Computer Programming

PDF version of October 2007

This PDF is version 0 from 14 October 2007.

The intent of this free downloadable college text book is to attempt to directly help poor and middle class students with the high cost of college text books by providing a high quality free alternative that can be used in the classroom for a subject that most college students are required to take.

This free downloadable book is based on and includes materials from http://www.OSdata.com . Materials from OSdata.com have already been used in more than 300 colleges and universities around the world and have been quoted in studies and policy decisions by the U.S. Navy and the government of the Federal Republic of Germany.

This is still a work in progress. Feedback and constructive criticism appreciated (especially feedback from professors who might want to use the finished book).

While this book is still being written, professors are free to use specific chapters (or portions of chapters) as class handouts to supplement existing for-profit text books. This same policy will continue to apply after the book is completed, but this policy offers usefulness for many classes right now today even though the book is still incomplete.

Poor students should not feel bad about using this book for free. You are exactly who this book is intended to help. You may optionally do volunteer work for the charitable organization of your choice (not political or religious activity — actual work for a charitable organization helping the poor, the elderly, the sick, the disabled, or the environment, etc.).

Distributed on the honor system. If you print out this book or read substantial portions on a computer screen (after the book is completed), please send a $10 donation to the author at: Milo, PO Box 1361, Tustin, CA, 92781, USA. Donations will help support further research and writing. You do not have to make multiple donations when you download new editions/versions of this book.

Those who make a donation have permission to print out future versions of this book, as well as back up and replacement copies of this book, for no additional donation (although additional donations would be appreciated).

Remember that any donations are voluntary and donations are not expected from those who are poor or otherwise might be burdened by the cost of making a donation. Corporations or rich people who want to help support the writing of this book are encouraged to make donations and will be specifically mentioned for their support.

author: Milo, PO Box 1361, Tustin, CA, 92781, USA

The e-mail address for contacting the author changes regularly to avoid spam. The current e-mail address for contacting the author is on the website OSdata.com.

credits

Elaine Higgins, Language Finger, Maureen and Mike Mansfield Library, University of Montana, http://www.lib.umt.edu/

Neal Ziring, The Language Guide, University of Michigan, http://www.engin.umd.umich.edu/CIS/course.des/cis400/index.html

copyright information

goal of this text book

The goal of this book is to provide a free downloadable text that can be used in college and high school computer programming classes.

According to the *Los Angeles Times* college text books average $120 each (late 2006) and the major book publishers are still jacking up the prices. According to the *LA Times*, many poor students are barred from higher education (even though they have financial aid or a scholarship) because they simply can’t afford the price of text books, which can be more than a thousand dollars a semester/quarter.

For-profit book publishing corporations use numerous malicious tricks to keep pushing up the price of text books and reduce the use of recycled used text books. Their racist purpose in engaging in these activities is to price text books out of the range of poor people so as to keep the mostly minority poor children from having the same equal access to education as rich white children.

The major book publishers put out new (more expensive) editions of text books every three years. It just happens to be that three years is the amount of time for a text book to saturate the used text book market and cut into sales of new books. The book publishers claim that this is mere coincidence with the timing of their new editions and that they only publish new editions when they need to make improvements on the existing text. They claim it is mere coincidence that these “necessary” improvements happen to exactly match the sales cycle for every text book they publish!

I do actively encourage students, teachers, and professional programmers to provide useful feedback and criticism to help make this project useful as a free downloadable college text book.

Donations of money to help support the writing and hosting of this project are greatly appreciated. See mailing address above.

using this text book

This book is divided into two major sections.

This organization reflects the way computer programming is normally taught: an overview class that gives a foundation in basic concepts, followed by a series of more advanced classes that go back over the same material in much more detail.

The first section provides an introduction and overview to computer programming.

This first section is further divided into general discussions and language specific discussions. It is generally unwise for beginners to attempt to learn more than one programming language at a time. Each programming language is color coded. The choice of colors is completely arbitrary and has nno meaning.

The second section provides a detailed examination and reference for advanced studies in computer programming and computer science.

Do not expect for this book to be assigned in the same order as it is written. There are many different ways to teach computer programming, so your particular professor or instructor is likely to change the order of presentation of the material, probably also deleting entire chapters and possibly inntroducing additional outside materials.

In particular, there is more material in the introductory section than can reasonably be covered in a single class. Your professor will decide which materials should be emphasized and which materials should be skipped. Some programming languages naturally emphasize some materials and don't include others.

Most schools start students on programming as quickly as possible. Don't be surprised if your professor skips over some or all of the early chapters and tells you that some or all is material that you should already know.

Once the introductory material has been covered, the advanced material can literally be taught in almost any order.

Important Note: For the sake of clarity, much of the material in the first section is watered down and simplified. Most exceptions (including some important ones) will be overlooked. Many details (including some important ones) will be ignored. Terminology will be used in a casual manner without formal definitions. Including all that information would just bog down the discussion and make it more difficult to understand the basic principles.

table of contents

picking a class

Before signing up for a class, ask other computer science majors about the professors. Getting hired as a professor is based on factors completely unrelated to the ability to teach. Further, the major factors for advancement in pay and prestige has *nothing* to do with teaching. Promotion is based primarily on the ability to continually be published. Colleges and universitites deliberately and intentionally ignore the ability to teach in making decisions about hiring and career advancement for professors!

Luckily, most professors learn something about education and teaching through years of dealing with students. And there are some professors who are naturally gifted at teaching.

You want to try to figure out from other students which are the best professors and then try to get into the classes taught by the professors who are best at teaching.

Don't be concerned about a reputation as a “tough grader”. How tough a professor is at grading has nothing to do with how good they are at teaching. A professor who is naturally gifted at teaching will prepare you for tougher grade standards.

Also take into consideration when a class is taught.

Teenagers and yougn adults experience “phase shift” from growth hormones. The release of growth hormones has the side effect of changing the natural inner circadian clock, making the teenager or yonng adult stay up late at night and making it very difficult to wake early in the morning. If you attempt to take an early morning class while experiencing phase shift, you will have a great deal of trouble learning and might fail a class that you'd easily pass if taken later in the day.

Night or evening classes are typically taught by working professionals rather than tenured professors. These instructors tend to have a less academic and more practical approach to teaching. If you need more help on the academic parts, take a day class. If you need more help on the nuts and bolts of programming, then consider taking an evening class.

syllabus

The syllabus is your friend. Reading the syllabus can easily result in a letter grade improvement.

Class identification The beginning of the syllabus will identify the class (including any identification numbers used by the school) and the professor (and any teaching assistants, proctors, or others involved in the class). As trivial as it may sound, make sure that the class ID matches the class you signed up for!

Professor’s name The syllabus will include the correct spelling of the professor’s name. Double check this before turning in anything in writing. There are few things more personal to someone than their own name. If you mess up on the professor’s name, the subconscious resentment will hurt your grade (even if the professor consciously tries to ignore the slight).

Contact information The syllabus will include information on how to contact the professor. This will typically include a telephone number, an e-mail address, and a room/building number for the professor’s office.

Office hours Every professor will have office hours. These may be set hours with an open door or it may be a notation that office hours are by appointment. Even if there are open office hours, it is worthwhile to arrange an appointment with the professor just to make sure that he or she is there at the appointed time and to gain a priority over students showing up without an appointment. Never be late to an appointment for office hours.

Note that you should make your appointment at the *end* of class, not at the beginning. A professor is always busy preparing for a class and simply won't have time to focus on student requests. At the end of the class the professor can better handle questions and requests for an appointment.

Professors almost always want their students to succeed. A professor will make a reasonable attempt to help a student learn. Note that if the amount of time and effort exceeds certain limits that the professor may recommend private tutoring. If the professor recommends private tutoring, ask for recommendations on who to take the lessons from. The professor will certainly know some higher level or graduate students who can use the extra income. Also, the school may have free tutoring available.

If you have learning or physical disabilities, set an appointment for office hours so that the professor will be aware of the difficulties. The professor may be able to arrange for alternatives to the regular coursework. Also the professor will be able to refer you to professional staff at the school who can offer you additional assistance in dealing with your learning or physical disability.

If you have life problems that come up (such as death in the family, severe illness, work related problems, etc.), make an appointment for office hours with the professor. While there may be little that the professor can do to help, there are possibilities that the professor may be able to change some deadlines or give an incomplete grade or offer some other kind of assistance. At a minimum the professor will be aware that you are trying your best to keep up with the classwork under difficult circumstances.

Note that in most schools an incomplete grade is at the professor's option. Schools discourage incomplete grades. The professor will have the final decision on whether the life difficulty is sufficient to warrant an incomplete and whether the student will have a reasonable ability to make up the incomplete grade in the future. Usually it is a better option to withdraw from the class and take it again when you are prepared.

Website Most professors will provide the URL for the class. This website or webpae(s) may have copies of the syllbus, programming asignments, and other materials.

Some professors may place their sldies and/or lecture notes on their website (usually after each specific class session).

Some professors may place answers to examinations, homework, or other assignments on the website (again, after the specific class session).

Some professors may list recommended websites that will have additional information useful for the class. These recommendations may be printed on the syllabus or may be on a links page on the professor's website.

Course description The course description is typically written in edu-speak, but it worth figuring out. This is your best hint for what the class will cover and what you will be expected to learn.

Use the course description to figure out if you are in the right class. Obviously if the class is a required one for your major, then you have no choice in the matter. But if you have a choice, the course description will help you figure out if you are taking the correct class from multiple alternatives.

The course description can also help you prepare your studies by letting you know the highest priorities of the professor. When you prepare for quizes, tests, midterms, and finals, the course description can help answer that vexing question of what is the most likely subject matter to appear on the examination.

Topics The syllabus may have a list of topics to be covered. This is typically easier to understand than the edu-speak of the course description and serves the same uses.

Require materials Check to see what is required. If there are particular forms required for taking examinations, these specific requirements will be listed on the syllabus.

If there are specific requirements for computers, hardware, software, or operating systems, these requirements will be listed. Sometimes you might be arrange for a reasonable alternative, but don't be surprised if the professor rejects alternatives from the requirements on the syllabus. It dramatically increases the work load on the professor to allow custom alternatives, so those typically won't be allowed unless there is a compelling reason (such as a physical disability).

Watch for media requirements, such as CD-Rs, floppy disks, DVDs, or other media. In particular pay attention to formats (such as DVD-R vs. DVD+R). A small mistake in format could result in a drop in letter grade or even failure.

If you do not understand what a required material is, ask the professor in class when he or she asks if anyone has questions about the syllabus. If you do not know where to obtain a particular required material, ask the professor. Professors will often know where you can get the best student discount on each item.

Textbook The syllabus will list required and optional texts.

Required texts are ones that the professor will expect you to obtain, usually by the second time the class meets.

Optional texts are ones that the professor knows will help you understand the material, but aren't essential to pass the class. if you can afford these optional texts, you should obtain and read them. Sometimes a professor will list classic reference books that are directly related to the class. These are great books for your long term professional library.

If you can't afford the text books assigned, you should check to see if there are one or more reserve copies available at the school library. You can't take the reserve copy from the library and often there are time limits on how long you can read it at the library, especially near finals.

Reserve copies may not be available because large book publishing corporations refuse to provide school libraries with a free copy and school libraries can't afford to keep purchasing replacement editions every three years. Often the reserve copy was purchased by your professor out of his or her own money. Students who can afford to forego selling books back to the college bookstore may want to consider donating their books to the library at the end of the class.

Grading Every professor will have different criteria for grading. You dramatically improve your chances of obtaining a higher letter grade by carefully reading the grading criteria.

The grading system may be a point system. If so, the syllabus will list the exact range of points needed for each letter grade. There may be limits on how many points can be obtained from particular activities.

The grading system may be a percentage system. If so, the syllabus will list the exact range of percentages needed for each letter grade. Rarely you may find that the possible percentages exceed 100%. In this case the professor is offering some alternatives on the methods for obtaining hgiher grades.

Attendance Almost every professor gives some credit towards mere attendance. This means that you arrived and are in your seat on time as the class starts and that you stay until the end of the class session.

Showing up for every class is the easiest way to improve your grade (not merely because of the credit for attendance, but also because of exposure to the professor's complete lectures.

Typically there will be a maximum number of absences allowed before the student is either automatically dropped or automatically fails. There may be specific listings of requirements for excused absences. Pay attention to these limits.

Late arrival Also pay attention to the policy regarding being late to class. The professor may give lower credit for late arrival. The professor may give *no* credit for late arrival.

Even if the professor gives no credit for late arrival, it is better to get at least part of lecture than to miss it entirely. if you do arrive late, be quiet and unobtrusive when taking your seat. Do not distract from the ongoing lecture.

Class participation Most professors also give credit for class participation. This is a very subjective grading standard and somewhat unfair to those who are socially shy.

Always pay attention and take good notes. The professor will notice this, especially in smaller classes.

Try to make sure that you ask relevant questions during class. Asking excessive questions is disruptive and annoying, but professors like a good mix of student questions, if only to get a feel for how well the class is learning the material.

If the professor asks that questions be saved for a question and aanswer period at the end of the class, write down your questions on a separate piece of paper so that you will remember them when the time for Q&A occurs.

If you are too shy to ask questions during class, write down your questions and ask at the end of class or during office hours. Professors generally prefer that questions be asked during class so that the educational experience can be shared by all of the students, but it is better to ask questions later than not at all.

Programming assignments The syllabus will list how many programming assignments will be required, when they will be due, and how much credit you will receive for each. There may be a brief description or title of each programming assignment. There will probably be a breakdown on how each programming assignment is graded.

Generally you will receive the instructions for each programming assignment on separate handouts throughout the class. This gives the professor the flexibility to modify programming assignments based on the progress of each individual class.

Programming assignments are always trivial and artificial in nature. There simply isn't enough time in a typical class to assign large scale programming assignments, so each programming assignment will have specific educational goals in mind. Very artificial standards will be required to focus attention on the specific educational goals of each programmign assignement.

When you receive a programming assignment, carefully read all of the requirements. Make sure that you udnerstand exactly what is expected of you. Immediately ask any questions about the requirements of the programming assignment (professors won't answer questions on *how* to do the assignment, but they will help you understand exactly what you are expected to accomplsih).

Because of the artificial nature of programming assignments the requirements may be highly unusual. You may see restrictions that simply wouldn't exist in real world programming. You may be asked to do specific things that would never occur in real world programming. It doesn't do you any good to rebel against the artifial nature of programming assignments.

The artificial nature of programming assignments can also give you useful feedback on whether you are solving the correct problem. If you find yourself spending a lot of time designing or coding something that isn't explicitly required by the programming assignment you may have wandered away from the assignment into a whole bunch of work that has nothing to do with your grade..

You will also find that the basic knowledge for every programming assignment should be provided in some combination of the lcetures and the textbook(s). You can use the progamming assignments as a guide for particular subject matters to pay attention to during lectures and during reading.

Class schedule Some professors may list the exact topics that will be covered in each class session. You can use this as a guide for what to read and study to prepare for each class.

The important things to notice in the class schedule:

- what topics will be covered in each class (if listed)
- days when the class won't meet
- when reading assignments are due
- when programming assignments are due
- when examinations will occur (especially midterms and finals)
- when writing assignments are due
- any other matters the professor feels is important enough to include in the syllabus

Reading assignments The professor will list the specific chapters of each required text that you are expected to read and the time by which you must have that material read.

Don't fall behind on your reading assignments. While some professors repeat in their lectures the same material covered in the textbook, many professors use the textbook as a starting point and delve into further details and nuance during lecture. If you haven't read the assigned text then you may get lost during the lecture. Also asking questions that were covered in the required reading may adversely affect your class participation grade (unless you are asking for a better explanation of something you didn't understand from the required reading).

Writing assignments The syllabus will list any required writing assignments. Introductory classes and classes intended for a general audience will almost always have some kind of written report or essay. Follow all of the rules that you hear in your English composition classes, as well as any custom instructions that appear on the syllabus.

Examinations The syllabus will list all examinations for the class and how much they count for your grade.

Note the exact date and class session of any midterm, final examination, or other major examination. Note any required materials (such as particular answer forms). Note whether the examinations are cumulative or not.

Particularly notice the date and time of the final exam. It is very common for the final exam to be longer than a normal class and to be scheduled at a time (and possibly even day) different than normal class sessions.

You may find an indication that there may be surprise quizes. Quizes, especially suprise quizes, are generally given at the beginning of the class session. This technique helps make sure that students aren't late for class. Quizes are also often given at the end of the class, possibly allowing slower students extra time to finish. Quizes at the beginning of a class tend to cover material from the assigned reading, while quizes aat the end of a class tend to cover the material in that session's lecture.

Academic rules The syllabus will generally have a summary of the school's policies on academic dishonesty and disruptive behavior. If you need a more complete copy, the school's administration can provide you with the full set of academic rules. If you are unclear on any rule, ask your professor.

Often there will be an additional handout that discusses the rules for use of the school's computing resources. Obey all of these rules, even if they seem absurd.

Note that cheating, especially on programming assignments, is extremely easy to spot. Because every student has a different personality and individual programming style, every student's work will have a distinctive look, even on the most trivial of programming assignments. Even a stupid grader will immediately notice if two or more student programs have a similar look.

Extra credit The syllabus will list any possibilities for extra credit work. If you think you might need the extra credit, start work early. You simply won't have the time to do extra credit work when finals crunch comes along.

If the professor doesn't list any extra credit, you can ask.

computer programming

Programming is problem solving and writing instructions for a computer.

The principles of programming are independent of the computer programming language used. Different languages have different strengths and weaknesses, making some kinds of programs easier or more difficult to write, but the basic principles remain the same regardless of language.

A skilled programmer should be able to switch to a new programming language in a few hours.

On the other hand, beginners should pick one language and learn it before attempting a second language. Normally this choice will be made by the school or the professor.

This free text book includes information on multiple programming languages. Unless instructed otherwise, you should concentrate on the language you are learning and skip over the others. Trying to learn the syntax and semantics of multiple programming languages at the same time as learning the basics of programming is a recipe for utter confusion.

size of programs

Programs are generally divided into three basic sizes: trivial, small, and large.

Trivial programs are programs that a skilled programmer can write in less than two days of coding.

Small programs are programs that one skilled programmer can write in less than one year of full time work.

Large programs are programs that require more than two to five man-years of labor, normally written by programming teams (which can exceed 1,000 skilled workers).

These estimates are approximate and there are obvious gaps in the gray zone between the sizes. Further, there can be huge difference sin individual abilities.

Larry Ellison wrote the first version of Oracle database by himself in about six months. That is a genius exception. Data bases typically take large teams (sometimes hundreds of programmers) at least a year.

Bill Gates, copying and pasting from the source code of three working open source versions, took more than six months to create a bug-filled BASIC compiler and then hired a team of six skilled programmers who spent more than six more months to get rid of enough bugs to make the compiler somewhat usable (a total of more than three man-years). That is an idiot exception. A BASIC compiler typically takes a skilled programmer a few hours to create. Note that Bill Gates takes credit for quickly having created a BASIC compiler, but according to other sources he was sued for having illegally used open source code for commercial purposes, forcing him to spend a great deal of time attempting to do a project that many programmers of the day could successfully finish in hours.

impact on good programming practices

Almost every program assigned in a class setting will be trivial, simply because there isn't enough time in a quarter or semester for longer programs.

Each programming assignment will concentrate on one or a small number of specific programming concepts.

The artificial nature of school programming assignments cause most students to question the utility of modern programming practices, especially the time and effort spent on form and documentation.

These practices are the result of decades of real world programming.

Successful programs tend to have a long lifetime. Programmers will have to look at existing source code, figure out what is going on, and then correctly modify the program to add new features or update existing features to meet changing real world conditions.

Basics of computer hardware

A **computer** is a programmable machine (or more precisely, a programmable sequential state machine). There are two basic kinds of computers: analog and digital.

Analog computers are analog devices. That is, they have continuous states rather than discrete numbered states. An analog computer can represent fractional or irrational values exactly, with no round-off. Analog computers are almost never used outside of experimental settings.

A **digital computer** is a programmable clocked sequential state machine. A digital computer uses discrete states. A binary digital computer uses two discrete states, such as positive/negative, high/low, on/off, used to represent the binary digits zero and one.

The French word **ordinateur**, meaning that which puts things in order, is a good description of the most common functionality of computers.

what are computers used for?

Computers are used for a wide variety of purposes.

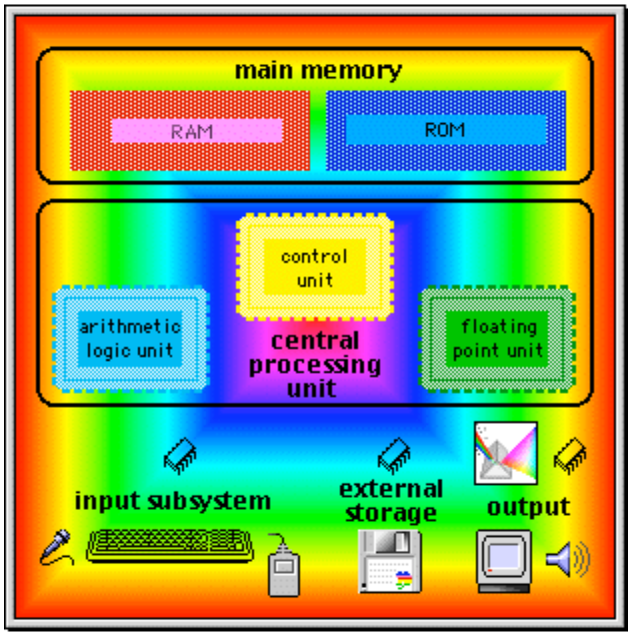
Data processing is commercial and financial work. This includes such things as billing, shipping and receiving, inventory control, and similar business related functions, as well as the "electronic office".

Scientific processing is using a computer to support science. This can be as simple as gathering and analyzing raw data and as complex as modelling natural phenomenon (weather and climate models, thermodynamics, nuclear engineering, etc.).

Multimedia includes **content creation** (composing music, performing music, recording music, editing film and video, special effects, animation, illustration, laying out print materials, etc.) and multimedia playback (games, DVDs, instructional materials, etc.).

parts of a computer

The classic crude oversimplication of a computer is that it contains three elements: processor unit, memory, and I/O (input/output). The borders between those three terms are highly ambigious, non-contiguous, and erratically shifting.



A slightly less crude oversimplification divides a computer into five elements: arithmetic and logic subsystem, control subsystem, main storage, input subsystem, and output subsystem.

processor

The **processor** is the part of the computer that actually does the computations. This is sometimes called an **MPU** (for main processor unit) or **CPU** (for central processing unit or central processor unit).

A processor typically contains an arithmetic/logic unit (**ALU**), control unit (including processor flags, flag register, or status register), internal buses, and sometimes special function units (the most common special function unit being a floating point unit for floating point arithmetic).

Some computers have more than one processor. This is called **multi-processing**.

The major kinds of digital processors are: CISC, RISC, DSP, and hybrid.

CISC stands for Complex Instruction Set Computer. Mainframe computers and minicomputers were CISC processors, with manufacturers competing to offer the most useful instruction sets. Many of the first two generations of microprocessors were also CISC.

RISC stands for Reduced Instruction Set Computer. RISC came about as a result of academic research that showed that a small well designed instruction set running compiled programs at high speed could perform more computing work than a CISC running the same programs (although very expensive hand optimized assembly language favored CISC).

DSP stands for Digital Signal Processing. DSP is used primarily in dedicated devices, such as MODEMs, digital cameras, graphics cards, and other specialty devices.

Hybrid processors combine elements of two or three of the major classes of processors.

arithmetic and logic

An arithmetic/logic unit (**ALU**) performs integer arithmetic and logic operations. It also performs shift and rotate operations and other specialized operations. Usually floating point arithmetic is performed by a dedicated floating point unit (**FPU**), which may be implemented as a co-processor.

An arithmetic/logic unit (**ALU**) performs integer arithmetic and logic operations. It also performs shift and rotate operations and other specialized operations. Usually floating point arithmetic is performed by a dedicated floating point unit (**FPU**), which may be implemented as a co-processor.

control

Control units are in charge of the computer. Control units fetch and decode machine instructions. Control units may also control some external devices.

A **bus** is a set (group) of parallel lines that information (data, addresses, instructions, and other information) travels on inside a computer. Information travels on buses as a series of electrical pulses, each pulse representing a one bit or a zero bit (there are trinary, or three-state, buses, but they are rare). An **internal bus** is a bus inside the processor, moving data, addresses, instructions, and other information between registers and other internal components or units. An **external bus** is a bus outside of the processor (but inside the computer), moving data, addresses, and other information between major components (including cards) inside the computer. Some common kinds of buses are the system bus, a data bus, an address bus, a cache bus, a memory bus, and an I/O bus.

main storage

Main storage is also called **memory** or internal memory (to distinguish from external memory, such as hard drives).

RAM is Random Access Memory, and is the basic kind of internal memory. RAM is called “random access” because the processor or computer can access *any* location in memory (as contrasted with sequential access devices, which must be accessed in order). RAM has been made from reed relays, transistors, integrated circuits, magnetic core, or anything that can hold and store binary values (one/zero, plus/minus, open/close, positive/negative, high/low, etc.). Most modern RAM is made from integrated circuits. At one time the most common kind of memory in mainframes was magnetic core, so many older programmers will refer to main memory as **core memory** even when the RAM is made from more modern technology. **Static RAM** is called static because it will continue to hold and store information even when power is removed. Magnetic core and reed relays are examples of static memory. **Dynamic RAM** is called dynamic because it loses all data when power is removed. Transistors and integrated circuits are examples of dynamic memory. It is possible to have battery back up for devices that are normally dynamic to turn them into static memory.

ROM is Read Only Memory (it is also random access, but only for reads). ROM is typically used to store thigns that will never change for the life of the computer, such as low level portions of an operating system. Some processors (or variations within processor families) might have RAM and/or ROM built into the same chip as the processor (normally used for processors used in standalone devices, such as arcade video games, ATMs, microwave ovens, car ignition systems, etc.). **EPROM** is Erasable Programmable Read Only Memory, a special kind of ROM that can be erased and reprogrammed with specialized equipment (but not by the processor it is connected to). EPROMs allow makers of industrial devices (and other similar equipment) to have the benefits of ROM, yet also allow for updating or upgrading the software without having to buy new ROM and throw out the old (the EPROMs are collected, erased and rewritten centrally, then placed back into the machines).

Registers and **flags** are a special kind of memory that exists inside a processor. Typically a processor will have several internal registers that are much faster than main memory. These registers usually have specialized capabilities for arithmetic, logic, and other operations. Registers are usually fairly small (8, 16, 32, or 64 bits for integer data, address, and control registers; 32, 64, 96, or 128 bits for floating point registers). Some processors separate integer data and address registers, while other processors have general purpose registers that can be used for both data and address purposes. A processor will typically have one to 32 data or general purpose registers (processors with separate data and address registers typically split the register set in half). Many processors have special floating point registers (and some processors have general purpose registers that can be used for either integer or floating point arithmetic). Flags are single bit memory used for testing, comparison, and conditional operations (especially conditional branching).

external storage

External storage (also called **auxillary storage**) is any storage other than main memory. In modern times this is mostly hard drives and removeable media (such as floppy disks, Zip disks, optical media, etc.). With the advent of USB and FireWire hard drives, the line between permanent hard drives and removeable media is blurred. Other kinds of external storage include tape drives, drum drives, paper tape, and punched cards. Random access or indexed access devices (such as hard drives, removeable media, and drum drives) provide an extension of memory (although usually accessed through logical file systems). Sequential access devices (such as tape drives, paper tape punch/readers, or dumb terminals) provide for off-line storage of large amounts of information (or back ups of data) and are often called I/O devices (for input/output).

input/output overview

Most external devices are capable of both input and output (I/O). Some devices are inherently input-only (also called read-only) or inherently output-only (also called write-only). Regardless of whether a device is I/O, read-only, or write-only, external devices can be classified as block or character devices.

A **character** device is one that inputs or outputs data in a stream of characters, bytes, or bits. Character devices can further be classified as serial or parallel. Examples of character devices include printers, keyboards, and mice.

A **serial** device streams data as a series of bits, moving data one bit at a time. Examples of serial devices include printers and MODEMs.

A **parallel** device streams data in a small group of bits simultaneously. Usually the group is a single eight-bit byte (or possibly seven or nine bits, with the possibility of various control or parity bits included in the data stream). Each group usually corresponds to a single character of data. Rarely there will be a larger group of bits (word, longword, doubleword, etc.). The most common parallel device is a printer (although most modern printers have both a serial and a parallel connection, allowing greater connection flexibility).

A **block** device moves large blocks of data at once. This may be physically implemented as a serial or parallel stream of data, but the entire block gets transferred as single packet of data. Most block devices are random access (that is, information can be read or written from blocks anywhere on the device). Examples of random access block devices include hard disks, floppy disks, optical discs (such as CD-R and DVD-R), and drum drives. Examples of sequential access block devcies include magnetic tape drives and high speed paper tape readers. Note that ordinary DVD videos are intended to be played primarily in a sequential manner, but the menu and special features are typically organized for random access.

input

Input devices are devices that bring information into a computer.

Pure input devices include such things as punched card readers, paper tape readers, keyboards, mice, drawing tablets, touchpads, trackballs, and game controllers.

Devices that have an input component include magnetic tape drives, touchscreens, and dumb terminals.

output

Output devices are devices that bring information out of a computer.

Pure output devices include such things as card punches, paper tape punches, LED displays (for light emitting diodes), monitors, printers, and pen plotters.

Devices that have an output component include magnetic tape drives, combination paper tape reader/punches, teletypes, and dumb terminals.

kinds of programming

There are two basic kinds of programming: system and application.

System programming deals with the use of a computer system. This includes such things as the operating system, device drivers for input and output devices, and systems utilities.

Application programming deals with the programs directly used by most people.

Application programming is generally divided further into scientific and business programming.

Scientific programming is work in the scientific, engineering, and mathematical fields. Often the programmers are the researchers who use the programs.

Business programming is work in the data processing field, including large scale business systems, web-based businesses, and office applications. It is exceedingly rare for these kinds of programs to be created by the person who uses them.

Another large category of programming is programming for **personal or home** use. This includes **games**. Historically, many advances in computer science occurred in the development of computer and video games.

Embedded systems are programs that are built into specific hardware, such as the computer systems in an automobile or microwave oven. These programs combine features of operating systems and application program into a single monolithic system.

Scripting is a simplified version of programming originally intended for use primarily by non-programmers. In practice, most non-programmers have trouble with scripting languages. Some professional programmers have built very useful, sometimes intricate systems using scripting languages, especially those contained in office software (such as word processors or spreadsheets).

programming languages

direct programming

Originally computers were programmed directly in a “language” that the computer understood.

This direct programming could involve directly wiring the program into the computer. In some cases, this involved a soldering iron. In other cases there was some kind of plug-board ot make it easier to change the programmed instructions. This method was known as **hard wiring**.

Large telegraph networks and later large telephone networks became so complex as to essentially be a computer on a system-wide basis. Many of the ideas (especially logic circuits) that were later necessary to create computers were first developed for large scale telegraph and telephone systems.

In some early computers the programming could be accomplished with a set of switches. The use of **front panel** switches (and corresponding indicator lights) continued as an option on many mainframe and minicomputer systems. Some microcomputer systems intended for hobbyists and for dedicated systems also had some kind of front panel switches.

Another method was the use of **punched cards**. This was a technology originally developed for controlling early industrial age factories, particularly large looms. The designs or patterns for the cloth would be programmed using punched cards. This made it easy to switch to new designs. Some of the large looms became so complex that they were essentially computers, although that terminology wasn’t used at the time.

machine code and object code

Both the front panel switch and the punched card methods involved the use of numeric codes. Each numeric code indicated a different machine instruction. The numbers used internally are known as **machine code**. The numbers on some external media, such as punched cards (or disk files) are known as **object code**.

assembly and assemblers

One of the early developments was a **symbolic assembler**. Instead of writing down a series of binary numbers, the programmer would write down a list of machine instructions, using human-readable symbols. A special program, the **assembler**, would convert these symbolic instructions into object or machine code.

Assembly languages have the advantage that they are easier to understand than raw machine code, but still give access to all of the power of the computer (as each assembler symbol translates directly into a specific machine instruction).

Assembly languages have the disadvantage that they are still very close to machine language. These can be difficult for a human to follow and understand and time-consuming for a human to write. Also, programs written in assembly are tied to a specific computer hardware and can’t be reused on another kind of computer.

The human readable version of assembly code is known as **source code** (it is the source that the assembler converts into object code).

high level languages

High level languages are designed to be easier to understand than assembly languages and allow a program to run on multiple different kinds of computers.

The source code written in high level languages need to be translated into object code. The two basic approaches are compilers and interpetters. Some programming languages are available in both interpretted and compiled versions.

High level languages have usually been designed to meet the needs of some particular kind of programming. For example, FORTRAN was originally intended for scientific programming. COBOL was originally intended for business programming and data processing. SQL was originally intended for data base queries. C was originally intended for systems programming. LISP was originally intended for list processing. PHP was originally intended for web scripting. Ada was originally intended for embedded systems. BASIC and Pascal were originally intended as teaching languages.

Some high level languages were intended to be general purpose programming languages. Examples include PL/I and Modula-2. Some languages that were originally intended for a specific purpose have turned into general purpose programming languages, such as C and Pascal.

interpreters

Interpreters convert each high level instruction into a series of machine instructions and then immediately run (or execute) those instructions. In some cases, the interpreter has a library of routines and looks up the correct routine from the library to handle each high level instruction.

compilers

Compilers convert a finished program (or section of a program) into object code. This is often done in steps. Some compilers convert high level language instructions into assembly language instructions and then an assembler is used to create the finished object code.

Some compilers convert high level language instructions into an **intermediate language**. This intermediate language is platform-independent (it doesn't matter which actual computer hardware is eventually used). The intermediate language is then converted into object code for a specific kind of computer. This approach makes it easier to move (or **port**) a compiler from one kind of computer to another. Only the last step (or steps) need to be rewritten, while the main complier is reused.

Compiled code almost always runs faster than interpretted code. An **optimizing compiler** examines a high level program and figures out ways to optimize the program so that it runs even faster.

linkers

As programs grow in size, requiring teams of programmers, there is a need to break them up into separate files so that different team members can work on their individual assignments without interfering with the work of others. Each file is compiled separately and then combined later.

Linkers are programs that combine the various parts of a large program into a single object program. Linkers also bring in support routines from **libraries**. These libraries contain utility and other support code that is reused over and over for lots of different programs.

Historically, linkers also served additional purposes that are no longer necessary, such as resolving relocatable code on early hardware (so that more than one program could run at the same time).

loaders

A **loader** is a program that loads programs into main memory so that they can be run. In the past, a loader would have to be explicitely run as part of a job. In modern times the loader is hidden away in the operating system and called automatically when needed.

editors

An **editor** is a program that is used to edit (or create) the source files for programming. Editors rarely have the advanced formatting and other features of a regular word processor, but sometimes include special tools and features that are useful for programming.

command line interface

A **command line interface** is an old-style computer interface where the programmer (or other person) controls the computer by typing lines of text. The text lines are used to give instructions (or commands) to the computer. The most famous example of a command line interface is the UNIX shell.

In addition to built-in commands, command line interfaces could be used to run programs. Additional information could be passed to a program, such as names of files to use and various “program switches” that would modify how a program operated.

development environment

A **development environment** is an integrated set of programs (or sometimes one large monolithic program) that is used to support writing computer software. Development environments typically include an editor, compiler (or compilers), linkers, and various additional support tools. Development environments may include their own limited command line interface specifically intended for programmers.

The term “development environment” can also be used to mean the collection of programs used for writing software, even if they aren’t integrated with each other.

Because there are a huge number of different development environments and a complete lack of any standardization, the methods used for actually typing in, compiling, and running a program are *not* covered by this book. Please refer to your local documentation for details.

standards and variants

Programming languages have traditionally been developed either by a single author or by a committee.

Typically after a new programming language is released, new features or modifications, called **variants**, start to pop-up. The different versions of a programming language are called **dialects**. Over time, the most popular of these variants become common place in all the major dialects.

If a programming language is popular enough, some international group or committee will create an official **standard** version of a programming language. The largest of these groups are **ANSI** (Ameican national Standards Institute) and **ISO** (International Orgnaization for Standardization).

While variants and dialects may offer very useful features, the use of the non-standard features will lock the program into a particular development environment or compiler and often will lock the program into a specific operating system or even hardware platform.

Use of official standaards allows for **portability**, which is the ability to move a program from one machine or operating system to another.

While variants were traditionally introduced in an attempt to improve a programming language, Microsoft started the practice of intentionally creating variants to lock developers into using Microsoft products. In some cases the Microsoft variants offered no new features, but merely chaanged from the established standard for the sake of being different. Microsoft lost a lawsuit with Sun Microsystems for purposely creating variants to Java in hopes of killing off Java in favor of Microsoft languages.

brief history of programming languages and other significant milestones

There have been literally thousands of programming languages, many of which have been lost to history.

200s BCE

The Antikythera mechanism, discovered in a shipwreck in 1900, is an early mechanical analog computer from between 150 BCE and 100 BCE. The Antikythera mechanism used a system of 37 gears to compute the positions of the sun and the moon through the zodiac on the Egyptian calendar, and possibly also the fixed stars and five planets known in antiquity (Mercury, Venus, Mars, Jupiter, and Saturn) for any time in the future or past. The system of gears added and subtracted angular velocities to compute differentials. The Antikythera mechanism could accurately predict eclipses and could draw up accurate astrological charts for important leaders. It is likely that the Antikythera mechanism was based on an astrological computer created by Archimedes of Syracuse in the 3rd century BCE.

1400s

The Inca created digital computers using giant loom-like wooden structures that tied and untied knots in rope. The knots were digital bits. These computers allowed the central government to keep track of the agricultural and economic details of their far-flung empire. The Spanish conquered the Inca during fighting that stretched from 1532 to 1572. The Spanish destroyed all but one of the Inca computers in the belief that the only way the machines could provide the detailed information was if they were Satanic divination devices. Archaeologists have long known that the Inca used knotted strings woven from cotton, llama wool, or alpaca wool called khipu or quipus to record accounting and census information, and possibly calendar and astronomical data and literature. In recent years archaeologists have figured out that the one remaining device, although in ruins, was clearly a computer.

1800s

Charles Babbage created the difference engine and the analytical engine, often considered to be the first modern computers. Augusta Ada King, the Countess of Lovelace, was the first modern computer programmer.

1945

Plankalkül (Plan Calculus), created in 1945 by Konrad Zuse for the Z3 computer in Nazi germany, may have been the first programming language (other than assemblers). This was a surprisingly advanced programming language, with many features that didn’t appear again until the 1980s.

1949

Short Code created in 1949. This programming language was compiled into machine code by hand.

1951

Grace Hopper started work on A-0 in 1951.

1952

Autocode, a symbolic assembler for the Manchester Mark I computer, was created in 1952 by Alick E. Glennie. Later used on other computers.

A-0 (also known as AT-3), the first compiler, was created in 1952 by Grace Murray Hopper. She later created A-2, ARITH-MATIC, MATH-MATIC, and FLOW-MATIC, as well as being one of the leaders in the development of COBOL. Grace Hopper was working for Remington rand at the time. Rand released the language as MATH-MATIC in 1957.

1954

FORTRAN (FORmula TRANslation) was created in 1954 by John Backus and other researchers at International Business Machines (now IBM). Released in 1957. FORTRAN is the oldest programming language still in common use. Identifiers were limited to six characters. Elegant representation of mathematic expressions, as well as relatively easy input and output. FORTRAN was based on A-0.

> “Often referred to as a *scientific language,* FORTRAN was the first *high-level* language, using the first compiler ever developed. Prior to the development of FORTRAN computer programmers were required to program in machine/assembly code, which was an extremely difficult and time consuming task, not to mention the dreadful chore of debugging the code. The objective during its design was to create a programming language that would be: simple to learn, suitable for a wide variety of applications, *machine independent,* and would allow complex mathematical expressions to be stated similarly to regular algebraic notation. While still being almost as efficient in execution as assembly language. Since FORTRAN was so much easier to code, programmers were able to write programs 500% faster than before, while execution efficiency was only reduced by 20%, this allowed them to focus more on the problem solving aspects of a problem, and less on coding.
>
> “FORTRAN was so innovative not only because it was the first high-level language [still in use], but also because of its compiler, which is credited as giving rise to the branch of computer science now known as *compiler theory*. Several years after its release FORTRAN had developed many different *dialects,* (due to special *tweaking* by programmers trying to make it better suit their personal needs) making it very difficult to transfer programs from one machine to another.” —Neal Ziring, The Language Guide, University of Michigan
>
> “Some of the more significant features of the language are listed below:” —Neal Ziring, The Language Guide, University of Michigan
>
> - **Simple to learn** - when FORTRAN was design one of the objectives was to write a language that was easy to learn and understand.
> - **Machine Independent** - allows for easy transportation of a program from one machine to another.
> - **More natural ways to express mathematical functions** - FORTRAN permits even severely complex mathematical functions to be expressed similarly to regular algebraic notation.
> - **Problem orientated language**
> - **Remains close to and exploits the available hardware**
> - **Efficient execution** - there is only an approximate 20% decrease in efficiency as compared to assembly/machine code.
> - **Ability to control storage allocation** -programmers were able to easily control the allocation of storage (although this is considered to be a dangerous practice today, it was quite important some time ago due to limited memory.

- **More freedom in code layout** - unlike assembly/machine language, code does not need to be laid out in rigidly defined columns, (though it still must remain within the parameters of the FORTRAN source code form).

1956

Researchers at MIT begin experimenting with direct keyboard input into computers.

IPL (Information Processing Language) was created in 1956 by A. Newell, H. Simon, and J.C. Shaw. IPL was a low level list processing language which implemented recursive programming.

1957

MATH-MATIC was released by the Rand Corporation in 1957. The language was derived from Grace Murray Hopper’s A-0.

FLOW-MATIC, also called B-0, was created in 1957 by Grace Murray Hopper.

The first commercial FORTRAN program was run at Westinghouse. The first compile run produced a missing comma diagnostic. The second attempt was a success.

The U.S. government created the Advanced Research Project Group (ARPA) in esponse to the Soviet Union’s launching of Sputnik. ARPA was intended to develop key technology that was too risky for private business to develop.

1958

FORTRAN II in 1958 introduced subroutines, functions, links to assembly language, loops, and a primitive For loop.

IAL (International Algebraic Logic) started as the project later renamed ALGOL 58. The theoretical definition of the language was published. No compiler.

LISP (LISt Processing) was created n 1958 and released in 1960 by John McCarthy of MIT. LISP is the second oldest programming language still in common use. LISP was intended for writing artificial intelligence programs.

> “Interest in artificial intelligence first surfaced in the mid 1950. Linguistics, psychology, and mathematics were only some areas of application for AI. Linguists were concerned with natural language processing, while psychologists were interested in modeling human information and retrieval. Mathematicians were more interested in automating the theorem proving process. The common need among all of these applications was a method to allow computers to process symbolic data in lists.
>
> “IBM was one of the first companies interested in AI in the 1950s. At the same time, the FORTRAN project was still going on. Because of the high cost associated with producing the first FORTRAN compiler, they decided to include the list processing functionality into FORTRAN. The FORTRAN List Processing Language (FLPL) was designed and implemented as an extention to FORTRAN.
>
> “In 1958 John McCarthy took a summer position at the IBM Information Research Department. He was hired to create a set of requirements for doing symbolic computation. The first attempt at this was differentiation of algebraic expressions. This initial experiment

produced a list of of language requirements, most notably was recursion and conditional expressions. At the time, not even FORTRAN (the only high-level language in existance) had these functions.

"It was at the 1956 Dartmouth Summer Research Project on Artificial Intelligence that John McCarthy first developed the basics behind Lisp. His motivation was to develop a list processing language for Artificial Intelligence. By 1965 the primary dialect of Lisp was created (version 1.5). By 1970 special-purpose computers known as Lisp Machines, were designed to run Lisp programs. 1980 was the year that object-oriented concepts were integrated into the language. By 1986, the X3J13 group formed to produce a draft for ANSI Common Lisp standard. Finally in 1992, X3J13 group published the American National Standard for Common Lisp." —Neal Ziring, The Language Guide, University of Michigan

"Some of the more significant features of the language are listed below:" —Neal Ziring, The Language Guide, University of Michigan

- **Atoms & Lists** - Lisp uses two different types of data structures, atoms and lists.
- **Atoms** are similar to identifiers, but can also be numeric constants
- **Lists** can be lists of atoms, lists, or any combination of the two
- **Functional Programming Style** - all computation is performed by applying functions to arguments. Variable declarations are rarely used.
- **Uniform Representation of Data and Code** - example: the list (A B C D)
    - a list of four elements (interpreted as data)
    - is the application of the function named A to the three parameters B, C, and D (interpreted as code)
- **Reliance on Recursion** - a strong reliance on recursion has allowed Lisp to be successful in many areas, including Artificial Intelligence.
- **Garbage Collection** - Lisp has built-in garbage collection, so programmers do not need to explicitly free dynamically allocated memory.

1959

COBOL (COmmon Business Oriented Language) was created in May 1959 by the Short Range Committee of the U.S. Department of Defense (DoD). The CODASYL committee (COnference on DAta SYstems Languages) worked from May 1959 to April 1960. Official ANSI standards included COBOL-68 (1968), COBOL-74 (1974), COBOL-85 (1985), and COBOL-2002 (2002). COBOL 97 (1997) introduced an object oriented version of COBOL. COBOL programs are divided into four divisions: identification, environment, data, and procedure. The divisions are further divided into sections. Introduced the RECORD data structure. Emphasized a verbose style intended to make it easy for business managers to read programs. Admiral Grace Hopper is recognized as the major contributor to the original COBOl language and as the inventor of compilers.

LISP 1.5 released in 1959.

" '**DYNAMO** is a computer program for translating mathematical models from an easy-to-understand notation into tabulated and plotted results. … A model written in DYNAMO consists of a number of algebraic relationships that relate the variables one to another.' Although similar to FORTRAN, it is easier to learn and understand. DYNAMO stands for DYNAmic MOdels. It was written by Dr. Phyllis Fox and Alexander L. Pugh, III, and was completed in 1959. It grew out of an earlier language called SIMPLE (for Simulation of Industrial Management Problems with Lots of Equations), written in 1958 by Richard K. Bennett." —Language Finger, Maureen and Mike Mansfield Library, University of Montana.

ERMA (Electronic Recording Method of Accounting), a magnetic ink and computer readable font, was created for the Bank of America.

1960

ALGOL (ALGOrithmic Language) was released in 1960. Major releases in 1960 (ALGOL 60) and 1968 (ALGOL 68). ALGOL is the first block-structured labguage and is considered to be the first second generation computer language. This was the first programming language that was designed to be machine independent. ALGOL introduced such concepts as: block structure of code (marked by BEGIN and END), scope of variables (local variables inside blocks), BNF (Backus Naur Form) notation for defining syntax, dynamic arrays, reserved words, IF THEN ELSE, FOR, WHILE loop, the := symbol for assignment, SWITCH with GOTOs, and user defined data types. ALGOL became the most popular programming language in Europe in the mid- and late-1960s.

C.A.R. Hoare invented the **Quicksort** in 1960.

1962

APL (A Programming Language) was published in the 1962 book *A Programming Language* by Kenneth E. Iverson and a subset was first released in 1964. The language APL was based on a notation that Iverson invented at Harvard University in 1957. APL was intended for mathematical work and used its own special character set. Particularly good at matrix manipulation. In 1957 it introduced the array. APL used a special character set and required special keyboards, displays, and printers (or printer heads).

FORTRAN IV was released in 1962.

Simula was created by Ole-Johan Dahl and Kristen Nygaard of the Norwegian Computing Center between 1962 and 1965. A compiler became available in 1964. Simula I and Simula 67 (1967) were the first object-oriented programming languages.

SNOBOL (StroNg Oriented symBOli Language) was created in 1962 by D.J. Farber, R.E. Griswold, and F.P. Polensky at Bell Telephone Laboratories. Intended for processing strings, the language was the first to use associative arrays, indexed by any type of key. Had features for pattern-matching, concatenation, and alternation. Allowed running code stored in strings. Data types: integer, real, array, table, pattern, and user defined types.

SpaceWarI, the first interactive computer game, was created by MIT students Slug Russel, Shag Graetz, and Alan Kotok on DEC's PDP-1.

1963

Work on PL/I started in 1963.

> "**Data-Text** was the "original and most general problem-oriented computer language for social scientists." It has the ability to handle very complicated data processing problems and extremely intricate statistical analyses. It arose when FORTRAN proved inadequate for such uses. Designed by Couch and others, it was first used in 1963/64, then extensively revised in 1971. The Data-Text System was originally programmed in FAP, later in FORTRAN, and finally its own language was developed." —Language Finger, Maureen and Mike Mansfield Library, University of Montana.

Sketchpad, an interactive real time computer drawing system, was created in 1963 by Ivan

Sutherland as his doctoral thesis at MIT. The system used a light pen to draw and manipulate geometric figures on a computer screen.

ASCII (American Standard Code for Information Interchange) was introduced in 1963.

1964

BASIC (Beginner's All-purpose Symbolic Instruction Code) was designed as a teaching language in 1963 by John George Kemeny and Thomas Eugene Kurtz of Dartmouth College. BASIC was intended to make it easy to learn programming. The first BASIC program was run at 4 a.m. May 1, 1964.

PL/I (Programming Language One) was created in 1964 at IBM's Hursley Laboratories in the United Kingdom. PL/I was intended to combine the scientific abilities of FORTRAN with the business capabilities of COBOL, plus additional facilities for systems programming. Also borrows from ALGOL 60. Originally called NPL, or New Programming Language. Introduces storage classes (automatic, static, controlled, and based), exception processing (On conditions), Select When Otherwise conditional structure, and several variations of the DO loop. Numerous data types, including control over precision.

RPG (Report Program Generator) was created in 1964 by IBM. Intended for creating commercial and business reports.

APL\360 implemented in 1964.

1965

SNOBOL 3 was released in 1965.

Attribute grammars were created in 1965 by Donald Knuth.

1966

ALGOL W was created in 1966 by Niklaus Wirth. ALGOL W included RECORDs, dynamic data structures, CASE, passing parameters by value, and precedence of operators.

Euler was created in 1966 by Niklaus Wirth.

FORTRAN 66 was released in 1966. The language was rarely used.

ISWIM (If You See What I Mean) was described in 1966 in Peter J. Landin's article *The Next 700 Programming Languages* in the Communications of the ACM. ISWIM, the first purely functional language, influenced functional programming languages. The first language to use lazy evaluation.

LISP 2 was released in 1966.

1967

Logo was created in 1967 (work started in 1966) by Seymour Papert. Intended as a programming language for children. Started as a drawing program. Based on moving a "turtle" on the computer screen.

Simula 67 was created by Ole-Johan Dahl and Kristen Nygaard of the Norwegian Computing Center in 1967. Introduced classes, methods, inheriteance, and objects that are instances of classes.

SNOBOL 4 (StroNg Oriented symBOli Language) was released in 1967.

CPL (Combined Programming Language) was created in 1967 at Cambridge and London Universities. Combined ALGOL 60 and functional language. Used polymorphic testing structures. Included the ANY type, lists, and arrays.

1968

ALGOL 68 in 1968 introduced the =+ token to combine assignment and add, UNION, and CASTing of types. It included the IF THEN ELIF FI structure, CASE structure, and user-defined operators.

Forth was created by Charles H. Moore in 1968. Stack based language.

ALTRAN, a variant of FORTRAN, was released.

ANSI version of COBOL defined.

Edsger Dijkstra wrote a letter to the Communications of the ACM claiming that the use of GOTO was harmful.

1969

BCPL (Basic CPL) was created in 1969 in England. Intended as a simplified version of CPL, includes the control structures For, Loop, If Then, While, Until Repeat, Repeat While, and Switch Case.

> "BCPL was an early computer language. It provided for comments between slashes. The name is condensed from "Basic CPL"; CPL was jointly designed by the universities of Cambridge and London. Officially, the "C" stood first for "Cambridge," then later for "Combined." -- Unofficially it was generally accepted as standing for Christopher Strachey, who was the main impetus behind the language." —Language Finger, Maureen and Mike Mansfield Library, University of Montana.

B (derived from BCPL) developed in 1969 by Ken Thompson of Bell Telephone Laboratories for use in systems programming for UNIX. This was the parent language of C.

SmallTalk was created in 1969 at Xerox PARC by a team led by Alan Kay, Adele Goldberg, Ted Kaehler, and Scott Wallace. Fully object oriented programming language that introduced a graphic environment with windows and a mouse.

RS-232-C standard for srial communication introduced in 1969.

UNIX created at AT&T Bell telephone Laboratories by Kenneth Thompson and Dennis Ritchie.

ARPA created ARPAnet, the forerunner of the Internet.

1970

Prolog (PROgramming LOGic) was created in 1972 in France by Alan Colmerauer with Philippe Roussel. Introduces Logic Programming.

Pascal (named for French religious fanatic and mathematician Blaise Pascal) was created in 1970 by Niklaus Wirth on a CDC 6000-series computer. Work started in 1968. Pascla intended as a teaching language to replace BASIC. Programs compiled to an intermediate P-code, that is platform independent.

Forth used to write the program to control the Kitt Peaks telescope.

BLISS was a systems programming language developed by W.A. Wulf, D.B. Russell, and A.N. Habermann at Carnegie Mellon University in 1970. BLISS was a very popular systems programming language until the rise of C. The original compiler was noted for its optimizing of code. Most of the utilities for DEC's VMS operating system were written in BLISS-32. BLISS was a typeless languagebased on expressions rather than statements. Expressions produced values, and possibly caused other actions, such as modification of storage, transfer of control, or looping. BLISS had powerful macro facilities, conditional execution of statements, subroutines, built-in string functions, arrays, and some automatic data conversions. BLISS lacked I/O instructions on the assumption that systems I/O would actually be built in the language.

1972

C was developed from 1969-1972 by Dennis Ritchie (with assistance by Brian W. Kernighan) of Bell Telephone Laboratories for use in systems programming for UNIX.

Pong, the first arcade video game, was introduced by Nolan Bushnell in 1972. His company was called Atari.

1973

ML (Meta Language) was created in 1973 by R. Milner of the University of Edinburgh. Functional language implemented in LISP.

Actoris a mathematical model for concurrent computation first published bby Hewitt in 1973.

"Actor is an object-oriented programming language. It was developed by the Whitewater Group in Evanston, Ill." —Language Finger, Maureen and Mike Mansfield Library, University of Montana.

ARPA created Transmission Control Protocol/Internet Protocol (TCP/IP) to network together computers for ARPAnet.

1974

SQL (Standard Query Language) was designed by Donald D. Chamberlin and Raymond F. Boyce of IBM in 1974.

AWK (first letters of the three inventors) was designed by Aho, Weinberger, and Kerninghan in 1974. Word processing language based on regular expressions.

Alphard (named for the brightest star in Hydra) was designed by William Wulf, Mary Shaw, and Ralph London of Carnegie-Mellon University in 1974. A Pascal-like language intended for data abstraction and verification. Make use of the "form", which combined a specification and an implementation, to give the programmer control over the impolementation of abstract data types.

"Alphard is a computer language designed to support the abstraction and verification techniques required by modern programming methodology. Alphard's constructs allow a programmer to isolate an abstraction, specifying its behavior publicly while localizing knowledge about its implementation. It originated from studies at both Carnegie-Mellon

University and the Information Sciences Institute." —Language Finger, Maureen and Mike Mansfield Library, University of Montana.

"**CLU** began to be developed in 1974; a second version was designed in 1977. It consists of a group of modules. One of the primary goals in its development was to provide clusters which permit user-defined types to be treated similarly to built-in types." —Language Finger, Maureen and Mike Mansfield Library, University of Montana.

1975

Scheme, based on LISP, was created by Guy Lewis Steele Jr. and Gerald Jay Sussman at MIT in 1975.

Tiny BASIC created by Dr. Wong in 1975 runs on Intel 8080 and Zilog Z80 computers.

RATFOR (RATional FORtran) created by Brian Kernigan in 1975. Used as a precompiler for FORTRAN. RATFOR allows C-like control structures in FORTRAN.

1976

Design System Language, a forerunner of PostScript, was created in 1976. The Forth-like language handles three dimensional databases.

SASL (Saint Andrews Static Language) was created by D. Turner in 1976. Intended for teaching functional programming. Based on ISWIM. Unlimited data structures.

CP/M, an operating system for microcomputers, was created by Gary Kildall in 1976.

1977

Icon, based on SNOBOL, was created in 1977 by faculty, staff, and students at the University of Arizona. Structured types include list, set, and table (dictionary).

OPS5 was created by Charles Forgy in 1977.

FP was presented by John Backus in his 1977 Turing Award lecture *Can Programming be Liberated From the von Neumann Style? A Functional Style and its Algebra of Programs*.

Modula (MODUlar LAnguage) was created by Niklaus Wirth, who started work in 1977. Modula-2 was released in 1979.

1978

CSP was created in 1978 by C.A.R. Hoare.

"C.A.R. Hoare wrote a paper in 1978 about parallel computing in which he included a fragment of a language. Later, this fragment came to be known as CSP. In it, process specifications lead to process creation and coordination. The name stands for Communicating Sequential Processes. Later, the separate computer language Occam was based on CSP." —Language Finger, Maureen and Mike Mansfield Library, University of Montana.

1979

Modula-2 was released in 1979. Created by Niklaus Wirth, who started work in 1977.

VisiCalc (VISIble CALculator) was created for the Apple II personal computer in 1979 by Harvard MBA candidate Daniel Bricklin and programmer Robert Frankston.

1980

dBASE II was created in 1980 by Wayne Ratliff at the Jet Propulsion Laboratories in Pasadena, California. The original version of the language was called **Vulcan**. Note that the first version of dBASE was called dBASE II.

1981

Relational Language was created in 1981 by Clark and Gregory.

1983

Ada was first released in 1983 (ADA 83), with major releases in 1995 (ADA 95) and 2005 (ADA 2005). Ada was created by the U.S. Department of Defense (DoD), originally intended for embedded systems and later intended for all military computing purposes. Ada is named for Augusta Ada King, the Countess of Lovelace, the first computer programmer in modern times.

Concurrent Prolog was created in 1983 by Shapiro.

Parlog was created in 1983 by Clark and Gregory.

C++ was developed in 1983 by Bjarne Stroustrup at Bell Telephone Laboratories to extend C for object oriented programming.

PostScript was created in 1982 by a team of researchers at Xerox PARC.

The University of California at Berkeley released a version of UNIX that included TCP/IP.

1984

Objective C, an extension of C inspired by SmallTalk, was created in 1984 by Brad Cox. Used to write NextStep, the operating system of the NeXt computer.

Standard ML, based on ML, was created in 1984 by R. Milner of the University of Edinburgh.

1985

PageMaker was created for the Apple Macintosh in 1985 by Aldus.

1986

Eiffel (named for Gustave Eiffel, designer of the Eiffel Tower) was released in 1986 by Bertrand Meyer. Work started on September 14, 1985.

> "Eiffel is a computer language in the public domain. Its evolution is controlled by Nonprofit International Consortium for Eiffel (NICE), but it is open to any interested party. It is intended to treat software construction as a serious engineering enterprise, and therefore is named for the French architect, Gustave Eiffel. It aims to help specify, design, implement, and change quality software." —Language Finger, Maureen and Mike Mansfield Library, University of Montana.

GAP (Groups, Algorithms, and Programming) was developed in 1986 by Johannes Meier, Werner Nickel, Alice Niemeter, Martin Schönert, and others. Intended to program mathematical algorithms.

1987

CAML (Categorical Abstract Machine Language) was created by Suarez, Weiss, and Maury in 1987.

Perl (Practical Extracting and Report Language) was created by Larry Wall in 1987. Intended to replace the Unix shell, Sed, and Awk. Used in CGI scripts.

HyperCard was created by William Atkinson in 1987. **HyperTalk** was the scripting language built into HyperCard.

Thomas and John Knoll created the program Display, which eventually became PhotoShop. The program ran on the Apple Macintosh.

Adobe released the first version of Illustrator, running on the Apple Macintosh.

1988

CLOS, an object oriented version of LISP, was developed in 1988.

Mathematica was developed in 1986.

Oberon was created in 1986 by Niklaus Wirth.

1989

HTML was developed in 1989.

Miranda (named for a character by Shakespeare) was created in 1989 by D. Turner. Based on SASL and ML. Lazy evaluation and embedded pattern matching.

1990

Haskell was developed in 1990.

Tim Berners-Lee of the European CERN laboratory created the World Wide Web on a NeXT computer.

In February of 1990, Adobe released the first version of the program PhotoShop (for the Apple Macintosh).

1991

Python (named for Monty Python Flying Circus) was created in 1991 by Guido van Rossum. A scripting language with dynamic types intended as a replacement for Perl.

Pov-Ray (Persistence of Vision) was created in 1991 by D.B.A. Collins and others. A language for describing 3D images.

Linux operating system was released on September 17, 1991, by Finnish student Linus Torvalds.

1992

Dylan was created in 1992 by Apple Computer and others. Dylan was originally intended for use with the Apple Newton, but wasn't finished in time.

1995

Java (named for coffee) was created by James Gosling and others at Sun Microsystems for embedded systems and released for applets in 1995. Original work started in 1991 as an interactive language under the name Oak. Rewritten for the internet in 1994.

JavaScript (originally called LiveScript) was created by Brendan Elch at Netscape in 1995. A scripting language for web pages.

PHP (PHP Hypertext Processor) was created by Rasmus Lerdorf in 1995.

Ruby was created in 1995 by Yukihiro Matsumoto. Alternative to Perl and Python.

1996

UML (Unified Modeling Language) was created by Grady Booch, Jim Rumbaugh, and Ivar Jacobson in 1996 by combining the three modeling languages of each of the authors.

1997

REBOL (Relative Expression Based Object Language) was created by Carl SassenRath in 1997. Extensible scripting language for internet and distributed computing. Has 45 types that use the same operators.

ECMAScript (named for the European standards group E.C.M.A.) was created in 1997.

2000

C# was created by Anders Hajlsberg of Microsoft in 2000. The main language of Microsoft's .NET.

2001

AspectJ (Aspect for Java) was created at the Palo Alto Research Center in 2001.

Scriptol (Scriptwriter Oriented Language) was created by Dennis G. Sureau in 2001. New control structuress include for in, while let, and scan by. Variables and literals are objects. Supports XML as data structure.

2004

Scala was created February 2004 by Ecole Polytechnique Federale de Lausanne. Object oriented language that implements Python features in a Java syntax.

Hello World

The cliche first program is Hello World — a simple program that types the message “Hello World”. Even experienced programmers will often write a simple Hello World program when learning a new programming language or switching to a new development environment.

Kernighan and Ritchie popularized the Hello World! program in their book *C Programming Language*.

Every programming language has its own peculair rules for the form of a program. The Hello World program is a fine illustration of the basic rules of a programming language.

All of the code below produces the same basic results: typing the phrase “Hello World”. You may want to take a brief look at how different languages can have very different methods for achieving the same results.

Ada

```
with Ada.Text_IO;
use Ada.Text_IO;

procedure HelloWorld is
begin
   Ada.Text_IO.Put_Line("Hello World");
end HelloWorld;
```

Ada was first released in 1983 (ADA 83), with major releases in 1995 (ADA 95) and 2005 (ADA 2005). Ada was created by the U.S. Department of Defense (DoD), originally intended for embedded systems and later intended for all military computing purposes.

Ada is named for Augusta Ada King, the Countess of Lovelace, the first computer programmer in modern times.

ALGOL

```
BEGIN
   OUTSTRING(2, "Hello World");
END.
```

ALGOL (ALGOrithmic Language) was first formalized in 1958 (ALGOL 58), with major releases in 1960 (ALGOL 60) and 1968 (ALGOL 68). ALGOL was originally intended for scientific computations.

ALGOL is considered to be the first second generation computer language.

ALGOL was a highly influential programming language. Most modern programming languages are descendants of ALGOL.

ALGOL introduced such concepts as: block structure of code, scope of variables, BNF notation for defining syntax, dynamic arrays, reserved words, and user defined data types.

BASIC

```
10 PRINT "Hello World"
```

BASIC (Beginner’s All-purpose Symbolic Instruction Code) was designed as a teaching language in 1963 by John George Kemeny and Thomas Eugene Kurtz of Dartmouth College.

C

```
#include <stdio.h>

main()
{
   printf("Hello World\n");
}
```

C was developed in 1972 by Dennis Ritchie of Bell Telephone Laboratories for use in systems programming for UNIX.

C++

```
#include <iostream.h>

int main(int argc, char *argv[])
{
   cout << "Hello World" << endl;
   return 0;
}
```

C++ was developed in 1983 by Bjarne Stroustrup at Bell Telephone Laboratories to extend C for object oriented programming.

COBOL

```
IDENTIFICATION DIVISION.
PROGRAN-ID. HelloWorld.
AUTHOR. Milo.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
INPUT-OUTPUT SECTION.

DATA DIVISION.
FILE SECTION.
WORKING-STORAGE SECTION.
LINKAGE SECTION.

PROCEDURE DIVISION.
DISPLAY "Hello World".
STOP RUN.
```

COBOL (COmmon Business Oriented Language) was created in 1959 by the Short Range Committee of the U.S. Department of Defense (DoD). Official ANSI standards included COBOL-68 (1968), COBOL-74 (1974), COBOL-85 (1985), and COBOL-2002 (2002). COBOL 97 (1997) introduced an object oriented version of COBOL.

Dylan

```
define method hello-world()
   format-out("Hello World\n");
end method hello-world;

hello-world();
```

Dylan was created in the early 1990s by Apple Computer and others. Dylan was originally intended for use with the Apple Newton, but wasn't finished in time.

Forth

```
: hello_world ." Hello World" ;
```

Forth was created by Charles H. Moore in 1968. Forth was a reference to Moore's claim that he had created a courth generation programming language.

FORTRAN

```
      PROGRAM HELLO
      WRITE(UNIT=*, FMT=*) 'Hello World'
      END
```

FORTRAN (FORmula TRANslation) was created in 1954 at International Business Machines (now IBM). FORTRAN is the oldest programming language still in common use.

HTML

```
<html>
<head>
<title>Hello World</title>
</head>
<body>
<p>Hello World</p>
</body>
</html>
```

HTML (HyperText Markup Language) was developed in 1989.

Java

```
import java.applet.*;
import java.awt.*;
Public class HelloWorld extends Applet
```

```
{
public void paint(Graphics g)
   {
      g.drawstring("Hello World".,10,10);
   }
}
```

Java (a reference to coffee) was created by Sun Microsystems and released in 1995.

LISP

```
(print "Hello World".)
```

LISP> (LISt Processing) was created n 1958 by John McCarthy of MIT. LISP is the second oldest programming language still in common use.

Logo

```
print [Hello World]
```

Logo was created in 1967 by Daniel G. Bobrow, Wally Feurzeig, and Seymour Papert.

Modula-2

```
MODULE HelloWorld;
FROM InOut IMPORT WriteString, WriteLn;
BEGIN
   WriteString('Hello World');
   WriteLn;
END HelloWorld.
```

Modula (MODUlar LAnguage) was created by Niklaus Wirth, who started work in 1977. Modula-2 was released in 1980.

Oberon

```
MODULE HelloWorld;
IMPORT Out;
BEGIN
   Out.Open;
   Out.String('Hello World');
END HelloWorld.
```

Oberon (named for a moon of Uranus) was created in 1986 by Niklaus Wirth.

Pascal

```
PROGRAM HelloWorld (OUTPUT);
BEGIN
   WRITELN('Hello World');
```

```
END.
```

Pascal (named for French religious fanatic and mathematician Blaise Pascal) was created in 1970 by Niklaus Wirth.

Perl

```
print "Hello World\n";
```

Perl (Practical Extracting and Report Language) was created by Larry Wall in 1987.

PHP

```
<?php
   echo "Hello World\n";
?>
```

PHP (PHP Hypertext Processor) was created by Rasmus Lerdorf in 1995.

PL/I

```
HELLO: PROCEDURE OPTIONS (MAIN);
   PUT SKIP LIST('HELLO WORLD');
END HELLO;
```

PL/I (Programming Language One) was created in 1964 at IBM’s Hursley Laboratories in the United Kingdom.

PostScript

```
/Courier findfont
14 scalefont
setfont
0 0 moveto
(Hello World) show
showpage
```

PostScript was created in 1982 by a team of researchers at Xerox PARC.

Prolog

```
?- write('Hello World'), nl.
```

Prolog (PROgramming LOGic) was created in 1972 by Alan Colmerauer with Philippe Roussel.

Python

```
print: "Hello World"
```

Python (named for Monty Python Flying Circus) was created in 1991 by Guido van Rossum.

Ruby

```
"Hello World\n".display
```

Ruby was created in 1995 by Yukihiro Matsumoto.

Shell Script (BASH)

```
echo Hello World
```

SmallTalk

```
'Hello World' out.
```

SmallTalk was created in 1969 at Xerox PARC by a team led by Alan Kay, Adele Goldberg, Ted Kaehler, and Scott Wallace.

SNOBOL

```
   OUTPUT = 'Hello World'
END
```

SNOBOL (StroNg Oriented symBOli Language) was created in 1962.

SQL

```
SELECT 'Hello World'
```

SQL (Standard Query Language) was designed by Donald D. Chamberlin and Raymond F. Boyce of IBM in 1974.

creating a program

Using your local development environment, you will want to try to type in the example for the language you are learning, then attempt to compile it, and finally attempt to run it. These are critical skills to learn before you can start learning how to program.

C

The typical steps for compiling a program in C on a UNIX machine are:

step	**command**	**input**	**output**
create source code	ed emacs use any text editor	type from keyboard or terminal	source code
check (for lexical errors)	lint	source code file	listing with warnings
preprocess	cc (or cpp)	source code file	c code file
compile (convert to assembly for specific hardware platform)	cc2	c code file	assembly source code file
assemble (for specific hardware platform)	asm (or as) (or masm)	assembly language file	a.out object code file
link	link	object code file	executable code
run	program name	file with executable code	results of program

listings and errors

Compilers can produce **listings**. These will add extra information to your original source code file. Many compilers have optional settings on what kinds of information are included.

Of particular interest are **errors**. There are two basic kinds of errors in programming: (1) errors in typing a language and (2) errors in programming logic.

A compiler generated listing will only show errors in the typing (such as mistakes in spelling or punctuation). These are known as **compile time errors** because they can be identified at the time the program is compiled.

You won't know about errors in programming logic until the program is actually running. These are known as **run time errors**. You will want to thoroughly test your programs to try to find any mistakes, but testing won't always reveal every error.

Errors vary in **severity**. In some cases the compiler may be able to correct errors of low severity for you. Some compilers let you control which errors are **flagged**, skipping over errors the compiler can correct. You should strive to eliminate all errors from your source code listing, even if the compiler is able to correct them for you.

Compilers also often produce **warning**. These are things that are not necessarily a problem. Some compilers let you decide whether to have warnings listed or not, and in some cases even let you decide which kinds of warnings are listed. You should strive to eliminate all warnings from your source code listing.

Many compilers have the option of inserting **line numbers** into your listing. These line numbers can be very useful in tracking down compile time errors. Note that it is rare that the compiler's idea of line numbers will end up exactly matching the physical lines in your original source file. The line numbers are typically based on the language's statements. Comments and blank lines are rarely counted as lines in a source code listing. Multiple statements on a single line are usually identified by the number of the first statement on the same physical line.

Many compilers will give you the option of stopping after the first error is detected or attemting to continue to compile. Note that the first error may confuse the compiler enough that it starts reporting bogus errors. This is known as **cascading errors** (named after cascading waterfalls).

Many compilers will give you the option of viewing the **symbol table**. This is a list of the identifiers (procedures, functions, labels, constants, variables, etc.). There are many uses for a symbol table in correcting your source code. As one example, it is possible that you misspelled an identifier. The code may be correct from a lexical point of view (with no errors listed), but won't work at run time. You might spot the incorrect spelling in your symbol table.

You may want to purposely introduce some errors into your Hello World program and see how your compiler reports them.

Compilers are notorious for cryptic error messages. The error messages are generally not standardized for any particualr language, but vary with each compiler. With practice, you will learn to read the error messages and use them to find and identify your mistakes.

A few days after initially writing this chapter, I spoke to a man who had been a BASIC programmer during the 1980s. He recounted that he had attempted to write a FORTRAN program and spent three weeks trying to track down a single error: accidently replacing a period with a comma. Sadly, this is typical of the debugging abilities of most programmers. There are very few places in FORTRAN where replacing a period with a comma will still produce a running program. It really shouldn’t have taken more than a few minutes to track down the exact line with the error and then identify the incorrect character through visual examination. If the program did compile and run, it shouldn’t have taken too much time to pin down the location of the error and then spot the incorrect character. Being methodical is a great way to not only locate bugs but also to avoid them in the first place.

free form vs. columns

Many early programming languages relied heavily on **punched cards** for entry, including requiring that specific elements of the program appear in specific **columns**. The Hollerith card had 80 columns.

Note that the gray columns in the following picture are much wider than depicted.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18-69	70	71	72	73	74	75	76	77	78	79	80
COBOL																												
sequence number						cmnt	A				B										identification							
FORTRAN																												
label					cont	FORTRAN statements															ignored							
PL/I																												
OS	PL/I statements																				sequence number							

Almost all modern programming languages are **free form**, meaning that the programmer has relative freedom to format a program in an easy to read and understand manner.

A **continuation character** is used to indicate that some element extends over more than one line. For example, in FORTRAN a programmer places a character other than space or blank in the sixth column ot indicate that the card is a continuation from the previous card.

Even though modern languages are generally free form, you will occassionally find vestigages of the older column/card view crop up in places.

Indentation should be used to visually make the source code more comprehensible and easier to read. Consider the following examples (from C and Pascal):

C

good

```
main()
{
   int i,j,k;
   k = 0;
   for(i = 1; i <= 10; i++)
   {
      for(j = 1; j <= 15; j++)
      {
         k++;
      }
   }
}
```

bad

```
main()
{
int i,j,k;
for(i = 1; i <= 10; i++) { for(j = 1; j <= 15; j++)
{ k++; } } }
```

Pascal

good

```
procedure SimpleLoop;
var
   i: integer;
   j: integer;
begin
   j := 0;
   for i := 1 to 10
      do begin
         j := j + i;
      end; {for}
end; {SimpleLoop}
```

bad

```
procedure SimpleLoop;
var i,j: integer; begin
j := 0; for i := 1 to 10 do begin j := j + i;
end; end;
```

Not only does good indentation make code easier to read, it also makes it easier to spot many kinds of simple typing errors, such as imbalance of matching pairs.

C

C is completely free form.

Pascal

Pascal is completely free form.

PHP

PHP is completely free form.

whitespace

Whitespace is used for clarity and ease of understanding. The term is borrowed from art and design.

While whitespace characters vary by language, they generally include any characters that don't print on the page, such as the space or blank character, tab, new line, carriage return, and form feed.

An important use for whitespace is to visually show the structure of a program. A programmer can use **indentation** to show logical and lexical blocks of code. Indentation and use of white space is normally optional. See example in preceeding chapter on free form vs columns.

Some languages use indentation as a required language feature. Blocks of code are marked by indentation. This approach is called the **off side rule** (a term borrowed from soccer). Some programming languages using the off side rule include: ISWIM (the first to introduce the idea), ABC, Curry, Haskell, Lispin, Miranda, Nemerle, Occam, Pliant, Python, and YAML.

C

Whitespace characters may be used almost anywhere in a c program. There are a few locations where whitespace is forbidden (such as between a function name and its argument list). There are a few places where whitespace is required (such as preventing ambiguity).

whitespace characters in c

name	ASCII (hex)	source code representation
blank or space	20	\040 or typed space
backspace	08	\b
horizontal tab	09	\t
vertical tab	0B	\v
form feed	0C	\f
newline or line feed	0A	\n
carriage return	0D	\r

Comments in c are supposed to be considered whitespace, but some compilers simply remove comments during parsing.

PHP

Whitespace characters in PHP include new line (\n), carriage returns (\r), spaces (or blanks), horizontal tabs (\t), vertical tabs (\v), and end of string characters (\0).

Whitespace is ignored in PHP.

comments

Comments are extra information intended for human readers of a program. A compiler or interpretter ignores comments (or converts them into whitespace).

A computer doesn't need comments. You could write a program without any comments and your compiler won't care. Any human trying to read your program will care.

Students often have trouble understanding why there is such an emphasis on comments. It seems like busy work.

The truth is that any program small enough to be used as a teaching assignment is probably so trivial that any competent programmer could look at the raw uncommented source code and figure out exactly how the program works.

Students rarely ever have to go back and modify a class assignment after it has been turned in for a grade.

In real life, successful programs last for years or decades. There will be a need for many different programmers will have to make modifications to a program long after the initial coding. In addition to any bug fixes, there will be new features added and changes in response to external conditions.

In the life cycle of non-trivial programs, the vast majority of the time and effort is in the maintenance of the program, not the initial design and coding.

Sooner or later the programmer making the changes will be someone different than the original programmer. Comments are the basic tool that the next programmer will use in figuring out how to make these inevitable changes.

In the earliest days of computers there wa a tendency to believe that programs would be written once and used for a brief period and then abandoned for new work.

Programmers started to realize that they were redoing work that they or someone else had already done. Reinventing the wheel. This led to the creation of **libraries** of code that all the programmers at a particular installation could share.

In order for the libraries to be useful, the programmer who wrote each library routine would have to leave behind some kind of documentation for other programmers to know what the library routine did and how to use it in their own programs.

External documentation included such things as a manual (or manual entry), flow charts, pseudocode, and numerous other paper documents.

Internal documentation included comments and naming things so that the name was meaningful.

In the first decades of computer programming, there was an emphasis on external documentation. Unfortunately, external documentation turned out to be unreliable.

In some cases the documentation was created before the program was written (such as a flow chart or pseudocode). When the actual software inevitably changed (as errors were located or new requirements were added by the bosses), the original flowcharts and other documentation wouldn't get changed. Sometimes the differences were minor annoyances. Sometimes the differences were major problems to the next programmer to work on the project.

If the external documentation was saved for after the actual coding, it would often be a sloppy afterthought. The programmer, anxious to get on to the next job, would rush through the creation of the external documentation. Important information would get left out. Sometimes incorrect information would be written down. Sometimes things would be simplified to the point of being useless. Often the programmer would simply not write any external documentation of any kind. And inevitably external documentation would have a tendency of getting lost or misplaced or even discarded.

Because of the inherent unreliability of external documentation, current software engineering emphasizes well written internal documentation. Internal documentation cna easily be updated as the program is being written. And comments are the primary method for internal documentation.

creating comments

There are four basic kinds of comments (not all languages support all three methods).

Comments can exist on separate lines (a single line of comment only). For some early langagues, this is the only option available.

Comments can exist over a series of multiple lines.

Comments can exist at the end of a line.

Comments can appear in the midst of a line.

A possible source of confusion is different methods for indicating a comment. For example, in Pascal, a comment can be indicated by placing it between braces or curly brackets { Pascal comment }. But in C, the curly brackets or braces are used to indicate a block of code. These kinds of differences are one reason that it is best that a beginning student only learn one programming language at a time.

Important: As a beginning student, only read the one language section that discusses the language you are learning. Attempting to learn multiple languages at once is a recipe for disaster.

C

Comments begin with the character pair `/*` and end with the character pair `*/`. Comments can extend over multiple lines and can occur anywhere a space or newline character can be used. Comments can not be placed in the middle of any identifier (such as the name of a variable, procedure, or constant).

```
/* This is an example of a single line comment in c */

/* This is an example of a
multiple line comment in c */

temporary= 't'; /* example of a comment at the end of a line in c */

temporary= 't'; /* example of a comment in a line of code in c */ other= 'r';
```

ANSI C requires that comments be replaced with a single space character. Some compilers instead ignore comments (delete them without replacing them with a space character). This can in certain situations produce unexpected errors.

A few c compilers allow nestable comments (comments within a comment). Most c compilers end nested comments at the first end pair */, which can result in unexpected errors.

```
        /* This is an example of a /* nested comment */ in c */
```

In most compilers, the comment above would end at the */ after the words “`nested comment`”, and the “`in c*/`” would produce an error.

To comment out a series of lines of code without having to worry about unnesting comments (and possibly adding them back in later if the code is added back in), use the preprocessor commands:

```
        #if 0
        lines of code, /*possibly including comments*/
        #endif
```

Note that if a comment starts before the location where you added the preprocessor commands or ends after the location where you ended the preprocessor commands, that you will still have a problem. Make sure to completely enclose any comments within the preprocessor commands `#if 0` and `#endif`.

Some c compilers require a blank or space character after the initial /* and require a blank or space character before the closing */. For greater readability, it is best to follow this rule even if your compiler doesn’t require it.

Pascal

Comments begin with the opening braces (or left curly bracket) `{` and end with the closing braces (or right curly bracket) `}`. Comments can extend over multiple lines and can occur anywhere a space or newline character can be used. Comments can not be placed in the middle of any keyword or identifier (such as the name of a variable, program, procedure, or constant).

```
        { This is an example of a single line comment in Pascal }

        { This is an example of a
        multiple line comment in Pascal }

        temporary:= 't'; { example of a comment at the end of a line in Pascal }

        temporary:= 't'; { example of a comment in a line of code } other:= 'r';
```

Pascal allows the option of using the character pair `(*` to start a comment and the character pair `*)` to end a comment. This was put into the language to support older computers that didn’t have the braces.

The ISO 7185 and ISO 10206 standards, as well as ASNI Pascal, allow mixing the two comment styles, such as `(* this is a mixed comment }` and `{ this is also a mixed comment *)`. Many Pascal compilers are confused by **mixed** comments, so you should avoid this in your source code.

```
        { This is an example of a mixed comment in Pascal *)
```

Many Pascal compilers allow nestable comments (comments within a comment). Some Pascal compilers end nested comments at the first } or *), which can result in unexpected errors.

```
        { This is an example of a { nested comment } in Pascal }
```

In many compilers, the comment above would end at the } after the words “`nested comment`”, and the “`in Pascal }`” would produce an error.

Comments are not terminated with a semicolon.

Some Pascal compilers allow using the C++ form of the character pair `//` to start a comment that extends to the end of that particular line.

```
// This is an example of a non-standard comment in Pascal
```

This non-standard approach should generally be avoided, but might be used as a quick method to comment out sections of source code without worrying about nested comments.

PHP

PHP supports the comment syles of C, C++, and UNIX shell scripting.

Comments begin with the character pair `/*` and end with the character pair `*/`. Comments can extend over multiple lines and can occur anywhere a space or newline character can be used. Comments can not be placed in the middle of any identifier (such as the name of a variable, function, or constant).

```
/* This is an example of a single line comment in PHP */

/* This is an example of a
multiple line comment in PHP */

$temporary= 't'; /* example of a comment at the end of a line in PHP */

$temporary= 't'; /* example of a comment in a line of code in PHP */ $other= 'r';
```

PHP ignores text inside comments. PHP treats comments as whitespace.

PHP does not allow nestable comments (comments within a comment).

```
/* This is an example of a /* nested comment */ in PHP */
```

In PHP, the comment above would end at the */ after the words “`nested comment`”, and the “`in c*/`” would produce an error.

PHP allows use of the C++ form of the character pair `//` to start a comment that extends to the end of that particular line or the ending PHP tag, whichever comes first.

```
// This is an example of a C++ style comment in PHP
```

PHP also allowsthe use of the UNIX shell script form of the pound character `#` to start a comment that extends to the end of that particular line or the ending PHP tag, whichever comes first.

```
# This is an example of a UNIX shell script style comment in PHP
```

To comment out a series of lines of code without having to worry about unnesting comments (and possibly adding them back in later if the code is added back in), use the C++ (//) or UNIX shell script (#) style at the beginning of each commented line.

For greater readability, it is best to place a space character after the initial ?* and before the trailing */.

The PHP 5.0.1 function string **php_strip_whitespace** (string $filename) will return the PHP source code in the file *filename* with all PHP comments and whitespace removed. This fucntion does not remove whitespace or comments from any HTML in the file.

in-line comments

Use comments throughout your program to let yourself or others know what is going on. Even if the program is solely for your own use, you may have trouble remembering important details a year or more after you write it.

Some programmers add a comment to the end of every line, often repeating the same information as the source code with slightly different language. This is a complete waste of time.

```
; BAD COMMENT!
MOV D0, D5      ; move the contents of the D5 register into the D0 register
RET
```

Add comments when they add information to the source code.

```
; GOOD COMMENT!
MOV D0, D5      ; result of function stored in D0 in preparation for return
RET
```

Use comments liberally throughout your source code to explain everything that you might forget or that another programmer will need to know to understand your code.

header comments

Procedures, functions, and other important blocks of code, as well as the beginning of programs, should have header comments that fully explain the purpose of the code, any conventions for using the code, and important information about the code that another programmer will need to understand.

Many working installations will have particular required formats for header comments. Most professors will have detailed requirements for header comments. Follow those required formats exactly as specified.

If your installation does not have a required format for header comments, go ahead and use a format that you are familar with (possibly one you learned in school).

I can not over emphasize the importance of good header comments.

The following example is from an assembly language routine, but it illustrates the basic idea.

```
;***********************************************************
;* FUNCTION NAME: Hexadecimal_Character_to_Binary
;* PURPOSE: Converts an ASCII character into its binary 4-bit equivalent
;* INPUTS: D0 register has an ASCII character in the ranges 0..9 or A..F or a..f
;*      space character translated into zero
;* OUTPUT D0.B register has a hexadecimal integer
;*      of nibble (4-bit) length
;*      entire register zero filled
;*      if invalid input, D0.L set to -1
;*      N flag cleared on successful translation
;*      N flag set on failure (invalid hex)
;* METHODS: Uses a table look-up for fast translation
;* REGISTERS: All registers other than D0 are preserved.
;* CALLS: none
;* CALLED BY: UTILITY routine widely used
;***********************************************************
```

There are three common methods for indicating a block of comments (examples shown only in C to save space, the principles are the same for any other language).

This first method is the most commonly used.

```
/************************************
* line of comment
* another line of comment
* yet another line of comment
************************************/
```

This second method is less clear.

```
/************************************
line of comment
another line of comment
yet another line of comment
************************************/
```

This third method makes each line a separate comment. This method works best if the right column lines up vertically.

```
/************************************/
/* line of comment                  */
/* another line of comment          */
/* yet another line of comment      */
/************************************/
```

There are two other important methods for languages (such as Java) that use the C++ approach of offering both /* */ and // comment styles.

This method is known as the Sun commenting style and is useful with the automatic document generator that comes with the Java Development Kit.

```
/**
 * line of comment
 * another line of comment
 * yet another line of comment
 */
```

This method is known as the Microsoft commenting style and is associated with the Visual C++ programming environment.

```
//////////////////////////////////////
// line of comment
// another line of comment
// yet another line of comment
```

building blocks of a program

The terminology and details for constructing a program vary by language, but tend to follow a few common principles. The following presentation is very informal. All of these items will be discussed in more detail later.

Programs are created from a series of characters (letters, digits, punctuation, and other characters) from the **source character set**.

Characters are grouped into **tokens**. Each token, which can be one or more characters, has a distinct meaning in the language. Comments and white space are generally not considered to be tokens. Depending on how a language defines tokens, they can include operators, literals, keywords (or reserved words), and identifiers.

C

C has five classes of tokens: operators, separators, identifiers, reserved words, and constants.

A token is distinguished by the fact that if you try to break it up into smaller elements that it will lose or change its meaning. To use a simple example, the characters 3.5 collectively have the clear meaning of three and a half, but individually mean three, period, and five. A token is **indivisible**.

Literals are symbols used to indicate a specific value. Anytime you encounter a raw number, such as 2 or 573.9901, you are seeing an example of a numeric literal. There are other kinds of literals beyond just numbers.

Operators are symbols (usually one or two characters) that indicate an operation (such as the plus sign [+] for addition).

Separators are symbols that are used to separate portions of your source code. Some languages don’t use this concept. Some languages consider whitespace to be a separator, while other languages don’t (even if whitespace is used to separate tokens).

Key words or **reserved words** are the commands (and related terminology) of a programming language (such as the ubiquitious IF command). Reserved words can not be used for any other purpose. Some languages (such as PL/I) allow key words to be reassigned to other purposes (which creates massive confusion and is a horribly bad programming practice). In most languages, there is no distinction between the terms keywords and reserved words.

Identifiers are names. This includes the names of programs, modules, procedures, functions, variables, constants, etc. In some languages this may include reserved words or key words. Note that in the c programming language, constants are not considered an identifier.

Tokens can be grouped together into **statements**. Some languages have subgroups of statements called **expessions**.

In some languages (such as C) there is a special character (or characters) that **terminates** (or marks the end of) a statment. In some languages (such as Pascal) there is a special character (or characters) that **seperates** statements. This is a subtle but important difference. SOme languages (such as FORTRAN) are line oriented, with one statement to a line (with a continuation character to allow for multiple line statements).

Statements may (in most modern languages) be grouped into **blocks** of code.

Most modern programming languages have a **header** and a **body**, although few call them by those exact names.

Generally **declarations** go into the header and statements and blocks of code go into the body.

SECTION 2

This section examines advanced topics.

At this point in the book you should be able to determine whether I have the programming knowledge and the writing skill to realistically create a free downloadable college text book on computer programming. Possibly a large corporation or someone who is rich might want to donate a few hundred dollars to support me at the poverty level so that I can complete this book quickly. Possibly a business owner within reasonable walking distance of south east Costa Mesa, California, might be willing to provide me with a paying job. I will do almost any work that is legal, ethical, and safe (even things like washing dishes and mopping floors). I will work for less than minimum wage. I will work hard. I am a legal natural born citizen of the United States.

Boolean algebra and logic

Boolean algebra is named for George Boole, who introduced the ideas in the 1854 work “An Investigation of the Law of Thought”. Claude Shannon showed the application of Boolean algebra to switching circuits in the 1938 work “Symbolic Analysis of Relay and Switching Circuits”.

Major applications of Boolean algebra include:

1. truth calculus
2. switching algebra
3. set theory (algebra of classes)

Boolean algebra is a pure mathematical system that deals with perfect abstracts.

Real world logic circuits are physically imperfect implementations of Boolean algebra. Electrical problems (such as noise, interference, and heat) can cause failure. Delays in signals reaching certain locations (such as slew rate and propogation delay) can slow down cmputers and logic circuits and can produce nightmarish problems for computer and circuit designers. As processors and logic circuits shrink in size, quantum effects can even start to interfere with correct operation.

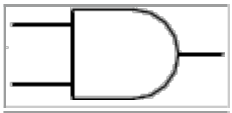
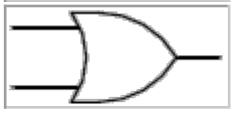
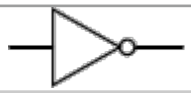
simple summary

Boolean algebra is binary.

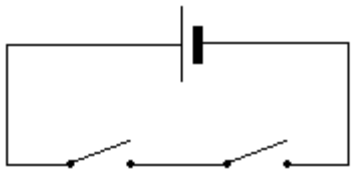
Objects can be one of two values: 1 or 0; true or false; high or low; positive or negative; closed or open; or any other pair of binary values.

The basic Boolean operations are AND, OR, and NOT.

The following chart shows the major interpretations of Boolean algebra:

Boolean Algebra	Truth Calculus		Switching Algebra	Logic Circuit	Set Theory Algebra of Classes	
$\cdot$ multiplication	$\wedge$	AND	series circuit		$\cap$	intersection
+ addition	$\vee$	OR	parallel circuit		$\cup$	union
0	F	FALSE	open circuit		$S(Z)$	null set
1	T	TRUE	short circuit		$S(U)$	universal set
$\overline{A}$	$\neg A$	NOT A	normally closed switch		$C(S)$	complemented set

AND requires both objects to be true for the result to be true. The AND works like a pair of switches in series. Both switches must be closed for current to flow.



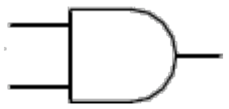
AND is conisdered to be Boolean multiplication and is represented by the middle dot symbol: · (such as A·B). As in ordinary algebra, AND (Boolean multiplication) can be written by dropping the middle dot (such as AB). There is no Boolean division operation.

The truth table for AND is as follows:

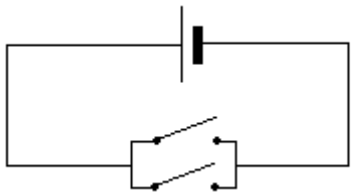
AND

A	B	result
0	0	0
1	0	0
0	1	0
1	1	1

The AND gate in logic circuits looks like:



OR (or inclusive or) requires either object to be true for the result to be true. The OR works like a pair of switches in parallel. Current will flow if either or both switches are closed.



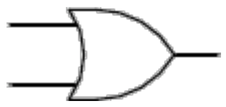
OR is conisdered to be Boolean addition and is represented by the plus symbol: **+** (such as (A+B). There is no Boolean subtraction operation.

The truth table for OR is as follows:

OR

A	B	result
0	0	0
1	0	1
0	1	1
1	1	1

The OR gate in logic circuits looks like:



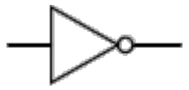
NOT (also called negation or complement) simply reverses the value of an object, changing true into false and changing false into true.

The truth table for NOT is as follows:

NOT

A	result
0	1
1	0

The NOT gate (or inverter) in logic circuits looks like:



As in ordinary algebra, in mixed expressions, all ANDs (Boolean multiplication) are performed *before* ORs (Boolean addition). For example, A+B·C is evaluated by ANDing B with C and then ORing A with the result of the first operation (BC).

Parenthesis can be used to change the ordinary order of evaluation. For example, (A+B)·C is evaluated by ORing A with B and then ANDing C with the result of the first operation (A+B). Parenthesis can be used for clarity.

Negation of a single variable or object is done *before* using the result in an expression. Negation of an entire expression is done *after* the expression is evaluated.

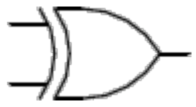
XOR (or exclusive or) is similar to the normal English meaning of the word “or” — a choice between two items, but not both or none. XOR is less commonly written EOR. The symbol for the XOR operation is $\oplus$.

The truth table for XOR (exclusive OR) is as follows:

XOR

A	B	result
0	0	0
1	0	1
0	1	1
1	1	0

The XOR gate in logic circuits looks like:



XOR is not considered to be a basic Boolean operation, but is widely used in logic expressions. XOR is the same as ((A) · (¬B)) + ((¬A) · (B)), extra parenthesis added for clarity.

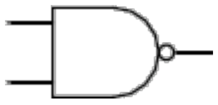
NAND is the combination of a NOT and an AND. NAND produces the oppposite of an AND.

The truth table for NAND (Not AND) is as follows:

NAND

A	B	result
0	0	1
1	0	1
0	1	1
1	1	0

The NAND gate in logic circuits looks like:



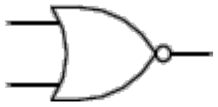
NOR is the combination of a NOT and an OR. NOR produces the opposite of OR.

The truth table for NOR (Not OR) is as follows:

NOR

A	B	result
0	0	1
1	0	0
0	1	0
1	1	0

The NOR gate in logic circuits looks like:



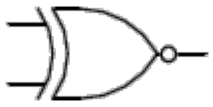
XNOR (or NXOR) is the combination of a NOT and a XOR. XNOR produces the opposite of XOR.

The truth table for XNOR (Not eXclusive OR) is as follows:

XNOR

A	B	result
0	0	1
1	0	0
0	1	0
1	1	1

The XNOR gate in logic circuits looks like:



postulates

Boolean algebra

Boolean algebra is an algebraic system consisting of binary elements and binary operations.

The **postulates** of Boolean algebra provide the foundation for the entire system. The order of the postulates varies greatly from author to author. Some parts of postulates are not strictly necessary (might be derived as theorems instead), but in introductory materials such as this one, are filled out to make details clear to students.

Basic set: X, Y, and Z are elements of the set S.
Note that some authorities use the elements A, B, and C instead.

Equivalence: Equivalence is defined for the set S such that:
if $X = Y$ and $Y = Z$
then $X = Z$

Operations: The operations + (Boolean addition) and · (Boolean multiplication) are defined such that:
$X + Y$ and $X \cdot Y$ are in the set S
NOTE: These two operations were informally introduced in the introduction chapter.

Values: All elements in the set S will take on the valuation:
$S = (0,1)$
NOTE: These two values are often interpreted as false (0) and true (1).

Complement Law: Each member of the set S has an inverse (or compliment), such that when $X = 0$, then $\neg X = 1$
NOTE: The compliment is also indicated by a "tick" mark after the variable (X') and by placing a bar (or horizontal line) over a variable.
NOTE: The Complement Law produces the following important relationships:
$X \cdot \neg X = 0$
$X + \neg X = 1$

Identity Law: The value 1 is the identity for Boolean multiplication and the value 0 is the identity for Boolean addition.
NOTE: The Identity Law produces the following important relationships:
$X + 0 = X$
$X + 1 = 1$

Cummulative Law: Boolean addition and Boolean multiplication are both cummulative:
$X + Y = Y + X$
$X \cdot Y = Y \cdot X$

Distributive Law: The Distributive Law in Boolean algebra highlights its different nature from normal linear algebra, as this would not be true for normal algebra:
$X \cdot (Y Z) = (X \cdot Y) + (X \cdot Z)$
$X + (Y Z) = (X + Y) \cdot (X + Z)$

There are slight variations and alternate axiomatizations of Boolean algebra. The following are often included as axioms in some works. These can be proven from the above. If these axioms are used, then some of the above can become theorems.

Duality Principle: Duality holds that for any valid expression of identity, the resulting expression obtained by interchanging 1 and 0 and · and +, is a valid dual.
NOTE: This gives:
for the expression: $X + \neg X = 1$
the dual is: $X \cdot \neg X = 0$

Idempotent Law: Property of a variable operating on itself.
$X + X = X$
$X \cdot X = X$

Associative Law:
$X \cdot (Y \cdot Z) = (X \cdot Y) \cdot Z$
$X + (Y + Z) = (X + Y) + Z$

Absorption Law:
$X \cdot (X + Y) = X$
$X + (X \cdot Y) = X$

Null Law:
$X + 1 = 1$
$X \cdot 0 = 0$

Note that Boolean algebra does not have a subtraction or a division operation, but it does have a complement operation that isn't found in normal linear algebra.

Assembly Language

introduction

This section examines assembly languages in a general manner. Specific examples of addressing modes and instructions from various processors are used to illustrate the general nature of assembly language.

- general
  - history
  - comparison of assembly and high level languages
  - nature of assembly language
- kinds of processors
  - complex instruction set computers (CISC)
  - reduced instruction set computers (RISC)
  - hybrid processors
  - special purpose processors
  - hypothetical processors
- executable instructions

general

Unlike the other programming languages catalogued here, assembly language is not a single language, but rather a group of languages. Each processor family (and sometimes individual processors within a processor family) has its own assembly language.

In contrast to high level languages, data structures and program structures in assembly language are created by directly implementing them on the underlying hardware. So, instead of catalogueing the data structures and program structures that can be built (in assembly language you can build any structures you so desire, including new structures nobody else has ever created), we will compare and contrast the hardware capabilities of various processor families.

This chapter does not attempt to teach how to program in assembly language. Because of the close relationship between assembly languages and the underlying hardware, this chapter will discuss hardware implementation as well as software.

history

history: The oldest non-machine language, allowing for a more human readable method of writing programs than writing in binary bit patterns (or even hexadecimal patterns).

comparison of assembly and high level languages

Assembly languages are close to a one to one correspondence between symbolic instructions and executable machine codes. Assembly languages also include directives to the assembler, directives to the linker, directives for organizing data space, and macros. Macros can be used to combine several assembly language instructions into a high level language-like construct (as well as other purposes). There are cases where a symbolic instruction is translated into more than one machine instruction. But in general, symbolic assembly language instructions correspond to individual executable machine instructions.

High level languages are abstract. Typically a single high level instruction is translated into several (sometimes dozens or in rare cases even hundreds) executable machine language instructions. Some early high level languages had a close correspondence between high level instructions and machine language instructions. For example, most of the early COBOL instructions translated into a very obvious and small set of machine instructions. The trend over time has been for high level languages to increease in abstraction. Modern object oriented programming languages are highly abstract (although, interestingly, some key object oriented programming constructs do translate into a very compact set of machine instructions).

Assembly language is much harder to program than high level languages. The programmer must pay attention to far more detail and must have an intimate knowledge of the processor in use. But high quality hand crafted assembly language programs can run much faster and use much less memory and other resources than a similar program written in a high level language. Speed increases of two to 20 times faster are fairly common, and increases of hundreds of times faster are occassionally possible. Assembly language programming also gives direct access to key machine features essential for implementing certain kinds of low level routines, such as an operating system kernel or microkernel, device drivers, and machine control.

High level programming languages are much easier for less skilled programmers to work in and for semi-technical managers to supervise. And high level languages allow faster development times than work in assembly language, even with highly skilled programmers. Development time increases of 10 to 100 times faster are fairly common. Programs written in high level languages (especially object oriented programming languages) are much easier and less expensive to maintain than similar programs written in assembly language (and for a successful software project, the vast majority of the work and expense is in maintenance, not initial development).

For information on how to interface high level languages and assembly languages, see the chpater on assembly/high level language interface<

availability

availability: Assemblers are available for just about every processor ever made. **Native** assemblers produce object code on the same hardware that the object code will run on. **Cross** assemblers produce object code on different hardware that the object code will run on.

structure

format: free form or column (depends on the assembly langauge)

nature: procedural language with one to one correspondence between language mnemonics and executable machine instructions.

> "Assembler languages occupy a unique place in the computing world. Since most assmebler-language statements are symbolic of individual machine-language instructions, the assembler-language programmer has the full power of the computer at his disposal in a way that users of other languages do not. Because of the direct relationship between assembler language and machine language, assembler language is used when high efficiency of programs is needed, and especially in areas of application that are so new and amorphous that existing program-oriented languages are ill-suited for describing the procedures to be followed." —**Assembler Language Programming** by George W. Struble, page vii

> "Perhaps the most glaring difference among the three types of languages [high level, assembly, and machine] is that as we move from high-level languages to lower levels, the

code gets harder to read (with understanding). The major advantages of high-level languages are that they are easy to read and are machine independent. The instructions are written in a combination of English and ordinary mathematical notation, and programs can be run with minor, if any, changes on different computers." —**VAX-11 Assembly Language Programming** by Sara Baase, page 1

"The second most visible difference among the different types of languages is that several lines of assembly language are needed to encode one line of a high-level language program." —**VAX-11 Assembly Language Programming** by Sara Baase, page 2

"There are a number of situations in which it is very desirable to use assembler language routines to do part of a job, and use some higher-level language for other parts. It makes sense to use higher-level languages such as Fortran, COBOL, or PL/I for parts of procedures for which they are well-suited, and supplement with assembler language routines for those parts of procedures for which the higher-level language is awkward or inefficient." —**Assembler Language Programming** by George W. Struble, page 427

"If one has a choice between assembly language and a high-level language, why choose assembly language? The fact that the amount of programming done in assembly language is quite small compared to the amount done in high-level languages indicates that one generally doesn't choose assembly language. However, there are situations where it may not be convenient, efficient, or possible to write programs in hihg-level languages. … Programs to control and communicate with peripheral devices (input and output devices) are usually written in assembly language because they use special instructions that are not available in high-level languages, and they must be very efficient. Some systems programs are written in assembly language for similar reasons. In general, since high-level languages are designed without the features of a particular machine in mind and a compiler must do its job in a standardized way to accomodate all valid programs, there are situations where to take advantage of special features of a machine, to program some details that are inaccessible from a high-level language, or perhaps to increase the efficiency of a program, one may reasonably choose to write in assembly language." —**VAX-11 Assembly Language Programming** by Sara Baase, page 3-4

"In situations where programming in a high-level language is not appropriate, it is clear that assembly language is to be preferred to machine language. Assembly language has a number of advantages over machine code aside from the obvious increase in readability. One is that the use of symbolic names for data and instruction labels frees the programmer from computing and recomputing the memory locations whenever a change is made in a program. Another is that assembly languages generally have a feature, called macros, that frees the [programmer] from having to repeat similar sections of code used in several places in a program. Assemblers fo many bookkeeping and other tasks for the user. Often compilers translate into assembly language rather than machine code." —**VAX-11 Assembly Language Programming** by Sara Baase, page 3

kinds of processors

Processors can broadly be divided into the categories of: CISC, RISC, hybrid, and special purpose.

Complex Instruction Set Computers (CISC) have a large instruction set, with hardware support for a wide variety of operations. In scientific, engineering, and mathematical operations with hand coded assembly language (and some business applications with hand coded assembly language), CISC processors usually perform the most work in the shortest time.

Reduced Instruction Set Computers (RISC) have a small, compact instruction set. In most business applications and in programs created by compilers from high level language source, RISC processors usually perform the most work in the shortest time.

Hybrid processors are some combination of CISC and RISC approaches, attempting to balance the advantages of each approach.

Special purpose processors are optimized to perform specific functions. Digital signal processors and various kinds of co-processors are the most common kinds of special purpose processors.

executable instructions

There are four general classes of machine instructions. Some instructions may have characteristics of more than one major group. The four general classes of machine instructions are: computation, data transfer, sequencing, and environment control.

- “**Computation:** Implements a function from n-tuples of values to m-tuples of values. The function may affect the state. Example: A divide instruction whose arguments are a single-length integer divisor and a double-length integer dividend, whose results are a single-length integer quotient and a single-length integer remainder, and which may produce a divide check interrupt.” —**Compiler Construction**, by William M. Waite and Gerhard Goos, page 52
- “**Data Transfer:** Copies information, either within one storage class or from one storage class to another. Examples: A move instruction that copies the contents of one register to another; a read instruction that copies information from a disc to main storage.” —**Compiler Construction**, by William M. Waite and Gerhard Goos, page 52
- “**Sequencing:** Alters the normal execution sequence, either conditionally or unconditionally. Examples: a halt instruction that causes execution to terminate; a conditional jump instruction that causes the next instruction to be taken from a given address if a given register contains zero.” —**Compiler Construction**, by William M. Waite and Gerhard Goos, page 53
- “**Environment control:** Alters the environment in which execution is carried out. The lateration may involve a trasnfer of control. Examples: An interrupt disable instruction that prohibits certain interrupts from occurring; a procedure call instruction that updates addressing registers, thus changing the program’s addressing environment.” —**Compiler Construction**, by William M. Waite and Gerhard Goos, page 53

Executable instructions can be divided into several broad categories of related operations:

- data movement
- address movement
- integer arithmetic
- floating arithmetic
- binary coded decimal
- advanced math
- data conversion
- logical
- shift and rotate
- bit manipulation
- character and string
- table operations
- high level language support
- program control

- condition codes
- input/output
    - MIX devices
- system control
- coprocessor and multiprocessor
- trap generating

data representation

This chapter examines data representation and number systems for assembly languages.

- data representation
    - size
    - endian
    - number systems
    - number representations
        - integer representations
            - sign magnitude
            - one’s complement
            - two’s complement
            - unsigned
        - floating point representations

data representation

Most data structures are abstract structures and are implemented by the programmer with a series of assembly language instructions. Many cardinal data types (bits, bit strings, bit slices, binary integers, binary floating point numbers, binary encoded decimals, binary addresses, characters, etc.) are implemented directly in hardware for at least parts of the instruction set. Some processors also implement some data structures in hardware for some instructions — for example, most processors have a few instructions for directly manipulating character strings.

An assembly language programmer has to know how the hardware implements these cardinal data types. Some examples: Two basic issues are bit ordering (big endian or little endian) and number of bits (or bytes). The assembly language programmer must also pay attention to word length and optimum (or required) addressing boundaries. Composite data types will also include details of hardware implementation, such as how many bits of mantissa, characteristic, and sign, as well as their order. In many cases there are machine specific encodings for some data types, as well as choice of character codes (such as ASCII or EBCDIC) for character and string implementations.

data size

The basic building block is the **bit**, which can contain a single piece of binary data (true/false, zero/one, north/south, positive/negative, high/low, etc.).

Bits are organized into larger groupings to store values encoded in binary bits. The most basic grouping is the **byte**. A byte is the smallest normally addressable quantum of main memory (which can be different than the minimum amount of memory fetched at one time). In modern computers this is almost always an eight bit byte, so much so that many skilled programmers believe that a byte is defined as being *always* eight bits. In the past there have been computers with seven, eight, twelve, and sixteen bits. There have also been bit slice computers where the common memory addressing approach is by single bit; in these kinds of computers the term byte actually has no meaning, although eight bits on these computers are likely to be called a byte. Throughout the rest of this discussion, assume the standard eight bit byte applies unless specifically stated otherwise.

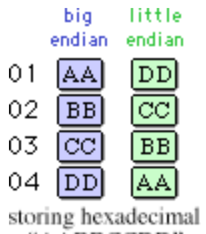
A **nibble** is half a byte, or four bits.

A **word** is the default data size for a processor. The default size does not apply in all cases. The word size is chosen by the processor’s designer(s) and reflects some basic hardware issues (such as internal or external buses). The most common word sizes are 16 and 32, but words have ranged from 16 to 60 bits. Typically there will be additional data sizes that are defined relative to the size of a word: **halfword**, half the size of a word; **longword**, usually double the size of a word; **doubleword**, usually double the size of a word (sometimes double the size of a longword); and **quadword**, four times the size of a word. Whether or not there is a space between the size designation and “word” is designated by the manufacturer, and varies by processor.

Some processors *require* that data be **aligned**. That is, two byte quantities must start on byte addresses that are multiples of two; four byte quantities must start on byte addresses that are multiples of four; etc. The general rule follows a progression of exponents of two (2, 4, 8, 16, ƒ). Some processors allow data to be unaligned, but this usually results in a slow down in performance.

- **DEC VAX** 16 bit [2 byte] word; 32 bit [4 byte] longword; 64 bit [8 byte]quadword; 132 bit [16 byte] octaword; data may be unaligned at a speed penalty
- **IBM 360/370** 32 bit [4 byte] word or full word (which is the smallest amount of data that can be fetched, with words being addresses by the highest order byte); 16 bit [2 byte] half-word; 64 bit [8 byte] double word; all data must be aligned on full word boundaries
- **Intel 80x86** 16 bit [2 byte] word; 32 bit [4 byte] doubleword; data may be unaligned at a speed penalty
- **MIX** byte of unspecified size, must work for both binary and decimal operations without programmer knowledge of size of byte, must be able to contain the values 0 to 63, inclusive, and must not hold more than 100 distinct values, six bits on a binary implementation, two digits on a decimal implementation; word is one sign and five bytes
- **Motorola 680x0** 8 bit byte; 16 bit [2 byte] word; 32 bit [4 byte] long or long word; 64 bit [8 byte] quad word; data may be unaligned at a speed penalty, instructions must be on word boundaries
- **Motorola 68300** 8 bit byte; 16 bit [2 byte] word; 32 bit [4 byte] long or long word; 64 bit [8 byte] quad word; data may be unaligned at a speed penalty, instructions must be on word boundaries

endian



storing hexadecimal “AABBCCDD”

Endian is the ordering of bytes in multibyte scalar data. The term comes from Jonathan Swift’s *Gulliver’s Travels*. For a given multibyte scalar value, big- and little-endian formats are byte-reversed mappings of each other. While processors handle endian issues invisibly when making multibyte memory accesses, knowledge of endian is vital when directly manipulating individual bytes of multibyte scalar data and when moving data across hardware platforms.

Big endian stores scalars in their “natural order”, with most significant byte in the lowest numeric byte address. Examples of big endian processors are the IBM System 360 and 370, Motorola 680x0, Motorola 68300, and most RISC processors.

Little endian stores scalars with the least significant byte in the lowest numeric byte address. Examples of little endian processors are the Digital VAX and Intel x86 (including Pentium).

Bi-endian processors can run in either big endian or little endian mode under software control. An example is the Motorola/IBM PowerPC, which has two separate bits in the Machine State Register (MSR) for controlling endian: the ILE bit controls endian during interrupts and the LE bit controls endian for all other processes. Big endian is the default for the PowerPC.

number systems

Binary is a number system using only ones and zeros (or two states).

Decimal is a number system based on ten digits (including zero).

Hexadecimal is a number system based on sixteen digits (including zero).

Octal is a number system based on eight digits (including zero).

Duodecimal is a number system based on twelve digits (including zero).

binary	octal	decimal	duodecimal	hexadecimal
0	0	0	0	0
1	1	1	1	1
10	2	2	2	2
11	3	3	3	3
100	4	4	4	4
101	5	5	5	5
110	6	6	6	6
111	7	7	7	7
1000	10	8	8	8
1001	11	9	9	9
1010	12	10	A	A
1011	13	11	B	B
1100	14	12	10	C
1101	15	13	11	D
1110	16	14	12	E
1111	17	15	13	F
10000	20	16	14	10
10001	21	17	15	11
10010	22	18	16	12
10011	23	19	17	13
10100	24	20	18	14
10101	25	21	19	15
10110	26	22	1A	16
10111	27	23	1B	17
11000	30	24	20	18

number representations

integer representations

Sign-magnitude is the simplest method for representing signed binary numbers. One bit (by universal convention, the highest order or leftmost bit) is the sign bit, indicating positive or negative, and the remaining bits are the absolute value of the binary integer. Sign-magnitude is simple for representing binary numbers, but has the drawbacks of two different zeros and much more complicates (and therefore, slower) hardware for performing addition, subtraction, and any binary integer operations other than complement (which only requires a sign bit change).

In **one’s complement** representation, positive numbers are represented in the “normal” manner (same as unsigned integers with a zero sign bit), while negative numbers are represented by complementing all of the bits of the absolute value of the number. Numbers are negated by complementing all bits. Addition of two integers is peformed by treating the numbers as unsigned integers (ignoring sign bit), with a carry out of the leftmost bit position being added to the least significant bit (technically, the carry bit is always added to the least significant bit, but when it is zero, the add has no effect). The ripple effect of adding the carry bit can almost double the time to do an addition. And there are still two zeros, a positive zero (all zero bits) and a negative zero (all one bits).

In **two’s complement** representation, positive numbers are represented in the “normal” manner (same as unsigned integers with a zero sign bit), while negative numbers are represented by complementing all of the bits of the absolute value of the number and adding one. Negation of a negative number in two’s complement representation is accomplished by complementing all of the bits and adding one. Addition is performed by adding the two numbers as unsigned integers and ignoring the carry. Two’s complement has the further advantage that there is only one zero (all zero bits). Two’s complement representation does result in one more negative number (all one bits) than positive numbers.

Two’s complement is used in just about every binary computer ever made. Most processors have one more negative number than positive numbers. Some processors use the “extra” neagtive number (all one bits) as a special indicator, depicting invalid results, not a number (NaN), or other special codes.

In **unsigned** representation, only positive numbers are represented. Instead of the high order bit being interpretted as the sign of the integer, the high order bit is part of the number. An unsigned number has one power of two greater range than a signed number (any representation) of the same number of bits.

bit pattern	*sign-mag.*	*one’s comp.*	*two’s comp*	*unsigned*
000	0	0	0	0
001	1	1	1	1
010	2	2	2	2
011	3	3	3	3
100	-0	-3	-4	4
101	-1	-2	-3	5
110	-2	-1	-2	6
111	-3	-0	-1	7

floating point representations

Floating point numbers are the computer equivalent of “scientific notation” or “engineering notation”. A floating point number consists of a fraction (binary or decimal) and an exponent (bianry or decimal). Both the fraction and the exponent each have a sign (positive or negative).

In the past, processors tended to have proprietary floating point formats, although with the development of an IEEE standard, most modern processors use the same format. Floating point numbers are almost always binary representations, although a few early processors had (binary coded) decimal representations. Many processors (especially early mainframes and early microprocessors) did not have any hardware support for floating point numbers. Even when commonly available, it was often in an optional processing unit (such as in the IBM 360/370 series) or coprocessor (such as in the Motorola 680x0 and pre-Pentium Intel 80x86 series).

Hardware floating point support usually consists of two sizes, called **single precision** (for the smaller) and **double precision** (for the larger). Usually the double precision format had twice as many bits as the single precision format (hence, the names single and double). Double precision floating point format offers greater range and precision, while single precision floating point format offers better space compaction and faster processing.

F_floating format (single precision floating), DEC VAX, 32 bits, the first bit (high order bit in a register, first bit in memory) is the sign magnitude bit (one=negative, zero=positive or zero), followed by 15 bits of an excess 128 binary exponent, followed by a normalized 24-bit fraction with the redundant most significant fraction bit not represented. Zero is represented by all bits being zero (allowing the use of a longword CLR to set a F_floating number to zero). Exponent values of 1 through 255 indicate true binary exponents of -127 through 127. An exponent value of zero together with a sign of zero indicate a zero value. An exponent value of zero together with a sign bit of one is taken as reserved (which produces a reserved operand fault if used as an operand for a floating point instruction). The magnitude is an approximate range of $.29*10^{-38}$ through $1.7*10^{38}$. The precision of an F_floating datum is approximately one part in 2^{23}, or approximately seven (7) decimal digits).

32 bit floating format (single precision floating), AT&T DSP32C, 32 bits, the first bit (high order bit in a register, first bit in memory) is the sign magnitude bit (one=negative, zero=positive or zero), followed by 23 bits of a normalized two’s complement fractional part of the mantissa, followed by an eight bit exponent. The magnitude of the mantissa is always normalized to lie between 1 and 2. The floating point value with exponent equal to zero is reserved to represent the number zero (the sign and mantissa bits must also be zero; a zero exponent with a nonzero sign and/or mantissa is called a “dirty zero” and is never generated by hardware; if a dirty zero is an operand, it is treated as a zero). The range of nonzero positive floating point numbers is $N = [1 * 2^{-127}, [2-2^{-23}] * 2^{127}]$ inclusive. The range of nonzero negative floating point numbers is $N = [-[1 + 2^{-23}] * 2^{-127}, -2 * 2^{127}]$ inclusive.

40 bit floating format (extended single precision floating), AT&T DSP32C, 40 bits, the first bit (high order bit in a register, first bit in memory) is the sign magnitude bit (one=negative, zero=positive or zero), followed by 31 bits of a normalized two’s complement fractional part of the mantissa, followed by an eight bit exponent. This is an internal format used by the floating point adder, accumulators, and certain DAU units. This format includes an additional eight guard bits to increase accuracy of intermediate results.

D_floating format (double precision floating), DEC VAX, 64 bits, the first bit (high order bit in a register, first bit in memory) is the sign magnitude bit (one=negative, zero=positive or zero), followed by 15 bits of an excess 128 binary exponent, followed by a normalized 48-bit fraction with the redundant most significant fraction bit not represented. Zero is represented by all bits being zero (allowing the use of a quadword CLR to set a D_floating number to zero). Exponent values of 1 through 255 indicate true binary exponents of -127 through 127. An exponent value of zero together with a sign of zero indicate a zero value. An exponent value of zero together with a sign bit of one is taken as reserved (which produces a reserved operand fault if used as an operand for a floating point instruction). The magnitude is an approximate range of $.29*10^{-38}$ through $1.7*10^{38}$. The precision of an D_floating datum is approximately one part in 2^{55}, or approximately 16 decimal digits).

register set

This chapter examines the use of registers in assembly language. Specific examples of registers from various processors are used to illustrate the general nature of assembly language.

register set

Registers are fast memory, almost always connected to circuitry that allows various arithmetic, logical, control, and other manipulations, as well as possibly setting internal flags.

Most early computers had only one data register that could be used for arithmetic and logic instructions. Often there would be additional special purpose registers set aside either for temporary fast internal storage or assigned to logic circuits to implement certain instructions. Some early computers had one or two address registers that pointed to a memory location for memory accesses (a pair of address registers typically would act as source and destination pointers for memory operations). Computers soon had multiple data registers, address registers, and sometimes other special purpose registers. Some computers have general purpose registers that can be used for both data and address operations. Every digital computer using a von Neumann architecture has a register (called the program counter) that points to the next executable instruction. Many computers have additional control registers for implementing various control capabilities. Often some or all of the internal flags are combined into a flag or status register.

accumulators

Accumulators are registers that can be used for arithmetic, logical, shift, rotate, or other similar operations. The first computers typically only had one accumulator. Many times there were related special purpose registers that contained the source data for an accumulator. Accumulators were replaced with data registers and general purpose registers. Accumulators reappeared in the first microprocessors.

- **Intel 8086/80286:** one word (16 bit) accumulator; named AX (high order byte of the AX register is named AH and low order byte of the AX register is named AL)
- **Intel 80386:** one doubleword (32 bit) accumulator; named EAX (low order word uses the same names as the accumulator on the Intel 8086 and 80286 [AX] and low order and high order bytes of the low order words of four of the registers use the same names as the accumulator on the Intel 8086 and 80286 [AH and AL])

- **MIX:** one accumulator; named A-register; five bytes plus sign

data registers

Data registers are used for temporary scratch storage of data, as well as for data manipulations (arithmetic, logic, etc.). In some processors, all data registers act in the same manner, while in other processors different operations are performed are specific registers.

- **MIX:** one extension register; named X-register; five bytes plus sign; can be concatenated on the right hand side of the A-register (accumulator)
- **Motorola 680x0, 68300:** 8 longword (32 bit) data registers; named D0, D1, D2, D3, D4, D5, D6, and D7

address registers

Address registers store the addresses of specific memory locations. Often many integer and logic operations can be performed on address registers directly (to allow for computation of addresses).

Sometimes the contents of address register(s) are combined with other special purpose registers to compute the actual physical address. This allows for the hardware implementation of dynamic memory pages, virtual memory, and protected memory.

The number of bits of an address register (possibly combined with information from other registers) limits the maximum amount of addressable memory. A 16-bit address register can address 64K of physical memory. A 24-bit address register can address address 16 MB of physical memory. A 32-bit address register can address 4 GB of physical memory. A 64-bit address register can address 1.8446744×10^{19} of physical memory. Addresses are always unsigned binary numbers.

- **MIX:** one jump registers; named J-register; two bytes and sign is always positive
- **Motorola 680x0, 68300:** 8 longword (32 bit) address registers; named A0, A1, A2, A3, A4, A5, A6, and A7 (also called the stack pointer)

general purpose registers

General purpose registers can be used as either data or address registers.

- **DEC VAX:** 16 word (32 bit) general purpose registers; named R0 through R15
- **IBM 360/370:** 16 full word (32 bit) general purpose registers; named 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A (or 10), B (or 11), C (or 12), D (or 13), E (or 14), and F (or 15)
- **Intel 8086/80286:** 8 word (16 bit) general purpose registers; named AX, BX, CX, DX, BP, SP, SI, and DI (high order bytes of the AX, BX, CX, and DX registers have the names AH, BH, CH, and DH and low order bytes of the AX, BX, CX, and DX registers have the names AL, BL, CL, and DL)
- **Intel 80386:** 8 doubleword (32 bit) general purpose registers; named EAX, EBX, ECX, EDX, EBP, ESP, ESI, and EDI (low order words use the same names as the general purpose registers on the Intel 8086 and 80286 and low order and high order bytes of the low order words of four of the registers use the same names as the general purpose registers on the Intel 8086 and 80286)
- **Motorola 88100:** 32 word (32 bit) general purpose registers; named r0 through r31

constant registers

Constant registers are special read-only registers that store a constant. Attempts to write to a constant register are illegal or ignored. In some RISC processors, constant registers are used to store commonly used values (such as zero, one, or negative one) — for example, a constant register containing zero can be used in register to register data moves, providing the equivalent of a clear instruction without adding one to the instruction set. Constant registers are also often used in floating point units to provide such value as pi or e with additional hidden bits for greater accuracy in computations.

- **Motorola 88100:** r0 (general purpose register 0) contains the constant 32 bit integer zero

floating point registers

Floating point registers are special registers set aside for floating point math.

index registers

Index registers are used to provide more flexibility in addressing modes, allowing the programmer to create a memory address by combining the contents of an address register with the contents of an index register (with displacements, increments, decrements, and other options). In some processors, there are specific index registers (or just one index register) that can only be used only for that purpose. In some processors, any data register, address register, or general register (or some combination of the three) can be used as an index register.

- **IBM 360/370:** any of the 16 general purpose registers may be used as an index register
- **Intel 80x86:** 7 of the 8 general purpose registers may be used as an index register (the ESP is the exception)
- **MIX:** five index registers; named I-registers I1, I2, I3, I4, and I5; five bytes plus sign
- **Motorola 680x0, 68300:** any of the 8 data registers or the 8 address registers may be used as an index register

base registers

Base registers or **segment registers** are used to segment memory. Effective addresses are computed by adding the contents of the base or segment register to the rest of the effective address computation. In some processors, any register can serve as a base register. In some processors, there are specific base or segment registers (one or more) that can only be used for that purpose. In some processors with multiple base or segment registers, each base or segment register is used for different kinds of memory accesses (such as a segment register for data accesses and a different segment register for program accesses).

- **IBM 360/370:** any of the 16 general purpose registers may be used as a base register
- **Intel 80x86:** 6 dedicated segment registers: CS (code segment), SS (stack segment), DS (data segment), ES (extra segment, a second data segment register), FS (third data segment register), and GS (fourth data segment register)
- **Motorola 680x0, 68300:** any of the 8 address registers may be used as a base register

control registers

Control registers control some aspect of processor operation. The most universal control register is the program counter.

program counter

Almost every digital computer ever made uses a **program counter**. The program counter points to the memory location that stores the next executable instruction. Branching is implemented by making changes to the program counter. Some processor designs allow software to directly change the program counter, but usually software only indirectly changes the program counter (for example, a JUMP instruction will insert the operand into the program counter). An assembler has a **location counter**, which is an internal pointer to the address (first byte) of the next location in storage (for instructions, data areas, constants, etc.) while the source code is being converted into object code.

The VAX uses the 16th of 16 general purpose registers as the program counter (PC). Almost the entire instruction set can directly manipulate the program counter, allowing a very rich set of possible kinds of branching.

The program counter in System/360 and 370 machines is contained in bits 40-63 of the program status word (PSW), which is directly accessible by some instructions.

- **IBM 360/370:** program counter is bits 40-63 of the program status word (PSW)
- **Intel 8086/80286:** 16-bit instruction pointer (IP)
- **Intel 80386:** 32-bit instruction pointer (EIP)
- **Motorola 680x0, 68300:** 32-bit program counter (PC)

processor flags

Processor flags store information about specific processor functions. The processor flags are usually kept in a **flag register** or a general **status register**. This can include **result flags** that record the results of certain kinds of testing, information about data that is moved, certain kinds of information about the results of compations or transformations, and information about some processor states. Closely related and often stored in the same processor word or status register (although often in a privileged portion) are **control flags** that control processor actions or processor states or the actions of certain instructions.

- **IBM 360/370:** program status word (PSW)
- **Intel 8086/80286:** 16-bit flag register (FLAGS); system flags, control flag, and status flags)
- **Intel 80386:** 32-bit flag register (EFLAGS); system flags, control flag, and status flags)
- **MIX:** an overflow toggle and a comparison indicator
- **Motorola 680x0, 68300:** 16-bit status register (SR); high byte is system byte and requires privileged access, low byte is user byte or condition code register (CCR)

A few typical **result flags** (with processors that include them):

- **auxilary carry** Set if a carry out of the most significant bit of a BCD operand occurs (binary coded decimal addition). Also commonly set if a borrow occurs in a BCD subtract. Used in Intel 80x86 [AF].
- **carry** Set if a carry out of the most significant bit of an operand occurs (addition). Also commonly set if a borrow occurs in a subtract. Used in Digital VAX [C], Intel 80x86 [CF], Motorola 680x0 [C], Motorola 68300 [C], Motorola M68HC16 [C].
- **comparison indicator** contains one of three values: less, equal, or greater. Used in MIX.
- **extend** Set to the value of the carry bit for arithmetic operations (used to support implementation of multi-byte arithmetic larger than that implemented directly by the hardware. Used in Motorola 680x0 [X], Motorola 68300 [X].
- **half carry** Set if a carry out of bit 3 of an operand occurs during BCD addition. Used in Motorola M68HC16 [H].
- **negative** Set if the most significant bit of a result is set. Used in Digital VAX [N], Motorola 680x0 [N], Motorola 68300 [N], Motorola M68HC16 [N].
- **overflow** Set if arithmetic overflow occurs. Used in Digital VAX [V], Intel 80x86 [OF], Motorola 680x0 [V], Motorola 68300 [V], Motorola M68HC16 [V].

- **overflow toggle** a single bit that is either on or off. Used in MIX.
- **parity** For odd parity machines, set to make an odd number of one bits; for an even parity machine, set to make an even number of one bits. Used in Intel 80x86 [PF]. The IBM 360/370 has odd parity on memory.
- **sign** Set for negative sign. Used in Intel 80x86 [SF].
- **trap** Set for traps. Used in Intel 80x86 [TF].
- **zero** Set if a result equals zero. Used in Digital VAX [Z], Intel 80x86 [ZF], Motorola 680x0 [Z], Motorola 68300 [Z], Motorola M68HC16 [Z].

Some conditions are determined by combining multiple flags. For example, if a processor has a negative flag and a zero flag, the equivalent of a positive flag is the case of both the negative and zero flags both simultaneously being cleared.

A few typical **control flags** (with processors that include them):

- **decimal overflow trap enable** Set if decimal overflow occurs (or conversion error on a VAX). Used in Digital VAX [DV].
- **direction flag** Determines the direction of string operations (set for autoincrement, cleared for autodecrement). Used in Intel 80x86 [DF].
- **floating underflow trap enable** Set if floating underflow occurs. Used in Digital VAX [FU].
- **integer overflow trap enable** Set if integer overflow occurs (or conversion error on a VAX). Used in Digital VAX [IV].
- **interupt enable** Set if interrupts enabled. Used in Intel 80x86 [IF].
- **i/o privilege level** Used to control access to I/O instructions and hardware (thereby seperating control over I/O from other supervisor/user states). Two bits. Used in Intel 80x86 [IO PL].
- **nested task flag** Used in Intel 80x86 [NF].
- **resume flag** Used in Intel 80x86 [RF].
- **virtual 8086 mode** Used to switch to virtual 8086 emulation. Used in Intel 80x86 [VM].

stack pointer

Stack pointers are used to implement a processor stack in memory. In many processors, address registers can be used as generic data stack pointers and queue pointers. A specific stack pointer or address register may be hardwired for certain instructions. The most common use is to store return addresses, processor state information, and temporary variables for subroutines.

- **IBM 360/370:** any of the 16 general purpose registers may be used as a stack pointer
- **Intel 8086/80286:** dedicated stack pointer (SP) combined with stack segment pointer (SS) to create address of stack
- **Intel 80386:** dedicated stack pointer (ESP) combined with stack segment pointer (SS) and the stack-frame base pointer (EBP) to create address of stack
- **Motorola 680x0, 68300:** dedicated user stack pointer (USP, A7) and system stack pointer (SSP, A7) for implicit stack pointer operations, as well as allowing any of the 8 address registers to be as explicit stack pointers

subroutine return pointer

Some RISC processors include a special **subroutine return pointer** rather than using a stack in memory. The return address for subroutine calls is stored in this register rather than in memory. More than one level of subroutine calls requires storing and saving the contents of this register to and from memory.

- **Motorola 88100:** r1 is a 32 bit register containing the return pointer generated by bsr and jsr instructions; the register can be read or overwritten by software and can even be used as a temporary general purpose data register

Basics of computer memory

Summary: Memory systems in computers. Access to memory (for storage of programs and data) is one of the most basic and lowest level activities of an operating system.

memory hardware issues

main storage

Main storage is also called **memory** or internal memory (to distinguish from external memory, such as hard drives). An older term is **working storage**.

Main storage is fast (at least a thousand times faster than external storage, such as hard drives). Main storage (with a few rare exceptions) is volatile, the stored information being lost when power is turned off.

All data and instructions (programs) must be loaded into main storage for the computer processor.

RAM is Random Access Memory, and is the basic kind of internal memory. RAM is called "random access" because the processor or computer can access *any* location in memory (as contrasted with sequential access devices, which must be accessed in order). RAM has been made from reed relays, transistors, integrated circuits, magnetic core, or anything that can hold and store binary values (one/zero, plus/minus, open/close, positive/negative, high/low, etc.). Most modern RAM is made from integrated circuits. At one time the most common kind of memory in mainframes was magnetic core, so many older programmers will refer to main memory as **core memory** even when the RAM is made from more modern technology. **Static RAM** is called static because it will continue to hold and store information even when power is removed. Magnetic core and reed relays are examples of static memory. **Dynamic RAM** is called dynamic because it loses all data when power is removed. Transistors and integrated circuits are examples of dynamic memory. It is possible to have battery back up for devices that are normally dynamic to turn them into static memory.

ROM is Read Only Memory (it is also random access, but only for reads). ROM is typically used to store thigns that will never change for the life of the computer, such as low level portions of an operating system. Some processors (or variations within processor families) might have RAM and/or ROM built into the same chip as the processor (normally used for processors used in standalone devices, such as arcade video games, ATMs, microwave ovens, car ignition systems, etc.). **EPROM** is Erasable Programmable Read Only Memory, a special kind of ROM that can be erased and reprogrammed with specialized equipment (but not by the processor it is connected to). EPROMs allow makers of industrial devices (and other similar equipment) to have the benefits of ROM, yet also allow for updating or upgrading the software without having to buy new ROM and throw out the old (the EPROMs are collected, erased and rewritten centrally, then placed back into the machines).

Registers and **flags** are a special kind of memory that exists inside a processor. Typically a processor will have several internal registers that are much faster than main memory. These registers usually have specialized capabilities for arithmetic, logic, and other operations. Registers are usually fairly small (8, 16, 32, or 64 bits for integer data, address, and control registers; 32, 64, 96, or 128 bits for floating point registers). Some processors separate integer data and address registers, while other processors have general purpose registers that can be used for both data and address purposes. A processor will typically have one to 32 data or general purpose registers (processors with separate data and address registers typically split the register set in half). Many processors have special floating point registers (and some processors have general purpose registers that can be used for either integer or floating point arithmetic). Flags are single bit memory used for testing, comparison, and conditional operations (especially conditional branching). For a much more detailed look at registers, see the chapter on registers.

external storage

External storage is any storage other than main memory. In modern times this is mostly hard drives and removeable media (such as floppy disks, Zip disks, optical media, etc.). With the advent of USB and FireWire hard drives, the line between permanent hard drives and removeable media is blurred. Other kinds of external storage include tape drives, drum drives, paper tape, and punched cards. Random access or indexed access devices (such as hard drives, removeable media, and drum drives) provide an extension of memory (although usually accessed through logical file systems). Sequential access devices (such as tape drives, paper tape punch/readers, or dumb terminals) provide for off-line storage of large amounts of information (or back ups of data) and are often called I/O devices (for input/output).

buffers

Buffers are areas in main memory that are used to store data (or instructions) being transferred to or from external memory.

basic memory software approaches

static and dynamic approaches

There are two basic approaches to memory usage: static and dynamic.

Static memory approaches assume that the addresses don’t change. This may be a virtual memory illusion, or may be the actual physical layout. The static memory allocation may be through absolute addresses or through PC relative addresses (to allow for relocatable, reentrant, and/or recursive software), but in either case, the compiler or assembler generates a set of addresses that can not change for the life of a program or process.

Dynamic memory approaches assume that the addresses can change (although change is often limited to predefined possible conditions). The two most common dynamic approaches are the use of stack frames and the use of pointers or handlers. Stack frames are used primarily for temporary data (such as fucntion or subroutine variables or loop counters). Handles and pointers are used for keeping track of dynamically allocated **blocks** of memory.

absolute addressing

To look at memory use by programs and operating systems, let’s first examine the more simple problem of a single program with complete control of the computer (such as in a small-scale embedded system or the earliest days of computing).

The most basic form of memory access is **absolute addressing**, in which the program explicitely names the address that is going to be used. An address is a numeric label for a specific location in memory. The numbering system is usually in bytes and always starts counting with zero. The first byte of physical memory is at address 0, the second byte of physical memory is at address 1, the third byte of physical memory is at address 2, etc. Some processors use word addressing rather than byte addressing. The theoretical maximum address is determined by the address size of a processor (a 16 bit address space is limited to no more than 65536 memory locations, a 32 bit address space is limited to approximately 4 GB of memory locations). The actual maximum is limited to the amount of RAM (and

ROM) physically installed in the computer.

A programmer assigns specific absolute addresses for data structures and program routines. These absolute addresses might be assigned arbitrarily or might have to match specific locations expected by an operating system. In practice, the assembler or complier determines the absolute addresses through an orderly predictable assignment scheme (with the ability for the programmer to override the compiler's scheme to assign specific operating system mandated addresses).

This simple approach takes advantage of the fact that the compiler or assembler can predict the exact absolute addresses of every program instruction or routine and every data structure or data element. For almost every processor, absolute addresses are the fastest form of memory addressing. The use of absolute addresses makes programs run faster and greatly simplifies the task of compiling or assembling a program.

Some hardware instructions or operations rely on fixed absolute addresses. For example, when a processor is first turned on, where does it start? Most processors have a specific address that is used as the address of the first instruction run when the processer is first powered on. Some processors provide a method for the start address to be changed for future start-ups. Sometimes this is done by storing the start address internally (with some method for software or external hardware to change this value). For example, on power up the Motorola 680x0, the processor loads the interrupt stack pointer with the longword value located at address 000 hex, loads the program counter with the longword value located at address 004 hex, then starts execution at the frshly loaded program counter location. Sometimes this is done by reading the start address from a data line (or other external input) at power-up (and in this case, there is usually fixed external hardware that always generates the same pre-assigned start address).

Another common example of hardware related absolute addressing is the handling of traps, exceptions, and interrupts. A processor often has specific memory addresses set aside for specific kinds of traps, exceptions, and interrupts. Using a specific example, a divide by zero exception on the Motorola 680x0 produces an exception vector number 5, with the address of the exception handler being fetched by the hardware from memory address 014 hex.

Some simple microprocessor operating systems relied heavily on absolute addressing. An example would be the MS-DOS expectation that the start of a program would always be located at absolute memory address x100h (hexadecimal 100, or decimal 256). A typical compiler or assembler directive for this would be the ORG directive (for "origin").

The key disadvantage of absolute addressing is that multiple programs clash with each other, competing to use the same absolute memory locations for (possibly different) purposes.

overlay

So, how do you implement multiple programs on an operating system using absolute addresses? Or, for early computers, how do you implement a program that is larger than available RAM (especially at a time when processors rarely had more than 1k, 2k, or 4k of RAM? The earliest answer was **overlay** systems.

With an overlay system, each program or program segment is loaded into the exact same space in memory. An overlay handler exists in another area of memory and is responsible for swapping overlay pages or overlay segments (both are the same thing, but different operating systems used different terminology). When a overlay segment completes its work or needs to access a routine in another overlay segment, it signals the overlay handler, which then swaps out the old program segment and swaps in the next program segment.

An overlay handler doesn't take much memory. Typically, the memory space that contained the overlay handler was also padded out with additional routines. These might include key device drivers, interrupt handlers, exception handlers, and small commonly used routines shared by many programs (to save time instead of continual swapping of the small commonly used routines).

In early systems, all data was **global**, meaning that it was shared by and available for both read and writes by any running program (in modern times, global almost always means available to a single entire program, no longer meaning available to all software on a computer). A section of memory was set aside for shared system variables, device driver variables, and interrupt handler variables. An additional area would be set aside as “scratch” or temporary data. The temporary data area would be available for individual programs. Because the earliest operating systems were batch systems, only one program other than the operating system would be running at any one time, so it could use the scratch RAM any way it wanted, saving any long term data to files.

relocatable software

As computer science advance, hardware started to have support for **relocatable** programs and data. This would allow an operating system to load a program anywhere convenient in memory (including a different location each time the program was loaded). This was a necessary step for the jump to interactive operating systems, but was also useful in early batch systems to allow for multiple overlay segments.

demand paging and swapping

Overlay systems were superceded by **demand paging** and **swapping** systems. In a **swapping** system, the operating system swaps out an entire program and its data (and any other context information).

In a **swapping** system, instead of having programs explicitely request overlays, programs were divided into pages. The operating system would load a program’s starting page and start it running. When a program needed to access a data page or program page not currently in main memory, the hardware would generate a **page fault**, and the operating system would fetch the requested page from external storage. When all available pages were filled, the operating system would use one of many schemes for figuring out which page to delete from memory to make room for the new page (and if it was a data page that had any changes, the operating system would have to store a temporary copy of the data page). The question of how to decide which page to delete is one of the major problems facing operating system designers.

program counter relative

One approach for making programs relocatable is **program counter relative addressing**. Instead of branching using absolute addresses, branches (including subroutine calls, jumps, and other kinds of branching) were based on a relative distance from the current program counter (which points to the address of the currently executing instruction). With PC relative addreses, the program can be loaded anywhere in memory and still work correctly. The location of routines, subroutines, functions, and constant data can be determined by the positive or negative distance from the current instruction.

Program counter relative addressing can also be used for determining the address of variables, but then data and code get mixed in the same page or segment. At a minimum, mixing data and code in the same segment is bad programming practice, and in most cases it clashes with more sophisticated hardware systems (such as protected memory).

base pointers

Base pointers (sometimes called segment pointers or page pointers) are special hardware registers that point to the start (or base) of a particular page or segment of memory. Programs can then use an absolute address within a page and either explicitly add the absolute address to the contents of a base pointer or rely on the hardware to add the two together to form the actual **effective address** of the memory access. Which method was used would depend on the processor capabilities and the operatign system design. Hiding the base pointer from the application program both made the program easier to compile and allowed for the operating system to implement program isolation, data/code isolation, protected memory, and other sophisticated services.

As an example, the Intel 80x86 processor has a code segment pointer, a data segment pointer, a stack segment pointer, and an extra segment pointer. When a program is loaded into memory, an operating system running on the Intel 80x86 sets the segment pointers with the beginning of the pages assigned for each purpose for that particular program. If a program is swapped out, when it gets swapped back in, the operating system sets the segment pointers to the new memory locations for each segment. The program continues to run, without being aware that it has been moved in memory.

indirection, pointers, and handles

A method for making data relocatable is to use **indirection**. Instead of hard coding an absolute memory address for a variable or data structure, the program uses a **pointer** that gives the memory address of the variable or data structure. Many processors have address pointer registers and a variety of indirect addressing modes available for software.

In the most simple use of address pointers, software generates the effective address for the pointer just before it is used. Pointers can also be stored, but then the data can’t be moved (unless there is additional hardware support, such as virtual memory or base/segment pointers).

Closely related to pointers are **handles**. Handles are two levels of indirection, or a pointer to a pointer. Instead of the program keeping track of an address pointer to a block of memory that can’t be moved, the program keeps track of a pointer to a pointer. Now, the operating system or the application program can move the underlying block of data. As long as the program uses the handle instead of the pointer, the operating system can freely move the data block and update the pointer, and everything will continue to resolve correctly.

Because it is faster to use pointers than handles, it is common for software to convert a handle into a pointer and use the pointer for data accesses. If this is done, there must be some mechanism to make sure that the data block doesn’t move while the program is using the pointer. As an example, the Macintosh uses a system where data blocks can only be moved at specific known times, and an application program can rely on pointers derived from handles remaining valid between those known, specified times.

stack frames

Stack frames are a method for generating temporary variables, especially for subroutines, functions, and loops. An are of memory is temporarily allocated on the system or process stack. In a simple version, the variables in the stack frame are accessed by using the stack pointer and an offset to point to the actual location in memory. This simple approach has the problem that there are many hardware instructions that change the stack pointer. The more sophisticated and stable approach is to have a second pointer called a **frame pointer**. The frame pointer can be set up in software using any address register. Many modern processors also have specific hardware instructions that allocate the stack frame and set up the frame pointer at the same time. Some processors have a specific hardware frame pointer register.

virtual memory

Virtual memory is a technique in which each process generates addresses as if it had sole access to the entire **logical address space** of the processor, but in reality **memory management hardware** remaps the logical addresses into actual physical addresses in **physical address space**. The DEC VAX-11 gets it name from this technique, VAX standing for Virtual Address eXtension.

Virtual memory can go beyond just remapping logical addresses into physical addresses. Many virtual memory systems also include software for page or segment swapping, shuffling portions of a program to and from a hard disk, to give the software the impression of having much more RAM than is actually installed on the computer.

OS memory services

Operating systems offer some kind of mechanism for (both system and user) software to access memory.

In the most simple approach, the entire memory of the computer is turned over to programs. This approach is most common in single tasking systems (only one program running at a time). Even in this approach, there often will be certain portions of memory designated for certain purposes (such as low memory variables, areas for operating system routines, memory mapped hardware, video RAM, etc.).

With hardware support for virtual memory, operating systems can give programs the illusion of having the entire memory to themselves (or even give the illusion there is more memory than there actually is, using disk space to provide the extra “memory”), when in reality the operating system is continually moving programs around in memory and dynamically assigning physical memory as needed. Even with this approach, it is possible that some virtual memory locations are mapped to their actual physical addresses (such as for access to low memory variables, video RAM, or similar areas).

The task of dividing up the available memory space in both of these approaches is left to the programmer and the compiler. Many modern languages (including C and C++) have service routines for allocating and deallocating blocks of memory.

Some operating systems go beyond basic flat mapping of memory and provide operating system routines for allocating and deallocating memory. The Macintosh, for example, has two heaps (a system heap and an application heap) and an entire set of operating system routines for allocating, deallocating, and managing blocks of memory. The NeXT goes even further and creates an object oriented set of services for memory management.

With hardware support for segments or demand paging, some operating systems (such as MVS and OS/2) provide operating system routines for programs to manage segments or pages of memory.

Memory maps (not to be confused with memory mapped I/O) are diagrams or charts that show how an operating system divides up main memory.

Low memory is the memory at the beginning of the address space. Some processors use designated low memory addresses during power on, exception processing, interrupt processing, and other hardware conditions. Some operating systems use designated low memory addresses for global system variables, global system structures, jump tables, and other system purposes.

address space and addressing modes

This chapter examines addressing modes in assembly language. Specific examples of addressing modes from various processors are used to illustrate the general nature of assembly language.

- address space
- address modes
    - absolute address
    - immediate data
    - inherent address
    - register direct
    - register indirect
        - address register indirect
        - address register indirect with postincrement
        - address register indirect with predecrement
        - address register indirect with preincrement
        - address register indirect with postdecrement
        - address register indirect with displacement
    - base register
    - register indirect with index register
        - address register indirect with index register
        - address register indirect with index register and displacement
        - absolute address with index register
    - memory indirect
        - memory indirect post indexed
        - memory indirect pre indexed
    - program counter relative
        - program counter indirect with displacement
        - program counter indirect with index and displacement
        - program counter memory indirect postindexed
        - program counter memory indirect preindexed

address space

Address space is the maximum amount of memory that a processor can address. Some processors use a multi-level addressing scheme, with main memory divided into segments or pages and some or all instructions mapping into the current segment(s) or page(s).

- **MIX:** 4000 words of storage

address modes

The basic addressing modes are: **register direct**, moving date to or from a specific register; **register indirect**, using a register as a pointer to memory; **program counter-based**, using the program counter as a reference point in memory; **absolute**, in which the memory addressis contained in the instruction; and **immediate**, in which the data is contained in the instruction. Some instructions will have an **inherent** or **implicit** address (usually a specific register or the memory contents pointed to by a specific register) that is implied by the instruction without explicit declaration.

One approach to processors places an emphasis on flexibility of addressing modes. Some engineers and programmers believe that the real power of a processor lies in its addressing modes. Most addressing modes can be created by combining two or more basic addressing modes, although building the combination in software will usually take more time than if the combination addressing mode existed in hardware (although there is a trade-off that slows down all operations to allow for more complexity).

In a purely othogonal instruction set, every addressing mode would be available for every instruction. In practice, this isn't the case.

Virtual memory, memory pages, and other hardware mapping methods may be layered on top of the addressing modes.

absolute address

In **absolute address** mode, the effective address in memory is part of the instruction. Some processors have full and short versions of absolute addressing (with short versions only pointing to a limited area in memory, normally starting at memory location zero). Unless overridden by hardware for virtual memory mapping, programs that use this address mode can not be moved in memory.

The most basic form of memory access is **absolute addressing**, in which the program explicitely names the address that is going to be used. An address is a numeric label for a specific location in memory. The numbering system is usually in bytes and always starts counting with zero. The first byte of physical memory is at address 0, the second byte of physical memory is at address 1, the third byte of physical memory is at address 2, etc. Some processors use word addressing rather than byte addressing. The theoretical maximum address is determined by the address size of a processor (a 16 bit address space is limited to no more than 65536 memory locations, a 32 bit address space is limited to approximately 4 GB of memory locations). The actual maximum is limited to the amount of RAM (and ROM) physically installed in the computer.

A programmer assigns specific absolute addresses for data structures and program routines. These absolute addresses might be assigned arbitrarily or might have to match specific locations expected by an operating system. In practice, the assembler or complier determines the absolute addresses through an orderly predictable assignment scheme (with the ability for the programmer to override the compiler's scheme to assign specific operating system mandated addresses).

This simple approach takes advantage of the fact that the compiler or assembler can predict the exact absolute addresses of every program instruction or routine and every data structure or data element. For almost every processor, absolute addresses are the fastest form of memory addressing. The use of absolute addresses makes programs run faster and greatly simplifies the task of compiling or assembling a program.

Some hardware instructions or operations rely on fixed absolute addresses. For example, when a processor is first turned on, where does it start? Most processors have a specific address that is used as the address of the first instruction run when the processer is first powered on. Some processors provide a method for the start address to be changed for future start-ups. Sometimes this is done by storing the start address internally (with some method for software or external hardware to change this value). For example, on power up the Motorola 680x0, the processor loads the interrupt stack pointer with the longword value located at address 000 hex, loads the program counter with the longword value located at address 004 hex, then starts execution at the frshly loaded program counter location. Sometimes this is done by reading the start address from a data line (or other external input) at power-up (and in this case, there is usually fixed external hardware that always generates the same pre-assigned start address).

Another common example of hardware related absolute addressing is the handling of traps, exceptions, and interrupts. A processor often has specific memory addresses set aside for specific kinds of traps, exceptions, and interrupts. Using a specific example, a divide by zero exception on the Motorola 680x0 produces an exception vector number 5, with the address of the exception handler being fetched by the hardware from memory address 014 hex.

Some simple microprocessor operating systems relied heavily on absolute addressing. An example would be the MS-DOS expectation that the start of a program would always be located at absolute memory address x100h (hexadecimal 100, or decimal 256). A typical compiler or assembler directive for this would be the ORG directive (for "origin").

The key disadvantage of absolute addressing is that multiple programs clash with each other (expecting to use the same absolute memory locations for different and competing purposes).

- **MIX:** two byte absolute addresses if I field is zero
- **Motorola 680x0, 68300:** 16 bit short and 32 bit long versions; syntax: xxx.W or xxx.L

immediate data

In **immediate data** address mode, the actual data is stored in the instruction. The sizes allowed for immediate data vary by processor and often by instruction (with some instructions having specific implied sizes).

- **Motorola 680x0, 68300:** byte (8 bit), word (16 bit), and long word (32 bit) versions; sign extended; syntax: #xxx

inherent address

Many instructions will have one or more **inherent** or **implicit** addresses. These are addresses that are implied by the instruction rather than explicitly stated. The two most common forms of inherent address are either a specific register or a memory location designated by the contents of a specific register.

register direct

In register direct address mode, the source and/or destination is a register.

Many processors distinguish between data and address register operations (note, in some cases a general purpose register can act as eeither an address or data register).

In **data register direct** operations, flags are typically set or cleared. Data that is smaller than the register may be sign extended or zero filled to fill the entire register, or may be placed only in the portion of the register necessary for the size of the data, leaving the rest of the register unchanged.

- **Motorola 680x0, 68300:** 32 bit data registers; data register direct operations set or clear flags; byte (8 bit), word (16 bit), and long versions (32 bit), only the low order portion of a destination register is changed; syntax: Dn

In **register to register** (RR) operations, data is transferred from one register to another register or an instruction uses a source and destination register.

- **IBM 360/370:** two byte instructions with a source and a destination register; 32 bit data registers; sets or clear flags; full word (32 bit) transfers; syntax: source, destination (as just a hexadecimal number or as a symbolic name)
- **Motorola 680x0, 68300:** instructions with a source and a destination register; 32 bit data registers; sets or clears flags; byte (8 bit), word (16 bit), and long versions (32 bit), only the low order portion of a destination register is changed; syntax: Dn, Dn

In **address register direct** operations, flags are not normally set or cleared. The address is usually sign extended to the full address size of the processor.

- **Motorola 680x0, 68300:** 32 bit address registers; address register direct operations do not modify flags; word (16 bit) and long versions (32 bit, 24 bits for the original 68000), word-size operands are sign-extended to 32 bits; syntax: An

register indirect

In register indirect address mode, the contents of the designated register are used as a pointer to memory. Variations of register indirect include the use of post- or pre- increment, post- or pre-decrement, and displacements.

In **address register indirect** operations, the designated register is used as a pointer to memory.

- **Motorola 680x0, 68300:** syntax: (An)

In **address register indirect with postincrement** operations, the designated register is used as a pointer to memory, and then the register is incremented by the size of the operation. This is useful for a loop where the same or similar operations are performed on consecutive locations in memory. This address mode can be combined with a complimentary predecrement mode for stack and queue operations.

- **Motorola 680x0, 68300:** syntax: (An)+

In **address register indirect with predecrement** operations, the designated register is decremented by the size of the operations, and then the designated register is used as a pointer to memory. This is useful for a loop where the same or similar operations are performed on consecutive locations in memory. This address mode can be combined with a complimentary postincrement mode for stack and queue operations.

- **Motorola 680x0, 68300:** syntax: -(An)

In **address register indirect with preincrement** operations, the designated register is incremented by the size of the operations, and then the designated register is used as a pointer to memory. This is useful for a loop where the same or similar operations are performed on consecutive locations in memory. This address mode can be combined with a complimentary postdecrement mode for stack and queue operations.

In **address register indirect with postdecrement** operations, the designated register is used as a pointer to memory, and then the register is decremented by the size of the operation. This is useful for a loop where the same or similar operations are performed on consecutive locations in memory. This address mode can be combined with a complimentary preincrement mode for stack and queue operations.

In **address register indirect with displacement** operations, the contents of the designated register are modified by adding or subtracting a dispacement integer, then used as a pointer to memory. The displacement integer is stored in the instruction, and if shorter than the length of a the processor's address space (the normal case), sign-extended before addition (or subtraction).

- **Motorola 680x0, 68300:** 16 bit displacement integers, sign-extended to 32 bits; syntax: d(An)

base registers

Base pointers (sometimes called segment pointers or page pointers) are special hardware registers that point to the start (or base) of a particular page or segment of memory. Programs can then use an absolute address within a page and either explicitly add the absolute address to the contents of a base pointer or rely on the hardware to add the two together to form the actual **effective address** of the memory access. Which method was used would depend on the processor capabilities and the operatign system design. Hiding the base pointer from the application program both made the program easier to compile and allowed for the operating system to implement program isolation, data/code isolation, protected memory, and other sophisticated services.

As an example, the Intel 80x86 processor has a code segment pointer, a data segment pointer, a stack segment pointer, and an extra segment pointer. When a program is loaded into memory, an operating system running on the Intel 80x86 sets the segment pointers with the beginning of the pages assigned for each purpose for that particular program. If a program is swapped out, when it gets swapped back in, the operating system sets the segment pointers to the new memory locations for each segment. The program continues to run, without being aware that it has been moved in memory.

register indirect with index register

In a register indirect with index register mode, two registers are added together to form the effective address of a pointer to memory. These are sometimes called the **base register** and **index register**. Many processors will have limits on which registers can be used for the base register and/or which registers can be used for the index register.

In **address/base register indirect with index register** operations, the contents of the index register are added to the contents of the base address register to form an effective address in memory. Some processors allow for designating that less than the full size of the index register be used in the computation, with the designated low order portion of the index register being sign-extended for the effective address computation. Some processors require that a designated low order portion of the index register be used in the computation, with the designated low order portion of the index register being sign-extended for the effective address computation.

In **address/base register indirect with index register and displacement** operations, the contents of the index register are added to the contents of the base address register and then an integer displacement is added or subtracted to form an effective address in memory. Some processors allow for designating that less than the full size of the index register be used in the computation, with the designated low order portion of the index register being sign-extended for the effective address computation. Some processors require that a designated low order portion of the index register be used in the computation, with the designated low order portion of the index register being sign-extended for the effective address computation. The integer displacement is stored in the instruction, and if shorter than the length of a the processor’s address space (the normal case), sign-extended before addition (or subtraction).

- **Motorola 680x0, 68300:** 8 bit, 16 bit, or 32 bit displacement integer; index register component can be word (16 bit) or long (32 bit) and can have a scale factor of 0, 1, 2, 4, or 8; syntax: d(An,Xn.s)

absolute address with index register

In **absolute address with index register** operations, the contents of an index register are added to an absolute address to form an effective address in memory.

- **MIX:** two byte absolute addresses with contents of one of five index registers

memory indirect

In memory indirect address mode, a location in memory contains a value that is used as a pointer (with or without additional effective address computations) to another location in memory.

In **memory indirect postindexed** operations, the processor calculates an intermediate memory address using a base register and a base displacement. The processor accesses the designated memory location, and adds the contents of the index register and an outer displacement to the memory value to yield the effective address. If either displacement and/or the index register is shorter than the length of a the processor's address space (the normal case), each is sign-extended before addition (or subtraction). Base and outer displacements are stored in the instruction.

- **Motorola 680x0, 68300:** 8 bit, 16 bit, or 32 bit base and outer displacement integers; index register component can be word (16 bit) or long (32 bit) and can have a scale factor of 0, 1, 2, 4, or 8; syntax: ([bAn], Xn.s, od)

In **memory indirect preindexed** operations, the processor calculates an intermediate memory address using a base register, a base displacement, and an index register. The processor accesses the designated memory location, and adds an outer displacement to the memory value to yield the effective address. If either displacement and/or the index register is shorter than the length of a the processor's address space (the normal case), each is sign-extended before addition (or subtraction). Base and outer displacements are stored in the instruction.

- **Motorola 680x0, 68300:** 8 bit, 16 bit, or 32 bit base and outer displacement integers; index register component can be word (16 bit) or long (32 bit) and can have a scale factor of 0, 1, 2, 4, or 8; syntax: ([bAn, Xn.s], od)

program counter relative

In program counter indirect addressing, the program counter is used as a reference for the effective address computation. This is most commonly used for short branching relative to the current program counter, allowing for object code that can be placed anywhere in memory.

One approach for making programs relocatable is **program counter relative addressing**. Instead of branching using absolute addresses, branches (including subroutine calls, jumps, and other kinds of branching) were based on a relative distance from the current program counter (which points to the address of the currently executing instruction). With PC relative addreses, the program can be loaded anywhere in memory and still work correctly. The location of routines, subroutines, functions, and constant data can be determined by the positive or negative distance from the current instruction.

Program counter relative addressing can also be used for determining the address of variables, but then data and code get mixed in the same page or segment. At a minimum, mixing data and code in the same segment is bad programming practice, and in most cases it clashes with more sophisticated hardware systems (such as protected memory).

In **program counter indirect with displacement** operations, the effective address is the sum of the address in the program counter and the displacement integer stored in the instruction. If the displacement integer is shorter than the length of a the processor's address space (the normal case), it is sign-extended before addition (or subtraction).

- **Motorola 680x0, 68300:** 16 bit displacement integer; syntax: dPC

In **program counter indirect with index and displacement** operations, the effective address is the sum of the address in the program counter, the contents of the index register, and the displacement integer stored in the instruction. If the displacement integer or designated portion of the index register is shorter than the length of a the processor's address space (the normal case), each is sign-extended before addition (or subtraction).

- **Motorola 680x0, 68300:** 8 bit, 16 bit, or 32 bit displacement integer; index register component can be word (16 bit) or long (32 bit) and can have a scale factor of 0, 1, 2, 4, or 8; syntax: dPC,Xn

In **program counter memory indirect postindexed** operations, the processor calculates an intermediate indirect memory address by adding a base displacement to the contents of the program counter. The value accessed at this memory location is added to the scaled contents of the index register and the outer displacement to yield the effective address. If either the base or outer displacement integer or designated portion of the index register is shorter than the length of a the processor's address space (the normal case), each is sign-extended before addition (or subtraction).

- **Motorola 680x0, 68300:** 8 bit, 16 bit, or 32 bit base and outer displacement integers; index register component can be word (16 bit) or long (32 bit) and can have a scale factor of 0, 1, 2, 4, or 8; syntax: ([dPC],Xn.s,od)

In **program counter memory indirect preindexed** operations, the processor calculates an intermediate indirect memory address by adding a base displacement and scaled contents of an index register to the contents of the program counter. The value accessed at this memory location is added to the outer displacement to yield the effective address. If either the base or outer displacement integer or designated portion of the index register is shorter than the length of a the processor's address space (the normal case), each is sign-extended before addition (or subtraction).

- **Motorola 680x0, 68300:** 8 bit, 16 bit, or 32 bit base and outer displacement integers; index register component can be word (16 bit) or long (32 bit) and can have a scale factor of 0, 1, 2, 4, or 8; syntax: ([dPC,Xn.s],od)

data movement

This chapter examines data movement instructions in assembly language. Specific examples of instructions from various processors are used to illustrate the general nature of assembly language.

- data movement
- address movement

data movement

Data movement instructions move data from one location to another. The source and destination locations are determined by the addressing modes, and can be registers or memory. Some processors have different instructions for loading registers and storing to memory, while other processors have a single instruction with flexible addressing modes. Data movement instructions generally have the greatest options for addressing modes. Data movement instructions typically come in a variety of sizes. Data movement instructions destroy the previous contents of the destination. Data movement instructions typically set and clear processor flags. When the destination is a register and the data is smaller than the full register size, the data might be placed only in the low order bits (leaving high order bits unchanged), or might be zero- or sign-extended to fill the entire register (some processors only use one choice, others permit the programmer to choose how this is handled). Register to register operations can usually have the same source and destination register.

Earlier processors had different instructions and different names for different kinds of data movement, while most modern processors group data movement into a single symbolic name, with different kinds of data movement being indicated by address mode and size designation. A **load** instruction loads a register from memory. A **store** instruction stores the contents of a register into memory. A **transfer** instruction loads a register from another register. In processors that have separate names for different kinds of data moves, a memory to memory data move might be specially designated as a “move” instruction.

An **exchange** instruction exchanges the contents of two registers, two memory locations, or a register and a memory location (although some processors only have register-register exchanges or other limitations).

Some processors include versions of data movement instructions that can perform simple operations during the data move (such as compliment, negate, or absolute value).

Some processors include instructions that can **save** (to memory) or **restore** (from memory) a block of registers at one time (useful for implementing subroutines).

Some processors include instructions that can move a block of memory from one location to another at one time. If a processor includes string instructions, then there will usually be a string instruction that moves a string from one location in memory to another.

- **MOVE** Move Data; Motorola 680x0, Motorola 68300; move a byte (MOVE.B 8 bits), word (MOVE.W 16 bits), or longword (MOVE.L 32 bits) of data; memory to memory, memory to register, register to memory, or register to register; moves of byte and word data to registers leaves high order bits unchanged; sets or clears flags
- **MOV** Move Data; Intel 80x86; move a byte (8 bits), word (16 bits), or doubleword (32 bits) of data; memory to register, register to memory, or register to register (cannot move data from memory to memory or from segment register to segment register); does not affect flags
- **MOV** Move Data; DEC VAX; move a byte (MOVB 8 bits), word (MOVW 16 bits), longword (MOVL 32 bits), quadword (MOVQ 64 bits), octaword (MOVQ 128 bits), single precision

floating (MOVF 32 bits), double precision floating (MOVD 64 bits), G floating (MOVG 64 bits), or H floating (MOVH 128 bits) of data; memory to memory, memory to register, register to memory, or register to register; moves of byte and word data to registers leaves high order bits unchanged; quadword, D float, and G float moves to or from registers are consecutive register pairs, octaword, and H float moves to or from registers are four consecutive registers; and sets or clears flags

- **PUSH** Push; Intel 80x86; decrement stack pointer and move a word (16 bits) or doubleword (32 bits) of data from memory or register (or byte of immediate data) onto stack; does not affect flags
- **PUSHL** Push Long; DEC VAX; decrement stack pointer (R14) and move a longword (32 bits) of data from memory or register onto stack; equivalent to MOVL src, -(SP), but shorter and executes faster; sets or clears flags
- **POP** Pop; Intel 80x86; move a word (16 bits) or doubleword (32 bits) of data from top of stack to register or memory and increment stack pointer; does not affect flags
- **LR** Load from Register; IBM 360/370; RR format; move a full word (32 bits) of data; register to register only; does not affect condition code
- **L** Load (from main storage); IBM 360/370; RX format; move a full word (32 bits) of data; main storage to register only; does not affect condition code
- **LH** Load Half-word; IBM 360/370; RX format; move a half-word (16 bits) of data; main storage to register only; does not affect condition code
- **LDA** Load A-register; MIX; move word or partial word field of data; main storage to accumulator only
- **LDX** Load X-register; MIX; move word or partial word field of data; main storage to extension register only
- **LD*i*** Load index-register; MIX; move word or partial word field of data; main storage to one of five index registers only
- **ST** Store (into main storage); IBM 360/370; RX format; move a full word (32 bits) of data; register to main storage only; does not affect condition code
- **STH** Store Half-word; IBM 360/370; RX format; move a half-word (16 bits) of data; register to main storage only; does not affect condition code
- **STA** Store A-register; MIX; move word or partial word field of data; accumulator to main storage only
- **STX** Store X-register; MIX; move word or partial word field of data; extension register to main storage only
- **ST*i*** Store index-register; MIX; move word or partial word field of data; one of five index registers to main storage only
- **MVI** MoVe Immediate; IBM 360/370; SI format; move a character (8 bits) of data; immediate data to register only; does not affect condition code
- **MOVEQ** Move Quick; Motorola 680x0, Motorola 68300; moves byte (8 bits) of sign-extended data (32 bits) to a data register; sets or clears flags
- **CLR** Clear; Motorola 680x0, Motorola 68300; clears a register or contents of a memory location (.B 8, .W 16, or .L 32 bits) to zero; clears flags for memory and data registers, does not modify flags for address register
- **CLR** Clear; DEC VAX; clears a scalar quantity in register or memory to zero (CLRB 8 bits, CLRW 16 bits, CLRL 32 bits, CLRQ 64 bits, CLRO 128 bits, CLRF 32 bit float, or CLRD 64 bit float), an integer CLR will clear the same size floating point quantity because VAX floating point zero is represented as all zero bits; quadword and D float clears of registers are consecutive register pairs, octaword clears to registers are four consecutive registers; equivalent to MOVx #0, dst, but shorter and executes faster; sets or clears flags
- **STZ** Store Zero; MIX; move word or partial word field of data, store zero into designated word or field of word of memory
- **EXG** Exchange; Motorola 680x0, Motorola 68300; exchanges the data (32 bits) in two data registers; does not affect flags
- **XCHG** Exchange; Intel 80x86; exchanges the data (16 bits or 32 bits) in a register with the AX or EAX register or exchanges the data (8 bits, 16 bits, or 32 bits) in a register with the contents of an effective address (register or memory); LOCK prefix and LOCK# signal asserted in XCGHs

involving memory; does not affect flags

- **MOVSX** Move with Sign Extension; Intel 80x86; moves data from a register or memory to a register, with a sign extension (conversion to larger binary integer: byte to word, byte to doubleword, or word to doubleword); does not affect flags
- **MOVZX** Move with Zero Extension; Intel 80x86; moves data from a register or memory to a register, with a zero extension (conversion to larger binary integer: byte to word, byte to doubleword, or word to doubleword); does not affect flags
- **MOVZ** Move Zero Extended; DEC VAX; moves an unsigned integer to a larger unsigned integer with zero extend, source and destination in register or memory (MOVZBW Byte to Word, MOVZBL Byte to Long, MOVZWL Word to Long); sets or clears flags
- **MCOM** Move Complemented; DEC VAX; moves the logical complement (one's complement) of an integer to register or memory (MCOMB 8 bits, MCOMW 16 bits, or MCOML 32 bits); sets or clears flags
- **LCR** Load Complement from Register; IBM 360/370; RR format; fetches a full word (32 bits) of data from one of 16 general purpose registers, complements the data, and stores a full word (32 bits) of data in one of 16 general purpose registers; register to register only; sets or clears flags
- **LPR** Load Positive from Register (absolute value); IBM 360/370; RR format; fetches a full word (32 bits) of data from one of 16 general purpose registers, creates the absolute value (positive) the data, and stores a full word (32 bits) of data in one of 16 general purpose registers; register to register only; sets or clears flags
- **MNEG** Move Negated; DEC VAX; moves the arithmetic negative of a scalar quantity to register or memory (MNEGB 8 bits, MNEGW 16 bits, MNEGL 32 bits, MNEGQ 64 bits, MNEGF 32 bit float, or MNEGD 64 bit float); if source is positive zero, result is also positive zero; sets or clears flags
- **LNR** Load Negative from Register (negative of absolute value); IBM 360/370; RR format; fetches a full word (32 bits) of data from one of 16 general purpose registers, creates the absolute value the data, complements (negative) the absolute value of the data, and stores a full word (32 bits) of data in one of 16 general purpose registers; register to register only; sets or clears flags
- **LDAN** Load A-register Negative; MIX; move word or partial word field of data, load sign field with opposite sign; main storage to accumulator only
- **LDXN** Load X-register Negative; MIX; move word or partial word field of data, load sign field with opposite sign; main storage to extension register only
- **LD*i*N** Load index-register Negative; MIX; move word or partial word field of data, load sign field with opposite sign; main storage to one of five index registers only
- **STZ** Store Zero; MIX; move word or partial word field of data, store zero into designated word or field of word of memory
- **MVC** MoVe Character; IBM 360/370; SS format; moves one to 256 characters (8 bits each) of data; main storage to main storage only; does not affect condition code
- **MOVE** Move (block); MIX; move the number of words specified by the F field from location M to the location specified by the contents of index register 1, incrementing the index register on each word moved
- **MOVEM** Move Multiple; Motorola 680x0, Motorola 68300; move contents of a list of registers to memory or restore from memory to a list of registers; does not affect condition code
- **LM** Load Multiple; IBM 360/370; RS format; moves a series of full words (32 bits) of data from memory to a series of general purpose registers; main storage to register only; does not affect condition code
- **STM** STore Multiple; IBM 360/370; RS format; moves contents of a series of general purpose registers to a series of full words (32 bits) in memory; register to main storage only; does not affect condition code
- **PUSHA** Push All Registers; Intel 80x86; move contents all 16-bit general purpose registers to memory pointed to by stack pointer (in the order AX, CX, DX, BX, original SP, BP, SI, and DI); does not affect flags
- **PUSHAD** Push All Registers; Intel 80386; move contents all 32-bit general purpose registers to memory pointed to by stack pointer (in the order EAX, ECX, EDX, EBX, original ESP, EBP, ESI, and EDI); does not affect flags

- **POPA** Pop All Registers; Intel 80x86; move memory pointed to by stack pointer to all 16-bit general purpose registers (except for SP); does not affect flags
- **POPAD** Pop All Registers; Intel 80386; move memory pointed to by stack pointer to all 32-bit general purpose registers (except for ESP); does not affect flags
- **STJ** Store jump-register; MIX; move word or partial word field of data; jump register to main storage only
- **MOVEP** Move Peripheral Data; Motorola 680x0, Motorola 68300; moves data (16 bits or 32 bits) from a data register to memory mapped peripherals or moves data from memory mapped peripherals to a data register, skipping every other byte

address movement

Address movement instructions move addresses from one location to another. The source and destination locations are determined by the addressing modes, and can be registers or memory. Address movement instructions can come in a variety of sizes. Address movement instructions destroy the previous contents of the destination. Address movement instructions typically do not modify processor flags. When the destination is a register and the address is smaller than the full register size, the data might be placed only in the low order bits (leaving high order bits unchanged), or might be zero- or sign-extended to fill the entire register (some processors only use one choice, others permit the programmer to choose how this is handled).

- **MOVEA.W** Move Address (Word); Motorola 680x0, Motorola 68300; move an address word (16 bits) as sign-extended data (32 bits); memory to address register or register to address register; does not modify flags
- **MOVEA.L** Move Address (Longword); Motorola 680x0, Motorola 68300; move an address longword (32 bits); memory to address register or register to address register; does not modify flags
- **LEA** Load Effective Address; Motorola 680x0, Motorola 68300; computes an effective address and loads the result into an address register
- **LA** Load Address; RX format; IBM 360/370; computes an effective address and loads the 24-bit result (zero extended to 32-bits) into a general purpose register; does not affect condition code
- **ENTA** Enter A-register; MIX; move word or partial word field contents of index register to A-register (accumulator)
- **ENTX** Enter X-register; MIX; move word or partial word field contents of index register to X-register (extension)
- **ENT*i*** Enter I-register; MIX; move word or partial word field contents of index register to designated index register
- **ENNA** Enter Negative A-register; MIX; move word or partial word field contents of index register to A-register (accumulator), opposite sign loaded
- **ENNX** Enter Negative X-register; MIX; move word or partial word field contents of index register to X-register (extension), opposite sign loaded
- **ENN*i*** Enter Negative I-register; MIX; move word or partial word field contents of index register to designated index register, opposite sign loaded
- **INCA** Increase A-register; MIX; add word or partial word field contents of memory to A-register (accumulator), overflow toggle possibly set
- **INCX** Increase X-register; MIX; add word or partial word field contents of memory to X-register (extension), overflow toggle possibly set
- **INC*i*** Increase I-register; MIX; add word or partial word field contents of memory to designated index register, overflow toggle possibly set
- **DECA** Decrease A-register; MIX; subtract word or partial word field contents of memory from A-register (accumulator), overflow toggle possibly set
- **DECX** Decrease X-register; MIX; subtract word or partial word field contents of memory from X-register (extension), overflow toggle possibly set
- **DEC*i*** Decrease I-register; MIX; subtract word or partial word field contents of memory from

designated index register, overflow toggle possibly set
- **PEA** Push Effective Address; Motorola 680x0, Motorola 68300; computes an effective address and pushes the result onto a stack (predecrementing an address register acting as a stack pointer)
- **LINK** Link Stack; Motorola 680x0, Motorola 68300
- **UNLK** Unlink Stack; Motorola 680x0, Motorola 68300

basic assignment

The assignment statement transfers data from a source to a destination. The destination is usually a variable. In most languages the source can be a constant, variable, function, or expression.

Ada

```
        target := source;
```

Ada distinguishes between an assignment and an assignment statement.

In Ada assignment is indicated by the character pair colon and equal `:=` and an assignment statement is terminated with a semicolon `;`.

The EBNF definition of an Ada assignment statement is:

assignment_statement ::= *variable*_name := expression;

Only one variable may be on the left side of an assignment statement. The value assigned must be the same type as the variable and within the legal range for that variable.

binary integer arithmetic

This chapter examines integer arithmetic instructions in assembly language. Specific examples of instructions from various processors are used to illustrate the general nature of assembly language.

integer arithmetic

See also binary integer data representations.

For most processors, integer arithmetic is faster than floating point arithmetic. This can be reversed in special cases such digital signal processors.

The basic four integer arithmetic operations are **addition**, **subtraction**, **multiplication**, and **division**. Arithmetic operations can be **signed** or **unsigned** (unsigned is useful for effective address computations). Some older processors don't include hardware multiplication and division. Some processors don't include actual multiplication or division hardware, instead looking up the answer in a massive table of results embedded in the processor.

A specialized, but common, form of addition is an **increment** instruction, which adds one to the contents of a register or memory location. For address computations, "increment" may mean the addition of a constant other than one. Some processors have "short" or "quick" addition instructions that extend increment to include a small range of positive values.

A specialized, but common, form of subtraction is an **decrement** instruction, which subtracts one from the contents of a register or memory location. For address computations, "decrement" may mean the subtraction of a constant other than one. Some processors have "short" or "quick" subtraction instructions that extend decrement to include a small range of values.

Compare instructions are used to examine one or more integers non-destructively. These are usually implemented by performing a subtraction in some shadow register or accumulator and then setting flags accordingly. Compare instructions can compare two integers, or can compare a single integer to zero. Triadic compare instructions compare a test value to an upper and lower limit, which can be useful for bounds and range checking.

Some processors have specific hardware support for large multi-byte integer arithmetic. Even if there is no specific support, generally carry and borrow flags can be used to implement software multi-byte arithmetic routines.

Some processors have other special integer arithmetic operations. A **clear** instruction sets a register or memory location to zero. Some processors have special instructions for setting a register to a special value (such as pi) with additional guard bits also being set appropriately. A sign **extend** operation takes a small value and sign extends it to a larger storage format (such as byte to word). An arithmetic **complement** gives the arithmetic complement of a number (one's complement). An arithmetic **negate** gives the arithmetic inverse of a number (subtract from zero; two's complement).

- **ADD** Arithmetic Addition; DEC VAX; signed addition of scalar quantities (8, 16, or 32 bit integer or 32 or 64 bit floating point) in general purpose registers or memory, available in two operand (first operand added to second operand with result replacing second operand) and three operand (first operand added to second operand with result placed in third operand) (ADDB2 add byte 2 operand, ADDB3 add byte 3 operand, ADDW2 add word 2 operand, ADDW3 add word 3 operand, ADDL2 add long 2 operand, ADDL3 add long 3 operand); clears or sets flags
- **ADD** Add Integers; Intel 80x86; integer add of the contents of a register or memory (8, 16, or 32 bits) to a memory location or a register; sets or clear flags

- **ADD** Add; Motorola 680x0, Motorola 68300; signed add of the contents of a data register (8, 16, or 32 bits) to a memory location or adds the contents of a memory location (8, 16, or 32 bits) to a data register; sets or clear flags
- **ADD** Add; MIX; add word or partial word field contents of memory to A-register (accumulator), overflow toggle set if result is too large for A-register
- **AR** Add Register; IBM 360/370; RR format; signed add of the contents of a general purpose register (32 bits) to a general purpose register (32 bits); register to register only; sets or clears flags
- **A** Add; IBM 360/370; RX format; signed add of the contents of a memory location (32 bits) to a general purpose register (32 bits); main memory to register only; sets or clears flags
- **AH** Add Half-word; IBM 360/370; RX format; signed add of the contents of a memory location (16 bits) to a general purpose register (low order 16 bits); main memory to register only; sets or clears flags
- **ADDA** Add Address; Motorola 680x0, Motorola 68300; unsigned add of the contents of a memory location or register (16 or 32 bits) to an address register; does not modify flags
- **ADDI** Add Immediate; Motorola 680x0, Motorola 68300; signed add of immediate data (8, 16, or 32 bits) to a register or memory location; sets or clears flags
- **ADDQ** Add Quick; Motorola 680x0, Motorola 68300; signed add of an immediate value of 1 to 8 inclusive to a register or memory lcoation; sets or clears flags for data registers and memory locations, does not modify flags for an address register
- **INC** Increment; DEC VAX; increments the integer contents of a general purpose register or contents of memory (INCB byte, INCW word, INCL longword); equivalent to ADDx2 #1, sum, but shorter and executes faster; clears or sets flags
- **INC** Increment by 1; Intel 80x86; increments the contents of a register or memory (8, 16, or 32 bits); sets or clear flags (does not modify carry flag)
- **ADWC** Add With Carry; DEC VAX; integer addition (32 bit) in general purpose registers or memory, first operand added to second operand and the C (carry) flag with result replacing second operand; used for multiprecision arithmetic; clears or sets flags
- **ADC** Add Integers with Carry; Intel 80x86; integer add of the contents of a register or memory (8, 16, or 32 bits) and the carry flag to a memory location or a register, used to implement multi-precision integer arithmetic; sets or clear flags
- **ADDX** Add Extended; Motorola 680x0, Motorola 68300; (signed add of a data register [8, 16, or 32 bits] and the extend bit to a data register) or (signed add of the contents of memory location [8, 16, or 32 bits] and the extend bit to the contents of another memory location while predecrementing both the source and destination address pointer registers), used to implement multi-precision integer arithmetic; sets or clears flags
- **ADAWI** Add Aligned Word Interlocked; DEC VAX; adds (16 bit integer) a source operand from a register or memory to a memory location that is word aligned while interlocking the memory location so that no other processor or device can read or write to the interlocked memory location, used to maintain operating system resource usage counts; and sets or clears flags
- **SUB** Subtract; DEC VAX; signed subtraction of scalar quantities (8, 16, or 32 bit integer) in general purpose registers or memory, available in two operand (first operand subtracted from second operand with result replacing second operand) and three operand (first operand subtracted from second operand with result placed in third operand) (SUBB2 subtract byte 2 operand, SUBB3 subtract byte 3 operand, SUBW2 subtract word 2 operand, SUBW3 subtract word 3 operand, SUBL2 subtract long 2 operand, SUBL3 subtract long 3 operand); clears or sets flags
- **SUB** Subtract Integers; Intel 80x86; integer subtraction of the contents of a register or memory (8, 16, or 32 bits) from a memory location or a register; sets or clear flags
- **SUB** Subtract; Motorola 680x0, Motorola 68300; signed subtract of the contents of a data register (8, 16, or 32 bits) from a memory location or subtracts the contents of a memory location (8, 16, or 32 bits) from a data register; sets or clear flags
- **SUB** Subtract; MIX; subtract word or partial word field contents of memory from A-register (accumulator), overflow toggle possibly set
- **SR** Subtract Register; IBM 360/370; RR format; signed subtract of the contents of a general purpose register (32 bits) from a general purpose register (32 bits); register to register only; sets

or clears flags

- **S** Subtract; IBM 360/370; RX format; signed subtract of the contents of a memory location (32 bits) from a general purpose register (32 bits); main memory to register only; sets or clears flags
- **SH** Subtract Half-word; IBM 360/370; RX format; signed subtract of the contents of a memory location (16 bits) from a general purpose register (low order 16 bits); main memory to register only; sets or clears flags
- **SUBA** Subtract Address; Motorola 680x0, Motorola 68300; unsigned subtract of the contents of a memory location or register (16 or 32 bits) from an address register; does not modify flags
- **SUBI** Subtract Immediate; Motorola 680x0, Motorola 68300; signed subtract of immediate data (8, 16, or 32 bits) from a register or memory location; sets or clears flags
- **SUBQ** Subtract Quick; Motorola 680x0, Motorola 68300; signed subtract of an immediate value of 1 to 8 inclusive from a register or memory lcoation; sets or clears flags for data registers and memory locations, does not modify flags for an address register
- **DEC** Decrement; DEC VAX; decrements the integer contents of a general purpose register or contents of memory (DECB byte, DECW word, DECL longword); equivalent to SUBx2 #1, sum, but shorter and executes faster; clears or sets flags
- **DEC** Decrement by 1; Intel 80x86; decrements the contents of a register or memory (8, 16, or 32 bits); sets or clear flags (does not modify carry flag)
- **SBWC** Subtract With Carry; DEC VAX; integer subtraction (32 bit) in general purpose registers or memory, first operand and the C (carry) flag subtracted from second operand with result replacing second operand; used for extended precision subtraction; clears or sets flags
- **SBB** Subtract Integers with Borrow; Intel 80x86; integer subtraction of the contents of a register or memory (8, 16, or 32 bits) and carry flag from a memory location or a register; sets or clear flags
- **SUBX** Subtract Extended; Motorola 680x0, Motorola 68300; (signed subtract of a data register [8, 16, or 32 bits] and the extend bit from a data register) or (signed subtract of the contents of memory location [8, 16, or 32 bits] and the extend bit from the contents of another memory location while predecrementing both the source and destination address pointer registers), used to implement multi-precision integer arithmetic; sets or clears flags
- **MUL** Multiply; DEC VAX; signed multiplication of scalar quantities (8, 16, or 32 bit integer) in general purpose registers or memory, available in two operand (first operand multiplied by second operand with result replacing second operand) and three operand (first operand multiplied by second operand with result placed in third operand) (MULB2 multiply byte 2 operand, MULB3 multiply byte 3 operand, MULW2 multiply word 2 operand, MULW3 multiply word 3 operand, MULL2 multiply long 2 operand, MULL3 multiply long 3 operand); clears or sets flags
- **MULS.W** Signed Multiply; Motorola 680x0, Motorola 68300; signed multiplication of a word (16 bits) from memory or a register by a word (16 bits) in a data register with a longword (32 bit) result stored in the entire data register; sets or clears flags
- **MULS.L** Signed Multiply; Motorola 680x0, Motorola 68300; signed multiplication of a longword (32 bits) from memory or a register by a longword (32 bits) in a data register with a longword (32 bit) result stored in the data register (high order 32 bits of product are discarded); sets or clears flags
- **MULS.L <ea>,Dh:Dl** Signed Multiply; Motorola 680x0, Motorola 68300; signed multiplication of a longword (32 bits) from a data register by a longword (32 bits) in a data register with a quadword (64 bit) result stored in the data registers (high order 32 bits of product in first register, low order 32 bits of product in second data register); sets or clears flags
- **MULU.W** Unsigned Multiply; Motorola 680x0, Motorola 68300; unsigned multiplication of a word (16 bits) from memory or a register by a word (16 bits) in a data register with a longword (32 bit) result stored in the entire data register; sets or clears flags
- **MULU.L** Unsigned Multiply; Motorola 680x0, Motorola 68300; unsigned multiplication of a longword (32 bits) from memory or a register by a longword (32 bits) in a data register with a longword (32 bit) result stored in the data register (high order 32 bits of product are discarded); sets or clears flags
- **MULU.L <ea>,Dh:Dl** Unsigned Multiply; Motorola 680x0, Motorola 68300; unsigned multiplication of a longword (32 bits) from a data register by a longword (32 bits) in a data

register with a quadword (64 bit) result stored in the data registers (high order 32 bits of product in first register, low order 32 bits of product in second data register); sets or clears flags

- **MUL** Unsigned Multiplication of AL or AX; Intel 80x86; unsigned multiplication of a byte (8 bits) from register or memory by the contents of the AL register with a word (16-bit) result in AX register, or unsigned multiplication of a word (16 bits) from register or memory by the contents of the AX register with a doubleword (32-bit) result in DX:AX register pair, or unsigned multiplication of a doubleword (32 bits) from register or memory by the contents of the EAX register with a quadword (64-bit) result in EDX:EAX register pair; uses an early out algorithm to speed up computations when possible; sets or clears flags
- **IMUL** Signed Integer Multiply; Intel 80x86; signed multiplication of a byte (8 bits), word (16 bits), or doubleword (32 bits) from register or memory by the contents of the EAX and EDX registers with result stored in the EAX and EDX registers, signed multiplication of a byte (8 bits), word (16 bits), or doubleword (32 bits) from register or memory by the contents of a register with truncated results (to size as operands) stored in the register, or signed multiplication of a byte (8 bits), word (16 bits), or doubleword (32 bits) from register or memory by the contents of an immediate value with truncated results (to size as operands) stored in any general register; uses an early out algorithm to speed up computations when possible; sets or clears flags
- **MUL** Multiply; MIX; multiply word or partial word field contents of memory to A-register (accumulator) with results stored in X-register and A-register pair, overflow toggle possibly set
- **MR** Multiply Register; IBM 360/370; RR format; signed multiply of the contents of an even numbered general purpose register (32 bits) by the contents of the immediately following odd numbered general purpose register (32 bits) with a 64-bit product in the register pair; register to register only; does not affect condition code
- **M** Multiply; IBM 360/370; RX format; signed multiply of the contents of a memory location (32 bits) by the contents of an odd numbered general purpose register (32 bits) with a 64-bit product in the register pair; main storage to register only; does not affect condition code
- **MH** Multiply Half-word; IBM 360/370; RX format; signed multiply of the contents of a memory location (16 bits) by the contents of a general purpose register (32 bits) with a 32-bit product (the high 8 bits of the true 48-bit product are discarded) in the register; main storage to register only; does not affect condition code
- **EMUL** Extended Multiply; DEC VAX; extended precision multiplication on operands in registers or memory, the first (longword) operand (multiplicand) is multiplied by the second (longword) operand (multiplier) giving a (doubleword) intermediary result which is stored in the fourth (doubleword) operand (product); clears or sets flags
- **DIVS.W** Signed Divide; Motorola 680x0, Motorola 68300; signed division of a longword (32 bits) in memory or a register by a word (16 bits) in a data register with a result of the quotient (16 bits) in the lower word and the remainder (16 bits) in the upper word of the data register; clears or sets flags
- **DIVS.L <ea>,Dq** Signed Divide; Motorola 680x0, Motorola 68300; signed division of a longword (32 bits) in memory or a register by a longword (32 bits) in a data register with a result of a longword (32 bit) quotient in the data register and the remainder being discarded; clears or sets flags
- **DIVS.L <ea>,Dr:Dq** Signed Divide; Motorola 680x0, Motorola 68300; signed division of a quadword (64 bits) in any two data registers by a longword (32 bits) in a data register with a result of the quotient (32 bits) in the second data register and the remainder (32 bits) in the third data register; clears or sets flags
- **DIVSL.L <ea>,Dr:Dq** Signed Divide; Motorola 680x0, Motorola 68300; signed division of a longword (32 bits) in a data register by a longword (32 bits) in a second data register with a result of the quotient (32 bits) in the first data register and the remainder (32 bits) in the second data register; clears or sets flags
- **DIVU.W** Unsigned Divide; Motorola 680x0, Motorola 68300; unsigned division of a longword (32 bits) in memory or a register by a word (16 bits) in a data register with a result of the quotient (16 bits) in the lower word and the remainder (16 bits) in the upper word of the data register; clears or sets flags
- **DIVU.L <ea>,Dq** Unsigned Divide; Motorola 680x0, Motorola 68300; unsigned division of a

longword (32 bits) in memory or a register by a longword (32 bits) in a data register with a result of a longword (32 bit) quotient in the data register and the remainder being discarded; clears or sets flags

- **DIVU.L <ea>,Dr:Dq** Unsigned Divide; Motorola 680x0, Motorola 68300; unsigned division of a quadword (64 bits) in any two data registers by a longword (32 bits) in a data register with a result of the quotient (32 bits) in the second data register and the remainder (32 bits) in the third data register; clears or sets flags
- **DIVUL.L <ea>,Dr:Dq** Unsigned Divide; Motorola 680x0, Motorola 68300; unsigned division of a longword (32 bits) in a data register by a longword (32 bits) in a second data register with a result of the quotient (32 bits) in the first data register and the remainder (32 bits) in the second data register; clears or sets flags
- **DR** Divide Register; IBM 360/370; RR format; signed divide of the contents of a general purpose register pair (64 bits) by the contents of a general purpose register (32 bits) with a 32-bit quotient in the odd numbered register and a 32-bit remainder in the even numbered register of the register pair; register to register only; does not affect condition code
- **D** Divide; IBM 360/370; RX format; signed divide of the contents of a general purpose register pair (64 bits) by the contents of a memory location (32 bits) with a 32-bit quotient in the odd numbered register and a 32-bit remainder in the even numbered register of the register pair; main storage to register only; does not affect condition code
- **DIV** Divide; DEC VAX; arithmetic division of scalar quantities (8, 16, or 32 bit integer) in general purpose registers or memory, available in two operand (first operand [divisor] divided from second operand [dividend] with result [quotient] replacing second operand) and three operand (first operand [divisor] divided from second operand [dividend] with result placed in third operand [quotient]) (DIVB2 divide byte 2 operand, DIVB3 divide byte 3 operand, DIVW2 divide word 2 operand, DIVW3 divide word 3 operand, DIVL2 divide long 2 operand, DIVL3 divide long 3 operand); clears or sets flags
- **EDIV** Extended Divide; DEC VAX; extended precision multiplication on operands in registers or memory, the second (longword) operand (dividend) is divided from the first (longword) operand (divisor) giving the third (longword) operand (quotient) and the the fourth (longword) operand (remainder); clears or sets flags
- **DIV** Unsigned Divide; Intel 80x86; unsigned division of the accumulator by a byte (8 bits), word (16 bits), or doubleword (32 bits) divisor of half the size of the dividend in the accumulator, with the results stored in the accumulator (byte divisor: dividend is in the AX register, quotient in the AL register, and remainder in the AH register; word divisor: dividend is in the DX:AX register pair, quotient in the AX register, and remainder in the DX register; doubleword divisor: dividend is in the EDX:AEX register pair, quotient in the EAX register, and remainder in the EDX register); non-integral quotients are truncated to integers toward 0; sets or clears flags
- **IDIV** Signed Integer Division; Intel 80x86; Intel 80x86; signed division of the accumulator by a byte (8 bits), word (16 bits), or doubleword (32 bits) divisor of half the size of the dividend in the accumulator, with the results stored in the accumulator (byte divisor: dividend is in the AX register, quotient in the AL register, and remainder in the AH register; word divisor: dividend is in the DX:AX register pair, quotient in the AX register, and remainder in the DX register; doubleword divisor: dividend is in the EDX:AEX register pair, quotient in the EAX register, and remainder in the EDX register); non-integral quotients are truncated to integers toward 0; sets or clears flags
- **DIV** Divide; MIX; divide word or partial word field contents of memory from A-register (accumulator) and X-register (extension) pair with quotient stored in A-register and remainder stored in X-register, overflow toggle possibly set
- **CMP** Compare; DEC VAX; arithmetic comparison between two scalar quantities (8, 16, or 32 bit integer or 32 or 64 bit floating point) in general purpose registers or memory (CMPB Byte, CMPW Word, CMPL Longword); clears or sets flags
- **TST** Test; DEC VAX; arithmetic comparison of a scalar quantities (8, 16, or 32 bit integer or 32 or 64 bit floating point) in general purpose registers or memory (TSTB Byte, TSTW Word, TSTL Longword) to zero; equivalent to CMPs src, #0, but shorter and executes faster; clears or sets flags

- **CMP** Compare; Motorola 680x0, Motorola 68300; compares a register or contents of a memory location (8, 16, or 32 bits) to contents of a data register (data register minus effective address contents); clears or sets flags
- **CMP** Compare Two Operands; Intel 80x86; compares a register or contents of a memory location (8, 16, or 32 bits) to contents of a register or memory location (subtract of second operand from first operand with no storage of results, but setting or clearing of flags); clears or sets flags
- **CMPA** Compare A-register; MIX; compare word or partial word field contents of memory with same word or partial field of A-register (accumulator), set comparison indicator
- **CMPX** Compare X-register; MIX; compare word or partial word field contents of memory with same word or partial field of X-register (extension), set comparison indicator
- **CMP*i*** Compare I-register; MIX; compare word or partial word field contents of memory with same word or partial field of designated index register, set comparison indicator
- **CMPA** Compare Address; Motorola 680x0, Motorola 68300; compares a register or contents of a memory location (16 or 32 bits) to contents of an address register (adress register minus effective address contents); clears or sets flags
- **CMPI** Compare Immediate; Motorola 680x0, Motorola 68300; compares immediate data (8, 16, or 32 bits) to contents of a register or memory (effective address contents minus immediate data); clears or sets flags
- **CMPM** Compare Memory; Motorola 680x0, Motorola 68300; compares the contents of two memory locations (8, 16, or 32 bits) with a post increment of both address pointer registers (second location minus first location); clears of sets flags
- **CMP2** Compare Register Against Bounds; Motorola 680x0, Motorola 68300; compares the contents of register (8, 16, or 32 bits) to a bounds pair (lower bound followed by upper bound), if both bounds are equal then this operation tests for a specific value; sets or clears flags
- **CLR** Clear; Motorola 680x0, Motorola 68300; clears a register or contents of a memory location (.B 8, .W 16, or .L 32 bits) to zero; clears flags for memory and data registers, does not modify flags for address register
- **CLR** Clear; DEC VAX; clears a scalar quantity in register or memory to zero (CLRB 8 bits, CLRW 16 bits, CLRL 32 bits, CLRQ 64 bits, CLRO 128 bits, CLRF 32 bit float, or CLRD 64 bit float), an integer CLR will clear the same size floating point quantity because VAX floating point zero is represented as all zero bits; quadword and D float clears of registers are consecutive register pairs, octaword clears to registers are four consecutive registers; equivalent to MOVx #0, dst, but shorter and executes faster; sets or clears flags
- **STZ** Store Zero; MIX; move word or partial word field of data, store zero into designated word or field of word of memory
- **EXT** Sign Extend; Motorola 680x0, Motorola 68300; sign extends a byte (8 bits) in a data register to a word (16 bits) or sign extends a word (16 bits) in a data register to a longword (32 bits); sets or clears flags
- **EXTB** Sign Extend Byte; Motorola 680x0, Motorola 68300; sign extends a byte (8 bits) in a data register to a longword (32 bits); sets or clears flags
- **NEG** Two's Complement Negation; Intel 80x86; subtracts the contents of a register or memory (8, 16, or 32 bits) from zero and store the results in the original register or memory location (arithmetic negation or arithmetic inverse); sets or clears flags
- **NEG** Negate; Motorola 680x0, Motorola 68300; subtracts the contents of a register or memory (8, 16, or 32 bits) from zero and store the results in the original register or memory location (arithmetic negation or arithmetic inverse); sets or clears flags
- **NEGX** Negate with Extend; Motorola 680x0, Motorola 68300; subtracts the contents of a register or memory location (8, 16, or 32 bits) and the extend bit from zero and stores the results in the original register or memory location (multi-precision negation); sets or clears flags

floating point arithmetic

This chapter examines floating point arithmetic instructions in assembly language. Specific examples of instructions from various processors are used to illustrate the general nature of assembly language.

floating point arithmetic

See also floating point data representations.

For most processors, integer arithmetic is faster than floating point arithmetic. This can be reversed in special cases such digital signal processors.

On many processors, floating point arithmetic is in an optional unit or optional coprocessor rather than being included on the main processor. This allows the manufacturer to charge less for the business machines that don't need floating point arithmetic.

The basic four floating point arithmetic operations are **addition**, **subtraction**, **multiplication**, and **division**. Some processors don't include actual multiplication or division hardware, instead looking up the answer in a massive table of results embedded in the processor.

Compare instructions are used to examine one or more floating point numbers non-destructively. These are usually implemented by performing a subtraction in some shadow register or accumulator and then setting flags accordingly. Compare instructions can compare two floating point numbers, or can compare a single floating point number to zero.

- **ADD** Arithmetic Addition; DEC VAX; signed addition of scalar quantities (32, 64, or 128 bit floating point) in general purpose registers or memory, available in two operand (first operand added to second operand with result replacing second operand) and three operand (first operand added to second operand with result placed in third operand) (ADDF2 add float 2 operand, ADDF3 add float 3 operand, ADDD2 add double float 2 operand, ADDD3 add double float 3 operand, ADDG2 add G float 2 operand, ADDG3 add G float 3 operand, ADDH2 add H float 2 operand, ADDH3 add H float 3 operand); clears or sets flags
- **SUB** Subtract; DEC VAX; signed subtraction of scalar quantities (32, 64, or 128 bit floating point) in general purpose registers or memory, available in two operand (first operand subtracted from second operand with result replacing second operand) and three operand (first operand subtracted from second operand with result placed in third operand) (SUBF2 subtract float 2 operand, SUBF3 subtract float 3 operand, SUBD2 subtract double float 2 operand, SUBD3 subtract double float 3 operand, SUBG2 subtract G float 2 operand, SUBG3 subtract G float 3 operand, SUBH2 subtract H float 2 operand, SUBH3 subtract H float 3 operand); clears or sets flags
- **MUL** Multiply; DEC VAX; signed multiplication of scalar quantities (32, 64, or 128 bit floating point) in general purpose registers or memory, available in two operand (first operand multiplied by second operand with result replacing second operand) and three operand (first operand multiplied by second operand with result placed in third operand) (MULF2 multiply float 2 operand, MULF3 multiply float 3 operand, MULD2 multiply double float 2 operand, MULD3 multiply double float 3 operand, MULG2 multiply G float 2 operand, MULG3 multiply G float 3 operand, MULH2 multiply H float 2 operand, MULH3 multiply H float 3 operand); clears or sets flags
- **EMOD** Extended Multiply and Integerize; DEC VAX; performs accurate range reduction of math function arguments, the floating point multiplier extension operand (second operand) is concatenated with the floating point multiplier (first operand) to gain eight additional low order fraction bits, the multiplicand operand (third operand) is multiplied by the extended multiplier operand, after multiplication the integer portion (fourth operand) is extracted and a 32 bit

(EMODF) or 64 bit (EMODD) floating point number is formed from the fractional part of the product by truncating extra bits, the multiplication is such that the result is equivalent to the exact product truncated (before normalization) to a fraction field of 32 bits in floating or 64 bits in double (fifth operand); clears or sets flags

- **DIV** Divide; DEC VAX; arithmetic division of scalar quantities (32, 64, or 128 bit floating point) in general purpose registers or memory, available in two operand (first operand [divisor] divided from second operand [dividend] with result [quotient] replacing second operand) and three operand (first operand [divisor] divided from second operand [dividend] with result placed in third operand [quotient]) (DIVF2 divide float 2 operand, DIVF3 divide float 3 operand, DIVD2 divide double float 2 operand, DIVD3 divide double float 3 operand, DIVG2 divide G float 2 operand, DIVG3 divide G float 3 operand, DIVH2 divide H float 2 operand, DIVH3 divide H float 3 operand); clears or sets flags
- **POLY** Polynomial Evaluation; DEC VAX; performs fast calculation of math functions, for degree times (second operand) evaluate a function using Horner's method, where d=degree (second operand), x=argument (first operand), and result = C[0] + x*(C[1] + x*(C[2] + … x*C[d])), float result stored in D0 register, double float result stored in D0:D1 register pair, the table address operand (third operand) points to a table of polynomial coefficients ordered from highest order term of the polynomial through lower order coefficients stored at increasing addresses, the data type of the coefficients must be the same as the data type of the argument operand (first operand), the unsigned word degree operand (second operand) specifies the highest numbered coefficient to participate in the evaluation (POLYF polynomial evaluation floating, POLYD polynomial evaluation double float); D0 through D4 registers modified by POLYF, D0 through D5 registers modified by POLYD; sets or clears flags
- **CMP** Compare; DEC VAX; arithmetic comparison between two scalar quantities (8, 16, or 32 bit integer or 32 or 64 bit floating point) in general purpose registers or memory (CMPF Floating, CMPD Double Float); clears or sets flags
- **TST** Test; DEC VAX; arithmetic comparison of a scalar quantities (8, 16, or 32 bit integer or 32 or 64 bit floating point) in general purpose registers or memory (TSTF Floating, TSTD Double Float) to zero; equivalent to CMPs src, #0, but shorter and executes faster; clears or sets flags

binary coded decimal arithmetic

This chapter examines binary coded decimal (BCD) instructions in assembly language. Specific examples of instructions from various processors are used to illustrate the general nature of assembly language.

binary coded decimals

Binary coded decimal (BCD) is a method for implementing lossless decimal arithmetic (including decimal fractions) on a binary computer. The most obvious uses involve money amounts where round-off error from using binary approximations is unacceptable. Some early computers used BCD exclusively.

Decimal digits (0-9) can be encoded in a nibble (half a byte), with some left over bit patterns (hexadecimal A-F). In BCD operations, the processor performs ordinary binary computations, then adjusts the result to conform to BCD. For example, if you add the binary number 5 (bit pattern 0101) to binary number 6 (bit pattern 0110), you get the binary result of 11 (bit pattern 1011, or hexadecimal B). With BCD arithmetic, the processor would adjust the result to make it into a valid BCD result (which in this case would be bit pattern 0001 0001).

BCD arithmetic includes **BCD addition**, **BCD subtraction**, **BCD multiplication**, **BCD division**, and **BCD negate**.

The Intel 80x86 series uses a two step approach for BCD arithmetic. Instead of having separate BCD instructions, the normal binary addition and subtraction instructions are used, then hardware instructions are used to adjust the results to correct BCD results. There are instuctions for both packed and unpacked adjustments. The advantage of this approach is greater flexibility (more addressing modes and choices of arithmetic operations because of the use of regular binary integer instructions in the first step). The disadvantage of this approach is that it is slower and takes more memory.

Pack (Motorola 680x0) converts byte encoded numeric data (such as ASCII or EBCDIC characters) into binary coded decimals. **Unpack** (Motorola 680x0) converts binary coded decimals into byte encoded numeric data (such as ASCII or EBCDIC characters). The ASCII adjustment field is $3030; the EBCDIC adjustment field is $F0F0.

- **ABCD** Add Decimal with Extend; Motorola 680x0, Motorola 68300; performs binary coded decimal addition of source plus destination plus extend bit, source and destination can be two data registers or two locations in memory with the two address pointer registers being predecremented; sets or clears flags
- **SBCD** Subtract Decimal with Extend; Motorola 680x0; performs binary coded decimal subtraction of source and extend bit from destination, source and destination can be two data registers or two locations in memory with the two address pointer registers being predecremented; sets or clears flags
- **NBCD** Negate Decimal with Extend; Motorola 680x0, Motorola 68300; performs tens complement of contents of a data register or memory location by performing decimal coded binary subtraction of destination and extend bit from zero; sets or clears flags
- **DAA** Decimal Adjust after Addition; Intel 80x86; adjusts the result of adding two valid packed decimal operands in AL register; sets or clears flags
- **DAS** Decimal Adjust after Subtraction; Intel 80x86; adjusts the result of subtracting two valid packed decimal operands in AL register; sets or clears flags
- **AAA** ASCII Adjust after Addition; Intel 80x86; changes the contents of register AL to a valid unpacked decimal number, and zeros the top 4 bits; sets or clears flags
- **AAS** ASCII Adjust after Subtraction; Intel 80x86; changes the contents of register AL to a valid

unpacked decimal number, and zeros the top 4 bits; sets or clears flags

- **AAM** ASCII Adjust after Multiplication; Intel 80x86; corrects the result of a multiplication of two valid unpacked decimal numbers, the high order digit is left in AH, the low order digit in AL; sets or clears flags
- **AAD** ASCII Adjust before Division; Intel 80x86; modifies the numerator in AH and AL to prepare for the division of two valid unpacked decimal operands so that the quotient produced by the division will be a valid unpacked decimal number, AH should contain the high-order digit and AL the low-order digit, this instruction adjusts the value and places the result in AL, AH will contain zero.; sets or clears flags
- **PACK** Pack; Motorola 680x0; converts converts byte encoded numeric data (such as ASCII or EBCDIC characters) into binary coded decimals using an adjustment field ($3030 for ASCII, $F0F0 for EBCDIC), either from data register to data register or memory location to memory location with predecrement of address pointers; does not modify flags
- **UNPK** Unpack; Motorola 680x0; converts converts converts binary coded decimals into byte encoded numeric data (such as ASCII or EBCDIC characters) using an adjustment field ($3030 for ASCII, $F0F0 for EBCDIC), either from data register to data register or memory location to memory location with predecrement of address pointers; does not modify flags

advanced math operations

This chapter examines advanced mathematics instructions in assembly language. Specific examples of instructions from various processors are used to illustrate the general nature of assembly language.

advanced math operations

- **EMOD** Extended Multiply and Integerize; DEC VAX; performs accurate range reduction of math function arguments, the floating point multiplier extension operand (second operand) is concatenated with the floating point multiplier (first operand) to gain eight additional low order fraction bits, the multiplicand operand (third operand) is multiplied by the extended multiplier operand, after multiplication the integer portion (fourth operand) is extracted and a 32 bit (EMODF) or 64 bit (EMODD) floating point number is formed from the fractional part of the product by truncating extra bits, the multiplication is such that the result is equivalent to the exact product truncated (before normalization) to a fraction field of 32 bits in floating or 64 bits in double (fifth operand); clears or sets flags
- **TBLS** Table Lookup and Interpolate (Signed, Rounded); Motorola 68300; signed lookup and interpolation of independent variable X from a compressed linear data table or between two register-based table entries of linear representations of dependent variable Y as a function of X; $ENTRY_{(n)}$ + {($ENTRY_{(n+1)}$ - $ENTRY_{(n)}$) * Dx[7:0]} / 256 into Dx; *table version:* data register low word contains the independent variable X, 8-bit integer part and 8-bit fractional part with assumed radix point located between bits 7 and 8, source effective address points to beginning of table in memory, integer part scaled to data size (byte, word, or longword) and used as offset from beginning of table, selected table entry (a linear representation of dependent variable Y) is subtracted from the next consecutive table entry, then multiplied by the interpolation fraction, then divided by 256, then added to the first table entry, and then stored in the data register; *register version:* data register low byte contains the independent variable X 8-bit fractional part with assumed radix point located between bits 7 and 8, two data registers contain the byte, word, or longword table entries (a linear representation of dependent variable Y), first data register-based table entry is subtracted from the second data register-based table entry, then multiplied by the interpolation fraction, then divided by 256, then added to the first table entry, and then stored in the destination (X) data register, the register interpolation mode may be used with several table lookup and interpolations to model multidimentional functions; rounding is selected by the 'R' instruction field, for a rounding adjustment of -1, 0, or +1; sets or clears flags
- **TBLSN** Table Lookup and Interpolate (Signed, Not Rounded); Motorola 68300; signed lookup and interpolation of independent variable X from a compressed linear data table or between two register-based table entries of linear representations of dependent variable Y as a function of X; $ENTRY_{(n)}$ * 256 + ($ENTRY_{(n+1)}$ - $ENTRY_{(n)}$) * Dx[7:0] into Dx; *table version:* data register low word contains the independent variable X, 8-bit integer part and 8-bit fractional part with assumed radix point located between bits 7 and 8, source effective address points to beginning of table in memory, integer part scaled to data size (byte, word, or longword) and used as offset from beginning of table, selected table entry (a linear representation of dependent variable Y) multiplied by 256, then added to the value determined by (selected table entry subtracted from the next consecutive table entry, then multiplied by the interpolation fraction), and then stored in the data register; *register version:* data register low byte contains the independent variable X 8-bit fractional part with assumed radix point located between bits 7 and 8, two data registers contain the byte, word, or longword table entries (a linear representation of dependent variable Y), first data register-based table entry is multiplied by 256, then added to the value determined by (first data register-based table entry subtracted from the second data register-based table entry, then multiplied by the interpolation fraction), and then stored in the destination (X) data register, the register interpolation mode may be used with several table lookup and interpolations to model multidimentional functions; sets or clears flags

- **TBLU** Table Lookup and Interpolate (Unsigned, Rounded); Motorola 68300; unsigned lookup and interpolation of independent variable X from a compressed linear data table or between two register-based table entries of linear representations of dependent variable Y as a function of X; $\text{ENTRY}_{(n)} + \{(\text{ENTRY}_{(n+1)} - \text{ENTRY}_{(n)}) * \text{Dx}[7:0]\} / 256$ into Dx; *table version:* data register low word contains the independent variable X, 8-bit integer part and 8-bit fractional part with assumed radix point located between bits 7 and 8, source effective address points to beginning of table in memory, integer part scaled to data size (byte, word, or longword) and used as offset from beginning of table, selected table entry (a linear representation of dependent variable Y) is subtracted from the next consecutive table entry, then multiplied by the interpolation fraction, then divided by 256, then added to the first table entry, and then stored in the data register; *register version:* data register low byte contains the independent variable X 8-bit fractional part with assumed radix point located between bits 7 and 8, two data registers contain the byte, word, or longword table entries (a linear representation of dependent variable Y), first data register-based table entry is subtracted from the second data register-based table entry, then multiplied by the interpolation fraction, then divided by 256, then added to the first table entry, and then stored in the destination (X) data register, the register interpolation mode may be used with several table lookup and interpolations to model multidimentional functions; rounding is selected by the 'R' instruction field, for a rounding adjustment of 0 or +1; sets or clears flags
- **TBLUN** Table Lookup and Interpolate (Unsigned, Not Rounded); Motorola 68300; unsigned lookup and interpolation of independent variable X from a compressed linear data table or between two register-based table entries of linear representations of dependent variable Y as a function of X; $\text{ENTRY}_{(n)} * 256 + (\text{ENTRY}_{(n+1)} - \text{ENTRY}_{(n)}) * \text{Dx}[7:0]$ into Dx; *table version:* data register low word contains the independent variable X, 8-bit integer part and 8-bit fractional part with assumed radix point located between bits 7 and 8, source effective address points to beginning of table in memory, integer part scaled to data size (byte, word, or longword) and used as offset from beginning of table, selected table entry (a linear representation of dependent variable Y) multiplied by 256, then added to the value determined by (selected table entry subtracted from the next consecutive table entry, then multiplied by the interpolation fraction), and then stored in the data register; *register version:* data register low byte contains the independent variable X 8-bit fractional part with assumed radix point located between bits 7 and 8, two data registers contain the byte, word, or longword table entries (a linear representation of dependent variable Y), first data register-based table entry is multiplied by 256, then added to the value determined by (first data register-based table entry subtracted from the second data register-based table entry, then multiplied by the interpolation fraction), and then stored in the destination (X) data register, the register interpolation mode may be used with several table lookup and interpolations to model multidimentional functions; sets or clears flags
- **POLY** Polynomial Evaluation; DEC VAX; performs fast calculation of math functions, for degree times (second operand) evaluate a function using Horner's method, where d=degree (second operand), x=argument (first operand), and result = C[0] + x*(C[1] + x*(C[2] + … x*C[d])), float result stored in D0 register, double float result stored in D0:D1 register pair, the table address operand (third operand) points to a table of polynomial coefficients ordered from highest order term of the polynomial through lower order coefficients stored at increasing addresses, the data type of the coefficients must be the same as the data type of the argument operand (first operand), the unsigned word degree operand (second operand) specifies the highest numbered coefficient to participate in the evaluation (POLYF polynomial evaluation floating, POLYD polynomial evaluation double float); D0 through D4 registers modified by POLYF, D0 through D5 registers modified by POLYD; sets or clears flags

data conversion

This chapter examines data conversion instructions in assembly language. Specific examples of instructions from various processors are used to illustrate the general nature of assembly language.

data conversion

Data conversion instructions change data from one format to another.

A **sign extension** operation takes a small value and sign extends it to a larger storage format (such as byte to word).

A **type conversion** operation changes data from one format to another (such as signed two's complement integer into binary coded decimal).

- **EXT** Sign Extend; Motorola 680x0, Motorola 68300; sign extends a byte (8 bits) in a data register to a word (16 bits) or sign extends a word (16 bits) in a data register to a longword (32 bits); sets or clears flags
- **CVT** Convert; DEC VAX; converts a signed quantity to a different signed data type, source and destination in register or memory, special rounded versions for certain floating conversions; sets or clears flags
  - **CVTBW** Convert Byte to Word; sign extend
  - **CVTBL** Convert Byte to Long; sign extend
  - **CVTWB** Convert Word to Byte; truncated
  - **CVTWL** Convert Word to Long; sign extend
  - **CVTLB** Convert Long to Byte; truncated
  - **CVTLW** Convert Long to Word; truncated
  - **CVTBF** Convert Byte to Floating; exact
  - **CVTBD** Convert Byte to Double float; exact
  - **CVTWF** Convert Word to Floating; exact
  - **CVTWD** Convert Word to Double float; exact
  - **CVTLF** Convert Long to Floating; rounded
  - **CVTLD** Convert Long to Double float; exact
  - **CVTFB** Convert Floating to Byte; truncated
  - **CVTDB** Convert Double float to Byte; truncated
  - **CVTFW** Convert Floating to Word; truncated
  - **CVTDW** Convert Double float to Word; truncated
  - **CVTFL** Convert Floating to Long; truncated
  - **CVTRFL** Convert Rounded Floating to Long; rounded
  - **CVTDL** Convert Double float to Long; truncated
  - **CVTRDL** Convert Rounded Double float to Long; rounded
  - **CVTFD** Convert Floating to Double float; exact
  - **CVTDF** Convert Double float to Floating; rounded
- **CBW** Convert Byte to Word; Intel 80x86; sign extends a byte (8 bits) in register AL to create a word (16 bits) in the AX register; does not affect flags
- **CWD** Convert Word to Doubleword; Intel 80x86; sign extends a word (16 bits) in register AX throughout the DX register to create a doubleword (32 bits); does not affect flags
- **CWDE** Convert Word to Doubleword Extended; Intel 80386; sign extends a word (16 bits) in register AX to create a doubleword (32 bits) in the EAX register; does not affect flags
- **CDQ** Convert Doubleword to Quadword; Intel 80386; sign extends a doubleword (32 bits) in register EAX to create a quadword (64 bits) in the EDX register; does not affect flags
- **EXTB** Sign Extend Byte; Motorola 680x0, Motorola 68300; sign extends a byte (8 bits) in a data register to a longword (32 bits); sets or clears flags

- **MOVSX** Move with Sign Extension; Intel 80x86; moves data from a register or memory to a register, with a sign extension (conversion to larger binary integer: byte to word, byte to doubleword, or word to doubleword); does not affect flags
- **MOVZX** Move with Zero Extension; Intel 80x86; moves data from a register or memory to a register, with a zero extension (conversion to larger binary integer: byte to word, byte to doubleword, or word to doubleword); does not affect flags
- **MOVZ** Move Zero Extended; DEC VAX; converts an unsigned integer to a larger unsigned integer, source and destination in register or memory (MOVZBW Byte to Word, MOVZBL Byte to Long, MOVZWL Word to Long); sets or clears flags
- **NUM** Convert to Numeric; MIX; converts byte encoded character code (MIX character code) in A-register/X-register pair to numeric data in the A-register (accumulator), does not change sign, overflow possible
- **CHAR** Convert to Characters; MIX; converts numeric data in the A-register (accumulator) into byte encoded character code (MIX character code) in A-register/X-register pair, does not change signs
- **PACK** Pack; Motorola 680x0; converts byte encoded numeric data (such as ASCII or EBCDIC characters) into binary coded decimals using an adjustment field ($3030 for ASCII, $F0F0 for EBCDIC), either from data register to data register or memory location to memory location with predecrement of address pointers; does not modify flags
- **UNPK** Unpack; Motorola 680x0; converts binary coded decimals into byte encoded numeric data (such as ASCII or EBCDIC characters) using an adjustment field ($3030 for ASCII, $F0F0 for EBCDIC), either from data register to data register or memory location to memory location with predecrement of address pointers; does not modify flags

logical operations

This chapter examines logical instructions in assembly language. Specific examples of instructions from various processors are used to illustrate the general nature of assembly language.

logical

Logical instructions typically work on a bit by bit basis, although some processors use the entire contents of the operands as whole flags (zero or not zero input, zero or negative one output). Typical logical operations include logical negation or logical complement (NOT), logical and (AND), logical inclusive or (OR or IOR), and logical exclusive or (XOR or EOR). Logical tests are a comparison of a value to a bit string (or operand treated as a bit string) of all zeros. Some processors have an instruction that sets or clears a bit or byte in registers or memory based on the processor condition codes.

- **NOT** Logical Complement; Motorola 680x0, Motorola 68300; calculates the one’s complement (logical negation) of the contents of memory or a register (8, 16, or 32 bits); sets or clears flags
- **NOT** One's Complement Negation; Intel 80x86; calculates the one’s complement (logical negation) of the contents of memory or a register (8, 16, or 32 bits); does not modify flags
- **AND** And Logical; Motorola 680x0, Motorola 68300; performs a logical AND of a source operand with a destination operand and stores the result in the destination operand (8, 16, or 32 bits), one of the two operands must be a data register, the other operand can be the contents of any register or memory location; sets or clears flags
- **ANDI** And Immediate; Motorola 680x0, Motorola 68300; performs a logical AND of the contents of a register or memory location (8, 16, or 32 bits) with an immediate; sets or clears flags
- **AND** Logical AND; Intel 80x86; performs a logical AND between two registers, a register and contents of a memory location, or an immediate operand and either the contents of a register or the contents of a memory location; byte (8 bits), word (16 bits), or doubleword (32 bits); sets or clears flags
- **OR** Inclusive Or Logical; Motorola 680x0, Motorola 68300; performs a logical inclusive OR of a source operand with a destination operand and stores the result in the destination operand (8, 16, or 32 bits), one of the two operands must be a data register, the other operand can be the contents of any register or memory location; sets or clears flags
- **ORI** Inclusive Or Immediate; Motorola 680x0, Motorola 68300; performs a logical inclusive OR of the contents of a register or memory location (8, 16, or 32 bits) with an immediate; sets or clears flags
- **OR** Logical Inclusive OR; Intel 80x86; performs a logical OR between two registers, a register and contents of a memory location, or an immediate operand and either the contents of a register or the contents of a memory location; byte (8 bits), word (16 bits), or doubleword (32 bits); sets or clears flags
- **BIS** Bit Set; DEC VAX; performs a logical inclusive OR of the bit mask (first operand, register or memory) with the source (second operand, register or memory), available in two operand (results stored in second operand) and three operand (results stored in third operand) (BISB2 bit set byte 2 operand, BISB3 bit set byte 3 operand, BISW2 bit set word 2 operand, BISW3 bit set word 3 operand, BISL2 bit set longword 2 operand, BISL3 bit set longword 3 operand); sets or clears flags
- **XOR** Exclusive OR; DEC VAX; performs a logical exclusive OR of the bit mask (first operand, register or memory) with the source (second operand, register or memory), available in two operand (results stored in second operand) and three operand (results stored in third operand) (XORB2 exclusive or byte 2 operand, XORB3 exclusive or byte 3 operand, XORW2 exclusive or word 2 operand, XORW3 exclusive or word 3 operand, XORL2 exclusive or longword 2 operand, XORL3 exclusive or longword 3 operand); sets or clears flags
- **EOR** Exclusive Or Logical; Motorola 680x0, Motorola 68300; performs a logical exclusive or (XOR) of a source operand with a destination operand and stores the result in the destination

operand (8, 16, or 32 bits), one of the two operands must be a data register, the other operand can be the contents of any register or memory location; sets or clears flags

- **EORI** Exclusive Or Immediate; Motorola 680x0, Motorola 68300; performs a logical exclusive or (XOR) of the contents of a register or memory location (8, 16, or 32 bits) with an immediate; sets or clears flags
- **XOR** Logical Exclusive OR; Intel 80x86; performs a logical exclusive OR between two registers, a register and contents of a memory location, or an immediate operand and either the contents of a register or the contents of a memory location; byte (8 bits), word (16 bits), or doubleword (32 bits); sets or clears flags
- **BIC** Bit Clear; DEC VAX; performs a complimented logical AND of the bit mask (first operand, register or memory) with the source (second operand, register or memory), available in two operand (results stored in second operand) and three operand (results stored in third operand) (BICB2 bit clear byte 2 operand, BICB3 bit clear byte 3 operand, BICW2 bit clear word 2 operand, BICW3 bit clear word 3 operand, BICL2 bit clear longword 2 operand, BICL3 bit clear longword 3 operand); sets or clears flags
- **TST** Test an Operand; Motorola 680x0, Motorola 68300; compares the contents of a register or memory location (8, 16, or 32 bits) with zero; sets or clears flags
- **TEST** Logical Compare; Intel 80x86; compares the contents of a register or memory location (8, 16, or 32 bits) with an immediate value or the contents of a register; sets or clears flags
- **BIT** Bit Test; DEC VAX; performs a logical AND of the bit mask (first operand, register or memory) with the source (second operand, register or memory) without modifying either operand and tests the resulting bits for being all zero (BITB byte, BITW word, BITL longword); sets or clears flags
- **Scc** Set According to Condition; Motorola 680x0, Motorola 68300; tests a condition code, if the condition is true then sets a byte (8 bits) of a data register or memory location to TRUE (all ones), if the condition is false then sets a byte (8 bits) of a data register or memory location to FALSE (all zeros): SCC, SCS, SEQ, SF, SGE, SGT, SHI, SLE, SLS, SLT, SMI, SNE, SPL, ST, SVC, SVS
- **SETcc** Set Byte on Condition cc; Intel 80x86; tests a condition code, if the condition is true then sets a byte (8 bits) of a data register or memory location to TRUE (all ones), if the condition is false then sets a byte (8 bits) of a data register or memory location to FALSE (all zeros): SETA, SETAE, SETB, SETBE, SETC, SETE, SETG, SETGE, SETL, SETLE, SETNA, SETNAE, SETNB, SETNBE, SETNC, SETNE, SETNG, SETNGE, SETNL, SETNLE, SETNO, SETNP, SETNS, SETNZ, SETO, SETP, SETPE, SETPO, SETS, SETZ

shift and rotate instructions

This chapter examines shift and rotate instructions in assembly language. Specific examples of instructions from various processors are used to illustrate the general nature of assembly language.

shift and rotate

Shift and rotate instructions move bit strings (or operand treated as a bit string).

Shift instructions move a bit string (or operand treated as a bit string) to the **right** or **left**, with excess bits discarded (although one or more bits might be preserved in flags). In **arithmetic shift left** or **logical shift left** zeros are shifted into the low-order bit. In **arithmetic shift right** the sign bit (most significant bit) is shifted into the high-order bit. In **logical shift right** zeros are shifted into the high-order bit.

Rotate instructions are similar to shift instructions, ecept that rotate instructions are circular, with the bits shifted out one end returning on the other end. Rotates can be to the left or right. Rotates can also employ an extend bit for multi-precision rotates.

A **swap** instruction swaps the high and low order portions of a register or contents of a series of memory locations.

The carry bit typically receives the last bit shifted out of the operand. Sometimes an extend bit will receive the last bit shifted out also. Somtimes an overflow bit is used to indicate a sign change has occurred.

- **ASH** Arithmetic Shift; DEC VAX; performs a bit shift on a longword or quadword, the first operand is a byte count, the second operand is the source longword or quadword in registers or memory, the third operand is the destination longword or quadword in registers or memory, positive counts causes a left shift with zeros entering in the least significant bits, negative count causes a right shift with the most significant bit being copied into the most significant bit, a zero count results in the destination being replaced by the unmodified source (ASHL arithmetic shift longword, ASHQ arithmetic shift quadword); sets or clears flags
- **ASL** Arithmetic Shift Left; Motorola 680x0, Motorola 68300; shifts the contents of a data register (8, 16, or 32 bits) or memory location (16 bits) to the left (towards most significant bit) by a specified amount (by 1 to 8 bits for an immediate operation on a data register, by the contents of a data register modulo 64 for a data register, or by 1 bit only for a memory location), with the high-order bit being shifted into the carry and extend flags, zeros shifted into the low-order bit, overflow flag indicating a change of sign; sets or clear flags
- **SAL** Shift Arithmetic Left; Intel 80x86; shifts the contents of a data register or memory location (8, 16, or 32 bits) to the left (towards most significant bit) by a specified amount (by 1, by 0 to 31 bits specified by an immediate operand, or by 0-31 bits specified by the contents of the CL register), with the high-order bit being shifted into the carry flag, zeros shifted into the low-order bit; sets or clear flags
- **ASR** Arithmetic Shift Right; Motorola 680x0, Motorola 68300; shifts the contents of a data register (8, 16, or 32 bits) or memory location (16 bits) to the right (towards the least significant bit) by a specified amount (by 1 to 8 bits for an immediate operation on a data register, by the contents of a data register modulo 64 for a data register, or by 1 bit only for a memory location), with the low-order bit being shifted into the carry and extend flags, the original high-order bit being replicated and shifted into the high-order bit; sets or clear flags
- **SAR** Shift Arithmetic Right; Intel 80x86; shifts the contents of a data register or memory location (8, 16, or 32 bits) to the right (towards least significant bit) by a specified amount (by 1, by 0 to 31 bits specified by an immediate operand, or by 0-31 bits specified by the contents of the CL register), with the low-order bit being shifted into the carry flag, the original high-order bit being

replicated and shifted into the high-order bit; sets or clear flags

- **SLA** Shift Left A-register; MIX; *byte* shift the contents of the A-register (leaving sign unchanged) to the left by the designated number of bytes, with zeros shifted in to low order bytes
- **SLAX** Shift Left AX-register; MIX; *byte* shift the contents of the A-register and X-register pair (leaving signs unchanged) to the left by the designated number of bytes, with zeros shifted in to low order bytes
- **LSL** Logical Shift Left; Motorola 680x0, Motorola 68300; shifts the contents of a data register (8, 16, or 32 bits) or memory location (16 bits) to the left (towards most significant bit) by a specified amount (by 1 to 8 bits for an immediate operation on a data register, by the contents of a data register modulo 64 for a data register, or by 1 bit only for a memory location), with the high-order bit being shifted into the carry and extend flags, zeros shifted into the low-order bit; sets or clear flags
- **SHL** Shift Logical Left; Intel 80x86; shifts the contents of a data register or memory location (8, 16, or 32 bits) to the left (towards most significant bit) by a specified amount (by 1, by 0 to 31 bits specified by an immediate operand, or by 0-31 bits specified by the contents of the CL register), with the high-order bit being shifted into the carry flag, zeros shifted into the low-order bit; sets or clear flags
- **SRA** Shift Right A-register; MIX; *byte* shift the contents of the A-register (leaving sign unchanged) to the right by the designated number of bytes, with zeros shifted in to high order bytes
- **SRAX** Shift Right AX-register; MIX; *byte* shift the contents of the A-register and X-register pair (leaving signs unchanged) to the right by the designated number of bytes, with zeros shifted in to high order bytes
- **LSR** Logical Shift Right; Motorola 680x0, Motorola 68300; shifts the contents of a data register (8, 16, or 32 bits) or memory location (16 bits) to the right (towards the least significant bit) by a specified amount (by 1 to 8 bits for an immediate operation on a data register, by the contents of a data register modulo 64 for a data register, or by 1 bit only for a memory location), with the low-order bit being shifted into the carry and extend flags, and zeros shifted into the high-order bit; sets or clear flags
- **SHR** Shift Logical Right; Intel 80x86; shifts the contents of a data register or memory location (8, 16, or 32 bits) to the right (towards least significant bit) by a specified amount (by 1, by 0 to 31 bits specified by an immediate operand, or by 0-31 bits specified by the contents of the CL register), with the low-order bit being shifted into the carry flag, and zeros shifted into the high-order bit; sets or clear flags
- **SHLD** Double Precision Shift Left; Intel 80x86; shifts the contents of a general purpose register (16 or 32 bits) to the left (towards most significant bit) by a specified amount (by 0 to 31 bits specified by an immediate operand or by 0-31 bits specified by the contents of the CL register) with the high-order bits being shifted into a general purpose register or memory location (the source register is unchanged); used to implement multiprecision shifts, “bit blts” (BIT BLock Transfers), or bit string extracts and inserts; sets or clear flags
- **SHRD** Double Precision Shift Right; Intel 80x86; shifts the contents of a general purpose register (16 or 32 bits) to the right (towards least significant bit) by a specified amount (by 0 to 31 bits specified by an immediate operand or by 0-31 bits specified by the contents of the CL register) with the low-order bits being shifted into a general purpose register or memory location (the source register is unchanged); used to implement multiprecision shifts, “bit blts” (BIT BLock Transfers), or bit string extracts and inserts; sets or clear flags
- **ROTL** Rotate Long; DEC VAX; performs a bit rotate on a longword, the first operand is a byte count, the second operand is the source longword in registers or memory, the third operand is the destination longword in registers or memory, positive counts causes a left rotate, negative count causes a right rotate, a zero count results in the destination being replaced by the unmodified source, bits shifted out one end are rotated back in the other end; sets or clears flags
- **ROL** Rotate Left; Motorola 680x0, Motorola 68300; rotates the contents of a data register (8, 16, or 32 bits) or a memory location (16 bits) to the left (towards the most significant bit) by a specified amount (by 1 to 8 bits for an immediate operation on a data register, by the contents of a data register modulo 64 for a data register, or by 1 bit only for a memory location), with the

high-order bit rotating into both the carry flag and the low-order bit; sets or clear flags

- **ROL** Rotate Left; Intel 80x86; rotates the contents of a general purpose register or a memory location (8, 16, or 32 bits) to the left (towards the most significant bit) by a specified amount (by 1 bit or by 0 to 31 bits specified by an immediate operand or by the contents of the CL register); sets or clear flags
- **SLC** Shift Left AX-register Circularly; MIX; *byte* shift the contents of the A-register and X-register pair (leaving signs unchanged) to the left by the designated number of bytes, with bytes shifted off low order end entering on high order end
- **ROR** Rotate Right; Motorola 680x0, Motorola 68300; rotates the contents of a data register (8, 16, or 32 bits) or a memory location (16 bits) to the right (towards the least significant bit) by a specified amount (by 1 to 8 bits for an immediate operation on a data register, by the contents of a data register modulo 64 for a data register, or by 1 bit only for a memory location), with the low-order bit rotating into both the carry flag and the high-order bit; sets or clear flags
- **ROR** Rotate Right; Intel 80x86; rotates the contents of a general purpose register or a memory location (8, 16, or 32 bits) to the right (towards the least significant bit) by a specified amount (by 1 bit or by 0 to 31 bits specified by an immediate operand or by the contents of the CL register); sets or clear flags
- **SRC** Shift Right AX-register Circularly; MIX; *byte* shift the contents of the A-register and X-register pair (leaving signs unchanged) to the right by the designated number of bytes, with bytes shifted off high order end entering on low order end
- **ROXL** Rotate Left with Extend; Motorola 680x0, Motorola 68300; rotates the contents of a data register (8, 16, or 32 bits) or a memory location (16 bits) to the left (towards the most significant bit) through the extend bit by a specified amount (by 1 to 8 bits for an immediate operation on a data register, by the contents of a data register modulo 64 for a data register, or by 1 bit only for a memory location), with the high-order bit rotating into both the carry flag and the extend bit and the extend bit rotating into the low-order bit; sets or clear flags
- **RCL** Rotate Through Carry Left; Intel 80x86; rotates the contents of a general purpose register or a memory location (8, 16, or 32 bits) to the left (towards the most significant bit) by a specified amount (by 1 bit or by 0 to 31 bits specified by an immediate operand or by the contents of the CL register) with the carry flag (CF) being treated as a high-order one-bit extension of the destination operand; sets or clear flags
- **ROXR** Rotate Right with Extend; Motorola 680x0, Motorola 68300; rotates the contents of a data register (8, 16, or 32 bits) or a memory location (16 bits) to the right (towards the least significant bit) through the extend bit by a specified amount (by 1 to 8 bits for an immediate operation on a data register, by the contents of a data register modulo 64 for a data register, or by 1 bit only for a memory location), with the low-order bit rotating into both the carry flag and the extend bit and the extend bit rotating into the high-order bit; sets or clear flags
- **RCR** Rotate Through Carry Right; Intel 80x86; rotates the contents of a general purpose register or a memory location (8, 16, or 32 bits) to the right (towards the least significant bit) by a specified amount (by 1 bit or by 0 to 31 bits specified by an immediate operand or by the contents of the CL register) with the carry flag (CF) being treated as as a low-order one-bit extension of the destination operand; sets or clear flags
- **SWAP** Swap; Motorola 680x0, Motorola 68300; exchanges the high order word (16 bits) with the low order word (16 bits) of a data register; sets or clears flags

bit and bit field manipulation

This chapter examines bit and bit string instructions in assembly language. Specific examples of instructions from various processors are used to illustrate the general nature of assembly language.

- bit manipulation
- bit field

bit manipulation

Bit manipulation instructions manipulate a specific bit of a bit string (or operand treated as a bit string). Bit **clear** changes the specified bit to zero. Bit **set** changes the specified bit to one. Bit **change** modifies a specified bit, clearing a one bit to zero and setting a zero bit to one. In some processors, the value of the bit before modification is tested. Bit **test** examines the value of a specified bit.

Bit scan instructions search a bit string for the first bit that is set or cleared (depending on the processor).

- **BTST** Bit Test; Motorola 680x0, Motorola 68300; tests the value of a specified bit in a register or memory location (8 bit memory operands, 32 bit register operands); sets or clears flags
- **BIT** Bit Test; Intel 80x86; tests the value of a specified bit in a general register or memory location; sets or clears flags
- **BCLR** Bit Test and Clear; Motorola 680x0, Motorola 68300; tests the value of a specified bit in a register or memory location (8 bit memory operands, 32 bit register operands), then clears the specified bit to zero; sets or clears flags
- **BTR** Bit Test and Reset; Intel 80x86; tests the value of a specified bit in a general register or memory location, then clears (resets) the specified bit to one; sets or clears flags
- **BSET** Bit Test and Set; Motorola 680x0, Motorola 68300; tests the value of a specified bit in a register or memory location (8 bit memory operands, 32 bit register operands), then sets the specified bit to one; sets or clears flags
- **BTS** Bit Test and Set; Intel 80x86; tests the value of a specified bit in a general register or memory location, then sets the specified bit to one; sets or clears flags
- **BCHG** Bit Test and Change; Motorola 680x0, Motorola 68300; tests the value of a specified bit in a register or memory location (8 bit memory operands, 32 bit register operands), then either clears a one bit to zero or sets a zero bit to one; sets or clears flags
- **BTC** Bit Test and Complement; Intel 80x86; tests the value of a specified bit in a general register or memory location, then complements; sets or clears flags
- **BSF** Bit Scan Forward; Intel 80x86; scans a word (16 bits) or doubleword (32 bits) in memory or a general register from low-order to high-order (starting from bit index zero) for a one-bit and store the index of the first set bit into a register (if no set bit is found, the value of the destination register is undefined); sets or clears flags
- **BSR** Bit Scan Reverse; Intel 80x86; scans a word (16 bits) or doubleword (32 bits) in memory or a general register from high-order to low-order (starting from bit index 15 of a word or index 31 of a doubleword) for a one-bit and store the index of the first set bit into a register (if no set bit is found, the value of the destination register is undefined); sets or clears flags

bit field

Bit field instructions make modifications to bit fields (or operands treated as bit fields). Bit field **insert** inserts a value into a bit field. Bit field **extract** extracts a signed or unsigned value from a bit field. Bit field **find first one** finds the first bit that is set (one) in a bit field. Bit field **test** evaluates a bit field and sets or clears flags. Bit field **test and set**evaluates a bit field and set or clear flags then sets the bit field. Bit field **test and clear**evaluates a bit field and set or clear flags then clears the bit field. Bit field **test and change**evaluates a bit field and set or clear flags then changes the bit field.

- **BFINS** Bit Field Insert; Motorola 680x0; inserts a bit field (1 to 32 bits) from a data register to a location in data register or memory located by field offset and field width; sets or clears flags
- **BFEXTU** Bit Field Extract Unsigned; Motorola 680x0; extracts an unsigned bit field (1 to 32 bits) in registers or memory located by field offset and field width and zero extends the field into a data register (32 bits); sets or clears flags
- **BFEXTS** Bit Field Extract Signed; Motorola 680x0; extracts a signed bit field (1 to 32 bits) in registers or memory located by field offset and field width and sign extends the field into a data register (32 bits); sets or clears flags
- **BFFFO** Bit Field Find First One; Motorola 680x0; searches a bit field (1 to 32 bits) in registers or memory located by field offset and field width for the most significant bit that is set to a value of one; set or clears flags
- **BFTST** Bit Field Test; Motorola 680x0; sets condition codes according to the value of a bit field (1 to 32 bits) in registers or memory located by field offset and field width
- **BIT** Bit Test; DEC VAX; performs a logical AND of the bit mask (first operand, register or memory) with the source (second operand, register or memory) without modifying either operand and tests the resulting bits for being all zero (BITB byte, BITW word, BITL longword); sets or clears flags
- **BFSET** Test Bit Field and Set; Motorola 680x0; sets condition codes according to the value of a bit field (1 to 32 bits) in registers or memory located by field offset and field width, then sets the field; sets or clear flags
- **BIS** Bit Set; DEC VAX; performs a logical inclusive OR of the bit mask (first operand, register or memory) with the source (second operand, register or memory), available in two operand (results stored in second operand) and three operand (results stored in third operand) (BISB2 bit set byte 2 operand, BISB3 bit set byte 3 operand, BISW2 bit set word 2 operand, BISW3 bit set word 3 operand, BISL2 bit set longword 2 operand, BISL3 bit set longword 3 operand); sets or clears flags
- **BFCLR** Test Bit Field and Clear; Motorola 680x0; ; sets condition codes according to the value of a bit field (1 to 32 bits) in registers or memory located by field offset and field width, then clears the field; sets or clear flags
- **BIC** Bit Clear; DEC VAX; performs a complimented logical AND of the bit mask (first operand, register or memory) with the source (second operand, register or memory), available in two operand (results stored in second operand) and three operand (results stored in third operand) (BICB2 bit clear byte 2 operand, BICB3 bit clear byte 3 operand, BICW2 bit clear word 2 operand, BICW3 bit clear word 3 operand, BICL2 bit clear longword 2 operand, BICL3 bit clear longword 3 operand); sets or clears flags
- **BFCHG** Test Bit Field and Change; Motorola 680x0; sets condition codes according to the value of a bit field (1 to 32 bits) in registers or memory located by field offset and field width, then complements the field; sets or clear flags
- **SHLD** Double Precision Shift Left; Intel 80x86; shifts the contents of a general purpose register (16 or 32 bits) to the left (towards most significant bit) by a specified amount (by 0 to 31 bits specified by an immediate operand or by 0-31 bits specified by the contents of the CL register) with the high-order bits being shifted into a general purpose register or memory location (the source register is unchanged); used to implement multiprecision shifts, “bit blts” (BIT BLock Transfers), or bit string extracts and inserts; sets or clear flags
- **SHRD** Double Precision Shift Right; Intel 80x86; shifts the contents of a general purpose register (16 or 32 bits) to the right (towards least significant bit) by a specified amount (by 0 to 31 bits specified by an immediate operand or by 0-31 bits specified by the contents of the CL register) with the low-order bits being shifted into a general purpose register or memory location (the source register is unchanged); used to implement multiprecision shifts, “bit blts” (BIT BLock Transfers), or bit string extracts and inserts; sets or clear flags
- **BSF** Bit Scan Forward; Intel 80x86; scans a word (16 bit) or doubleword (32 bit) bit string for the first set (one) bit; scans from low-order to high-order (starting from bit index zero); sets or

clears flags

- **BSR** Bit Scan Reverse; Intel 80x86; scans a word (16 bit) or doubleword (32 bit) bit string for the first set (one) bit; scans from high-order to low-order (starting from bit index 15 of a word or index 31 of a doubleword); sets or clears flags

character and string operations

This chapter examines string and character instructions in assembly language. Specific examples of instructions from various processors are used to illustrate the general nature of assembly language.

string and character operations

- **MVC** MoVe Character; IBM 360/370; SS format; moves one to 256 characters (8 bits each) of data; main storage to main storage only; does not affect condition code

Character Codes

Summary: This chapter has charts of various common computer character codes.

ASCII American Standard Code for Information Interchange. Seven (7) bit code.

EBCDIC Extended Binary Coded Decimal Interchange Code (used primarily on IBM mainframes). Eight (8) bit code.

Punched Card Code used for Hollerith (punched) cards (punching positions given in decimal). Thirteen (13) bit code.

International Morse Code The code originally developed for telegraph (Morse Code) had an equal number of bits per character. International Morse Code has a variable number of bits per character, with characters that occur more frequently having a smaller number of bits. Bits are represented by unipolar current pulses of long and short duration (usually 3:1, with pauses used to separate characters).

International Cable Code International Cable Code is similar to International Morse Code, except that its bits are represented by bipolar current pulses of uniform duration (positive matching Morse's short pulse and negative matching Morse's long pulse). Also, there is typically a great deal of variation in punctuation codes.

HTML metacharacters HTML has both numeric codes and name codes (metacharacters) for producing printable characters. HTML name codes are case sensitive. Older browsers typically don't support very many name codes (that is, numeric codes are more likely to work in more browsers). Some of the new metacharacters only have name codes (no numeric code). Not all metacharacters work on all platforms or with all fonts (the widest support is on the Macintosh and Windows, which are each slightly different). The universally supported name codes are: &quot;, &num;, &amp;, &lt;, and &gt; (which are also essential as HTML escape metacharacters).

NOTE: With some browsers, you may have to wait until the entire page is loaded before the following links will work.

- control codes
- decimal digits
- Roman letters
- punctuation marks and special characters
- sorted by numeric value

Values are in hexadecimal or binary, except as otherwise noted

Note: Each table will not display until the entire table has been downloaded to your computer. Please be patient.

control codes

control code	meaning	punched card	EBCDIC	ASCII
ACK	acknowledge	0-6-8-9	2E	06
BEL	bell or alarm	0-7-8-9	2F	07
BS	backspace	11-6-9	16	08

BYP	bypass	0-4-9	24	
CAN	cancel	11-8-9	18	18
CC	cursor control	11-2-8-9	1A	
CR	carriage return	12-5-8-9	0D	0D
CU1	customer use 1	11-3-8-9	1B	
CU2	customer use 2	0-3-8-9	2B	
CU3	customer use 3	3-8-9	3B	
DC1	device control 1	11-1-9	11	11
DC2	device control 2	11-2-9	12	12
DC3	device control 3			13
DC4	device control 4	4-8-9	3C	14
DEL	delete	12-7-9	07	7F
DLE	data link escape	12-11-1-8-9	10	10
DS	digit select	11-0-1-8-9	20	
EM	end of medium	11-1-8-9	19	19
ENQ	enquiry	0-5-8-9	2D	05
EOT	end of transmission	7-9	37	04
ESC	escape	0-7-9	27	1B
ETB	end of transmission block	0-6-9	26	17
ETX	end of text	12-3-9	03	03
FF	form feed	12-4-8-9	0C	0C
FS	field separator	0-2-9	22	
FS	file separator	11-4-8-9	(1C)	1C
GS	group separator	11-5-8-9	(1D)	1D
HT	horizontal tabulation	12-5-9	05	09
IFS	interchange file separator	11-4-8-9	1C	(1C)
IGS	interchange group separator	11-5-8-9	1D	(1D)
IL	idle	11-7-9	17	
IRS	interchange record separator	11-6-8-9	1E	(1E)
IUS	interchange unit separator	11-7-8-9	1F	(1F)
LC	lower case	12-6-9	06	
LF	line feed	0-5-9	25	0A
NAK	negative acknowledge	5-8-9	3D	15
NL	new line	11-5-9	15	
NUL	null	12-0-1-8-9	00	00
PF	punch off	12-4-9	04	
PN	punch on	4-9	34	
RES	restore	11-4-9	14	
RS	reader stop	5-9	35	

RS	record separator	11-6-8-9	(1E)	1E
SI	shift in	12-7-8-9	0F	0F
SM	set mode	0-2-8-9	2A	
SMM	start of manual message	12-2-8-9	0A	
SO	shift out	12-6-8-9	0E	0E
SOH	start of heading	12-1-9	01	01
SOS	start of significance	0-1-9	21	
SP	space	no punches	40	20
STX	start of text	12-2-9	02	02
SUB	substitute	7-8-9	3F	1A
SYN	synchronous idle	2-9	32	16
TM	tape mark	11-3-9	13	
UC	upper case	6-9	36	
US	unit separator	11-7-8-9	(1F)	1F
VT	vertical tabulation	12-3-8-9	0B	0B

control codes for International Morse Code:

SOS	. . . _ _ _ . . .	Break	_ . . . _ . _
Attention	_ . _ . _	Understand	. . . _ .
CQ	_ . _ . _ _ . _	Error	. . . .
DE	_ . . .	OK	. _ .
Go Ahead	_ . _	End of Message	. _ . _ .
Wait	. _ . . .	End of Work	. . . _ . _

decimal digits

digit	Morse	punched card	BCD	EBCDIC	ASCII	HTML numeric
0	_ _ _ _ _	0	F0	F0	30	&#48;
1	. _ _ _ _	1	F1	F1	31	&#49;
2	. . _ _ _	2	F2	F2	32	&#50;
3	. . . _ _	3	F3	F3	33	&#51;
4	. . . . _	4	F4	F4	34	&#52;
5	. . . . .	5	F5	F5	35	&#53;
6	_	6	F6	F6	36	&#54;
7	_ _ . . .	7	F7	F7	37	&#55;
8	_ _ _ . .	8	F8	F8	38	&#56;
9	_ _ _ _ .	9	F9	F9	39	&#57;

Roman letters

letter	Morse	punched card	BCD	EBCDIC	ASCII	HTML numeric	HTML name	notes
A	. _	12-1	C1	C1	41	&#65;		
À						&#192;	&Agrave;	UPPERCASE A with accent grave
Á						&#193;	&Aacute;	UPPERCASE A with accent acute
Â						&#194;	&Acirc;	UPPERCASE A with accent circumflex
Ã						&#195;	&Atilde;	UPPERCASE A with tilde
Ä						&#196;	&Auml;	UPPERCASE A with dieresis (umlaut)
Å						&#197;	&Aring;	UPPERCASE A with ring
Æ						&#198;	&AElig;	UPPERCASE AE diphthong (ligature)
B	_ . . .	12-2	C2	C2	42	&#66;		
C	_ . _ .	12-3	C3	C3	43	&#67;		
Ç						&#199;	&Ccedil;	UPPERCASE C with cedilla
D	_ . .	12-4	C4	C4	44	&#68;		
&#208;						&#208;	&ETH;	UPPERCASE ETH
E	.	12-5	C5	C5	45	&#69;		
È						&#200;	&Egrave;	UPPERCASE E with accent grave
É						&#201;	&Eacute;	UPPERCASE E with accent acute
Ê						&#202;	&Ecirc;	UPPERCASE E with accent circumflex
Ë						&#203;	&Euml;	UPPERCASE E with umlaut (dieresis)
F	. . _ .	12-6	C6	C6	46	&#70;		
G	_ _ .	12-7	C7	C7	47	&#71;		
H	. . . .	12-8	C8	C8	48	&#72;		
I	. .	12-9	C9	C9	49	&#73;		
Ì						&#204;	&Igrave;	UPPERCASE I with accent grave
Í						&#205;	&Iacute;	UPPERCASE I with accent acute
Î						&#206;	&Icirc;	UPPERCASE I with accent circumflex
Ï						&#207;	&Iuml;	UPPERCASE I with umlaut (dieresis)
J	. _ _ _	11-1	D1	D1	4A	&#74;		

punctuation marks and special characters

† indicates a hexadecimal value shared by two different BCD characters.

punctuation	mark	punched card	BCD	EBCDIC	ASCII	HTML numeric	HTML name
accent acute	´					&#180;	&acute;
accent grave	`				60	&#96;	&grave;
ampersand	&	12	50†	50	26	&#38;	&amp;
angle quote mark, left	«					&#171;	&laquo;
angle quote mark, right	»					&#187;	&raquo;
apiece sign	@	4-8	7C†	7C	40	&#64;	
apostrophe (typewriter)	'	5-8		7D	27	&#39;	&apos;
apostrophe (typographical)	’					&#146;	
asterisk	*	11-4-8	5C	5C	2A	&#42;	&ast;
at sign	@	4-8	7C†	7C	40	&#64;	
back slash	\				5C	&#92;	&bsol;
bar, broken vertical	¦					&#166;	&brvbar;
bar, high	¯					&#175;	&macr;
bar, low	_	0-5-8		6D	5F	&#95;	&lowbar;
bar, vertical	|	12-7-8		4F	7C	&#124;	&verbar;
blank (space)		no punches	40	40	20	&#32;	
braces, curly, left	{				7B	&#123;	&lcub;
braces, curly, right	}				7D	&#125;	&rcub;
bracket, square, left	[	12-8-5	4D		5B	&#91;	&lsqb;
bracket, square, right	]	11-8-5	5D		5D	&#93;	&rsqb;
broken vertical bar	¦					&#166;	&brvbar;
bullet	•					&#149;	
caret	^				5E	&#94;	&circ;
cedilla	¸					&#184;	&cedil;
cent mark	¢	12-2-8		4A		&#162;	&cent;
circumflex	^				5E	&#94;	&circ;
colon	:	2-8	7D	7A	3A	&#58;	&colon;
comma	,	0-3-8	6B	6B	2C	&#44;	&comma;
copyright sign	©					&#169;	&copy;
curly braces, left	{				7B	&#123;	&lcub;
curly braces, right	}				7D	&#125;	&rcub;

currency sign, general	¤					&#164;	&curren;
dagger, single	†					&#134;	
dagger, double	‡					&#135;	
dash, em	—					&#151;	&mdash;
dash, en	–					&#150;	&ndash;
degree sign	°					&#176;	&deg;
division sign	÷					&#247;	&divide;
dollar sign	$	11-3-8	5B	5B	24	&#36;	&dollar;
dot	.	12-3-8	75	75	2E	&#46;	
double dagger	‡					&#135;	
double quotation mark (typewriter)	"	7-8		7F	22	&#34;	&quot;
double quote mark, right (typographical)	”					&#148;	
double quote mark, left (typographical)	“					&#147;	
dropped double quote	„					&#132;	
dropped single quote	‚					&#130;	
ellipses (three dots)	…					&#133;	&ldots;
em dash	—					&#151;	&mdash;
en dash	–					&#150;	&ndash;
equal sign	=	6-8	7B†	7E	3D	&#61;	
exclamation point	!	11-2-8		5A	21	&#33;	
exclamation point, inverted	¡					&#161;	&iexcl;
feminine ordinal indicator	ª					&#170;	&ordf;
foot mark	'	5-8	7D†	7D	27	&#39;	
forward slash	/	0-1	61	61	2F	&#47;	
fraction, one-half	$^1/_2$					&#189;	&frac12;
fraction, one-quarter	$^1/_4$					&#188;	&frac14;
fraction, three-quarters	$^3/_4$					&#190;	&frac34;
function sign	ƒ					&#131;	
general currency sign	¤					&#164;	&curren;
greater than sign	>	0-6-8	7E	6E	3E	&#62;	&gt;
half	$^1/_2$					&#189;	&frac12;
high bar	¯					&#175;	&macr;
hyphen	-	11	60	60	2D	&#45;	&hyphen;
hyphen, soft						&#173;	&shy;
inch mark	"	7-8		7F	22	&#34;	

inverted exclamation point	¡					&#161;	&iexcl;
inverted question mark	¿					&#191;	&iquest;
left angle quote mark	«					&#171;	&laquo;
left curly braces	{				7B	&#123;	&lcub;
left double quote mark (typographical)	“					&#147;	
left parenthesis	(	12-5-8	6C†	4D	28	&#40;	&lpar;
left single quote mark (typographical)	‘					&#145;	
left square bracket	[	12-8-5	4D		5B	&#91;	&lsqb;
left tag	<	12-4-8	4E	4C	3C	&#60;	&lt;
less than sign	<	12-4-8	4E	4C	3C	&#60;	&lt;
logical NOT	¬	11-7-8		5F		&#172;	&not;
logical OR	|	12-7-8		4F	7C	&#124;	&verbar;
low bar	_	0-5-8		6D	5F	&#95;	&lowbar;
macron	¯					&#175;	&macr;
masculine ordinal indicator	º					&#186;	&ordm;
micro sign	µ					&#181;	&micro;
middle dot	·					&#183;	&middot;
minus sign	-	11	60	60	2D	&#45;	
multiply sign	&#215;					&#215;	&times;
multiplication sign	·					&#183;	&middot;
non-breaking space						&#160;	&nbsp;
not sign	¬	11-7-8		5F		&#172;	&not;
number sign	&num;	3-8	7B†	7B	23	&#35;	&num;
ordinal indicator, feminine	ª					&#170;	&ordf;
ordinal indicator, masculine	º					&#186;	&ordm;
paragraph mark	¶					&#182;	&para;
parenthesis, left	(	12-5-8	6C†	4D	28	&#40;	&lpar;
parenthesis, right	)	11-5-8	4C†	5D	29	&#41;	&rpar;
percent	%	0-4-8	6C†	6F	25	&#37;	&percnt;
period	.	12-3-8	75	75	2E	&#46;	&period;
pilcrow	¶					&#182;	&para;
plus sign	+	12-6-8	50†	4E	2B	&#43;	
plus-or-minus sign	±					&#177;	&plusmn;
pound sign	&num;	3-8	7B†	7B	23	&#35;	&num;
pound sterling sign	£					&#163;	&pound;
prime	'	5-8		7D	27	&#39;	
quarter	$^1/_4$					&#188;	&frac14;

question mark	?	0-7-8	C0	6F	3F	&#63;	
question mark, inverted	¿					&#191;	&iquest;
quotation mark (typewriter)	"	7-8		7F	22	&#34;	&quot;
quote mark, left angle	«					&#171;	&laquo;
quote mark, left double (typographical)	“					&#147;	
quote mark, left single (typographical)	‘					&#145;	
quote mark, right angle	»					&#187;	&raquo;
quote mark, right double (typographical)	”					&#148;	
quote mark, right single (typographical)	’					&#146;	
registered trademark sign	®					&#174;	&reg;
right angle quote mark	»					&#187;	&raquo;
right curly braces	}				7D	&#125;	&rcub;
right double quote mark (typographical)	”					&#148;	
right parenthesis	)	11-5-8	4C†	5D	29	&#41;	&rpar;
right single quote mark (typographical)	’					&#146;	
right square bracket	]	11-8-5	5D		5D	&#93;	&rsqb;
right tag	>	0-6-8	7E	6E	3E	&#62;	&gt;
salinity	‰					&#137;	
section sign	§					&#167;	&sect;
semicolon	;	11-6-8	5E	5E	3B	&#59;	&semi;
single dagger	†					&#134;	
single quote mark (typewriter)	'	5-8	7D†	7D	27	&#39;	
single quote mark, left (typographical)	‘					&#145;	
single quote mark, right (typographical)	’					&#146;	
slash	/	0-1	61	61	2F	&#47;	
soft hyphen						&#173;	&shy;
solid bullet	•					&#149;	
space		no punches	40	40	20	&#32;	
space, non-breaking						&#160;	&nbsp;
spacing cedilla	¸					&#184;	&cedil;
spacing dieresis	¨					&#168;	
spacing macron	¯					&#175;	&macr;

square	◻	12-8-4	4C†				
square bracket, left	[	12-8-5	4D		5B	&#91;	&lsqb;
square bracket, right	]	11-8-5	5D		5D	&#93;	&rsqb;
squared	^{2}					&#178;	&sup2;
superscript 1	^{1}					&#185;	&sup1;
superscript 2	^{2}					&#178;	&sup2;
superscript 3	^{3}					&#179;	&sup3;
three-quarters	$^{3}/_{4}$					&#190;	&frac34;
tick mark	`				60	&#96;	
tilde	~				7E	&#126;	&tilde;
trademark sign	™					&#153;	&trade;
trademark, registered	®					&#174;	&reg;
tripled	^{3}					&#179;	&sup3;
typewriter double quotation mark	"	7-8		7F	22	&#34;	&quot;
typewriter single quotation mark	'	5-8	7D†	7D	27	&#39;	
typographical apostrophe	’					&#146;	
typographical left double quote mark	“					&#147;	
typographical left single quote mark	‘					&#145;	
typographical right double quote mark	”					&#146;	
typographical right single quote mark	”					&#146;	
umlaut	¨					&#168;	&uml;
underscore	_	0-5-8		6D	5F	&#95;	&lowbar;
vertical bar	|	12-7-8		4F	7C	&#124;	&verbar;
vertical bar, broken	¦					&#166;	&brvbar;
virgule	/	0-1	61	61	2F	&#47;	
yen sign	¥					&#165;	&yen;

† indicates a hexadecimal value shared by two different BCD characters.

sorted by numeric value

hex	binary	decimal	BCD	EBCDIC	ASCII	HTML
00	00000000	0		NUL	NUL	
01	00000001	1		SOH	SOH	

02	00000010	2		STX	STX	
03	00000011	3		ETX	ETX	
04	00000100	4		PF	EOT	
05	00000101	5		HT	ENQ	
06	00000110	6		LC	ACK	
07	00000111	7		DEL	BEL	
08	00001000	8			BS	
09	00001001	9			HT	
0A	00001010	10		SMM	LF	
0B	00001011	11		VT	VT	
0C	00001100	12		FF	FF	
0D	00001101	13		CR	CR	
0E	00001110	14		SO	SO	
0F	00001111	15		SI	SI	
hex	**binary**	**decimal**	**BCD**	**EBCDIC**	**ASCII**	**HTML**
10	00010000	16		DLE	DLE	
11	00010001	17		DC1	DC1	
12	00010010	18		DC2	DC2	
13	00010011	19		TM	DC3	
14	00010100	20		RES	DC4	
15	00010101	21		NL	NAK	
16	00010110	22		BS	SYN	
17	00010111	23		IL	ETB	
18	00011000	24		CAN	CAN	
19	00011001	25		EM	EM	
1A	00011010	26		CC	SUB	
1B	00011011	27		CU1	ESC	
1C	00011100	28		IFS	FS	
1D	00011101	29		IGS	GS	
1E	00011110	30		IRS	RS	
1F	00011111	31		IUS	US	
hex	**binary**	**decimal**	**BCD**	**EBCDIC**	**ASCII**	**HTML**
20	00100000	32		DS	space	
21	00100001	33		SOS	!	
22	00100010	34		FS	"	&quot;
23	00100011	35			#	&num;
24	00100100	36		BYP	$	&dollar;
25	00100101	37		LF	%	&percnt;
26	00100110	38		ETB	&	&amp;

27	00100111	39		ESC	' (apos)	&apos;
28	00101000	40			(	&lpar;
29	00101001	41			)	&rpar;
2A	00101010	42		SM	*	&ast;
2B	00101011	43		CU2	+	
2C	00101100	44			, (comma)	&comma;
2D	00101101	45		ENQ	-	&hyphen;
2E	00101110	46		ACK	.	&period;
2F	00101111	47		BEL	/	
hex	**binary**	**decimal**	**BCD**	**EBCDIC**	**ASCII**	**HTML**
30	00110000	48			0	
31	00110001	49			1	
32	00110010	50		SYN	2	
33	00110011	51			3	
34	00110100	52		PN	4	
35	00110101	53		RS	5	
36	00110110	54		UC	6	
37	00110111	55		EOT	7	
38	00111000	56			8	
39	00111001	57			9	
3A	00111010	58			:	&colon;
3B	00111011	59		CU3	;	&semi;
3C	00111100	60		DC4	<	&lt;
3D	00111101	61		NAK	=	
3E	00111110	62			>	&gt;
3F	00111111	63		SUB	?	
hex	**binary**	**decimal**	**BCD**	**EBCDIC**	**ASCII**	**HTML**
40	01000000	64	space	space	@	
41	01000001	65			A	
42	01000010	66			B	
43	01000011	67			C	
44	01000100	68			D	
45	01000101	69			E	
46	01000110	70			F	
47	01000111	71			G	
48	01001000	72			H	
49	01001001	73			I	
4A	01001010	74		¢	J	
4B	01001011	75	.	.	K	

4C	01001100	76	□)	<	L	
4D	01001101	77	[	(	M	
4E	01001110	78	<	+	N	
4F	01001111	79		|	O	
hex	**binary**	**decimal**	**BCD**	**EBCDIC**	**ASCII**	**HTML**
50	01010000	80	&+	&	P	
51	01010001	81			Q	
52	01010010	82			R	
53	01010011	83			S	
54	01010100	84			T	
55	01010101	85			U	
56	01010110	86			V	
57	01010111	87			W	
58	01011000	88			X	
59	01011001	89			Y	
5A	01011010	90		!	Z	
5B	01011011	91	$	$	[	&lsqb;
5C	01011100	92	*	*	\	&bsol;
5D	01011101	93	]	)	]	&rsqb;
5E	01011110	94	;	;	^	&circ;
5F	01011111	95		¬	_	&lowbar;
hex	**binary**	**decimal**	**BCD**	**EBCDIC**	**ASCII**	**HTML**
60	01100000	96	-	-	`	&grave;
61	01100001	97	/	/	a	
62	01100010	98			b	
63	01100011	99			c	
64	01100100	100			d	
65	01100101	101			e	
66	01100110	102			f	
67	01100111	103			g	
68	01101000	104			h	
69	01101001	105			i	
6A	01101010	106			j	
6B	01101011	107	,	,	k	
6C	01101100	108	%(	%	l	
6D	01101101	109		—	m	
6E	01101110	110		>	n	
6F	01101111	111		?	o	
hex	**binary**	**decimal**	**BCD**	**EBCDIC**	**ASCII**	**HTML**

70	01110000	112			p	
71	01110001	113			q	
72	01110010	114			r	
73	01110011	115			s	
74	01110100	116			t	
75	01110101	117			u	
76	01110110	118			v	
77	01110111	119			w	
78	01111000	120			x	
79	01111001	121			y	
7A	01111010	122		:	z	
7B	01111011	123	#-	#	{	&lcub;
7C	01111100	124	@'	@	|	&verbar;
7D	01111101	125	:	' (apos)	}	&rcub;
7E	01111110	126	>	=	~	&tilde;
7F	01111111	127		"	DEL	
hex	**binary**	**decimal**	**BCD**	**EBCDIC**	**ASCII**	**HTML**
80	10000000	128				
81	10000001	129		a		
82	10000010	130		b		
83	10000011	131		c		
84	10000100	132		d		
85	10000101	133		e		&ldots;
86	10000110	134		f		
87	10000111	135		g		
88	10001000	136		h		&circ;
89	10001001	137		i		
8A	10001010	138				
8B	10001011	139				
8C	10001100	140				
8D	10001101	141				
8E	10001110	142				
8F	10001111	143				
hex	**binary**	**decimal**	**BCD**	**EBCDIC**	**ASCII**	**HTML**
90	10010000	144				
91	10010001	145		j		
92	10010010	146		k		
93	10010011	147		l		
94	10010100	148		m		

95	10010101	149		n		
96	10010110	150		o		&ndash;
97	10010111	151		p		&mdash;
98	10011000	152		q		
99	10011001	153		r		&trade;
9A	10011010	154				
9B	10011011	155				
9C	10011100	156				
9D	10011101	157				
9E	10011110	158				
9F	10011111	159				
hex	**binary**	**decimal**	**BCD**	**EBCDIC**	**ASCII**	**HTML**
A0	10100000	160				&nbsp;
A1	10100001	161				&iexcl;
A2	10100010	162		s		&cent;
A3	10100011	163		t		&pound;
A4	10100100	164		u		&curren;
A5	10100101	165		v		&yen;
A6	10100110	166		w		&brvbar;
A7	10100111	167		x		&sect;
A8	10101000	168		y		&uml;
A9	10101001	169		z		&copy;
AA	10101010	170				&ordf;
AB	10101011	171				&laquo;
AC	10101100	172				&not;
AD	10101101	173				&shy;
AE	10101110	174				&reg;
AF	10101111	175				&macr;
hex	**binary**	**decimal**	**BCD**	**EBCDIC**	**ASCII**	**HTML**
B0	10110000	176				&deg;
B1	10110001	177				&plusmn;
B2	10110010	178				&sup2;
B3	10110011	179				&sup3;
B4	10110100	180				&acute;
B5	10110101	181				&micro;
B6	10110110	182				&para;
B7	10110111	183				&middot;
B8	10111000	184				&cedil;
B9	10111001	185				&sup1;

BA	10111010	186				&ordm;
BB	10111011	187				&raquo;
BC	10111100	188				&frac14;
BD	10111101	189				&frac12;
BE	10111110	190				&frac34;
BF	10111111	191				&iquest;
hex	**binary**	**decimal**	**BCD**	**EBCDIC**	**ASCII**	**HTML**
C0	11000000	192	?			&Agrave;
C1	11000001	193	A	A		&Aacute;
C2	11000010	194	B	B		&Acirc;
C3	11000011	195	C	C		&Atilde;
C4	11000100	196	D	D		&Auml;
C5	11000101	197	E	E		&Aring;
C6	11000110	198	F	F		&AElig;
C7	11000111	199	G	G		&Ccedil;
C8	11001000	200	H	H		&Egrave;
C9	11001001	201	I	I		&Eacute;
CA	11001010	202				&Ecirc;
CB	11001011	203				&Euml;
CC	11001100	204				&Igrave;
CD	11001101	205				&Iacute;
CE	11001110	206				&Icirc;
CF	11001111	207				&Iuml;
hex	**binary**	**decimal**	**BCD**	**EBCDIC**	**ASCII**	**HTML**
D0	11010000	208	!			&ETH;
D1	11010001	209	J	J		&Ntilde;
D2	11010010	210	K	K		&Ograve;
D3	11010011	211	L	L		&Oacute;
D4	11010100	212	M	M		&Ocirc;
D5	11010101	213	N	N		&Otilde;
D6	11010110	214	O	O		&Ouml;
D7	11010111	215	P	P		&times;
D8	11011000	216	Q	Q		&Oslash;
D9	11011001	217	R	R		&Ugrave;
DA	11011010	218				&Uacute;
DB	11011011	219				&Ucirc;
DC	11011100	220				&Uuml;
DD	11011101	221				&YAcute;
DE	11011110	222				&THORN;

DF	11011111	223				&szlig;
hex	**binary**	**decimal**	**BCD**	**EBCDIC**	**ASCII**	**HTML**
E0	11100000	224				&agrave;
E1	11100001	225				&aacute;
E2	11100010	226	S	S		&acirc;
E3	11100011	227	T	T		&atilde;
E4	11100100	228	U	U		&auml;
E5	11100101	229	V	V		&aring;
E6	11100110	230	W	W		&aelig;
E7	11100111	231	X	X		&ccedil;
E8	11101000	232	Y	Y		&egrave;
E9	11101001	233	Z	Z		&eacute;
EA	11101010	234				&ecirc;
EB	11101011	235				&euml;
EC	11101100	236				&igrave;
ED	11101101	237				&iacute;
EE	11101110	238				&icirc;
EF	11101111	239				&iuml;
hex	**binary**	**decimal**	**BCD**	**EBCDIC**	**ASCII**	**HTML**
F0	11110000	240	0	0		&eth;
F1	11110001	241	1	1		&ntilde;
F2	11110010	242	2	2		&ograve;
F3	11110011	243	3	3		&oacute;
F4	11110100	244	4	4		&ocirc;
F5	11110101	245	5	5		&otilde;
F6	11110110	246	6	6		&ouml;
F7	11110111	247	7	7		&divide;
F8	11111000	248	8	8		&oslash;
F9	11111001	249	9	9		&ugrave;
FA	11111010	250				&uacute;
FB	11111011	251				&ucirc;
FC	11111100	252				&uuml;
FD	11111101	253				&yacute;
FE	11111110	254				&thorn;
FF	11111111	255				&yuml;
hex	**binary**	**decimal**	**BCD**	**EBCDIC**	**ASCII**	**HTML**

table operations

This chapter examines table instructions in assembly language. Specific examples of instructions from various processors are used to illustrate the general nature of assembly language.

table operations

- **TBLS** Table Lookup and Interpolate (Signed, Rounded); Motorola 68300; signed lookup and interpolation of independent variable X from a compressed linear data table or between two register-based table entries of linear representations of dependent variable Y as a function of X; $ENTRY_{(n)} + \{(ENTRY_{(n+1)} - ENTRY_{(n)}) * Dx[7:0]\} / 256$ into Dx; *table version:* data register low word contains the independent variable X, 8-bit integer part and 8-bit fractional part with assumed radix point located between bits 7 and 8, source effective address points to beginning of table in memory, integer part scaled to data size (byte, word, or longword) and used as offset from beginning of table, selected table entry (a linear representation of dependent variable Y) is subtracted from the next consecutive table entry, then multiplied by the interpolation fraction, then divided by 256, then added to the first table entry, and then stored in the data register; *register version:* data register low byte contains the independent variable X 8-bit fractional part with assumed radix point located between bits 7 and 8, two data registers contain the byte, word, or longword table entries (a linear representation of dependent variable Y), first data register-based table entry is subtracted from the second data register-based table entry, then multiplied by the interpolation fraction, then divided by 256, then added to the first table entry, and then stored in the destination (X) data register, the register interpolation mode may be used with several table lookup and interpolations to model multidimentional functions; rounding is selected by the 'R' instruction field, for a rounding adjustment of -1, 0, or +1; the result is an 8-, 16-, or 24-bit integer and eight-bit fraction; interpolation resolution is limited to 1/256th the distance between consecutrive table entries, X should be considered an integer in the range $0 \leq X \leq 65535$; sets or clears flags
- **TBLSN** Table Lookup and Interpolate (Signed, Not Rounded); Motorola 68300; signed lookup and interpolation of independent variable X from a compressed linear data table or between two register-based table entries of linear representations of dependent variable Y as a function of X; $ENTRY_{(n)} * 256 + (ENTRY_{(n+1)} - ENTRY_{(n)}) * Dx[7:0]$ into Dx; *table version:* data register low word contains the independent variable X, 8-bit integer part and 8-bit fractional part with assumed radix point located between bits 7 and 8, source effective address points to beginning of table in memory, integer part scaled to data size (byte, word, or longword) and used as offset from beginning of table, selected table entry (a linear representation of dependent variable Y) multiplied by 256, then added to the value determined by (selected table entry subtracted from the next consecutive table entry, then multiplied by the interpolation fraction), and then stored in the data register; *register version:* data register low byte contains the independent variable X 8-bit fractional part with assumed radix point located between bits 7 and 8, two data registers contain the byte, word, or longword table entries (a linear representation of dependent variable Y), first data register-based table entry is multiplied by 256, then added to the value determined by (first data register-based table entry subtracted from the second data register-based table entry, then multiplied by the interpolation fraction), and then stored in the destination (X) data register, the register interpolation mode may be used with several table lookup and interpolations to model multidimentional functions; the result is an 8-, 16-, or 24-bit integer and eight-bit fraction; interpolation resolution is limited to 1/256th the distance between consecutrive table entries, X should be considered an integer in the range $0 \leq X \leq 65535$; sets or clears flags
- **TBLU** Table Lookup and Interpolate (Unsigned, Rounded); Motorola 68300; unsigned lookup and interpolation of independent variable X from a compressed linear data table or between two register-based table entries of linear representations of dependent variable Y as a function of X; $ENTRY_{(n)} + \{(ENTRY_{(n+1)} - ENTRY_{(n)}) * Dx[7:0]\} / 256$ into Dx; *table version:* data register

low word contains the independent variable X, 8-bit integer part and 8-bit fractional part with assumed radix point located between bits 7 and 8, source effective address points to beginning of table in memory, integer part scaled to data size (byte, word, or longword) and used as offset from beginning of table, selected table entry (a linear representation of dependent variable Y) is subtracted from the next consecutive table entry, then multiplied by the interpolation fraction, then divided by 256, then added to the first table entry, and then stored in the data register; *register version:* data register low byte contains the independent variable X 8-bit fractional part with assumed radix point located between bits 7 and 8, two data registers contain the byte, word, or longword table entries (a linear representation of dependent variable Y), first data register-based table entry is subtracted from the second data register-based table entry, then multiplied by the interpolation fraction, then divided by 256, then added to the first table entry, and then stored in the destination (X) data register, the register interpolation mode may be used with several table lookup and interpolations to model multidimentional functions; rounding is selected by the ‘R’ instruction field, for a rounding adjustment of 0 or +1; the result is an 8-, 16-, or 24-bit integer and eight-bit fraction; interpolation resolution is limited to 1/256th the distance between consecutrive table entries, X should be considered an integer in the range $0 \leq X \leq 65535$; sets or clears flags

- **TBLUN** Table Lookup and Interpolate (Unsigned, Not Rounded); Motorola 68300; unsigned lookup and interpolation of independent variable X from a compressed linear data table or between two register-based table entries of linear representations of dependent variable Y as a function of X; $ENTRY_{(n)} * 256 + (ENTRY_{(n+1)} - ENTRY_{(n)}) * Dx[7:0]$ into Dx; *table version:* data register low word contains the independent variable X, 8-bit integer part and 8-bit fractional part with assumed radix point located between bits 7 and 8, source effective address points to beginning of table in memory, integer part scaled to data size (byte, word, or longword) and used as offset from beginning of table, selected table entry (a linear representation of dependent variable Y) multiplied by 256, then added to the value determined by (selected table entry subtracted from the next consecutive table entry, then multiplied by the interpolation fraction), and then stored in the data register; *register version:* data register low byte contains the independent variable X 8-bit fractional part with assumed radix point located between bits 7 and 8, two data registers contain the byte, word, or longword table entries (a linear representation of dependent variable Y), first data register-based table entry is multiplied by 256, then added to the value determined by (first data register-based table entry subtracted from the second data register-based table entry, then multiplied by the interpolation fraction), and then stored in the destination (X) data register, the register interpolation mode may be used with several table lookup and interpolations to model multidimentional functions; the result is an 8-, 16-, or 24-bit integer and eight-bit fraction; interpolation resolution is limited to 1/256th the distance between consecutrive table entries, X should be considered an integer in the range $0 \leq X \leq 65535$; sets or clears flags

high level language support

This chapter examines high level language support instructions in assembly language. Specific examples of instructions from various processors are used to illustrate the general nature of assembly language.

- assembly/high level language interface
- high level language support

assembly/high level language interface

Because so many search requests are asking about how to interface between high level languages and assembly language, I have added this brief discussion.

Unfortunately, I can only provide you with some general guidelines rather than specific details. You will need to do further research on your own. If your high level compiler allows you to see an assembly language representation of its output, you should be able to figure out the details on your own.

Each processor has its own conventions on subroutine and function linkage. Each high level language also has its own conventions regarding subroutine and function linkage. Some operating systems add their own conventions to the mix. As these three sets of conventions are often at odds, you will need to do your own research to determine how any conflicts in conventions are resolved on your system (again, if your compiler produces an assembly language representation, you can use these listings and carefully crafted test code to determine your local rules).

C treats all subroutines as functions. Unless otherwise declared, each function returns an integer (which is usually the default size of a word on the processor). Parameters are passed by value by pushing the data onto the stack in right to left order (based on the function declaration). Objects less than four bytes (such as boolean, integer, and character) are sign-extended to four bytes. C arrays are passed by pointer. Sometimes C will break the rule and instead pass a pointer for a large data item (such as C arrays), so you may want to examine sample object code to see how various large data structures are handled. Depending on the processor, C function results are returned on the top of the stack (typically the space for the function return value is allocated on the stack after all of the parameters have been passed) or in a register (usually the default scratch register for that processor). The calling routine is responsible for removal of parameters.

Pascal has both subroutines and functions. For subroutines, there is no space allocated on the stack for a return value, while for functions the space for the return value is allocated on the stack after all of the parameters are passed. Parameters are pushed onto the stack in left to right order (based on the subroutine or function declaration). Parameters that are 32 bits or smaller are passed by value (the actual data is placed on the stack), while parameters that are greater than 32 bits are passed by reference (a pointer to the data is placed on the stack). Parameters of variable length are always passed by reference (a pointer) regardless of actual size (this applies in particular to pascal strings). If a data type is not an exact multiple of a byte (such as a bit string or set or certain kinds of ennumerated data), then the data element will be rounded up to the nearest byte, usually with zero padding in the high order bits (check your compiler output). Booleans are passed as a single byte (0 or 1), but boolean function results are returned as a pair of bytes with the boolean in the high byte. VARs are passed as four byte pointers. On some processors, data types that are not exact multiples of 16 or 32 bits are zero padded (high order bytes) to a 16 bit or 32 bit size (again, check your compiler output). The called routine is normally responsible for removal of parameters (other than a function return parameter).

high level language support

Many processors have instructions designed to support constructs common in high level languages. Ironically, a few high level language constructs have been based on specific hardware instructions on specific processors. One famous example is the computed GOTO (three possible branches based on whether the tested value is positive, zero, or negative), which is based on a hardware instruction in an early IBM processor (and if anyone can loan or give me a data book on the processor, I sure would appreciate it).

Most modern processors have some kind of **loop** instructions. These are some variation on the theme of testing for a condition and/or making a count with a short branch back to complete a loop if the exit condition fails.

Many modern processors have some kind of hardware support for **temporary data** storage (for the temporary variables used in subroutines and functions), combining special hardware instructions with argument and/or frame and/or stack pointers.

Bounds check instructions are used to check if an array reference is out of bounds.

- **CMP2** Compare Register Against Bounds; Motorola 680x0, Motorola 68300; compares the contents of register (8, 16, or 32 bits) to a bounds pair (lower bound followed by upper bound), if both bounds are equal then this operation tests for a specific value; sets or clears flags
- **BOUND** Check Array Index Against Bounds; Intel 80x86; compares the contents of register (16 or 32 bits) to a bounds pair (lower bound followed by upper bound) in memory (source register contains address in memory of the first of two consecutive bounds), if the check fails, then an Interrupt 5 occurs; does not modify flags
- **CHK** Check Register Against Bounds; Motorola 680x0, Motorola 68300; compares a word (16-bits) or longword (32-bits) value in a data register to a lower bound of zero and a two's complement upper bound specified in a data register or memory, a value of less than zero or greater than the upper bound results in a CHK instruction exception, vector number 6; sets or clears flags
- **CHK2** Check Register Against Two Bounds; Motorola 680x0, Motorola 68300; compares a byte (8-bits), word (16-bits), or longword (32-bits) value in a data or address register to a bounds pair specified in memory, a value of less than the lower bound or greater than the upper bound results in a CHK instruction exception, vector number 6; sets or clears flags
- **DBcc** Test Condition, Decrement, and Branch; Motorola 680x0, Motorola 68300; used to implement DO loops, WHILE loops, UNTIL loops, and similar constructs, starts by testing a designated condition, if the test is true then no additional action is taken and the program continues to the next instruction (exiting the loop), if the test is false then the designated data register is decremented, if the result is exactly -1 then the program continues to the next instruction (exiting the loop), otherwise the program makes a short (16 bit) branch to continue the loop: DBCC, DBCS, DBEQ, DBF, DBGE, DBGT, DBHI, DBLE, DBLS, DBLT, DBMI, DBNE, DBPL, DBT, DBVC, DBVS
- **LOOP** Loop While ECX Not Zero; Intel 80x86; used to implement DO loops, decrements the ECX or CX (count) register and then tests to see if it is zero, if the ECX or CX register is zero then the program continues to the next instruction (exiting the loop), otherwise the program makes a byte branch to contine the loop; does not modify flags
- **LOOPE** Loop While Equal; Intel 80x86; used to implement DO loops, WHILE loops, UNTIL loops, and similar constructs, decrements the ECX or CX (count) register and then tests to see if it is zero, if the ECX or CX register is zero or the Zero Flag is clear (zero) then the program continues to the next instruction (to exit the loop), otherwise the program makes a byte branch (to continue the loop); equivalent to LOOPZ; does not modify flags
- **LOOPNE** Loop While Not Equal; Intel 80x86; used to implement DO loops, WHILE loops, UNTIL loops, and similar constructs, decrements the ECX or CX (count) register and then tests to see if it is zero, if the ECX or CX register is zero or the Zero Flag is set (one) then the program continues to the next instruction (to exit the loop), otherwise the program makes a byte branch (to continue the loop); equivalent to LOOPNZ; does not modify flags

- **LOOPNZ** Loop While Not Zero; Intel 80x86; used to implement DO loops, WHILE loops, UNTIL loops, and similar constructs, decrements the ECX or CX (count) register and then tests to see if it is zero, if the ECX or CX register is zero or the Zero Flag is set (one) then the program continues to the next instruction (to exit the loop), otherwise the program makes a byte branch (to continue the loop); equivalent to LOOPNE; does not modify flags
- **LOOPZ** Loop While Zero; Intel 80x86; used to implement DO loops, WHILE loops, UNTIL loops, and similar constructs, decrements the ECX or CX (count) register and then tests to see if it is zero, if the ECX or CX register is zero or the Zero Flag is clear (zero) then the program continues to the next instruction (to exit the loop), otherwise the program makes a byte branch (to continue the loop); equivalent to LOOPE; does not modify flags
- **JCXZ** Jump if Count Register Zero; Intel 80x86; conditional jump if CX (count register) is zero; used to prevent entering loop if the count register starts at zero; does not modify flags
- **JECXZ** Jump if Extended Count Register Zero; Intel 80x86; conditional jump if ECX (count register) is zero; used to prevent entering loop if the count register starts at zero; does not modify flags
- **LINK** Link Stack; Motorola 680x0, Motorola 68300
- **UNLK** Unlink Stack; Motorola 680x0, Motorola 68300

program control and condition codes/flags

This chapter examines program control instructions in assembly language. Specific examples of instructions from various processors are used to illustrate the general nature of assembly language.

- program control
- condition codes

program control

Program control instructions change or modify the flow of a program.

The most basic kind of program control is the **unconditional branch** or **unconditional jump**. **Branch** is usually an indication of a short change relative to the current program counter. **Jump** is usually an indication of a change in program counter that is not directly related to the current program counter (such as a jump to an absolute memory location or a jump using a dynamic or static table), and is often free of distance limits from the current program counter.

The pentultimate kind of program control is the **conditional branch** or **conditional jump**. This gives computers their ability to make decisions and implement both loops and algorithms beyond simple formulas.

Most computers have some kind of instructions for **subroutine call** and **return** from subroutines.

There are often instructions for **saving** and **restoring** part or all of the processor state before and after subroutine calls. Some kinds of subroutine or return instructions will include some kinds of save and restore of the processor state.

Even if there are no explicit hardware instructions for subroutine calls and returns, subroutines can be implemented using jumps (saving the return address in a register or memory location for the return jump). Even if there is no hardware support for saving the processor state as a group, most (if not all) of the processor state can be saved and restored one item at a time.

NOP, or no operation, takes up the space of the smallest possible instruction and causes no change in the processor state other than an advancement of the program counter and any time related changes. It can be used to synchronize timing (at least crudely). It is often used during development cycles to temporarily or permanently wipe out a series of instructions without having to reassemble the surrounding code.

Stop or **halt** instructions bring the processor to an orderly halt, remaining in an idle state until restarted by interrupt, trace, reset, or external action.

Reset instructions reset the processor. This may include any or all of: setting registers to an initial value, setting the program counter to a standard starting location (restarting the computer), clearing or setting interrupts, and sending a reset signal to external devices.

- **BRA** Branch; Motorola 680x0, Motorola 68300; short (16 bit) unconditional branch relative to the current program counter
- **JMP** Jump; Motorola 680x0, Motorola 68300; unconditional jump (any valid effective addressing mode other than data register)
- **JMP** Jump; Intel 80x86; unconditional jump (near [relative displacement from PC] or far; direct or indirect [based on contents of general purpose register, memory location, or indexed])
- **JMP** Jump; MIX; unconditional jump to location M; J-register loaded with the address of the

instruction which would have been next if the jump had not been taken

- **JSJ** Jump, Save J-register; MIX; unconditional jump to location M; J-register unchanged
- **Jcc** Jump Conditionally; Intel 80x86; conditional jump (near [relative displacement from PC] or far; direct or indirect [based on contents of general purpose register, memory location, or indexed]) based on a tested condition: JA/JNBE, JAE/JNB, JB/JNAE, JBE/JNA, JC, JE/JZ, JNC, JNE/JNZ, JNP/JPO, JP/JPE, JG/JNLE, JGE/JNL, JL/JNGE, JLE/JNG, JNO, JNS, JO, JS
- **Bcc** Branch Conditionally; Motorola 680x0, Motorola 68300; short (16 bit) conditional branch relative to the current program counter based on a tested condition: BCC, BCS, BEQ, BGE, BGT, BHI, BLE, BLS, BLT, BMI, BNE, BPL, BVC, BVS
- **JOV** Jump on Overflow; MIX; conditional jump to location M if overflow toggle is on; if jump occurs, J-register loaded with the address of the instruction which would have been next if the jump had not been taken
- **JNOV** Jump on No Overflow; MIX; conditional jump to location M if overflow toggle is off; if jump occurs, J-register loaded with the address of the instruction which would have been next if the jump had not been taken
- **Jcc** Jump on Condition; MIX; conditional jump to location M based on comparison indicator; if jump occurs, J-register loaded with the address of the instruction which would have been next if the jump had not been taken; JL (less), JE (equal), JG (greater), JGE (greater-or-equal), JNE (unequal), JLE (less-or-equal)
- **JAcc** Jump on A-register; MIX; conditional jump to location M based on A-register (accumulator); if jump occurs, J-register loaded with the address of the instruction which would have been next if the jump had not been taken; JAN (negative), JAZ (zero), JAP (positive), JANN (nonnegative), JANZ (nonzero, JAMP (nonpositive)
- **JXcc** Jump on X-register; MIX; conditional jump to location M based on X-register (extension); if jump occurs, J-register loaded with the address of the instruction which would have been next if the jump had not been taken; JXN (negative), JXZ (zero), JXP (positive), JXNN (nonnegative), JXNZ (nonzero, JXMP (nonpositive)
- **J*i*cc** Jump on I-register; MIX; conditional jump to location M based on one of five I-registers (index); if jump occurs, J-register loaded with the address of the instruction which would have been next if the jump had not been taken; J*i*N (negative), J*i*Z (zero), J*i*P (positive), J*i*NN (nonnegative), J*i*NZ (nonzero, J*i*MP (nonpositive)
- **DBcc** Test Condition, Decrement, and Branch; Motorola 680x0, Motorola 68300; used to implement DO loops, WHILE loops, UNTIL loops, and similar constructs, starts by testing a designated condition, if the test is true then no additional action is taken and the program continues to the next instruction (exiting the loop), if the test is false then the designated data register is decremented, if the result is exactly -1 then the program continues to the next instruction (exiting the loop), otherwise the program makes a short (16 bit) branch (continueing the loop): DBCC, DBCS, DBEQ, DBF, DBGE, DBGT, DBHI, DBLE, DBLS, DBLT, DBMI, DBNE, DBPL, DBT, DBVC, DBVS
- **LOOP** Loop While ECX Not Zero; Intel 80x86; used to implement DO loops, decrements the ECX or CX (count) register and then tests to see if it is zero, if the ECX or CX register is zero then the program continues to the next instruction (exiting the loop), otherwise the program makes a byte branch to contine the loop; does not modify flags
- **LOOPE** Loop While Equal; Intel 80x86; used to implement DO loops, WHILE loops, UNTIL loops, and similar constructs, decrements the ECX or CX (count) register and then tests to see if it is zero, if the ECX or CX register is zero or the Zero Flag is clear (zero) then the program continues to the next instruction (to exit the loop), otherwise the program makes a byte branch (to continue the loop); equivalent to LOOPZ; does not modify flags
- **LOOPNE** Loop While Not Equal; Intel 80x86; used to implement DO loops, WHILE loops, UNTIL loops, and similar constructs, decrements the ECX or CX (count) register and then tests to see if it is zero, if the ECX or CX register is zero or the Zero Flag is set (one) then the program continues to the next instruction (to exit the loop), otherwise the program makes a byte branch (to continue the loop); equivalent to LOOPNZ; does not modify flags
- **LOOPNZ** Loop While Not Zero; Intel 80x86; used to implement DO loops, WHILE loops, UNTIL loops, and similar constructs, decrements the ECX or CX (count) register and then tests

to see if it is zero, if the ECX or CX register is zero or the Zero Flag is set (one) then the program continues to the next instruction (to exit the loop), otherwise the program makes a byte branch (to continue the loop); equivalent to LOOPNE; does not modify flags

- **LOOPZ** Loop While Zero; Intel 80x86; used to implement DO loops, WHILE loops, UNTIL loops, and similar constructs, decrements the ECX or CX (count) register and then tests to see if it is zero, if the ECX or CX register is zero or the Zero Flag is clear (zero) then the program continues to the next instruction (to exit the loop), otherwise the program makes a byte branch (to continue the loop); equivalent to LOOPE; does not modify flags
- **JCXZ** Jump if Count Register Zero; Intel 80x86; conditional jump if CX (count register) is zero; used to prevent entering loop if the count register starts at zero; does not modify flags
- **JECXZ** Jump if Extended Count Register Zero; Intel 80x86; conditional jump if ECX (count register) is zero; used to prevent entering loop if the count register starts at zero; does not modify flags
- **Scc** Set According to Condition; Motorola 680x0, Motorola 68300; tests a condition code, if the condition is true then sets a byte (8 bits) of a data register or memory location to TRUE (all ones), if the condition is false then sets a byte (8 bits) of a data register or memory location to FALSE (all zeros): SCC, SCS, SEQ, SF, SGE, SGT, SHI, SLE, SLS, SLT, SMI, SNE, SPL, ST, SVC, SVS
- **SETcc** Set Byte on Condition cc; Intel 80x86; tests a condition code, if the condition is true then sets a byte (8 bits) of a data register or memory location to TRUE (all ones), if the condition is false then sets a byte (8 bits) of a data register or memory location to FALSE (all zeros): SETA, SETAE, SETB, SETBE, SETC, SETE, SETG, SETGE, SETL, SETLE, SETNA, SETNAE, SETNB, SETNBE, SETNC, SETNE, SETNG, SETNGE, SETNL, SETNLE, SETNO, SETNP, SETNS, SETNZ, SETO, SETP, SETPE, SETPO, SETS, SETZ
- **BSR** Branch to Subroutine; Motorola 680x0, Motorola 68300; pushes the address of the next instruction following the subroutine call onto the system stack, decrements the system stack pointer, and changes program flow to a location (8, 16, or 32 bits) relative to the current program counter
- **JSR** Jump to Subroutine; Motorola 680x0, Motorola 68300; pushes the address of the next instruction following the subroutine call onto the system stack, decrements the system stack pointer, and changes program flow to the address specified (any valid effective addressing mode other than data register)
- **CALL** Call Procedure; Intel 80x86; pushes the address of the next instruction following the subroutine call onto the system stack, decrements the system stack pointer, and changes program flow to the address specified (near [relative displacement from PC] or far; direct or indirect [based on contents of general purpose register or memory location])
- **RTS** Return from Subroutine; Motorola 680x0, Motorola 68300; fetches the return address from the top of the system stack, increments the system stack pointer, and changes program flow to the return address
- **RET** Return From Procedure; Intel 80x86; fetches the return address from the top of the system stack, increments the system stack pointer, and changes program flow to the return address; optional immediate operand added to the new top-of-stack pointer, effectively removing any arguments that the calling program pushed on the stack before the execution of the corresponding CALL instruction; possible change to lesser privilege
- **RTR** Return and Restore Condition Codes; Motorola 680x0, Motorola 68300; transfers the value at the top of the system stack into the condition code register, increments the system stack pointer, fetches the return address from the top of the system stack, increments the system stack pointer, and changes program flow to the return address
- **IRET** Return From Interrupt; Intel 80x86; transfers the value at the top of the system stack into the flags register, increments the system stack pointer, fetches the return address from the top of the system stack, increments the system stack pointer, and changes program flow to the return address; optional immediate operand added to the new top-of-stack pointer, effectively removing any arguments that the calling program pushed on the stack before the execution of the corresponding CALL instruction; possible change to lesser privilege
- **RTD** Return and Deallocate; Motorola 680x0, Motorola 68300; fetches the return address from

the top of the system stack, increments the system stack pointer by the specified displacement value (effectively deallocating temporary storage space from the stack), and changes program flow to the return address
- **RTE** Return from Exception; Motorola 680x0, Motorola 68300; transfers the value at the top of the system stack into the status register, increments the system stack pointer, fetches the return address from the top of the system stack, increments the system stack pointer by a displacement value designated by format mode (effectively deallocating temporary storage space from the stack, the amount of space varying by type of exception that occurred), and changes program flow to the return address; *privileged instruction* (supervisor state)
- **MOVEM** Move Multiple; Motorola 680x0, Motorola 68300; move contents of a list of registers to memory or restore from memory to a list of registers
- **LM** Load Multiple; IBM 360/370; RS format; moves a series of full words (32 bits) of data from memory to a series of general purpose registers; main storage to register only; does not affect condition code
- **STM** STore Multiple; IBM 360/370; RS format; moves contents of a series of general purpose registers to a series of full words (32 bits) in memory; register to main storage only; does not affect condition code
- **PUSHA** Push All Registers; Intel 80x86; move contents all 16-bit general purpose registers to memory pointed to by stack pointer (in the order AX, CX, DX, BX, original SP, BP, SI, and DI); does not affect flags
- **PUSHAD** Push All Registers; Intel 80386; move contents all 32-bit general purpose registers to memory pointed to by stack pointer (in the order EAX, ECX, EDX, EBX, original ESP, EBP, ESI, and EDI); does not affect flags
- **POPA** Pop All Registers; Intel 80x86; move memory pointed to by stack pointer to all 16-bit general purpose registers (except for SP); does not affect flags
- **POPAD** Pop All Registers; Intel 80386; move memory pointed to by stack pointer to all 32-bit general purpose registers (except for ESP); does not affect flags
- **STJ** Store jump-register; MIX; move word or partial word field of data; jump register to main storage only
- **NOP** No Operation; Motorola 680x0, Motorola 68300; no change in processor state other than an advance of the program counter
- **NOP** No Operation; MIX; no change in processor state other than an advance of the program counter
- **STOP** Stop; Motorola 680x0, Motorola 68300; loads an immediate operand into the program status register (both user and supervisor portions), advances program counter to next instruction, and stops the processor from fetching and executing instructions; *privileged instruction* (supervisor state)
- **HLT** Halt; MIX; stop machine, computer restarts on next instruction
- **LPSTOP** Low Power Stop; Motorola 68300; loads an immediate operand into the program status register (both user and supervisor portions), advances program counter to next instruction, and stops the processor from fetchhing and executing instructions, the new interrupt mask is copied to the external bus interface (EBI), internal clocks are stopped, the processor remains stopped until a trace, higher interrupt than new mask, or reset exception occurs; *privileged instruction* (supervisor state)

condition codes

Condition codes are the list of possible conditions that can be tested during conditional instructions. Typical conditional instructions include: conditional branches, conditional jumps, and conditional subroutine calls. Some processors have a few additional data related conditional instructions, and some processors make every instruction conditional. Not all condition codes available for a processor will be implemented for every conditional instruction.

Zero is mathematically neither positive nor negative, but for processor condition codes, most processors treat zero as either a positive or a negative numbers. Processors that treat zero as a positive number include the Motorola 680x0 and Motorola 68300.

- **A** above; Intel 80x86; unsigned conditional transfer; equivalent to NBE; (not carry flag and not zero flag)
- **AE** above or equal; Intel 80x86; unsigned conditional transfer; equivalent to NB; (not carry flag
- **B** below; Intel 80x86; unsigned conditional transfer; equivalent to NAE; (carry flag)
- **BE** below or equal; Intel 80x86; unsigned conditional transfer; equivalent to NA; (carry flag or zero flag)
- **C** carry; Intel 80x86; unsigned conditional transfer; (carry flag)
- **CC** Carry Clear; Motorola 680x0, Motorola 68300; not carry flag
- **CS** Carry Set; Motorola 680x0, Motorola 68300; carry flag
- **E** equal; Intel 80x86; unsigned conditional transfer; equivalent to Z; (zero flag)
- **EQ** Equal; Motorola 680x0, Motorola 68300; zero flag
- **F** False (never true); Motorola 680x0, Motorola 68300; never
- **G** greater; Intel 80x86; signed conditional transfer; equivalent to NLE; (not ((sign flag xor overflow flag) or zero flag))
- **GE** Greater or Equal; Motorola 680x0, Motorola 68300; (negative flag and overflow flag) or (not negative flag and not overflow flag)
- **GE** greater or equal; Intel 80x86; signed conditional transfer; equivalent to NL; (not (sign flag xor overflow flag))
- **GT** Greater Than; Motorola 680x0, Motorola 68300; (negative flag and overflow flag and not zero flag) or (not negative flag and not overflow flag and not zero flag)
- **HI** High; Motorola 680x0, Motorola 68300; not carry flag and not zero flag
- **L** less; Intel 80x86; signed conditional transfer; equivalent to NGE; (sign flag xor overflow flag)
- **LE** Less or Equal; Motorola 680x0, Motorola 68300; (zero flag) or (negative flag and not overflow flag) or (not negative flag and overflow flag)
- **LE** less or equal; Intel 80x86; signed conditional transfer; equivalent to NG; ((sign flag xor overflow flag) or zero flag)
- **LS** Low or Same; Motorola 680x0, Motorola 68300; carry flag or zero flag
- **LT** Less Than; Motorola 680x0, Motorola 68300; (negative flag and not overflow flag) or (not negative flag and overflow flag)
- **MI** Minus; Motorola 680x0, Motorola 68300; negative flag
- **NA** not above; Intel 80x86; unsigned conditional transfer; equivalent to BE; (carry flag or zero flag)
- **NAE** not above nor equal; Intel 80x86; unsigned conditional transfer; equivalent to B; (carry flag)
- **NB** not below; Intel 80x86; unsigned conditional transfer; equivalent to AE; (not carry flag)
- **NBE** not below nor equal; Intel 80x86; unsigned conditional transfer; equivalent to A; (not carry flag and not zero flag)
- **NC** not carry; Intel 80x86; unsigned conditional transfer; (not carry flag)
- **NE** Not Equal; Motorola 680x0, Motorola 68300; not zero flag
- **NE** not equal; Intel 80x86; unsigned conditional transfer; equivalent to NZ; (not zero flag)
- **NG** not greater; Intel 80x86; signed conditional transfer; equivalent to LE; ((sign flag xor overflow flag) or zero flag)
- **NGE** not greater nor equal; Intel 80x86; signed conditional transfer; equivalent to L; (sign flag xor overflow flag)
- **NL** not less; Intel 80x86; signed conditional transfer; equivalent to GE; (not (sign flag xor overflow flag))
- **NLE** not less nor equal; Intel 80x86; signed conditional transfer; equivalent to G; (not ((sign flag xor overflow flag) or zero flag))
- **NO** not overflow; Intel 80x86; signed conditional transfer; (not overflow flag)
- **NP** not parity; Intel 80x86; unsigned conditional transfer; equivalent to PO; (not parity flag)
- **NS** not sign (positive or zero); Intel 80x86; signed conditional transfer; (not sign flag)
- **NZ** not zero; Intel 80x86; unsigned conditional transfer; equivalent to NE; (not zero flag)
- **O** overflow; Intel 80x86; signed conditional transfer; (overflow flag)

- **P** parity; Intel 80x86; unsigned conditional transfer; equivalent to PE; (parity flag)
- **PE** parity; Intel 80x86; unsigned conditional transfer; equivalent to P; (parity flag)
- **PL** Plus; Motorola 680x0, Motorola 68300; not negative flag
- **PO** parity odd; Intel 80x86; unsigned conditional transfer; equivalent to NP; (not parity flag)
- **S** sign (negative); Intel 80x86; signed conditional transfer; (sign flag)
- **T** True (always true); Motorola 680x0, Motorola 68300; always
- **VC** Overflow Clear; Motorola 680x0, Motorola 68300; not overflow flag
- **VS** Overflow Set; Motorola 680x0, Motorola 68300; overflow flag
- **Z** zero; Intel 80x86; unsigned conditional transfer; equivalent to E; (zero flag)

input/output instructions

This chapter examines input/output instructions in assembly language. Specific examples of instructions from various processors are used to illustrate the general nature of assembly language.

- input/output
    - MIX devices

input/output

Input/Output (I/O) instructions are used to input data from peripherals, output data to peripherals, or read/write input/output controls. Early computers used special hardware to handle I/O devices. The trend in modern computers is to map I/O devices in memory, allowing the direct use of any instruction that operates on memory for handling I/O.

- **IN** Input; MIX; initiate transfer of information from the input device specified into consecutive locations starting with M, block size implied by unit
- **OUT** Output; MIX; initiate transfer of information from consecutive locations starting with M to the output device specified, block size implied by unit
- **IOC** Input-Output Control; MIX; initiate I/O control operation to be performed by designated device
- **JRED** Jump Ready; MIX; Jump if specified unit is ready (completed previous IN, OUT, or IOC operation); if jump occurs, J-register loaded with the address of the instruction which would have been next if the jump had not been taken
- **JBUS** Jump Busy; MIX; Jump if specified unit is not ready (not yet completed previous IN, OUT, or IOC operation); if jump occurs, J-register loaded with the address of the instruction which would have been next if the jump had not been taken

MIX devices

Information on the devices for the hypothetical MIX processor’s input/output instructions.

unit number	peripheral	block size	control
t	Tape unit no. *i* ($0 \leq i \leq 7$)	100 words	M=0, tape rewound; M < 0, skip back M records; M > 0, skip forward M records
d	Disk or drum unit no. *d* ($8 \leq d \leq 15$)	100 words	position device according to X-register (extension)
16	Card reader	16 words	
17	Card punch	16 words	
18	Printer	24 words	IOC 0(18) skips printer to top of following page
19	Typewriter and paper tape	14 words	paper tape reader: rewind tape

system control instructions

This chapter examines system control instructions in assembly language. Specific examples of instructions from various processors are used to illustrate the general nature of assembly language.

system control

System control instructions control some basic element of the system or processor state.

Many system control instructions are **privileged**, meaning that only certain trusted routines are allowed to use them. This is implemented by having privilege states. The most simple version is two states: **user** and **supervisor** states. The user state can’t run any privileged instructions, while the supervisor state can run all instructions. Some processors have more than two privilege states, allowing greater granularity of freedom to increasingly trusted operations.

The most basic kind of system control instructions are those that modify the condition codes or user portion of a status register.

Closely related are instructions that modify an entire status word or status register. The more powerful version is a privileged instruction and includes access to portions of the status register that can control or modify other processes.

Machine control instructions directly affect the entire processor. **Stop** or **halt** instructions bring the processor to an orderly halt, remaining in an idle state until restarted by interrupt, trace, reset, or external action.

Reset instructions reset the processor. This may include any or all of: setting registers to an initial value, setting the program counter to a standard starting location (restarting the computer), clearing or setting interrupts, and sending a reset signal to external devices.

Trap generating instructions generate a system trap. This includes a transition to a privileged state and turns control over to a routine with supervisor permission. This allows user processes to communicate with and make requests of the operating system. Note that it is common for some parts of an operating system to run in normal user mode so as to limit potential damage if something goes wrong.

Memory management instructions control memory and how memory is mapped and accessed by user and system routines. These instructions are almost always privileged and vary greatly from processor to processor (although the general capabilities and effects are pretty standard).

- **MOVE <ea>, CCR** Move to Condition Codes Register; Motorola 680x0, Motorola 68300; moves data from data register, memory, or immediate data to user condition codes register
- **MOVE CCR, <ea>** Move from Condition Codes Register; Motorola 680x0, Motorola 68300; moves data from user condition codes register to data register or memory
- **ANDI #data, CCR** And Immediate to Condition Codes Register; Motorola 680x0, Motorola 68300; logical AND of the immediate data with the user condition codes register
- **ORI #data, CCR** Or Immediate to Condition Codes Register; Motorola 680x0, Motorola 68300; logical inclusive OR of the immediate data with the user condition codes register
- **EORI #data, CCR** Exclusive Or Immediate to Condition Codes Register; Motorola 680x0, Motorola 68300; logical exclusive OR of the immediate data with the user condition codes register
- **MOVE <ea>, SR** Move to Status Register; Motorola 680x0, Motorola 68300; moves data from data register, memory, or immediate data to entire status register; *privileged instruction*

(supervisor state)

- **MOVE SR, <ea>** Move from Status Register; Motorola 680x0, Motorola 68300; moves data from entire status register to data register or memory; *privileged instruction* (supervisor state)
- **ANDI #data, SR** And Immediate to Status Register; Motorola 680x0, Motorola 68300; logical AND of the immediate data with the entire status register; *privileged instruction* (supervisor state)
- **ORI #data, SR** Or Immediate to Status Register; Motorola 680x0, Motorola 68300; logical inclusive OR of the immediate data with the entire status register; *privileged instruction* (supervisor state)
- **EORI #data, SR** Exclusive Or Immediate to Status Register; Motorola 680x0, Motorola 68300; logical exclusive OR of the immediate data with the entire status register; *privileged instruction* (supervisor state)
- **MOVE USP** Move User Stack Pointer; Motorola 680x0, Motorola 68300; moves the contents of the user stack pointer to or from the specified address register; *privileged instruction* (supervisor state)
- **MOVEC** Move Control Register; Motorola 680x0, Motorola 68300; moves the contents of the specified address or data register to the specified control register or moves the contents of the specified control register to the specified data or address register (zero filled to 32 bits), control registers: Source Function Code (SFC) register, Destination Function Code (DFC) register, Cache Control Register (CACR), User Stack Pointer (USP), Vector Base Register (VBR), Cache Address Register (CAAR), Master Stack Pointer (MSP), Interrupt Stack Pointer (ISP); *privileged instruction* (supervisor state)
- **MOVES** Move Address Space; Motorola 680x0, Motorola 68300; moves information between address spaces (allowing data communication across process boundaries), either moving data (8, 16, or 32 bits) from a specified address or data register to a memory location in the address space specified by the destination fucntion code (DFC) register or moves date (8, 16, or 32 bits) from a memory location in the address space specified by the source function code (SFC) register to the specified data or address register; does not modify flags; *privileged instruction* (supervisor state)
- **STOP** Stop; Motorola 680x0, Motorola 68300; loads an immediate operand into the program status register (both user and supervisor portions), advances program counter to next instruction, and stops the processor from fetchhing and executing instructions; *privileged instruction* (supervisor state)
- **LPSTOP** Low Power Stop; Motorola 68300; loads an immediate operand into the program status register (both user and supervisor portions), advances program counter to next instruction, and stops the processor from fetchhing and executing instructions, the new interrupt mask is copied to the external bus interface (EBI), internal clocks are stopped, the processor remains stopped until a trace, higher interrupt than new mask, or reset exception occurs; *privileged instruction* (supervisor state)
- **RESET** Reset External Devices; Motorola 680x0, Motorola 68300; asserts the NOT RESET signal for 512 clock periods, resetting all external devices, no internal changes other than incrementing program counter to the next instruction; *privileged instruction* (supervisor state)
- **RTE** Return from Exception; Motorola 680x0, Motorola 68300; transfers the value at the top of the system stack into the status register, increments the system stack pointer, fetches the return address from the top of the system stack, increments the system stack pointer by a displacement value designated by format mode (effectively deallocating temporary storage space from the stack, the amount of space varying by type of exception that occurred), and changes program flow to the return address; *privileged instruction* (supervisor state)

coprocessor and multiprocessor operations

This chapter examines multiprocessor and coprocessor instructions in assembly language. Specific examples of instructions from various processors are used to illustrate the general nature of assembly language.

coprocessor and multiprocessor operations

Multiprocessor instructions are used to coordinate activity between multiple processors.

Some multiprocessor instructions are designed to allow the processors to communicate with each other. A **test and set** instruction is used to implement flags or semaphores between processors. A **compare and swap** instruction is used to implement more sophsticated communications between multiple processors (such as counters or queue pointers) or secure updates of shared system control data structures in a multi-processing environment. **Interlocked** instructions are used to update counters, flags, and semaphores while locking out any other processors or devices from changing or reading the memory location while it is being updated.

- **TAS** Test and Set an Operand; Motorola 680x0; tests the current value of the operand at the effective address in memory or a data register (obviously, data register operands aren’t multiprocessor, but the instruction still works) and sets or clears the N (negative) and Z (zero) condition codes accordingly, then sets the high order bit of the operand to one; this operation uses a read-modify-write memory cycle that completes the operation without interruption; sets or clears flags
- **CAS** Compare and Swap with Operand; Motorola 680x0; Compares the operand at an effective address in memory to a compare operand in a data register, if the operands are equal, the update operand is transferred from a data register to the original effective address, if the operands are unequal, the operand at the effective address is transferred to the data register that contained the compare operand; this operation uses a read-modify-write memory cycle that completes the operation without interruption; sets or clears flags
- **CAS2** Dual Operand Compare and Swap; Motorola 680x0; Compares the operand at an effective address in memory to a compare operand in a data register, if the operands are equal, compares the operand at a second effective address in memory to a second compare operand in a data register, if the second operand is also equal, the update operands are transferred from a data register to the pair of original effective addresses, if either pair of operands are unequal, the operands at the pair of effective addresses are transferred to the data registers that contained the compare operands; this operation uses a read-modify-write memory cycle that completes the operation without interruption; sets or clears flags
- **ADAWI** Add Aligned Word Interlocked; DEC VAX; adds (16 bit integer) a source operand from a register or memory to a memory location that is word aligned while interlocking the memory location so that no other processor or device can read or write to the interlocked memory location, used to maintain operating system resource usage counts; and sets or clears flags

trap generating instructions

This chapter examines trap generating instructions in assembly language. Specific examples of instructions from various processors are used to illustrate the general nature of assembly language.

trap generating

Trap generating instructions generate an exception that transfer control from software (usually application programs) to the operating system.

Operating system **traps** provide a mechanism to change to higher privilege levels (if they exist on the processor) and usually include a mechanism for identifying what kind of trap has occurred. This allows application programs to make requests of the operating system and may provide a hardware mechanism for switching from a user mode to a superviser, kernel, or other higher privilege level for the operating system response to the request.

A **breakpoint** instruction is used with external debugging hardware. The breakpoint instruction replaces an ordinary instruction (or the first part of an instruction) and relies on external debugging hardware to supply the missing instruction (or part of an instruction).

Various **check** instructions will test for conditions, trapping if the test fails. One common example is a check against bounds or limits.

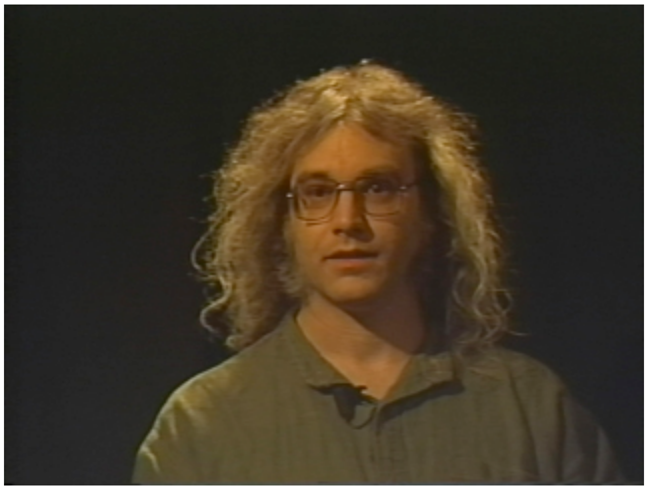
Most processors have one or more **illegal** instructions. These are usually instructions that haven't been implemented yet or instructions that have been dropped from a processor line. Many processors generate a trap or exception upon encountering an illegal instruction. Some processors will execute illegal instructions, which can lead to undocumented operations. Undocumented operations tend to change from one batch of processors to another and are highly unreliable. Some processors reserve an opcode that is guaranteed to always be illegal and always generate a trap or exception.

- **TRAP** Trap; Motorola 680x0, Motorola 68300; generates a trap exception with a trap vector (32 plus an immediate value in the range of 0 to 15); does not modify flags
- **TRAPcc** Trap on Condition; Motorola 680x0, Motorola 68300; if the tested condition is ture, generates a trap exception with vector 7, with optional word (16-bits) or longword (32-bits) immediate operand being available for the software trap handler: TRAPCC, TRAPCS, TRAPEQ, TRAPF, TRAPGE, TRAPGT, TRAPHI, TRAPLE, TRAPLS, TRAPLT, TRAPMI, TRAPNE, TRAPPL, TRAPT, TRAPVC, TRAPVS
- **Axxx** A-line Trap; Motorola 680x0, Motorola 68300; generates an a-line trap with the 12 bit value in the byte and a half of the instruction word used as a vector into a trap table; reserved for use by computer hardware manufacturers to provide software routines or implement supporting hardware features (used in the Macintosh to provide operating system calls); does not modify flags
- **BKPT** Breakpoint; Motorola 680x0, Motorola 68300; asserts a breakpoint acknowledge bus cycle with an immediate breakpoint vector (0 to 7) on address lines A2-A4 (not to be confused with address registers), if external hardware terminates the cycle, the data (one instruction word) on the bus is inserted into the instruction pipe, otherwise generates an illegal instruction exception; does not modify flags
- **CHK** Check Register Against Bounds; Motorola 680x0, Motorola 68300; compares a word (16-bits) or longword (32-bits) value in a data register to a lower bound of zero and a two's complement upper bound specified in a data register or memory, a value of less than zero or greater than the upper bound results in a CHK instruction exception, vector number 6; sets or clears flags
- **CHK2** Check Register Against Two Bounds; Motorola 680x0, Motorola 68300; compares a byte

(8-bits), word (16-bits), or longword (32-bits) value in a data or address register to a bounds pair specified in memory, a value of less than the lower bound or greater than the upper bound results in a CHK instruction exception, vector number 6; sets or clears flags

- **BOUND** Check Array Index Against Bounds; Intel 80x86; compares the contents of register (16 or 32 bits) to a bounds pair (lower bound followed by upper bound) in memory (source register contains address in memory of the first of two consecutive bounds), if the check fails, then an Interrupt 5 occurs; does not modify flags
- **ILLEGAL** Take Illegal Instruction Trap; Motorola 680x0, Motorola 68300; generates an illegal instruction exception, vector 4; does not modify flags
- **TRAPV** Trap; Motorola 680x0, Motorola 68300; if the overflow condition is set, generates a TRAPV exception, vector 7; does not modify flags
- **INT** Interrupt; Intel 80x86; pushes flags register and return address on stack and then generates a software call to the interrupt handler designated by the immediate operand (0 to 255) as in index into the Interrupt Descriptor Table (IDT), in Protected Mode the IDT is an array of eight-byte descriptors, in Real Address Mode the IDT is an array of doubleword (32 bit) pointers, the first 32 entries are reserved to Intel (matching hardware interrupts and exceptions), the base address of the IDT is the contents of the IDTR
- **INTO** Interrupt if Overflow; Intel 80x86; if the Overflow flag is set, pushes flags register and return address on stack and then generates a software call to the fourth (4) interrupt handler in the Interrupt Descriptor Table (IDT), in Protected Mode the IDT is an array of eight-byte descriptors, in Real Address Mode the IDT is an array of doubleword (32 bit) pointers, the first 32 entries are reserved to Intel (matching hardware interrupts and exceptions), the base address of the IDT is the contents of the IDTR
- **IRET** Return From Interrupt; Intel 80x86; transfers the value at the top of the system stack into the flags register, increments the system stack pointer, fetches the return address from the top of the system stack, increments the system stack pointer, and changes program flow to the return address; optional immediate operand added to the new top-of-stack pointer, effectively removing any arguments that the calling program pushed on the stack before the execution of the corresponding CALL instruction; possible change to lesser privilege
- **RTE** Return from Exception; Motorola 680x0, Motorola 68300; transfers the value at the top of the system stack into the status register, increments the system stack pointer, fetches the return address from the top of the system stack, increments the system stack pointer by a displacement value designated by format mode (effectively deallocating temporary storage space from the stack, the amount of space varying by type of exception that occurred), and changes program flow to the return address; *privileged instruction* (supervisor state)

Milo



contact information

real mail

Milo, PO Box 1361, Tustin, Calif, 92781, USA.

PDF version history

0 14 October 2007.

author: Milo, PO Box 1361, Tustin, CA, 92781, USA

This book handcrafted on Macintosh computers using Tom Bender’s Tex-Edit Plus.