)^2$ ***approximately,*** or
$V = L * C * (V_{oh}-V_{ol}) / (T_r)^2$ using the output voltage and rise time

Because of the high speed (nS) and large (Amps) peak currents, even the small nano-Henry inductance can induce a voltage transient on the order of volts. (The instantaneous voltage across an inductor is V = L * di/dt.) For typical high speed signals nano-Henries*amps/nanoseconds = volts! This effect is minimized by the use of minimum circuit interconnect trace lengths, wider ground traces, power and ground planes, and small, surface mounted IC packages that have very short leads.

For example, a CMOS output driving a 100 pF load with a rise time of 2 nS would induce a voltage across a typical 1 nH inductance of the chip's ground lead:

$$V = 1\ nH * 100\ pF * (4.5 - 0.5\ V) / (2\ nS)^2 = 0.1\ V$$

While a voltage of 0.1 V or 100 mV may not seem like much, remember that a part with many outputs, such as a processor, will sometimes switch many outputs at the same time, and ***the current that flows through those pins all has to flow through a single ground pin***. An 8-bit output will cause 0.8 volt pulse or ground bounce. If the processor drives an 8-bit data bus and a 16 bit address bus low at the same time, this would result in a 2.4 volt bounce! The ground bounce voltage across the ground lead inductance results in a different ground voltage reference for the chip while the chip's ground is bouncing. Needless to say, this ground bounce can cause a logic level to change during the brief pulse, which can cause trouble with circuits, such as clock signals, which are edge sensitive. This is why high speed logic devices may have multiple, short ground pins, and may only be available in small, surface mounted packages. To make things even worse, if two devices overlap slightly in time driving the bus, very large current transients may briefly generate even larger currents that in turn generate larger ground bounce pulses. This can disturb several chips on the board at the same time.

The power supply leads are also subject to bounce for exactly the same reasons, and even though the power supply is not used as a logic voltage reference, the resulting drop in the local power supply voltage to the chip can result in errors.

While exact ground lead inductances may prove difficult or impossible to measure, there is always some inductance in the ground lead, and the longer the lead, the greater the inductance. The example above illustrates another reason why it makes sense to avoid logic that is faster then necessary, and to use very short ground and power wires. In fact, high speed PC boards should use separate inner layers of a multi-layer board to provide large ground and power planes, allowing the chips' power and ground leads to be connected using very short wires.

The magnitude of the bounce depends upon the number and direction of logic transitions, so the noise is also data dependent! This is an apparently intermittent hardware design fault with symptoms that act like a software bug, since it may only happen at certain points in executing a program, with certain data values.

The example also shows why it is so important to maintain sufficient tolerance to noise in the logic. This noise tolerance is referred to as noise margin, which is covered in the next section. Noise margin analysis is especially important in a high speed logic design, to prevent transient logic errors, which are extremely difficult to track down. This is another example of how a proper analysis and worst case design can save a lot of time and money while delivering much higher quality and ultimately reliability. In the next section, the noise margin analysis process is described in detail.

LOGIC FAMILY IC CHARACTERISTICS AND INTERFACING

The three most common logic families are:

TTL Transistor Transistor Logic (also known as bipolar logic)
NMOS N-channel Metal Oxide Semiconductor field effect transistor logic
CMOS Complementary (n- and p- channel) MOS logic

All three logic families have versions with TTL compatible inputs, once the most common type, followed by later NMOS and CMOS. Because of its lower power density and relatively high circuit density however, CMOS has become the most common form of logic, particularly in high density and low power battery operated systems. TTL logic uses bipolar transistors requiring input drive currents on the order of 100's of microamperes to a few milliamperes depending on the version. Input voltage ranges for TTL level compatible logic are generally 0 to 0.8 Volts for logic zero and 2.4 to 5 Volts for logic one. Output voltages are from 0 to 0.4 Volts for logic zero and 2.8 to 5 Volts for logic one. The 0.4 Volt difference is called the **noise margin** voltage, because additive noise at or below this level will not change zeros to ones or vice-versa. The **logic threshold voltage (V_T)** or "0/1 decision point" for TTL logic is typically around 1.5 volts. It may range anywhere between 0.8 and 2.0 volts depending upon supply voltage, temperature, and varies from one device to another. For TTL circuits, the noise margin is at least 0.4 volts. Interconnecting different logic families, such as CMOS and TTL, requires the designer to assure the compatibility of the logic signal voltage levels, and adapt the circuit as necessary to maintain appropriate noise margins. The equivalent resistance or impedance of the signal network also has an impact on the noise in a specific circuit. High impedance inputs are more prone to noise than low impedance inputs. The interface design process is illustrated by an example at the end of this chapter.

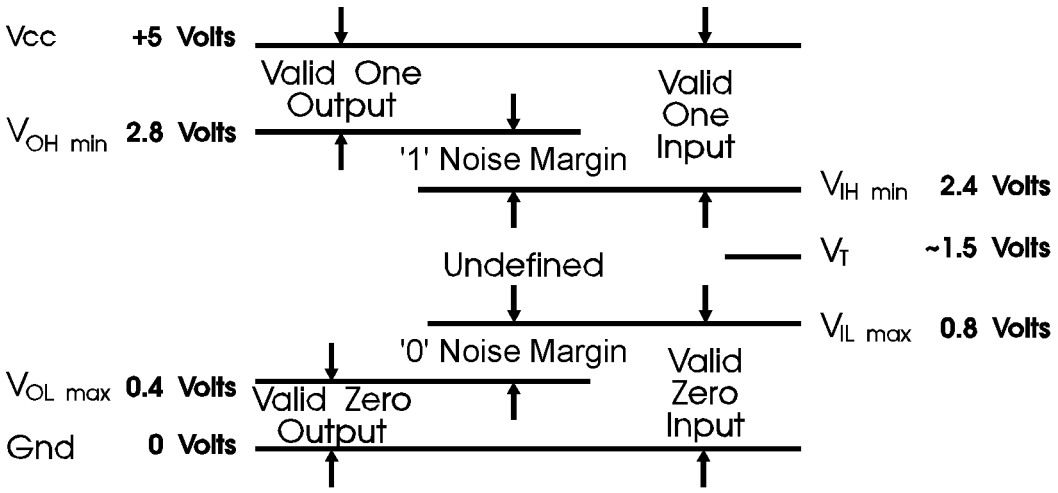


Figure 3-10) Typical TTL Logic Voltages and Noise Margin

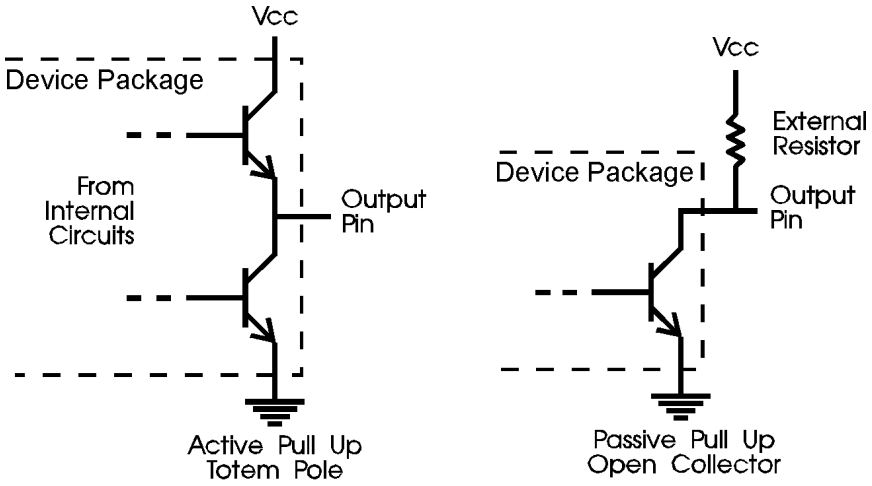


Figure 3-11) TTL Outputs: Totem Pole and Open Collector

TTL logic is capable of sinking high currents and is used for driving very fast, large, heavily loaded buses. Both active and passive pull-up output devices are used with TTL. The active pull up, referred to as a **totem-pole output**, uses one transistor to source current and one to sink it. The passive pull-up uses a transistor to sink current, and a resistor connected to V+ as a current source. If a pull up resistor is not connected to the gate's output pin, and the collector is connected only to the output pin, it is referred to as an **open collector output**. In both cases, the output current sinking capabilities are greater than current source capacity. Many devices can sink a few mas, but can only source 100s of uA.

TTL and CMOS logic are available in several versions, some common ones are:

74xx series standard TTL
74LSxx series Low power Schottky clamped TTL
74ALSxx Advanced LS TTL
74Fxx (Fast) high speed TTL

74HCxx High speed CMOS with CMOS compatible inputs (Vt = ~Vcc/2)
74HCTxx High speed CMOS with TTL compatible inputs (Vt = ~1.5V)
74FCTxx High speed CMOS with TTL compatible inputs (Vt = ~1.5V)
74ACTxx Advanced high speed CMOS with TTL compatible inputs
74BCTxx Very high speed CMOS/Bipolar with TTL compatible inputs

Schottky logic (74ALSxx 74LSxx and 74Sxx) incorporates a low V_f (forward voltage drop) Schottky diode across the collector-base junction of a transistor to prevent it from saturating. This increases the speed for turning the transistor off. TTL is generally used where low cost, output drive, and high speed are important, and there is no objection to the relatively high power consumption and resulting heat.

NMOS logic was used for moderate complexity logic IC's like the more mature microprocessors. Most NMOS logic ICs have TTL compatible voltage specs and operate at a lower power and speed than TTL. The power consumed by NMOS lies between TTL and CMOS, as does its speed. The input current is nearly zero since the MOSFETs have extremely high input resistance. Unfortunately, they do have fairly large input capacitance, limiting the circuit speed. The output configurations are similar to TTL except the transistors are n-channel field effect transistors (FETs) rather than bipolar NPN. Both active totem pole and passive (open drain) outputs are used in microprocessor and microcontrollers. Because of the constant operating current drain, these devices tend to be limited in size and complexity.

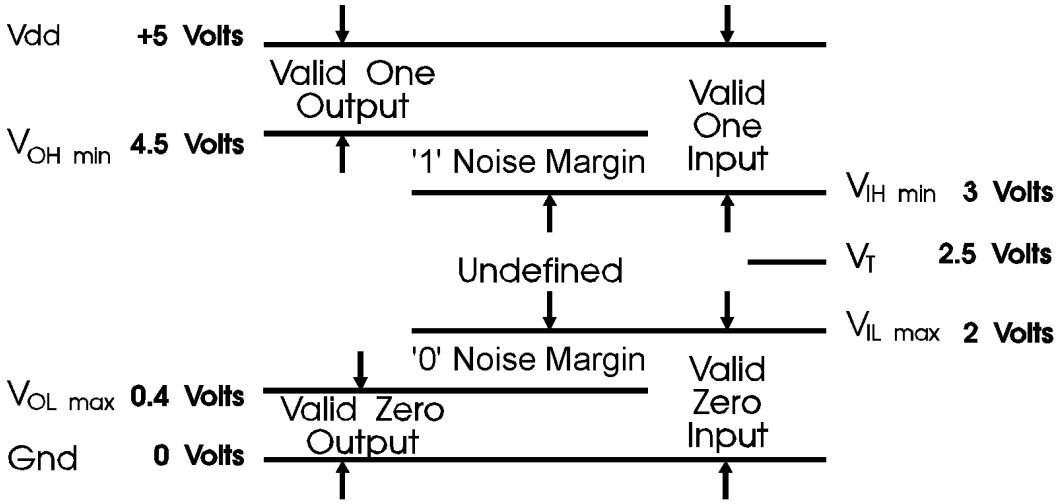


Figure 3-12) Typical CMOS Logic Voltages and Noise Margin

CMOS logic has a significant advantage since it does not use any significant amount of power when it is static (not changing state). Most of the power used in an operating device is due to the charge and discharge of internal capacitance and the current transient when both N and P devices are partially on. As a result, power consumption is a function of clock rate for CMOS devices. Some processors are even designed to take advantage of this fact by incorporating

"sleep" or low power modes stopping some or all of the clock operations when nothing important is going on. This is frequently required for battery operated systems to maintain a reasonable battery life. Another advantage is the standard CMOS logic threshold is one half the supply voltage, and the output voltages tend to be very close to Vcc and ground voltage, resulting in higher noise margins than those of TTL devices. This is particularly important for CMOS devices that operate at reduced power supply voltage. CMOS devices are available which operate at 3 volts or less.

Because CMOS logic is inherently symmetrical, the rise and fall times tend to be nearly equal. The symmetry also results in equal source and sink capabilities. The inherent increase in noise margin makes CMOS less susceptible to noise than TTL and NMOS. CMOS devices operating at voltages other than 5V, such as 3.3V, will have a threshold voltage corresponding to Vcc/2. Some versions of CMOS logic operate with a reduced noise margin in order to have TTL compatible input voltages. This is accomplished by artificially lowering the input threshold voltage to 1.5 volts, the same as used for TTL. These TTL input threshold compatible circuits have a T in their number (74HCT, 74BCT, etc.) indicating they have TTL compatible inputs. A series of high speed logic compatible with the TTL logic family in function and input voltage is the 74HCTxx (High speed CMOS TTL compatible) series. The advantage of the 'T' series CMOS devices is they can be driven directly by devices having TTL output voltage levels. The 'T' series of CMOS devices has the disadvantage that the noise margin is less than it is for true CMOS compatible inputs due to the shifted threshold voltage. The 74HCxx series is pure CMOS with a threshold voltage of one-half the supply voltage (2.5V for a 5Vcc) and correspondingly higher noise margins. As a result, a standard TTL output V_{OHmin} of 2.8 volts is not enough to guarantee a logic one value for a 74HCxx gate input.

Interfacing TTL Compatible Signals to 5V CMOS

Interfacing a CMOS output to a TTL input is a direct connection, as long as the CMOS output is capable of sinking the TTL device's input low current. Interfacing a TTL output to a standard CMOS input requires the use of at least a pull up resistor. A resistor on the TTL output to Vcc will ensure the output voltage is pulled high enough to guarantee the logic one output signal is interpreted as a logic one by the CMOS input. Another useful technique when using five volt logic to drive CMOS circuits, is to use a higher voltage open collector or open drain output with a pull up resistor connected to the higher supply voltage. This level shifting technique can also be used for driving other high voltage circuits such as high voltage outputs. In either case, the objective is to guarantee there is sufficient noise margin to guarantee a valid logic one when the TTL compatible output drives a CMOS input.

It is important to note that when a TTL output is pulled above its normal output high voltage, it will not source any significant current. This is because the TTL

output source is equivalent to a high resistance in series with a voltage source that is effectively limited to around 3 volts, due to internal design constraints. As the output voltage increases until it equals the internal voltage, the output can no longer source any current. When the voltage is increased beyond the internal circuitry (up to a limit of Vcc), the internal circuitry is equivalent to a reverse biased diode, so only leakage currents in the sub-micro-Amp range will flow into the output device. As a result, the effect of a TTL output on external circuits is negligible when the pin is pulled high by an external resistor.

Also, a 5 V TTL compatible output is often compatible with a 3 V CMOS device input, since the CMOS threshold (Vcc/2 = 1.5 V) is the same as a 5 V TTL gate (TTL Vt = 1.5 V). Most of the 3 V CMOS devices are designed to withstand a 5 V input signal, so it is often possible to interface 5 V TTL outputs directly to 3 V CMOS inputs. However, if the 3 V CMOS inputs are not designed to handle 5 V inputs, the CMOS device could be destroyed with an input signal greater than 3 V, so it is important to verify this. A 3 V CMOS device output will be close to 3 V, so it can drive a 5 V TTL compatible input directly.

A 3 V CMOS output would probably be marginal driving a 5 V CMOS input (Vt = Vcc/2 = 2.5 V), leaving less than 0.5 volts of noise margin. Since the 3 V CMOS output generally cannot withstand a pull-up resistor to 5 volts, it is necessary to add a level shifting IC to convert 3 V logic levels to 5 V.

Level shifters are available for converting logic levels from one family to another, including 3V to and from 5V, or 5 V TTL to +/- V ECL (Emitter Coupled Logic), and 5 V levels to +/-12 V RS-232 signals. There are also special ICs for driving output loads requiring either a high voltage or high current output, such as a light, motor or relay. Most microcontrollers have very weak output drive capability, so external driver ICs may be necessary. These would typically be needed to drive LEDs, a vacuum fluorescent display, or a motor. Solid state relays even allow large AC loads to be controlled by a micro. Likewise, there are other devices (i.e. optical isolators), allowing high voltages (like 110VAC inputs) to be safely converted to logic levels for input to a microcontroller. Devices that use potentially hazardous high voltages should be isolated from the rest of the circuitry for reasons of safety. While it may be possible to connect such devices directly to our circuits, they would allow us to come into contact with potentially fatal voltages. Unfortunately, the standard 50 or 60 cycle AC power supply used almost everywhere has the unfortunate characteristic that it is very nearly the optimal voltage to guarantee that a human heart will stop functioning due to muscle fibrillation. Customer death by electrocution is sure to result in the next of kin hiring an attorney to relieve you of all your assets… Unless of course they're *your* next of kin! There are many isolation devices available, most of which use the same basic approach.

The isolation can be accomplished using optical or magnetic means, which can provide a barrier to transient voltages that can be on the order of thousands of volts. The barrier is transparent, and so allows light to pass, but is made of a good

insulator to prevent electrical current from flowing across the boundary. The figure below shows a simple optical isolation circuit.

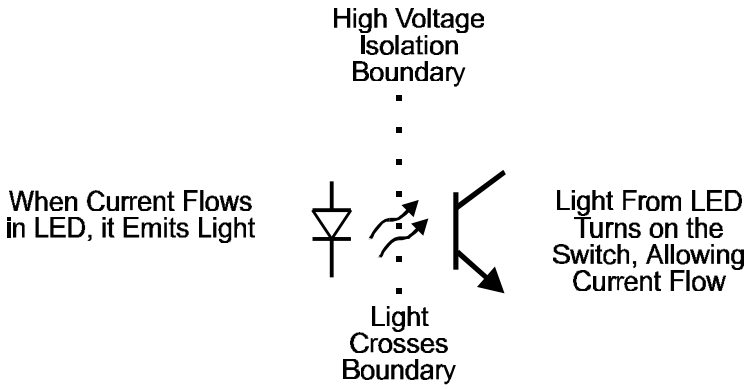


Figure 3-13) Optical Isolation Allows Connection to Hazardous Voltages

This isolation approach can be used to input high voltages to a microcontroller safely by connecting the LED to a high voltage source in series with a resistor and protective diode to limit the LED's current and prevent the LED from being exposed to the potentially destructive reverse voltage. The output transistor will then be turned on whenever the LED is turned on by one half of the AC power cycle. This is useful for time of day clock functions, since the AC power main frequency is maintained very accurately by the power utilities over a period of time. The output switch can be connected to the processor counter or interrupt input, allowing the processor to keep track of time and synchronize its operation with the AC power cycle.

High voltage outputs can also be controlled safely by using the micro's output to turn on the LED that turns the output switch on. In this case, another type of switch such as an SCR (Silicon Controlled Rectifier) or TRIAC (an AC version of the SCR) is used rather than a transistor. SCR and TRIAC switches can be obtained to handle relatively large AC loads, such as lamps, and motors. These devices are often referred to as Solid State Relays (SSR), since they are equivalent to an electromechanical relay, except that they are implemented with solid state semiconductor devices instead of using a coil to move a switch. Both isolated inputs and outputs are available in complete modules that have all the necessary circuits to monitor and control high voltage and power devices, using optical isolation for safety. They have microcontroller compatible I/O on one side that is optically isolated from the high power outputs on the other side.

Very often, even when safety is not an issue, microcontroller chips simply cannot handle the voltages or currents required interfacing with other devices. In some cases it is required when connecting one logic family to another, incompatible

family, such as Emitter Coupled Logic (ECL) levels or RS-232 interfaces, utilizing negative voltages.

Sometimes, a plain old fashioned electromechanical relay is a better solution, since relays usually have contact resistances that are far lower than can be found in a semiconductor switch. In some cases, a simple transistor or MOSFET switch can be used to control a load operating at voltages which are greater than the logic supply, such as motors, solenoid actuators, and relays which may require 12 or more volts to operate.

The circuitry required to interface between logic levels and high level circuits is described in detail elsewhere, including an excellent book titled *The Art of Electronics*, by Horowitz and Hill. If you don't already have this book, and you have to do much electronic design or interfacing, you should definitely have a copy of this very handy book.

The real world is an analog place, and interfacing between the discrete, digital world of computers and the real world also requires significant attention. The interface between low level analog signals and logic is handled in another chapter of this book.

At this point, it is time to look at some simple examples, so we can see exactly how a worst case analysis should be performed. The next section illustrates part of the worst case analysis for a real laboratory instrument that is still used in the healthcare industry. This product's poor reliability was seriously inconvenient for the medical staff and patients who depend upon it, and if it had lead to an incorrect diagnosis, a truly fatal error! It is in these types of applications that worst case design is most important, but the cost of unreliable hardware in the field almost always greatly exceeds the cost of avoiding the problem by using proper design and analysis techniques. Now let's turn our attention to the analysis of the worst case noise margin for an 8051 based design example.

Example Noise Margin Analysis Spreadsheet

The following spreadsheet shows the results of a noise margin on a design that was already in production at the time of the analysis. The product's users had complained about intermittent glitches, and the author was consulted to determine the source of the problem. After a quick look at a few of the noise margin values, it became obvious that there were deficiencies in the design in that area. A portion of the spreadsheet used in that analysis is included below, with problems shown in ***bold italic underline*** font.

8051 Noise Margin Analysis - Sample									**Noise Margin**	
OUTPUT					INPUT					
Signal	Pin(s)	Source	Vol max	Voh min	Load(s)	Signal	Vil max	Vih min	logic zero	logic one
PSEN/	29	8051	0.40	2.00	EPROM	OE/	0.80	2.00	0.40	***0.00***
RD/	17	8051	0.40	2.00	SRAM	OE/	0.80	2.20	0.40	***-0.20***
(P3.7)			0.40	2.00	82C55	RD/	0.80	2.00	0.40	***0.00***
WR/	16	8051	0.40	2.00	SRAM	WR/	0.80	2.20	0.40	***-0.20***
(P3.6)			0.40	2.00	82C55	WR/	0.80	2.00	0.40	***0.00***
A15 (P2.7)	28	8051	0.40	2.00	74LS138	A	0.80	2.00	0.40	***0.00***
A8..14	21-27	8051	0.40	2.00	SRAM	A8..14	0.80	2.20	0.40	***-0.20***
(P2.0-P2.6)			0.40	2.00	EPROM	A8..14	0.80	2.00	0.40	***0.00***
			0.40	2.00	GAL	A8..14	0.80	2.00	0.40	***0.00***
ALE	30	8051	0.40	2.00	74LS373	LE	0.80	2.00	0.40	***0.00***
AD0..7	39-32	8051	0.40	2.00	74LS373	A0..7	0.80	2.00	0.40	***0.00***
(P0.0-P0.7)			0.40	2.00	SRAM	D0..7	0.80	2.20	0.40	***-0.20***
			0.40	2.00	82C55	D0..7	0.80	2.00	0.40	***0.00***
		SRAM	0.40	2.20	8051	D0..7	0.80	2.40	0.40	***-0.20***
		EPROM	0.45	2.40	8051	D0..7	0.80	2.40	0.35	***0.00***
		82C55	0.40	3.50	8051	D0..7	0.80	2.40	0.40	1.10
RAM Enable		16V8	0.50	2.40	SRAM	/CE	0.80	2.20	0.30	0.20
EPROM Enable		16V8	0.50	2.40	EPROM	/CE	0.80	2.00	0.30	0.40

The first column is the signal name, followed by the pin number and chip which is the source of the signal, followed by the source's worst case output voltages, Volmax and Vohmin. The next columns list the loads on the signals and their respective worst case input voltages Vilmax and Vihmin. The noise margins are shown in the last two columns, Vil - Vol for the logic zero case, and Voh - Vih for the logic one case. As can be seen, the logic zero noise margins are all probably acceptable, as the lowest value is 0.3 volts. The logic one noise margin is zero or negative for most of the devices listed, which is completely unacceptable. Any noise on the power supply, ground or the signal lines themselves can easily cause a logic input to interpret the wrong logic state, causing an error. An interesting thing to observe is that none of them were very far out of spec, and the instrument worked perfectly most of the time. These problems can be virtually impossible to find in the field. Hooking up a test instrument like a scope or logic analyzer to the

problem signals often makes the problem go away, due to changing the ground currents and impedances of the circuit. The specs that cause the problem in this case are the high Vih specs of the loads, especially the SRAM chip. The example design in the sheet above represents a relatively common problem with devices that are advertised as "compatible" with other logic families. The solution to the problem is very simple and inexpensive: the addition of pull-up resistors to the signals that have zero or negative noise margin in the logic one state. This also impacts the output low current that must be handled by the signal source chip outputs, so it must be taken into account in the load analysis and pull up resistors should be chosen accordingly.

It is important to note that there are four sources listed for AD0..7, since there are four devices that drive the data bus. Only the data paths that are used need to be evaluated vs. loading analysis, where unused paths load the bus. The load analysis for another similar design is shown in the next worksheet, which tabulates the capabilities of the various driving devices, and the loads which are presented to them. The first three columns (signal, pin and source) identify the signal source, the next three (IOL, IOH and CL), list the corresponding source's output drive current and capacitive load values. The next two columns (load, and signal) identify the load's signal names. The Qty column is the number of loads in the case of multiple signals connected to the same output, or the number of inches of wire in the case of the wire capacitance. The next three columns (IIL, IIH, and Cin) define the load characteristic of a single input's input current and input capacitance. For the interconnect wiring, Cin is the estimated stray wiring capacitance per inch of the printed circuit trace. The last three columns show the extended totals and grand totals for each signal, followed by the design margin, which should be a positive number. In this case there is only one problem, due to excessive capacitive loading of the SRAM when it drives the data bus, AD0..7.

The output capacitive load specs are usually found as notes within the AC section of the chip specification listing the various timing parameters. This is because the capacitive loading affects the rise and fall time of the signal, so the capacitance value is really used as a test condition for the timing measurements. Input capacitance may be difficult to find in the specification sheet, it may be in a different "family" specification sheet or handbook, or may not be specified at all. When it is not specified, substituting values for similar parts in the same type of package can make a reasonable estimate.

Load Analysis Worksheet Example

Source						Load			Unit Load			Total		
			uA	uA	pF				uA	uA	pF	uA	uA	pF
Signal	**Pin#**	**Source**	**IOL**	**IOH**	**CL**	**Load**	**Signal**	Qty	**IIL**	**IIH**	**Cin**	**IIL**	**IIH**	**Cin**
PSEN/	29	8051	3200	-60	100	EPROM	OE/	1	-1	1	12	-1	1	12
						wire cap		2			2			4
											Total	-1	1	16
											Margin	**3199**	**59**	**84**
RD/	17	8051	1600	-60	80	SRAM	OE/	1	-1	1	7	-1	1	7
(P3.7)						82C55	RD/	1	-1	1	10	-1	1	10
						wire cap		3			2			6
											Total	-2	2	23
											Margin	**1598**	**58**	**57**
WR/	16	8051	1600	-60	80	SRAM	WR/	1	-1	1	7	-1	1	7
(P3.6)						82C55	WR/	1	-1	1	10	-1	1	10
						wire cap		3			2			6
											Total	-2	2	23
											Margin	**1598**	**58**	**57**
A15	28	8051	1600	-60	80	74LS138	A	1	-200	20	10	-200	20	10
(P2.7)						wire cap		2			2			4
											Total	-200	20	14
											Margin	**1400**	**40**	**66**
A8..14	21-7	8051	1600	-60	80	SRAM	A8..14	1	-1	1	7	-1	1	7
(P2.0-P2.6)						EPROM	A8..14	1	-1	1	12	-1	1	12
						wire cap		3			2			6
											Total	**-2**	**2**	**25**
											Margin	**1598**	**58**	**55**
ALE	30	8051	3200	-60	100	74LS373	LE	1	-400	20	10	-400	20	10
						wire cap		2			2			4
											Total	**-400**	**20**	**14**
											Margin	**2800**	**40**	**86**
AD0..7	39-2	8051	3200	-800	100	74LS373	A0..7	1	-400	20	10	-400	20	10
(P0.0-P0.7)						SRAM	D0..7	1	-1	1	7	-1	1	7
						EPROM	D0..7	1	-1	1	12	-1	1	12
						82C55	D0..7	1	-10	10	20	-10	10	20
						wire cap		5			2			10
											Total	**-412**	**32**	**59**
											Margin	**2788**	**768**	**41**
		SRAM	1600	-600	50	74LS373	A0..7	1	-400	20	10	-400	20	10
						8051	D0..7	1	-1	1	20	-1	1	20
						EPROM	D0..7	1	-1	1	12	-1	1	12
						82C55	D0..7	1	-10	10	20	-10	10	20
						wire cap		5			2			10
											Total	**-412**	**32**	**72**
											Margin	**1188**	**568**	**-22**
		EPROM	1600	-600	100	74LS373	A0..7	1	-400	20	10	-400	20	10
						SRAM	D0..7	1	-1	1	7	-1	1	7
						8051	D0..7	1	-1	1	12	-1	1	12
						82C55	D0..7	1	-10	10	20	-10	10	20
						wire cap		5			2			10
											Total	**-412**	**32**	**59**
											Margin	**1188**	**568**	**41**
		82C55	1600	-60	80	74LS373	A0..7	1	-400	20	10	-400	20	10
						8051	D0..7	1	-1	1	20	-1	1	20
						EPROM	D0..7	1	-1	1	12	-1	1	12
						SRAM	D0..7	1	-1	1	7	-1	1	7
						wire cap		5			2			10
											Total	**-403**	**23**	**59**
											Margin	**1197**	**37**	**21**

The SRAM output is specified with a Cload value of 50 pF, which is a relatively low value. By using a very low load capacitance, the SRAM's timing specs look good due to shorter than normal rise and fall times, since the chip is not driving a

realistic load. This is a good example of a manufacturer's "specsmanship." They are intentionally playing games with the test conditions to make their device appear to be better than it is. That way when someone looks at their timing specs, the shorter rise and fall times make their chip appear to be faster than another equivalent chip that is specified with a larger capacitive load value, when the chips are actually identical. Unfortunately, this practice is all too common, so that the designer must view the claims on the cover of a data sheet very critically. If it looks too good to be true, then it probably is.

When an output like this is operated with actual capacitive load greater than the test conditions, the related timing specs for the device must be derated, due to the degraded rise and fall times that will occur. As long as the load capacitance is no more than twice the spec value, this will be sufficient. The excess C load will increase the stress on the driver. If the overload is much greater than 2x, the device can be overstressed due to the relatively large currents that will flow into the load capacitance on transitions when the C is charged and discharged through the driving output. As long as the output is not overloaded too much, the resulting increase in the rise/fall time can be estimated, resulting in a derated timing spec. All we have to do is calculate the additional rise time and add that to the timing values specified in the data sheet. In order to do that, we need to evaluate the output circuit's performance. This can be accomplished by noting that the output current drives the load capacitance from a logic low to high or vice versa. For our purposes, we will assume that the interconnect does not behave like a transmission line, which is most often the case for garden variety microcontroller components. If the chips used have a fast rise time and trace length greater than about $1/6^{th}$ the edge length of the pulse, then it is necessary to analyze the circuit as a transmission line. In this case we will look at the simpler problem.

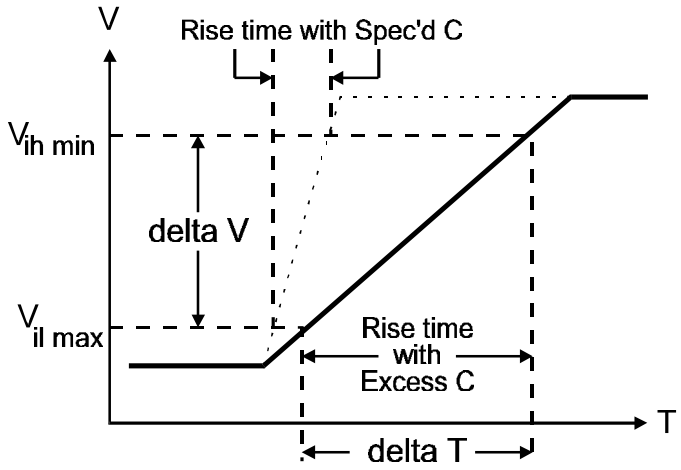


Figure 3-14) Derating Delay for Excess C_L

By assuming a constant current charging the capacitance, the voltage will ramp linearly from one logic level to the other. To make a rough estimate, we can use the source's output current and load capacitance to determine the signal slew rate, and the difference between the high and low logic levels to determine the delay.

Next is a simple example showing how to derate the timing based on the approximation technique described above. First we make the assumption that the signal timing measurements in the data sheet are made under the specified test conditions, usually with the output loaded by R_L and C_L in parallel to ground. The output delay specifications in the data sheet include the internal delay as well as the rise time. The output drive current charges C_L within the specified time. The circuit can be divided into two parts: the specified load, and the additional output current available to drive the excess load C. So the additional delay (delta T) we are looking for depends upon the left over drive current (delta I) which is available to charge the excess load capacitance (delta C). The equation is as follows:

Delta T = (delta V * delta C) / (delta I)

Example

An SRAM is specified with a 50 nS access time, but the outputs are overloaded with respect to the C_L spec in the data sheet. What access time spec should be used for the actual conditions specified below?

The output is specified to drive C_L = 50 pF, but the actual load is 100 pF.
The output is specified to drive 20 mA into the load, but the load is only 10 mA.
The driven device has input voltage specs Vilmax = 0.4 V, Vihmin = 3.4 V.

Spec values:	Actual Values:	Difference:
C_L = 50 pF	100 pF	50 pF = delta C
Io = 20 mA	10 mA	10 mA = delta I

Voltage: Vih - Vil = 3.4 - 0.4 = 3 V = delta V

Delta T = (delta V * delta C) / (delta I)

Delta T = (3 V * 50 pF) / (10 mA) = 15 nS

So in this case 15 nS should be added to all the output delay specs for the driving device. The access time used should be:

Taa(actual) = Taa(spec) + (delta T) = 50 nS + 15 nS = **65 nS**

Since the output current from most devices is larger at the beginning of the transition and smaller near the end of the transition, the approximation is only a rough guide. Also, the delta V calculation is conservative, since the input

threshold voltage is typically half way between the Vih and Vil values. So, the estimate, as shown, will usually be conservative compared to actual performance. All of the above must be used with caution, and is only an approximation of the additional delay caused by excess C_L, so it is wise to allow additional margin in the timing for any derated specs.

Example 3-2 - Worst Case Loading Analysis

An LSTTL gate is to be used to drive one LSTTL load and a CMOS processor clock input. An interface must be made which will guarantee the CMOS input voltage requirement will be met with the same noise margin as a standard LSTTL input. The LSTTL and CMOS gates have the specs as defined in the tables below:

LSTTL gate DC Parameters:

Symbol	Parameter	min	typ	max	Units	Conditions
V_{IL}	Input Low voltage	-0.3		0.8	V	
V_{IH}	Input High voltage	2.4		Vcc+0.3	V	
I_{IL}	Input Low current		-120	-360	uA	
I_{IH}	Input High current		30	60	uA	

Absolute Maximum Operating Conditions:

Symbol	Parameter	min	typ	max	Units	Conditions
V_{OL}	Output Low voltage		0.2	0.4	V	@ I_{OL} max
V_{OH}	Output High voltage	2.8	3.5		V	@ I_{OH} max
I_{OL}	Output Low current	3.2	8		mA	@ V_{OL} max
I_{OH}	Output High current	-600	-1000		uA	@ V_{OH} min

Note: Test conditions R_L = 1K, C_L = 100 pF

CMOS gate DC Parameters:

Symbol	Parameter	min	typ	max	Units	Conditions
V_{IL}	Input Low voltage			2.0	V	
V_{IH}	Input High voltage	3.0			V	
I_I	Input leakage current			<1	uA	

Absolute Maximum Operating Conditions:

Symbol	Parameter	min	typ	max	Units	Conditions
V_{OL}	Output Low voltage			0.4	V	@ I_{OL} max
V_{OH}	Output High voltage	4.5			V	@ I_{OH} max
I_{OL}	Output Low current	3.2			mA	@ V_{OL} max
I_{OH}	Output High current	600			uA	@ V_{OH} min
C_{in}	Input Capacitance			20	pF	

Note: Test conditions R_L = 5K, C_L = 150 pF

3-2 Answer:

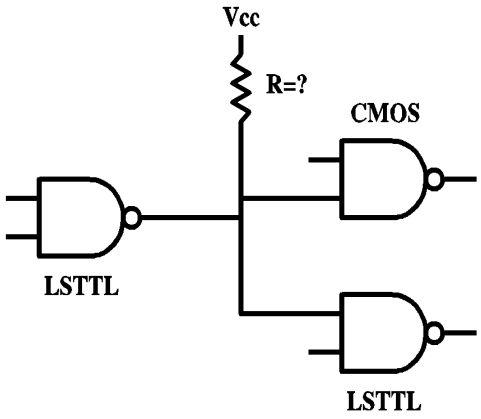


Figure 3-15) TTL to CMOS Interface Example

Since the LSTTL V_{OL} is 0.4 V and the CMOS V_{IL} is 2.0 V, the CMOS input low voltage is compatible with the LSTTL low output voltage. However, the LSTTL output high voltage of V_{OH} = 2.8V is not sufficient to meet the CMOS input high V_{IHmin} = 3.0V. A pull-up resistor is required to allow the LSTTL output to go to a higher voltage, $V_{IH} + V_{noise\ margin}$ = 3.0 + 0.4 = 3.4 V. There is no exact solution, but the range of resistors meeting the requirements can be determined.

The lowest resistor value that will work is the value which will source enough current so the LSTTL output is just able to sink the resistor current plus the additional LSTTL load when the signal is low and still meets the maximum output low voltage spec. There is negligible DC current flowing from the CMOS input. The voltage across the resistor is Vcc - $V_{OL\ max.}$ for the LSTTL input, or 5 - 0.4 = 4.6 V. The current required is $I = I_{ILmax} + I_{RPU}$ where I_{ILmax} is the current coming from the LSTTL input load and I_{RPU} is the current flowing through the pull up resistor. The current the LSTTL output must sink is the sum of the I_{IL} of the LSTTL load and the current through the pull up resistor.

The equation is:
$I_{OLmin} >= I_{ILmax} + I_{RPU} = 360\ uA + (Vcc - V_{OL\ max}) / R_{min}$

Solving for R_{min} :
$R_{min} >= (5 - 0.4\ V) / (3.2\ mA - 360\ uA) = 4.6\ V / 2.84\ mA = 1.62\ K$
R_{min} is 1.62 K Ohms.

This value is also greater than specified as a test load of 1K.

The maximum acceptable value, R_{max}, is determined by the minimum output high voltage that will guarantee a CMOS high input plus noise margin. The resistor must be able to supply the LSTTL maximum input high current and not have too

large a voltage drop across it. This will determine the upper limit for the resistor value.

Specifically, the resistor voltage is:

Vcc - (CMOS $V_{IH\ min}$ + $V_{noise\ margin}$) = 5 - (3.0 + 0.4) = 1.6 V

This voltage is maintained while sourcing the LSTTL $I_{IH\ max}$ of 60 uA.

Solving for R_{max} :
R_{max} <= 1.6 V / 60 uA = 26.7 K Ohms max.

Thus, the acceptable range for the pull up resistor is
1.62 K Ohms <= R_{PU} <= 26.7 K Ohms.

An acceptable standard value such as 10K would be appropriate.

Another limit relates to the rise time of the signal under load, due to the R-C time constant of the pull-up resistor charging the load capacitance, C_L. From the example above, let's see what the effect of this time constant is on the selection of the resistor value.

The maximum R value can be approximated by the equation:

$R = T / C_L$ where T is the rise time and C_L is the total load Capacitance

Ignoring the Ioh current of the LSTTL driver, if the circuit above had an allowable rise time T = 50 nS and C_L = 20 pF, then the maximum R value would be:

R_{max} = 50 nS / 20 pF = **2.5 K Ohms max** to maintain the 50 nS rise time.

So a better choice might be a standard 2.2 K Ohm pull-up resistor. Since the driver will supply some current to charge the load capacitance, this is a fairly conservative value. We would also have to allow for the additional rise time as part of the timing analysis for the low-to-high transition.

Example 3-3 - Worst Case Timing Analysis

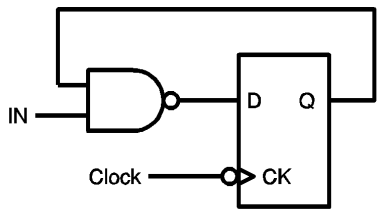


Figure 3-16) Example: Worst Case Timing

An LSTTL gate is used to enable the D input of a flip flop frequency divider. The timing of the input signals must conform to the combined specs of both devices, as defined in the tables below:

Flip-Flop Timing Specs:

Symbol	Min	typ	Max	units
T_{SU}	10			nS
T_H	1			nS
T_{PCKQ}			15	nS
T_{PWCK}	10			nS
F_{CLK}			50	MHz

Gate Timing Specs:

Symbol	min	typ	max	units
T_{PHL}	1	2	5	nS
T_{PLH}	2	4	6	nS

Test conditions R_L = 1K, C_L = 100 pF

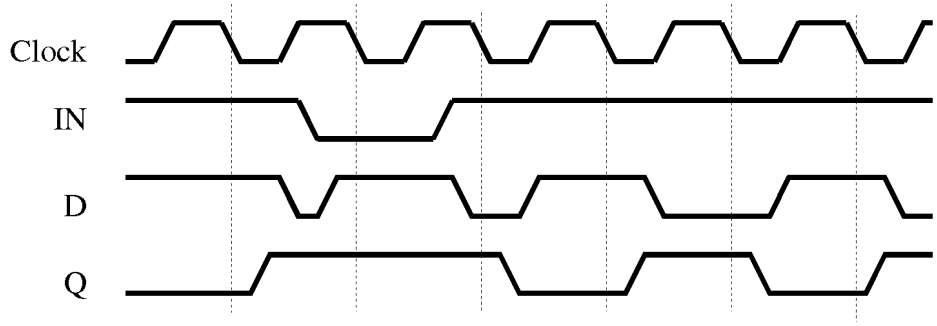


Figure 3-17) Functional Timing Diagram for Figure 3-16

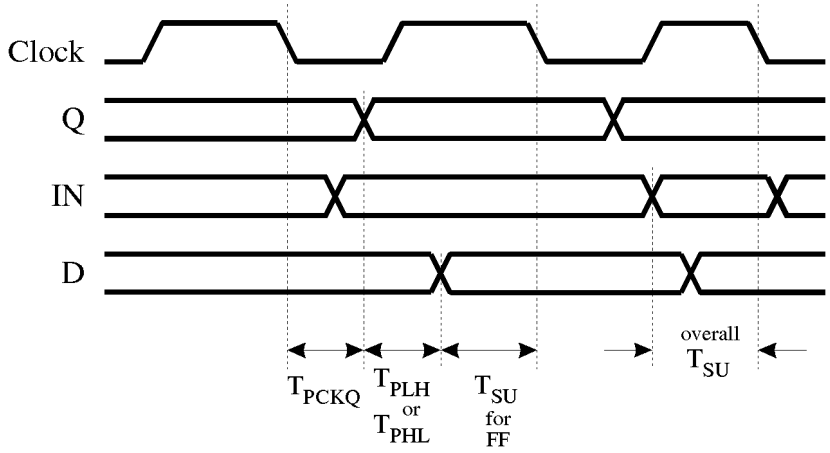


Figure 3-18) Specification Timing Diagram for Figure 3-16

Example 3-3 (cont'd): For the circuit and specifications shown in Figure 3-16, what is the maximum guaranteed clock rate?

3-3 Answer:

From the timing figures on the previous page, note the minimum clock cycle time is defined by the sum of the following: the time it takes for the transition from the active edge of the clock for the signal at D to propagate

through the flip flop, through the NAND gate and the time the signal must be stable before the next clock. The maximum propagation times and minimum setup times are used as they are the most severe requirements.

$T_{PCKQ} + T_{PLH} + T_{SU} = 15 + 6 + 10 = 31$ nS

$f = 1/t = 1/31\text{nS} = 32.26$ MHz

2) What are the setup and hold time requirements for the overall circuit as shown?

The overall setup time is lengthened by the delay of the NAND gate, therefore the system setup time is the sum of the flip flop setup time and the worst case propagation delay.

$T_{SU}(\text{system}) = T_{PLH} + T_{SU}(\text{flip-flop}) = 16$ nS minimum

For the overall system hold time, the hold time of the flip flop is offset by the minimum delay through the NAND gate, as this is the minimum amount of time that can be counted on to delay a changing D input to the flip flop.

$T_H(\text{system}) = T_H(\text{flip-flop}) - T_{PHL}(\text{min}) = 1 - 1 = 0$ nS

The delay in the D signal path reduced the hold time requirement from 1 nS to 0 nS, meaning the input can change at the same time as the clock edge or later. This is actually an improvement on the performance of the flip-flop by itself, which requires that the D line be held stable for 1 nS after the clock edge.

Chapter 3 Problems

For the following problems, refer to the loading example and Figure 3-15.

1. A) If a 10K pull-up resistor is used, how many additional LSTTL loads can be connected?

   B) How many CMOS loads could be added?

2. What could be done to increase the number of LSTTL loads?

For the following problem, refer to the timing example and Figure 3-13.

3. Using the same D flip-flop specified in the example, how fast could it be clocked if the /Q output was directly connected to the D input? (Eliminating the gate from the circuit.)

4. Under what conditions would the addition of a pull-up or pull-down resistor increase the fanout of a logic output?

5. What, if anything, can be done to increase fanout when it is limited by AC (capacitive) loading?

6. For a 32 bit CMOS 5 volt microprocessor that has a 32 bit address bus and a separate 32 bit data bus, and the processor has a 1 nS rise time and 0.5 nH of ground inductance on a board made from glass epoxy material. The processor has output high and low voltages of 4.5 and 0.5 volts respectively and drives a capacitance of 100 pF on the address and data buses. How long can the printed circuit traces be before they must be considered as transmission lines?

7. For the same processor and conditions described in problem 6, what is the worst case ground bounce voltage that can be expected?