Programmed Introduction to MIPS Assembly Language

Bradley Kjell, Central Connecticut State University

Revised Draft, June 2002

This is a course in assembly language programming of the MIPS processor. It emphasizes the topics needed for study of computer architecture: bits, bit patterns, operations on bit patterns, and how bit patterns represent instructions and data. This course is equivalent to a semester-long junior college or university course (except, perhaps, for the emphasis on bit patterns).

The only equipment you need for this course is a PC. The only software you need is the SPIM simulator of the MIPS R2000 processor, and a text editor. The simulator is available by free download (see appendix A). Example programs are presented using an MS Windows operating system. However, you can use any platform that runs SPIM. (There are many).

Assembly Language is normally taken the semester after a course in a high level programming language (such as Java or C). This course assumes that you have this background, although no specific programming language is required.

Read the pages of this course *actively*. Think about and answer the question at the bottom of each page. (This style of tutorial is called *programmed learning*. It is very effective for technical material). Most questions call for just a little thought. Some call for pencil and paper. Keep a pencil and a scrap of paper next to your keyboard. Each chapter is about 15 pages long. Spend several minutes per page. You can read each chapter in about 30 minutes.

Works best with IE 5.0 at 800 by 600

Part 1: Prelude to Assembly Language

Assembly language: what it is, why it is studied, and where it is used.

Appendices

Index

Main Tutorial Menu

created: 06/25/96; 2nd edition: 03/17/01

CHAPTER 1 — INTRODUCTION

This is a tutorial in assembly language for the MIPS R2000 processor chip using the SPIM simulator. Assembly language is used to write programs in terms of the basic operations of a processor. The **architecture** of a computer is a logical description of its components and its basic operations. In "pure" assembly language the architecture of the processor chip is clearly visible: one assembly language statement corresponds to one basic operation of the processor. With assembly language the programmer is clearly aware of the processor that will run the program.

"Pure" assembly language is rare. Usually, for programmer convenience, assembly language is **extended** with statements that correspond to several basic machine operations. MIPS assembly language does this, but the processor chip is still visible. Programs in high level languages such as C or Pascal are (mostly) independent of the processor they run on. The processor is not visible in the language. This chapter starts out a tour of assembly language.

Chapter Topics:

QUESTION 1:

Do all processor chips have the same architecture?

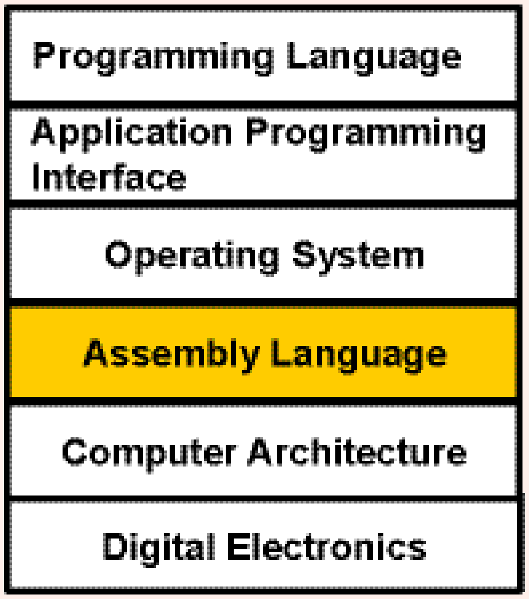
Next

Do all processor chips have the same architecture?

A good answer might be:

No. Each family of processor chip (MIPS, PIC, SPARC, Alpha, Motorola, Intel, et al.) has its own architecture.

Different Processor Architectures



The **architecture** of a computer is a description of its components and its operations.

Each processor family has its own architecture. Assembly language is a programming view of the architecture of a processor. So each type of processor has its own assembly language.

When you study an assembly language, you study the architecture of a particular processor. The study of any assembly language increases your professional understanding of computers These notes are about the MIPS processor, which is a nice processor to study. The concepts in MIPS assembly are universal.

These notes discuss assembly language from a computer science perspective. They explain what is going on in a computer system at the **assembly language level**. This is a view that is above the electronic view of computer architecture, but is below the operating systems view of the computer system. Computer scientists understand computers at many levels. They understand how the many levels are built one on top of another. Each of these levels corresponds to one or more courses in a computer science degree program.

QUESTION 2:

(Rhetorical) Does your understanding of computers depend on which assembly language you study?

Next

A good answer might be:

No. Probably a well-designed modern assembly language is best, but any one is OK.

Fundamentals

The MIPS architecture is modern and well-designed. MIPS chips were designed from the ground up in 1985. The design includes the best ideas of computer architecture.

An assembly language program describes exactly what the hardware should do, step by step, in terms of the basic operations of the computer. In a high level programming language like C or Java a programmer is mostly unaware of computer architecture. The same source program can run (after compiling) on any processor family.

These notes are about fundamental assembly-level computer architecture. To do this effectively, it is necessary (in my view) to acutally learn the assembly language and details of a particular processor, and to write programs for it. This is somewhat like those experiments you did in high school chemistry. Their goal was to teach you the fundamentals of chemistry, not to teach you how to make test tubes full of colorful water. But without the colorful experiments your understanding of chemistry might remain abstract and vague, and would soon be forgotten.

QUESTION 3:

At the time of this writing (Spring 2001) tens of thousands of dot.com workers have been laid off.

1. How many of them were making car payments on a Jaguar?
2. How many of them knew assembly language?

A good answer might be:

1. How many of them were making car payments on a Jaguar?

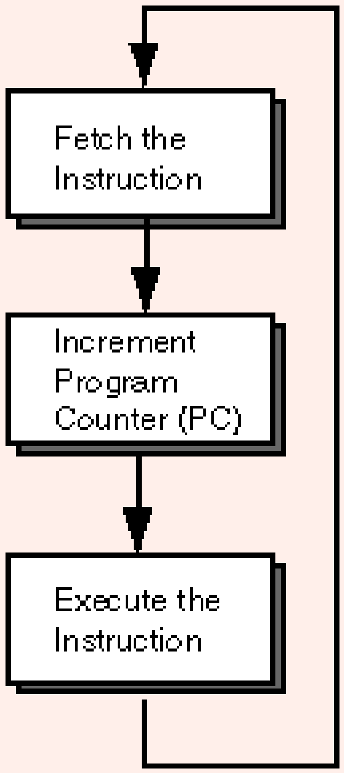
Must Sell! Make Offer!

   - Many of them.
2. How many of them knew assembly language?
   - Few of them.

Employment is still strong for computer scientists. It is weak for technicial workers. The used car lots of Silicon Valley are full of repossessed Jaguars (according to a recent news story).

Basic Machine Cycle

Most processors endlessly repeat three basic steps. Each **machine cycle** results in the execution of one *machine instruction*. A modern processor performs millions of machine cycles per second.



A **machine instruction** is a pattern of bits that corresponds to a fundamental operation of the processor, such as adding two 32-bit integers together, or testing if a value is zero. The details of the machine cycle differ between processor families. The machine cycle of most processor chips looks like the following:

> **Fetch the Instruction.** The *program counter* (PC) is part of the processor. It holds the address of the instruction. The machine instruction is fetched from memory.
>
> **Increment the Program Counter.** The program counter now holds the address of the next instruction.
>
> **Execute the Instruction.** The operations specified by the current machine instruction are performed.

QUESTION 4:

In what order are machine instructions executed by the basic machine cycle?

Next

In what order are machine instructions executed by the basic machine cycle?

A good answer might be:

Sequential order.

Machine Instructions

Instructions are normally executed in sequential order. Program loops and branches require machine instructions that alter the normal sequence. Operations like "add two integers" and "compute bit-wise or" are operations that a processor does in one machine cycle.

A machine instruction is a pattern of bits that directs the processor to perform one machine operation. Here is the machine instruction that directs the MIPS processor to add two 32-bit registers together (a register is a part of the processor that can hold a bit pattern).

```
0000 0001 0010 1011 1000 0000 0010 0000
```

The instruction is 32 bits long. Each bit is 0 or 1. When bit patterns are displayed in a book they are usually shown in groups of four (as here). Of course, the spaces are a convention of printing and don't really exist as part of the bit pattern.

Some of the bits in the instruction specify the operation (adding two registers), other bits specify which two registers to add. A few more bits say were to put the result.

QUESTION 5:

Inspect the instruction. Is it clear what the bits of the instruction mean?

Is it clear what the bits mean?

A good answer might be:

No.

Assembly Language Statement

The layout of a machine instruction is part of the architecture of a processor chip. Without knowing the layout you can't tell what the instruction is. Even if you know the layout, it is hard to remember what the patterns mean and hard to write machine instructions "by hand".

A statement in pure assembly language corresponds to one machine instruction. Assembly language is much easier to write than machine code. Here is the previous machine instruction and the assembly language that it corresponds to:

```
machine instruction                          assembly statement

0000 0001 0010 1011 1000 0000 0010 0000      add $t0,$t1,$t3
```

The instruction means: add the integers in registers T1 and T2 and put the result in register T0. To create the machine instruction from the assembly language statement a translation program called an **assembler** is used.

Humans find assembly language easier to use than machine language for many reasons.

- It is hard for humans to keep track of those ones and zeros.
- By using symbols programmers gain flexibility in describing the computation.
- Assembly language is a compact notation.

Enhanced assembly language includes additional convenience features. It has instructions (called **pseudoinstructions**) that correspond to several machine instructions. Often a **macroprocessor** is used to add even more features.

QUESTION 6:

Once you know assembly language, is it hard to write an assembly language statement?

A good answer might be:

No.

Program Translation

The assembly language statement says the same thing as the machine language instruction, but must be translated before it can be used. An assembly language *program* consists of assembly language statements, statements that define data, and some additional information that helps in the translation. Here is a fragment of an assembly language program and its translation into bit patterns.

```
machine instructions                      assembly statements

0011 0100 0000 0001 0000 0000 0000 1001       ori  $1, $0, 9
0000 0000 0100 0001 0000 0000 0001 1000       mult $2, $1
0000 0000 0000 0000 0100 0000 0001 0010       mflo $8
0011 0100 0000 0001 0000 0000 0000 1001       ori  $1, $0, 5
0000 0000 0100 0000 0000 0000 0100 1000       div  $8, $1
```

Years ago, to run a program written in FORTRAN you used a compiler to translate the program into assembly language. Then you used an assembler to translate the assembly language into machine language. Finally, you loaded the machine language into the computer's memory and executed your program.

Modern compilers usually translate a source program directly into machine language which is contained in a file called an **object module** (more on this later). But for now let us think about translating FORTRAN into assembly language.

FORTRAN is a high level language. It is intended to run on all types of computers, regardless of architecture. FORTRAN statements can be written without knowing the architecture of the computer, and can be run on any computer (after translation).

QUESTION 7:

Do you think that languages such as COBOL, C, and Pascal are translated like FORTRAN?

Next

A good answer might be:

Yes.

Several Translations

All programming languages other than machine language must be translated before they can be executed. A high level language is independent of architecture, but it requires a specific translator (compiler) for each architecture. The more modern the language, the more distant the source code is from the machine language. FORTRAN is old and is closer to machine language than most languages. Here is a statement in FORTRAN:

```
result = 6*alpha+beta
```

Here is a translation of that statement into MIPS assembly language:

```
lw $t0,alpha          # copy alpha to register T0
lw $t1,beta           # copy beta  to register T1
mul $t2,$t0,6         # multiply T0 times 6; result in T2
add $t2,$t2,$t1       # add T2 and T1; result in T2
sw  $t2,result        # copy answer to result
```

Here is a translation of that statement into Digital Equipment Corporation VAX assembly language:

```
MULL3   #6,ALPHA,R5
ADDL3   R5,BETA,RESULT
```

QUESTION 8:

Is the architecture of the VAX similar to that of MIPS?

Next

A good answer might be:

No. The two machines are very different.

Machine Language

There is not just one language called "assembly language." Each assembly language is for one type of processor only. MIPS assembly language is only for MIPS processors. The VAX assembly language is only for VAX processors. There is a different assembly language for IBM mainframe computers, and others for Intel-based PCs.

Assembly language describes computations *in terms of the hardware* of a particular processor. A high level computer programming language (ideally) describes computations *in terms of the problem being solved* . Since there are many types problems, there are many high level languages, each designed for particular types of problems. For example, object-oriented languages, describe computations in terms of the objects of the problem and operations with them.

It is much easier to program a computer in a high level language than in assembly language, especially when the programming language matches the problem. There will never be a universal programming language since no one language is best for all problems.

QUESTION 9:

(Hard thought question:) What type of problem is assembly language best suited for?

Next

(Hard thought question:) What type of problem is assembly language best suited for?

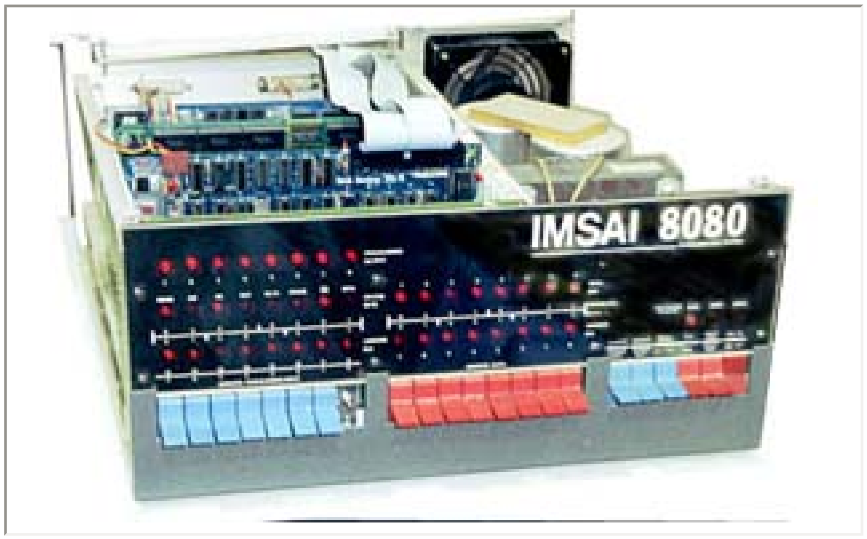
A good answer might be:

Problems that involve the very computer the program is running on.

Main Storage

Assembly language *does* match the problem when the problem is the operation of the computer system. Assembly language is used for operating systems, compilers, communications, low-level graphics, and other programs where the architecture of the processor must be visible. Often with these program the crucial parts are written in assembly and the rest in a high level language (usually C). The most common use of assembly language is in programming **embedded systems**. Here a processor chip is an integral part of a machine built for a specific purpose. Examples are microwave ovens, VHS players, automobile fuel systems, and game consoles. MIPS chips are commonly used in these (and for this reason more MIPS chips are sold each year than any other processor).

Now let us move on to the memory of a computer system. Eight bits make up a **byte**. A **bit** is a single on/off value. Early computers had rows of toggle switches which were used to set the values of bits in main memory. You could program these computers by entering the bits of each machine instruction. A light above each switch showed whether it was on or off. Modern computers have more convenient methods for moving bit patterns into memory.



A Beautiful Computer

The bytes that make up the machine instructions of a program are stored in **main memory** and fetched into the processor as needed. Data is also kept in main memory. Keeping both data and instructions in main memory is one of the characteristics of a **Von Neumann** machine, the name for the basic design of most modern computers.

In the MIPS, as in most computers, each byte of main memory has an address. An address is a integer that uniquely identifies the byte. The addresses run from 0 up to about four billion. However, on the MIPS, user programs and data (such as you will write) are restricted to the first two billion addresses.

QUESTION 10:

It it necessary to install four billion bytes of RAM on a MIPS computer? (Hint: the address range of a MIPS is the same as that of a Pentium).

Next

A good answer might be:

No. Both processors have the same range of logical addresses, but there need not be actual RAM behind each address.

Architecture vs. Implementation

The *architecture* of a computer is a logical description of its conponents and its basic operations.

The MIPS family of computers all have the same assembly-level architecture. This means that all MIPS machines can be programmed using the same assembly language. The actual electronics that implement the architecture may differ greatly between versions of the chip. This is analogous to "car architecture". Two Jaguars may appear the same to a driver (same styling, same user controls, same operation) but have different components under the hood. The architecture is the same, but the implementation (and performance) is different. Keep this in mind as you visit the car lots in Silicon Valley.

The architecture of a MIPS is different from the architecture of a Pentium. Both are VonNeumann machines, so they look the same in general, but the details of operation are completely different. They have different machine languages, and hence different assembly languages.

QUESTION 11:

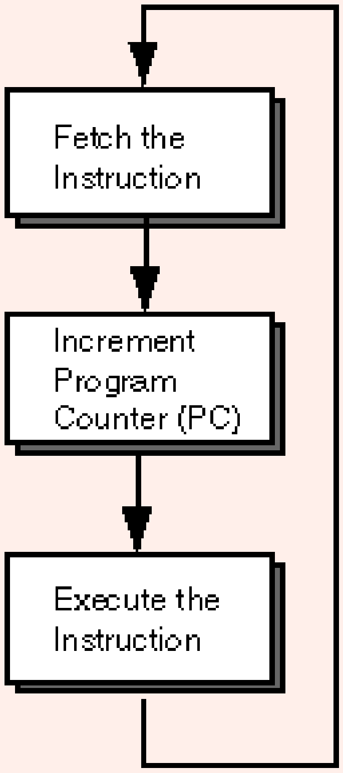
(Hard Thought Question:) Must a machine language program be run on an actual processor chip (i.e., on hardware)? Hint: Think about Java.

Next

A good answer might be:

No. Sometimes machine instructions (such as Java .class files) are *interpreted* by software.

Emulation



Apple computers can run Microsoft Windows ".exe" files (executable files) intended for Intel computers even though these machines have different architectures. This is done by running a program on the Apple that *emulates* (imitates) an Intel processor.

A processor's architecture is a logical description of its components and its basic operations. This says nothing about how the architecture is implemented. The operations need not be done in electronics. They can be done with software. A program that implements a processor's architecture is the logical equivalent of an implementation in silicon. Such a program is called an **emulator**. Any computer can run an emulator of any kind of computer. Of course, somebody has to write the emulator, and that is difficult.

The basic flow chart of an emulator program is the machine cycle of the emulated processor. But now the steps are implemented in software, not in silicon. The emulated machine cycle involves perhaps thousands of program statements. When the emulator is running, each simulated machine cycle requires the execution of many instructions on the host processor. So emulators run much slower than the processor they are emulating.

This course uses an emulator for the MIPS architecture. Using an emulator has many advantages over using the actual hardware. The biggest advantage is that the architecture described in these notes will exactly match the architecture implemented by the emulator. You can download exactly what you need to use to follow these notes. The emulator runs on most common desktop computers; PC, Mac, or ... MIPS.

QUESTION 12:

Is it possible to run an emulator for a MIPS processor on an actual hardware MIPS processor?

Next

A good answer might be:

Of course. Say that the emulator is written in C. Compile the emulator and run it on a MIPS computer. Now the program running on the MIPS is a MIPS emulator. This is a very computer-science-like notion.

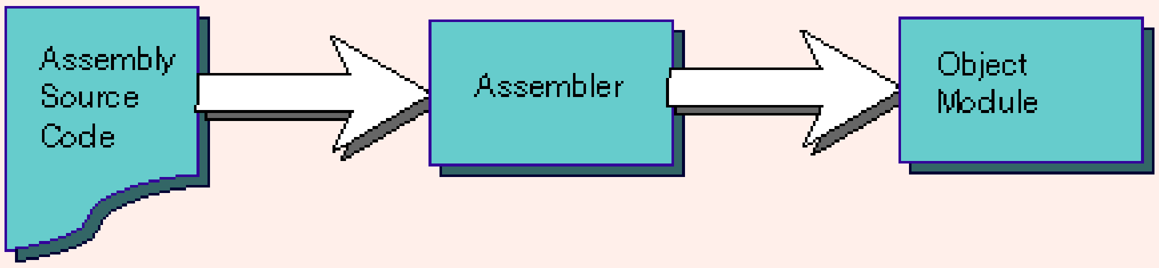
Separate Translation

An emulator running on the processor it is emulating is often used for debugging. Working with an emulator is usually more convenient than dealing with the hard silicon of the actual hardware.

A true computer nut (a "hacker", in the correct sense) will wonder if an emulator of (say) an Apple can run on an emulator of (say) an Intel machine that is running on a MIPS machine. Yes, as long as the various emulators exist.

For very many years (in computer years) assembly language was used for serious enterprise-wide programming for one reason: separate source files could be individually assembled, and later combined (linked) into the full executable. Each source file was assembled into an object module. This is called **separate assembly** (or *separate translation*).

An **object module** is a file that contains a machine language program that is not quite ready to run. It may not be ready to run for several reasons: the source code might not be completely translated (since more information is needed), memory locations for code and data might not be assigned, and more code is needed before the program is complete. An assembler creates an object module for each source file of assembly language.



QUESTION 13:

(Software Engineering Question:) Why translate many individual source files? Why not have one, big, complete program that translates into a complete, ready-to-run executable?

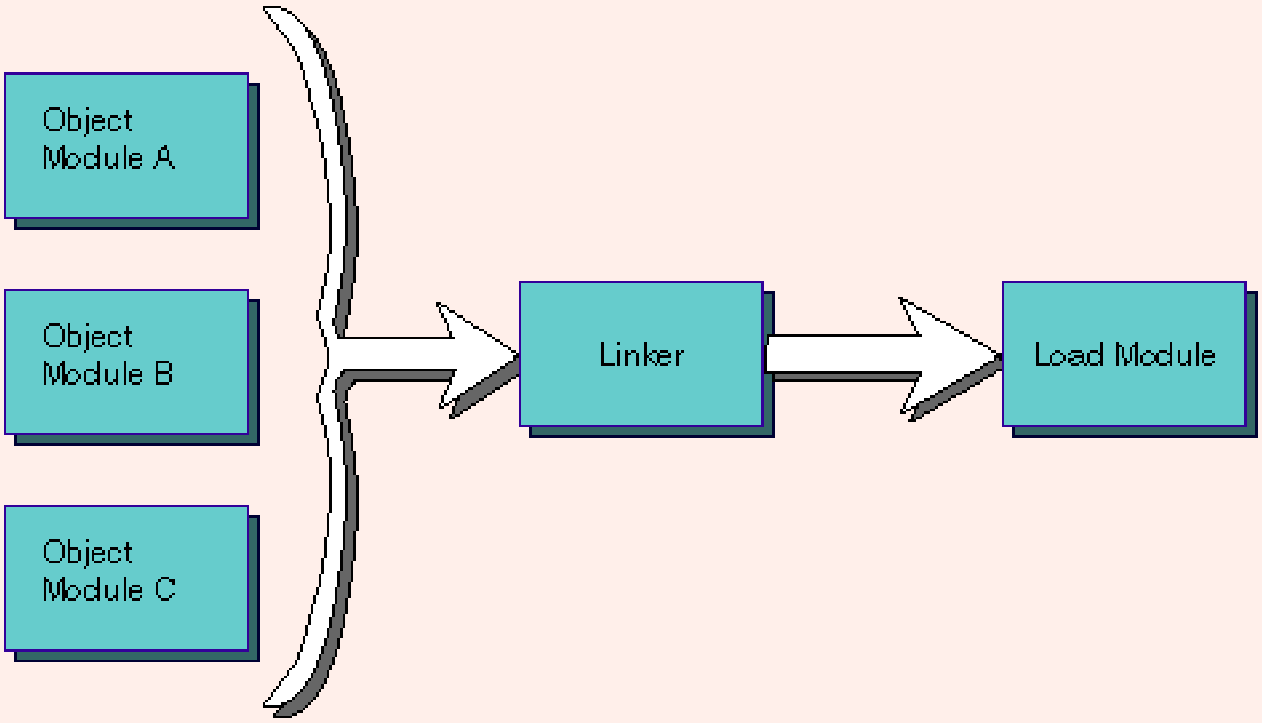
Next

A good answer might be:

Enterprise-wide programs are hundreds of thousands of lines long and take hundreds of programmers and support staff. It just would not work if they were all working on one big monster source program.

Load Modules

In assembly language the pieces of a big application program can be written as separate source files, and assembled into separate object modules. The separate object modules can then be combined by a linker into a **load module**, an executable file.



This is how all modern software of any appreciable size is written. The object modules do not need to be created from assembly language source. A compiler creates object modules that look the same as those from the assembler. In the picture, object module 1 might be from an assembly program, object module 2 might be from a FORTRAN program, and object module 3 might be from a COBOL program.

For many years, the only common languages that supported separate translation were Assembly, FORTRAN, and COBOL. For many years, the only common languages used in production were Assembly, FORTRAN, and COBOL. BASIC did not support separate compilation; BASIC was never used for writting big applications. Pascal did not support separate compilation; Pascal was never used for writting big applications. C supported separate compilation; C is now the dominant language for writting applications.

QUESTION 14:

C compilers come with libraries of useful functions. The *standard math library* contains functions like *sin*, *cos*, and *rand*. What do you suppose a software library is?

Next

A good answer might be:

A library is a collection of object modules which can be linked to the object module of your program.

End of the Chapter

You may wish to review the following terms. Click on the hypertext phrase to link to the the lesson where the term was discussed. To return to this list, use the "back arrow" on your web browser.

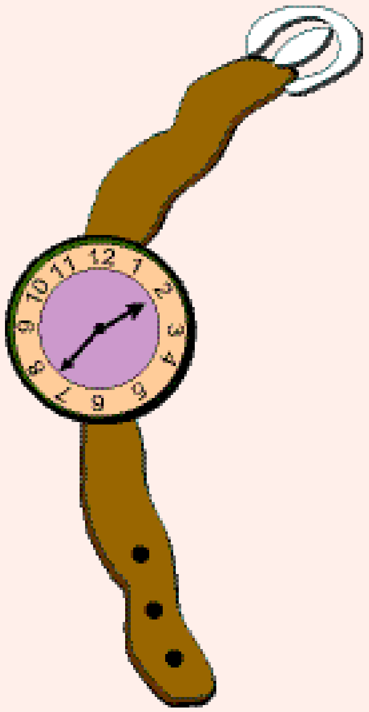
- Computer architecture
- The basic machine cycle, and machine instructions.
- Assembly language
- Pseudoinstructions.
- Program translation
- Main Memory.
- Uses of assembly language
- Emulation of a processor.
- Object modules and separate assembly.
- Load modules

contents

You have reached the end of the chapter.

revised: 01/18/99; 01/16/00; 09/01/01

CHAPTER 2 — Analog and Binary Signals



Note: This is a copy of chapter 2 of the on-line java tutorial located at http://chortle.ccsu.edu. If some of this is new, you might benefit from reading the first four chapters of that tutorial.

Everyone knows that computers are "digital" and that they use "binary" numbers. You probably have some idea of what these terms mean. But we need to be clear about their precise meaning and to be clear about why computers *are* digital.

Chapter Topics:

- *Binary signals.*
- *Analog signals.*
- *Advantages of Binary.*
- *Bits.*
- *Noisy signals.*

Study these topics carefully if you want to know what is really going on with computers.

QUESTION 1:

Why is a digital wristwatch called "digital?"

Next

A good answer might be:

Because it shows time using definite digits, rather than positions of hands that have to be measured. (It would be less correct to say "because it has a computer inside.")

Binary

Binary means "two states." The two states are sometimes called "1" and "0", or called "true" and "false", or called "on" and "off", (or other names.) The essential characteristic is that a single binary device can be in just one of two possible states.

A **bit** is a single "on/off" value.

A good example is a toggle switch, such as a light switch. You can turn it "on" or "off" but not (in normal operation) anything else. A light switch holds one bit of information.

A light dimmer is not a binary device: it has many positions between "off" and "fully on". If you want a light dimmer in to be set to 25%, you must carefully adjust it.

QUESTION 2:

Which of the following is a binary device:

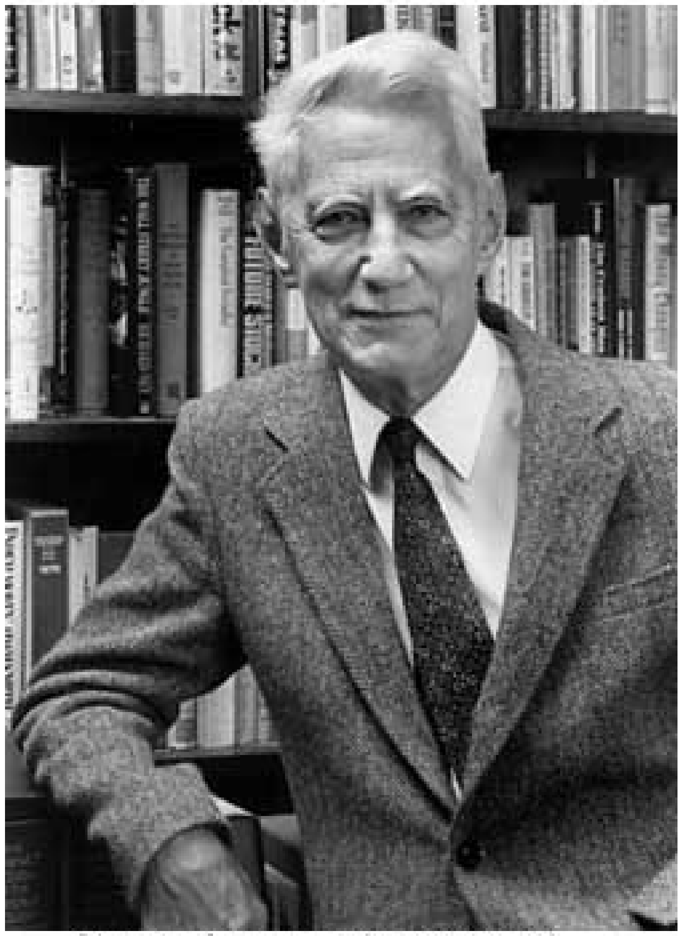
- The ignition switch of an automobile.
- The hour hand of a clock.
- A button on a hand calculator.
- The volume control on a stereo.

Next

A good answer might be:

- The ignition switch of an automobile. No
- The hour hand of a clock. No
- A button on a hand calculator. Yes
- The volume control on a stereo. No

Why Computers use Binary


Claude Shannon (1916-2001)

The ignition switch of an automobile is **discrete**—it has definite states—but there are usually more than two states (off, running, start, acc...) A button on a hand calculator is a binary device. It is either on or off. Ordinarily it is "off." When you push it, it is "on." It springs back to "off" when you release it, which is different from a toggle switch, but it still is a binary device.

There are many advantages to binary. Here are four (somewhat overlapping) important reasons for using binary:

1. Simple; easy to build.
2. Unambiguous signals (hence noise immunity).
3. Flawless copies can be made.
4. Anything that can be represented with some sort of pattern can be represented with patterns of bits.

These characteristics of binary were realized by Claude Shannon, a mathematician at Bell Telephone Laboratories. His 1948 paper *A Mathematical Theory of Communication* is the foundation of information theory and much of computer science.

QUESTION 3:

Which is easier to build:

Next

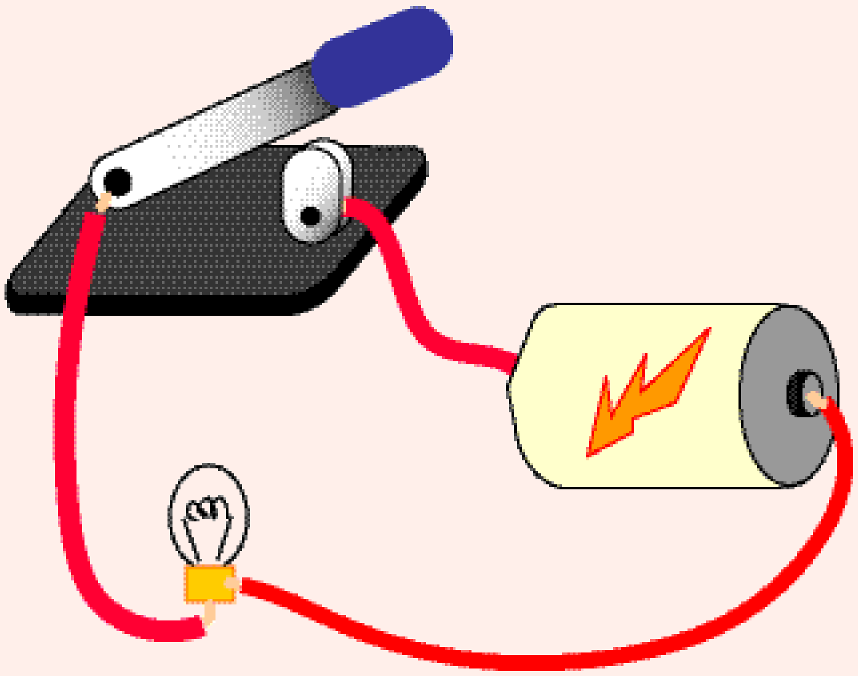
- An on/off switch.
- A light dimmer.

A good answer might be:

- An on/off switch.

Advantages of Binary 1: Simple; easy to build

An on/off switch is simple and easy to build. An on/off switch moves two pieces of metal together or moves them apart. A light dimmer must gradually and smoothly change the current that reaches the light. It has more components that an on/off switch and must be carefully assembled. An accurate dimmer (where 25% means *exactly* 25%) is even harder to build.



The same is true for the tiny devices inside of a silicon chip. On/off switches are relatively easy to fabricate. *Easy to build* means that the devices are cheap, small, and reliable, and millions of them can be put in a small area.

QUESTION 4:

Which of the following methods for producing books is simple and easy?

Next

- Each book is individually made by scratching marks onto tablets of wet clay.
- Each book is individually made by copying the characters onto paper with pen and

ink.

- Many copies of the book are made by carving flat wooden blocks with the text then printing the text on paper.
- Many copies of the book are made by setting movable type and printing the text on paper.

A good answer might be:

- Setting type.

Advantages of Binary 2: Unambiguous Signals

Carving a block of wood into a book page requires high technical and artistic skill. The task is complex and labor intensive. A small mistake might ruin the whole block. The Renaissance woodcarvers were as skilled as woodcarvers ever have been.

Setting type is comparatively easy. Little artistic or technical skill is needed. Individual pieces of type are just dropped into place. The method is robust and reliable; mistakes can be corrected. The type can be reused many times.

It is a paradox that advances in technology often lead to simple methods. Woodcarving is complex; setting type is easy. The computer revolution (ca. 1950—) is often compared to the Gutenberg revolution (ca. 1450—).

Review the advantages of binary:

1. Simple; easy to build.
2. Unambiguous signals (hence noise immunity).
3. Flawless copies can be made.
4. Anything that can be represented with patterns can be represented with patterns of bits.

QUESTION 5:

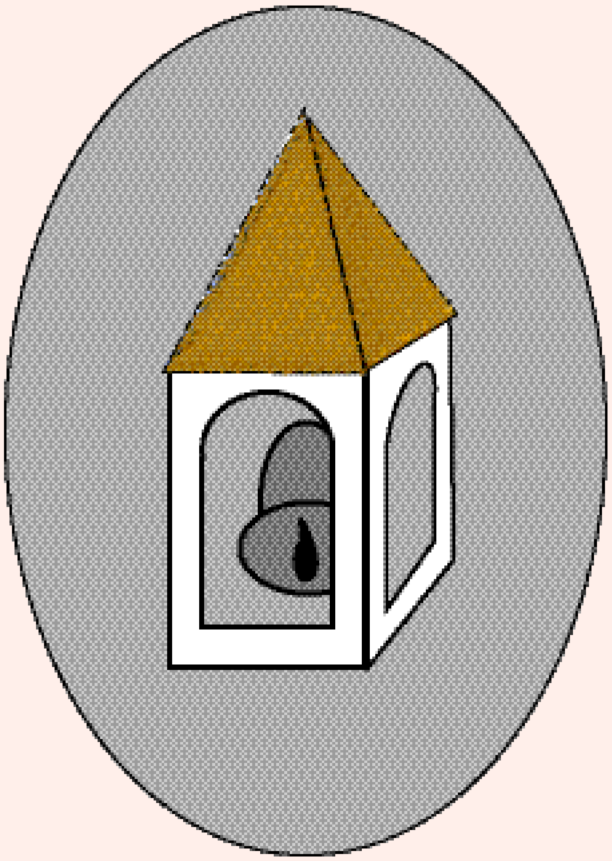
Which is easier to do: (a) determine exactly how bright a light is, or (b) decide if it is on or off?

Next

A good answer might be:

On or Off.

Old North Bit



Unambigous Signals: Consider Paul Revere, waiting for news of the attack of the British troops. He is expecting to see a signal lantern in the tower of Old North Church telling him how the British are attacking:

> 1.32456 if by land, 1.71922 if by sea.

The signal shines out! ...and Paul Revere's famous ride is delayed for several hours as he tries to figure out just how bright that signal is.

Lack of ambiguity is a tremendous advantage. The signal that Paul Revere was actually waiting for that night in 1775 was:

> One (lantern) if by land, and two (lanterns) if by sea.

...an easily interpreted signal. All Paul Revere had to do was count. Such signals are called **discrete** because they have a fixed number of definite states. Sometimes the word **digital** is used to mean the same thing.

QUESTION 6:

Why can a Chinese abacus be regarded as a "digital computer."

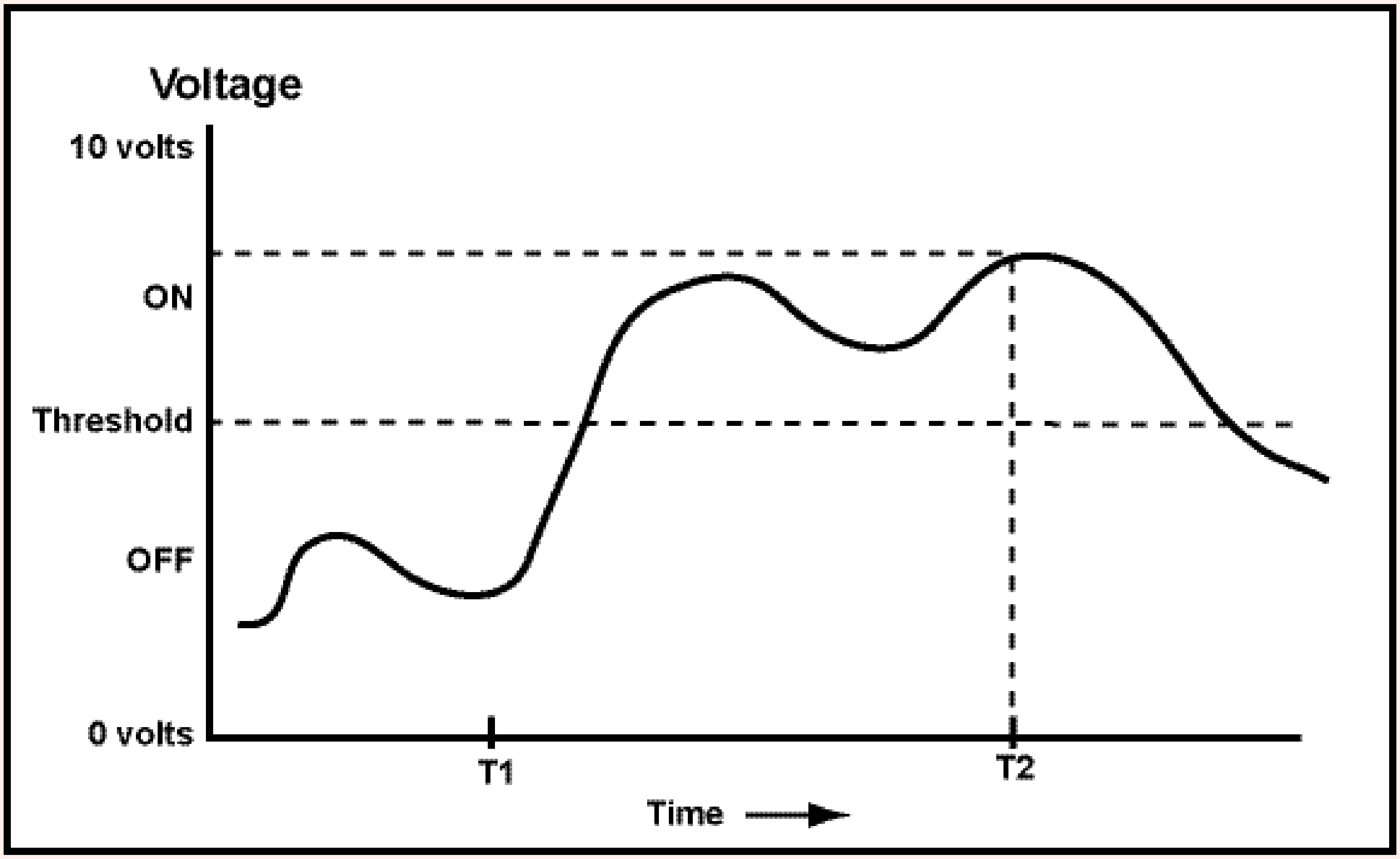
Next

A good answer might be:

The computation is done by moving beads into definite positions. A bead is either placed where it counts as part of a number, or it is placed where it does not count. This "all or nothing" operation means that an abacus is a discrete device.

An Analog Signal

An **analog signal** may continuously change in value. Its values can be anything within a range of values, and its exact value at any time is important. The graph below represents an audio signal. The exact value at each time is part of the information it contains. For example, the value at time "T2" must be measured exactly.



Now say that you are observing the voltage of a wire. It has been agreed that any voltage below a **threshold** will be counted as an "off" signal, and that any value above the threshold will be counted as an "on" signal.

QUESTION 7:

At time "T2", is the signal "on" or "off"?

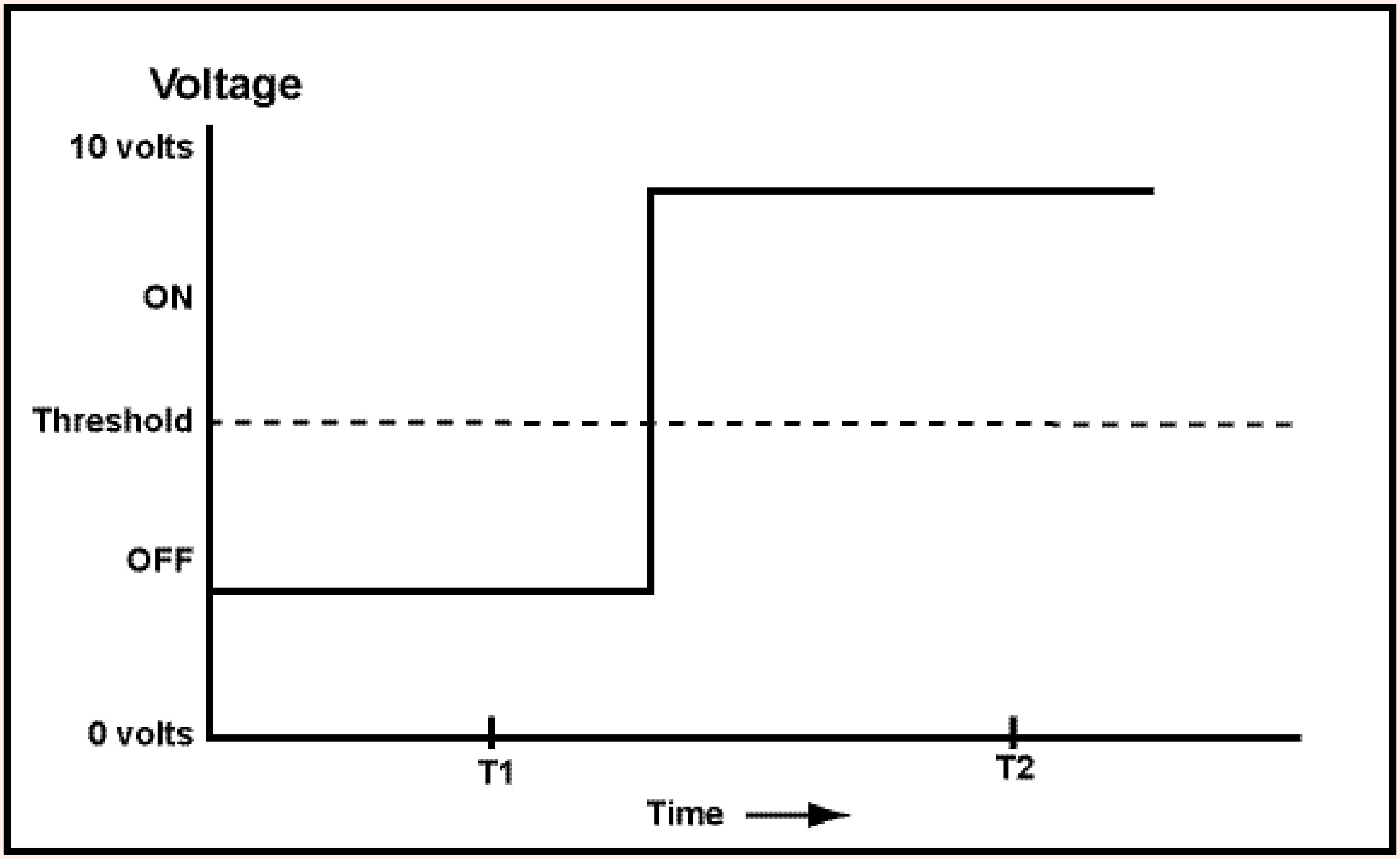
Next

A good answer might be:

ON. The analog signal (continuously changing voltage signal) moves up and down, but at time "T2" it is clear that it is above the threshold. Exact measurement is not needed.

Binary Signal

Analog signals continuously change in value. By using a threshold, analog signals can represent binary data ("on/off" data). It is easy and fast (for electronics, as well as for humans) to determine if a voltage is above or below a threshold. The figure shows a signal that transmits "off" then "on". (The signal is examined at times T1 and T2.)



QUESTION 8:

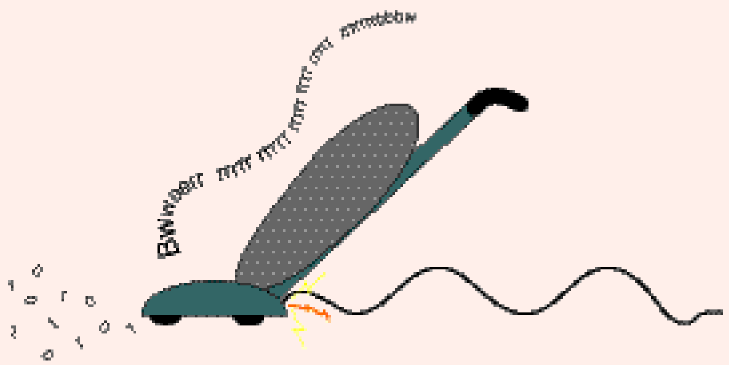
Is is clear that the signal is "off" at time T1 and "on" at time T2?

Next

A good answer might be:

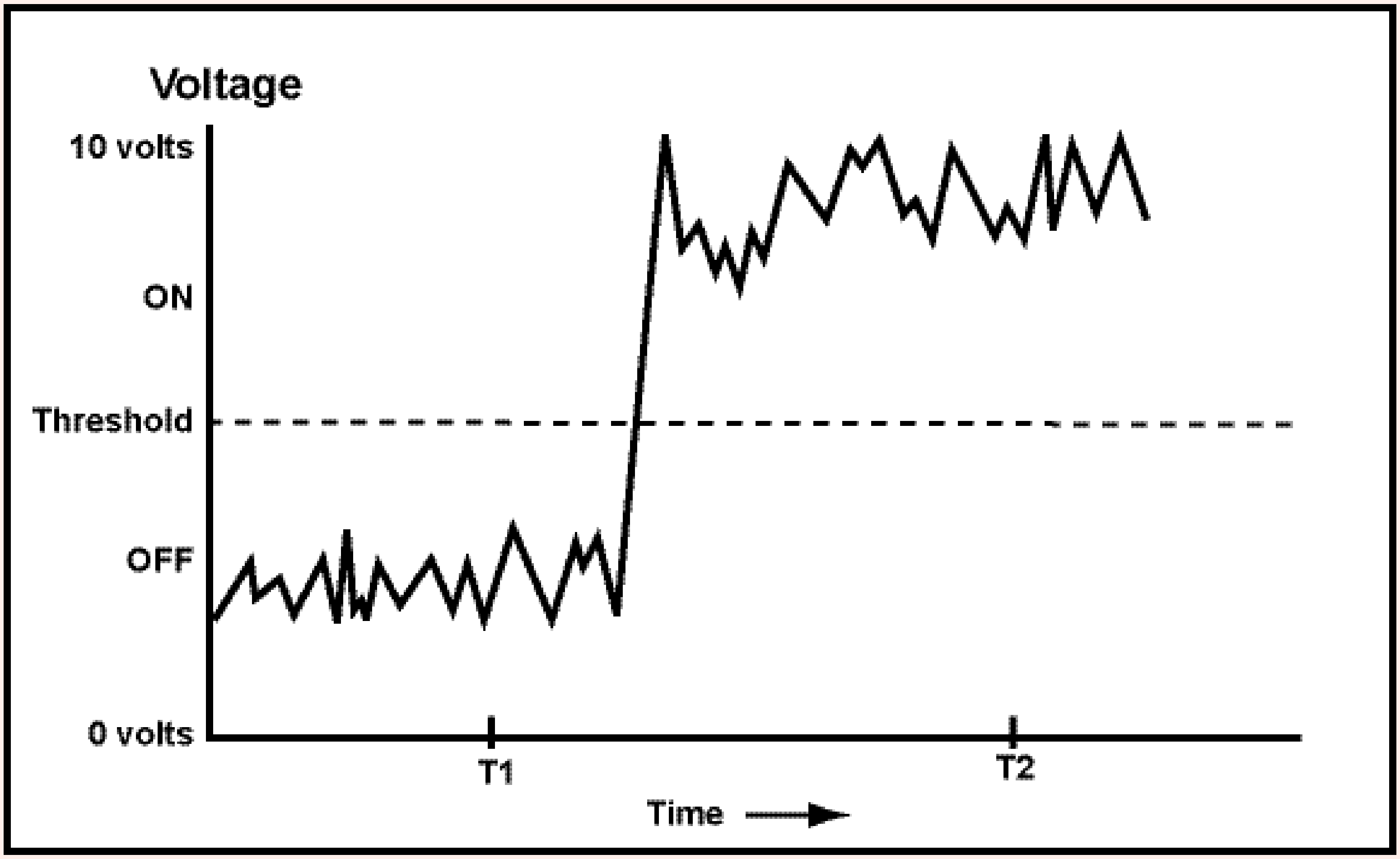
Yes.

Imperfect Transmission



The "ons" and "offs" of previous signal are clear. But what if the signal is sent down a long wire and someone nearby turns on a vacuum cleaner? The graph shows the signal at the other end of the wire.

Even though the signal is noisy (at the analog level), the binary values are transmitted perfectly. You (and the electronics) can still tell that at time T1 the signal represents "off" and that at time T2 the signal represents "on" The receiving end just needs to get the binary values.



Since only the "on" "off" information matters, the analog noise is irrelevant, and the original signal is received perfectly (so far as the binary information goes.)

QUESTION 9:

If the signal were regarded as an analog signal, and exact values were important, would some information present in the first signal (before the noise) have been lost?

Next

A good answer might be:

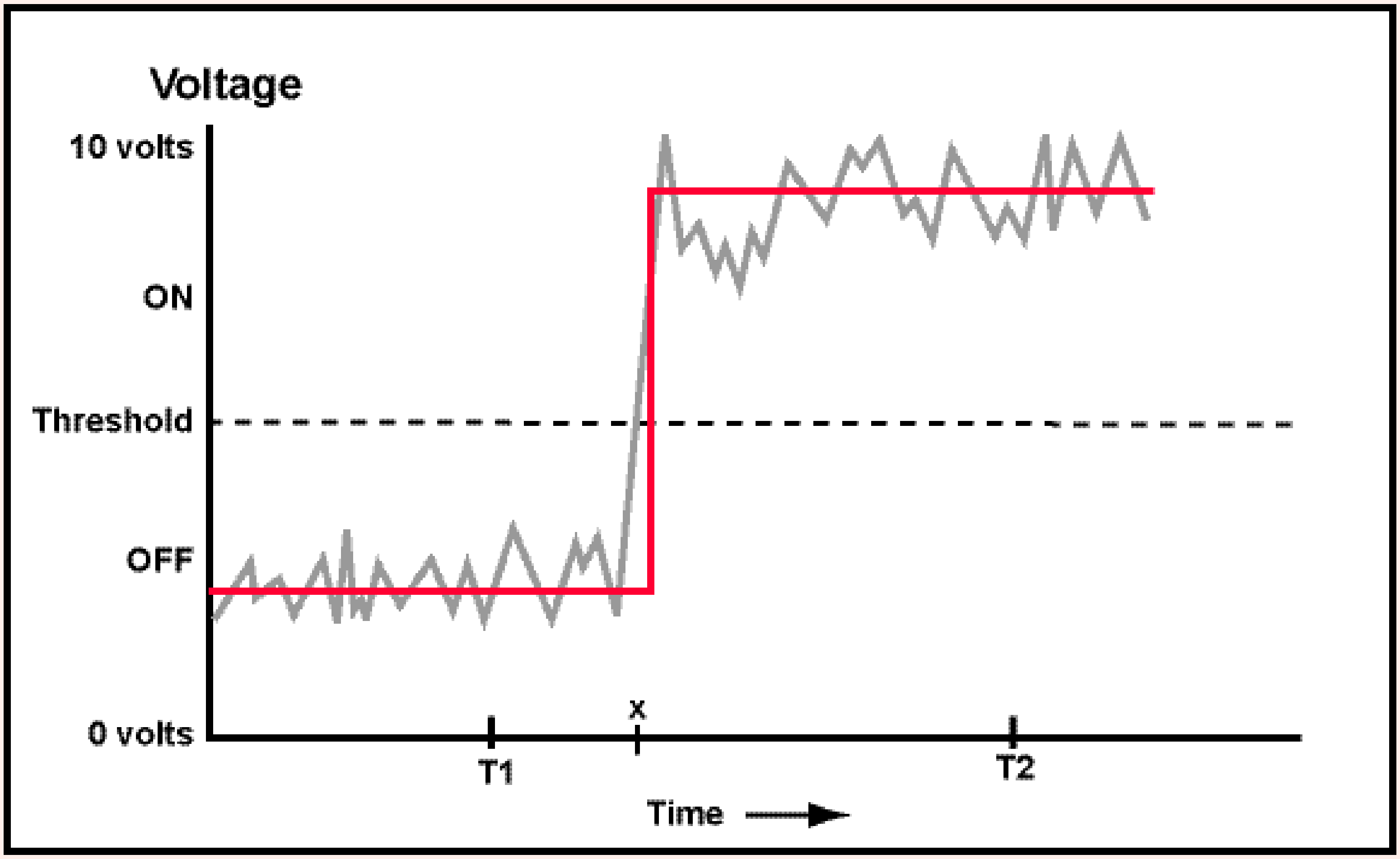
Yes—if the signals represented a singer's voice, the noisy signal would sound, er.. well... noisy. After just one copy, information has been lost.

Advantages of Binary 3: Flawless copies can be made.

Review the advantages of binary:

1. Simple; easy to build.
2. Unambiguous signals (hence noise immunity).
3. Flawless copies can be made.
4. Anything that can be represented with some sort of pattern can be represented with patterns of bits.

Flawless copies can be made: The receiving end of the signal is only interested in the binary values. All it has to do it check if the signal is above or below the threshold. This can be done perfectly (as long as the noise is not too great.) For example, here is the noisy signal with the "on"/"off" values recovered from it:



The original signal has been recovered flawlessly. This process can occur as many times as needed with a perfect copy made each time. This is essential in a computer system, where bit patterns (patterns of one and zero, or on and off) are copied back and forth between the processor and memory millions of times a second. The copies have to be perfect.

QUESTION 10:

Something might be wrong here. Is the signal is "on" or "off" at the time "x"? This is not easy to tell. What is worse, a different amount of noise changes the answer. What can be done about this problem? (Hint: must the value of the signal be known at all times?)

Next

A good answer might be:

The system is built so that the signal is tested only at particular times, and that changes in the signal occur between these times.

Clocks

Digital systems are built so that the "on" "off" (binary) value is only tested at certain times, giving the wire (or transistor, or...) a chance to change its state between these times. This is why computer systems have a "clock" — to keep all these times synchronized. So faster clocks mean wires can be tested more times per second, and the whole system runs faster.

Processor chips (and the computers that contain them) are often described in terms of their clock speed. Clock speed is measured in **Hertz**, where one Hertz is one clock tick per second. The symbol **MHz** means **mega Hertz**, a million clock ticks per second.

A 700 MHz Pentium processor checks binary values 700 million times in each second. In between these times values are allowed to change and settle down. The faster a processor chip is, the more times per second values can be tested, and the more decisions per second can be made.

QUESTION 11:

Which is faster, a 400 MHz Pentium processor or a 800 MHz Pentium processor?

Next

A good answer might be:

The 800 MHz processor, since it checks values 800 million times per second (twice as many as the 400 MHz processor).

(However, if the processors are different types there are other factors besides clock speed that determine how much each processor can do in a second.)

Advantages of Binary Representation 4: Representing Anything

Recall that last advantage of binary:

- Anything that can be represented with patterns can be represented with patterns of bits.

Since data of all kinds is stored in computer memory (main and secondary) using the same electronic methods, this means that endless perfect copies can be made of any type of data or program.

This idea is that any system of symbols can be translated into bit patterns. An example is how English characters are represented as eight-bit patterns. The agreeement about what patterns represent what characters is called ASCII. The hardware and the software of a computer system (usually) follow this agreement when the data is "text". (You will learn more about this later). Other types of data (non-character data) are represented using other methods.

QUESTION 12:

Can charaters from alphabets other than English be represented?

Next

A good answer might be:

Of course. Any type of symbol at all can be assigned a binary pattern to represent it.

More on Representing Anything in Binary

Japanese and Chinese characters, also, have been translated into bit patterns, and computers can manipulate those symbols just as easily as ASCII. Unicode is an agreement created by an international committee on how to represent characters using 16 bits. Here are 16 bits 111110011111110 and here is the character it represents in Unicode: 茶

Say that the international committee decides to represent a new Chinese character. How can they do this? Easy: they find a bit pattern not yet used to represent any symbol and assign the new character to that pattern.

The correspondence between human language symbols and bit patterns is arbitrary. All you have to do is be sure that you are using enough bits so that all the symbols of the language have a unique bit pattern to represent them.

QUESTION 13:

Can printed music be represented using binary?

A good answer might be:

Sure — any symbols can. There are "music processor" programs for printed music that work like word processor programs for printed text.

Symbols and Patterns

Recall that last advantage of binary:

- Anything that can be represented with patterns can be represented with patterns of bits.

What about representing things other than the written characters of a language? This is a deep topic, and entire books have been written on it. Here is a very sloppy discussion: Pick some subject. Use English sentences to describe it. Represent those sentences in ASCII (characters encoded as byte patterns.) Now the subject is represented in binary. If something can be represented in English, then it can be represented in binary.

Notice that this says nothing about "meaning" or "understanding." Printed books don't understand their own contents. A digital version of the book (say on CD ROM) doesn't understand the book, either. It mearly holds the information, waiting for a human mind to do the understanding. However the book has been represented as bit patterns.

Nobody said that binary representations are easy to use. Some representation methods are very useful for computers (for instance, using binary patterns to represent numbers), others are nearly usless. Much of the work that computer scientists do is figuring out how to represent things using binary in ways that are useful. For example, much work has been done in the last ten years in figuring out how best to represent image and audio data.

Important Point:

All that computer memory (and other parts) holds is bit patterns. What those bit patterns represent depends on how they are used.

QUESTION 14:

Sometimes people say, "All that a computer can handle is numbers. It doesn't understand anything else." Do you think that this is correct?

Next

A good answer might be:

No. Any type of symbolic data can be represented using binary, not just numbers. At the electronic level, everything is a binary pattern (which some people call a "number"), so the statement is sort-of correct.

End of this chapter.

You may wish to review the following terms. Click on the hypertext phrase to go the page where the term was discussed.

- What "binary" means.
- What a "bit" is.
- The advantages of using binary in a computer.
- What "analog" means.
- How thresholds are used.
- The master clock of a computer.
- Memory contents as bit patterns.

contents

Back to the main menu.

You have reached the end of the chapter.

created: 06/25/96; 2nd edition: 05/13/01; little bitty changes: 09/01/01

CHAPTER 3 — BITS and BIT PATTERNS

Computers represent data and instructions with patterns of bits. You must become familiar with bit patterns! This chapter will help you. It discusses the fundamentals of bit patterns.

Chapter Topics:

- *Patterns of bits.*
- *The number of patterns that can be formed for N bits.*
- *How to systematically list all the patterns for N bits.*
- *Multiplying powers of two.*
- *Bytes, kilobytes, megabytes, and gigabytes.*
- *Names for four-bit patterns.*
- *Hexadecimal names for bit patterns.*
- *Octal names for bit patterns.*

In most computer documentation, 8 contiguous bits are called a **byte**. A bit holds a zero or a one, possibly representing the on/off condition of a switch.

QUESTION 1:

How many patterns can be formed with a single bit?

Next

How many patterns can be formed with a single bit?

A good answer might be:

Two • 0 and 1 (or "off" and "on", or "false" and "true", or ...)

Patterns of Bits

A bit can be 0 or 1. With one bit there are two possible patterns. How many patterns can be formed with two bits? Here is a complete list:

```
0 0
0 1
1 0
1 1
```

Looks like 4 patterns.

QUESTION 2:

Is the pattern 0 1 different from the pattern 1 0?

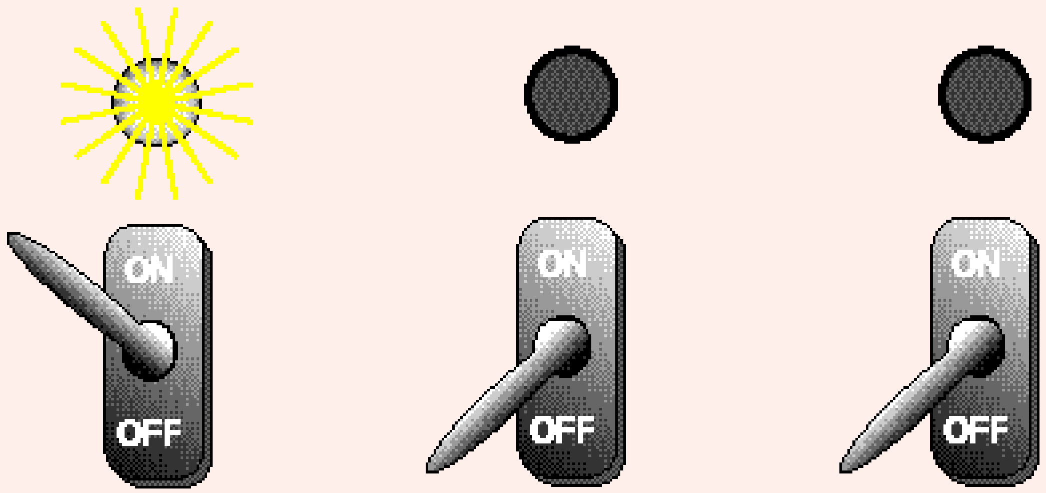
Next

A good answer might be:

Yes • the position of a bit matters.

How Many Patterns with Three Bits?

How many patterns can be formed with three bits? Let's list them:



```
0 0 0
0 0 1
0 1 0
0 1 1
1 0 0
1 0 1
1 1 0
1 1 1
```

Looks like 8 patterns.

QUESTION 3:

Is the number of patterns that can be formed with N bits greater than the number of bits?

Next

Is the number of patterns that can be formed with N bits greater than the number of bits?

A good answer might be:

Yes • much greater. This simple fact is of fundamental importance to computer science.

Listing Patterns Systematically

There is a standard method for listing all of the patterns that can be formed with a given number of bits. First, list all of the patterns with one bit.

```
0
1
```

When you increase the number of bits by one (from one to two) make two copies of the first list:

```
0
1

0
1
```

Within each copy, each row is unique. Now, make unique each row in the combined list. Put "0" in front of each line of the first copy, and put "1" in front of each line of the second copy:

```
0 0
0 1

1 0
1 1
```

Now each line is unique and you have a complete list of the possible patterns. The number of unique patterns with 2 bits is double that with 1 bit.

For additional bits, repeat the trick for each new bit. With three bits, make two copies of the two bit list, put "0" in front of the lines of the first copy and "1" in front of the lines of the second copy. Now you have a list containing all possible patterns of three bits.

QUESTION 4:

How many patterns can be formed from three bits?

A good answer might be:

8 patterns can be formed from three bits.

How Many Patterns?

The three bit list has 8 lines (patterns). Two copies of it gives you 16 lines. Each line is made unique by prefixing the first copy with "0" and the second copy with "1". The table (below) shows the number of patterns for 1, 2, 3 and 4 bits.

Number of Bits	Number of Patterns	Number of Patterns as power of two
1	2	2^1
2	4	2^2
3	8	2^3
4	16	2^4

How many patterns with 5 bits? Make two copies of the 4-bit patterns (16 patterns per copy). Make the patterns unique by prefixing "0" to the first 16 patterns and "1" to the second 16. You now have 16×2 = 2^5 unique patterns. This demonstrates the following:

Number of possible patterns of N bits = 2^N

Memorize this fact. Better yet, make lists of patterns (as above) and play around until you understand. Do this now. It is a very useful formula. If you allow yourself to get muddled on it, you will waste much time in this and future courses.

How many patterns can be formed with 10 bits? Plugging into the formula,

$2^{10} = 1024$.

This number occurs often in computer science. 1024 bytes is called a **kilobyte**, abbreviated **K**and pronounced "Kay".

QUESTION 5:

In the past, some computers used 16 bits to form memory addresses. Assuming no special tricks (which such limited machines often used), how many bytes maximum could be held in main storage?

A good answer might be:

64K bytes: $2^{16} = 2^{(6 + 10)} = 2^6 \times 2^{10} = 64K$

More About Patterns

Many calculations involving bit patterns use the following familiar fact of arithmetic. (Although the fact is familiar, confusion is even more familiar. Be sure you know this factoid.)

$$2^{(N+M)} = 2^N \times 2^M$$

It is not too much work to extend the table, as shown at left. You can always make this table from scratch, but memorizing a few key values does not hurt.

Number of Bits	Number of Patterns	Number of Patterns as power of two
1	2	2^1
2	4	2^2
3	8	2^3
4	16	2^4
5	32	2^5
6	64	2^6
7	128	2^7
8	256	2^8
9	512	2^9
10	1024	2^{10}

The numbers of patterns that can be formed with 10 or more bits are usually expressed in terms of kilobytes (= 2^{10}) or megabytes (= 2^{20}). For example, how many patterns can be formed from 24 bits?

$$2^{24} = 2^4 \times 2^{20} = 16 \text{ Meg}$$

The power of two (24) splits into a small part (2^4) and a part that has a name (2^{20} = Meg). This is a useful trick you can use to amaze your friends and impress employers.

Some audio cards use 12 bits to represent the sound level at an instant in time (12 bits per sample). How many signal levels are represented?

$$2^{12} = 2^2 \times 2^{10} = 4\text{K levels}$$

QUESTION 6:

You have a choice between a graphics card that uses 6 bits to represent colors, or one that uses 8 bits to represent colors. The 8 bit board is somewhat more expensive than the 6 bit board. Which should you choose?

A good answer might be:

The 8 bit board, since it can represent many more colors. Assume that each bit pattern represents a color. With 6 bits, the board could only have 2^6 = 64 colors; with 8 bits, it could have 2^8 = 256 colors, a considerable improvement.

Pattern Names

Consider the following pattern:

```
0010100010101010
```

It is not easy to work with. It is convenient to break bit patterns into 4-bit groups (called **nibbles**):

```
0010 1000 1010 1010
```

There are 16 (= 2^4) possible patterns in a nibble; each pattern has a name, as seen in the table.

You might be tempted to call those 4-bit patterns "binary numbers". *Resist that temptation.* The bit patterns in computer main memory are used for very many purposes; representing integers is just one of them. The fundamental concept is "bit patterns". Don't confuse this concept with one of its many uses: "representing numbers".

Hexadecimal Names			
nibble	**pattern name**	**nibble**	**pattern name**
0000	0	1000	8
0001	1	1001	9
0010	2	1010	A
0011	3	1011	B
0100	4	1100	C
0101	5	1101	D
0110	6	1110	E
0111	7	1111	F

The above bit pattern can be written using the pattern names:

```
0010 1000 1010 1100  = 28AC
```

Bits are grouped into nibbles starting at the right. Then each nibble is named. This method of giving names to patterns is called **hexadecimal**.

QUESTION 7:

What is the name of the pattern: 1011 0110

Next

What is the name of the pattern: 1011 0110

A good answer might be:

B6

More Hex Practice

(Not as done at Mugworts)

Hexadecimal Names			
nibble	**pattern name**	**nibble**	**pattern name**
0000	0	1000	8
0001	1	1001	9
0010	2	1010	A
0011	3	1011	B
0100	4	1100	C
0101	5	1101	D
0110	6	1110	E
0111	7	1111	F

If there are not enough bits at the left to form a complete group of four, add zero bits *to the left*, (but be sure that it is clear by context how many bits you are describing). For example:

```
1010000010000010     =
1010 0000 1000 0010  =  A082
```

Another example:

```
10100110101111      =
10 1001 1010 1111   =
0010 1001 1010 1111 = 29AF
```

Usually '0x' is placed at the front of a pattern name to show that it is a hexadecimal pattern name:

```
0x0010  =  0000 0000 0001 0000
0xFACE  =  1111 1010 1100 1110
```

Please understand that *hexadecimal pattern names are used by humans for talking about bit patterns.* Inside the computer there are only bits and their patterns. Hexadecimal is used in books and documents (outside the computer) to describe these bit patterns.

QUESTION 8:

What is the hexadecimal pattern name of the following bit pattern:

01101010

A good answer might be:

```
01101010 = 0110 1010 = 6A
```

Zeros On the Left

Sometimes documentation describes bit patterns in groups of three. Three-bit groups are named using the first eight pattern names of hexadecimal. This method is called **octal notation** . A bit pattern can be named using hexadecimal names, octal names, or many other notations.

Octal Names			
nibble	**pattern name**	**nibble**	**pattern name**
000	0	100	4
001	1	101	5
010	2	110	6
011	3	111	7

```
01101010 = 01 101 010 =  152 (octal)
01101010 = 0110 1010  = 0x6A (hex)
```

Octal is awkward to use with 8-bit bytes. Bytes don't evenly split into octal pattern names. But you should know about it. Historically, some computer documentation used octal pattern names. Also, in several programming languages (C and Java among them) octal notation is signaled by a leading zero:

```
0152    (octal)    = 001 101 010
0x152   (hex)      = 0001 0101 0010
152     (decimal)  = 1001 1000
```

I have lost an unfortunate number of hours in total despair over disfunctional programs that could not possibly be wrong, only to discover a constant buried deep in the code that started with a "0" when it should not have.

Adding zeros to the left of a pattern creates a new pattern. The new pattern has its own name. 0x0 = 0000 is a different pattern than 0x00 = 0000 0000. Sadly, people are not consistent about this, and depending on context, both patterns might be called "0x0".

When the number of bits is not a multiple of four it is conventional to add zero bits to the left, and then to name the pattern as usual.

QUESTION 9:

What is the hexadecimal pattern name of the following bit pattern:

Next

10 1101 0101

A good answer might be:

0x2D5

End of the Chapter

If you are a bit muddled, you may wish to review some of the following. Click on a topic to go the the lesson where the term was discussed. Otherwise go out and get a nibble to eat.

- Byte
- Listing bit patterns
- Number of patterns with N bits
- kilobyte
- Multiplying powers
- Nibble Names
- Hexadecimal
- Octal

You have reached the end of the chapter.

created: 06/25/96; 2nd edition: 05/13/01

CHAPTER 4 — COMPUTER SYSTEMS

This chapter discusses how computer systems are organized, with special attention paid to main memory. Most of this you probably know already, but look it over anyway.

QUESTION 1:

Does a computer system based upon one type of processor chip (say MIPS) look about the same as a system based upon another type of chip (say Pentium)?

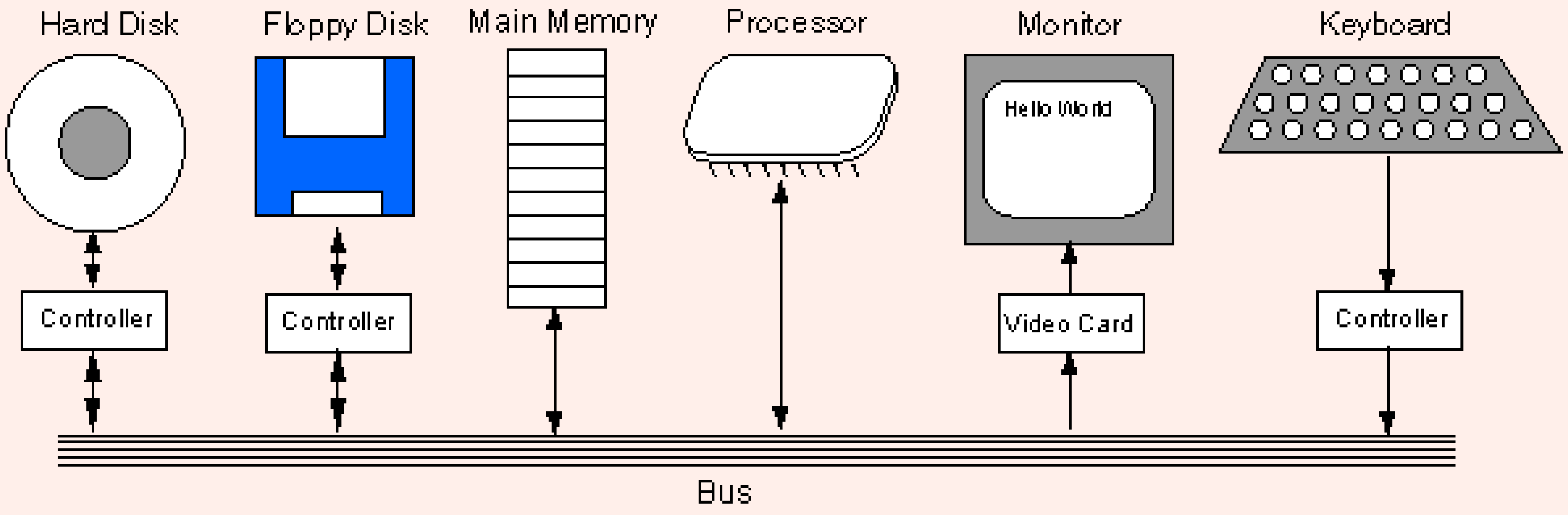
Next

A good answer might be:

Yes. The main components are about the same and work about the same way.

Computer System Components

The diagram is a general view of how desktop and workstation computers are organized. Different systems have different details, but in general all computers consist of components (processor, memory, controllers, video) connected together with a *bus*. Physically, a **bus** consists of many parallel wires, usually printed (in copper) on the main circuit board of the computer. Data signals, clock signals, and control signals are sent on the bus back and forth between components. A particular type of bus follows a carefully written standard that describes the signals that are carried on the wires and what the signals mean. The PCI standard (for example) describes the PCI bus used on most current PCs.



Main Components of a Computer System

The processor continuously exectues the machine cycle, executing machine instructions one by one. Most instructions are for an arithmetical, a logical, or a control operation. A machine operation often involves access to main storage or involves an i/o controller. If so, the machine operation puts data and control signals on the bus, and (may) wait for data and control signals to return. Some machine operations take place entierly inside the processor (the bus is not involved). These operations are very fast.

QUESTION 2:

Do you think that the various components can put signals and data on the bus at any arbitrary time?

Next

A good answer might be:

No. The various devices must cooperate somehow so their data and signals don't get mixed.

Input/output Controllers

The way devices connected to a bus cooperate is another facet of a bus standard.

Input/output controllers receive input and output requests from the central processor, and then send device-specific control signals to the device they control. They also manage the data flow to and from the device. This frees the central processor from involvment with the details of controlling each device. I/O controllers are needed only for those I/O devices that are part of the system.

Often the I/O controllers are part of the electronics on the main circuit board (the *mother board*) of the computer. Sometimes an uncommon device requires its own controller which must be plugged into a connector (an expansion slot) on the mother board.

QUESTION 3:

(Review:) Is there a difference between the memory used to hold programs and the memory used to hold data?

Next

A good answer might be:

No. Potentially any byte of main memory can hold part of a program or part of some data.

Main Memory

In practice, data and instructions are often placed in different sections of memory, but this is a matter of software organization, not a hardware requirement. Also, most computers have special sections of memory that permanently hold programs (firmware stored in ROM), and other sections that are permanently used for special purposes.

Main memory (also called *main storage*, or just *memory*) holds the bit patterns of machine instruction and the bit patterns of data. Memory chips and the electronics that controls them are concerned only with saving bit patterns and returning them when requested. No distinction is made between bit patterns that are intended as instructions and bit patterns that are intended as data. The amount of memory on a system is often described in terms of:

Kilobyte:	2^{10} = 1024 bytes.
Megabyte:	2^{20} = 1024 Kilobytes
Gigabyte:	2^{30} = 1024 Megabytes
Terabyte:	2^{40} = 1024 Gigabytes

Main memory these days (Summer 2001) ranges in capacity from 32 Meg to 512 Meg. Hard disks and other secondary storage devices are ten or hundreds of Gig. Backup storage comes in sizes as large as several terabytes.

QUESTION 4:

What is (on most computers) the smallest addressable unit of memory?

Next

A good answer might be:

A byte.

Addresses

Each byte of main storage has an **address**. Most modern processors use 32-bit addresses, so there are 2^{32} possible addresses. Think of main storage as if it were an array:

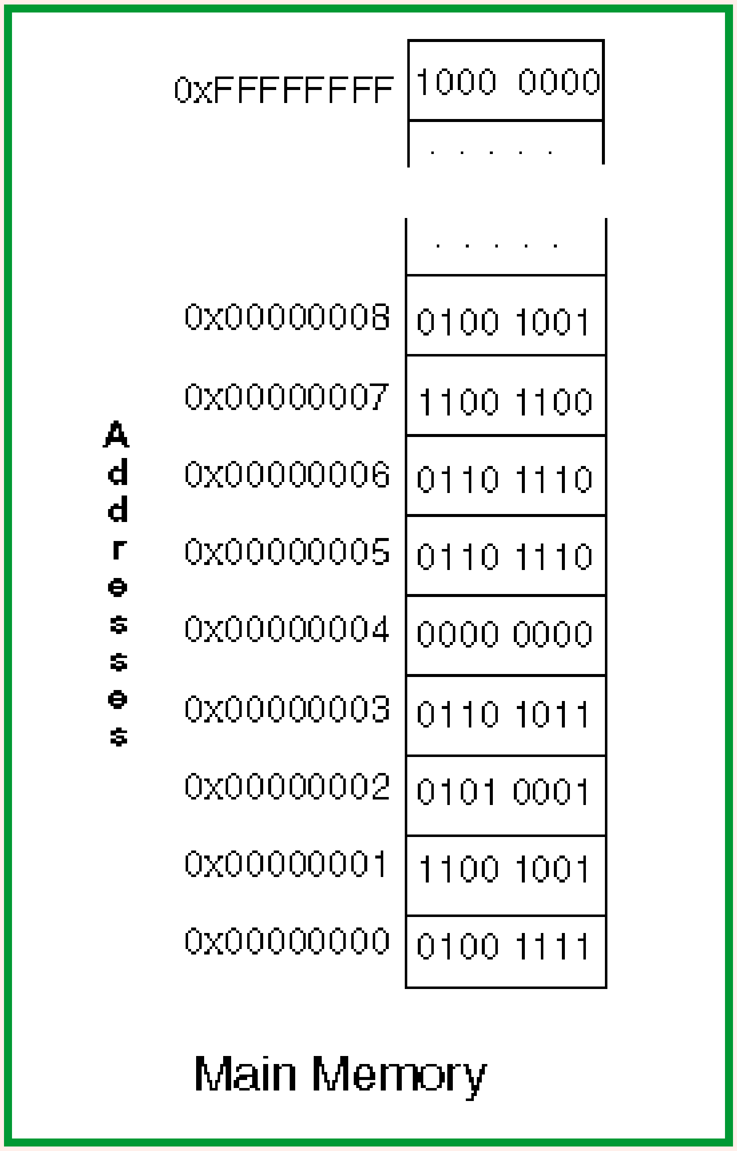
```
byte[0x00000000 ... 0xFFFFFFFF] mainStorage;
```

A main storage address is an index into memory. A 32-bit address is the address of a single byte. Thirty-two wires of the bus contain an address (there many more bus wires for timing and control).

Sometimes people talk about addresses like 0x2000, which looks like a pattern of just 16 bits. But this is just an abbreviation for the full 32-bit address. The actual address is 0x00002000.

Older MIPS processsors, the R2000 (designed in 1985) and the R3000, use 32-bit addresses. Recent MIPS processors (from 1991 to present) use 64-bit addresses. Processor chips from Intel will have 64-bit addresses in the near future.

The assembly language of this course is for the MIPS 3000 chip, so we will use 32-bit addresses. The assembly language of the recent MIPS chips is similar. For embedded systems programming, 32-bit addresses is often more than enough, so the 32-bit chips are still used.



QUESTION 5:

What is the hexadecimal name for the 32-bit pattern that consists of all 1 bits? (Hint: look at the picture.)

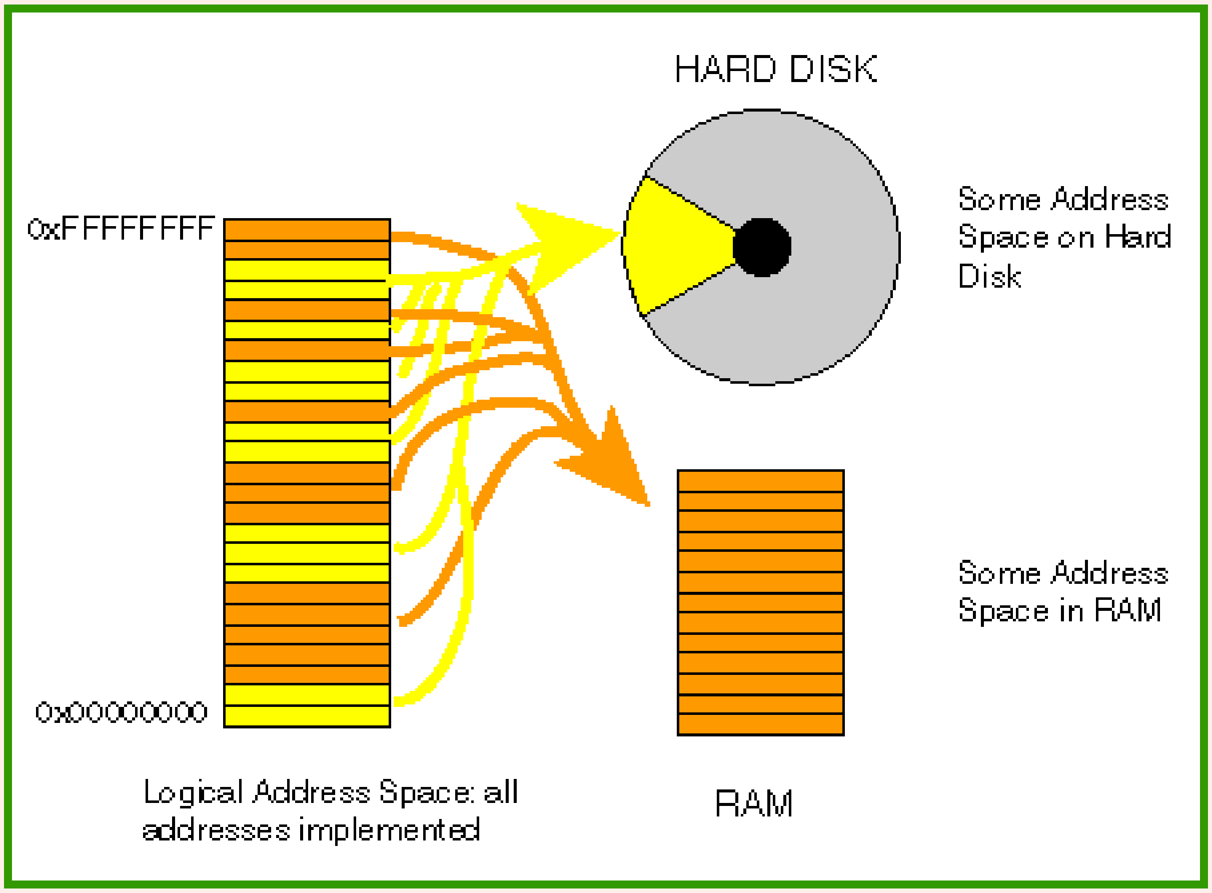
Next

What is the hexadecimal name for the 32-bit pattern that consists of all 1 bits?

A good answer might be:

0xFFFFFFFF

Virtual Memory


HARD DISK
Some Address Space on Hard Disk
0xFFFFFFFF
0x00000000
Some Address Space in RAM
Logical Address Space: all addresses implemented
RAM

The MIPS R3000 has an address space of 2^{32} bytes. A Gigabyte = 2^{30}, so it has 4 gigabytes of address space. Ideally, all of these memory locations would be implemented using memory chips (usually called RAM). RAM costs about $2 per megabyte. Installing the maximum amount of memory as RAM would cost 2^{12} × $2 = $8192. This might be more than you want to spend. Hard disk storage costs much less per megabyte. A 20 gigabyte hard disk costs about $90 (summer, 2001).

On modern computers, the full address space is present no matter how much RAM has been installed. This is done by keeping some parts of the full address space on disk and some parts in RAM. The RAM, the hard disk, some special electronics, and the operating system work together to provide the full 32 bit address space. To a user or an applications programmer it looks as if there are all 2^{32} bytes of main memory.

This method of providing the full address space by using a combination of RAM memory and the hard disk is called **virtual memory**. The word *virtual* means "appearing to exist, but not really there." Some computer geeks have a virtual social life.

QUESTION 6:

Which is faster: RAM memory access or hard disk access?

Next

Which is faster: physical memory access or the hard disk access?

A good answer might be:

Physical memory access.

Cache Memory

Disk access is very slow compared to RAM access. Potentially, using a combination of real memory and the hard disk to implement the address space could greatly slow down program execution. However, with clever electronics and a good operating system, using virtual memory is only slightly slower than physical memory.

Computer systems also have **cache memory**. Cache memory is very fast RAM that is inside (or close to) the processor. It duplicates the addresses and contents of the areas of main storage that are heavily used. The processor does not have to use the system bus to get or store data in cache memory. Access to cache memory is much faster than to normal main memory.

Like virtual memory, cache memory is invisible to most programs. It is an electronic detail below the level of abstraction provided by assembly language. Hardware keeps cache up to date and in synch with main storage. Your programs are unaware that there is cache memory and virtual memory. They just see "main memory". Application programs don't contain instructions that say "store this in cache memory", or say "get this from virutal memory". They only reference main memory its contents. The hardware makes sure that the program gets or stores the correct byte, no matter where it really is.

QUESTION 7:

Since memory is looks like an array of bytes, is it necessary for each item of data in memory to be one byte long?

Next

A good answer might be:

No. Most data and instructions are several bytes long and occupy several consecutive memory addresses.

Contents of Memory

The electronics of the memory system knows nothing about instructions, numbers, characters, or other data. The memory system merely stores bit patterns. The idea that some of these patterns are integers, some are characters, and some are instructions (and so on) is contained in the program you write, not in the electronics. Good programs are written to keep the various uses of memory in separate sections, but that is a programming convention, not a requirement of electronics.

> Any byte in main storage can contain any 8-bit pattern. No byte of main storage can contain anything but an 8-bit pattern. There is nothing in the memory system of a computer that says what a pattern represents.

QUESTION 8:

When first turned on, many computer systems test their RAM by writing various patterns to various locations and then reading the patterns back. Do you think this is a good test?

Next

A good answer might be:

It is a good test. The job of memory is to hold bit patterns, and to return them when requested, and that is what is tested.

End of the Chapter

You have reached the end this chapter. Test your memory by clicking on a topic to jump to the page where it is discussed. Remember to click on the "back arrow" button to get back to this page.

- The components of a computer system.
- Device controllers.
- Kilobytes, Megabytes, and Gigabytes
- Memory addresses
- Virtual memory.
- Cache memory

contents

You have reached the end of the chapter.

created: 07/04/96; 2nd edition: 03/20/01; cleaned and oiled: 09/17/01

Chapter 5 — REPRESENTING CHARACTERS WITH BIT PATTERNS

Patterns of bits represent many types of things. This chapter shows how bit patterns are used to represent characters.

- *ASCII*
- *Control characters.*
- *Teletype Machines.*
- *.asciiz and null terminated strings.*
- *Disk files.*
- *Text files.*
- *Binary files.*
- *Executable files.*

QUESTION 1:

What else (besides characters and integers) can be represented with bit patterns?

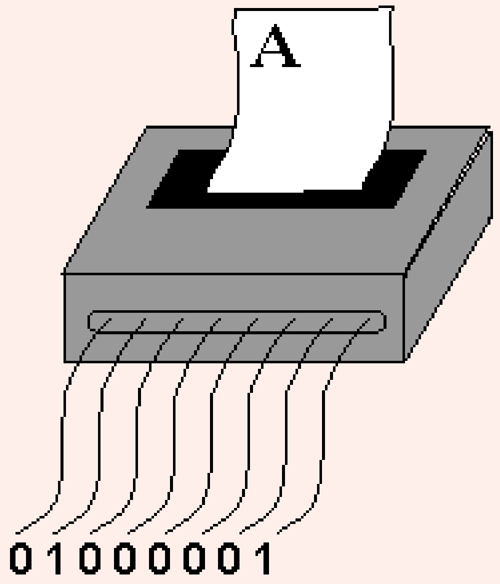
Next

A good answer might be:

Machine instructions. (Many answers are correct; anything symbolic can be rep^d with bit patterns: graphics, music, floating point numbers, internet locations, video, ...)

Representing Characters

A groups of 8 bits is a byte. Typically one character is represented with one byte. The agreement by the American Standards Committee on what pattern represents what character is called ASCII. (There are several ways to pronounce "ASCII". Frequently it is pronounced "ásk-ee"). Most microcomputers and many mainframes follow this standard.



When a printer built to print ASCII receives the ASCII pattern for "A" (along with some control signals), it prints an "A". Printers built to other specifications (typically for mainframe computers) might print some completely different character if sent the same pattern.

Some ASCII patterns do not correspond to a printable character. For example, the pattern 0x00 (ie. 0000 0000) is the NUL character. NUL is often used as a marker in collections of data. The pattern 0x0A is the LF, linefeed character, sent to a printer to indicate that it should advance one line. The patterns 0x00 through 0x1F are called **control characters** and are used to control input and output devices. The exact use of a control character depends on the particular output device. Many control characters were originally used to control the mechanical functions of a teletype machine.

QUESTION 2:

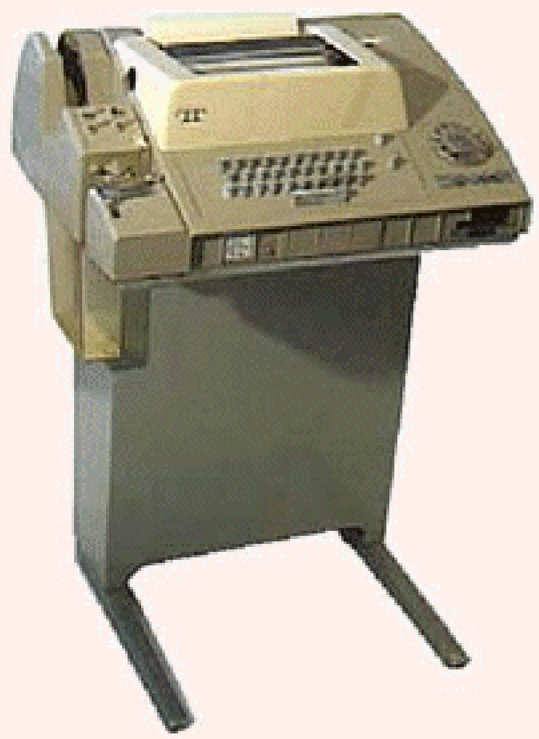
Could a computer terminal be a mechanical device???

Next

A good answer might be:

Yes. The better ones smelled of fresh machine oil and chattered pleasantly as they worked.

Teletype Machines



Teletype machines were used from 1910 until 1980 (?) to send and receive characters over telegraph lines. They were made of electrical and mechanical parts. They printed characters on a roll of paper. Various mechanical actions of the machine were requested from a distance by sending *control characters* over the line. A common control sequence was "carriage return" followed by "linefeed".

In the early days of small computers (1972-1982) teletypes were often the sole means of input and output to the computer. These tty (teletype) machines printed with fixed-size characters (like their electric typewritter cousins). The tag used in HTML for fixed-size (non-proportional) font is <tt> -- which stands for "TeleType". This paragraph is set inside <tt> ... </tt> tags.

Some models of tty's could be used off-line (not connected to anything). The user could slowly and carefully type a message (or program) and have it recorded as holes punched on a paper tape. The device on the left of the machine in the photo is the paper tape reader/puncher. Once the paper tape was correct, the tty was put on-line and the paper tape was quickly read in. In those days, paper tape was *mass storage*.

The web pages of the North American Communications Museum Off-site link have more information on teletypes.

QUESTION 3:

Can the bit patterns that are used to represent characters represent other things in other contexts?

Next

Can the patterns that are used to represent characters represent other things in other contexts?

A good answer might be:

Yes.

ASCII Chart

Hex	Char	Hex	Char	Hex	Char	Hex	Char
00	nul	20	sp	40	@	60	`
01	soh	21	!	41	A	61	a
02	stx	22	"	42	B	62	b
. . .		. . .		. . .		. . .	
0A	lf	2A	*	4A	J	6A	j
. . .		. . .		. . .		. . .	
1E	rs	3E	>	5E	^	7E	~
1F	us	3F	?	5F	_	7F	del

The chart shows some patterns used in ASCII to represent characters. The first printable character is SP (space) and corresponds to the bit pattern 0010 0000.

Space is a character, just like any other. Although not visible in the shortened chart, the upper case alphabetical characters appear in order A,B,C, ..., X, Y, Z with no gaps. There is a gap between upper case and lower case letters. The lowercase characters also appear in order a,b,c,...x, y, z.

The last pattern is 0x7F which is 0111 1111. corresponding to the DEL (delete) character. For a complete list of ASCII representations, see the appendix.

QUESTION 4:

How many of the total number of 8-bit patterns correspond to a character, (including control characters)? (Hint: look at the pattern for DEL).

Next

A good answer might be:

50 Percent, 128 out of 256

ASCII Sequences

The chart for ASCII shows all possible 7-bit patterns. There are twice as many 8-bit patterns. Some computer makers use the additional patterns to represent various things. Old MS Windows PCs, for example, used these extra patterns to represent symbols and graphics characters. Some printers will not work with these additional patterns.

Part of what an assembler does is to assemble the ASCII bit patterns that you have asked to be placed in memory. Here is a section of an assembly language program:

```
        .asciiz     "ABC abc"
```

Here are the bit patterns that the assembler will produce in the object module:

```
        41 42 43 20 61 62 63 00
```

The .asciiz part of the source code asked the assembler to assemble the characters between the quote marks into ASCII bit patterns. The first character, "A", corresponds to the bit pattern 0x41. The second character, "B", corresponds to the bit pattern 0x42. The fourth character, " " (space), corresponds to the bit pattern 0x20. The final bit pattern 0x00 (NUL) is used by the assembler to show the end of the string of characters.

QUESTION 5:

What will be the assembled patterns for this assembly code:

```
        .asciiz   "A B"
```

Next

A good answer might be:

41 20 42 00

Files

Disk files consist of blocks of bytes holding bit patterns. Usually the patterns are recorded on magnetic media such as hard disks or tape. Although the actual physical arrangement varies, you can think of a file as a contiguous block of bytes. The "DIR" command of DOS lists the number of bytes in each file (along with other information).

Ignoring implementation details (which vary from one operating system to the next), a file is a sequence of bytes on magnetic media. What can each byte contain? A byte on a magnetic disk can hold one of 2^8 (256) possible patterns, the same as a byte in main storage. Reading a byte from disk into a byte of main storage copies the pattern from one byte to another.

(Actually, for efficiency, disk reads and writes are always done in blocks of 128 bytes or more at a time).

So files, as main memory, contain bit patterns. What the bit patterns of a file represent depends on how they are used. For example, often a file contains bytes that represent characters according to the ASCII convention. Such a file is called a **text file**, or sometimes an **ASCII file**. What makes the file a text file is the knowledge about how the file was created and how it is to be used.

QUESTION 6:

You (acting as an "English Language Application") find a battered old book in a stall outside a bookshop. You open the book to a random page and see:

> Cum Luna caelo fulget, et auream Pompam sereno pandit in ambitu Ignes coruscantes, voluptas Mira trahit, retinetque visus.

Is this book suitable for you (in your role as an English application)?

A good answer might be:

No. The individual letters are the same as used in your expected context (English) but in the book their context is different.

Text Files

A computer application (program) is the context for the bit patterns of its input and output files. Although there are some standard contexts (such as for text files), many applications use a context that is their own. If you could somehow inspect the surface of a disk and see the bit patterns stored in a particular file, you would not know what they represented without additional knowledge.

Often people talk of "text files" and "MS Word files" and "executable files" as if these were completely different things. But these are just sloppy, shorthand phrases. For example, when one says "text file" one really means:

> **Text File:** A file containing a sequence of bytes. Each byte holds a bit pattern which represents a printable character or one of several allowed control characters (using the ASCII encoding scheme). Not all control characters are allowed. The file can be used with a text editor and can be sent to a hardware device that expects ASCII character codes.

Files containing bytes that encode printable characters according to the ASCII convention have about half of the possible patterns. Computer monitors and printers behave correctly with the ASCII patterns, but often can't deal with the other patterns.

QUESTION 7:

(DON'T do this with an important file:) In Notepad (or other text editor) open an EXE file (such as a compiled C program). What happens?

In Notepad open an EXE file (such as a compiled C program). What happens?

A good answer might be:

There many bit patterns the monitor can't handle; the display is messed up.

Executable Files

For extra fun, try sending an EXE file to a printer. Make sure you can stop the printer if you need to. Sometimes it will print endless sheets of paper covered with gibberish. When one says "executable file" one really means:

> **Executable File:** A file containing a sequence of bytes. Each byte holds a bit pattern that represents part of a machine instruction for a particular processor. The operating system can load (copy) an executable file into main storage and can then execute the program.

A byte in an executable file can contain any possible 8-bit pattern. A file like this often is called a **Binary File**. This is misleading vocabulary. All files represent their information as binary patterns. When one says "MS Word file" one really means:

> **Word File:** A file containing a sequence of bytes holding bit patterns created by the MS Word program, which are understood only by that program (and a few others).

There is nothing special about the various "types" of files. Each is a sequence of bytes. Each byte holds a bit pattern. A byte can hold one of 256 possible patterns (although some file types allow only 128 or fewer of these patterns). When longer bit patterns are needed they are held in several contiguous bytes.

QUESTION 8:

Say that you want to compress a text file with a file compression utility like *WinZip*. About what percent compression can you expect? (Hint: if you have been paying attention, you can figure out this answer).

Next

A good answer might be:

50%

Binary File

The bytes of a text file hold 128 out of the 256 possible 8-bit patterns. Each byte is used for half of its information carrying capacity. So a text file holds about half the information that it could. A compression utility makes (nearly) maximum use of the bytes and will compress a text file to about 50% of its original size.

All files are sequences of bytes containing binary patterns (bit patterns). But people often say **binary file** when they mean:

> **Binary File** (colloquial)**:** a file in which a byte might contain any of the possible 256 patterns (in contrast to a *text file* in which a byte may only contain one of the 128 ASCII patterns, or fewer).

An EXE file is a binary file, as is a Word file, as is an Excel file, ... , as are all files except text files. People are often not careful, and sometimes say "binary file" when they really mean "executable file". The phrase "binary file" became common amoung MS/DOS users because DOS file utilities made a distinction between text files and all others.

Using the wrong type of file with an application can cause chaos. Don't send an executable file to a printer, or open an MS Word file with a text editor. Some applications are written to deal with several types of files. MS Word can recognize text files and files from other word processors.

QUESTION 9:

Why are word processor files not text files?

Next

A good answer might be:

More things are represented than just the characters of ASCII: fonts, how to format, graphics, etc. All these things are represented with bit patterns. Different word processors use different bit patterns to indicate these things.

End of the Chapter

You have reached the end this chapter. You may wish to review the following. Click on a subject that interests you to go to where it was discussed. To return here, click on the "back arrow" button of your teletypewritter.

- ASCII
- Teletype machines.
- ASCII chart.
- .asciiz assembly language.
- Text files.
- Executable files.
- Binary files.

You have reached the end of the chapter.

created: 07/04/96; 2nd edition: 03/20/01; corrected: 09/19/01

Chapter 6 — Number Representation

Patterns of bits can represent many different things. Anything that can be represented with any kind of symbols can be represented with bit patterns. This chapter shows how bit patterns are used to represent integers.

- *Numbers and representations of numbers.*
- *Positional representation of integers.*
- *Decimal representation.*
- *Binary representation.*
- *Converting representations.*

QUESTION 1:

What is your favorite number?

Next

A good answer might be:

VII

What is a Number?

In ordinary English, the word "number" has two different meanings (at least). One meaning is the concept, the other meaning is how the concept is represented. For example, think about

> "the number of eggs in an egg carton"

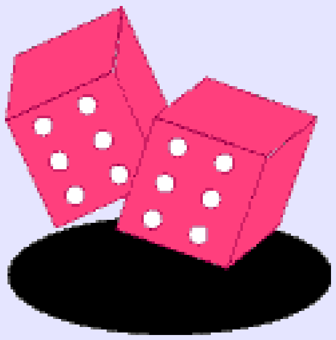
You know the number I mean. Now write it down on paper.

QUESTION 2:

How many eggs in a carton of eggs?

Next

A good answer might be:



Representations

What you wrote is a **representation** of the number. Here is one representation:

XII

This may not be the representation you used for your number. Here is another representation for that same number:

///// ///// //

Perhaps you used this representation:

12

or this:

twelve

or this:

1100_2

QUESTION 3:

Are XII, ///// ///// //, and 12 different numbers?

A good answer might be:

No, but the question is vague.

Representations

To be precise you would say that XII, ///// ///// //, and 12 are different ways to **represent** the same number.

There are many ways to represent that number. The number is always the same number. But a variety of ink marks (or pencil marks, or chalk smudges, or characters on a computer monitor) can represent it. In normal conversation and in writing "number" often means *the representation of a number.* For example, you might call. . .

12

. . . a "number". It would be tedious to call it "an instance of the representation of the number using Arabic digits with the decimal system." But that is what it is.

Computer Science is concerned with using patterns of bits to represent things. It is important to be clear about the difference between the thing, and a representation of the thing. For example, there are many ways to represent an integer using bit patterns in a computer. The integer is conceptual. The bit pattern held in a memory location is only a representation.

QUESTION 4:

Is XII even or odd?

Next

Is XII even or odd?

A good answer might be:

Even. The representation does not change the properties the number (although it often makes it easier to determine those properties).

Positional Notation

The decimal system is a **positional notation** for representing numbers. This means that the representation consists of a string of digits. The position of each digit in the string is significant. Think of each position as a numbered slot where a digit can go.

——4 ——3 ——2 ——1 ——0

Positional notation has many advantages. It is compact: "78" is shorter than "LXXIIX". Computation is easy. LXXIIX times XLIIV is hard to do using only the Roman system.

QUESTION 5:

Is XLMCIIV a legitimate Roman-system representation of an integer?

Next

A good answer might be:

Yuck!

Decimal Notation

The Roman numeral above is not a correct representation of an integer. But it takes some inspection to decide this. Decimal positional representation is clearly better.

You already know how decimal notation works (and probably would be happy to skip this topic).

324 means: $3 \times 100 + 2 \times 10 + 4 \times 1$

which is: $3 \times 10^2 + 2 \times 10^1 + 4 \times 10^0$

Remember that:

$B^0 = 1$, no matter what number B is.

Rules for Positional Notation.

1. The base B is (usually) a positive integer.
2. There are B "digits" representing zero up to (B minus one).
3. Positions correspond to integer powers of B, starting with power zero, and increasing right to left.
4. The digit placed at a position shows how many times that power of B is included in the number.

QUESTION 6:

Fill in the blanks:

$7305 = 7 \times 10^{---} + 3 \times 10^{---} + 0 \times 10^{---} + 5 \times 10^{---}$

Next

A good answer might be:

$$7305 = 7 \times 10^3 + 3 \times 10^2 + 0 \times 10^1 + 5 \times 10^0$$

Decimal Notation

For base 10 representation (often called *decimal representation*), the rules of positional notation are:

1. The base is 10.
2. There are 10 "digits", 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 .
3. Positions correspond to integer powers of 10, starting with power 0 at the rightmost digit, and increasing right to left.
4. The digit placed at a position shows how many times that power of 10 is included in the number.

A compact way of writing this is:

$$a_n\, a_{n-1}\, a_{n-2} \cdots a_2\, a_1\, a_0 = \sum_{j=0}^{n} a_j\, 10^j$$

$$= a_n\, 10^n + a_{n-1}\, 10^{n-1} + a_{n-2}\, 10^{n-2} + \cdots + a_2\, 10^2 + a_1\, 10^1 + a_0\, 10^0$$

Any integer can serve as the base for a positional representation system. Five can serve as a base.

QUESTION 7:

Here are the rules for positional notation. Fill in the blanks to work with base five:

Next

1. The base is _______.
2. There are __________ "digits": _____, _____, _____, _____, _____ .
3. Positions correspond to integer powers of ____, starting with power ____ at the rightmost digit, and increasing right to left.
4. The digit placed at a position shows how many times that power of ____is included in the number.

A good answer might be:

1. The base is five.
2. There are five "digits": 0,1, 2, 3, 4 .
3. Positions correspond to integer powers of five, starting with power 0 at the rightmost digit, and increasing right to left.
4. The digit placed at a position shows how many times that power of five is included in the number.

Base Five Notation

The number "five" in base five is represented by "10" That is:

$$\mathbf{1} \times (\text{five})^1 + \mathbf{0} \times (\text{five})^0$$

The base in any positional representation is written as "10" (assuming that the usual symbols 0, 1, ... are used). Here is a number represented in base five: 421. It means:

$$421 = \mathbf{4} \times \text{five}^2 + \mathbf{2} \times \text{five}^1 + \mathbf{1} \times \text{five}^0$$

Remember, the symbol "5" can not be used in the base five system. Actually, the above sum should be written as follows. (This looks just like base ten, but remember, here "10" is "five".)

$$421 = 4 \times 10^2 + 2 \times 10^1 + 1 \times 10^0$$

QUESTION 8:

Fill in the table.

Next

Base five Representation	Base ten Representation
0	

1	
2	
3	
4	
10	

A good answer might be:

Base five Representation	Base ten Representation
0	0
1	1
2	2
3	3
4	4
10	5

Changing Representations

You may wish to change the representation of a number from base 5 notation into base 10 notation. Do this by symbol replacement. Change the base 5 symbols into the base 10 symbol equivalents. Then do the arithmetic.

421_5	=	$4 \times {10_5}^2 + 2 \times {10_5}^1 + 1 \times {10_5}^0$
the number rep^d in base 10	=	$4 \times {5_{10}}^2 + 2 \times {5_{10}}^1 + 1 \times {5_{10}}^0$
the number rep^d in base 10	=	$4 \times 25 + 2 \times 5 + 1 \times 1$
the number rep^d in base 10	=	$100 + 10 + 1 = 111_{10}$

The above process is often called "converting a number from base 5 to base 10". But in fact, the *number* is not being converted, its *representation* is being converted. The characters "421 (base five)" and "111 (base ten)" represent the *same number*.

QUESTION 9:

Change the representation of 102_5 from base five to base ten.

Next

Change the representation of 102_5 from base five to base ten.

A good answer might be:

$102_5 = 1 \times {5_{10}}^2 + 0 \times {5_{10}}^1 + 2 \times {5_{10}}^0 = 25_{10} + 2_{10} = 27_{10}$

Base Seven

Here is another example: 326_7. This means:

$3 \times \text{seven}^2 + 2 \times \text{seven}^1 + 6 \times \text{seven}^0$

To write the number in decimal, write the powers of seven in decimal and perform the arithmetic:

$3 \times {7_{10}}^2 + 2 \times {7_{10}}^1 + 6 \times {7_{10}}^0$

$3 \times 49_{10} + 2 \times 7_{10} + 6 \times 1$

$147_{10} + 14_{10} + 6_{10} = 167_{10}$.

QUESTION 10:

Can 682_7 be rewritten in base ten notation?

Next

A good answer might be:

No. It is a meaningless representation because the digit 8 can not be used with base seven.

Secret Numbers

The symbols chosen for digits need not be the usual symbols. Here is a system for base four:

zero == @   one == !   two == #   three == $

Here is a number written in that system: !$@#

$$!\$@\# == !\times(!@)^{\$} + \$\times(!@)^{\#} + @\times(!@)^{!} + \#\times(!@)^{@}$$

The base in this system is written !@ which uses the digits of the system to mean one times the first power of the base plus zero times the zeroth power of the base.

This example illustrates that positional notation can be used without the usual digits 0, 1, 2, ..., 9. If the base is B, then B digit-like symbols are needed.

QUESTION 11:

(Frequent Midterm Question:) What is the base ten representation of the number?

Next

A good answer might be:

!$@#	=	$! \times (!@)^{\$}$	+	$\$ \times (!@)^{\#}$	+	$@ \times (!@)^{!}$	+	$\# \times (!@)^{@}$
	=	1×4^3	+	3×4^2	+	0×4^1	+	2×4^0
	=	1 × 64	+	3 × 16	+	0 × 4	+	2 × 1
	=	64	+	48	+	0	+	2
	=	114						

Bit Patterns

Bit patterns ... 10110110110 ... are sloppily called "binary numbers" even when they represent other things (such as characters or machine instructions). But soon we shall use bit patterns to represent numbers.

Consider base two.

QUESTION 12:

Fill in the blanks in the rules for binary positional notation:

1. The base is _______.
2. There are _______ "digits": ____, ____.
3. Positions correspond to integer powers of ____, starting with power ____ at the rightmost digit, and increasing right to left.
4. The digit placed at a position shows how many times that power of ____is included in the number.

A good answer might be:

1. The base is 2.
2. There are 2 "digits": 0, 1.
3. Positions correspond to integer powers of two, starting with power 0 at the rightmost digit, and increasing right to left.
4. The digit placed at a position shows how many times that power of two is included in the number.

Representing Numbers using Base Two

From the rules for positional notation there are two digits. Usually "0" and "1" are chosen. In this system the base, two, will be written "10", the first power of two plus zero times the zeroth power of two. Each place in a representation stands for a power of two. Often this is called the **binary** system. Here is an example:

$$\begin{array}{lcccccccccc}
1011 & = & 1 \times (10)^{11} & + & 0 \times (10)^{10} & + & 1 \times (10)^{1} & + & 1 \times (10)^{0} \\
 & = & 1 \times 2^{3} & + & 0 \times 2^{2} & + & 1 \times 2^{1} & + & 1 \times 2^{0} \\
 & = & 1 \times 8 & + & 0 \times 4 & + & 1 \times 2 & + & 1 \times 1 \\
 & = & 8 & + & 0 & + & 2 & + & 1 \\
 & = & 11 & & & & & &
\end{array}$$

The first line above is written entirely in base two, and shows what the string "1011" means with positional notation. Both base its powers are written in base two. The next line writes the base and its powers using decimal notation (this is what is usually done, even though it mixes notation). The remaining lines finish converting the base into decimal.

QUESTION 13:

What is 0110 (binary representation) in base ten?

Next

A good answer might be:

$$0110 = 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$$
$$= 0 + 4 + 2 + 0$$
$$= 6$$

Powers of Two

In a binary representation a particular power of two is either included in the sum or not, since the digits are either "1" or "0". In converting representations, it is convenient to have a table.

Power of 2	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Decimal	**1024**	**512**	**256**	**128**	**64**	**32**	**16**	**8**	**4**	**2**	**1**
Include?											

Here is an 8-bit pattern: 0110 1001. If it represents a number (using binary positional notation), convert the notation to decimal by including the powers of two matching a "1" bit.

QUESTION 14:

Copy 1s from the bit pattern to the last row of the table, starting at the right. Compute the sum of the corresponding decimal numbers.

Next

Here is an 8-bit pattern: 0110 1001.

A good answer might be:

Power of 2	10	9	8	7	6	5	4	3	2	1	0
Decimal	1024	512	256	128	64	32	16	8	4	2	1
Include?				1	1			1			1

Sum = 128 + 64 + 8 + 1 = 201_{10}

End of the Chapter

You have reached the end this chapter. You are now in a position to review the following subjects.

- Number vs. its representation
- Positional number representation
- Decimal number representation
- Changing bases of a representation
- Binary number representation
- Power of two table.

contents

You have reached the end of the chapter.

created: 07/04/96; 2nd edition: 03/20/01; corrected 09/19/01

Chapter 7 — Binary and Hex Integer Representation

Patterns of bits can represent many different things. Anything that can be represented with any kind of symbols can be represented with bit patterns. This chapter shows how bit patterns are used to represent integers.

- *Left and right shifts.*
- *Unsigned binary representation.*
- *Familiar binary integers.*
- *Hexadecimal representation.*
- *Equivalence of hexadecimal representation and bit pattern names.*
- *Converting representations from hexadecimal to binary.*
- *Converting representations from decimal to any base.*

QUESTION 1:

A particular number is represented by 1010 (binary representation). What is the number represented in base ten?

Next

A good answer might be:

$$1010 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$$
$$= 8 + 0 + 2 + 0$$
$$= 10$$

$1010 = 10_{10}$

It is convenient to remember the above fact. If you know it, then it takes just a moment to recognize $1001 = 9_{10}$, $1011 = 11_{10}$, and others. To convert larger binary representations to decimal representation, use a table. You can create this table from scratch.

Power of 2	10	9	8	7	6	5	4	3	2	1	0
Decimal	**1024**	**512**	**256**	**128**	**64**	**32**	**16**	**8**	**4**	**2**	**1**
Include?				**1**	0	**1**	0	**1**	0	**1**	0

Now think of the bits of the binary representation as turning on or off the numbers to include in the sum. For example, with **1010 1010** the various powers are turned on, as above.

QUESTION 2:

A particular number is represented by 1010 1010 (binary representation). What is the number represented in base ten?

Next

Power of 2	10	9	8	7	6	5	4	3	2	1	0
Decimal	1024	512	256	128	64	32	16	8	4	2	1
Include?				1	0	1	0	1	0	1	0

A good answer might be:

Adding up the "turned on" powers of two gives: $128 + 32 + 8 + 2 = 170_{10}$

Favorite Binary Numbers

Here is a list of positive integers represented in both 8-bit binary and decimal. You will encounter these patterns frequently. For now, only positive integers are being represented. Negative integers come later.

Binary	Decimal	Notes
0000 0000	0	
0000 0001	1	$2^0 = 1 = 2^1 - 1$
0000 0011	3	$2^1 + 2^0 = 3 = 2^2 - 1$
0000 0100	4	2^2
0000 0111	7	$2^2 + 2^1 + 2^0 = 2^3 - 1$
0000 1000	8	2^3
0000 1111	15	$2^4 - 1$
0001 0000	16	2^4
0001 1111	31	$2^5 - 1$
0010 0000	32	2^5
0111 1111	?	?

1000 0000	**128**	2^7

Look at the table and see if you can discover a useful pattern.

QUESTION 3:

An important number was left out of the table. What do you suppose is the decimal equivalent of 0111 1111 ?

Next

A good answer might be:

You may have noticed that 0111 1111 is one less than 1000 0000, which is 128. So 0111 1111 represents of 127_{10}.

Further Famous Bit Patterns

Say that a number is represented with all ones right to left and then all zeros, like this: 0111 1111. To find the decimal equivalent, turn on the next highest bit and turn off all others and subtract one, like this: 1000 0000 - 1. Now look up the decimal value corresponding to the single "on" bit: 1000 0000 - 1 = 128 -1 = 127.

Here is another table of bit patterns (regarded as representing positive integers) and their decimal equivalent. See if you can notice something interesting.

```
  Binary     Decimal    Notes
---------    -------   ------
0000 0001       1
0000 0010       2        1 × 2
0000 0100       4        2 × 2
0000 1000       8        4 × 2

0000 0011       3
0000 0110       6        3 × 2
0000 1100      12         6 × 2
0001 1000      24        12 × 2

0000 0111       7
0000 1110      14         7 × 2
0001 1100      28        14 × 2
0011 1000      56        28 × 2
```

QUESTION 4:

The bit pattern 0011 0010 represents 50_{10}. What bit pattern represents 100_{10}?

Next

A good answer might be:

0011 0010 = 50_{10} ; so 0110 0100 = 100_{10}

Left Shift

Useful Fact: If the number N is represented by a bit pattern X, then X0 represents 2N.

If 00110010 represents 50_{10} , then 001100100 represents 100_{10}. Often you keep the same number of bits in the shifted pattern like this: 01100100 represents 100_{10}.

This is called "shifting left" by one bit. It is often used in hardware to multiply by two. If you must keep the same number of bits (as is usually true for computer hardware) then make sure that no "1" bits are lost on the left.

QUESTION 5:

With 8 bits, there are 2^8 patterns. What is the largest positive integer that can be represented in 8 bits using base two?

A good answer might be:

$2^8 - 1 = 256 - 1 = 255$. One of the 256 patterns (0000 0000) is used for zero.

Largest Positive Integer in N Bits

The representation scheme we are looking at is called **unsigned binary** because no negative numbers are represented. Often when people say "binary number" this is what they mean. Here are some of its characteristics:

1. With N bits and base two representation, the integers 0, 1, 2, ... , $2^N - 1$ can be represented.
   - So, for instance, with 8 bits, the integers 0, 1, ..., $2^8 - 1$ can be represented. This is 0 ... 255.
2. With N bits, zero is represented by 0....0....0 (all 0's).
3. $(2^N - 1)$ is represented by 1....1....1 (all 1's).

These facts are **NOT always true** for other representation schemes! Know what scheme is used before you decide what a pattern represents.

QUESTION 6:

Without doing any calculation, which of the following is the decimal equivalent of 1111 1111 1111?

- 2048
- 4095
- 16384
- 18432

Look at fact 3 in the list and think a bit.

Hint: Actually, you should think about a bit.

A good answer might be:

4095

Hopefully you did something clever: you realized that the represented number is $2^N - 1$ which must be an odd number. (2^N means 2 × 2 × 2 ... × 2 it must be even. So $2^N - 1$ must be odd.)

So you picked the only odd number in the list. Or else you ignored the question.

Base 16 Representation

Rules for Positional Notation.

1. The base B is (usually) a positive integer.
2. There are B "digits" representing zero up to (B minus one).
3. Positions correspond to integer powers of B, starting with power zero, and increasing right to left.
4. The digit placed at a position shows how many times that power of B is included in the number.

Rule 1 says any positive integer can be used as a base. Let's use sixteen as a base. Rule 2 says we need sixteen symbols to use as digits. Here is the usual choice:

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
zero	one	two	three	four	five	six	seven	eight	nine	ten	eleven	twelve	thirteen	forteen	fifteen

Since there are only ten of the usual digits, letters are used for the remaining six hex "digits".

Base sixteen representation is called the **Hexadecimal** system, sometimes further shortened to **Hex**.

QUESTION 7:

Fill in the blanks with BASE SIXTEEN digits

Next

31A (base sixteen) =

____ $\times$ sixteen^2 +____ $\times$ sixteen^1 + ____ $\times$ sixteen^0

A good answer might be:

31A (base sixteen) =

3 × sixteen^2 + **1** × sixteen^1 + **A** × sixteen^0

Converting a Rep^n to Decimal Rep^n

As usual, to represent the number using base ten, write the base and "digits" in base ten, and then do the arithmetic:

31A (base sixteen) = 3 × sixteen^2 + 1 × sixteen^1 + A × sixteen^0

$= 3 \times 16^2 + 1 \times 16^1 + 10 \times 16^0$

$= 3 \times 256 + 1 \times 16 + 10 \times 1 = 794_{10}$

You don't have to remember powers of 16 to do this conversion. All you need is powers of 2. For example, $16^2 = 2^4 \times 2^4 = 2^8 = 256$.

As another example: $16^3 = 2^4 \times 2^4 \times 2^4 = 2^{12} = 2^2 \times 2^{10} = 4K$

QUESTION 8:

More practice: What integer is represented by 1B2 (base sixteen) ?

Next

1B2 = ____ × sixteen^2 + ____ × sixteen^1 + ____ × sixteen^0

1B2 = $\_\_\_\__{10} \times 16^2 + \_\_\_\__{10} \times 16^1 + \_\_\_\__{10} \times 16^0$

1B2 = $\_\_\_\__{10} \times \_\_\_\__{10} + \_\_\_\__{10} \times \_\_\_\__{10} + \_\_\_\__{10} \times \_\_\_\__{10}$

1B2 = $\_\_\_\__{10}$

A good answer might be:

$1B2 = 1 \times \text{sixteen}^2 + B \times \text{sixteen}^1 + 2 \times \text{sixteen}^0$

$1B2 = 1_{10} \times 16^2 + 11_{10} \times 16^1 + 2_{10} \times 16^0$

$1B2 = 1_{10} \times 256_{10} + 11_{10} \times 16_{10} + 2_{10} \times 1_{10}$

$1B2 = 256_{10} + 176_{10} + 2_{10}$

$1B2 = 434_{10}$

Shifting by one Place

You probably know how in base ten to multiply a number by 10: add a zero to the end. So 83 × 1**0** = 83**0**. This works because:

$$83 = 8 \times 10^1 + 3 \times 10^0$$

$$83 \times 10 = \left(8 \times 10^1 + 3 \times 10^0 \right) \times 10$$

$$= 8 \times 10^2 + 3 \times 10^1$$

$$= 830$$

The same trick works in any base: if a number is represented by (say) XYZ in base B, then XYZ0 represents that number times B.

QUESTION 9:

What is sixteen times 8B3 ?

Next

What is sixteen times 8B3 ?

A good answer might be:

8B30

Shifting by one Place in Base Sixteen

Adding a 0 to the end of an integer written in base sixteen multiplies the integer by sixteen. To multiply an integer by sixteen^N, add N zeros on the right:

$8B3 \times 10^3 = 8B3000$ (Here 10 means sixteen.)

Erasing a digit on the right divides an integer by sixteen (integer division). So 8B3 div sixteen = 8B (with a remainder of 3).

QUESTION 10:

A (base 16) = ________ (base 10) = ________ (base 2)

Next

A good answer might be:

A (base 16) = 10 (base 10) = 1010 (base 2)

Rep^n in Base Sixteen, Ten, and Two

Here is a chart that shows integers zero through fifteen and their positional representation using base sixteen, ten, and two. For example, $0110_2 = 2^2 + 2^1 = 6_{ten} = 6_{sixteen}$

Rep^n in base sixteen	Rep^n in base ten	Rep^n in base two		Rep^n in base sixteen	Rep^n in base ten	Rep^n in base two
0	0	0000		8	8	1000
1	1	0001		9	9	1001
2	2	0010		A	10	1010
3	3	0011		B	11	1011
4	4	0100		C	12	1100
5	5	0101		D	13	1101
6	6	0110		E	14	1110
7	7	0111		F	15	1111

QUESTION 11:

What is the name of this pattern of four bits, using the pattern naming scheme: 1010 ?

Next

What is the name of this 4-bit pattern: 1010 ?

A good answer might be:

A • this is the name of the pattern the four bits make.

Base 16 Rep^n and 4-bit Pattern Names

The name of the pattern 1010 is the same as the hex digit that corresponds to the number represented by 1010_2.

When a 4-bit pattern is regarded as a number in (unsigned) base two, that number in hex is the same as the bit pattern name.

Base Sixteen Digits:	When describing Bit Patterns:	Hex digits in Binary
0 is a digit, representing zero. . . . A is a digit, representing ten. . . . F is a digit, representing fifteen.	0 is the name of the pattern 0000 . . . A is the name of the pattern 1010 . . . F is the name of the pattern 1111	0 in 4-bit binary is 0000 . . . A in 4-bit binary is 1010 . . . F in 4-bit binary is 1111

The name of a 4-bit pattern (regarded as an abstract pattern) is the same as the hexadecimal digit whose representation in binary is that 4-bit pattern. This is not accidental.

QUESTION 12:

Here is an integer represented in base two: 1011. What is the representation in base two of the number times sixteen?

Next

Hint: remember that trick with appending zeros.

A good answer might be:

1011 0000. Appending four zeros multiplies the number being represented by 2^4.

Converting Hex Rep^n into Binary Rep^n

It is easy to convert between base sixteen and base two rep^n:

- (In base two) Shifting left by four bits is equivalent to multiplication by sixteen.
- (In base hex) Shifting left by one digit is equivalent to multiplication by sixteen.

To see how this works, look at this integer represented in base two and in base sixteen:

base two		base sixteen
1010	=	A

Now multiply each by sixteen:

base two		base sixteen
1010 0000	=	A0

Groups of four bits (starting from the right) match powers of sixteen, so each group of four bits matchs a digit of the hexadecimal rep^n. Let us rewrite the integer C6D in binary:

$$\text{C6D} = \text{C} \times \text{sixteen}^2 + 6 \times \text{sixteen}^1 + \text{D} \times \text{sixteen}^0$$

$$= \text{C} \times (2^4)^2 + 6 \times (2^4)^1 + \text{D} \times (2^4)^0$$

$$= 1100 \times (2^4)^2 + 0110 \times (2^4)^1 + 1101 \times (2^4)^0$$

$$= 1100 \times 2^8 + 0110 \times 2^4 + 1101 \times 1$$

Using the idea that each multiplication by two is equivalent to appending a zero to the right, this is:

$$= 1100\ 0000\ 0000 + 0110\ 0000 + 1101$$

C6D = 1100 0110 1101

Each digit of hex can be converted into a 4-bit binary number, each place of a hex number stands for a power of 2^4. It stands for a number of 4-bit left shifts. So you can convert a hexadecimal representation of a number into an unsigned binary represenation directly by replacing each hex digit with its 4-bit binary equivalent. For example:

```
  1    A    4    4    D      (Hex   Rep^n)

0001 1010 0100 0100 1101     (Binary Rep^n)
```

QUESTION 13:

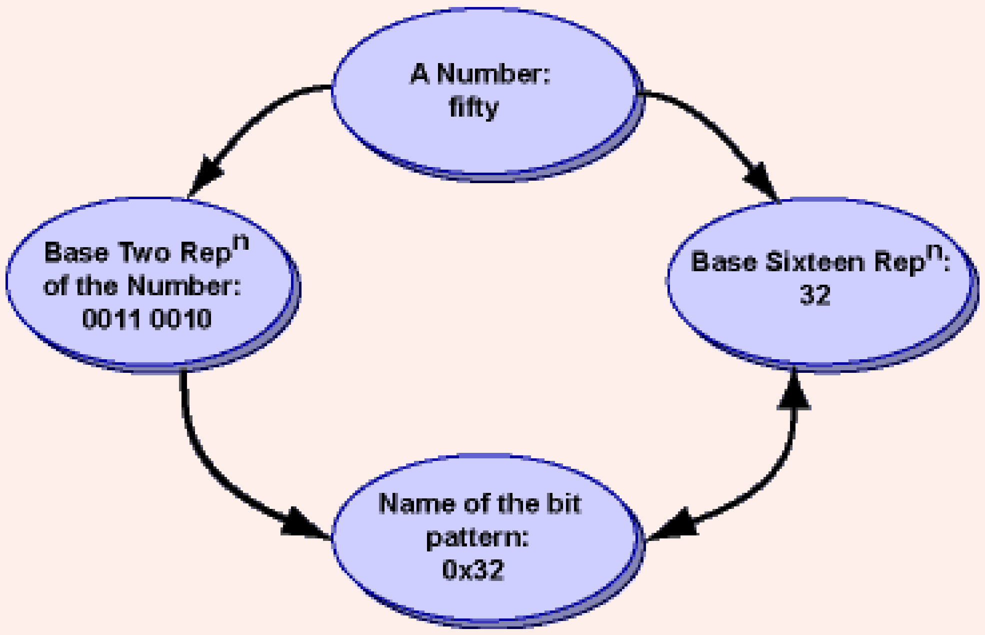
What is the name of the binary pattern 0001 1010 0100 0100 1101 ?

A good answer might be:

1A44D

Pattern Name and Number Rep^n

The pattern name of a bit pattern is the same as the base sixteen representation of the number that the bit pattern represents in base two.


A Number:
fifty
Base Two Rep^n of the Number: 0011 0010
Base Sixteen Rep^n: 32
Name of the bit pattern: 0x32

Converting between base 16 representation and base 2 representation is easy because 16 is a power of 2. Another base where this is true is base 8 (octal), since 8 is 2^3. In base 8, the digits are 0, 1, 2, 3, 4, 5, 6, 7. The binary equivalents of the digits are 000, 001, 010, 011, 100, 101, 110, 111.

Each place in a base 8 number corresponds to a left shift of 3 places, in the bit pattern of its binary rep^n.

Be careful: in some computer languages (Java and C, for instance) a number written with a leading zero signifies octal. So 0644 means base eight, and 644 means base ten, and 0x644 means base sixteen.

QUESTION 14:

What is the binary equivalent of 4733 octal?

A good answer might be:

100 111 011 011

Converting between Representations

With effort, you could directly translate between octal and hex notation. But it is much easier to use binary as an intermediate:

octal <——> binary <——> hexadecimal

You can convert a representation of a number in base B directly to a representation in base Y. But it is usually more convenient (for paper and pencil calculations) to use decimal or binary as an intermediate:

Base B <——> Decimal <——> Base Y

Base B <——> Binary <———> Base Y

QUESTION 15:

Change the representation of this number from hexadecimal to octal (base 8):

0x1A4

Next

A good answer might be:

```
          binary
    hex      groups of four

   0x1A4  =  0001 1010 0100 =


     binary         octal
     ungrouped      groups of three
```

000110100100 = 000 110 100 100 = 0644_{eight}

Remember to group bits starting from the right. If the left group is one or two bits short, add zero bits on the left as needed.

Decimal to base B

You know how to convert from Base B to Decimal. Now for converting from Decimal to base B. The algorithm converts *number* from base 10 representation to base B respresentation.

div means integer division and *mod* means modulo. *number div B* is the number of times B goes into number. *number mod B* is the left-over amount. For example 15 div 6 = 2 and 15 mod 6 = 3.

Here is an example: convert 54_{10} to hex representation. The base is 16. The first execution of the loop body calculates *digit[0]* the right-most digit of the hex number.

Algorithm: Convert a number from base 10 to base B rep^{n}

```
place  = 0;
number = number to be converted

while (number > 0 )
{
  digit[place] = number mod B ;
  number       = number div B ;
  place        = place + 1 ;
}
```

QUESTION 16:

What is digit[0] = 54 mod 16 ?

What is number = 54 div 16 ?

A good answer might be:

Since 54 = 16 × 3 with a remainder of 6:

digit[0] = 54 mod 16 = 6,

number = 54 div 16 = 3

Hex Digits from Right to Left

The first execution of the loop body calculates the digit for place 0, the rightmost digit. Notice what has happened: the first execution says that:

$$54_{10} \;=\; 3 \times 16^1 + 6 \times 16^0$$

The first execution of the loop body calculates the digit for place 0, the rightmost digit. The next execution yields the digit for place 1. This calculates digit[1] = 6 mod 16 = 6 and number = 3 div 16 = 0. After this the algorithm is done, and the result is 54_{10} = 0x36

Algorithm: Convert a number from base 10 to base B rep^n

```
place  = 0;
number = number to be converted

while (number > 0 )
{
  digit[place] = number mod B ;
  number       = number div B ;
  place        = place + 1 ;
}
```

If you are enthused about this (could happen) you might wish to use mathematical induction to prove that the algorithm is correct. A normal person would just prefer another example.

QUESTION 17:

Convert 247_{10} to hexadecimal.

A good answer might be:

number = 247
247 div 16 = 15 r 7   so digit[0] = 7

number = 15
 15 div 16 = 0 r 15   so digit[1] = F
("15" is in base 10,
"F" is the hex digit)

Result: 247_{10} = 0xF7

Another Conversion

Checking the result: 0xF7 = F × sixteen + 7 × 1 = 15 × 16 + 7 = 240 + 7 = 247_{10}

The algorithm can be described as: *Divide the number by the base. The remainder is the digit. The quotient becomes the number. Repeat.* The digits come out right to left.

Here is another base conversion. Find pencil and paper and try it.

QUESTION 18:

Convert 1033_{10} to hex.

A good answer might be:

0x904

```
1033 div 16 = 64 r 9;  digit[0] = 9
  64 div 16 =  4 r 0;  digit[1] = 0
   4 div 16 =  0 r 4;  digit[2] = 4
```

Decimal to base 5

The conversion algorithm works to convert a decimal representation of a number to any base. Here is a conversion of 1033_{10} to base 5:

```
  ___206___  r 3
5 | 1033

  ____41___  r 1
5 |  206

  _____8___  r 1
5 |   41

  _____1___  r 3
5 |    8

  _____0___  r 1
5 |    1
```

So $1033_{10} = 13113_5$ (the first digit produced is the rightmost). Checking the results:

$$\begin{aligned} 13113_5 &= 1 \times 5^4 + 3 \times 5^3 + 1 \times 5^2 + 1 \times 5^1 + 3 \times 5^0 \\ &= 1 \times 625 + 3 \times 125 + 1 \times 25 + 1 \times 5 + 3 \\ &= 625 + 375 + 25 + 5 + 3 \\ &= 1033_{10} \end{aligned}$$

QUESTION 19:

Convert 100_{10} to base 3.

A good answer might be:

100_{10} =

100 div 3 = 33 r 1;
33 div 3 = 11 r 0;
11 div 3 = 3 r 2;
3 div 3 = 1 r 0;
1 div 3 = 0 r 1

So $100_{10} = 10201_3$.

Converting Base 3 to base 7

Checking the answer:

$$10201_3 = 1 \times 3^4 + 0 \times 3^3 + 2 \times 3^2 + 0 \times 3^1 + 1 \times 3^0$$
$$= 1 \times 81 + 0 \times 27 + 2 \times 9 + 0 \times 3 + 1$$
$$= 81 + 18 + 1 = 100$$

To convert from a representation in base three to a representation in base seven, use base ten as an intermediate:

$$\_\_\_\__3 \longrightarrow \_\_\_\__{10} \longrightarrow \_\_\_\__7$$

QUESTION 20:

What is the base 7 representation of 10201_3?

A good answer might be:

```
100 div 7 = 14 r 2;
 14 div 7 =  2 r 0;
  2 div 7 =  0 r 2;
```

So $10201_3 = 100_{10} = 202_7$

End of the Chapter

You have reached the end this chapter. You may wish to base your study on a review the following. Click on a subject that interests you to go to where it was discussed. To get back here, click on the "back arrow" button of your browser.

- Familiar binary integers.
- Shifting left a binary pattern.
- Unsigned binary representation.
- Hexadecimal representation.
- Shifting left a hexadecimal pattern.
- Equivalence of hexadecimal strings and bit pattern names
- Converting representations, between base 16 and base 2.
- Converting a decimal representation to any base.

contents

You have reached the end of the chapter.

created: 07/04/96; 2nd edition: 05/26/01; additions: 09/25/01

Chapter 8 — BINARY ADDITION and TWO's COMPLEMENT

Digital computers use bit patterns to represent many types of objects. Various operations can be performed on objects. Computers perform operations on bit patterns. With a good representation scheme, bit patterns represent objects and bit pattern manipulations represent operations on objects.

An important example of this is the Binary Addition Algorithm, where two bit patterns representing two integers are manipulated to create a third pattern which represents the sum of the integers.

QUESTION 1:

Compute the following. Give the answer in binary notation.

- 0 + 0 = ?
- 0 + 1 = ?
- 1 + 0 = ?
- 1 + 1 = ?

Next

A good answer might be:

In binary:

- 0 + 0 = 0
- 0 + 1 = 1
- 1 + 0 = 1
- 1 + 1 = 10

The Binary Addition Algorithm

Here is an important algorithm. All computer scientists and computer engineers know it. It is a bit-pattern manipulation procedure that is built into the hardware of (nearly) all computers. It is called the **binary addition algorithm**.

Let us start by adding 1-bit integers. The operation is performed on three operands (three input values). Arrange the bits into a column. The bit at the top of the column is called the "carry into the column".

To add two 1-bit (representations of) **integers:** Count the number of ones in a column and write the result in binary. The right bit of the result is placed under the column of bits. The left bit is called the "carry out of the column". The following table shows the outcomes with all possible operands.

Binary Addition Table

```
                                 the carry into the column
                                                         |
                                                         |
     1    1    1    1    0    0    0    0  <---
     1    0    1    0    1    0    1    0
     1    1    0    0    1    1    0    0
    ---  ---  ---  ---  ---  ---  ---  ---
    11   10   10   01   10   01   01   00
                                   ^
                                   |
                     the carry out of the column
```

QUESTION 2:

Perform the following additions:

```
 1      0      1
 0      1      1
 1      0      0
---    ---    ---
```

A good answer might be:

```
 1     0     1
 0     1     1
 1     0     0
---   ---   ---
10    01    10
```

N-bit Binary Addition Algorithm

Adding a column of bits is as easy as counting. It is also easy to do electronically. When you take a course in digital logic you will probably build a circuit to do it.

Now let us look at the full, n-bit, binary addition algorithm. The algorithm takes two **operands** and produces one result. An operand is the data that an algorithm operates on.

To add two N-bit (representations of) **integers:** Proceed from right-to-left, column-by-column, until you reach the leftmost column. For each column, perform 1-bit addition. Write the carry-out of each column above the column to its left. The bit is the left column's carry-in.

```
           0       10      110      0110
0110     0110     0110     0110     0110
0111 ==> 0111 ==> 0111 ==> 0111 ==> 0111
----     ----     ----     ----     ----
            1       01      101     1101
```

QUESTION 3:

Confirm that this addition is correct. (1) Check that the binary addition algorithm was performed correctly, then (2) translate the operands into decimal, and then (3) perform the addition in decimal.

Next

0110 = $\_\_\_\_\__{10}$

```
0111   =   ______10
----       -----
1101   =   ______10
```

A good answer might be:

```
0110   =  6₁₀
0111   =  7₁₀
----     ---
1101   = 13₁₀
```

A Longer Example

Here is another example:

```
             0          10                        110        1 110       11 110
 0110 1110    0110 1110    0110 1110       0110 1110    0110 1110    0110 1110
+ 0001 0111    0001 0111    0001 0111      0001 0111    0001 0111    0001 0111
 ---------    ---------    ---------      ---------    ---------    ---------
              1          01                        101        0101       0 0101


  111 110      1111 110     01111 110
 0110 1110    0110 1110    0110 1110
 0001 0111    0001 0111    0001 0111
 ---------    ---------    ---------
   00 0101     000 0101    1000 0101
```

The carry out of the left column in the final sum can be discarded, in this case. But in general you must be careful with it. See the following pages.

Check the answer by converting to decimal representation and doing the addition in that base (the same numbers are being added, but now represented in a different way, so the sum is the same.)

```
 01111 110
  0110 1110  =  110₁₀
+ 0001 0111  =   23₁₀
  ----------   -----
  1000 0101  =  133₁₀
```

QUESTION 4:

Do the following:

Next

```
  10
+ 01
----
```

A good answer might be:

```
  00
  10
+ 01
----
  11
```

This is another case where the answer fits into the same number of bits as the numbers to add.

Details, Details

Here are some details:

1. Usually the operands and the result have a fixed number of bits (usually 8, 16, 32, or 64). These are the sizes that processors use to represent integers.
2. If the bits in the leftmost columns of the sum are zero, include them in the answer to keep the result the same bit-length as the operands.
3. Compute the carry-out of the leftmost column, but don't write it as part of the answer (because there is no room.)
4. When the bit patterns are regarded as representing positive integers (*unsigned binary representation*), a carry-out of 1 from the leftmost column means the sum does not fit into the fixed number of bits. This is called **Overflow**.
5. When the bit patterns are regarded as representing positive and negative integers (as described in the last pages of this chapter), then a carry-out of 1 from the leftmost column is not necessarily overflow.

QUESTION 5:

A particular processor chips (say the MIPS R2000) has registers of size 32 bits. How many bits does this processor typically use each time binary addition is performed?

Next

A good answer might be:

32 bits.

Hex Addition

Here is a problem with 4 bits:

```
1111
 0111
 1001
 ----
 0000
Overflow was detected.
```

Two four-bit numbers are added, but the sum does not fit in four bits. If we were using five bits the sum would be 1 0000. But with four bits there is no room for the significant "1". Because the carry out from the most significant column of the sum (the left column) is "1", the 4-bit result is not valid.

The electronic circuits of a computer can easily detect overflow by checking if the carry-out of the leftmost column is a zero or a one. A program might branch to an error handling routine when overflow is detected.

QUESTION 6:

Add these numbers, represented in eight bits:

```
0010 1100
0101 0101
---------
```

A good answer might be:

```
0 1111 100
  0010 1100
  0101 0101
  ---------
  1000 0001
```

(No overflow)

More Practice

Practice performing this algorithm mechanically, like a computer. Detect overflow by looking at the carry-out of the leftmost column. You don't need to convert the representation to decimal.

QUESTION 7:

Add these numbers, represented in eight bits:

Next

```
0110 1100
1001 1111
---------
```

A good answer might be:

```
1 1111 100
  0110 1100
  1001 1111
  ---------
  0000 1011
```

(overflow detected)

Hex Addition

Sometimes you need to add two numbers represented in hexadecimal. Usually it is easiest to convert to binary representation to do the addition. Then convert back to hexadecimal.

Remember that converting representations from hexadecimal to binary is easily done by using the equivalence of hex digits and bit pattern names.

QUESTION 8:

Perform the following addition:

```
0F4A
420B
----
```

Next

A good answer might be:

```
       1 11    1  1
0F4A → 0000 1111 0100 1010
420B → 0100 0010 0000 1011
----   --------------------
       0101 0001 0101 0101  → 0x5155
```

Additional Addition Practice

The carry out of zero from the left column in the binary addition indicates that the sum can be contained in that number of bits

Sometimes a hex addition problem is easy enough to do without converting to binary:

```
014A
4203
----
434D
```

It may be helpful to use your digits (fingers) in doing this: to add A+3 extend three fingers and count "A... B... C... D".

QUESTION 9:

Compute the following sum using 8 bits. Is there overflow?

```
1101 0010
0110 1101
---------
```

Next

A good answer might be:

0011 1111

```
 11
  1101 0010    210_10
  0110 1101    109_10
  ---------    ---
  0011 1111     63_10
```

The carry bit of 1 indicates overflow.

Yet More Addition Practice

The *correct application* of the "Binary Addition Algorithm" sometimes gives *incorrect results* (because of overflow). With paper-and-pencil arithmetic overflow is not a problem because you can use as many columns as needed.

Detecting Overflow with Unsigned Binary

When the "Binary Addition Algorithm" is used with unsigned binary integer representation, the result is CORRECT only if *the CARRY OUT* of the high order column is ZERO.

But digital computers use fixed bit-lengths for integers, so overflow is possible. For instance the MIPS 3000 processor can represent integers in 8, 16, or 32 bits. When 8-bit operands are added overflow is certainly possible. Later MIPS processors include a 64-bit integer data type, but even so, overflow is possible.

QUESTION 10:

Compute the following sum using 8 bits:

Next

0000 0001

```
1111 1111
---------
```

A good answer might be:

```
 11111 111
  0000 0001     1₁₀
  1111 1111   255₁₀
  ---------  -----
  0000 0000    00₁₀
```

The carry bit of 1 out of the high order column (leftmost) indicates an overflow.

Negative Integers

Unsigned binary representation can not be used to represent negative numbers. With paper and pencil numbers, a number is made negative by putting a negative sign in front of it: 17_{10} negated = -17_{10}. You might hope to do the same with binary:

```
0001 1000    negated =   -0001 1000  ???
```

Unfortunately, you can't put a negative sign in front of a bit pattern in computer memory. Memory holds only patterns of 0's and 1's. Somehow negative integers must be represented using bit patterns. But this is certainly possible. Remember those advantages of binary?

1. Easy to build.
2. Unambiguous signals (hence noise immunity).
3. Can be copied flawlessly.
4. **Anything that can be represented with patterns can be represented with patterns of bits.**

If we can represent negative integers with paper and pencil (thus using patterns) we certainly can represent negative integers with patterns of bits.

QUESTION 11:

Let us say that you need to represent an equal number of positive and negative integers in eight bits. How many negative numbers can represented? How many positive numbers?

Next

Just for fun, can you think of a way to do this? Get out some scratch paper and scratch for a while with eight-bit patterns.

A good answer might be:

Since there are 256 possible bit patterns with 8 bits, there could be 128 positive and 128 negative integers.

You may have thought of the sign-magnitude method, discussed below.

Sign-Magnitude Rep^n

There are many schemes for representing negative integers with patterns of bits. One scheme is **sign-magnitude**. It uses one bit (usually leftmost) to indicate the sign. "0" indicates a positive integer, and "1" indicates a negative integer. The rest of the bits are used for the magnitude of the number. So -24_{10} is represented as:

1001 1000

The sign "1" means negative
The magnitude is 24 (in 7-bit binary)

QUESTION 12:

With 8-bit sign-magnitude representation, what positive integers can be represented and what negative integers can be represented?

Next

A good answer might be:

-127_{10} ... 0 ... $+127_{10}$

Problems with Sign-Magnitude

One of the eight bits is used for the sign, which leaves seven bits for the magnitude. The magnitude uses 7-bit unsigned binary, which can represent 0_{10} (as 000 0000) up to 127_{10} (as 111 1111). The eighth bit makes these positive or negative, resulting in -127_{10}, ... -0, 0, ... 127_{10}.

One pattern corresponds to "minus zero", 1000 0000. Another corresponds to "plus zero", 0000 0000.

There are several problems with sign-manitude. It works well for representing positive and negative integers (although the two zeros are bothersome). But it does not work well in computation. A good representation method (for integers or for anything) must not only be able to represent the objects of interest, but must support operations on those objects.

This is what is wrong with Roman Numerals: they can represent any integer you want, but they are very, very poor when used in computation.

QUESTION 13:

Can the "binary addition algorithm" be used with sign-magnitude rep^n? Try adding +16 with -24:

```
0001 0000    16
1001 1000   -24
---------   ---
```

A good answer might be:

```
  1
0001 0000     16₁₀
1001 1000    -24₁₀
---------    ----
1010 1000    -40₁₀ (wrong!)
```

Patterns that Add Up to Zero

The "binary addition algorithm" does NOT work with sign-magnitude. There are algorithms that do work with sign-magnitude, and some early computers were built to use them. (And other computers were built using other not-ready-for-prime-time algorithms). It took several years of experience for computer science to decide that a better way had to be found.

There is a better way. Recall a question and its answer from several pages ago:

```
11111 111
 0000 0001  =  1₁₀
 1111 1111  =  ?
 ---------     -----
 0000 0000  =  0₁₀
```

QUESTION 14:

A number added to one results in a zero. What could that number be?

Next

A good answer might be:

Might be minus one.

Patterns that Add to Zero

Minus one added to plus one gives zero. So if a particular bit pattern results in the pattern that represents zero when added to one, it can represent minus one.

```
11111 111
 0000 0001  =   1₁₀
 1111 1111  =  -1
 ---------     -----
 0000 0000  =   0₁₀
```

There is a carry out of one from the high order column, but that is fine *in this situation*. The "binary addition algorithm" correctly applied gives the correct result in the eight bits. Look at another interesting case:

```
11111 11
 0000 1110  =   14₁₀
 1111 0010  =   ?
 ---------     -----
 0000 0000  =   0₁₀
```

QUESTION 15:

A number added to 14_{10} results in a zero. What could that number be?

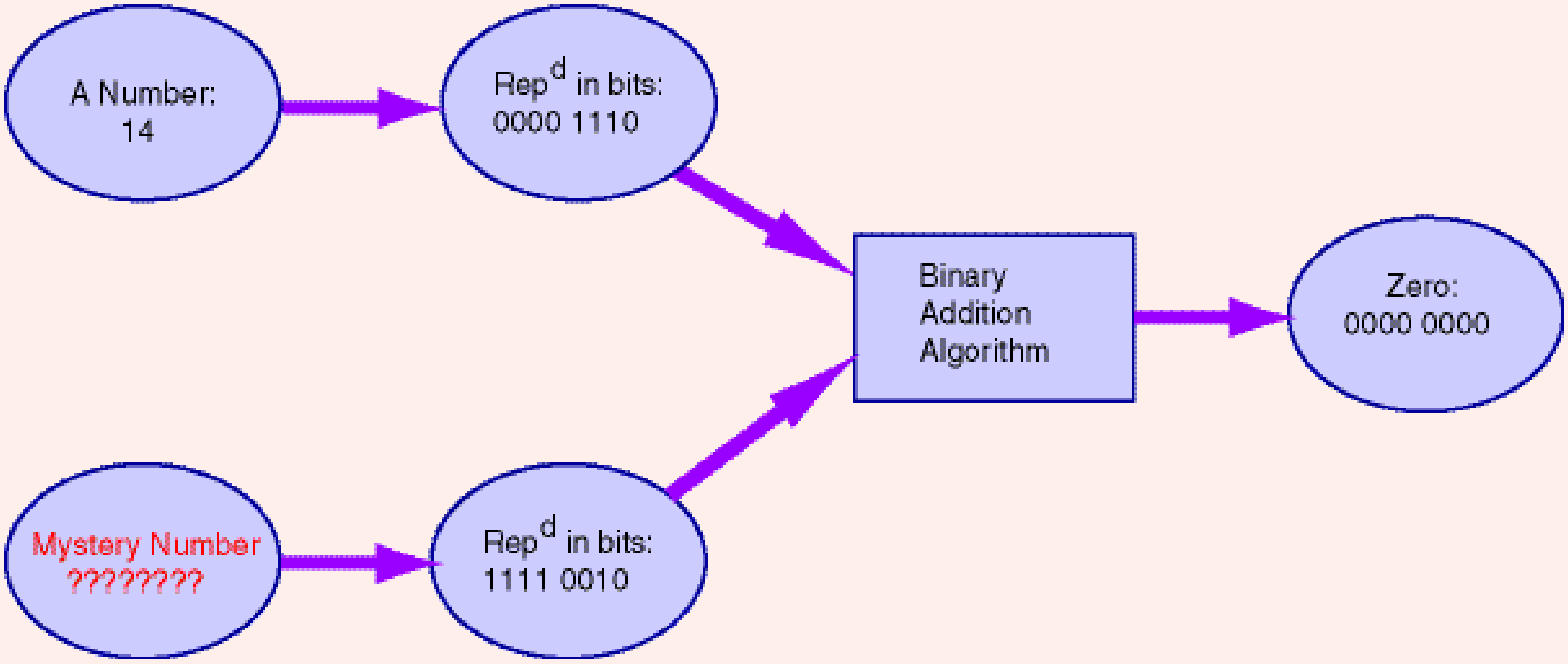
Next

A good answer might be:

Might be -14.

Negative Fourteen

The pattern "1111 0010" might be a good choice for "negative fourteen". In the picture, the only good choice for "Mystery Number" is negative fourteen.



For every bit pattern of N bits there is a corresponding bit pattern of N bits which produces an N-bit zero when the two patterns are used as operands for the binary addition algorithm. Each pattern can be thought of as representing the negative of the number that is represented by the other pattern.

QUESTION 16:

Find the 8-bit pattern that gives eight zero-bits when added to the bit pattern for 6_{10}. (Hint: start at the right column, figure out what the ? has to be in each column, and work your way left.)

Next

```
0000 0110  =  6₁₀
???? ????  = -6₁₀
---------    ----
0000 0000     0₁₀
```

A good answer might be:

```
11111 11
 0000 0110  =   6₁₀
 1111 1010  =  -6₁₀
 ---------    ----
 0000 0000     0₁₀
```

Two's Complement

This representation scheme is called **two's complement**. It is one of many ways to represent negative integers with bit patterns. But it is now the nearly universal way of doing this. Integers are represented in a fixed number of bits. Both positive and negative integers can be represented. When the pattern that represents a positive integer is added to the pattern that represents the negative of that integer (using the "binary addition algorithm"), the result is zero. The carry out of the left column is discarded.

Here is how to figure out which bit-pattern gives zero when added (using the "binary addition algorithm") to a another pattern.

How to Construct the Negative of an Integer in Two's Complement:

Start with an N-bit rep^n of an integer. To calculate the N-bit rep^n of the negative integer:

1. Reflect each bit of the bit pattern (change 0 to 1, and 1 to 0).
2. Add one.

An example:

```
The positive integer: 0000 0001   ( one )
Reflect each bit:     1111 1110
Add one:              1111 1111   ( minus one )
```

QUESTION 17:

Fill in the blanks:

Next

The positive integer: 0000 0110 (6_{10})

Reflect each bit: ____ ____

Add one: ____ ____ (-6_{10})

A good answer might be:

The positive integer: 0000 0110 (6_{10})

Reflect each bit: 1111 1001

Add one: 1111 1010 (-6_{10})

Negative Six Again

This resulted in the same rep^n for minus six as we figured out before. Here is what is going on: When each bit of a pattern is reflected then the two patterns added together make all 1's. This works for all patterns:

```
0110 1010   pattern
1001 0101   pattern reflected
---------
1111 1111   all columns filled
```

Adding one to this pattern creates a pattern of all zero's:

```
11111 111    row of carry bits
 1111 1111   all columns filled
 0000 0001   one
 ---------
 0000 0000
```

But instead, the one is added to the "pattern reflected."

pattern + pattern reflected = all ones

pattern + pattern reflected + one = all zeros

pattern + (pattern reflected + one) = all zeros

pattern + (two's complement of pattern) = all zeros

Don't be too upset if the above is confusing. Mostly all you need is to get used to this stuff. Of course, this takes practice.

QUESTION 18:

What is the two's complement of 0100 0111?

Next

A good answer might be:

0100 0111 reflect $\rightarrow$ 1011 1000 add one $\rightarrow$ 1011 1001

It is always a good idea to check your results:

```
 11111 111
  0100 0111
+ 1011 1001
  ---------
  0000 0000
```

Two's Complement Integers

What is the two's complement of zero?

```
   zero =  0000 0000
reflect =  1111 1111
add one =  0000 0000
```

Using the algorithm with the rep^n for zero results in the rep^n for zero. This is good. Usually "minus zero" is regarded as the same thing as "zero". Recall that with the sign-magnitude method of representing integers there where both "plus" and "minus" zero.

What integers can be represented in 8-bit two's complement? Two's complement represents both positive and negative integers. So one way to answer this question is to start with zero, check that it can be represented, then check one, then check minus one, then check two, then check minus two ... Let's skip most of those steps and check 127_{10}:

```
    127 =  0111 1111        check:   0111 1111
reflect =  1000 0000                 1000 0001
add one =  1000 0001                 ---------
                                     0000 0000
   -127 =  1000 0001
```

Now try 128_{10}:

QUESTION 19:

128 = 1000 0000

reflect = ____ ____

add one = ____ ____

A good answer might be:

```
      128  =  1000 0000

  reflect  =  0111 1111

  add one  =  1000 0000   ??????
```

Range of Integers with 2's Complement

This is not good. A non-zero integer and its negative can't both be represented by the same pattern. So +128 can not be rep^d in eight bits. The maximum positive integer that can be rep^d in eight bits is 127_{10}. The negative integers $-1 .. -127_{10}$ also can be represented.

What number is represented by 1000 0000? Add the rep^n of 127_{10} to it:

Range of N Bit 2's Complement
$-2^{(n-1)}$ ... 0 ... $2^{(n-1)} - 1$
eg n= 8 $-2^{(8-1)} = -128$ ... 0 ... $2^{(8-1)} - 1 = 127$

```
1000 0000    = ?
0111 1111    = 127₁₀
---------
1111 1111    = -1₁₀
```

A good choice for ? is -128_{10}. Therefore 1000 0000 represents -128_{10}. Eight bits can represent the numbers -128_{10} ... 0 ... $+127_{10}$.

QUESTION 20:

How many integers are there in the range -128_{10} ... 0 ... $+127_{10}$.

How bit patterns can be formed with 8 bits?

Next

A good answer might be:

256

256

Every pattern of the 256 patterns has been assigned an integer to represent.

The "Sign Bit"

The algorithm that creates the rep^n of the negative an integer works with both positive and negative integers. Start with N and complement it (make its negative): you get -N. Now complement -N and you get the original N.

0110 1101 $\rightarrow$ 1001 0010 add one $\rightarrow$ 1001 0011

1001 0011 reflect $\rightarrow$ 0110 1100 add one $\rightarrow$ 0110 1101

With N-bit two's comp rep^n, the high order bit is "0" for positive integers and "1" for negative integers. This is a fortunate result. The high order bit is sometimes called the **sign bit**. But it is not really a sign (it does not play a separate role from the other bits). It takes part in the "binary addition algorithm" just as any bit.

QUESTION 21:

Does the following **four-bit** two's complement represent a negative or a positive integer?

1001

Next

A good answer might be:

negative

The Sign of the Four Bits

Be Certain of this: it is by happy coincidence that the high order bit of a two's comp. integer is 0 for positive and 1 for negative. But, to create the two's comp. rep^n of the negative of a number you must "reflect, add one". Changing the sign bit by itself will not work.

To convert N bits of two's complement representation into decimal representation:

1. If the integer is negative, complement it to get a positive integer.
2. Convert (positive) integer to decimal (as usual).
3. If the integer was originally negative, put "-" in front of the decimal.

The number rep^d by 1001 is negative (since the "sign bit" is one), so complement:

$$1001 \rightarrow 0110 + 1 \rightarrow 0111$$

Convert the result to decimal: 0111 = 7_{10}. Add a negative sign: -7_{10}. So (in 4-bit two's comp. rep^n) 1001 represents -7_{10}.

QUESTION 22:

What is the decimal rep^n of this 8-bit two's complement integer: 1001 1111

Next

A good answer might be:

$1001\ 1111 \quad \text{reflect} \rightarrow \quad 0110\ 0000 \quad \text{add one} \rightarrow \quad 0110\ 0001$

$\text{convert to decimal} \rightarrow \quad 2^6 + 2^5 + 1 = 97_{10}$

$\text{put sign in front} \rightarrow \quad -97_{10}$

Overflow Detection in 2's Complement

The "binary addition algorithm" can be applied to any pair of bit patterns. The adder electonics inside the microprocessor performs this operation with any two bit patterns you send it. You send it bit patterns. It does its job. It is up to you (as the writter of the program) to be sure that the operation makes sense.

Overflow is detected in a different way for each representation. In the following exhibit the "binary addition algorithm" is applied to two bit patterns. Then the results are looked at as unsigned binary and then as two's complement:

Algorithm applied to bit patterns	**interpretation of the patterns as unsigned binary**	**interpretation of the patterns as 2's complement**
1111 11 0111 0011 1101 0001 --------- 0100 0100	115_{10} 209_{10} ---- 68_{10} (incorrect)	115_{10} -47_{10} ---- 68_{10} (correct)

The "binary addition algorithm" was performed on the operands. The result is either correct or incorrect depending on how the bit patterns are interpreted. If the bit patterns are regarded as unsigned binary integers, then overflow happened. If the bit patterns are regarded as two's comp integers, then the result is correct.

Detecting two's complement overflow

When the "binary addition algorithm" is used with operands in two's complement representation, the result is correct if *the carry INTO* the high order column is the same as *the carry OUT OF* the high order column. The carry bits can both be zero or both be one.

QUESTION 23:

Which case applies to the 2's comp result, above:

Next

- (correct result:) The carry into the left column is the SAME as the carry out.
- (incorrect result:) The carry into the left column is DIFFERENT from the carry out.

A good answer might be:

- (correct result:) The carry into the left column is the SAME as the carry out.

Carry In = Carry Out

With two's comp rep^n the result of addition is correct if the carry into the high order column is the same as the carry out of the high order column. The carry can be one or zero. Overflow is detected by comparing two bits, an easy thing to do with electronics. Here are some more examples:

No Overflow	**No Overflow**	**Overflow**	**Overflow**
11111 111	00000 011	01111 100	10000 000
0011 1111 (63_{10})	1100 0001 (-63_{10})	0011 1111 (63_{10})	1100 0001 (-63_{10})
1101 0101 (-43_{10})	0010 1011 (43_{10})	0110 0100 (100_{10})	1001 1100 (-100_{10})
---------	---------	---------	---------
0001 0100 (20_{10})	1110 1100 (-20_{10})	1010 0011 (-93_{10})	0101 1101 (93_{10})

QUESTION 24:

Perform the "binary addition algorithm" on the following 8-bit two's complement numbers. Is the result correct or not?

```
1011 1101
1110 0101
---------
```

Next

A good answer might be:

The result is correct.

```
11111 1 1
 1011 1101
 1110 0101
 ---------
 1010 0010
```

Subtraction in Two's Complement

Since the carry into the high order column is the same as the carry out of the high order column, the result is correct. We didn't even need to figure out what decimal numbers the patterns represented. Looking at the situation in detail it turns out that everything is correct:

11111 1 1
 1011 1101 (-67_{10})
 1110 0101 (-27_{10})

 1010 0010 (-94_{10})

The "binary addition algorithm" is used for subtraction and addition. To subtract two numbers represented in two's complement, negate the number to be subtracted and then add. Overflow is detected as usual.

QUESTION 25:

Subtract 0001 1011 from 0011 0001. Is the result correct or not?

Next

A good answer might be:

0001 0110 (correct)

Multipurpose Algorithm

The number to be subtracted is negated by the usual means (reflect the bits, add one):

$$0001\ 1011 \rightarrow 1110\ 0101$$

Then the "binary addition algorithm" is used:

```
11100 001
 0011 0001    =  49₁₀
 1110 0101    = -27₁₀
 ---------    -----
 0001 0110      22₁₀
```

Since the carry into the most significant column is the same as the carry out of that column, the result is correct. We are getting quite a bit of use out of the "binary addition algorithm". It is used to:

- Add integers represented in unsigned binary.
- Add integers represented in two's complement binary.
- Subtract integers represented in two's complement binary.

It can't be used to subtract integers represented in unsigned binary. But if subtraction is being performed all the data should be using two's complement representation, or somebody made a mistake. In the programming language Java all integers (int and long) are represented using two's complement.

QUESTION 26:

Subtract 0101 1001 from 0010 0101 Is the result correct or not?

Next

A good answer might be:

1100 1100 (correct)

More Practice Subtraction

The number to be subtracted is negated by the usual means (reflect the bits, add one):

0101 1100 $\rightarrow$ 1010 0111

Then the "binary addition algorithm" is used:

```
00100 111
 0010 0101   =  37₁₀
 1010 0111   = -89₁₀
 ---------    -----
 1100 1100     -52₁₀
```

Since the carry into the most significant column is the same as the carry out of that column the result is correct. The answer came out negative, but that is fine.

You must practice the "binary addition algorithm" until you are sick of it. Or you can change your major to Art History (and leave the high paying jobs to the rest of us). Here is another problem:

QUESTION 27:

Subtract 0101 1001 from 1110 0101. Is the result correct or not?

Next

A good answer might be:

1000 1100   (correct)

No! Not More Practice Subtraction!

The number to be subtracted is negated by the usual means, as in the previous problem. Then the "binary addition algorithm" is used:

```
11100 111
 1110 0101   = -27₁₀
 1010 0111   = -89₁₀
 ---------    -----
 1000 1100    -116₁₀
```

Since the carry into the most significant column is the same as the carry out of that column the result is correct. The answer came out negative, but that is fine.

A computer does several million of these operations per second. Surely you can do another one? How about if I give you two whole seconds?

QUESTION 28:

Subtract 0111 1000 from 1011 0000. Is the result correct or not?

Next

A good answer might be:

0011 1000 (incorrect)

End of the Chapter

The integer to be subtracted is complemented:

$$0111\ 1000 \rightarrow 1000\ 0111 + 1 \rightarrow 1000\ 1000$$

...then added to the other:

```
10000 000
 1011 0000  = - 80₁₀
 1000 1000  = -120₁₀
 ---------    -----
 0011 1000     +56₁₀ !??!!
```

Since the carry into the most significant column is *NOT* the same as the carry out, the result is not correct.

You have reached the end this chapter. You may wish to review the following unless your brain has overflowed. Click on a subject that interests you to go to where it was discussed. To get back here, click on the "back arrow" button of your browser.

- The binary addition algorithm
- Details of the binary addition algorithm.
- Hexadecimal addition.
- Overflow in unsigned binary addition.
- Sign-magnitude representation
- Two's complement representation
- The sign bit with two's complement representation.
- Overflow in two's complement addition.

You have reached the end of the chapter.

created: 06/06/01

CHAPTER 9 --- First SPIM Program

This chapter discusses how to run the SPIM simulator. A small MIPS program is used as an example.

Chapter Topics:

- *Starting SPIM*
- *The SPIM user interface*
- *Writing an assembly source program*
- *Assembling and loading a program*
- *Running a program*

This chapter goes through the steps of running a program on SPIM (the MIPS simulator). Details will be discussed in later chapters.

QUESTION 1:

(Review) What is a **register**?

Next

A good answer might be:

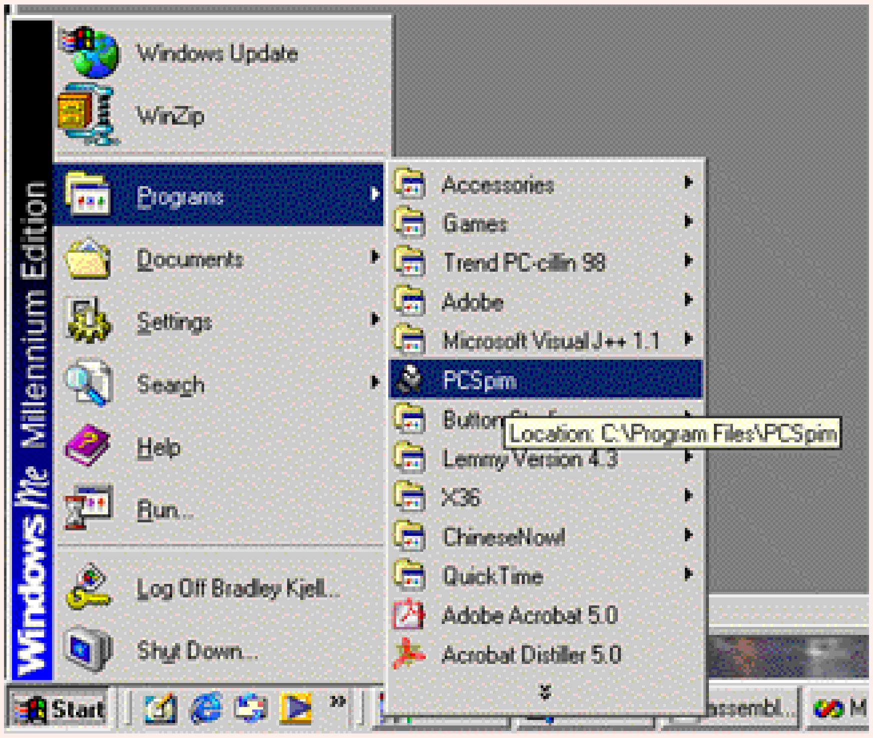
A **register** is a part of the processor that holds a bit pattern. Processors have many registers.

Starting SPIM

MIPS R2000 processors have 32 general purpose registers, each holding 32 bits.

The first example SPIM program puts bit patterns representing integers into two registers. Then it adds the two patterns together. The screen shots for this example are from a *MS Win ME* system, but any Windows OS will be similar. Unix and Linux should be close.

To start SPIM click on its icon, reached (on my system) by first clicking on the start button.



If you have not installed SPIM see the appendix.

QUESTION 2:

How else can SPIM be started?

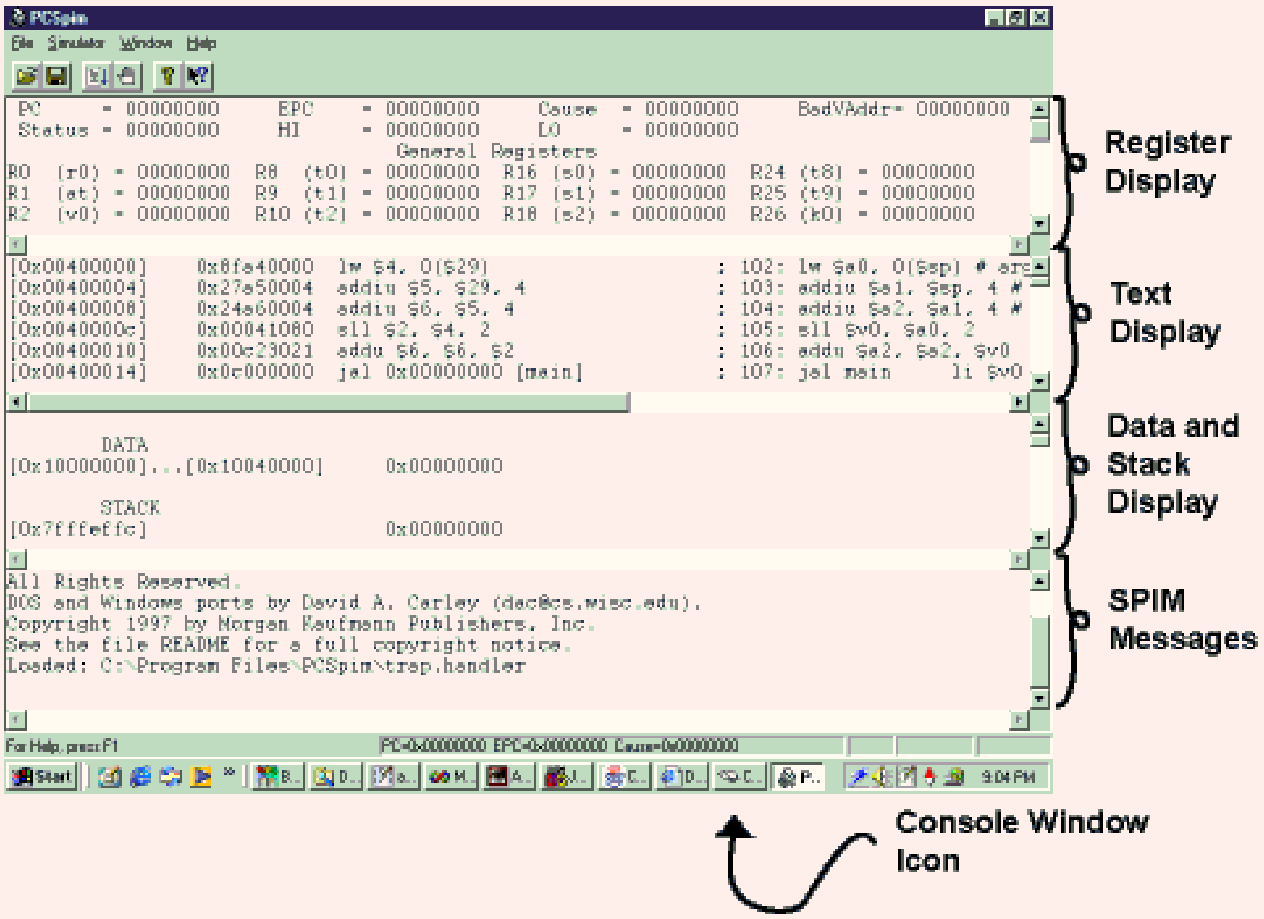
Next

A good answer might be:

You could use the Run command from the Start menu, or type *pcSpim* in a DOS window if *pcSpim.exe* is in the current directory.

Opening Window

On windows machines, the opening screen is as at right. The screen is divided into four parts:



```
 PC      = 00000000    EPC     = 00000000    Cause    = 00000000    BadVAddr= 00000000
 Status  = 00000000    HI      = 00000000    LO       = 00000000
                              General Registers
R0  (r0) = 00000000  R8  (t0) = 00000000  R16 (s0) = 00000000  R24 (t8) = 00000000
R1  (at) = 00000000  R9  (t1) = 00000000  R17 (s1) = 00000000  R25 (t9) = 00000000
R2  (v0) = 00000000  R10 (t2) = 00000000  R18 (s2) = 00000000  R26 (k0) = 00000000
```

```
[0x00400000]    0x8fa40000  lw $4, 0($29)                  ; 102: lw $a0, 0($sp) # arg
[0x00400004]    0x27a50004  addiu $5, $29, 4               ; 103: addiu $a1, $sp, 4 #
[0x00400008]    0x24a60004  addiu $6, $5, 4                ; 104: addiu $a2, $a1, 4 #
[0x0040000c]    0x00041080  sll $2, $4, 2                  ; 105: sll $v0, $a0, 2
[0x00400010]    0x00c23021  addu $6, $6, $2                ; 106: addu $a2, $a2, $v0
[0x00400014]    0x0c000000  jal 0x00000000 [main]          ; 107: jal main      li $v0
```

```
        DATA
[0x10000000]...[0x10040000]     0x00000000

        STACK
[0x7fffeffc]                    0x00000000
```

```
All Rights Reserved.
DOS and Windows ports by David A. Carley (dac@cs.wisc.edu).
Copyright 1997 by Morgan Kaufmann Publishers, Inc.
See the file README for a full copyright notice.
Loaded: C:\Program Files\PCSpim\trap.handler
```

```
PC=0x00000000  EPC=0x00000000  Cause=0x00000000
```

1. **Register Display**: This shows the contents (bit patterns in hex) of all 32 general purpose registers and the floating point registers.
2. **Text Display**: This shows the assembly language program source, the machine instructions (bit patterns in hex) they correspond to, and the addresses of their memory locations.
3. **Data and Stack Display**: This shows the sections of MIPS memory that hold ordinary data and data which has been pushed onto a stack.
4. **SPIM Messages**: This shows messages from the simulator (often error messages).

Text is the name for machine language bit patterns intended for eventual execution. The word "program" is often ambiguous, so "text" is used. "Text" is a machine language program waiting to become part of an executing process.

Character output from the simulated computer is in the SPIM console window, shown as an icon in the picture.

QUESTION 3:

Is there a difference between messages from *the simulator* and messages from *the simulated computer*?

Next

A good answer might be:

Yes.

Writing an Assembly Program

Messages from the *simulated computer* appear in the console window when an assembly program that is running (in simulation) writes to the (simulated) monitor. If a real MIPS computer were running you would see the same messages on a real monitor.

Messages from the *simulator* are anything the simulator needs to write to the user of the simulator. These are error messages, prompts, and reports.

Now that the simulator is running you need to assemble and load a program. Depending on the settings of the simulator, there already may be some machine instructions in simulated memory. This code assists in running your program. If you start the simulator from the Simulator menu this code will run, but it will be caught in an infinite loop. To stop it, click on Simulator; Break.

A **source file** (in assembly language or in any high level language) is the text file containing programming language statements created (usually) by a human programmer. An editor like *Notepad* will work. You will probably want to use a better editor, but as a common ground I'll use *Notepad*. (I like *Code-Genie* by DolfySoft, http://www.code-genie.com, but any text editor will be fine). Use whatever editor you use for Java (or for your usual programming language).

QUESTION 4:

(Review) What type of files does *Notepad* create?

Next

A good answer might be:

Text files---files of bytes that can be interpreted as ASCII characters.

Two plus Three

Word processors usually create "binary" files and so are not suitable for creating source files. They can be forced to output a text file, but a real programming editor is much nicer. With your program (text) editor create a file called *addup.asm*. (With most text editors and Web browsers you can *copy* the following code from the Web page and then *paste* into the editor).

```
## Program to add two plus three
        .text
        .globl  main

main:
        ori     $8,$0,0x2       # put two's comp. two into register 8
        ori     $9,$0,0x3       # put two's comp. three into register 9
        addu    $10,$8,$9       # add register 8 and 9, put result in 10

## End of file
```

The first "#" of the first line is in column one. The character "#" starts a comment; everything on the line from "#" to the right is ignored. Sometimes I use two in a row for emphasis, but only one is needed.

Each of the three lines following *main:* corresponds to one machine instruction.

QUESTION 5:

(Review:) What is a *machine instruction*?

Next

A good answer might be:

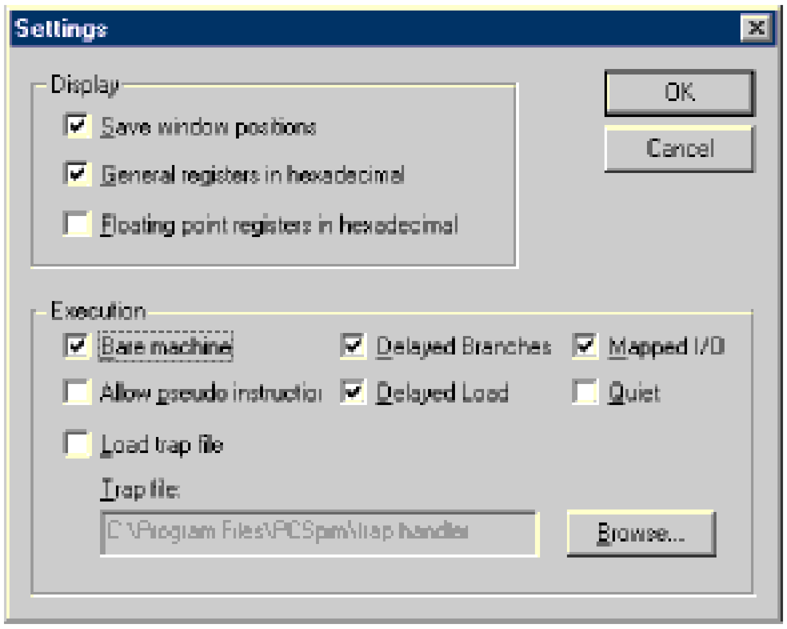
A *machine instruction* is a pattern of bits that asks for one machine operation to be executed.

Setting up SPIM

Each MIPS machine instruction is 32 bits (four bytes) long. The three lines after *main:* call for three machine instructions. The remaining lines consist of information for the assembler and comments (for the human).

For this first program some SPIM options must be set. In the menu bar, click on Simulator then Settings to get the settings dialog. Select the following options:

ON	Save window positions
ON	General registers in hexadecimal
OFF	Floating point registers in hexadecimal
ON	Bare machine
OFF	Allow pseudo instructions
OFF	Load trap file
ON	Delayed Branches
ON	Delayed Load
ON	Mapped I/O
OFF	Quiet



Display
- [✓] Save window positions
- [✓] General registers in hexadecimal
- [ ] Floating point registers in hexadecimal

Execution
- [✓] Bare machine
- [ ] Allow pseudo instructio
- [ ] Load trap file
- [✓] Delayed Branches
- [✓] Delayed Load
- [✓] Mapped I/O
- [ ] Quiet

Trap file: C:\Program Files\PCSpim\trap.handler

These settings simulate a bare machine with no user conveniences. Later we will include the conveniences.

QUESTION 6:

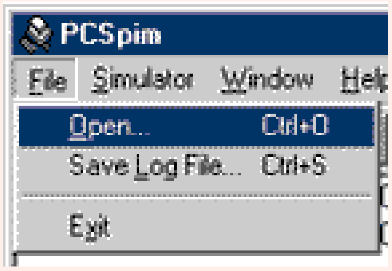
(Thought Question) Do most actual computers start up as a bare machine?

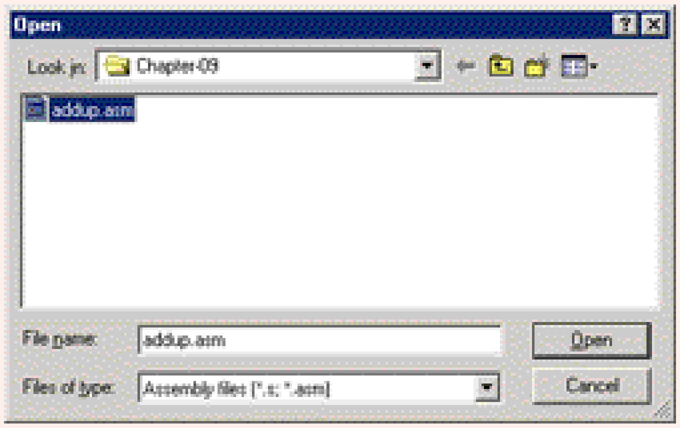
(Thought Question) Do most actual computers start up as a *Bare machine*?

A good answer might be:

No.

Loading the Source File





Usually there is some **firmware** (permanent machine code in EEPROM) in a special section of the address space. This starts running on power-up and loads an operating system. SPIM can simulate some basic firmware, but we have turned off that option.

Load the program into the SPIM simulator by clicking File then Open. Click on the name (*addup.asm*) of your source file. You may have to navigate through your directories using the file dialog box.

If there are mistakes in *addup.asm*, SPIM's message display panel shows the error messages. Use your editor to correct the mistakes, save the file then re-open the file in SPIM.

QUESTION 7:

Can the text editor and SPIM both be open at the same time?

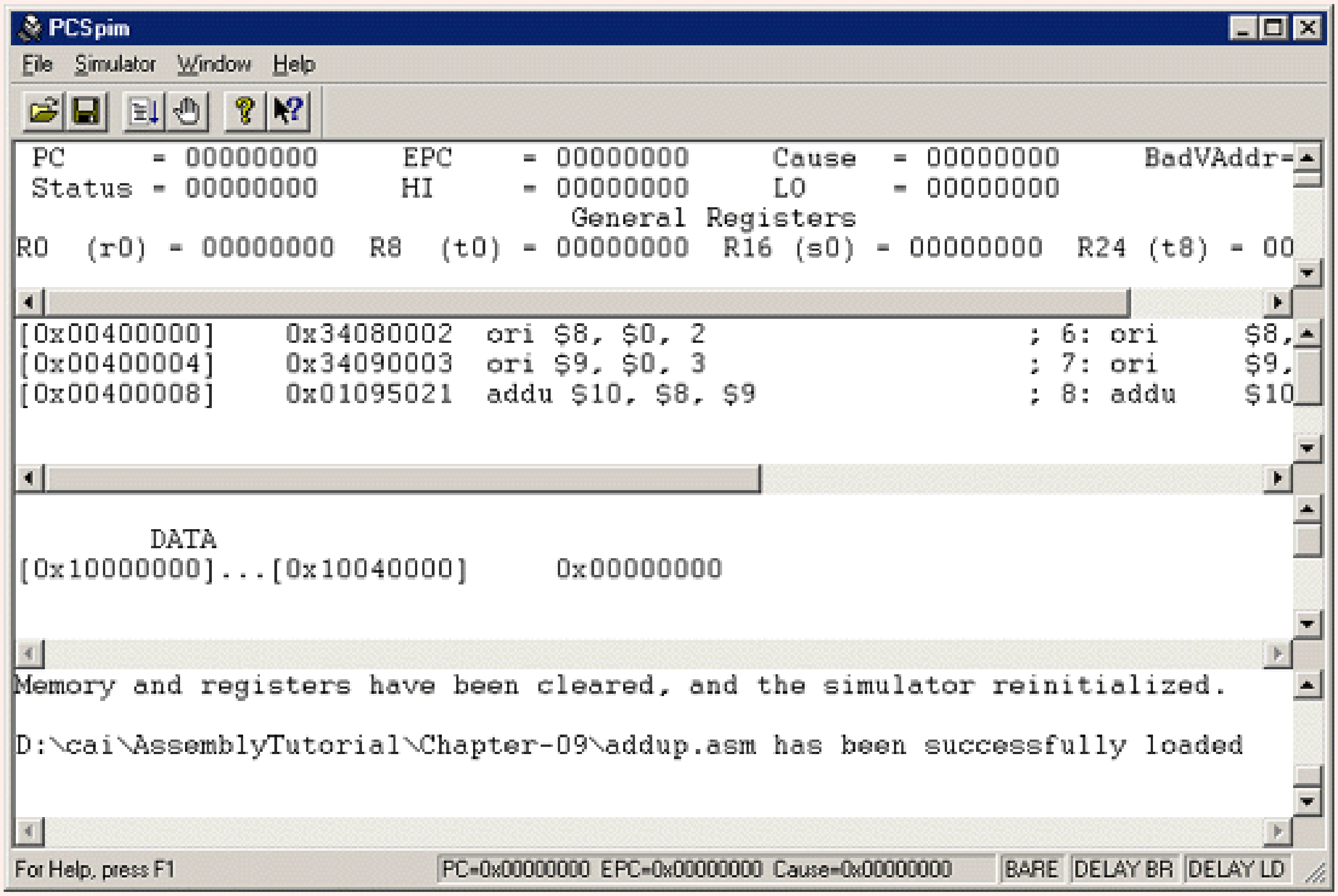
Next

A good answer might be:

Sure. You can correct errors with the text editor, save the file to the disk, and then re-load it into SPIM.

Assembling the Program

Loading the source file into SPIM does two things: (1) The file is assembled into machine instructions, and (2) the instructions are loaded into SPIM's memory. The *text display* shows the result.



```
 PC     = 00000000     EPC    = 00000000     Cause   = 00000000     BadVAddr=
 Status = 00000000     HI     = 00000000     LO      = 00000000
                              General Registers
R0  (r0) = 00000000  R8  (t0) = 00000000  R16 (s0) = 00000000  R24 (t8) = 00

[0x00400000]    0x34080002  ori $8, $0, 2                    ; 6: ori     $8,
[0x00400004]    0x34090003  ori $9, $0, 3                    ; 7: ori     $9,
[0x00400008]    0x01095021  addu $10, $8, $9                 ; 8: addu    $10

        DATA
[0x10000000]...[0x10040000]     0x00000000

Memory and registers have been cleared, and the simulator reinitialized.

D:\cai\AssemblyTutorial\Chapter-09\addup.asm has been successfully loaded

PC=0x00000000  EPC=0x00000000  Cause=0x00000000
```

The text display is the second window from the top. You should see some of the sorce file in it and the machine instructions they assembled into. The leftmost column are addresses in simulated memory.

QUESTION 8:

Inspect the text display (your own or the above picture).

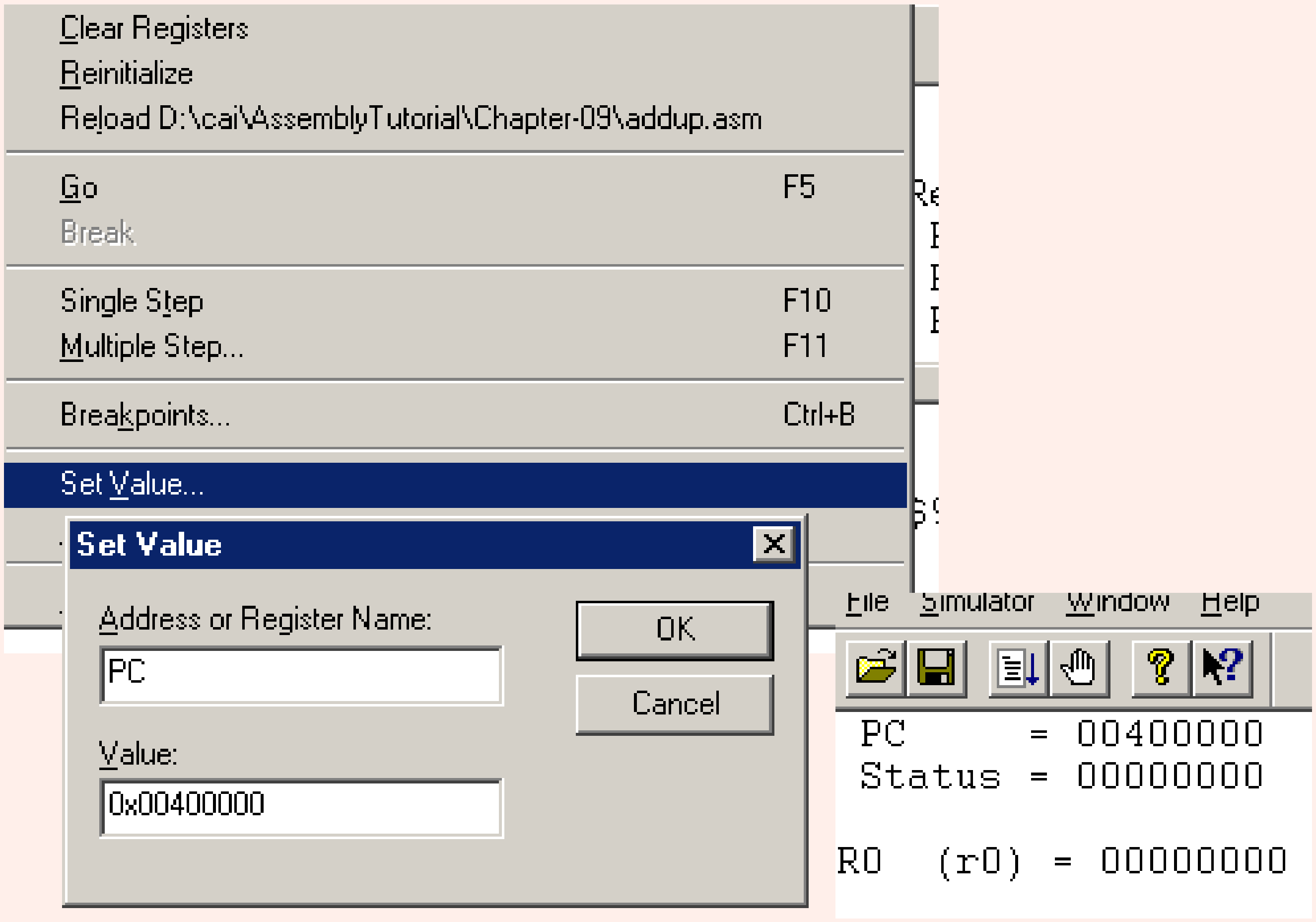
1. What machine instruction (bit pattern) did your first instruction (ori $8,$0,0x2) assemble into?
2. At what address in memory was it loaded?

A good answer might be:

1. What machine instruction (bit pattern) did your first instruction assemble into?
    - 0x34080002
2. At what address in memory was it loaded?
    - 0x00400000

Setting the PC

The



```
PC       = 00400000
Status   = 00000000

R0   (r0)  = 00000000
```

program counter is part of the processor that contains the address of (the first byte of) the current machine instruction. In the *register display* (top window) you see that it is initialized to zero. This must be changed to 0x00400000 , the address of our first instruction. To do this, select (click on) Simulator; Set Value in the menu bar.

In the set value dialog, type **PC** in the top text box and **0x00400000** in the bottom text box. Click on OK and the PC (in the register display) should change.

QUESTION 9:

A user types "400000" into the value box, and clicks OK. The PC changes to 00061a80. What happened?

Next

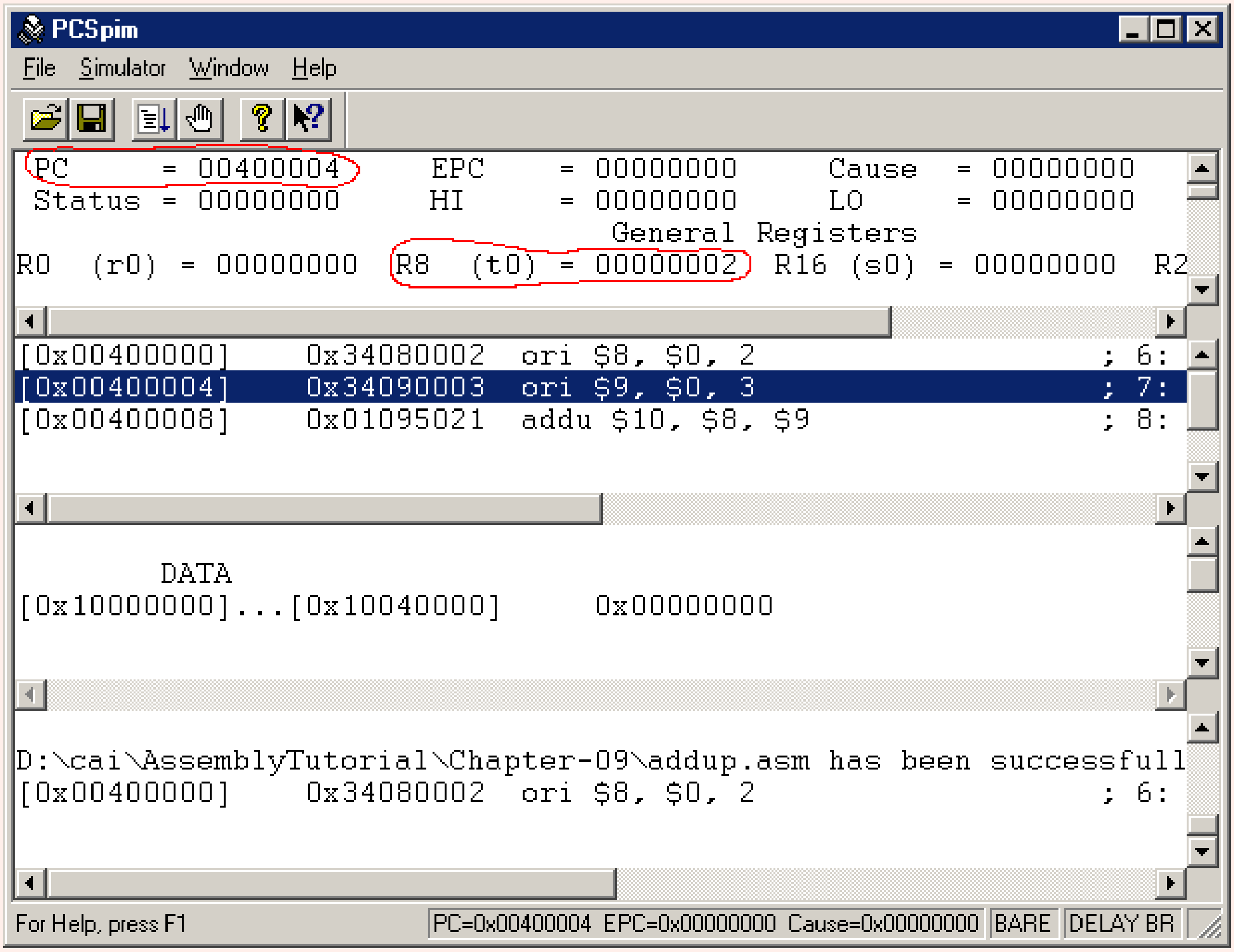
A user types "400000" into the value box, clicks OK, and sees the PC change to 00061a80. What Happened?

A good answer might be:

Without a leading "0x" the characters "400000" are taken to be an integer expressed in decimal.

Running the Program

Push F10 to execute one instruction. The first instruction executes, loading register R8 with a 2 (see the register display). The PC advances to the next instruction 0x00400004 and the message display window shows the instruction that just executed.



```
 PC      = 00400004     EPC     = 00000000     Cause   = 00000000
 Status  = 00000000     HI      = 00000000     LO      = 00000000
                              General Registers
R0   (r0) = 00000000  R8   (t0) = 00000002  R16 (s0) = 00000000  R2

[0x00400000]    0x34080002  ori $8, $0, 2                        ; 6:
[0x00400004]    0x34090003  ori $9, $0, 3                        ; 7:
[0x00400008]    0x01095021  addu $10, $8, $9                     ; 8:

         DATA
[0x10000000]...[0x10040000]     0x00000000

D:\cai\AssemblyTutorial\Chapter-09\addup.asm has been successfull
[0x00400000]    0x34080002  ori $8, $0, 2                        ; 6:

For Help, press F1     PC=0x00400004  EPC=0x00000000  Cause=0x00000000  BARE  DELAY BR
```

Push F10 two more times to execute the remaining instructions. Each instruction is 32 bits (four bytes) long, so the PC changes by four each time. After the third instruction, R8 will have the sum of two plus three.

QUESTION 10:

What is the sum of 0x0000002 and 0x0000003 ?

Next

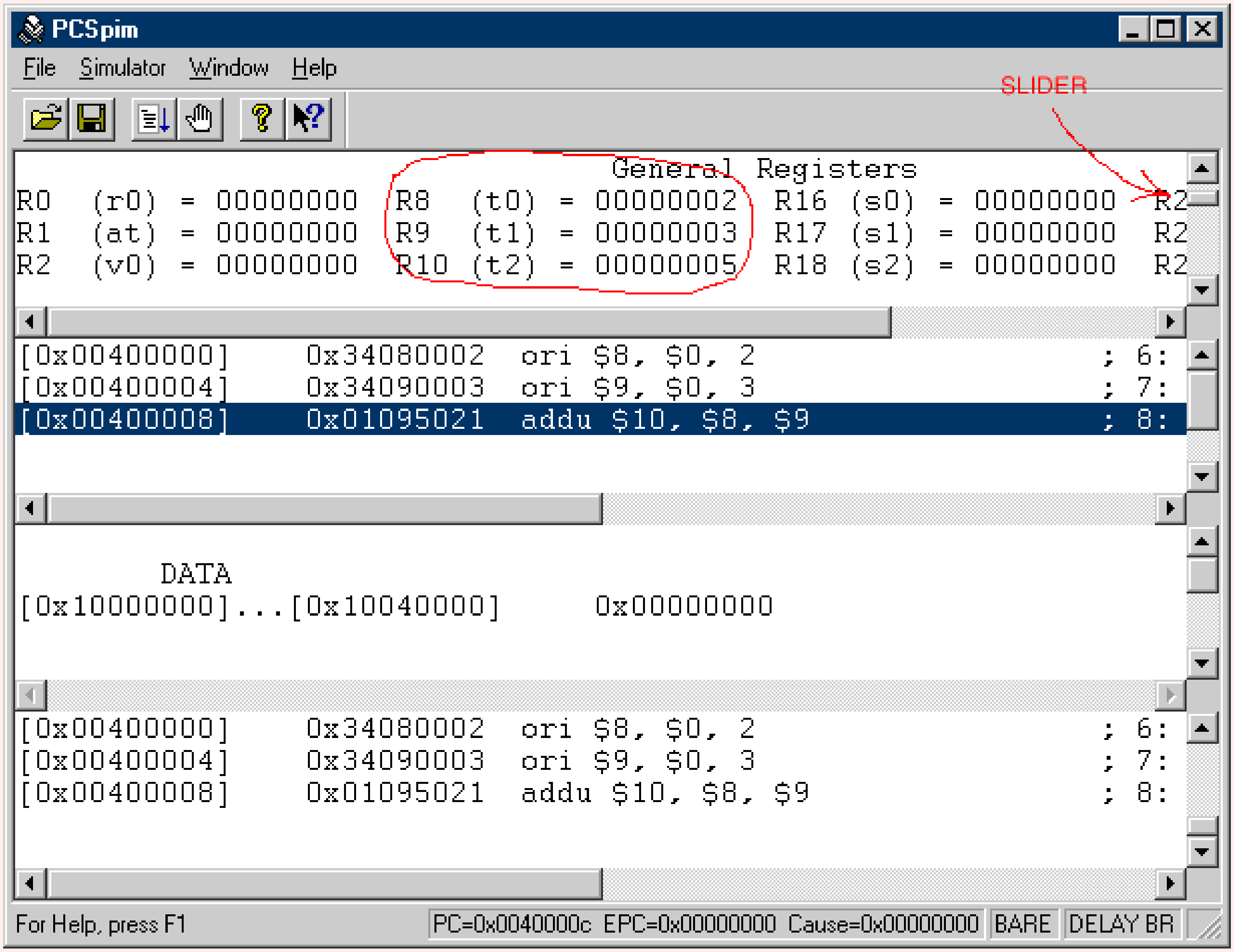
What is the sum of 0x0000002 and 0x0000003 ?

A good answer might be:

0x0000005

Program's Results

The bit patterns for these small integers are easy to figure out. You may have to use the slider on the register display to see register ten.



```
                              General Registers
R0   (r0)  = 00000000  R8   (t0)  = 00000002  R16  (s0)  = 00000000  R2
R1   (at)  = 00000000  R9   (t1)  = 00000003  R17  (s1)  = 00000000  R2
R2   (v0)  = 00000000  R10  (t2)  = 00000005  R18  (s2)  = 00000000  R2

[0x00400000]    0x34080002  ori $8, $0, 2                        ; 6:
[0x00400004]    0x34090003  ori $9, $0, 3                        ; 7:
[0x00400008]    0x01095021  addu $10, $8, $9                     ; 8:

        DATA
[0x10000000]...[0x10040000]     0x00000000

[0x00400000]    0x34080002  ori $8, $0, 2                        ; 6:
[0x00400004]    0x34090003  ori $9, $0, 3                        ; 7:
[0x00400008]    0x01095021  addu $10, $8, $9                     ; 8:

For Help, press F1    PC=0x0040000c  EPC=0x00000000  Cause=0x00000000  BARE  DELAY BR
```

If you push F10 again, the PC will point at a word in memory that contains bits not intended to be a machine instruction. However the simulator will try to execute those bits. A real processor would "crash" at this point. (This is sometimes called "falling off the end of a program"). The simulator prints an error message in the bottom panel.

QUESTION 11:

What happens in real computers when the end of a program is reached?

Next

A good answer might be:

The program returns control to the operating system.

Explanation of the Program

There are various ways for a program executing on a real machine to return control to the operating system. But we have no OS, so for now we will single step instructions. Hopefully you are wondering how the program works. Here it is again:

```
## Program to add two plus three
        .text
        .globl  main

main:
        ori     $8,$0,0x2       # put two's comp. two into register 8
        ori     $9,$0,0x3       # put two's comp. three into register 9
        addu    $10,$8,$9       # add register 8 and 9, put result in 10

## End of file
```

## Program to add two plus three is a comment. It is ignored by the assembler and results in no machine instructions.

.text is a *directive*. A directive is a statement that tells the assembler something about what the programmer wants, but does not itself result in any machine instructions. This directive tells the assembler that the following lines are ".text" -- source code for the program.

.globl main is another directive. It says that the identifier *main* will be used outside of this source file (that is, used "globally") as the label of a particular location in main memory.

QUESTION 12:

(Memory test) Where was the first machine instruction placed in memory?

Next

(Memory test) Where was the first machine instruction placed in memory?

A good answer might be:

0x00400000

Explanation Continued

The blank line is ignored. Next comes **main:** This defines a **symbolic address** (sometimes called a **statement label**). A symbolic address is a symbol (an identifier) that is the source code name for a location in memory. In this program, **main** stands for the address of the first machine instruction (which turns out to be 0x00400000). Using a symbolic address is much easier than using a numerical address. With a symbolic address, the programmer refers to memory locations by name and lets the assembler figure out the numerical address.

```
## Program to add two plus three
        .text
        .globl  main

main:
        ori     $8,$0,0x2       # put two's comp. two into register 8
        ori     $9,$0,0x3       # put two's comp. three into register 9
        addu    $10,$8,$9       # add register 8 and 9, put result in 10

## End of file
```

The symbol main is global. This means that several source files are able to use main to refer to the same main storage address.

QUESTION 13:

(Test your intuition:) Is it likely that several sections of a software system need to refer to each other?

Next

Is it likely that several sections of a software system need to refer to each other?

A good answer might be:

Yes.

Loading Register Eight

Yes. All software consists of interconnected modules. A connection is made when one module refers to an address in another.

The next line: **ori $8,$0,0x2** translates into a 32-bit machine instruction. The machine instruction, upon execution, puts a 32-bit two's complement positive two into register eight (details later).

```
## Program to add two plus three
        .text
        .globl  main

main:
        ori     $8,$0,0x2       # put two's comp. two into register 8
        ori     $9,$0,0x3       # put two's comp. three into register 9
        addu    $10,$8,$9       # add register 8 and 9, put result in 10

## End of file
```

The instruction after that translates into a machine instruction that (upon execution) puts a three into register nine. The final instruction translates into a machine instruction that (upon execution) adds register eight to register nine and puts the 32-bit result into register ten.

QUESTION 14:

What algorithm do you suppose is used for the 32-bit addition?

What algorithm do you suppose is used for the 32-bit addition?

A good answer might be:

The Binary Addition Algorithm (of Chapter 8)

Run Time

It is awkward to keep saying *"the instruction, after it is assembled and loaded into main memory, upon execution does ...."* Instead one says *"at **run time** the instruction does ..."*. For example,

At run time, the instruction **ori $8,$0,0x2** loads register eight with a two.

Even sloppier is,

"the instruction **ori $8,$0,0x2** loads register eight with a two."

It is vital that you understand that this phrase is a short way of saying the longer phrase. In a computer it is the bit patterns of machine instructions that cause things to happen, and things happen only at run time.

Sometimes one talks about **assembly time**, the phase where the assembler is creating bit patterns out of the source file.

QUESTION 15:

(Review) In a typical computer system (not a simulated one) where does the assembler place the bit patterns it creates?

Next

(Review) In a typical computer system (not a simulated one) where does the assembler place the bit patterns it creates?

A good answer might be:

In an object module, stored as a disk file.

End of Chapter

With the SPIM simulator the object module step is skipped. The assembler puts the bit patterns directly into (simulated) main memory. To end SPIM, click on File; exit in the menu bar.

You have reached the end of this chapter. You may wish to review the following. Click on a blue subject that interests you to go to where it was discussed.

- The SPIM user interface.
- Starting SPIM
- Source files
- Settings dialog
- Loading the source file.
- The two things that SPIM does to a loaded file.
- Setting the program counter
- Single stepping SPIM
- Assembly language directive
- Symbolic address
- Run time

Click here Go to Table of Contents

You have reached the end of the chapter.

created: 06/09/01

CHAPTER 10 --- A MIPS Programming Model

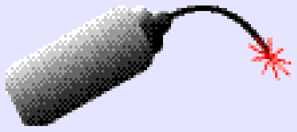
This chapter presents a *programming model* for the MIPS processor. A programming model is an abstract view of a processor that is appropriate for programming but omits details that are not needed for that task. It is the view of the machine a programmer uses when programming.

Chapter Topics:

- *Basic MIPS Programming model.*
- *Memory*
- *Registers*
- *Machine cycle*
- *Control flow*

QUESTION 1:

As you drive your car down the highway, do you think much about the ignition coil?



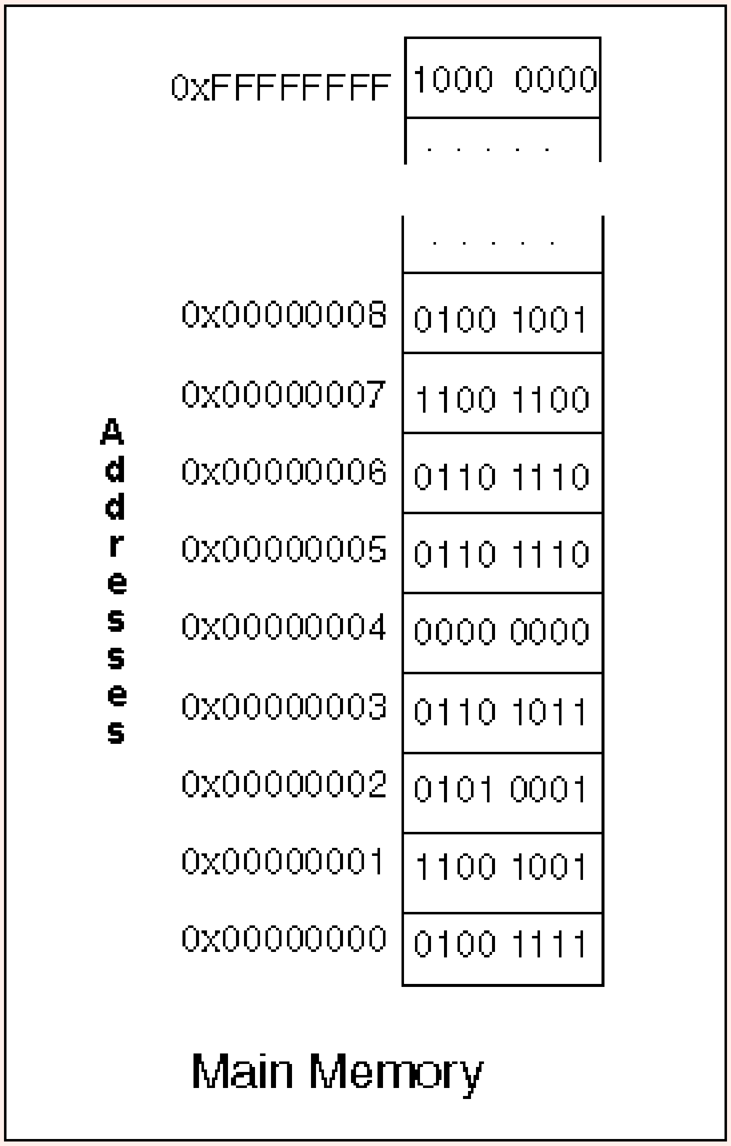
Next

A good answer might be:

No•you use an abstract "drivers view" of the car which is appropriate for driving but omits unnecessary details. Of course you don't *always* use that model. If it starts raining and your car starts sputtering you might start thinking about the coil.

Memory Model

MIPS computer systems nearly always use cache and virtual memory. But our abstract view of memory does not include them. The purpose of virtual memory is to make it appear as if a program has the full address space available. So our abstract view of memory has the full address space. The purpose of cache is to transparently speed up memory access. So our abstract view of memory does not show cache. Our abstract view is as follows:



DATA:

MIPS memory is an array of 2^{32} bytes. Each byte has a 32-bit address. Each byte can hold an 8-bit pattern, one of the 256 possible 8-bit patterns. The addresses of MIPS main memory range from 0x00000000 to 0xFFFFFFFF.

However user programs and data are restricted to the first 2^{31} bytes. The last half of the address space is used for specialized purposes.

OPERATIONS:

- **Load:** a bit pattern starting at a designated address in memory is copied into a register

inside the processor.
- **Store:** a bit pattern is copied from a processor register to memory at a designated address.

Bit patterns are copied to/from memory and the processor in groups of one, two, four, or eight contiguous bytes. Only the address of the first byte of memory is specified.

QUESTION 2:

- Does a **load** operation change the bit pattern in memory?
- Does a **store** operation change the bit pattern in memory?

Next

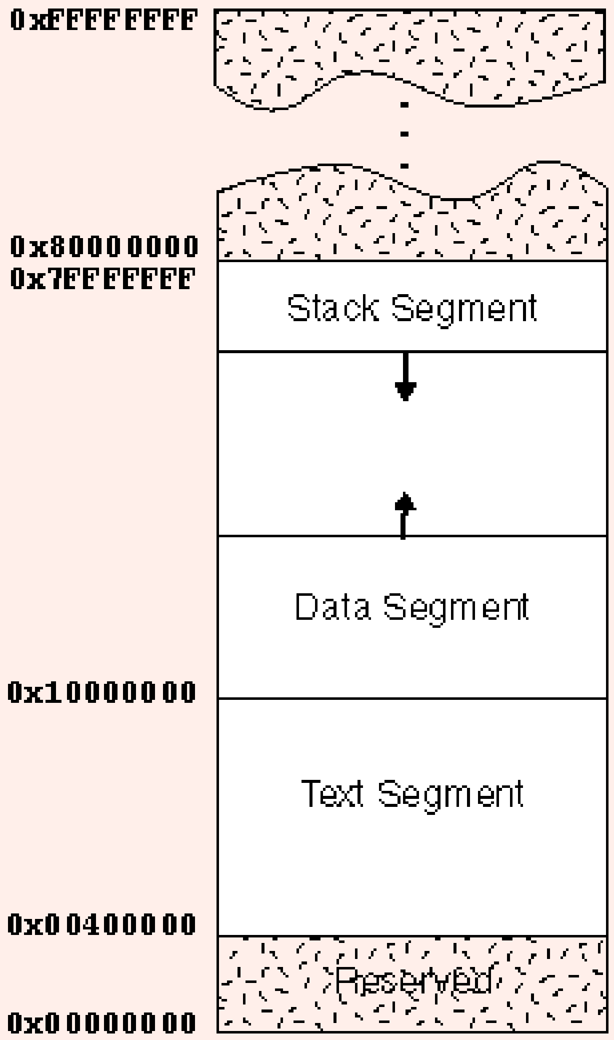
A good answer might be:

- Does a **load** operation change the bit pattern in memory? NO
- Does a **store** operation change the bit pattern in memory? YES

Memory Layout

Load and store operations copy the bit pattern from the source into the destination. The source (register or memory) does not change. Of course, the pattern at the destination is replaced by the pattern at the source.

However, it is convenient for programmers and systems software to organize memory so that instructions and data are separate. At right is the way MIPS systems often lay out their memory.



Although the address space is 32 bits, the top addresses from 0x80000000 to 0xFFFFFFFF are not available to user programs. They are used for the operating system and for ROM. When a MIPS chip is used in an embedded controller the control program probably exists in ROM in this upper half of the address space

The parts of address space accessible to a user program are divided as follows:

Text Segment: This holds the machine language of the user program (the *text*).

Data Segment: This holds the data that the program operates on. Part of the data is *static*. This is data that is allocated by the assembler and whose size never changes. Values in it do change; "static" means the size in bytes does not change during execution. On top of the static data is the *dynamic* data. This is data that is allocated and deallocated as the program executes. In "C" dynamic allocation and deallocation is done with *malloc()* and *free()*.

Stack Segment: At the top of user address space is the stack. With high level languages, local variables and parameters are pushed and popped on the stack as procedures are activated and deactivated.

QUESTION 3:

(Thought Question) As the program runs, the data segment grows upward (as dynamic variables are allocated) and the stack grows downward (as procedures get called). Is this sensible? (Hint: how much user memory is left when the two segments meet?)

Next

As the program runs, the data segment grows upward (as dynamic variables are allocated) and the stack grows downward (as procedures get called). Is this sensible?

A good answer might be:

Yes. Rather than allocate a fixed amount of memory for each, this arrangement means each can grow into available memory. When the two meet, there is no memory left.

Registers

Often several bytes in a row (contiguous bytes) are used, so there is terminology for how many bytes are used. Each manufacturer has their own idea of what to call groupings larger than a byte. The following is used for MIPS chips.

- *byte* — eight bits.
- *word* — four bytes, 32 bits.
- *double word* — eight bytes, 64 bits.

A block of contiguous memory is referred to by the address of its first byte (ie. the byte with the lowest address.) Most MIPS instructions involve a fixed number of bytes (for example,)

Often you need a number of bits other than one of the standard amounts. Use the next largest standard amount, and remember to be careful. Attempting to use the very minimum number of bits is more complicated than it is worth and is a rich source of errors in assembly language programming.

A **register** is a part of the processor that can hold a bit pattern. On the MIPS, a register holds 32 bits. There are many registers in the processor, but only some of them are visible in assembly language. The others are used by the processor in carrying out its operations.

- A **load** operation copies a bit pattern from memory into a register.
- A **store** operation copies a bit pattern from a register into memory.

The registers that are visible in assembly language are called **general purpose** registers and **floating point** registers. There are 32 32-bit general purpose registers. In assembly language, these registers are named $0, $1, $2, ... , $31. There are 16 floating point registers. They will be discussed later.

One of the general purpose registers is hard-wired to always contain the value 0x00000000 (all zero bits).

QUESTION 4:

Which register $0, $1, $2, ... , $31 do you suppose always contains all zero bits?

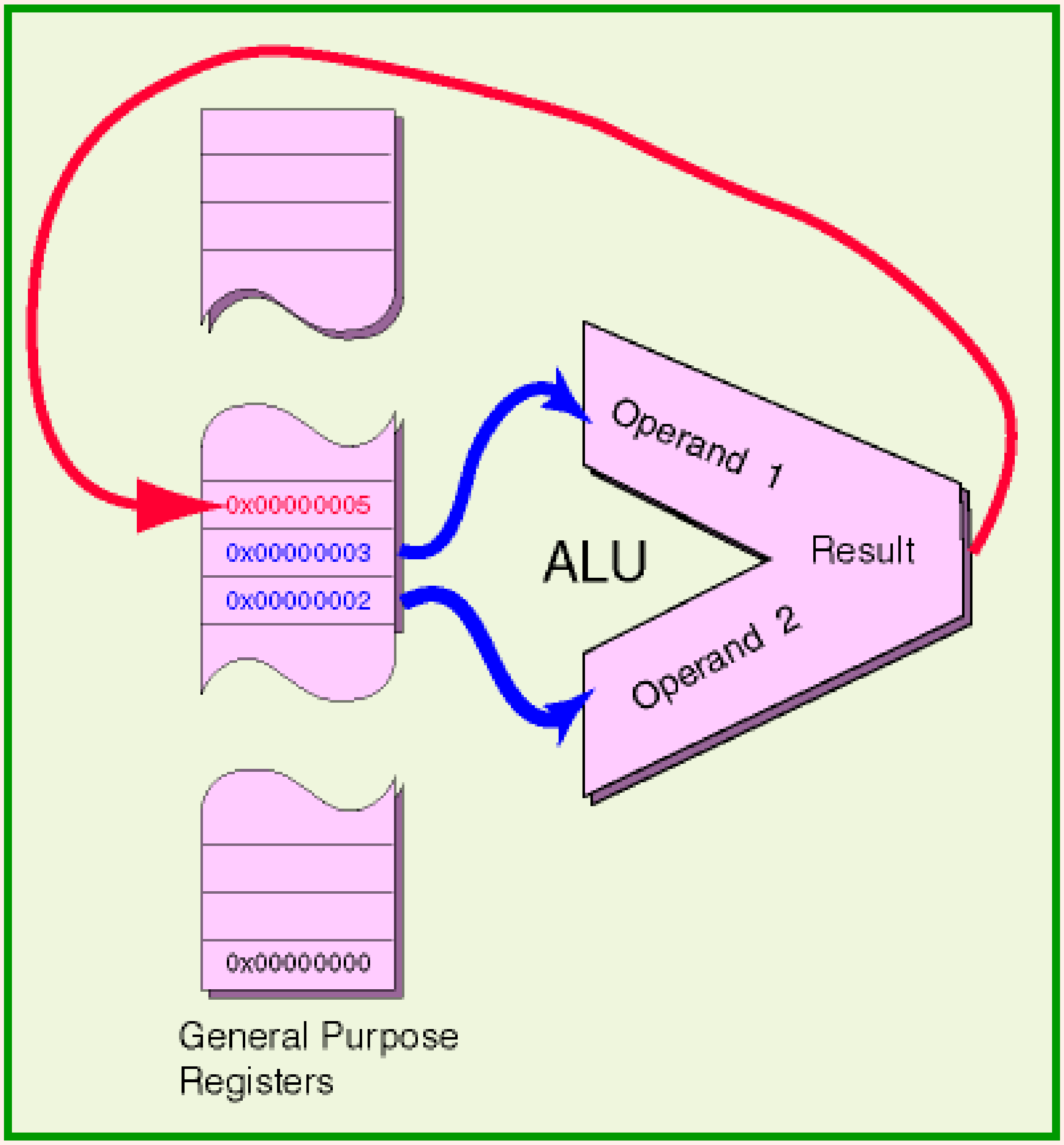
Next

A good answer might be:

$0

The bit pattern 0x00000000 occurs so frequently in machine language that register $0 is wired to permanently hold it. That bit pattern represents the integer zero, a very common integer. It also represents a **null**, which is used to mark the end of character strings and some data structures.

Registers and the ALU



The **arithmetic/logic unit (ALU)** of a processor performs integer arithmetic and logical operations. For example, one of its operations is to add two 32-bit integers. An integer used as input to an operation is called an **operand**. One operand for the ALU is always contained in a register. The other may be in a register or may be part of the machine instruction. The result of the operation is put into a general purpose register.

Machine instructions that use the ALU specify four things:

1. The operation to perform.

2. The the first operand (often in a register).
3. The second operand (often in a register).
4. The register that receives the result.

The picture shows a 32-bit addition operation. The operands are register $8 and register $9. The result in put in register $10. Here is how that instruction is written as assembly language:

```
addu    $10,$8,$9
```

QUESTION 5:

Here is another instruction that involves the ALU, written as assembly language:

Next

```
subu    $25,$16,$17
```

Identify the following:

1. The operation: _________________
2. The location of one operand: _________________
3. The location of the other operand: _________________
4. The destination of the result: _________________

A good answer might be:

1. The operation: subtraction of two 32-bit integers
2. The location of one operand: register $16
3. The location of the other operand: register $17
4. The destination of the result: register $25

At run time: Register $25 <-- (Register $16) - (Register $17)

Machine Code

On some processors (such as the VAX) the ALU can operate directly on data contained in main memory. However, this requires extra data paths and control logic and slows down execution. On the MIPS, operands for the ALU and the destination for the result are inside the chip (usually registers).

Here is the assembly language for the add instruction:

```
addu    $10,$8,$9
```

Here is the machine code it translates into:

```
0x01095021
```

Here is that as a bit pattern:

```
0000 0001 0000 1001 0101 0000 0010 0001
```

Here is the bit pattern arranged into different groups (as given in the MIPS documentation). Under each group is what the pattern means as a machine instruction. (A group of bits that is part of a larger bit pattern is often called a **field**). Each field of a machine instruction has a specific function.

```
  0    1    0    9    5    0    2    1     -- machine instruction in hex

0000 0001 0000 1001 0101 0000 0010 0001    -- machine instruction in bits

000000 01000 01001 01010 00000 100001      -- fields of the instructuion
```

```
opcode oprnd oprnd dest  ----- 2ndary      -- meaning of the fields

ALUop   $8    $9   $10          addu
```

Inspect the groups until you see how the instruction specifies the four things: (1) the operation, (2) operand 1, (3) operand 2, (4) result destination. The 12 bits that specify the **addu** operation are in two groups at the start and end of the instruction. The first group is called the **opcode** and the second group is called the **secondary opcode**. The register numbers are readable as binary integers. Some bits of the instruction are left as zero. You have to look at MIPS documentation to figure all this out.

QUESTION 6:

Here is the machine instruction again, but the pattern that specifies the destination is blank. Fill it in so the destination is register $11.

Next

```
000000 01000  01001 _ _ _ _ _ 00000 100001

addu    $8    $9    $11           addu
       op1   op2    dest
```

A good answer might be:

```
000000 01000  01001 0 1 0 1 1 00000 100001

addu    $8    $9    $11          addu
       op1   op2   dest
```

Register Use Conventions

Congratulations! (if you correctly answered the question). You have just done some machine language programming. Who needs that old assembler, anyway?

General purpose registers are those that assembly language programs work with (other than floating point registers). The general purpose registers are numbered $0 through $31. However, by convention (and sometimes by hardware) different registers are used for different purposes.

Register Number	Mnemonic Name	Conventional Use
$0	zero	Permanently 0
$1	$at	Assembler Temporary (reserved)
$2, $3	$v0, $v1	Value returned by a subroutine
$4-$7	$a0-$a3	Arguments to a subroutine
$8-$15	$t0-$t7	Temporary (not preserved across a function call)
$16-$23	$s0-$s7	Saved registers (preserved across a function call)
$24, $25	$t8, $t9	Temporary
$26, $27	$k0, $k1	Kernel (reserved for OS)
$28	$gp	Global Pointer
$29	$sp	Stack Pointer
$30	$fp	Frame Pointer
$31	$ra	Return Address (Automatically used in some instructions)

In addition to a

number $0 • $31, registers have a mnemonic name (a name that reminds you of its use). For example register $0 has the mnemonic name *zero*. The table shows the 32 registers and their conventional use.

Registers $0 and $31 are the only two that behave differently from the others. Register $0 is permanently wired to contain zero bits. Register $31 is automatically used by some subroutine linkage instructions to hold the return address.

If this looks totally cryptic, don't worry. Don't try to memorize this. You will get used to what you need to know after writting a few programs.

QUESTION 7:

A program has just calculated an important value which is contained in register $8. The program now calls a function to print out the value. What might be in $8 upon return?

Next

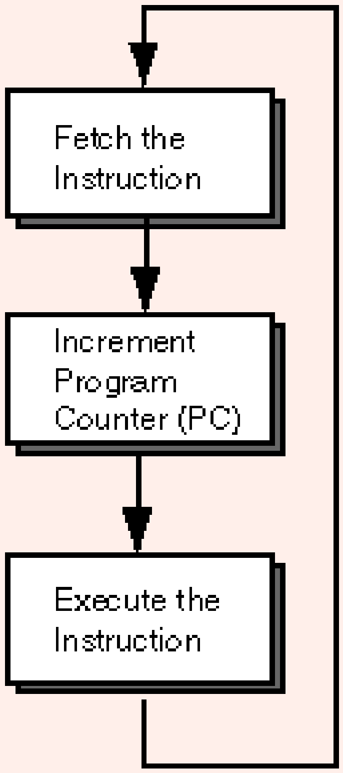
A program has just calculated an important value which is contained in register $8. The program now calls a function to print out the value. What might be in $8 upon return?

A good answer might be:

The register will contain an unpredictable value, since $8 is also $t0 a non-preserved temporary.

Model Machine Cycle

Of course as long as an executing program makes no function calls the values in the temporary registers are safe.



The MIPS assembly programming model includes a machine cycle (repeated here from chapter one). The MIPS endlessly cycles through three basic steps. Each cycle executes one machine instruction. Everything the processor does is done by a sequence of machine operations. So, also, everything any program does is ultimately done by machine operations.

Fetch the next Instruction. The program counter contains the address of the next machine instruction. The instruction is fetched from memory.

Increment the PC. The address in the program counter is incremented by four.

Execute the Instruction. The machine operation specified by the instruction is performed.

QUESTION 8:

Why is the program counter (PC) incremented by four each time?

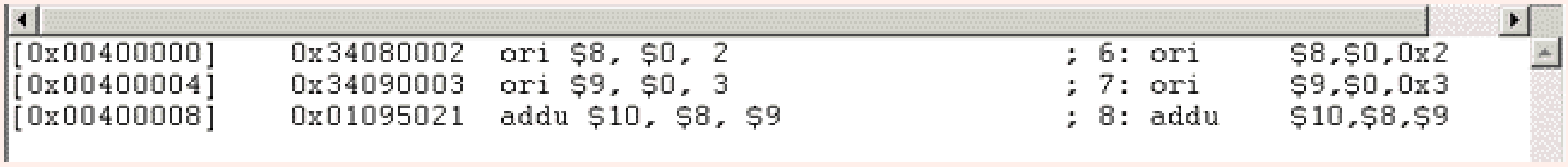
A good answer might be:

Each MIPS instruction is 32 bits (four bytes) long. The PC is incremented to point at the next instruction.

Sequential Execution

MIPS machine instructions are all 32 bits wide (4 bytes). Normally, instructions are executed one after another in the same sequence that they have been placed in memory, starting with the lowest address and going up. The execution sequence can be changed with a *branch* or a *jump* machine instruction.

Here, for example, is the program from the previous chapter. The three machine instructions have been placed at locations 0x00400000, 0x00400004, and 0x00400008, and are executed in that order.



```
[0x00400000]    0x34080002  ori $8, $0, 2                ; 6: ori     $8,$0,0x2
[0x00400004]    0x34090003  ori $9, $0, 3                ; 7: ori     $9,$0,0x3
[0x00400008]    0x01095021  addu $10, $8, $9             ; 8: addu    $10,$8,$9
```

QUESTION 9:

After the instruction at 0x00400008 has executed, what happens?

After the instruction at 0x00400008 has executed, what happens?

A good answer might be:

The processor fetches the 32-bit pattern (whatever it is) at address 0x0040000C and tries to execute it as an instruction.

Control

A bit pattern that is fetched as an instruction is interpretted as an instruction. The bits determine what is done in the next machine cycle. If the pattern makes no sense as an instruction then normal control flow is interrupted. If it can be interpretted as an instruction then it will be executed, whatever it does.

The **control point** of an executing program is the position of the instruction being executed. When an instructiion is being executed (in the third step of the machine cycle) the program counter holds the address of the instruction after the control point.

Normally the control point moves sequentially through the machine instructions. On the MIPS this means it normally moves through memory in steps of four bytes (32 bits) at a time. Usually "control point" is shortened to "control" and the phrase *flow of control* means how control moves through memory.

If control flow leads to an address in memory, then the four bytes starting at that address are fetched as a machine instruction. The processor has no other way to tell instructions from data. It is common for the control point of a buggy program to enter a section of data. This sometimes leads to mystifying results.

By software convention, data and instructions are placed in different sections of memory. (This helps prevent mystifying results). But this is not a requirement of the architecture.

QUESTION 10:

Most computer systems start running an operating systems when power is applied. When an application program runs, the operating system passes control to the application.

Next

What must the application do when it is finished running?

What must the application do when it is finished running?

A good answer might be:

Pass control back to the operating system.

Multitasking

With DOS, programs were passed control, ran until completion, then passed control back to DOS. So only one application ran at a time. Worse, an application that messed up might never return control to DOS and the whole system would freeze.

Modern computer systems (1960 and later for mainframes, from 1978 for microcomputers, from about 1995 for Windows PCs) include features that can interrupt the control flow of an executing program. After the flow is interrupted, the operating system can give control to another application. It this is done many times a second the computer system appears to be simultaneously executing several applications. This trick is called **multitasking**.

The MIPS chip has very good support for multitasking. But this is an advanced topic not included in our basic programming model.

QUESTION 11:

Do human brains do multitasking?

Next

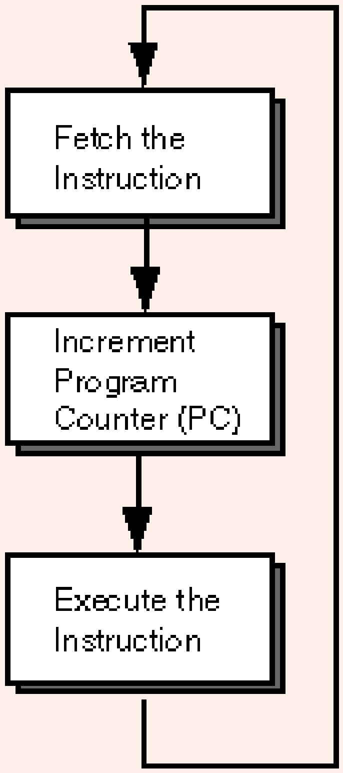
Are you smarter than DOS?

A good answer might be:

Human brains work (mostly) in parallel. Multitasking gives the *appearance* of parallel processing. Brains and some computers actually do it.

Model Summary

Here is a summary of the basic MIPS programming model. Click on the hyperlinks for more details.



Machine Instructions: Machine instructions are thirty-two bits wide. Bit patterns specify the operation, the operands, and the destination for the result. Basic operations are arithmetic, logic, memory access, and control branches.

Machine Cycle: The machine cycle is illustrated at right. Execution proceeds sequentially one instruction at a time. The control point indicates the instruction about to execute. ALU operations never directly access memory.

Registers: Thirty-two 32-bit wide general purpose registers, which have various conventional uses. Register $0 is hard-wired to 32 zero-bits.

Memory: Thirty-two bit address space, but only the lower half (most of it anyway) is for user programs. User memory is further divided (by software convention) into *text*, *data*, and *stack* segments.

contents Go to Table of Contents

You have reached the end of the chapter.

created: 07/04/01

CHAPTER 11 — Immediate Operands and Bitwise Logic

Some machine instructions contain data as part of the instruction. This data is an *immediate operand.* The **ori** instruction used in the previous chapter contains an immediate operand.

In a bitwise logic operation two bit patterns are "lined up" and then an operation such as OR or AND is performed between pairs of bits. This chpater discusses several such operations and them discusses some of the MIPS *addition* operations.

Chapter Topics:

- *Immediate Operands.*
- *Bitwise Logic.*
- *Bitwise OR, AND, and XOR*
- *Format of Machine Instructions.*
- *Zero Extension of a Bit Pattern.*
- *OR Immediate with $0 as a Load Instruction.*

QUESTION 1:

Could a 32-bit MIPS instruction hold a 16-bit two's complement integer?

Next

A good answer might be:

Yes.

Immediate Operand

A machine instruction can use some of its bits as one of the operands for a machine operation. This is called an **immediate operand**. For example, here is one of the instructions from the previous example program:

```
address       machine code    assembly code
------------  --------------  --------------
0x00400000    0x34080002      ori $8,$0,0x2
                    ----
```

OR Operation on Bits				
first operand	0	0	1	1
second operand	0	1	0	1
	—	—	—	—
result	0	1	1	1

The last 16 bits (four nibbles) of the machine instruction contain the operand 0x0002. (The assembly language instruction can just say "0x2"). The machine instruction tells the ALU to perform a **bitwise OR** between the contents of register $0 and the immediate operand 0x0002. The result is put in register $8.

A **bitwise** operation is where a logical operation is performed on the bits of each column of the operands. Here is the bitwise OR between two 8-bit patterns:

```
0110 1100    operand
0101 0110    operand
---------
0111 1110    result
```

QUESTION 2:

- What bit pattern is contained in register $0 ?
- How many bits are there per MIPS instruction?

Next

A good answer might be:

- What bit pattern is contained in register $0 ?
  - All zero bits
- How many bits are there per MIPS instruction?
  - 32 bits

Zero Extension

Here is the instruction we are looking at:

```
address       machine code    assembly code
__________________________    __________________________    ______________________________

0x00400000    0x34080002      ori $8,$0,0x2
                ----
```

Sixteen bits of immediate operand **0000 0000 0000 0010** are to be bitwise ORed with thirty-two bits of register zero, **0000 0000 0000 0000 0000 0000 0000 0000**. This would not ordinarily be possible. However, MIPS **zero extends** the sixteen-bit operand so the operands are the same length. Sometimes this is called *padding with zeros* on the left.

```
 zero extension
---- ---- ---- ----
0000 0000 0000 0000 0000 0000 0000 0010   -- zero extended immediate operand
0000 0000 0000 0000 0000 0000 0000 0000   -- data in register $0
---------------------------------------
0000 0000 0000 0000 0000 0000 0000 0010   -- result, put in register $8
```

An OR operation is done in each column. The 32-bit result is placed in register $8.

QUESTION 3:

How are the immediate operand 0x2 and the final contents of $8 related?

Next

How are the immediate operand 0x2 and the final contents of $8 related?

A good answer might be:

The 16-bit 0x2 immediate operand has been zero-extended and copied into register $8.

OR Immediate Instruction

Because the OR operation was done with the zeros in register $0, the result was a copy of the zero-extended immediate operand. Copying a bit pattern into a register is usually called **loading** the register. Register $8 was *loaded* with a 32-bit pattern. The pattern could represent a postive two. If so, register $8 was loaded with postive two. Here is a description of the **ori** instruction when used to load a register:

```
ori  d,$0,const    # register d <-- const.
                   #     const is a 16-bit pattern, so
                   #     0x0000 ... const ... 0xFFFF
```

If `const` represents an integer, then 0 <= `const` <= 65535. The three operands of the assembly instruction d, $0, and `const` must appear in that order. Here is a description of the **ori** instruction used as a bitwise OR instruction:

```
ori  d,s,const    # register d <-- bitwise OR of const
                  #     with the contents of register $s
                  #     const is a 16-bit pattern, so
                  #     0x0000 ... const ... 0xFFFF
```

The three operands of the instruction must appear in the correct order, and `const` must be within the specified range. If the immediate operand in a source instruction is less than sixteen bits (such as 0x2) the assembler expands it to sixteen. If it is more than sixteen bits the assebler writes an error message.

QUESTION 4:

Is the following instruction correct?

```
ori  $0,$9,0x32
```

Next

A good answer might be:

```
ori  $0,$9,0x32
```

Incorrect. It says to put the result into register $0 (which is always zero and can't be loaded with anything else).

OR Immediate used with Non-zero Operands

The assembler part of the SPIM simulator does not write an error message for the above mistaken instruction. But the instruction does not change register zero when executed.

Here is a tiny program that bitwise ORs two patterns. First one pattern is loaded into register $8, then the register is ORed with an immediate operand. The result goes into register $10.

```
## Program to bitwise OR two patterns
        .text
        .globl  main

main:
        ori      $8,$0,0x0FA5       # put first pattern into register $8
        ori     $10,$8,0x368F       # or ($8) with second pattern.  Result to $10.

## End of file
```

We'll assemble and run the program in a bit. You can do this now, if you want (remember you can use copy-and-paste from this page to your text editor). First, let us predict the result.

QUESTION 5:

Here are the two patterns, written both in bit patterns and in hex. Perform the bitwise OR operation. (Do it with the bits, then re-write the answer in hex).

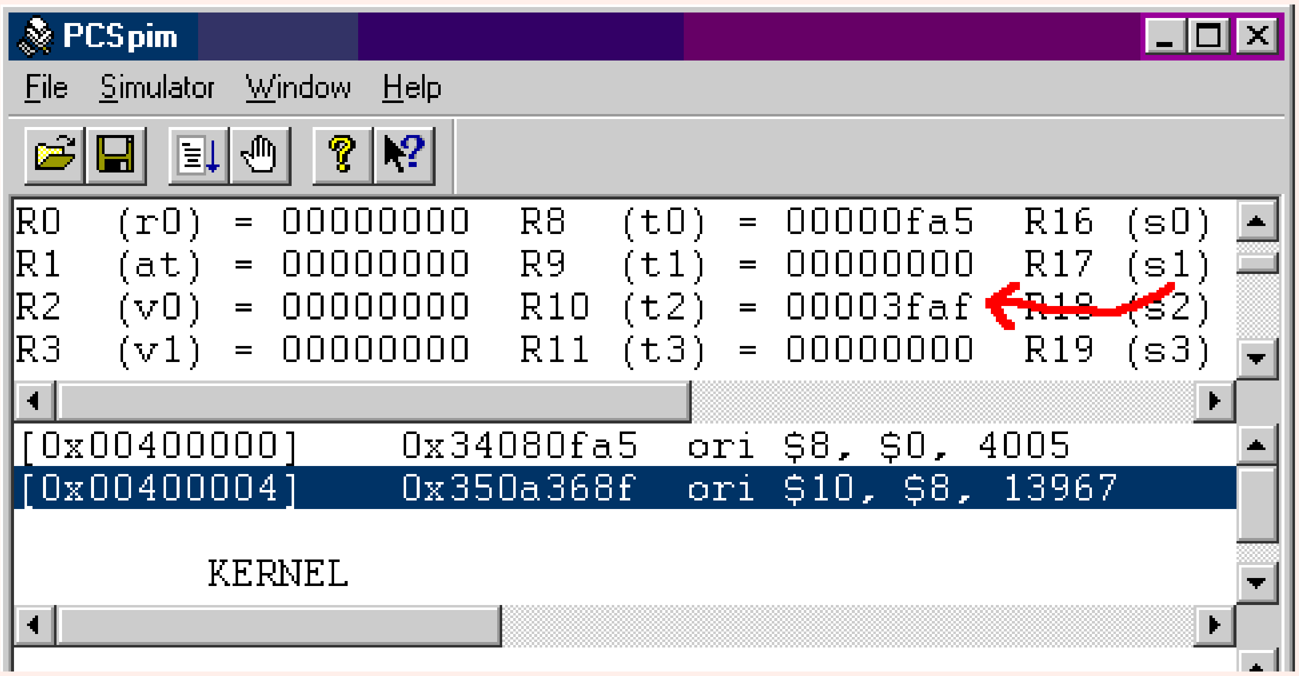
Next

```
0000 1111 1010 0101     0FA5

0011 0110 1000 1111     368F

---- ---- ---- ----     ----
```

A good answer might be:

```
0000 1111 1010 0101     0FA5

0011 0110 1000 1111     368F

---- ---- ---- ----     ----

0011 1111 1010 1111     3FAF
```

Running the Program



```
R0  (r0) = 00000000  R8  (t0) = 00000fa5  R16 (s0)
R1  (at) = 00000000  R9  (t1) = 00000000  R17 (s1)
R2  (v0) = 00000000  R10 (t2) = 00003faf  R18 (s2)
R3  (v1) = 00000000  R11 (t3) = 00000000  R19 (s3)

[0x00400000]     0x34080fa5  ori $8, $0, 4005
[0x00400004]     0x350a368f  ori $10, $8, 13967
```

To run the program do six things (look at chapter 9):

1. Create a source file.
2. Start SPIM.
3. Load the source file.
4. Set simulator switches (only the following matter at this time):
   - **ON** --- general registers in hexadecimal.
   - **ON** --- bare machine.
   - **OFF** --- allow pseudo instructions.
   - **OFF** --- load trap file.
5. Initialize the PC to the first instruction address.
6. Push F10 once per instruction.

The picture shows the result of running the program. The result in $10 is what was expected. The source code is at the right of each line in the code window (not seen in the cropped window of the picture). The version of the code in the middle column gives the bit patterns in decimal.

QUESTION 6:

Here is the complete line for the first instruction from the source window:

```
[0x00400000]    0x34080fa5    ori  $8, $0, 4005       ori  $8,$0,0x0FA5
```

Look at the 32-bit machine instruction. Do you see the immediate operand in it?

A good answer might be:

```
[0x00400000]    0x34080fa5      ori  $8, $0, 4005     ori  $8,$0,0x0FA5
```

Yes. How could you miss? The last 16 bits are the immediate operand.

ORI Machine Code

Below is the machine code for the instruction. In the third line the bits have been grouped according to their functional meaning. Documentation for the MIPS shows this grouping for each instruction. It is not something you can determine by inspection.

Look this over to get an idea of how it works. The right six bits of the instruction are the **opcode**, the bit pattern that specifies the machine operation. The next group of five bits specifies the operand register. The group of five after that specifies the destination register. The remaining bits are the immediate operand.

```
  3    4    0    8    0    f    a    5     -- machine instruction in hex

0011 0100 0000 1000 0000 1111 1010 0101    -- machine instruction in bits

001101  00000 01000 0000 1111 1010 0101    -- fields of the instructuion

opcode  oper  dest  immediate operand      -- meaning of the fields
        -and  reg.
        reg.

ori     $0     $8    0    f    a    5
```

QUESTION 7:

Here is the **`ori`** instruction again, but the patterns that specify the operand register, the destination register, and the immediate operand are blank. Fill in the blanks so the operand register is $10, the destination register is $9, and the immediate operand is 0xAFF0

Next

```
001101 _ _ _ _ _  _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

ori    $10        $9      a      f      f      0

     operand      dest        immediate operand
     reg.         reg.
```

A good answer might be:

```
        0 1 0 1 0  0 1 0 0 1  1 0 1 0  1 1 1 1  1 1 1 1  0 0 0 0
001101 _ _ _ _ _  _ _ _ _ _  _ _ _ _  _ _ _ _  _ _ _ _  _ _ _ _

ori    $10        $9        a        f        f        0

        operand    dest        immediate operand
        reg.       reg.
```

Machine Instructions Compared

Here again is the **`ori`** machine instruction:

```
  3   4   0   8   0   f   a   5    -- machine instruction in hex

001101  00000 01000 0000 1111 1010 0101    -- fields of the instructuion

opcode  oper  dest  immediate operand      -- meaning of the fields
        -and  reg.
        reg.

ori    $0    $8    0   f   a   5
```

The layout of this machine instruction is different from the **addu** instruction we looked at in chapter 10:

```
  0   1   0   9   5   0   2   1    -- machine instruction in hex

000000 01000 01001 01010 00000 100001      -- fields of the instructuion

opcode oprnd oprnd dest  ----- 2ndary      -- meaning of the fields

ALUop   $8    $9   $10          addu
```

Both instructions are 32 bits wide (as are all MIPS R2000/R3000 instructions). The first six bits is the opcode, which calls for an ALU operation. The **addu** instruction further specifiies the operation in the last six bits, the **secondary opcode.** The **addu** instruction does not have an immediate operand.

QUESTION 8:

Do the *fields* of two different machine instructions necessarily have the same size and same meaning?

Next

Do the *fields* of two different machine instructions necessarily have the same size and same meaning?

A good answer might be:

No.

Uses of Bitwise Logic

The first six bits of the instruction (the opcode) specify the machine operation. (Sometimes the secondary opcode is needed as well). The opcode also specifies the fields of the rest of the instruction and their meaning. A human needs to look at documentation to firgure this out. The MIPS chip is hardwired to do it.

Recall (from the second chapter) the fourth advantage of binary:

1. Simple; easy to build.
2. Unambiguous signals (hence noise immunity).
3. Flawless copies can be made.
4. Anything that can be represented with some sort of pattern can be represented with patterns of bits.

Most computers have built-in integer arithmetic operations and often built-in floating point operations. But computers are used for much more than numbers! Bit patterns are used to represent very many things. Bitwise logic is needed when computing with bit patterns in all their various uses.

For example, a document created with a word processor has font and formatting information embedded within it. The information is encoded with bit patterns in ways specific to each word processor family. A word processor program must use bitwise operations to process these codes.

QUESTION 9:

What type of program reads in strings of symbols encoded in ascii and outputs bit patterns that encode data and operations?

Next

A good answer might be:

An assembler.

Handy AND Immediate Instruction

An assembler uses bit manipulation to put together (to "assemble) the bit patterns of each machine instruction. This is a typical assignment in a systems programming course. Of course, you will likely write your assembler in C and will use its bitwise operators. But they ultimately become the processor's bitwise operations.

AND Operation on Bits				
first operand	0	0	1	1
second operand	0	1	0	1
	—	—	—	—
result	0	0	0	1

The **`andi`** instruction does a bitwise AND of two 32-bit patterns. At run time the 16-bit immediate operand is padded on the left with zero bits to make it a 32-bit operand.

```
andi d,s,const    # register d <-- bitwise AND of immediate operand const
                  #          and the contents of register $s.
                  #          const is a 16-bit pattern, so
                  #          0x0000 ... const ... 0xFFFF
```

The three operands of the instruction must appear in the correct order, and `const` must be within the specified range. If the immediate operand in a source instruction is less than sixteen bits (such as 0x2) the assembler expands it to sixteen. If it is more than sixteen bits the assebler writes an error message.

QUESTION 10:

Is the following instruction correct? What does it do?

```
andi $8,$0,0xFFFF
```

Next

Is the following instruction correct? What does it do?

```
andi $8,$0,0xFFFF
```

A good answer might be:

It is correct, but not very sensible. It ANDs the contents of the zero register (all zeros) with the immediate operand and puts the result in register $8. Of course, the result is all zeros, regardless of the immediate operand.

Exclusive Or Immediate

Filling a register with all zero bits is called **clearing** the register. Clearing a register is common, but the above instruction is not the best way to do it.

An **exclusive OR** is nearly the same as the more common OR (the *inclusive OR*) except that the result is zero when both operands are one.

XOR Operation on Bits				
first operand	0	0	1	1
second operand	0	1	0	1
	__	__	__	__
result	0	1	1	0

Here is a description of the assembly language instruction. The machine language for the instruction looks much the same as the **`ori`** and the **`andi`** instruction.

```
xori d,s,const     # register d <-- bitwise XOR of immediate operand const
                   #         and the contents of register $s.
                   #         const is a 16-bit pattern, so
                   #         0x0000 ... const ... 0xFFFF
```

The three operands of the instruction must appear in the correct order, and `const` must be within the specified range. If the immediate operand in a source instruction is less than sixteen bits (such as 0x2) the assembler expands it to sixteen. If it is more than sixteen bits the assebler writes an error message.

QUESTION 11:

Here are the two patterns, written both in bit patterns and in hex. Perform the bitwise XOR operation. (Do it with the bits, then re-write the answer in hex).

Next

```
0000 1111 1010 0101     0FA5

0011 0110 1000 1111     368F

---- ---- ---- ----     ----
```

A good answer might be:

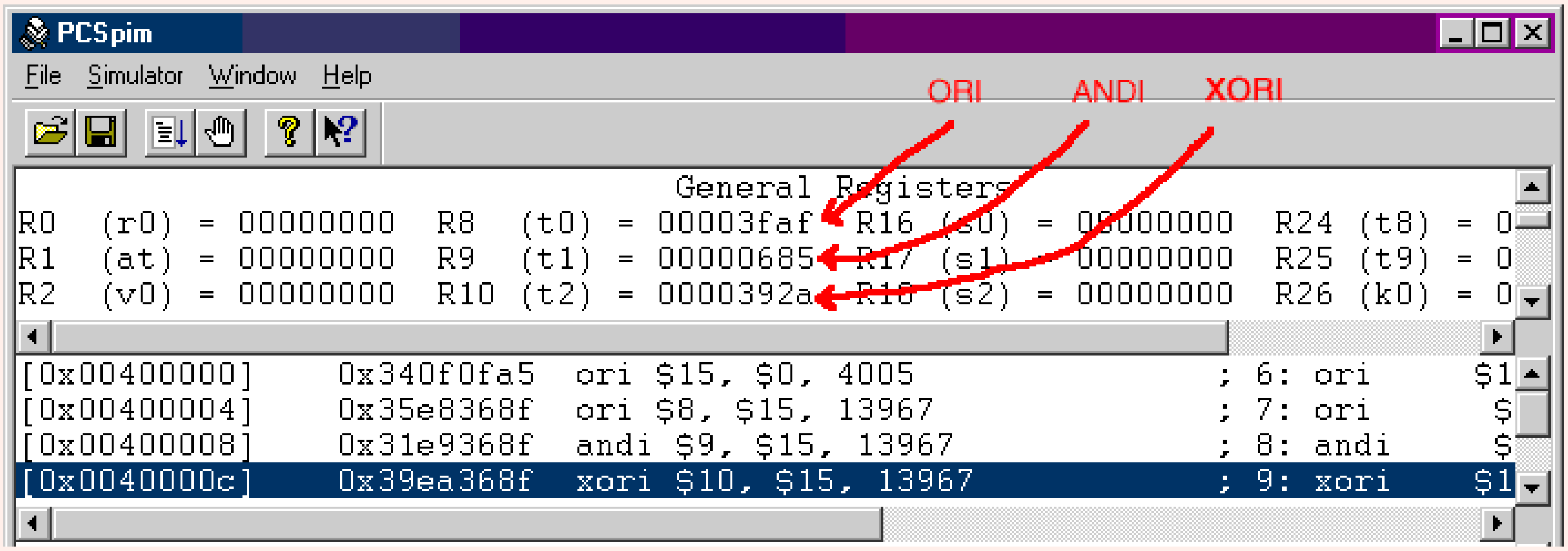
```
0000 1111 1010 0101     0FA5
0011 0110 1000 1111     368F
---- ---- ---- ----     ----
0011 1001 0010 1010     392A
```

Example Program

Here is a program that does all three bitwise operations between the same two patterns. The register is different in each case.

```
## Program to bitwise OR, AND, and XOR two patterns
        .text
        .globl  main

main:
        ori     $15, $0,0x0FA5       # put bit pattern register into $15
        ori      $8,$15,0x368F       # OR with second pattern
        andi      $9,$15,0x368F       # AND with second pattern
        xori    $10,$15,0x368F       # XOR with second pattern

## End of file
```

Running it in SPIM (pushing F10 four times) results in the following:



```
                              General Registers
R0  (r0) = 00000000  R8  (t0) = 00003faf  R16 (s0) = 00000000  R24 (t8) = 0
R1  (at) = 00000000  R9  (t1) = 00000685  R17 (s1) = 00000000  R25 (t9) = 0
R2  (v0) = 00000000  R10 (t2) = 0000392a  R18 (s2) = 00000000  R26 (k0) = 0
```

```
[0x00400000]    0x340f0fa5  ori $15, $0, 4005                ; 6: ori     $1
[0x00400004]    0x35e8368f  ori $8, $15, 13967               ; 7: ori     $
[0x00400008]    0x31e9368f  andi $9, $15, 13967              ; 8: andi    $
[0x0040000c]    0x39ea368f  xori $10, $15, 13967             ; 9: xori    $1
```

QUESTION 12:

What is the exclusive-OR of a bit pattern with itself? Try this with a few bit patterns written on a scrap of paper.

What is the exclusive-OR of a bit pattern with itself?

A good answer might be:

The result is all zero-bits.

End of the Chapter

Because the two operand are the same, the only two XOR operations involved are 0 XOR 0 = 0 and 1 XOR 1 = 0.

You have reached the end of this chapter. It's only logical that you wish to review. Click on a blue subject that interests you to immediately go to where it was discussed. To get back here, click on the "back arrow" button of your browser.

- Immediate Operands.
- Bitwise Logic.
- ORI instruction.
- Machine code or the ORI instruction
- ANDI instruction
- XORI instruction.

contents Go to Table of Contents

You have reached the end of the chapter.

created: 07/08/01

CHAPTER 12 — Shift Instructions and Logic Instructions

Chapter Topics:

- *Logical shift instructions (sll and srl).*
- *No-op as a shift instruction.*
- *Register-to-register bitwise logic instructions (or, and, xor, and nor).*
- *NOT as a NOR operation.*
- *MOVE as an OR operation.*
- *Example program: instruction assembly.*

QUESTION 1:

(Review:) If a bit pattern that represents an integer (say 0110 0001) is shifted left by one bit (say 1100 0010) what does the new bit pattern represent?

Next

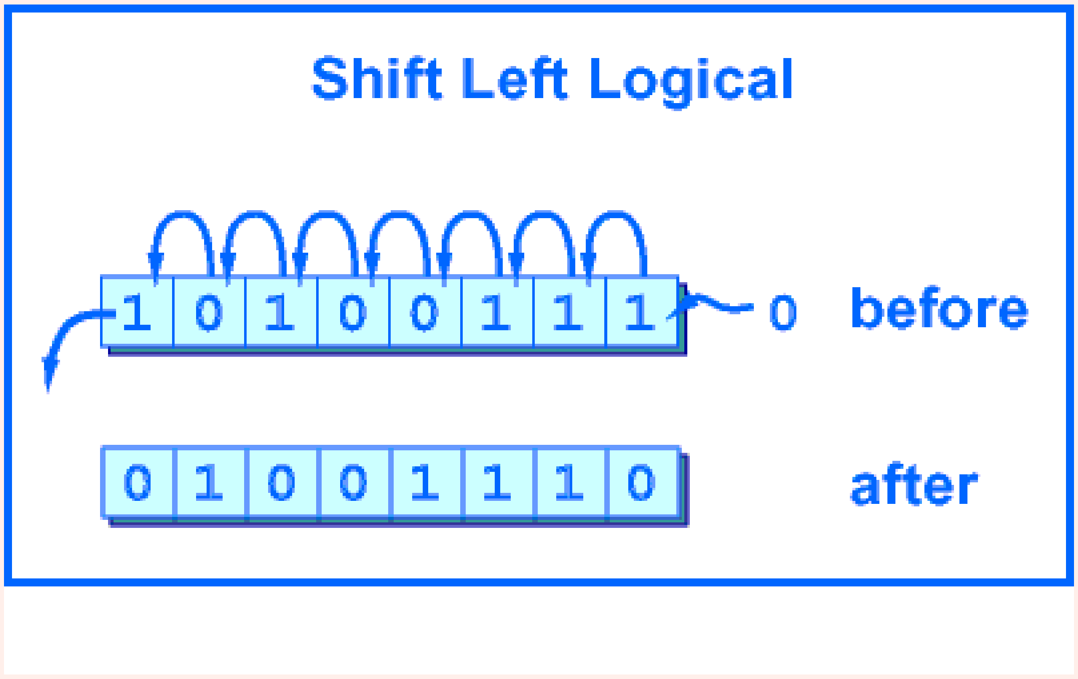
(Review:) If a bit pattern that represents an integer N (say 0011 0001) is shifted left by one bit (say 0110 0010) what does the new bit pattern represent?

A good answer might be:

2N, twice the original integer. 0011 0001 = 49_{10}. 0110 0010 = 98_{10}. (But you need to worry about what happens with the high-order bit if you shift too much).

Shift Left Logical

A **logical left shift** of one position moves each bit to the left by one. The low-order bit gets a zero (in all cases) and the high-order bit is discarded.



Shifting by two positions is the same as performing a one-position shift two times. Shifting by zero positions leaves the pattern unchanged. Shifting an N-bit pattern left logical by N or more makes all N bits zero.

MIPS performs the operation on a 32-bit register and puts the result in a 32-bit register.

```
sll  d,s,shft      # $d <-- logical left shift of $s by shft positions,
                   #         where  0 <= shft < 32
```

The ALU (arithmetic/logic unit) which does the operation pays no attention to what the bits mean. If the bits represent an unsigned integer, then a left shift is equivalent to multiplying the integer by two.

QUESTION 2:

Here is an 8-bit pattern. Shift it left (logical) by two. Write the hex for the new pattern

```
0x6F    0110 1111    <-- before

                  <-- after
```

A good answer might be:

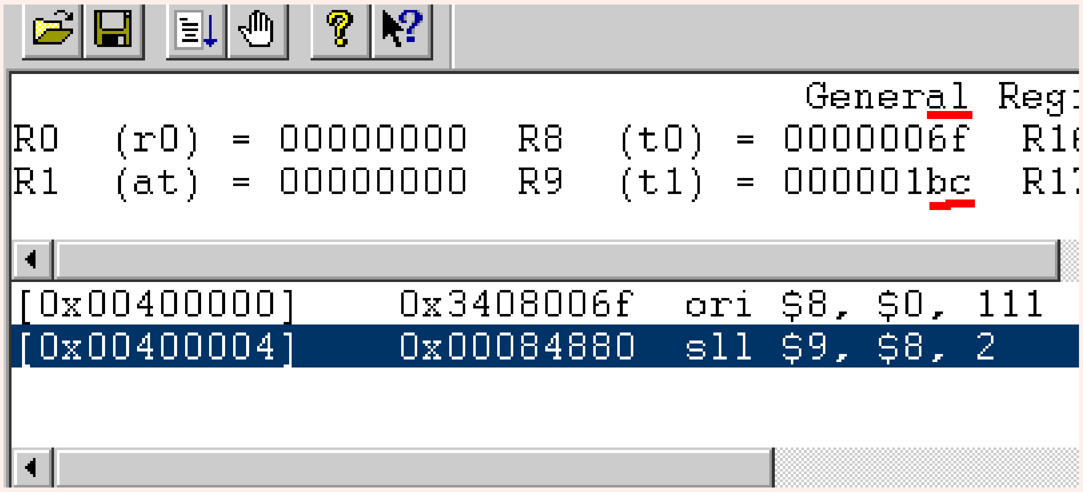
```
0x6F     0110 1111     <-- before

0xBC     1011 1100     <-- after
```

Shifty Program

This program does the shift you just performed by hand. It does it with 32-bit patterns, but for the right-most eight bits the result is the same.

```
## shiftTwo.asm
##
## Program to logical shift left a pattern
        .text
        .globl  main

main:
        ori     $8, $0, 0x6F        # put bit pattern register into $8
        sll     $9, $8, 2          # shift left logical by two

## End of file
```

Running the program does this:



```
                                      General  Regi
R0   (r0) = 00000000  R8   (t0) = 0000006f   R16
R1   (at) = 00000000  R9   (t1) = 000001bc   R17

[0x00400000]     0x3408006f   ori $8, $0, 111
[0x00400004]     0x00084880   sll $9, $8, 2
```

QUESTION 3:

Do you think it would be OK to shift the contents of register $8 and put the result back in register $8 :

Next

```
ori      $8, $0, 0x6F       # put bit pattern register into $8
sll      $8, $8, 2          # shift left logical by two
```

```
ori     $8, $0, 0x6F      # put bit pattern register into $8
sll     $8, $8, 2         # shift left logical by two
```

A good answer might be:

Yes, this is OK.

Shifting in Place

When an ALU operation is performed: (1) data is copied from the register(s) into the ALU. Then (2), the ALU does the operation. Next (3) the result is written to the designated result register. The result register can be an operand register without a problem because the operand data was transfered to the ALU in step (1).

Sending the result back to the source register is called **shifting in place**. This phrase is misleading because actually the shift is done in the ALU and the register gets a copy. Here is the machine instruction corresponding to the sll in the program:

```
  0    0    0    8    4    8    8    0    -- machine instruction in hex

0000 0000 0000 1000 0100 1000 1000 0000    -- machine instruction in bits

000000 00000  01000 01000  00010 000000    -- fields of the instructuion

opcode ----- source dest   shft  2ndary    -- meaning of the fields

ALUop        $8    $8      2     sll
```

The `sll` operation is signified by the 6-bit zero at the beginning of the instruction and the 6-bit zero at the end. The remaining fields give the source register ($8 in this case), the destination register ($8 in this case), and the amount of positions to shift (2).

QUESTION 4:

Study the above instruction. Write the instruction for *shift-left-logical the pattern in $0 by zero positions and leave the result in $0*:

```
        sll     $0, $0, 0           # fun machine instruction
```

Hint: on the midterm I expect you to write the 32-bit pattern for this instruction without hesitation! Memorize it now!

Not too hard to memorize zero.

A good answer might be:

```
   0    0    0    0    0    0    0    0    -- machine instruction in hex

0000 0000 0000 0000 0000 0000 0000 0000    -- machine instruction in bits

000000 00000  00000 00000  00000 000000    -- fields of the instructuion

opcode -----  $0    $0       0   2ndary    -- meaning of the fields

sll         source  dest  shft   sll
```

No-Op

Register $0 always contains a 32-bit zero (have you heard this before?) so shifting it left by zero positions and attempting to put the result back in $0 does nothing. Any instruction that attempts to alter $0 does nothing, but this instruction is the preferred way of doing nothing.

A machine instruction that does nothing is called (in official computer science jargon) a **no-op**. The *no operation* instruction is surprisingly useful, especially for MIPS.

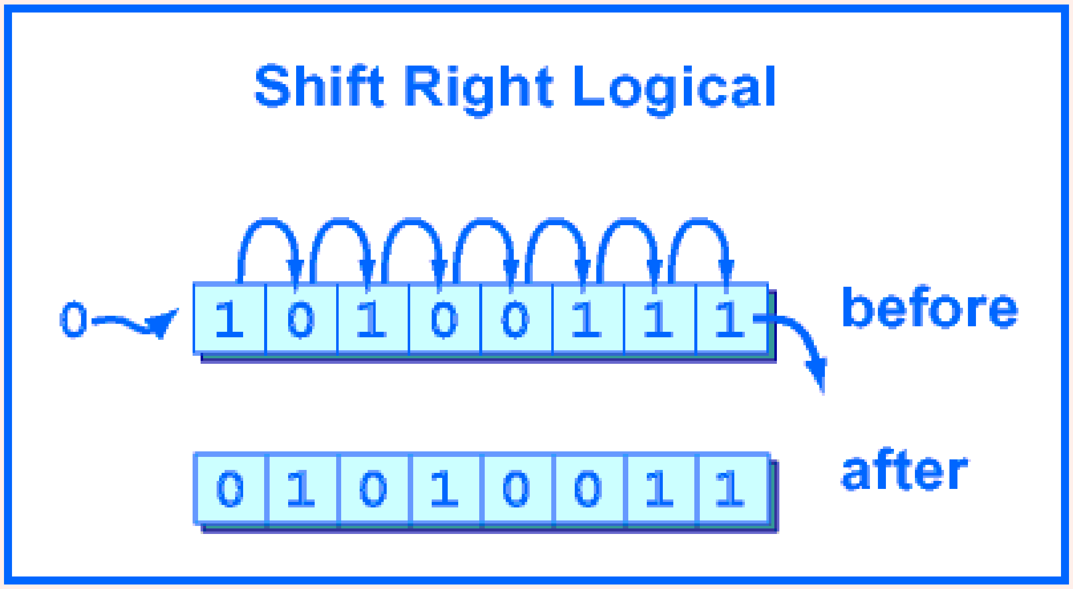
QUESTION 5:

What do you call a human who does nothing?

A good answer might be:

shiftless

Logical Shift Right



MIPS also has a **logical right shift** instruction. It moves bit to the right by a number of positions less than 32. The high-order bit gets zeros and the low-order bits are discarded.

If the bit pattern is regarded as an unsigned integer, or a positive two's comp. integer, then a right shift of one bit position performs an integer divide by two. A right shift by N positions performs an integer divide by 2^N.

The "trick" of dividing an integer by shifting should not be used in place of the MIPS arithmetic divide instruction (which will be covered in a few chapters). If you mean "divide" that is what you should write. But the trick is often used in hardware, and sometimes pops up in odd software uses, so you should know about it.

```
srl  d,s,shft      # $d <-- logical right shift of $s by shft positions.
                   #        shft is a 4-bit integer, 0 <= shft < 32
```

QUESTION 6:

(Trick Question:) What is the net result if a bit pattern is logical left shifted 2 positions and the logical right shifted 2 positions?

Next

(Trick Question:) What is the net result if a bit pattern is logical left shifted 2 positions then logical right shifted 2 positions?

A good answer might be:

The two bits on the right and left of the pattern are guaranteed to be zero; the middle bits are not changed.

OR Instruction

MIPS has many instructions that use two registers as operands and puts the result in a register. All the action takes place inside the processor: the data comes from registers, the ALU performs the operation, and the result is written to a register. All this can be done quickly.

The assembly language must specify four things: (1) the operation, (2) the first operand register, (3) the second operand register, and (4) the destination register. Of course, the machine language must encode the same four things in bit patterns.

Here is the register to register OR instruction:

```
or  d,s,t       # $d <-- bitwise OR between $s with $t.
```

Recall that the result of OR is zero only when both operand bits are zero.

QUESTION 7:

What is the bitwise OR of the following?

```
FEED
BECA
----
```

A good answer might be:

```
FEED      1111 1110 1110 1101
BECA      1011 1110 1100 1010
----      ---- ---- ---- ----
FEEF      1111 1110 1110 1111
```

It is usually easiest to translate into bits, do the OR, then translate back to hex.

AND Instruction

MIPS has a register-to-register AND instruction:

```
and  d,s,t        # $d <-- bitwise AND between $s with $t.
```

Recall that the result of AND is one only when both operand bits are one.

QUESTION 8:

What is the bitwise AND of the following?

```
FEED
BECA
----
```

A good answer might be:

```
FEED     1111 1110 1110 1101
BECA     1011 1110 1100 1010
----     ---- ---- ---- ----
BEC8     1011 1110 1100 1000
```

XOR Instruction

You will not be surprised to find that there is a register-to-register XOR instruction:

```
xor  d,s,t        # $d <-- bitwise XOR between $s with $t.
```

Recall that the result of XOR is one when only one operand bit is one.

QUESTION 9:

What is the bitwise XOR of the following?

```
FEED
BECA
----
```

A good answer might be:

```
FEED      1111 1110 1110 1101
BECA      1011 1110 1100 1010
----      ---- ---- ---- ----
4027      0100 0000 0010 0111
```

NOR Instruction

There is a register-to--register bitwise NOR instruction. There is no immediate operand NOR instruction. NOR is equivalent to performing the OR operation, then complementing the bits (change 0 to 1 and 1 to 0). Here is the assembly language for it:

```
nor  d,s,t    # $d <-- bitwise NOR
              # between $s with $t.
```

NOR Operation on Bits

first operand	0	0	1	1
second operand	0	1	0	1
	—	—	—	—
result	1	0	0	0

QUESTION 10:

What is the bitwise NOR of the following?

```
FEED
BECA
----
```

Next

A good answer might be:

```
FEED      1111 1110 1110 1101
BECA       1011 1110 1100 1010
----      ---- ---- ---- ----
0110      0000 0001 0001 0000
```

Summary

The table shows the register-to-register bitwise logic operations. Register **`d`** is the destination (where the result goes); the other two registers **`s`** and **`t`** contain the operands.

AND	OR	XOR	NOR
and d,s,t	or d,s,t	xor d,s,t	nor d,s,t
$d <— $s and $t	$d <— $s or $t	$d <— $s xor $t	$d <— $s nor $t

Here is a summary of the operations:

operand	0	0	1	1
operand	0	1	0	1
AND	0	0	0	1
OR	0	1	1	1
XOR	0	1	1	0
NOR	1	0	0	0

QUESTION 11:

Fill in the following:

Next

0 NOR 0 = ____	NOT 0 = ____
1 NOR 0 = ____	NOT 1 = ____

Now answer the question: NOT X = X ____ ____

A good answer might be:

0 NOR 0	=	1	NOT 0	=	1
1 NOR 0	=	0	NOT 1	=	0

Now: NOT X = X NOR 0

NOT as NOR with $0

The NOT operation is done by using the NOR instruction with $0 as one of the operands:

```
nor  d,s,$0       # $d <-- bitwise NOT of $s.
```

NOT Operation on Bits

first operand	0	1
	—	—
result	1	0

QUESTION 12:

Say that register $8 contains a bit pattern. OR is performed between it and register $0. Register $9 is to be the destination. What is the effect?

Next

```
or  d,s,$0
```

A good answer might be:

The bit pattern in $s is moved to $d ($s is unchanged).

```
or  d,s,$0      # $d <-- contents of $s.
```

MOVE as OR with Zero

Copying the pattern in a source register to a destination register is called a **move** operation, even though the source register does not change.

```
or  d,s,$0      # $d <-- contents of $s.
```

QUESTION 13:

(Review:) How can a particular bit pattern be loaded into a register?

Next

How can a particular bit pattern be loaded into a register?

A good answer might be:

With an **`ori`** instruction.

Example Program

Here are the fields of the **`ori`** instruction. The numbers in the first row are bit positions, numbered from the low-order bit on the right. The opcode of the instruction is 0xD. The source register is **`s`**, the destination register is **`d`**, and the immediate operand is **`const`**.

31....26	25...21	20...16	15..................0
0xD	s	d	unsigned const

Our example program will assemble the machine code that corresponds to the assembly language instruction:

```
ori  $8,$9,0x004A
```

The instruction will be assembled in register $25. The program uses bitwise and shift operations. Let us say that this example is part of a larger program. It is wise to initialize $25 to contain all zeros (this is called *clearing* the register).

QUESTION 14:

What assembly language instruction clears register $25 ?

Next

What assembly language instruction clears register $25 ?

A good answer might be:

```
or  $25,$0,$0       # $d <-- $0
```

(There are others that work, but this is the clearest).

Target Instruction

The program starts with:

```
or  $25,$0,$0       # clear $25
```

Now the correct bit patterns are placed into the fields of the instruction. The instruction we are assembling is:

```
ori  $8,$9,0x004A
```

QUESTION 15:

Fill in the third row with the bit patterns. Use hexadecimal (assume that the bit patterns are truncated on the left to make them the right size).

Next

ori $8,$9,0x004A			
31....26	25...21	20...16	15..................0
0xD	s	d	unsigned const

A good answer might be:

ori $8,$9,0x004A			
31....26	25...21	20...16	15..................0
0xD	s	d	unsigned const
0xD	0x9	0x8	0x004A

Individual Field Values

There are four fields in the instruction. We know the bit pattern that goes into each field. Let us put those patterns in registers $11, $12, $13, and $14. The program (so far) looks like:

```
or    $25,$0,$0        # clear $25
ori   $11,$0,________  # opcode
ori   $12,$0,________  # operand $s
ori   $13,$0,________  # dest. $d
ori   $14,$0,________  # immediate operand
```

QUESTION 16:

Fill in the field values

Next

A good answer might be:

See below

Shifting the Opcode

The **`ori`** puts its immediate operand in the low order bits (the right-most bits) of its destination register. So now, with the following code, the registers have the correct patterns but the patterns are not in the correct fields.

```
or    $25,$0,$0        # clear $25
ori   $11,$0,0xD       # opcode
ori   $12,$0,0x9       # operand $s
ori   $13,$0,0x8       # dest. $d
ori   $14,$0,0x004A    # immediate operand
```

Register $11 contains the opcode in its right-most bits, like this:

31....26	25...21	20...16	15..................0
000000	00000	00000	0000 0000 0000 1101

However, the opcode is required to be in the high-order six bits, 26 through 31. A **`sll`** will shift the contents of $11 into the correct position.

QUESTION 17:

Fill in the amount by which to shift. (Hint: determine which bit should be the new left-most bit. Count the number of bits between its old and new position).

Next

```
sll   $11,$11,_____    # left shift $11 by ______ places
```

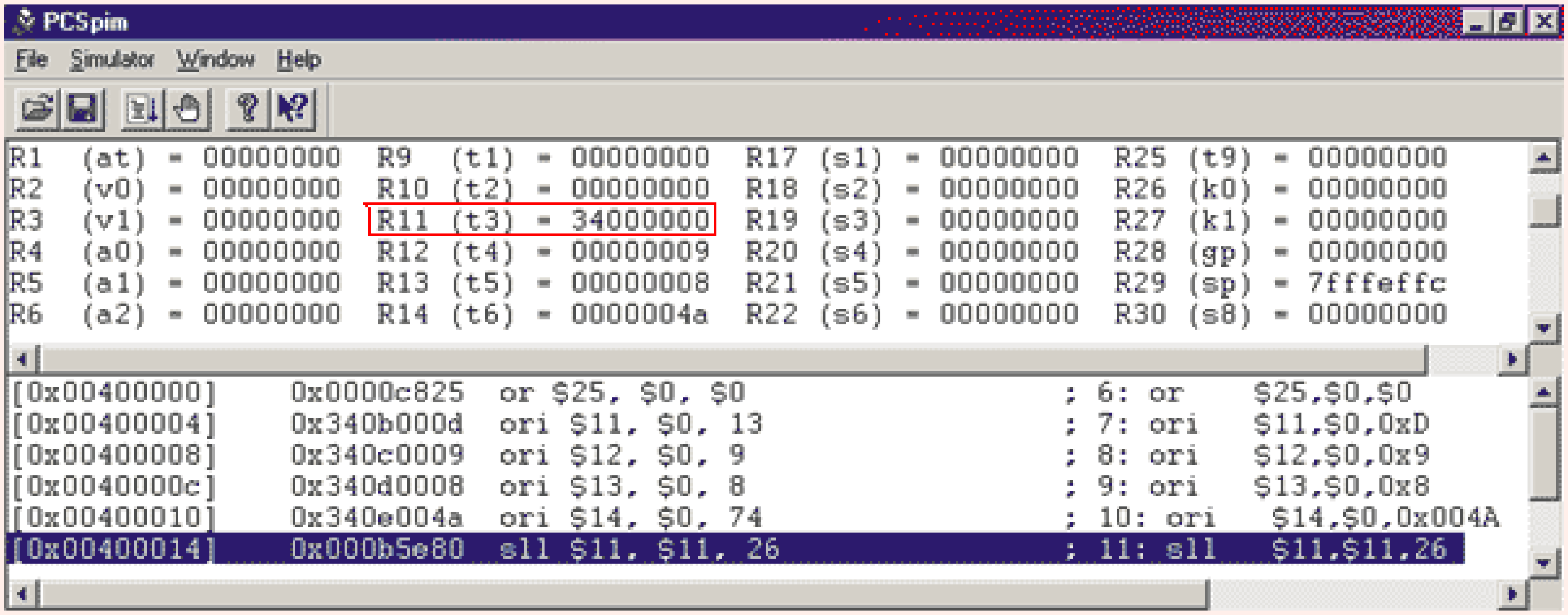
A good answer might be:

```
sll   $11,$11,26    # left shift $11 by 26 places
```

Program So Far

Arithmetic gives us the same answer. Bit position 5 contains the right-most bit of the six-bit opcode. It must be shifted into position 31, and 31-5 = 26.

Here is a run of the program so far. The source code is seen at the right in the window.



```
R1  (at) = 00000000  R9  (t1) = 00000000  R17 (s1) = 00000000  R25 (t9) = 00000000
R2  (v0) = 00000000  R10 (t2) = 00000000  R18 (s2) = 00000000  R26 (k0) = 00000000
R3  (v1) = 00000000  R11 (t3) = 34000000  R19 (s3) = 00000000  R27 (k1) = 00000000
R4  (a0) = 00000000  R12 (t4) = 00000009  R20 (s4) = 00000000  R28 (gp) = 00000000
R5  (a1) = 00000000  R13 (t5) = 00000008  R21 (s5) = 00000000  R29 (sp) = 7fffeffc
R6  (a2) = 00000000  R14 (t6) = 0000004a  R22 (s6) = 00000000  R30 (s8) = 00000000

[0x00400000]    0x0000c825  or $25, $0, $0        ; 6: or    $25,$0,$0
[0x00400004]    0x340b000d  ori $11, $0, 13       ; 7: ori   $11,$0,0xD
[0x00400008]    0x340c0009  ori $12, $0, 9        ; 8: ori   $12,$0,0x9
[0x0040000c]    0x340d0008  ori $13, $0, 8        ; 9: ori   $13,$0,0x8
[0x00400010]    0x340e004a  ori $14, $0, 74       ; 10: ori   $14,$0,0x004A
[0x00400014]    0x000b5e80  sll $11, $11, 26      ; 11: sll   $11,$11,26
```

Great! Now we have the correct bit pattern placed in the correct 6-bit field. Only one problem: it's in the wrong register. The instruction is supposed to be in register $25.

QUESTION 18:

Think of an assembly instruction that will put the opcode into register $25. Here is a start:

```
or   $25,$___,$___    # move opcode into target register
```

Next

A good answer might be:

```
or   $25,$25,$11    # move opcode into target register
```

Alternative:

```
or   $25,$11,$0     # move opcode into target register
```

Source Register

Now move the bit pattern for the source register $9 into the correct field. Register $12 contains the pattern in its right-most bits, like in the left table. The pattern should be in bits 25...21 of register $25 as at right.

31....26	25...21	20...16	15..................0
000000	00000	00000	0000 0000 0000 1001

`ori      $8,$9,0x004A`			
31....26	25...21	20...16	15..................0
0xD	s	d	unsigned const
0xD	0x9	0x8	0x004A

The same method can be used for this as for the opcode. There is a slight problem: the opcode already in $25 must not be erased.

QUESTION 19:

Fill in the next part of the program

```
sll   $12,$12,_____      # left shift s by _____ places
or    $25,______,_____   # move s into target register
```

Next

A good answer might be:

```
sll   $12,$12,21      # left shift s by 21 places
or    $25,$25,$12     # move s into target register
```

ORing in a Field

The low-order bit of the pattern for $s is in bit position 0 (of register $12). It must be moved to position 21 of the instruction. Next, the **`or`** copies it into the instruction being assembled.

The **`or`** works like this:

	31....26	25...21	20...16	15............................0
Register $12 (before)	000000	01001	00000	0000 0000 0000 0000
Register $25 (before)	001101	00000	00000	0000 0000 0000 0000
Register $25 (after)	001101	01001	00000	0000 0000 0000 0000

QUESTION 20:

Would this instruction have worked as well?

```
or    $25,$0,$12     # move s into target register
```

Next

Would this instruction have worked as well?

```
        or    $25,$0,$12     # move s into target register
```

A good answer might be:

No---the 32-bit result ($0 OR $12) is moved into $25, destroying the opcode already there.

```
$25 <-- ($0  OR $12)     or  $25,$0,$12     # wrong
$25 <-- ($25 OR $12)     or  $25,$25,$12    # right
```

Destination Register and Const

ori $8,$9,0x004A			
31....26	25...21	20...16	15...................0
opcode	s	d	unsigned const
0xD	0x9	0x8	0x004A

The same method is used with the bit pattern that designates the destination register. The **`const`** part is in register #13, and it is in the correct field. It needs to be copied into $25 The complete program follows. The comments on each line have been improved. Comments should describe the work done in solving the problem. They should **not** describe the instruction. Of course, when these notes explain an instruction the comment will often do the same. But that is a poor style, in general.

The following can be copied into the clipboard and pasted into your program editor, then saved to a file and run with SPIM.

```
## Program to assemble the instruction ori     $8,$9,0x004A
##
        .text
        .globl  main

main:
or    $25,$0,$0        # clear $25
ori   $11,$0,0xD       # opcode
ori   $12,$0,0x9       # operand $s
ori   $13,$0,0x8       # dest. $d
ori   $14,$0,0x004A    # immediate operand

sll   $11,$11,26       # shift opcode into position
or    $25,$25,$11      # or it into the instruction
```

```
sll   $12,$12,21       # shift operand $s into position
or    $25,$25,$12      # or it into the instruction

sll   $13,$13,16       # shift dest $d into position
or    $25,$25,$13      # or it into the instruction

or    $25,$25,$14      # or const into the instruction

## end of file
```

QUESTION 21:

Would it be possible to *dis-assemble* the instruction in $25?

Next

Would it be possible to *dis-assemble* the instruction in $25?

A good answer might be:

Sure. Bit logic operations can take appart the various fields and put them in different registers.

A Run of the Program

Disassembly sounds like a really great programming assignment. But now let's run the assembler program:

The machine instruction in register $25 is correct, as expected. It would be nice to save the instruction to memory at this point, but you don't know how to do this yet.

QUESTION 22:

Could this program be written using fewer registers?

Could this program be written using fewer registers?

A good answer might be:

Yes. You only need two. Sounds like another programming exercise.

End of Chapter

You have reached the end of this chapter. You may wish to shift into review mode. Click on a blue subject that interests you to go to where it was discussed. To get back here, click on the "back arrow" button of your browser.

- Logical shift left.
- No-operation instruction as a logical left shift.
- Logical shift right.
- or register-to-register instruction
- and register-to-register instruction
- xor register-to-register instruction
- nor register-to-registerinstruction
- not operation
- move operation

Go to Table of Contents

You have reached the end of the chapter.

created: 07/08/01

CHAPTER 13 — Add and Subtract Instructions

This chapter discusses the MIPS instructions for performing 32-bit integer addition and subtraction. Some topics of integer representation with bit patterns are reviewed.

Chapter Topics:

- *Overflow in unsigned binary and two's complement (review).*
- *The add and addu instructions.*
- *Sign extention*
- *The addi and addiu instructions.*
- *The sub and subu instructions.*
- *Using addi to load a register with a negative integer.*

QUESTION 1:

Say that a processor has a full set of bit manipulation instructions. Can it do arithmetic with these instructions?

Next

A good answer might be:

Yes. (And it can do floating point arithmetic as well).

Arithmetic as Bit Manipulation

Integers are represented with bit patterns, so integer operations are bit manipulation operations. Some very small, very fast processors provide no data manipulation instructions other than bit pattern manipulation. Adding two integers is done by implementing the Binary Addition Algorithm (see chapter 8) with these bit instructions.

Luckily, MIPS has instructions that perform integer arithmetic. The normal size of an integer is 32 bits (the same as the size of a register). Longer or shorter integer artihmetic is done using bit manipulation instructions in combination with 32-bit arithmetic instructions.

QUESTION 2:

The MIPS `addu` instruction performs the Binary Addition Algorithm on two 32-bit patterns. What integer representation method can be used with it?

Next

- Unsigned Binary?
- Two's Complement?
- Both?

A good answer might be:

- Both

Binary Addition Algorithm

The Binary Addition Algorithm works for both methods of integer representation. The same MIPS instruction (addu) is used for both. However, the overflow condition is different for each representation.

Binary Addition Algorithm: detecting overflow	
Unsigned Binary	**Two's Complement**
The result is correct if *the carry out* of the high order column is zero.	The result is correct if *the carry into* the high order column is the same as *the carry out of* the high order column. The carry bits can both be zero or both be one.

QUESTION 3:

Use the Binary Addition Algorithm on these 8-bit patterns:

Next

```
1010 1011
0101 0101
---------
```

Does overflow happen for:

- Unsigned Binary?
- Two's Complement?

A good answer might be:

```
 11111 111
  1010 1011
  0101 0101
  ---------
  0000 0000
```

Does overflow happen for:

- Unsigned Binary? Overflow -- The carry out is one.
- Two's Complement? In Range -- The carry in is the same as the carry out.

The addu Instruction

The addu instruction performs the Binary Addition Algorithm on the contents of two 32-bit registers and places the result in the destination register. The destination register can be the same as one of the source registers. The addu instruction mechanically grinds through the Binary Addition Algorithm, producting a 32-bit result from two 32-bit operands. Overflow is ignored (that is what the "u" at then end of the mnemonic means).

```
addu  d,s,t        # $d <-- s + t.  no overflow trap
```

There is another instruction, add, which causes a *trap* when two's complement overflow is detected. Other than that, it is the same as addu. A **trap** is an interruption in the normal machine cycle. Typically on a computer system a trap results in sending control back to the operating system.

```
add   d,s,t        # $d <-- s + t.  with overflow trap
```

Most assembly programmers deal with overflow by making sure that the operands won't cause it. Usually the addu instruction is used. Until you know how to handle a trap that is the approach we will take.

QUESTION 4:

What is the range of integers that can be rep^{d} with 32-bit two's complement? -2^{----} to +2^{----} - 1. (Pick an exponent for each "2").

Next

A good answer might be:

What is the range of integers that can be rep^d with 32-bit two's complement? -2^{31} to $+2^{31} - 1$.

(There are 2^{32} bit patterns. Half of them are for negative integers, and the remaining are for the positive integers and zero).

Example Program

Here is the previous addition problem extended to 32 bits.

```
carry  --> 0000 0000 0000 0000 0000 0001 1111 111
           0000 0000 0000 0000 0000 0000 1010 1011       000000AB
           0000 0000 0000 0000 0000 0000 0101 0101       00000055
           ---------------------------------------       --------
           0000 0000 0000 0000 0000 0001 0000 0000       00000100
```

What was unsigned overflow with 8-bit unsigned arithmetic is within range for 32-bit arithmetic (both unsigned and two's comp.)

```
## AddSome.asm
##
## Program to demonstrate addition
        .text
        .globl  main

main:
        ori      $8, $0, 0xAB       # put 0x000000AB into $8
        ori      $9, $0, 0x55       # put 0x00000055 into $9
        addu     $10,$9, $8         # $10 <-- sum

## End of file
```

QUESTION 5:

Are registers $9 and $8 changed by the addu instruction in this program?

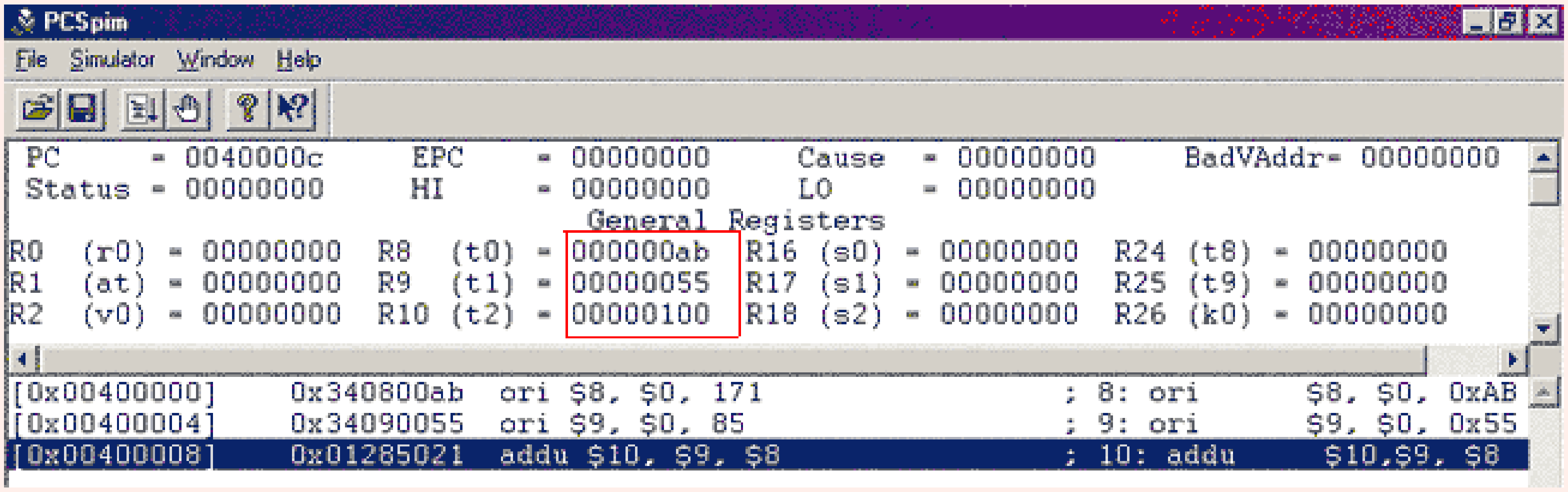
Next

A good answer might be:

No. Operand registers are not changed, unless one is also the destination register.

Run of the Program

Here is a run of the program. The results are as expected. Notice that the decimal interpretation of the bit patterns is listed in one version of the source code.



```
 PC     = 0040000c     EPC    = 00000000     Cause  = 00000000     BadVAddr= 00000000
 Status = 00000000     HI     = 00000000     LO     = 00000000
                                 General Registers
R0  (r0) = 00000000  R8  (t0) = 000000ab  R16 (s0) = 00000000  R24 (t8) = 00000000
R1  (at) = 00000000  R9  (t1) = 00000055  R17 (s1) = 00000000  R25 (t9) = 00000000
R2  (v0) = 00000000  R10 (t2) = 00000100  R18 (s2) = 00000000  R26 (k0) = 00000000

[0x00400000]   0x340800ab  ori $8, $0, 171          ; 8: ori     $8, $0, 0xAB
[0x00400004]   0x34090055  ori $9, $0, 85           ; 9: ori     $9, $0, 0x55
[0x00400008]   0x01265021  addu $10, $9, $8         ; 10: addu    $10,$9, $8
```

To express integers in decimal notation omit the leading "0x".

```
ori     $8, $0, 171       # put 171 into $8
ori     $9, $0,  85       # put  85 into $9
addu    $10,$9,  $8       # $10 <-- sum
```

Of course, the same bit patterns are loaded into the registers. The assembler accepts both notations for integers.

QUESTION 6:

(Review:) Can ori $d,$0,const be used with a negative integer for const?

Next

Can ori $d,$0,const be used with a negative integer for const?

A good answer might be:

No. const is a 16-bit immediate operand that is zero-extended to a 32-bit integer when it is copied to $d. So it can't be negative.

Negating a Two's Comp. Integer

There will be much better ways to do this coming up, but for now say that the problem is to add +146 with -82. Loading a register with +146 is easy. But how can we load the -82?

Recall that a two's comp. integer is made negative by reflecting the bits then adding one.

QUESTION 7:

Say that register $8 has been loaded with +82: ori $8,$0,82 What instructions (that have been discussed so far in these notes) can do the following:

- Reflect the bits in $8
- Add one to $8

A good answer might be:

- Reflect the bits in $8:
  nor $8,$8,$0
- Add one to $8:
  ori $9,$0,1
  addu $8,$8,$9

Example Program

Here is a program that does that. There are much better ways to load a register with a negative integer. However, this is a nice example of bit manipulation.

```
## handMadeNeg.asm
##
## Program to demonstrate two's complement negative
##
        .text
        .globl  main

main:
        ori     $7, $0, 146       # put +146 into $7
        ori     $8, $0,  82       # put 82 into $8
        nor     $8, $8,  $0       # reflect
        ori     $9, $0,   1       #
        addu    $8, $8,  $9       # add 1: $8 = -82
        addu    $10, $7,  $8      # (+146) + (-82)

## End of file
```

QUESTION 8:

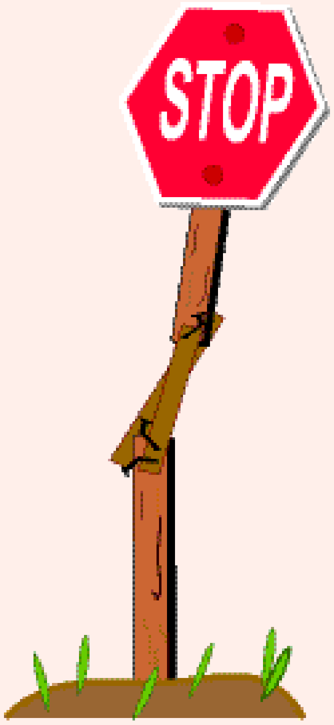
146-82 = ____ ?   in hex = _____ ?

Next

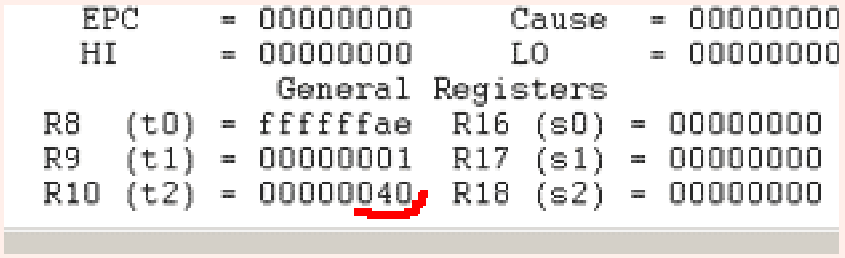
A good answer might be:

146-82 = 64   in hex = 0x40

Sign Extension



A run of the program produces the expected result.



```
  EPC       = 00000000      Cause     = 00000000
  HI        = 00000000      LO        = 00000000
                General  Registers
R8  (t0) = ffffffae  R16 (s0) = 00000000
R9  (t1) = 00000001  R17 (s1) = 00000000
R10 (t2) = 00000040  R18 (s2) = 00000000
```

So that one could be added to register $8, the one was first loaded into another register. It would be nice if there were an "add one" instruction. Many processors have such an instruction. MIPS has an "add immediate" instruction, addiu d,s,const.

The immediate operand of this instruction is 16 bits (as are all MIPS immediate operands). However, when extended to a 32-bit operand by the ALU it is **sign extended**: The value of the left-most bit of the immediate operand (bit 15) is copied to all bits to the left (into the high-order bits). So if the 16-bit immediate operand is a 16-bit two's complement negative integer, the 32-bit ALU operand is a 32-bit version of the same negative integer. The left-most bit of a two's comp. integer is sometimes called the "sign bit".

QUESTION 9:

Here is a 16-bit two's complement negative one:

FF FF = 1111 1111 1111 1111

Sign-extend it to 32 bits:

Next

__ __ __ __ = ???? ???? ???? ???? ???? ???? ???? ????

A good answer might be:

A 16-bit two's complement negative one:

```
FF FF  = 1111 1111 1111 1111
```

Sign-extended:

```
FF FF FF FF  =  1111 1111 1111 1111 1111 1111 1111 1111
```

The sign-extended version is a 32-bit negative one.

The Fond addiu Instruction

The MIPS addiu instruction includes a 16-bit immediate operand. When the ALU uses it, the immediate operand is sign-extended to 32 bits. If two's comp. overflow occurs it is ignored.

```
addiu   d,s,const        # $d <-- s + const.
                         # Const is 16-bit two's comp. sign-extended to 32 bits
                         # when the addition is done. No overflow trap.
```

There is also an add immediate instruction that does trap overflow. We won't use it:

```
addi    d,s,const        # $d <-- s + const.
                         # Const is 16-bit two's comp. sign-extended to 32 bits
                         # when the addition is done. Overflow trap.
```

QUESTION 10:

Here is the previous program, that added +146 with -82. Rewrite it using the addiu instruction. Put the result in $10.

Next

```
ori      $7, $0, 146       # put +146 into $7
ori      $8, $0,  82       # put 82 into $8
nor      $8, $8,  $0       # reflect
ori      $9, $9,   1       #
```

```
        addu    $8, $8,  $9        # add 1: $8 = -82
        addu   $10, $7,  $8        # (+146) + (-82)
```

A good answer might be:

```
ori      $7, $0, 146        # put +146 into $7
addiu    $10,$7,-82         # add -82
```

The program is much shorter.

The subu Instruction

MIPS has two integer subtraction instructions:

```
subu   d,s,t        # $d <-- s - t . No overflow trap.
                    # This is equivalent to $d <-- s + (-t)
                    # where (-t) is reflect-add-one of t.

sub    d,s,t        # $d <-- s - t . Trap overflow!
                    # This is equivalent to $d <-- s + (-t)
                    # where (-t) is reflect-add-one of t.
```

QUESTION 11:

When ori $d,$0,const is used to load $d, const is 16-bit unsigned binary. Say that you want to load $8 with a negative 86. Will the following work?

Next

```
addiu    $8,$0,-86  # $8 <-- -86
```

```
        addiu    $8,$0,-86
```

A good answer might be:

Yes. The immediate operand -86 is *sign*-extended to 32 bits then added to a 32-bit zero. The sum (-86) is loaded into $8.

The Absent Subtract Immediate

You would expect that since there are add, addu, addi, addiu and since there are sub, subu that there would be subtract immediate instructions. But there are not. The add immediate instruction is used. To subtract 201 from register $10 using an immediate operand, do this:

```
        addiu    $8,$10,-201    #  $8 <-- $10 - 201
```

Say that we want to compute $5 \times x - 74$ where the value x is in register $8. MIPS has an integer multiply instruction, but let us say that we don't want to use it. How can $5 \times x$ be done using the instructions you have seen so far?

QUESTION 12:

How could you compute 4 × $8 + $8 ?

Next

How could you compute 4× $8 + $8 ?

A good answer might be:

Multiply $8 by four by shifting left two positions, then add it the original $8

Example Program

Here is the program. Unfortunately, there are a few blanks. This would be a good time to use that scratch pad and pencil next to your keyboard.

```
## slowMult.asm
##
## Program to calculate 5 × x - 74
##
## Register Use:
##  $8   x
##  $9   result

        .text
        .globl  main

main:
        ori      $8, $0, 12          # put x into $8
        sll      $___, $___,  ___     # $___ <-- 4x
        addu     $___, $___, $___     # $___ = 5x
        addiu    $___, $___,-74       # $___ = 5x - 74

## End of file
```

QUESTION 13:

Fill in the blanks to finish the program. The final result should be in register $9.

Next

A good answer might be:

The complete program is below.

Filled Blanks

```
## slowMult.asm
##
## Program to calculate 5×x - 74
##
## Register Use:
##  $8   x
##  $9   result

        .text
        .globl  main

main:
        ori      $8, $0, 12        # put x into $8
        sll      $9, $8,  2        # $9 <-- 4x
        addu     $9, $9, $8        # $9 = 5x
        addiu    $9, $9,-74        # $9 = 5x - 74

## End of file
```

QUESTION 14:

Could the program be written to use just one register?

Next

A good answer might be:

No, because the original value x is used several times and needs to be in a register.

End of Chapter

(The program could be written with on register if you could use main memory to store x. But these notes have not told you how, yet).

You have reached the end of this chapter. Click on a blue subject that interests you to go to where it was discussed. To get back here, click on the "back arrow" button of your browser.

- Overflow in unsigned and two's comp.
- The addu and add instructions.
- Decimal constants in assembly language.
- Sign extension of a two's comp. integer.
- The addiu and addi instructions.
- The subu and sub instructions

Go to Table of Contents

You have reached the end of the chapter.

created: 07/12/01

CHAPTER 14 --- Integer Multiplication Instructions

This chapter discusses the MIPS instructions for performing 32-bit integer multiplication. Some topics of integer representation with bit patterns are reviewed.

Chapter Topics:

- *Integer multiplication and division.*
- *The **hi** and **lo** registers.*
- *The mult* and multu instructions.
- *The div* and divu instructions.
- *The mfhi* and mflo instructions.
- *Arithmetic shift right.*
- *The sra Instruction.*

QUESTION 1:

Multiply 99_{10} times 99_{10}: __________. How many decimal places does each operand (99) take: _________. How many decimal places does the result take: _________.

Next

A good answer might be:

Multiply 99_{10} times 99_{10}: 9801. How many decimal places does each operand (99) take: 2. How many decimal places does the result take: 2.

Twice the Number of Places

The product of two N-place decimal integers may need 2N places. This is also true for numbers expressed in any base. In particular, the product of two integers expressed with N-bit binary may need 2N bits for the product. Here two 8-bit unsigned integers are multiplied (using the usual paper-and-pencil multiplication algorithm, but with binary arithmetic):

$$
\begin{array}{rcc}
\text{10110111} & \text{B7} & 183_{10} \\
\text{10100010} & \text{A2} & 162_{10} \\
\text{--------} & \text{--} & \text{---} \\
\text{00000000} & & \\
\text{10110111.} & & \\
\text{00000000..} & & \\
\text{00000000...} & & \\
\text{00000000....} & & \\
\text{10110111.....} & & \\
\text{00000000......} & & \\
\text{10110111.......} & & \\
\text{---------------} & \text{----} & \text{-----} \\
\text{111001111001110} & \text{73CE} & 29646_{10}
\end{array}
$$

The two 8-bit operands result in a 15-bit product.

QUESTION 2:

Is a 32-bit general register likely to hold the result of multiplying two other 32-bit registers?

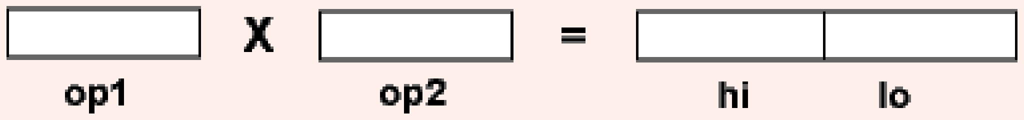
Next

A good answer might be:

No. In general, 64 bits are needed.

MIPS Multiply Unit

The multiply unit of MIPS contains two 32-bit registers: **hi** and **lo**. These are not general purpose registers. When two 32-bit operands are multiplied, **hi** and **lo** hold the 64 bits of the result. Bits 32 through 63 are in **hi** and bits 0 through 31 are in **lo**.



Here are the instructions that do this. The operands are contained in general-purpose registers.

```
mult    s,t      # hilo <-- $s * $t.  two's comp operands

multu   s,t      # hilo <-- $s * $t.  unsigned operands
```

Integer multiplication is slightly different for signed and unsigned operands, so there are two instructions. Integer multiply will never cause a trap.

QUESTION 3:

Two small integers are multiplied. Where is the result?

Next

A good answer might be:

If the result is small enough it will be contained in lo, and hi will contain all zeros.

The mfhi and mflo Instructions

We will mostly write programs that keep the result under 32 bits in length. There are two instructions that move the result of a multiply into a general purpose register:

```
mfhi    d       #  d <-- hi.  Move From Hi

mflo    d       #  d <-- lo.  Move From Lo
```

Rule: Do no use a multiply or a divide instruction within two instructions after mflo or mfhi. The reason for this involves the way the MIPS pipeline works, and is a future topic. On the SPIM simulator this rule does not matter.

QUESTION 4:

How small are the the operands of multu if the result fits in lo?

How small are the the operands of multu if the result fits in lo?

A good answer might be:

The operands must fit into 16 bits, or must be less than $2^{16} - 1 = 64K - 1$.

Example Program

Let us write a program that evaluates the same formula as the previous example: $5 \times x - 74$ where the value x is in register $8. Assume that x is two's complement. Here is the program:

```
## newMult.asm
##
## Program to calculate 5 × x - 74
##
## Register Use:
##  $8   x
##  $9   result

        .text
        .globl  main

main:
        ori     $8,   $0, 12       # put x into $8
        ori     $___, $0,  5       # put 5 into $___
        mult    $___, $___         #  ___ <-- 5x
        mflo    $___               # $___ = 5x
        addiu   $___, $___,-74     # $___ = 5x - 74

## End of file
```

QUESTION 5:

Fill in the blanks.

Next

A good answer might be:

The answer is below.

Completed Program

```
## newMult.asm
##
## Program to calculate 5 × x - 74
##
## Register Use:
##  $8   x
##  $9   result

        .text
        .globl  main

main:
        ori     $8, $0, 12      # put x into $8
        ori     $9, $0,  5      # put 5 into $9
        mult    $9, $8          # lo <-- 5x
        mflo    $9              # $9 = 5x
        addiu   $9, $9,-74      # $9 = 5x - 74

## End of file
```

QUESTION 6:

What does the "u" mean in each of the following instructions:

- addu ______________________
- multu______________________

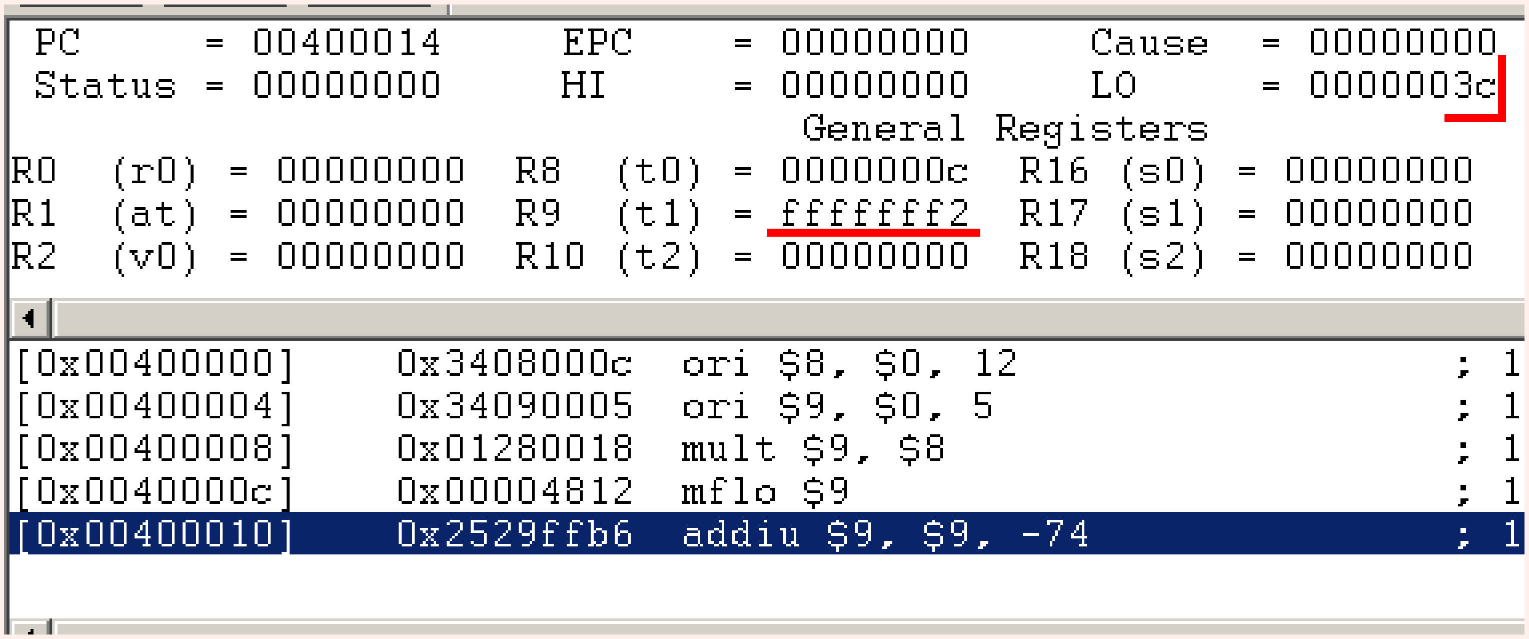
Next

A good answer might be:

- addu Do not trap on overflow.
- multu Operands are unsigned.

A Run of the Program

The mult assumes two's complement operands. Here is a run of the program:



```
 PC     = 00400014    EPC     = 00000000    Cause  = 00000000
 Status = 00000000    HI      = 00000000    LO     = 0000003c
                              General Registers
R0  (r0) = 00000000  R8  (t0) = 0000000c  R16 (s0) = 00000000
R1  (at) = 00000000  R9  (t1) = fffffff2  R17 (s1) = 00000000
R2  (v0) = 00000000  R10 (t2) = 00000000  R18 (s2) = 00000000

[0x00400000]    0x3408000c  ori $8, $0, 12                ; 1
[0x00400004]    0x34090005  ori $9, $0, 5                 ; 1
[0x00400008]    0x01280018  mult $9, $8                   ; 1
[0x0040000c]    0x00004812  mflo $9                       ; 1
[0x00400010]    0x2529ffb6  addiu $9, $9, -74             ; 1
```

The result is as expected. 5 × 12 - 74 = -14 = 0xFFFFFFF2. The result 5 × 12 = 60_{ten} = 0x3C remains in lo.

QUESTION 7:

Use integer division (in base ten) to calculate the quotient and remainder of:

- 99 / 2 = ________ R _______
- 99 / 50 = ________ R _______

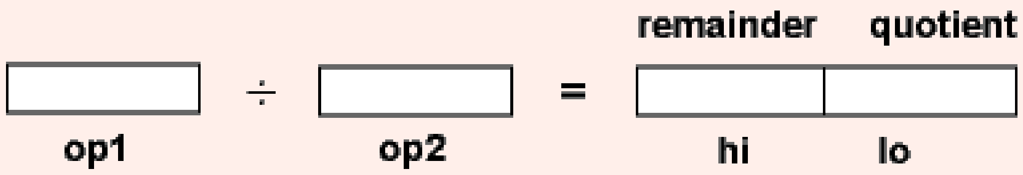
Next

A good answer might be:

- 99 / 2 = 48 R 1
- 99 / 50 = 1 R 49

The div and the divu Instructions

With N-place integer division there are two results, an N-place quotient and an N-place remainder. With 32-bit operands there will be (in general) two 32-bit results. MIPS uses the **hi** and **lo** registers for the results:



Here are the MIPS instructions for integer divide. The "u" means operands and results are in unsigned binary.

```
div    s,t       #  lo <-- s div t
                 #  hi <-- s mod t
                 #  two's complement

divu   s,t       #  lo <-- s div t
                 #  hi <-- s mod t
                 #  unsigned
```

QUESTION 8:

(Review:) What instruction would be used to move the quotient into register $8?

Next

A good answer might be:

mflo $8

The instructions mflo and mfhi are used to get the results of an integer divide.

Example Program

For this example say that we wish to calculate (y + x) / (y - x). The argument x is in $8; y is in $9. The quotient is to be placed in $10 and the remainder in $11. Assume two's complement integers. Here is the program. Sadly, it has some holes:

```
## divEg.asm
##
## Program to calculate (y + x) / (y - x)
##
## Register Use:
##  $8   x
##  $9   y
##  $10  x/y
##  $11  x%y

        .text
        .globl  main

main:
        ___      $8,   $0,  8        # put x into $8
        ___      $9,   $0, 36        # put y into $9
        addu     $10,  $__, $__      # $10  <-- (y+x)
        subu     $11,  $__, $__      # $11  <-- (y-x)
        div      $__,  $__           # hilo <-- (y+x)/(y-x)
        ____     $10                 # $10  <-- quotient
        ____     $11                 # $11  <-- remainder

## End of file
```

QUESTION 9:

Fill in the holes.

Next

A good answer might be:

See Below

Filled Holes

Here is the complete program:

```
## divEg.asm
##
## Program to calculate (y + x) / (y - x)
##
## Register Use:
##  $8   x
##  $9   y
##  $10  x/y
##  $11  x%y

        .text
        .globl  main

main:
        ori     $8,   $0,  8         # put x into $8
        ori     $9,   $0, 36         # put y into $9
        addu    $10,  $9, $8         # $10  <-- (y+x)
        subu    $11,  $9, $8         # $11  <-- (y-x)
        div     $10,  $11            # hilo <-- (y+x)/(y-x)
        mflo    $10                  # $10  <-- quotient
        mfhi    $11                  # $11  <-- remainder

## End of file
```

QUESTION 10:

(36+8) / (36-8) = _____ R _____

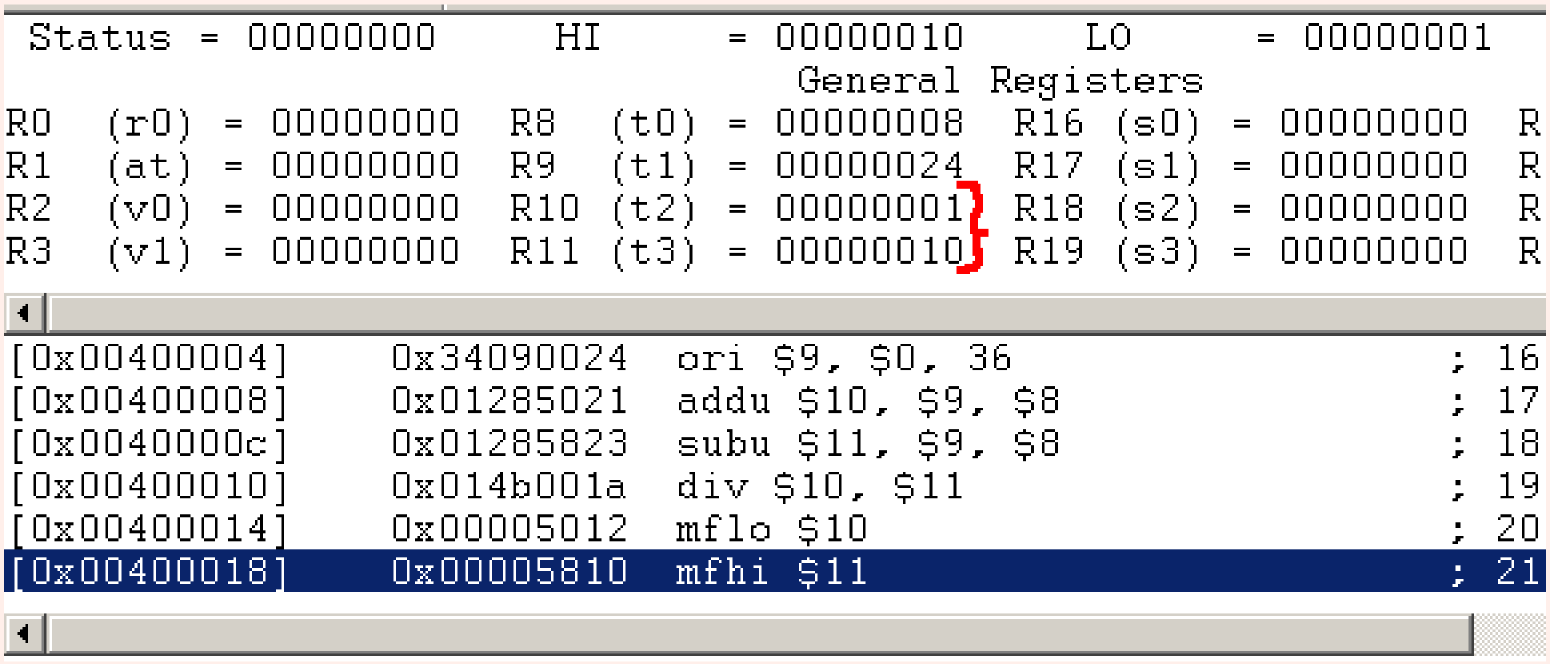
Next

A good answer might be:

(36+8) / (36-8) = 1 R 16, or 0x1 R 0x10

A Run of the Program

Here is an example run of the program:



```
 Status  = 00000000     HI      = 00000010     LO      = 00000001
                                  General Registers
R0   (r0)  = 00000000  R8   (t0)  = 00000008  R16 (s0)  = 00000000  R
R1   (at)  = 00000000  R9   (t1)  = 00000024  R17 (s1)  = 00000000  R
R2   (v0)  = 00000000  R10  (t2)  = 00000001  R18 (s2)  = 00000000  R
R3   (v1)  = 00000000  R11  (t3)  = 00000010  R19 (s3)  = 00000000  R

[0x00400004]    0x34090024  ori $9, $0, 36                          ; 16
[0x00400008]    0x01285021  addu $10, $9, $8                        ; 17
[0x0040000c]    0x01285823  subu $11, $9, $8                        ; 18
[0x00400010]    0x014b001a  div $10, $11                            ; 19
[0x00400014]    0x00005012  mflo $10                                ; 20
[0x00400018]    0x00005810  mfhi $11                                ; 21
```

As usual, a stunning success.

QUESTION 11:

Here is the 16-bit two's complement rep^n for -16.

1111 1111 1111 0000

Perform a logical shift right by two positions. Is the resulting pattern the correct rep^n for -16/4?

Next

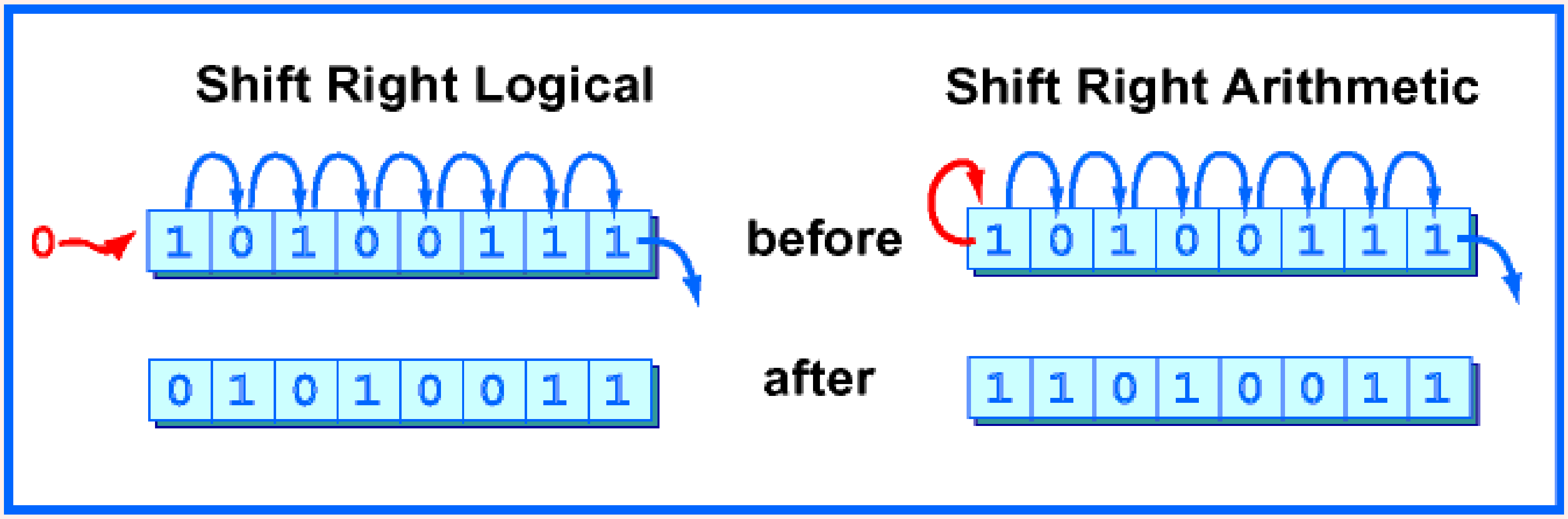
A good answer might be:

1111 1111 1111 0000 ---> 0011 1111 1111 1100

Is the resulting pattern the correct rep^n for -16/4? No. The result represents a large positive number, not -4

Shift Right Arithmetic

The problem is that a shift right logical moves zeros into the high order bit. This is correct in some situations, but not for dividing two's complement negative integers. An **arithmetic** right shift replicates the sign bit as needed to fill bit positions:



QUESTION 12:

Is there a need for an *arithmetic shift left* instruction?

Next

Is there a need for an *arithmetic shift left* instruction?

A good answer might be:

No. A logical shift left moves zeros into the low-order bit, which is correct for both signed and unsigned integers.

The sra Instruction

Here is the MIPS instruction that performs an arithmetic shift right:

```
sra    d,s,shft   #  $d <-- s shifted right
                  #  shft bit positions.
                  #  0 =< shft < 31
```

Sometimes you need to divide by two. This instruction is faster and more convenient than the div instruction.

QUESTION 13:

Does the sra instruction give the correct results for *unsigned* integers?

Next

Does the sra instruction work correctly for *unsigned* integers?

A good answer might be:

No. For unsigned integers the "sign bit" should not be replicated.

End of Chapter

You have reached the end of this chapter. Click on a blue subject that has your undivided attention to go to where it was discussed. To get back here, click on the "back arrow" button of your browser.

- The number of places resulting from an integer multiplication.
- The number of places resulting from an integer division.
- The hi and lo registers.
- The mult and multu instructions.
- The mfhi and mflo instructions.
- The div and divu instructions.
- Arithmetic shift right.
- The sra instruction.

contents Go to Table of Contents

You have reached the end of the chapter.

created: 07/12/01; slight edits: 06/12/02

CHAPTER 15 — Memory Access Instructions

This chapter discusses how to get data from memory into registers, and how to write data to memory from registers. Addresses are calculated at run time by adding an **offset** (part of the machine instruction) to an address held in a **base register**.

Chapter Topics:

- *Load and store.*
- *Data alignment.*
- *Byte order (little endian and big endian).*
- *The* lw *and* sw *instructions.*
- *The load delay slot.*
- *Base registers and address calculation.*
- *The* lui *instruction.*
- *Symbolic addresses.*

QUESTION 1:

(Review:) What is the name of the operation that copies data from main memory into a register?

Next

A good answer might be:

A register is **loaded** from memory.

Load and Store

All arithmetic and logic operations on MIPS use data in registers. To operate on data in main memory, the data must first be put into a register. A **load** operation copies data from main memory into a register. A **store** operation copies data from a register into main memory .

When a word (4-bytes) is loaded or stored the memory address must be a multiple of four. This is called an alignment restriction. Addresses that are a multiple of four are called **word aligned**. This restriction makes the hardware simpler and faster.

The MIPS instruction that load a word into a register is the `lw` instruction. The store word instruction is `sw`. Each must specify a register and a memory address (details in a few pages).

QUESTION 2:

Which of the following addresses are word aligned?

- 0x000AE430
- 0x00014432
- 0x000B0737
- 0x0E0D8844

Hint: how can you multiply by four in binary?

A good answer might be:

How can you multiply by four in binary? By shifting left 2 positions. So if the rightmost two (or more) bit positions are zero, the integer is a multiple of four.

- 0x000AE430 Yes.
- 0x00014432 No.
- 0x000B0737 No.
- 0x0E0D8844 Yes.

Big Endian and Little Endian

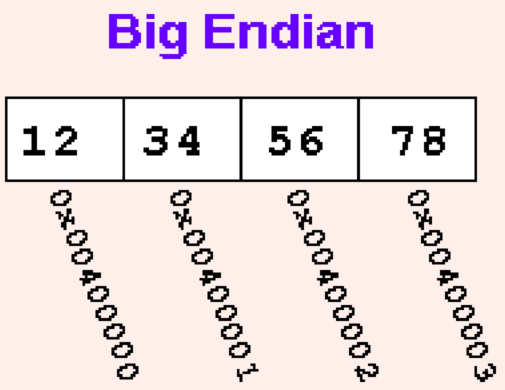
The **lowest address** is used for the address of a block of contiguous bytes. For example, the address of a four-byte word is the lowest address of the four bytes.

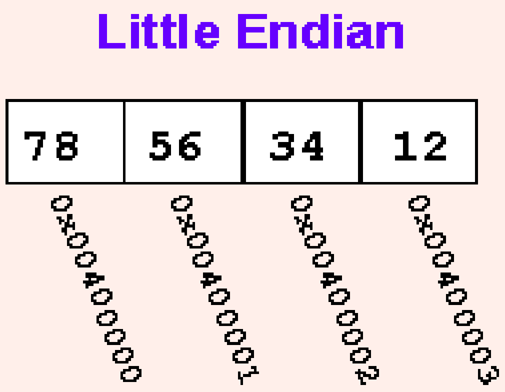
Another issue is how a 32-bit pattern is held in a word of memory. There are 32 bits in the word and 32 bits in the pattern, but a choice has to be made about which byte of memory gets what part of the pattern. There are two ways that computers can do this:

> **Big Endian Byte Order:** The most significant byte (the "big end") of the data is placed at the lowest addressed byte. The rest of the word is placed in order in the next three bytes in address space.
>
> **Little Endian Byte Order:** The least significant byte (the "little end") of the data is placed at the lowest addressed byte. The rest of the word is placed in order in the next three bytes in address space.

In these definitions, the 32-bit word is regarded as a 32-bit unsigned integer. The "most significant" byte is the one for the largest powers of two: 2^{31}, ..., 2^{24}. The "least significant" byte is the one for the smallest powers of two: 2^7, ..., 2^0.

For example, say that the 32-bit word 0x12345678 is at address 0x00400000. The most significant byte is 0x12; the least significant is 0x68. Here are the two byte orders:





Within a **byte** the order of the bits is always the same (no matter how the bytes themselves are arranged).

QUESTION 3:

Write the bit pattern that is contained in the byte (above) that contains 0x12.

Write the bit pattern that is contained in the byte (above) that contains 0x12.

A good answer might be:

0001 0010

Within a byte, position 7 is the most signficant bit, usually printed as the left-most bit. So a byte that contains 0x12 looks the same for both byte orderings.

Byte Order of MIPS and SPIM

The MIPS processor chip can be set up in hardware to use either byte ordering. A computer system designer makes whatever choice best fits the rest of the computer system. SPIM simulator uses the byte ordering of the computer it is running on.

- Intel 80x86: little-endian.
- Macintosh: big-endian.

The examples in these notes were done on a Windows/Intel computer. If you are using a Macintosh there will be occasional differences.

QUESTION 4:

In programs that deal with data in word-sized chunks or bigger, is the byte ordering visible?

Next

In programs that deal with data in word-sized chunks or bigger, is the byte ordering visible?

A good answer might be:

No. If a word is loaded from memory the electronics puts the bytes into the register in the correct order. If a word is stored to memory the bytes are written to memory in the correct order. It makes no difference which order is the "correct" order.

MIPS Addresses

The MIPS instruction that load a word into a register is the `lw` instruction. The store word instruction is `sw`. Each must specify a register and a memory address. A MIPS instruction is 32 bits (always). A MIPS memory address is 32 bits (always). How can a load or store instruction refer to an address that is the same size as itself?

Any instruction that refers to memory uses a **base register**. The base register contains a 32-bit address. The instruction contains an **offset**, a 16-bit signed integer. The sum of the address in the base register with the (sign-extended) offset forms the memory address. Here is the load word instruction:

```
lw   t,off(b)      # $t <-- Word from memory address b+off
                   # b is a register. off is 16-bit two's complement.
```

QUESTION 5:

Load the word at address 0x00400060 into register $8. Register $10 contains 0x00400000. Write the `lw` instruction in assembly language:

```
lw $8, _____(   )
```

Next

A good answer might be:

```
0x00400060 --- address of data
0x00400000 --- address in $10
$8         --- destination register
```

The instruction is:

```
lw $8,0x60($10)
```

Machine Instruction for Load Word

Here is the machine code version of the instruction. It specifies the *base register*, the *destination register*, and the *offset*. It does not directly contain the memory address.

```
100011  01010 01000 0000 0000 0110 0000 -- fields of the instruction

opcode  base  dest       offset        -- meaning of the fields

lw      $10   $8     0    0    6    0
```

Here is how this instruction is executed:

1. The 32-bit address in $10 is fetched. 0x00400000
2. The offset is sign-extended to 32 bits. 0x00000060
3. The memory address is the 32-bit sum of the above. 0x00400060
4. The word at that address is fetched from memory. get 4 bytes starting at 0x00400060
5. After a one machine cycle delay the word is loaded into $8. $8 <--The 4 bytes.

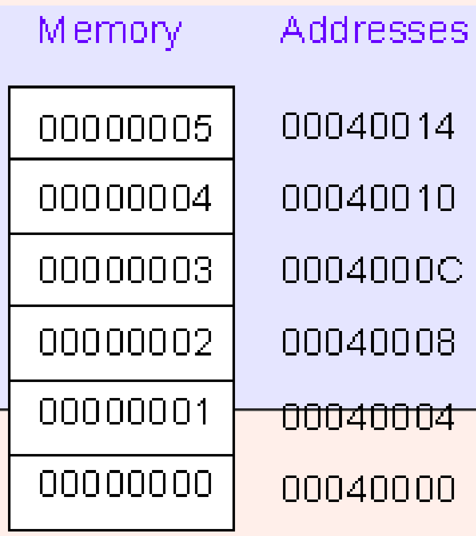
There is a one machine cycle delay before the data from memory is available. This is the **load delay slot**. But the processor keeps executing instructions. The instruction after a lw instruction should not use the data that is being loaded. Sometimes the instruction after the lw is a no-operation instruction.

QUESTION 6:

Conditions are as follows and memory is as at right. Write the instruction that puts the value 0x00000004 into register $12.

- Register $12 contains 0xFFFFFFFF
- Register $13 contains 0x00040000

lw $\_\_\_\_\_ , \_\_\_\_\_($)



Memory	Addresses
00000005	00040014
00000004	00040010
00000003	0004000C
00000002	00040008
00000001	00040004
00000000	00040000

A good answer might be:

lw $12, 0x10($13)

The original contents of $12 are irrelevant; they are replaced with the 32 bits from memory (memory remains unchanged).

Store Word Instruction

The store word instruction, sw contains the same specifications as the load word instruction, but now the word of data is copied to memory. The register is not changed.

```
sw   t,off(b)      # Word at memory address (b+off) <-- $t
                   # b is a register. off is 16-bit two's complement.
```

As with the lw instruction, the memory address must be word aligned (a multiple of four).

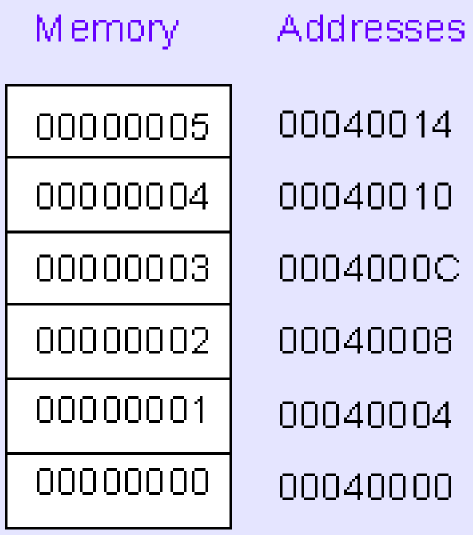
QUESTION 7:

Conditions are as follows and memory is as at right. Write the instruction that puts the word $0xFFFFFFFF into register memory location 0x0004000C.

- Register $12 contains 0xFFFFFFFF
- Register $13 contains 0x00040014

sw $_____ , _____($   )

Hint: it is OK to specify the 16-bit offset as a signed decimal integer.



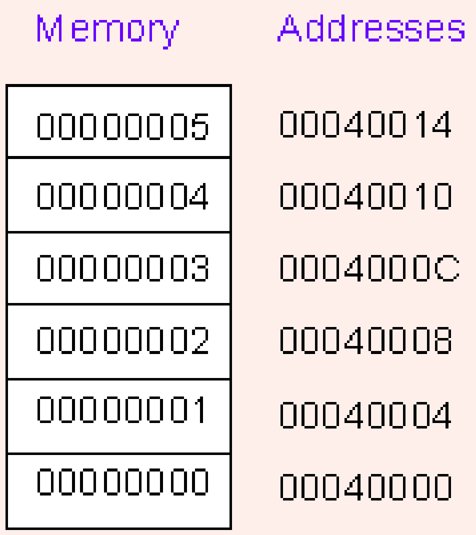
Memory	Addresses
00000005	00040014
00000004	00040010
00000003	0004000C
00000002	00040008
00000001	00040004
00000000	00040000

A good answer might be:

sw $12 , 0xFFF8($13) or **sw $12 , -8($13)**

Setting up the Base Register

The first instruction of the answer expresses minus eight using 16-bit two's complement. This is the bit pattern that is actually contained in the machine instruction. This is awkward to read and to calculate. The second instruction uses signed decimal notation to specify minus eight. The assembler translates both assembler instructions into exactly the same machine instruction.



Memory	Addresses
00000005	00040014
00000004	00040010
00000003	0004000C
00000002	00040008
00000001	00040004
00000000	00040000

With the assistance of a 32-bit base register, a 32-bit lw or sw instruction can reference memory But how does the base address get into the base register? This is where the lui (**l**oad **u**pper **i**mmediate) instruction is useful. It copies the 16-bit immediate operand of the machine instruction to the *upper two bytes* of the designated register.

```
lui  t,const  # upper two bytes of $t <-- two byte const
              # lower two bytes of $t <-- 0x0000
```

Sometimes this is all that you need. For example, say that memory is as in the picture, and that you want to load the word at 0x00040010 into $12. The lui instruction can set up the base register:

lui $13, 0x________
lw $12, 10($13)

QUESTION 8:

Complete the lui instruction.

A good answer might be:

```
lui $13, 0x0004
lw  $12, 10($13)
```

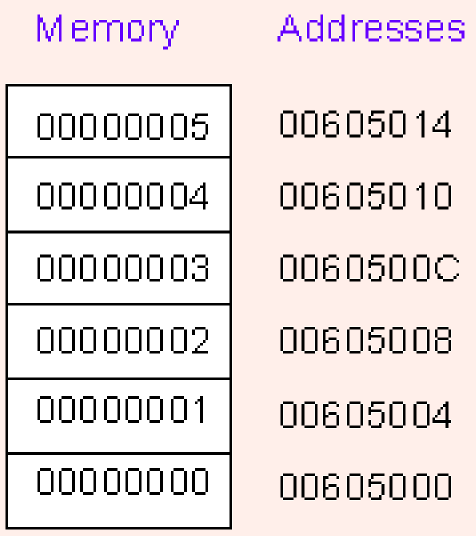
After the lui instruction $13 contains 0x**0004**0000

Filling in the bottom Half

By using the lui instruction the base register can be loaded with multiples of 0x00010000. But often you want a more specific address in the base register. Use the ori instruction to fill the bottom 16 bits.

Also, addiu can be used to do the same thing. Be careful NOT to use addi, because it does sign extension of its 16-bit immediate operand. This can cause problems.

Say that memory is as at right. The lw instruction (below) will load the word at 0x0060500C into $12.



Memory	Addresses
00000005	00605014
00000004	00605010
00000003	0060500C
00000002	00605008
00000001	00605004
00000000	00605000

```
lui $13, 0x________
ori $13, 0x________
lw  $12, 0xC($13)
```

QUESTION 9:

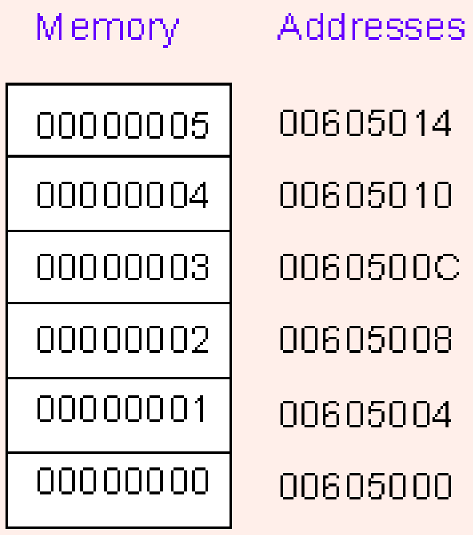
Complete the instruction sequence.

A good answer might be:

```
lui $13, 0x0060
ori $13, 0x5000
lw  $12, 0xC($13)
```

Alternate Sequence

Other sequences of instructions also will work: Because the "upper half" of an address is 16 bits and the offset of the lw instruction is 16 bits, the two in combination can address any byte of memory.



The problem was to load $12 with the word at address 0x0060500C. Here is another way to do it: Split the address into halves: 0x0060 and 0x500C. Load the top half into $13 and use the bottom half as the offset.

```
lui $13, 0x0060
lw  $12, 0x500C($13)
```

An array of int in C is implemented as a sequence of words in successive word-aligned memory locations. For example, the diagram shows a possible run-time implementation of:

```
int data[] = {0, 1, 2, 3, 4, 5};
```

QUESTION 10:

What is the most sensible address to have in the base register for processing this array?

A good answer might be:

The address of data[0]: 0x00605000. In fact, in ANSI C, the identifier for an array (in this case *data*) stands for the address of its first element. At run time this address will likely be in a base register.

Example Program

You may be thinking that there has got to be an easier way to load a register from memory. At the machine language level there is not. However, the assembler has features that make it much easier to write lw and sw instructions. These are discussed in a later chapter.

Example Program. Evaluate the polynomial $5x^2 -12x + 97$ for a value **x** in memory. Store the result at location **poly** in memory.

QUESTION 11:

How many lw instructions will be needed?

How many sw instructions will be needed?

Next

A good answer might be:

How many lw instructions will be needed? One, near the start of the program to load **x** into a register.

How many sw instructions will be needed? One, near the end of the program to save the result in **poly**.

Symbolic Address

In the description of this problem, memory locations were called **x** and **poly**. Of course, at run time, addresses are 32-bit integers. But at assembly language time it is convenient to use names for memory locations. These names are called **symbolic addresses**. One of the most important features of an assembler is support for symbolic addresses. In the following example we will ignore some of this support in favor of explaining how the hardware instructions work. Here is a start on the program:

```
## poly.asm
##
## evaluate  5x^2 -12x + 97
##

        .text
        .globl  main

main:

        . . . . many instructions

        .data
x:      .word   17            # base register points here
poly:   .word   0

## End of file
```

The assembler directive **.data** means: "*here is the start of the data section of memory*". The assembler directive **.word** means: "*put a 32-bit two's complement integer here*".

QUESTION 12:

The assembler in SPIM automatically assembles the .data section starting at address 0x10000000.

Next

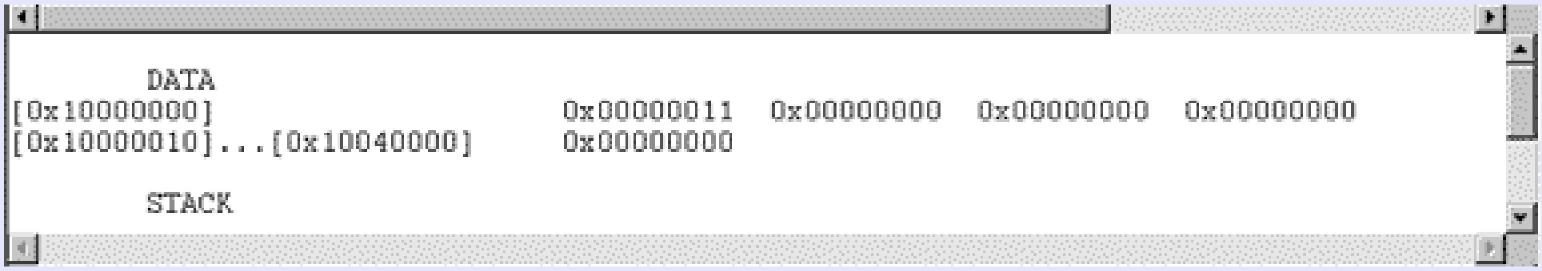
1. What address corresponds to the symbolic address **x**?
2. What address corresponds to the symbolic address **poly**?

A good answer might be:

The assembler in SPIM automatically assembles code starting at address 0x10000000.

1. What address corresponds to the symbolic address **x**? 0x10000000
2. What address corresponds to the symbolic address **poly**? 0x10000004

Here is what this part of the SPIM simulation looks like:



```
        DATA
[0x10000000]                  0x00000011  0x00000000  0x00000000  0x00000000
[0x10000010]...[0x10040000]   0x00000000

        STACK
```

More Code

Here is an important addition to the program. Decide on the registers you need and what they are used for. Then write down your decisions! This is crucial for getting things correct.

```
## poly.asm
##
## evaluate  5x^2 -12x + 97
##
## Register Use:
##
## $10 base register, address of x
## $11 x
## $12 value of the polynomial
## $13 temporary

        .text
        .globl  main

main:
        lui   $10,_______     #  Init base register
        lw    $11,0($_____)  #  Load x

        ori   $12,$0,_____    #  Initialize the accumulator
```

```
                        #  during the "load delay slot"

        . . .  many instructions

        sw    $12,___($___)  #  Store result in poly

        .data
x:      .word   17          # base register points here
poly:   .word   0

## End of file
```

A register where a value is built up after several calculations is called an **accumulator**. (Some old processors have a single, special register that is used for this purpose. But MIPS has many general purpose registers for this).

Remember that data loaded from memory is not available to the instruction following the load. The instruction after a lw, in the "load delay slot", should not try to use the loaded data.

QUESTION 13:

Fill in the blanks. Look at the previous answer to help with the lui instruction. Use it to load the upper half of the base register with the upper half of the first data address.

Next

A good answer might be:

See below.

Second Term

Now fill in the blanks so that the second term is evaluated and added to the accumulator.

```
## poly.asm
##
## evaluate  5x^2 -12x + 97
##
## Register Use:
##
## $10 base register, address of x
## $11 x
## $12 value of the polynomial
## $13 temporary

        .text
        .globl  main

main:
        lui   $10,0x1000     #  Init base register
        lw    $11,0($10)     #  Load x

        ori   $12,$0,97      #  Initialize the accumulator
                             #  during the "load delay slot"

        ori   $13,$0,12      #  evaluate second term
        mult  $__,$__        #  12x
        ____  $13            #  assume 32 bit result
        subu  $__,$__,$__    #  accumulator = -12x +97

       ....   instructions

        sw    $12,4($10)     #  Store result in poly

        .data
x:      .word   17           # base register points here
poly:   .word   0

## End of file
```

QUESTION 14:

Fill in the blanks.

Next

A good answer might be:

All blanks filled, as below:

Third Term

At this point all we need to do is square x, multiply by five, and add the result to the accumulator. After squaring x we don't need its value anymore, so x^2 can be put back into register $11.

```
## poly.asm
##
## evaluate  5x^2 -12x + 97
##
## Register Use:
##
## $10 base register, address of x
## $11 x
## $12 value of the polynomial
## $13 temporary

        .text
        .globl  main

main:
        lui   $10,0x1000     #  Init base register
        lw    $11,0($10)     #  Load x

        ori   $12,$0,97      #  Initialize the accumulator
                             #  during the "load delay slot"

        ori   $13,$0,12      #  evaluate second term
        mult  $11,$13        #  12x
        mflo  $13            #  assume 32 bit result
        subu  $12,$12,$13    #  accumulator = -12x +97

                             #  evaluate third term
        mult  $__,$__        #  x^2
        mflo  $__            #  assume 32 bit result

        . . . . instructions

        sw    $12,4($10)     #  Store result in poly
        .data
x:      .word   17          #  base register points here
poly:   .word   0

## End of file
```

QUESTION 15:

Fill in the blanks.

A good answer might be:

See below.

More of the Third Term

Here is the rest of the program:

```
## poly.asm -- complete program
##
## evaluate  5x^2 -12x + 97
##
## Register Use:
##
## $10 base register, address of x
## $11 x
## $12 value of the polynomial
## $13 temporary

        .text
        .globl  main

main:
        lui   $10,0x1000     #  Init base register
        lw    $11,0($10)     #  Load x

        ori   $12,$0,97      #  Initialize the accumulator
                             #  during the "load delay slot"

        ori   $13,$0,12      #  evaluate second term
        mult  $11,$13        #  12x
        mflo  $13            #  assume 32 bit result
        subu  $12,$12,$13    #  accumulator = -12x +97

                             #  evaluate third term
        mult  $11,$11        #  x^2
        mflo  $11            #  assume 32 bit result
        ori   $13,$0,___     #  5
        mult  $___,$___      #  5x^2
        mflo  $13            #
        addu  $12,$___,$___  #  accumulator = 5x^2-12x+97

        sw    $12,4($10)     #  Store result in poly
        .data
x:      .word   17
poly:   .word   0

## End of file
```

Happily, after filling in the blanks, the program is finished.

QUESTION 16:

Fill in the blanks.

Next

A good answer might be:

See below.

Complete Program

Here is the complete program. You may wish to copy it to the clipboard and paste it into your text editor. Now you can save it to a file and run it with SPIM.

```
## poly.asm -- complete program
##
## evaluate  5x^2 -12x + 97
##
## Register Use:
##
## $10 base register, address of x
## $11 x
## $12 value of the polynomial
## $13 temporary

        .text
        .globl  main

main:
        lui   $10,0x1000     #  Init base register
        lw    $11,0($10)     #  Load x

        ori   $12,$0,97      #  Initialize the accumulator
                             #  during the "load delay slot"

        ori   $13,$0,12      #  evaluate second term
        mult  $11,$13        #  12x
        mflo  $13            #  assume 32 bit result
        subu  $12,$12,$13    #  accumulator = -12x +97

                             #  evaluate third term
        mult  $11,$11        #  x^2
        mflo  $11            #  assume 32 bit result
        ori   $13,$0,5       #  5
        mult  $11,$13        #  5x^2
        mflo  $13            #
        addu  $12,$12,$13    #  accumulator = 5x^2-12x+97
        sw    $12,4($10)     #  Store result in poly

        .data
x:      .word   17
poly:   .word   0

## End of file
```

Of course, the program should be tested with a careful selection of values for **x**. A production quality program would document the upper and lower bounds for **x**.

QUESTION 17:

Suggest three values for **x** for use in testing.

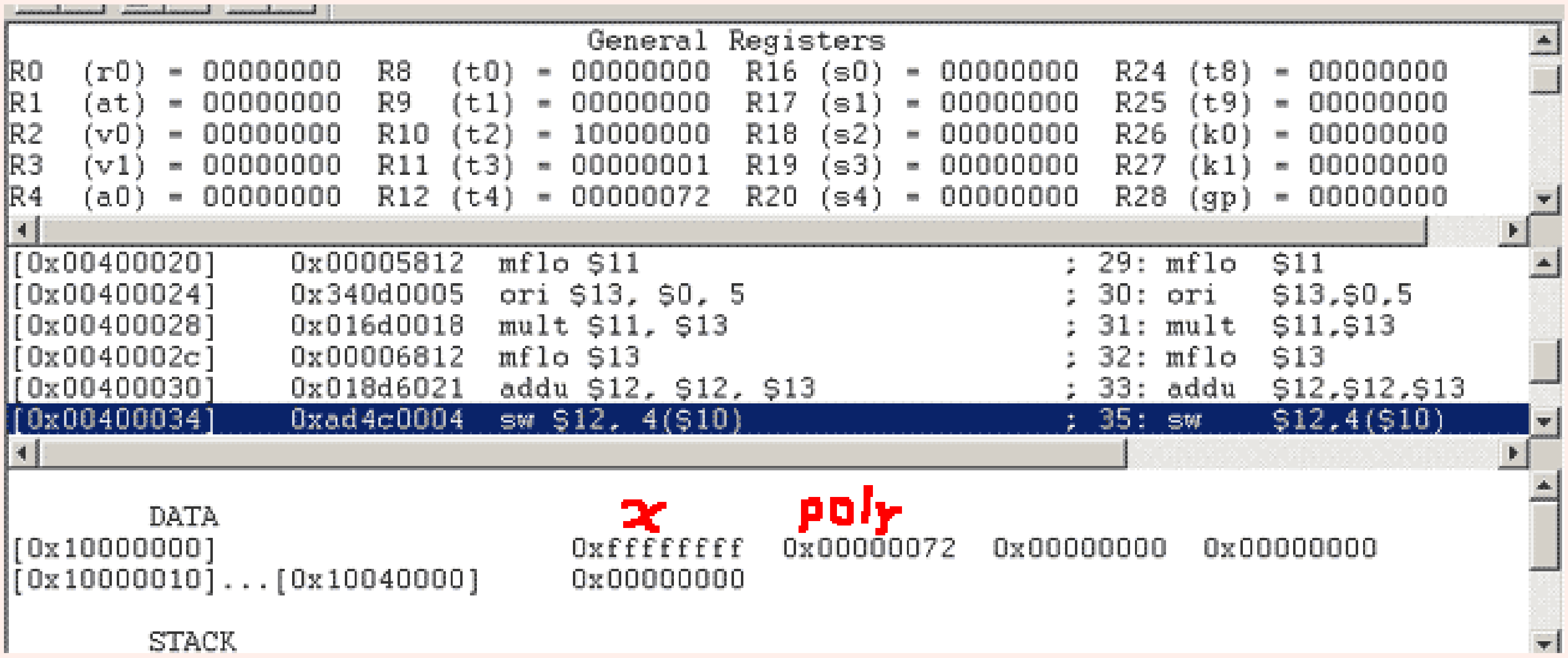
Suggest three values for **x** for use in testing.

A good answer might be:

0, 1, -1. Of course you don't stop there, but running with these three values often reveals problems.

A Run of the Program

Here is a run of the program with **x** = -1. The result, 0x72 = 114_{10} is correct. As always, running is done by single-stepping (pushing F10). The PC is initialized to 0x00400000.



```
                              General Registers
R0  (r0) = 00000000  R8  (t0) = 00000000  R16 (s0) = 00000000  R24 (t8) = 00000000
R1  (at) = 00000000  R9  (t1) = 00000000  R17 (s1) = 00000000  R25 (t9) = 00000000
R2  (v0) = 00000000  R10 (t2) = 10000000  R18 (s2) = 00000000  R26 (k0) = 00000000
R3  (v1) = 00000000  R11 (t3) = 00000001  R19 (s3) = 00000000  R27 (k1) = 00000000
R4  (a0) = 00000000  R12 (t4) = 00000072  R20 (s4) = 00000000  R28 (gp) = 00000000

[0x00400020]    0x00005812  mflo $11                   ; 29: mflo  $11
[0x00400024]    0x340d0005  ori $13, $0, 5             ; 30: ori   $13,$0,5
[0x00400028]    0x016d0018  mult $11, $13              ; 31: mult  $11,$13
[0x0040002c]    0x00006812  mflo $13                   ; 32: mflo  $13
[0x00400030]    0x018d6021  addu $12, $12, $13         ; 33: addu  $12,$12,$13
[0x00400034]    0xad4c0004  sw $12, 4($10)             ; 35: sw    $12,4($10)

       DATA
[0x10000000]                    0xffffffff  0x00000072  0x00000000  0x00000000
[0x10000010]...[0x10040000]     0x00000000

       STACK
```

Create a source file and play around with the program. Put some bugs into the program and see what they do. Experimentally determine the range allowed for **x**.

QUESTION 18:

How can you solve for the allowed range of **x**?

Next

How can you solve for the allowed range of **x**? (Don't actually do this unless you want to).

A good answer might be:

The result, **poly** must fit into 32 bits, two's complement. So -2^{31} <= 5x^2 -12x + 97 <= 2^{31} - 1. Further algebraic fussing gives the range of **x**. (See the exercises).

End of Chapter

You have reached the end of this chapter. Click on a blue subject to go to where it was discussed. To get back here, click on the "back arrow" button of your browser.

- word alignment
- Byte order: little endian and big endian
- MIPS and SPIM byte order
- address calculation
- lw instruction
- Operation of the lw instruction
- load delay slot
- sw instruction
- lui instruction
- symbolic address
- .word directive
- .data directive
- register use table
- accumulator

Go to Table of Contents

You have reached the end of the chapter.

created: 07/16/01; small edits: 06/12/02

CHAPTER 16 — More Memory Access

This chapter discusses some more instructions that load registers (from memory) and that store registers (to memory). These instructions are used less frequently than lw and sw.

Chapter Topics:

- *Load byte and store byte:* lb, lbu, *and* sb
- *Load halfword and store halfword:* lh, lhu, *and* sh
- *Arithmetic with less than 32 bits.*

QUESTION 1:

(Review:) What is the smallest addressable unit of main memory?

Next

A good answer might be:

A byte

Loading a Single Byte

There are two instructions that load a byte from a memory address. The address is calculated at run time by adding an offset to a base register (just as with the load word and store word instructions). The instructions differ in how the 8-bit byte is put into the 32-bit register.

```
lb   t,off(b)  # $t <-- Sign-extended byte
               # from memory address b+off
               # b is a base register.
               # off is 16-bit two's complement.
```

The `lb` instruction copies bit 7 of the byte to bits 8-31 of the register (all bits to the right of bit 7). Use this instruction when the byte is regarded as an 8-bit signed integer in the range -128...+127 and you want it to become a 32-bit integer in the same range.

```
lbu   t,off(b) # $t <-- Zero-extended byte
               # from memory address b+off
               # b is a base register.
               # off is 16-bit two's complement.
```

The `lbu` instruction fills the bits to the right of bit 7 with zeros. Use this instruction when the byte is regarded as a ascii character or 8-bit unsigned integer.

QUESTION 2:

- Memory at 0x10000007 contains the byte 0xA4
- Register $8 contains 0x10000000

What is put in register $10 when the following instruction is executed:

```
lb    10,7($8)
```

A good answer might be:

0xFFFFFFA4   Bit 7 of 0xA4 is one; lb extends that bit to the high order three bits of $10.

Storing a Single Byte

Loading and storing bytes is used for processing text and for low-level systems programs (such as assemblers and operating systems). Graphics programs, also, make frequent use of these operations. Both operations could be done using lw and sw along with bit manipulation instructions, but it is convenient and fast to have byte length load and store.

There is a single instruction for storing a byte:

```
sb    t,off(b)   # The byte at off+b <-- low-order
                 # byte from register $t.
                 # b is a base register.
                 # off is 16-bit two's complement.
```

There is no need for two "store byte" instructions. Whatever is in the low-order (right) byte of the register is copied to memory. The other bytes of the register are ignored. Of course, the register does not change.

QUESTION 3:

- Memory at 0x10000519 contains the byte 0x44
- Register $8 contains 0x10000400
- Register $10 contains 0xFA034183

Write the instruction that replaces the "0x44" in memory with "0x83".

A good answer might be:

```
sb    10,0x119($8)
```

All Addresses Great and Small

There are no alignment requirements for the three instructions: lb, lbu, and sb. Any byte in memory that is acceptable for program data can be used. This is fortunate since character strings are usually stored in successive bytes. (Remember that there are areas of MIPS main memory reserved for special uses).

Byte load and store instructions are often used for input and output with media that must be used on many systems. Often the data written to magnetic tape by one government agency is used by other agencies using different computers. To make the data transportable, the format of the data is described byte by byte. The format must be followed, regardless of the byte order of the computers writing or reading the data.

QUESTION 4:

Say that the format of a tape record requires big-endian integers. Complete the following instructions so that the integer in register $9 is written to the four bytes starting at address 0x10000000 (the tape i/o buffer, let us pretend). Put the most significant byte at the starting address.

Next

```
lui  $8,0x1000      # $8 is base register
sb   $9,____($8)    # least significant byte
srl  $9,$9,_____    # move next byte to low order
sb   $9,____($8)    # bits 8-15
srl  $9,$9,_____    # move next byte to low order
sb   $9,____($8)    # bits 16-23
srl  $9,$9,_____    # move next byte to low order
sb   $9,____($8)    # most significant byte
```

A good answer might be:

```
lui  $8,0x1000      # $8 is base register
sb   $9,3($8)       # least significant byte
srl  $9,$9,8        # move next byte to low order
sb   $9,2($8)       # bits 8-15
srl  $9,$9,8        # move next byte to low order
sb   $9,1($8)       # bits 16-23
srl  $9,$9,8        # move next byte to low order
sb   $9,0($8)       # most significant byte
```

Tape Writer

The least significant byte of the register is written to memory first (because it is already where the sb instruction needs it). Then the remaining bytes of $9 are shifted into the right-most byte one by one and written to memory. Here is a complete version of the program:

```
## endian.asm
##
## copy $9 to memory in big-endian order
##
## Register Use:
## $8  --- first byte of the tape block
## $9  --- 4-byte integer

      .text
      .globl  main

main:
      lui  $9,0x1234      # put data in $9
      ori  $9,0x5678      #
      lui  $8,0x1000      # $8 is base register
      sb   $9,3($8)       # least significant byte
      srl  $9,$9,8        # move next byte to low order
      sb   $9,2($8)       # bits 8-15
      srl  $9,$9,8        # move next byte to low order
      sb   $9,1($8)       # bits 16-23
      srl  $9,$9,8        # move next byte to low order
      sb   $9,0($8)       # most significant byte

      .data
tape:                     # base register points here
```

```
        .space 1024        # tape buffer (1K bytes)

    ## End of file
```

The .space directive reserves bytes in memory, in this case 1024_{10} bytes. Pretend this is the buffer from which a tape record will be written. The example program uses just the first four bytes.

QUESTION 5:

What is the *symbolic address* of the first byte of the .data section? What main storage address will it have at run time?

Next

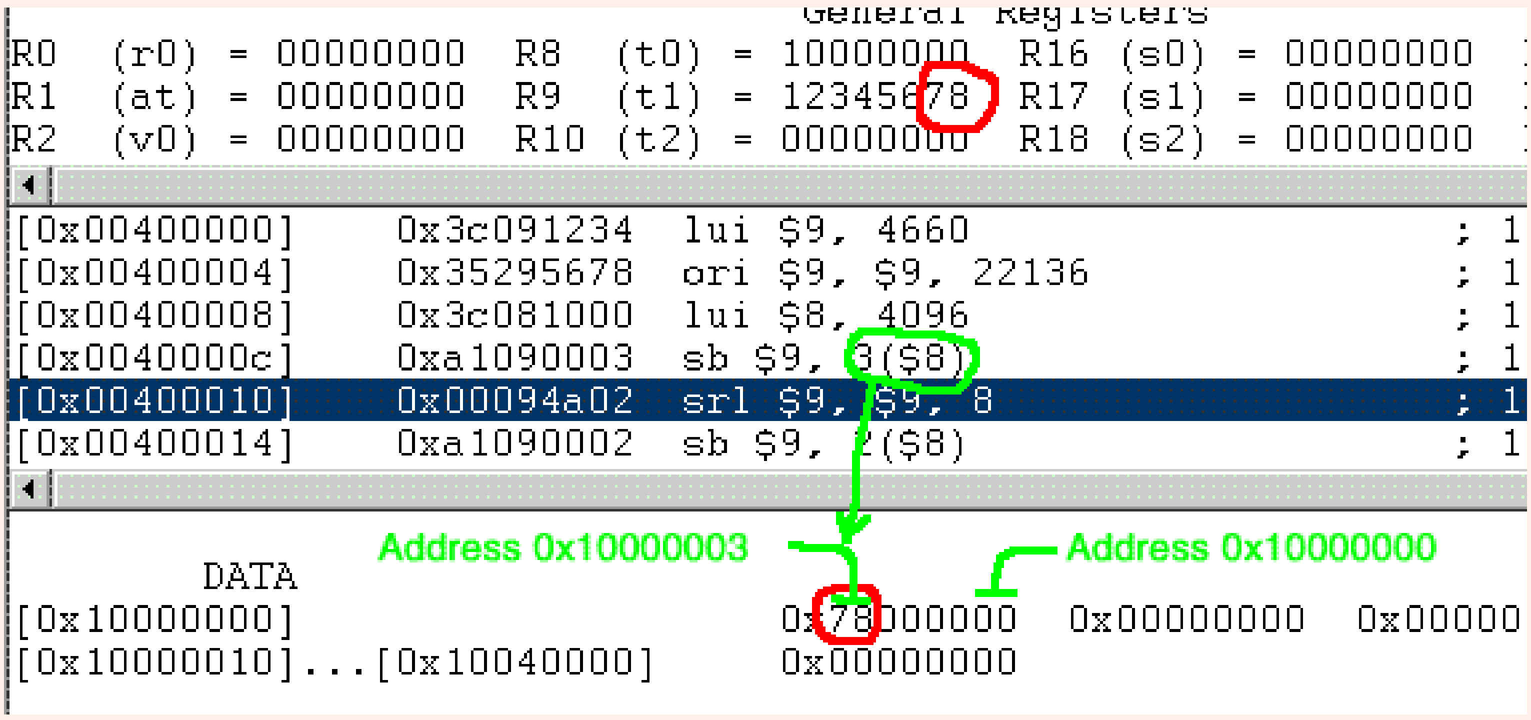
A good answer might be:

What is the *symbolic address* of the first byte of the .data section? tape What main storage address will it have at run time? 0x10000000

The main storage address for the first byte of the data section is 0x10000000 by default (of the SPIM assembler). There is nothing in the program that says this.

A Run of the Program

The SPIM display shows data in groups of 4-byte words with the most significant byte on the left. This makes the data readable by humans. Within each group of four, the byte with the lowest address is on the right. Here is how SPIM looks after the first byte of the register has been stored to the buffer:



```
                                 General Registers
R0   (r0) = 00000000  R8   (t0) = 10000000  R16 (s0) = 00000000
R1   (at) = 00000000  R9   (t1) = 12345678  R17 (s1) = 00000000
R2   (v0) = 00000000  R10  (t2) = 00000000  R18 (s2) = 00000000

[0x00400000]    0x3c091234  lui $9, 4660                  ; 1
[0x00400004]    0x35295678  ori $9, $9, 22136             ; 1
[0x00400008]    0x3c081000  lui $8, 4096                  ; 1
[0x0040000c]    0xa1090003  sb $9, 3($8)                  ; 1
[0x00400010]    0x00094a02  srl $9, $9, 8                 ; 1
[0x00400014]    0xa1090002  sb $9, 2($8)                  ; 1

        DATA
[0x10000000]                     0x78000000  0x00000000  0x00000
[0x10000010]...[0x10040000]      0x00000000
```

The byte is where it should be in memory, although you have to look at the display carefully to see this.

QUESTION 6:

Which byte of $9 should go into address $0x10000000?

Which byte of $9 should go into address $0x10000000?

A good answer might be:

0x12 --- the "big end" goes into the first address for big-endian byte order.

Eventually, after a few more right shifts, it gets there.

Loading Halfwords

A MIPS **halfword** is two bytes. This, also, is a frequently used length of data. An ANSI C short integer is usually two bytes. So, MIPS has load halfword and store halfword instructions. There are two load halfword instructions. One extends the sign bit of the halfword in memory into the upper two bytes of the register. The other extends with zeros.

```
lh   t,off(b)   # $t <-- Sign-extended halfword
                # starting at memory address b+off.
                # b is a base register.
                # off is 16-bit two's complement.

lhu  t,off(b)   # $t <-- zero-extended halfword
                # starting at memory address b+off.
                # b is a base register.
                # off is 16-bit two's complement.
```

Halfword addresses must be **halfword aligned**. Attempting to load a halfword from an unaligned address will cause a trap.

QUESTION 7:

How can you tell if an address is halfword aligned?

Next

A good answer might be:

It is divisible by two. Addresses divisible by two have a zero in the low-order bit.

Storing Halfwords

Only one store halfword instruction is needed. The low-order two bytes of the designated register are copied to memory, no matter what the upper two bytes are. Of course, the register in not changed when its data is copied to memory.

```
sh    t,off(b)    # Halfword at off+b <-- low-order
                  # two bytes from $t.
                  # b is a base register.
                  # off is 16-bit two's complement.
```

MIPS instructions behave the same no matter how data has been loaded into a register. For example, an addu instruction does a full 32-bit addition even if one of the operand registers was loaded with lh (or lb).

QUESTION 8:

Perform these two addition problems:

```
1010 1110        0000 0000 0000 0000 0000 0000 1010 1110
1100 0110        0000 0000 0000 0000 0000 0000 1100 0110
---------      ---------------------------------------
```

Next

A good answer might be:

```
1   1 11                          1    1 11
 1010 1110      0000 0000 0000 0000 0000 0000 1010 1110
 1100 0110      0000 0000 0000 0000 0000 0000 1100 0110
 ---------   --------------------------------------
 1111 0100      0000 0000 0000 0000 0000 0001 1111 0100
```

All Arithmetic is 32-bits

MIPS has no instructions for byte arithmetic nor for halfword arithmetic. For a single operation (such as above), the results in the low-order bytes are the same. The carry out of the high-order bit (of the byte or halfword) is part of the 32-bit result. Further operations may increasingly involve high-order bits. The result after several operations will be a correct result, but may not fit into the original size of the operands.

QUESTION 9:

Now add a third operand to the previous sum:

Next

```
 1111 0100      0000 0000 0000 0000 0000 0001 1111 0100
 1010 1100      0000 0000 0000 0000 0000 0000 1010 1100
 ---------   --------------------------------------
```

A good answer might be:

```
11111 1                         11 1111 1
 1111 0100      0000 0000 0000 0000 0000 0001 1111 0100
 1010 1100      0000 0000 0000 0000 0000 0000 1010 1100
 ---------    --------------------------------------
 1010 0000      0000 0000 0000 0000 0000 0010 1010 0000
```

It would not be correct to store the low order byte of the above result and claim that it is the sum of the three operands.

Low-order Result not always Correct

A further problem is that in general the result in the low order byte is not the same as would result from several true 8-bit arithmetic operations. For example, divide the above results by four (by shifting right twice). The low-order bytes are different.

These are problems that compiler writers face. For example, ANSI C short int variables should behave the same way on all computers. But 16-bit math does not behave the same way on all computer architectures! To make MIPS follow the standard, several extra machine operations must be inserted between each arithmetic operation. On MIPS (and other computers) 16-bit arithmetic is much slower than 32-bit arithmetic.

Naive programmers sometimes use short ints with the expectation that their program will run faster. Depending on the hardware and the compiler, the opposite might be true!

QUESTION 10:

Crytography programs often treat characters as 8-bit integers and transform them with arithmetic operations. Suppose a cryptography program is written in C for a Windows system. When compiled on a Macintosh system it runs, but produces different results! You have been given the job of making the Mac version work identically to the Windows version. What must you do?

Next

A good answer might be:

Probably the problem is with differences in small integer arithmetic between the two systems. You will have to carefully look at variable declarations in the program and will have to study the arithmetic. To make the Mac version identical to the Windows version, you may have to write parts of it in assembly.

End of the Chapter

You have reached the end of this chapter. Click on a blue subject to go to where it was discussed. To get back here, click on the "back arrow" button of your browser.

- Loading a byte into a register.
- Storing a byte from a register.
- The .space assembler directive.
- How SPIM displays the data section.
- Loading a halfword into a register.
- Storing a halfword from a register.
- Byte and halfword arithmetic.

contents Go to Table of Contents

You have reached the end of the chapter.

created: 07/18/01; small edits: 10/20/01

CHAPTER 17 — Jump and Branch Instructions

Much of the power of computers (and all machines) is their ability to repeat actions and their ability to alter their operation depending on data. Modern programming languages express these abilities using syntactic structures. Repeated action (iteration) is done with a *while* structure. Altered operation (alteration) is done with an *if-then-else* structure.

Assembly language does not have these control structures. It must build these structures out of machine-level instructions. *Conditional branch instructions* and *jump* instructions are used to implement both iteration and branching.

Chapter Topics:

- *Jump Instruction:*
  - j instruction (jump)
- *Conditional Branch Instructions (two register):*
  - beq instruction (branch equal)
  - bne instruction (branch not equal)

QUESTION 1:

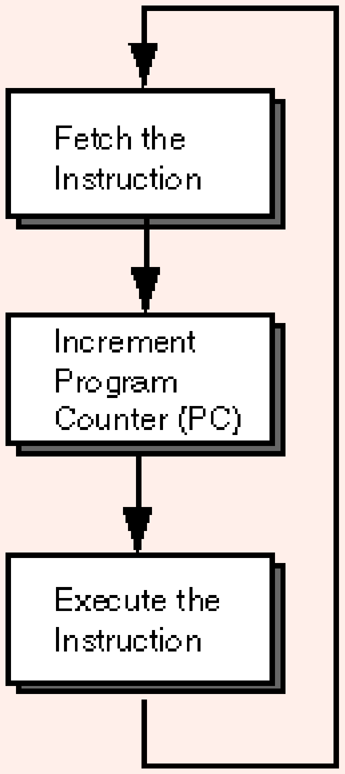
Do machine instructions in MIPS memory have addresses?

Next

A good answer might be:

Yes. All bytes of main memory have addresses.

Review of the Machine Cycle



Each machine cycle executes one machine instruction. At the top of the machine cycle, the **PC** (program counter) contains the address of an instruction to fetch from memory. The instruction is fetched into the processor and is prepared for execution. In the middle of the machine cycle the **PC** is incremented by four to point to the following instruction. Then the current instruction is executed and the cycle repeats. The cycle executes instructions in sequence.

When a **jump** instruction is executed it puts a new address into the PC. Now the instruction that is fetched at the top of the next machine cycle is the instruction at the new address. Instead of executing the next instruction in sequence, the processor "jumps" to an instruction somewhere else in memory. Execution now continues sequentially from the new address.

However, it takes some time for the PC to change. Before it changes, the next sequential instruction has already been fetched and is executed. The instruction that follows a jump instruction in memory (or that follows a **branch** instruction) is said to be in the **branch delay** slot.

The reason for this delay is that MIPS is **pipelined.** The processor holds a queue of instructions being prepared for execution (a "pipe"). The instruction after a jump instruction is already in the pipe when the jump is being executed. (The SPIM simulator allows you to turn this feature off, but this is not an option with actual R2000 hardware).

The instruction that follows the jump instruction in memory (in the branch delay slot) will be executed. The instruction that executes after it is the instruction at the new address. Often the branch delay slot is filled with a no-op instruction.

QUESTION 2:

(Review:) What does a no-op instruction do?

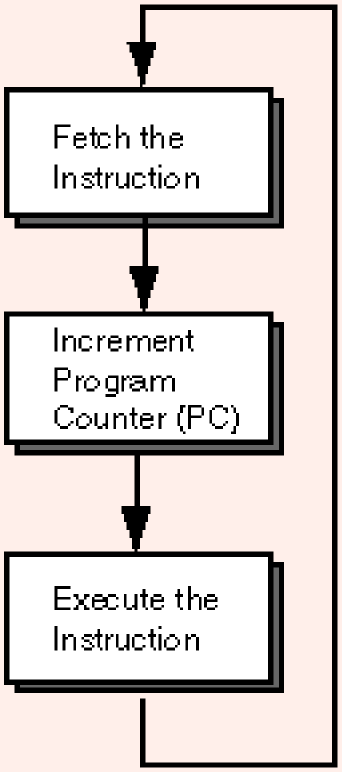
A good answer might be:

A no-op instruction is any instruction that has no effect. A common no-op instruction is sll $0,$0,0.

Altering the PC

Here is a sequence of instructions shown in outline. Once started, the four instructions execute in an unending loop. The last instruction, a no-op, fills the branch delay slot to give the PC time to change.

Address	Instruction	PC after executing this instruction
....	....	00400000
00400000	load	00400004
00400004	add	00400008
00400008	jump 0x400000	0040000C
0040000C	no-op	00400000



Inspect the sequence of addresses in the PC. A loop structure has been created with the jump instruction. Notice that the effect of the jump instruction (loading 0x400000 into the PC) is not seen until after the branch delay slot.

QUESTION 3:

(Review:) How many bits are in a MIPS memory address?

Next

A good answer might be:

32. The same size as a MIPS instruction.

The Jump Instruction

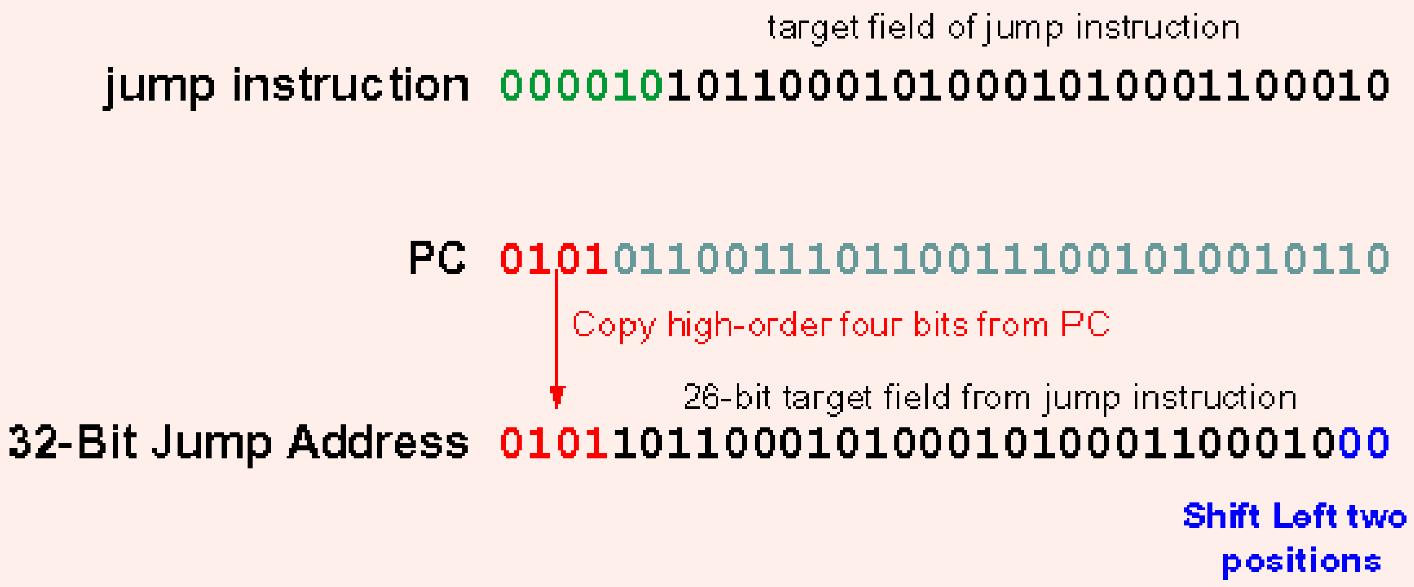
How does a 32-bit instruction specify a 32-bit address? Some of the instruction's bits must be used for the op-code. Here is the assembly language version of the jump instruction.

```
j    target     # after a delay of one machine cycle,
                # PC  <-- address of target
```

Here is the machine language form of the instruction:

```
  6         26
000010 00000000000000000000000000    -- fields of the instructuion

opcode      target             -- meaning of the fields
```

There is room in the instruction for a 26-bit address. The 26-bit target address field is transformed into a 32-bit address. Instructions always start on an address that is a multiple of four (they are word-aligned). So the low order two bits of a 32-bit instruction address are always "00". Shifting the 26-bit target left two places results in a 28-bit word-aligned address. Now four bits are concatenated to the high-order end of the 28-bit address to form a 32-bit address. These four bits are copied from the top four bits of the PC.


target field of jump instruction
jump instruction 00001010110001010001010001100010
PC 01010110011101100111001010010110
Copy high-order four bits from PC
26-bit target field from jump instruction
32-Bit Jump Address 01011011000101000101000110001000
Shift Left two positions

QUESTION 4:

While this is going on, what instruction's address is in the PC?

Next

A good answer might be:

The PC contains the address that follows the address of the jump instruction.

Most Jumps (and Branches) are Local

Most jumps and branches are to a nearby address. The jump address and the address of the instruction following the jump instruction are likely to have the same high-order four bits. Of course, a compiler or assembly language programmer must be careful to ensure that this is so.

A jump instruction can't jump to anywhere in the full 32-bit address space, but must jump to somewhere within the following range of addresses:

```
wxyz 0000 0000 0000 0000 0000 0000 0000
...
wxyz 1111 1111 1111 1111 1111 1111 1100
```

Here, **wxyz** represents the high-order four bits of the PC. Almost always the jump instruction and the jump address are both within this range.

All these details may look terrible to you at this point. Don't worry: (1) its not as bad as it looks, and (2) usually the assembler does all the work. (But for now, *you* get to do the work).

QUESTION 5:

Here is a great idea! Why not implement the jump instruction without using an op-code? Build the processor so that when execution encounters a 32-bit address it automatically jumps to that address.

Will this scheme work?

A good answer might be:

No. Addresses are 32-bit patterns, machine instructions are 32-bit patterns, and many data are 32-bit patterns. There is no way to tell them apart. Here is an example. Look at this bit pattern: 0x00000000 Is this the address of the first byte of memory, or the sll $0,$0,0 instruction?

Jump Practice

The following program illustrates the jump instruction. For simplicity, all instructions other than the jump instruction are no-ops. The jump instruction jumps to the first instruction of the program. The very last instruction fills the delay slot.

Address	Machine Instruction	Assembly Instruction
00400000	0000 0000 0000 0000 0000 0000 0000 0000	sll $0,$0,0
00400004	0000 0000 0000 0000 0000 0000 0000 0000	sll $0,$0,0
00400008	0000 0000 0000 0000 0000 0000 0000 0000	sll $0,$0,0
0040000C	0000 0000 0000 0000 0000 0000 0000 0000	sll $0,$0,0
00400010	000010 00 0001 0000 0000 0000 0000 0000	j firstInstruction
00400014	0000 0000 0000 0000 0000 0000 0000 0000	sll $0,$0,0

The left-most six bits of the j machine instruction are the opcode. You need to decide if the next 26 bits are correct.

QUESTION 6:

1. Write the full 32-bit jump address: 0x00400000 ______________________________
2. Write the 26-bit field of the jump instruction: ______________________________

Next

3.	Shift it left two positions:	________________________________
4.	What are the high-order four bits of the PC?	____
5.	Copy (4) to the left of (3):	____ ________________________________
6.	Is (5) the same as (1)?	_____

A good answer might be:

1.	Write the full 32-bit jump address: 0x00400000	0000 0000 0100 0000 0000 0000 0000 0000
2.	Write the 26-bit field of the jump instruction:	00 0001 0000 0000 0000 0000 0000
3.	Shift it left two positions:	0000 0100 0000 0000 0000 0000 0000
4.	What are the high-order four bits of the PC?	0000
5.	Copy (4) to the left of (3):	0000 0000 0100 0000 0000 0000 0000 0000
6.	Is (5) the same as (1)?	Yes

Symbolic Address

With some trickery, a 26-bit field can specify a 32-bit address. But it is a nuisance to figure out! If you were doing machine language programming, that is what you would have to do. But the assembler does the work for you. Here is a tiny program:

```
## jump.asm
##

        .text
        .globl  main

main:
        sll    $0,$0,0
        sll    $0,$0,0
        sll    $0,$0,0
        sll    $0,$0,0
        j      main
        addiu $8,$8,1
## End of file
```

It is similar to the previous example. The symbolic address main stands for the address of the first instruction. The instruction `j main` tells the assembler to assemble a machine instruction with the proper 26-bit field.

The branch delay slot is filled with an instruction that increments register $8. SPIM initializes all registers to zero when it starts.

QUESTION 7:

After the loop has executed five times, what value will be in register $8?

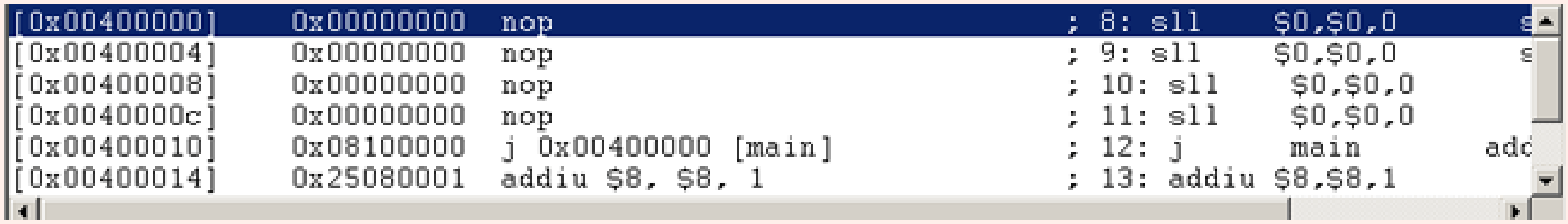
Next

A good answer might be:

Five. Each time the loop executes, the instruction in the branch delay slot increments $8.

Assembled Program

Here is a SPIM view of the program. When you run it, remember to set the value of the PC to 0x400000. To see the branch delay it should be enabled in the options menu.



```
[0x00400000]    0x00000000  nop                        ; 8: sll     $0,$0,0         s
[0x00400004]    0x00000000  nop                        ; 9: sll     $0,$0,0         s
[0x00400008]    0x00000000  nop                        ; 10: sll     $0,$0,0
[0x0040000c]    0x00000000  nop                        ; 11: sll     $0,$0,0
[0x00400010]    0x08100000  j 0x00400000 [main]        ; 12: j       main          add
[0x00400014]    0x25080001  addiu $8, $8, 1            ; 13: addiu $8,$8,1
```

The assembler constructed the same machine language jump instruction as we did by hand. (Compare with two pages back). Using a symbolic address is certainly easier.

QUESTION 8:

Is the jump instruction what we need to construct a **`while`** loop?

Next

A good answer might be:

No. You can build unending loops (infinite loops) with it. Additional instructions are needed for while loops and for if-then-else structures.

Conditional Branches

A **conditional branch** instruction branches to a new address only if a certain condition is true. Usually the condition is about the values in registers. Here is the beq (branch on equal) instruction:

```
beq  u,v,addr   # if register $u == register $v
                #     PC  <-- addr
                #     after a delay of one machine cycle.
                # else
                #     no effect.
```

The bit patterns in two registers are compared. If the bit patterns are the same, the PC is changed to the branch address. There is a branch delay following the instruction (just as for a jump instruction).

QUESTION 9:

Will a 32-bit address fit inside the 32-bit beq instruction?

Next

A good answer might be:

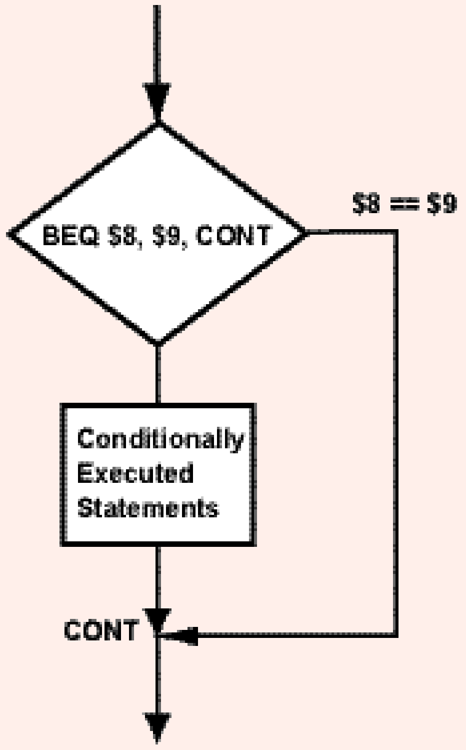
No. Same problem as with the j instruction.

Ifs and Whiles

More trickery is used to create a 32-bit branch address out of the smaller sized field of the machine instruction. But let's skip that for now.

Branch instructions (including the beq instruction) are used to implement both loops and branches. At right is a flowchart showing an optional branch. Here is the assembly for it:

```
        ...             # load values into $8 and $9
        beq  $8,$9,cont    # branch if equal
        nop                # branch delay slot
        ...             # conditionally
        ...             # executed
        ...             # statements
     cont:   add   $10,$10,$11  # always executed
```



QUESTION 10:

Must the contents of the registers in this example be numeric?

Next

A good answer might be:

No. beq tests if the same 32-bit pattern is in each register. The pattern can represent anything.

Branch on Not Equal

Here is the bne (branch on not equal) instruction:

```
bne  u,v,addr   # if register $u =!= register $v
                #     PC  <-- addr
                #     after a delay of one machine cycle.
                # else
                #     no effect.
```

QUESTION 11:

Can a branch instruction implement a two-way decision (an *if-then-else*)?

Next

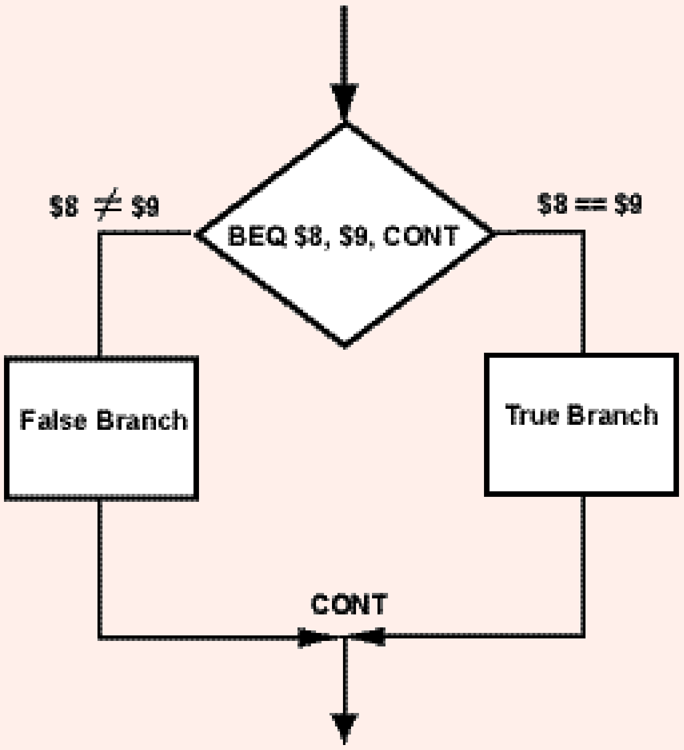
A good answer might be:

Yes.

Two-way Decision

A two-way decision (alternation) is written in assembly language using both a conditional branch and a jump instruction:

```
        ...            # load values into
                       # $8 and $9
        beq  $8,$9,equal  # branch if equal
        nop               # branch delay slot
        ...            #
        ...            #  false branch
        ...            #
        j    cont
        nop
equal:  ...              #
        ...            #  true branch
        ...            #
cont:   add   $10,$10,$11  # always executed
```



Of course, any of the conditional branch instructions may be used. If you want the "true" branch to come first and the "false" branch to come second (as in an *if-then-else* of Java or C) you may have to pick a different branch instruction.

QUESTION 12:

In an *if-then-else* structure the two branches of control always come together at the first statement outside of the structure. Is this necessarily so in assembly language?

Next

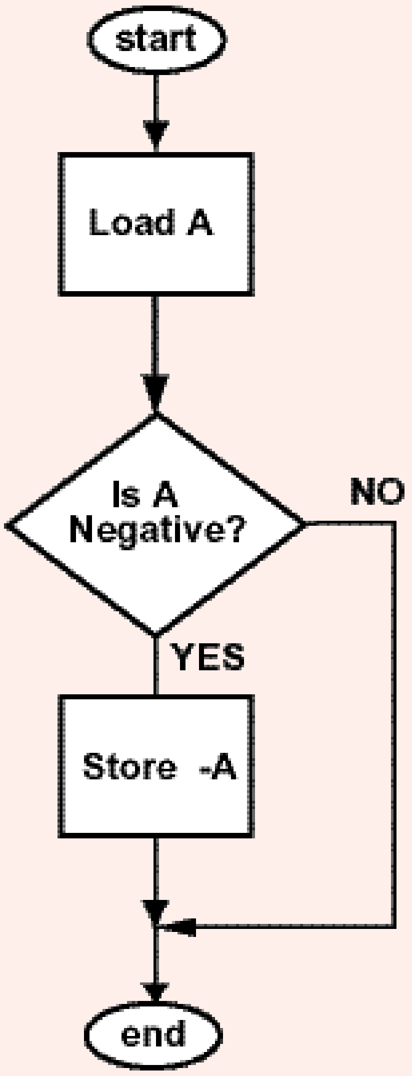
A good answer might be:

No.

Absolute Value

You can build a real rats-nest of code with assembly language. Avoid this by imitating the structures of a high level language. Draw a flowchart of the program or to rough it out in C or Java before coding. Put comments in the assembly source program before you add code to show how the code and the flowchart correspond.

The flowchart at right shows a program to calculate the absolute value of the integer at symbolic address "A". Here is a start on a program that follows that logic:



```
## absVal.asm
##
## Calculate the absolute value of A

      .text
      .globl  main

main:
# Get A
        lui   $10,0x1000     #  Init base register
        lw    $8,___($10)    #  Load A
        ___   $0,$0,0        #  no-op

# Is A Negative?


# Store -A


done:

      .data
A:    .word   -1

## End of file
```

Assume that "A" will be the word starting at address 0x10000000. The `lui` instruction points the base register $10 at that address.

QUESTION 13:

Fill in the blanks.

Next

A good answer might be:

The program below has the blanks filled.

Shifting the Sign Bit

To determine if "A" is negative, check if its sign bit is one. To do this, logically shift the sign bit into bit position 0 of a register. The register will be zero if "A" is positive.

```
## absVal.asm
##
## Calculate the absolute value of A

      .text
      .globl  main

main:
# Get A
        lui   $10,0x1000     #  Init base register
        lw    $8,0($10)      #  Load A
        sll   $0,$0,0        #  no-op

# Is A negative?
        srl   $9,$8,_____       #  Shift sign bit to position 0
        beq   $___,$_____,done  #  sign bit == zero, done
        sll   $0,$0,0


# Store -A


done:

      .data
A:    .word   -1

## End of file
```

The branch delay slot is filled with a no-op.

QUESTION 14:

Fill the blanks.

A good answer might be:

The blanks are filled, below.

Store -A

The sign bit is shifted right 31 bit positions. This puts it in the low-order bit of the destination register ($9 in this case). To test if $9 is zero, use branch-on-equal with register $0 (which is always zero).

```
## absVal.asm
##
## Calculate the absolute value of A

      .text
      .globl  main

main:
# Get A
        lui   $10,0x1000     #  Init base register
        lw    $8,0($10)      #  Load A
        sll   $0,$0,0        #  no-op

# Is A negative?
        srl   $9,$8,31       #  Shift sign bit to position 0
        beq   $0,$9,done     #  sign bit == zero, done
        sll   $0,$0,0

# Store -A
        sub   $8,$___,$8     #  negate A
        sw    $8,___($___)   #  save it

done:   sll   $0,$0,0

      .data
A:    .word   -1

## End of file
```

Now calculate -A and store it back into word "A". The instruction at done is a no-op.

QUESTION 15:

Fill the blanks.

Next

A good answer might be:

The complete program is given below

Complete Program

Here is the complete program, suitable for copying to a text file and running with SPIM.

```
## absVal.asm
##
## Calculate the absolute value of A
##
## Registers:
##   $8  --- A, two's comp. integer
##   $9  --- sign bit of A
##   $10 --- base register for .data section

        .text
        .globl  main

main:
# Get A
        lui   $10,0x1000     #  Init base register
        lw    $8,0($10)      #  Load A
        sll   $0,$0,0

# Is A Negative?
        srl   $9,$8,31       #  Shift sign bit to position 0
        beq   $0,$9,done     #  sign bit == zero, done
        sll   $0,$0,0

# Store -A
        sub   $8,$0,$8       #  negate A
        sw    $8,0($10)      #  save it

done:   sll   $0,$0,0        #  target of the branch

        .data
A:      .word   -1

## End of File
```

QUESTION 16:

Would the program work if "A" were unsigned binary?

A good answer might be:

No.

End of the Chapter

You have reached the end of this chapter. Click on a blue subject to go to where it was discussed, unless you're absolutely sure you know it. To get back here, click on the "back arrow" button of your browser.

- The branch delay slot of MIPS.
- How changing the PC implements a branch.
- The jump instruction.
- The range of addresses to which a jump can go.
- Symbolic addresses.
- The branch on equal instruction.
- The branch on not equal instruction.

You have reached the end of the chapter.

created: 07/21/01; corrections: 08/22/01

CHAPTER 18 — Set Instructions

This chapter describes two additional branch instructions and two conditional **set** instructions. Set instructions set a register to 1 or 0 depending on whether a condition is true or false.

Chapter Topics:

- *More Conditional Branch Instructions (one operand register):*
  - bltz instruction (branch on less than zero)
  - bgez instruction (branch on greater or equal to zero)
- *Conditional Set Instructions*
  - slt instruction (set on less than)
  - sltu instruction (set on less than unsigned)

These new instructions add convenience in implementing loops and branches within programs.

QUESTION 1:

Are relational operators like >, <, >=, and <= useful in a programming language?

Next

A good answer might be:

Of course.

Branch on Less than Zero, Branch on Greater than Zero

MIPS has several ways to implement relational operators. Here are two more branch instructions. These instructions compare the contents of a register to zero. The register's contents is assumed to represent an integer with two's complement.

```
bltz   s,label       #  Branch if the two's comp. integer
                     #  in register s is < 0
                     #  There is a one instruction delay slot.

bgez   s,label       #  Branch if the two's comp. integer
                     #  in register s is >= 0
                     #  There is a one instruction delay slot.
```

The first instruction branches if the integer is strictly less than zero. The other branches if the integer is greater than or equal to zero.

Both of these instructions are followed by a branch delay slot. This means that the instruction in that slot will always execute, and the branch (if it happens) will not happen until after that instruction executes.

QUESTION 2:

Rewrite the following C (or Java) if-statement without using the not (the !).

```
if ( !( a < 0 ) ) { ... }

if ( a _____ 0  ) { ... }
```

Next

A good answer might be:

```
if ( !( a < 0 ) ) { ... }

if ( a >= 0  ) { ... }
```

The two operators < and >= cover all possible integer values.

Set on Less Than

The **set** instructions are used to implement relational operators. However, they do not in themselves alter the flow of control. They set a register to 1 or 0 to show the relation between two values. The slt instruction is used with two's complement integers:

```
                # $s and $t contain
                # two's comp. integers
                #
slt   d,s,t     # if ( $s < $t )
                #    d <-- 1
                # else
                #    d <-- 0
```

The sltu instruction is used with unsigned integers:

```
                # $s and $t contain
                # unsigned integers
                #
sltu  d,s,t     # if ( $s < $t )
                #    d <-- 1
                # else
                #    d <-- 0
```

QUESTION 3:

Could these instructions be used with character data?

Next

A good answer might be:

Yes. If the characters to be compared are loaded in the low-order byte of two registers, and the rest of the bits are zero, either instruction could be used.

Set on Less Than Immediate

The other two set instructions compare an operand register with an immediate value in the instruction. There is a version for two's complement:

```
                    #  $s and imm contain
                    #  two's comp. integers
                    #
slti  d,s,imm       #  if ( $s < imm )
                    #    d <-- 1
                    #  else
                    #    d <-- 0
```

And a version for unsigned integers:

```
                    #  $s and imm contain
                    #  unsigned integers
                    #
sltiu  d,s,imm      #  if ( $s < imm )
                    #    d <-- 1
                    #  else
                    #    d <-- 0
```

In both, the immediate field of the machine instruction is 16 bits wide.

QUESTION 4:

How is the 16-bit immediate field extended to 32 bits for the comparison? By sign extention or by zero extension?

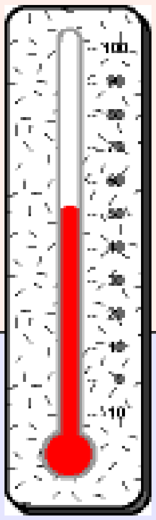
- slti:
- sltiu:

Next

A good answer might be:

- slti: by sign extension.
- sltiu: by zero extension.

Temperature Range Tester



Say that you are writing a control program for a robot spray painter. The allowed temperature range for the paint is 30 degrees to 55 degrees Celsius. The driver program for the temperature sensor puts the temperature in register $2.

Your program will test if the unsigned integer in register $2 is in range. If in range, register $3 is set to 1, otherwise to 0.

QUESTION 5:

Sketch a flow chart for this program.

Next

A good answer might be:

With assembly language it is essential to make a plan before coding.

Start on the Program

The flowchart for the program is at right. The trick of setting a flag to a default value in advance of the test is common. Here is an outline of the program:

```
## tempRange.asm
##
## Check that   30 <= temp <= 55
## Set flag to 1 if in range, to 0 if out of range
##
## Registers:
## $2 --- temperature
## $3 --- in/out range indicator flag
## $8 --- scratch

      .text
      .globl  main
# Set range indicator to 1
main:   ori     $3,$0,1         #  set to 1

# Test  30 <= temp <= 55
        sltiu   $8,$2,_____     #  $8=1 if temp <= 55
        beq     $8,$_____,_____ #  0? out of range
        sll     $0,$0,0         #  delay

       . . . .                  #  more instructions

# Out of Range: set range indicator to 0
out:
       . . . .                  #  set to 0

cont:  sll     $0,$0,0          #  target for the jump

## End of file
```

The range test is in two parts. The first part (in this program) tests if temp is less than <u>or equal</u> to 55. However, the machine instruction is "set on less than". If temp is out of range a branch is taken to out. The branch is followed by a no-op for the branch delay.

QUESTION 6:

Fill in the blanks.

A good answer might be:

See Below

More Blanks

The immediate operand used in the set instruction is changed to 56 to implement "less than or equal". Notice that the assembly language uses decimal numbers for temperatures. This is fine. The assembler translates the decimal representation of the source file into the correct bit pattern for the machine instruction.

The next part of the program tests if temp is less than 30. Be careful with the branch instruction so that it branches for the correct condition.

```
## tempRange.asm
##
## Check that   30 <= temp <= 55
## Set flag to 1 if in range, to 0 if out of range
##
## Registers:
## $2 --- temperature
## $3 --- in/out range indicator flag
## $8 --- scratch

      .text
      .globl  main
# Set range indicator to 1
main:   ori     $3,$0,1         #  set to 1

# Test  30 <= temp <= 55
        sltiu   $8,$2,56        #  $8=1 if temp <= 55
        beq     $8,$0,out       #  0? out of range
        sll     $0,$0,0         #  delay

        sltiu   $8,$2,_____     #  $8=1 if  temp < 30
        beq     $8,$0,_____     #  0? in range
        sll     $0,$0,0         #  delay

# Out of Range: set range indicator to 0
out:
        ori     $3,____,____    #  set to 0
```

```
cont:   sll     $0,$0,0         #  target for the jump

## End of file
```

QUESTION 7:

Fill in the blanks.

Next

A good answer might be:

The completed program follows.

Complete Program

Here is the complete program, suitable to copy to a file and to run with SPIM. When you run it, set the PC to 0x400000 (as usual) and also use the set value menu to set R2 to a temperature. Run the program with different temperatures and check that $3 is set correctly.

```
## tempRange.asm
##
## Check that   30 <= temp <= 55
## Set flag to 1 if in range, to 0 if out of range
##
## Registers:
## $2 --- temperature
## $3 --- in/out range indicator flag
## $8 --- scratch

      .text
      .globl  main

# Set range indicator to 1
main:   ori     $3,$0,1         #  set to 1

# Test  30 <= temp <= 55
        sltiu   $8,$2,56        #  $8=1 if temp <= 55
        beq     $8,$0,out       #  0? out of range
        sll     $0,$0,0         #  delay

        sltiu   $8,$2,30        #  $8=1 if  temp < 30
        beq     $8,$0,cont      #  0? in range
        sll     $0,$0,0         #  delay

# Out of Range: set range indicator to 0
out:    ori     $3,$0,0         #  clear to 0

cont:   sll     $0,$0,0         #  target for the jump
## End of file
```

QUESTION 8:

Could the no-op instructions (the `sll`) be removed without affecting the program?

Next

A good answer might be:

No.

Delay Slot Bug

The program can be made slightly shorter by removing the no-op instruction filling the first delay slot. The instruction following it (the sltiu) will always execute, sometimes uselessly, but never will do damage.

```
# Set range indicator to 1
        ori     $3,$0,1         #  set to 1

# Test  30 <= temp <= 55
        sltiu   $8,$2,56        #  $8=1 if temp <= 55
        beq     $8,$0,out       #  0? out of range
        sll     $0,$0,0         #  delay

        sltiu   $8,$2,30        #  $8=1 if  temp < 30
        beq     $8,$0,cont      #  0? in range
        sll     $0,$0,0         #  delay

# Out of Range: set range indicator to 0
out:
        ori     $3,$0,0         #  clear to 0

cont:   sll     $0,$0,0         #  target for the jump
## End of file
```

The second no-op, however, is essential. If it is missing, the next instruction, the ori sets the flag to zero regardless of the branch instruction. This is a common bug, and can be very frustrating because *sometimes* the result is correct.

QUESTION 9:

(Review:) What other type of instruction is followed by a delay slot?

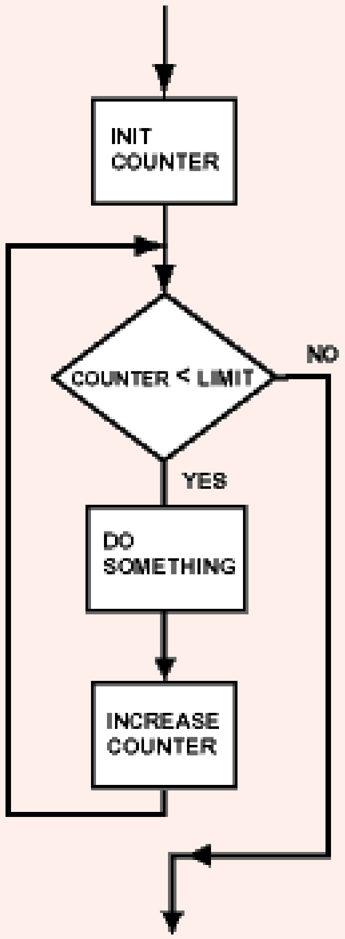
Next

A good answer might be:

Load instructions.

Counting Loop

A common program loop is controlled by an integer that counts up from a initial value to an upper limit. Such a loop is called a **counting loop**. The integer is called a *loop control variable*. Loops are implemented with the conditional branch, jump, and conditional set instructions.



A loop has three parts that must be correct:

1. The counter must be initialized.
2. The test must end the loop on the correct count.
3. The counter must be increased.

It is easy to get these wrong in a high-level programming language. It is remarkably easy to get them wrong in assembly language.

Usually you want a **top-driven** loop such as the one at right, where the test is performed at the top before control enters the loop body. Be clear about the loop you want before you program it, because assembly language allows any sort of weird loop.

QUESTION 10:

Is the following loop (in C) correct in all three parts? It is intended to execute 10 times starting at zero.

Next

```
int j;

j = 0;
while ( j < 10 )
{
        . . .
        j++ ;
    }
```

```
int j;

j = 0;
while ( j < 10 )
{
    . . .
    j++ ;
}
```

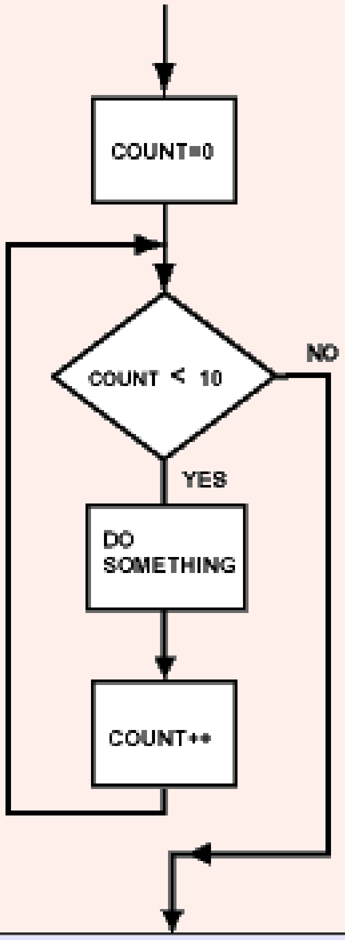
A good answer might be:

The loop is correct. (Although j changes to 10 at the bottom of the last iteration, this is the normal way for loops to work).

Assembly Language Loop

Here is an assembly version of the counting loop, without the branch delay slots filled:

```
#
#  branch delay slots not filled
#
init:   ori    $8,$0,0        # count = 0

test:   sltiu  $9,$8,10       # count < 10
        beq    $9,$0,endLp

        . . .                 # do stuff

        addiu  $8,$8,1        # count++ ;
        j      test

endLp:  sll    $0,$0,0        # branch target
```



QUESTION 11:

Find and fill in the branch delay slots.

Next

A good answer might be:

See Below.

Complete Loop

Here is the loop with the branch delay slots filled. One could be clever and eliminate the last no-op, but let's not.

```
#
#  branch delay slots  filled
#
init:
        ori    $8,$0,0        # count = 0

test:   sltiu  $9,$8,10       # count < 10
        beq    $9,$0,endLp    # end loop if count >= 10
        sll    $0,$0,0        # delay

                              # do stuff

        addiu  $8,$8,1        # count++ ;
        j      test
        sll    $0,$0,0        # delay

endLp:  sll    $0,$0,0        # branch target
```

The no-op at endLp is not filling a branch delay slot. It is there for convenience in running the program with SPIM.

With a few assembly language directives, the code is ready to run. Step through the code and watch $8 (count) increase from 0 to 0xA.

QUESTION 12:

Examine the program. How could you modify it to compute the sum of the integers 0 through 9?

Next

A good answer might be:

See below.

Summing Program

The loop is already correct for the problem. Don't change it. Computing the sum is done by adding just two statements:

```
##
##  Sum of integers 0 .. 9
##
## Registers:
##  $8  --- loop control
##  $9  --- scratch
## $10  --- sum

init:   ori    $10,$0,0       # sum = 0
        ori    $8,$0,0        # count = 0
test:   sltiu  $9,$8,10       # count < 10
        beq    $9,$0,endLp    # end loop if count >= 10
        sll    $0,$0,0        # delay

        addu   $10,$10,$8     # sum += count

        addiu  $8,$8,1        # count++ ;
        j      test
        sll    $0,$0,0        # delay

endLp:  sll    $0,$0,0        # jump target
```

QUESTION 13:

Of all the slls in the program, which one is the most dangerous to remove?

Next

A good answer might be:

The first one.

End of Chapter

You have reached the end of this chapter. Click on a blue subject to go to where it was discussed, To get back here, click on the "back arrow" button of your browser.

You have reached the end of the chapter.

created: 07/27/01

CHAPTER 19 — Structured Programming

This chapter discusses two topics of universal interest in computer science: What machine instructions are needed in a processor, and How to build programs that work. The SPIM instructions you have looked at so far are more than enough for a processor. The *structured programming* approach is used to build reliable programs.

Chapter Topics:

- Computing power.
- The minimum set of machine instructions.
- Throughput
- Complex instructions sets vs. Reduced instruction sets.
- Structured programming.
- The minimum set of control structures.

Much of the chapter is about the following question, which, oddly, often remains unanswered in introductory courses.

QUESTION 1:

Are Java, C, and C++ structured languages?

A good answer might be:

Yes, all modern high level programming languages are structured because they include if-then-else and while structures (and other structures).

Computing Power

So far, these chapters have covered several kinds of instructions:

- Bit-wise logic instructions.
- Integer arithmetic instructions.
- Memory access instructions.
- Instructions that conditionally alter the program flow.

You might wonder how many instructions a processor must have in order to be as powerful as another. The answer is: the above set of instructions is more than enough. But the idea of computer "power" is somewhat vague. Sometimes it means "speed" and sometimes it means "what can be computed" and usually it means a fuzzy combination of both. Let us use the following definition:

> **Computing Power:** Two processors have the same computing power if they can run the same programs (after translation into each processor's machine language) and produce the same results

For example, say that two processors have the same power. Then if one processor can run a particular program, then the other one must be able to run it, and both processors must produce the same result. This must be true for all programs (after appropriate compilation into native machine language).

Sometimes the result a program produces depends on the compiler. For example, different compilers for "C" use different number of bits for the data type *int*. But that is an effect of the compiler, not of the processor. All that matters for processor "power" is that it is possible to translate identical programs into machine language appropriate for each processor and that these machine language programs produce the same result on each processor.

Processor speed is left out of the definition. It is helpful to regard computing power and processor speed as separate aspects. Memory and peripherals (such as graphics boards) are also left out.

QUESTION 2:

Can a program that uses 64-bit integers run on a processor that has 32-bit words?

A good answer might be:

Yes. Integers of any size can be used on a processor no matter what the word size. This is done by using both the integer arithmetic instructions of the processor (if any) and bit manipulation instructions.

Equal Power Processors

For example: 16-bit Intel microprocessors could run programs that used 64-bit integer arithmetic. This was done by compiling 64-bit operations into several 16-bit machine operations.

As a more modern example: Pentium-1 and Pentium-4 processors can run the same programs. One (the P-4) has a faster machine cycle than the other. And one (the P-4) has more machine instructions than the other. If you had a C program that computed something, both processors could run it, and whatever it computed would come out the same on both (assuming appropriate settings of the compilers). The run time would be far longer on the P-1 than on the P-4, but run time is not part of the definition.

QUESTION 3:

In 1952 the SWAC digital computer was programmed to find perfect numbers (integer numbers whose integer divisors sum up to the number). After hours of computation it found the perfect number 2281).

Will a modern Pentium processor find the same perfect numbers as the SWAC?

A good answer might be:

Of course. My 1.7 GHz processor finds 2281 almost immediately.

Minimum Instruction Set

But, in the definition of processor power, my very fast processor is no more powerful than the SWAC.

Processors must control their peripheral devices, and send and receive information from them. They do this by writing and reading bit patterns on the system bus, and by reading and writing pre-assigned memory addresses. So a processor that can read and write memory can also control hard disks and graphics boards. Special "multimedia" instructions do not add extra power. Of course, they do add some convenience.

What machine instructions must a processor absolutely have?

> **Important Fact:** All processors that support the fundamental machine instructions of bit manipulation, conditional branching, and memory access have the same computing power. All processors have these instructions (and more). All processors are equivalent in computing power (in the sense of the previous definition).

Arithmetic (both integer and floating point) can be done with bit manipulation instruction, so arithmetic instructions are not fundamental (but are almost always included in a processor).

Above a certain minimum set of instructions adding new instructions does not add to the computing power of a processor. (To learn more about this topic, take a course in *Foundations of Computation* or in *Mathematical Logic*).

QUESTION 4:

(Thought question:) Why do most processors have many more instructions than the minimum set needed for full computing power?

A good answer might be:

For convenience and throughput.

Convenience and Throughput

Convenience is important. You likely have been frustrated trying to find the right MIPS instructions to use in your programs. Sometimes it feels like fitting square pegs into round holes. How much more convenient to have a wide selection of pegs, even if they don't add computing power!

Convenience is not just or human programmers. Compilers and other systems software create machine programs out of machine instructions. They are easier to write, and less buggy when there is a rich set of instructions.

Throughput is how much computing a processor (or full computer system) can perform in a unit of time. Say that a processor can perform 50 million instructions in one second. The more computing each instruction does, the greater the throughput. So most processors have instructions that do much more than the bare minimum.

QUESTION 5:

Which of the following improvements increase the throughput of a computer system?

- Faster machine cycle (500 MHz to 1000 MHz).
- More bits on the system bus (32 bits to 64 bits).
- More main memory (128 Meg to 512 Meg).
- Larger hard disk (20 Gig to 40 Gig).
- Faster data transfer rate of hard disk (40 MBps to 80 MBps).
- Bigger monitor (17" to 21").
- Many, big, complex machine instructions.

A good answer might be:

- Faster machine cycle (500 MHz to 1000 MHz).
    - **Yes. More instructions done per second.**
- More bits on the system bus (32 bits to 64 bits).
    - **Yes. More data moved per transfer operation.**
- More main memory (128 Meg to 512 Meg).
    - **Yes. Data can be kept in fast main memory rather than slow disk memory.**
- Larger hard disk (20 Gig to 40 Gig).
    - **No, assuming that all data and software fit on the smaller size.**
- Faster data transfer rate of hard disk (40 MBps to 80 MBps).
    - **Yes. Processor spends less time waiting for data.**
- Bigger monitor (17" to 21").
    - **No.**
- Many, complex machine instructions.
    - **Maybe, maybe not.**

CISC vs RISC

A **CISC** (Complex Instruction Set Computer) processor has many instructions, some of them complex instructions that do a lot per machine cycle. The Intel processors are CISC. A **RISC** (Reduced Instruction Set Computer) processor has few instructions, most of them simple. It may take several instructions to do the equivalent of a CISC instruction. The MIPS processors are RISC.

If everything else were the same, CISC would have greater throughput. A larger silicon chip is needed for the many complex instructions. Data must be moved through greater distances, and that takes more time. This slows down the clock cycle. The instructions themselves take more time to execute (usually more than several RISC instructions). Fine tuning the chip for speed is difficult when there are many instruction types. The simple instructions are compromised by the complex ones. But the simple instructions are the most frequently executed!

Worse, it is hard for a compiler to make use of complex instructions. They frequently compile programs into machine programs that use only simple instructions. The MMX (multi-media extension) instructions added to recent Pentium chips are not used by any compiler.

The idea of RISC is that it is better to have a small, sleek, fast instruction set than to have a big collection of poorly coordinated, ungainly, complex instructions.

For the last decade or so RISC chips have been ahead of CISC chips. But a modern CISC chip has greater throughput than an older RISC chip. Consumer and office computers use CISC chips to be compatible with existing Windows software. High-end workstations and recently designed systems (such as embedded systems) typically use RISC.

QUESTION 6:

Recall the fable of *The Tortise and the Hare*. Which processor chip is the tortise? Which is the hare?

A good answer might be:

Which processor chip is the tortise? **CISC**. Which is the hare? **RISC**. Right now the hare is ahead.

Bug-free Software

Examine (for a moment) the following program excerpt. (Trick question:) Is it correct?

```
start:   ori   $8,$0,4      # $8 = 4
         ori   $9,$0,12     # $9 = 12
         addu  $10,$8,$9    # $10 = 12+4 = 16
         sll   $10,$10,2    # $10 = 16*4 = 64
```

Ordinarily, you would say "yes", assuming that the comments are correct. But this assumes that execution starts at start. What if execution started at the addu instruction? Registers $8 and $9 would probably contain different numbers. That could happen if the following were somewhere in the program:

```
         ori   $8,$0,99     # $8 = 99
         ori   $9,$0,43     # $9 = 43
         j     start+8      # jump to the second statement after start
```

start is a symbolic address that stands for the first instruction's run time address. start+8 stands for the address 8 bytes away. The jump instruction transfers control to that address.

QUESTION 7:

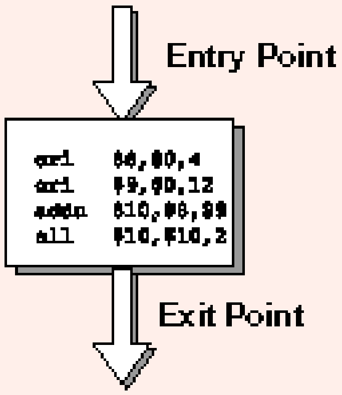
Is there a way to *prevent* distant statements from jumping into the middle of a block?

A good answer might be:

No.

Bug-free Blocks

Bugs can happen when control jumps into the middle of a block of code. If the block was written to execute from start to finish, the starting statements need to execute. *Don't jump into blocks,* or bugs will happen.



This is one of the ideas of **structured programming**. A block of code is a list of instructions that has one **entry point** (where execution starts) and one **exit point** (where execution leaves the block). The entry point has well defined *entry conditions*. The exit point has well defined *exit conditions*.

For the block to execute correctly, execution must start at the entry point, and the entry conditions must be met. When execution leaves the block, the exit conditions are true (if the block itself is bug free).

QUESTION 8:

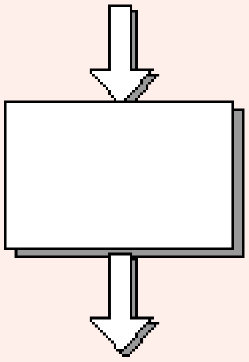
What if the block is **not** bug free? Where will you find the bug?

A good answer might be:

In the block.

Structure Rule One: Code Block

If the entry conditions are correct, but the exit conditions are wrong, the bug must be in the block. This is not true if execution is allowed to jump into a block. The bug might be anywhere in the program. Debugging under these conditions is much harder.



Rule 1: a code block is structured

> **Rule 1 of Structured Programming:** A code block is structured. In flow charting terms, a box with a single entry point and single exit point is structured.

This may look obvious, but that is the idea. Structured programming is a means of making it clear if a program is correct.

In assembly language there is no syntax for showing program blocks. You think about them when you design the program, and when you draw a flowchart. But in coding you just follow your design.

QUESTION 9:

How is a code block implemented in assembly language?

A good answer might be:

By writing statements intended to execute ins sequence and not jumping into the middle of them.

Implementing Rule One

Statements automatically execute in sequence. There is no language support for enforcing the single entry/single exit idea. A programmer must consciously follow the rule.

It might look as though the rule could be followed by only jumping to labeled statements. But a statement in the middle of an intended block can have a label, as in the following:

```
start:   ori   $8,$0,4      # $8 = 4
         ori   $9,$0,12     # $9 = 12
midblk:  addu  $10,$8,$9    # $10 = 12+4 = 16
         sll   $10,$10,2    # $10 = 16*4 = 64

         ....

         ori   $8,$0,99     # $8 = 99
         ori   $9,$0,43     # $9 = 43
         j     midblk       # jump to the second statement after start
```

QUESTION 10:

Is there a syntax for defining code blocks in high-level languages like Pascal, C, or Java?

Is there a syntax for defining code blocks in high-level languages like Pascal, C, or Java?

A good answer might be:

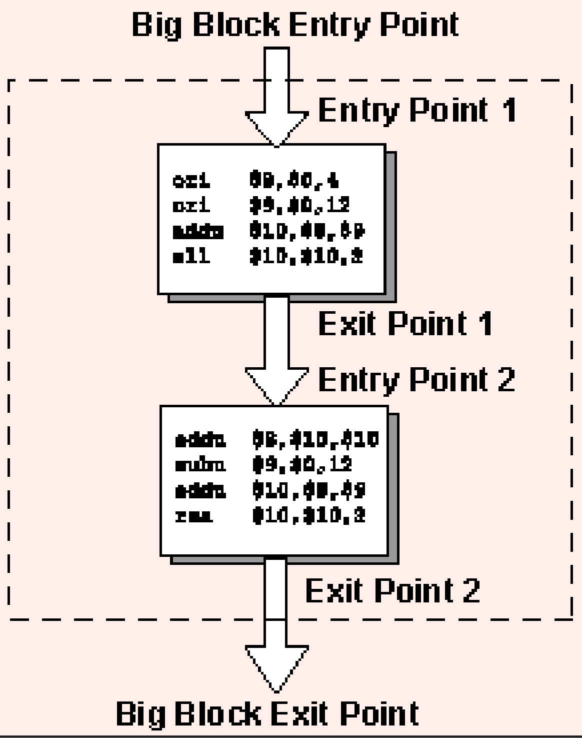
Yes: in Pascal **`begin end`** delimit a block. In C and Java **`{ }`** delimit a block.

Sequence

Pascal, C, and Java are structured languages. A block of statements in those languages has one entry point and one exit point.

What about two blocks in a row? Say that the exit conditions of block 1 are the correct entry conditions for block 2. Then the two blocks can follow in sequence.

The set of two blocks can be regarded as a one big block. If the entry conditions for block 1 are correct, then the exit conditions for block 1 are correct, then the entry conditions for block 2 are correct, then (finally) the exit condtions for block 2 are correct.



```
ori     $8,$0,4
ori     $9,$0,12
addu    $10,$8,$9
sll     $10,$10,2
```

```
addu    $8,$10,$10
subu    $9,$0,12
addu    $10,$8,$9
sra     $10,$10,2
```

QUESTION 11:

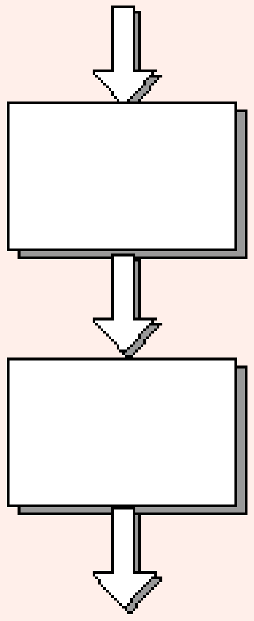
Look at the big block (in dotted lines). If the big block entry conditions are correct, what do you know about the big block exit conditions?

A good answer might be:

If the big block entry conditions are correct, then the big block exit conditions are correct.

Structure Rule Two: Sequence

Any sequence of blocks is correct if the exit and entry conditions match up. Execution enters each block at the block's entry point, and leaves through the block's exit point. The whole sequence can be regarded as a single block, with an entry point and an exit point.



Rule 2: a sequence of code blocks is structured

Rule 2 of Structured Programming: Two or more code blocks in sequence are structured.

The assembly language implementation of this rule is the same as rule one: a programmer must consciously follow the rule. This means that there must be no jumps elsewhere in the code to points inside the blocks.

QUESTION 12:

Are **if-then-else** structures possible in assembly language?

A good answer might be:

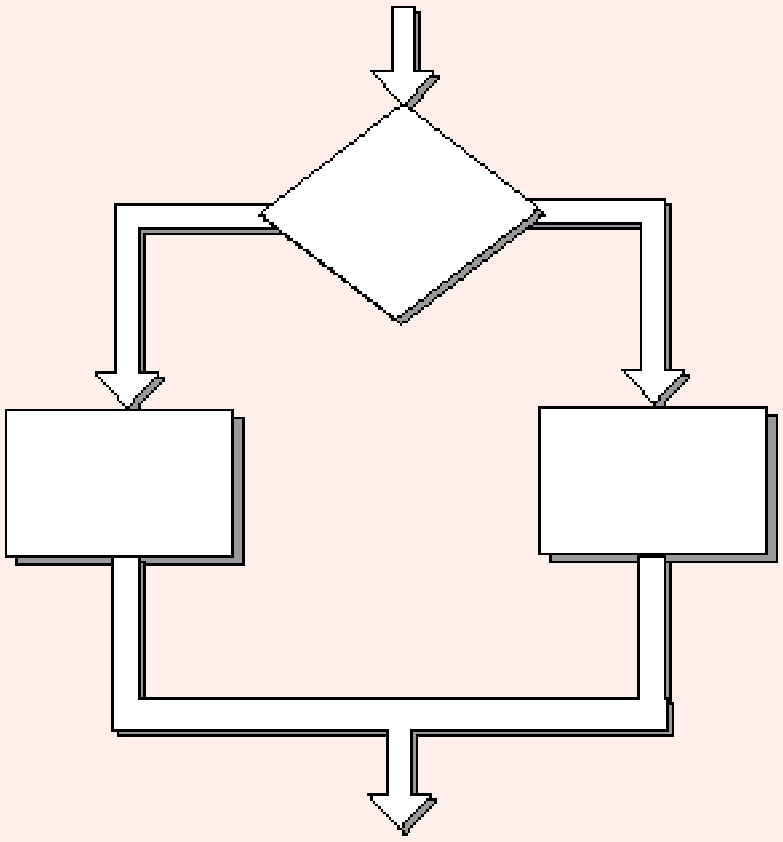
Yes.

Structure Rule Three: Alternation

If-then-else is sometimes called **alternation** (because there are alternative choices). In structured programming, each choice is a code block. If alternation is arranged as in the flowchart at right, then there is one entry point (at the top) and one exit point (at the bottom). The structure should be coded so that if the entry conditions are satsified, then the exit conditions are fulfilled (just like a code block).

Rule 3 of Structured Programming: The alternation of two code blocks is structured.

An example of an entry condition for an alternation structure is: *register $8 contains a signed integer*. The exit condition might be: *register $8 contains the absolute value of the signed integer*. The branch structure is used to fulfill the exit condition.


Rule 3: an alternation of code blocks is structured

QUESTION 13:

Can the condition tested for the alternation be complicated?

A good answer might be:

Sure. It might involve many assembly statements

Implementing Alternation

There is no explicit support for alternation in assembly language. The programmer must do it in pieces, as in the following:

```
        ...              #
        beq  $8,$9,else    # branch if equal
        nop                # branch delay slot
        ...              #
        ...              #   false branch
        ...              #
        j    cont
        nop
else:   ...              #
        ...              #   true branch
        ...              #
endif:  add   $10,$10,$11  # always executed
```

This is just an example, not the only way alternation is done. The true branch can come first, if that is more convenient. Conceptually, the branch chosen depends on the outcome of a true/false condition. In implementation, the condition might involve several branch and other instructions.

QUESTION 14:

Is an **if-endif** (single alternative) structure structured?

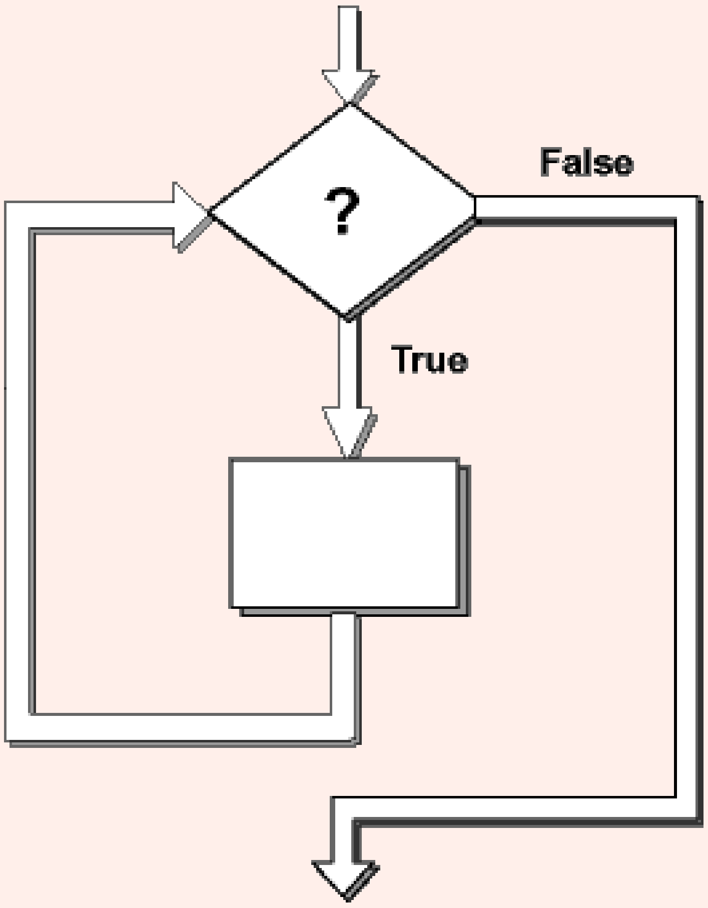
A good answer might be:

Yes. It is regarded as if-then-else with an empty else-branch.

Structure Rule Four: Iteration

If **iteration** (*while-loop*) is arranged as at right, it also has one entry point and one exit point. With careful coding, the entry point has conditions that must be satisfied and the exit point has conditions that will be fulfilled. There are no jumps into the structure from external points of the code.

Rule 4 of Structured Programming: The iteration of a code block is structured.



Rule 4: The iteration of a code block is structured

As with the other structures, an iteration structure must be built out of assembly language pieces. There are various ways to implement iteration. The following is typical

```
while:  bltz   $8,endWh

        . . .

        j      while
        sll    $0,$0,0

endWh:  sll    $0,$0,0
```

The particular instructions that make up the test depend on circumstances.

QUESTION 15:

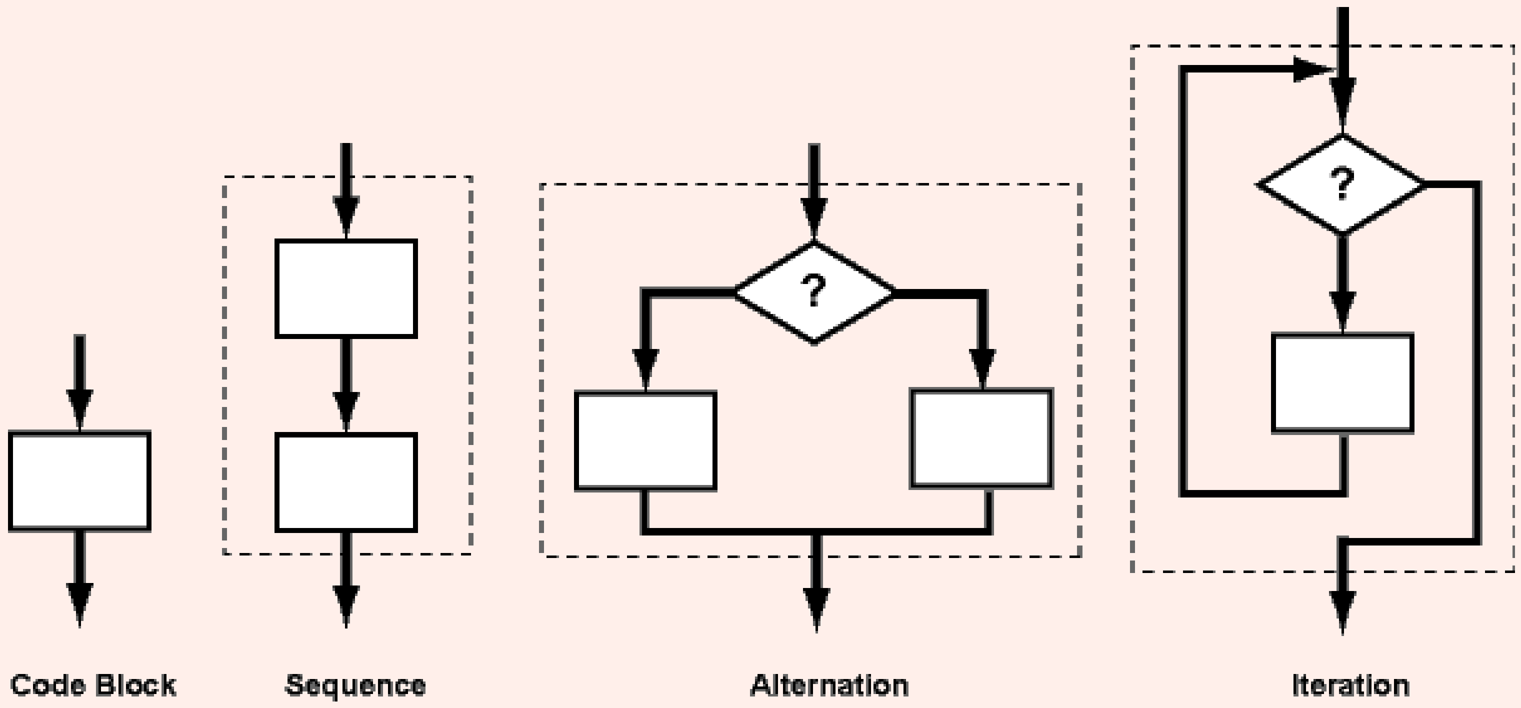
In a structured language (such as Pascal, C, or Java) can an alternation structure be put inside an iterations structure? (can the body of a while-loop contain an if-then-else?)

A good answer might be:

Of course

Looks like a Code Block

In all of the structures so far: there has been an entry point at the top and an exit point at the bottom. There are no jumps into or out of the structure. The entry point has entry conditions that are expected to be satisfied for the code to work, and exit conditions that are fulfilled if they are. Each of the structures can be considered to be a code block.



It is OK for a structure to have jumps within it (in assembly language, this is how some structures are implemented). From outside each of the above structures, the structure looks like a code block.

QUESTION 16:

Could a code block acting as a loop body be replaced by an alternation structure?

A good answer might be:

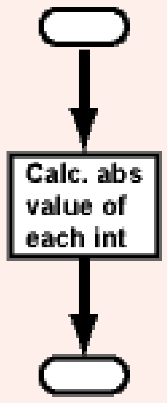
Yes

Structure Rule Five: Nesting Structures

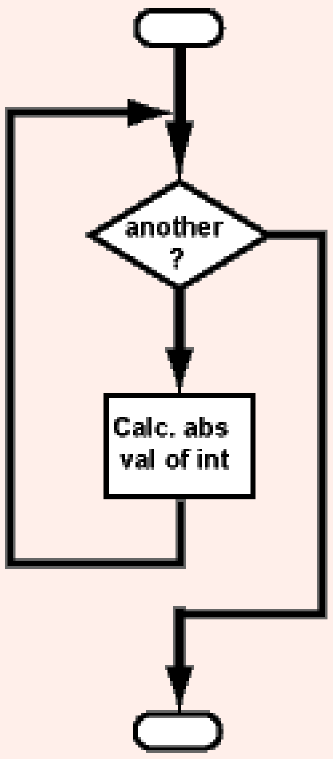
In flowcharting terms, where ever there is a code block, it can be expanded into any of the structures. Or, going the other direction, if there is a portion of the flowchart that has a single entry point and a single exit point, it can be summarized as a single code block.

> **Rule 5 of Structured Programming:** A structure (of any size) that has a single entry point and a single exit point is equivalent to a code block.

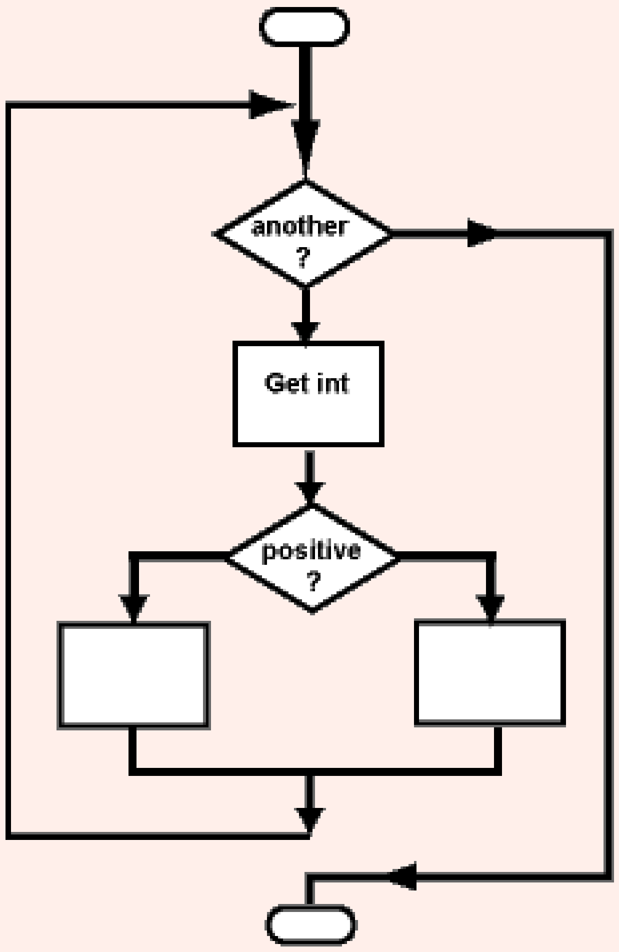
For example, say that you are designing a program to go through a list of signed integers calculating the absolute value of each one. You might (1) first regard the program as one block, then (2) sketch in the iteration required, and finally (3) put in the details of the loop body.



1. First View



2. Second View



3. Third View

Or, you might go the other way. Once the absolute value code is working, you can regard it as a single code block to be used as a component of a larger program.

QUESTION 17:

(Thought question:) Would it be a good idea to implement a complicated code block as a subroutine (as a function)?

A good answer might be:

Yes. We will do this in a later chapter.

Computational Power of Structured Programming

You might think that these rules are OK for ensuring stable code, but that they are too restrictive. Something must be lost. But nothing is lost. Böhm and Jacopini proved that any program can be written in a structured style. No computing power is lost by restricting control flow to the forms of structured programming.

The other control structures you know from other languages, such as case statements, *do-until*, *do-while*, and *for* are not needed. They are sometimes convenient. They are usually included as part of structured programming. In assembly language they add little convenience (since you have to implement them yourself). I recommend using only the structures listed in this chapter.

Any program you need to write in assembly can be written in a structured style. You should think about a problem and the means of solving it in exactly the way you would using a high-level, structured language. Once you have the solution designed, the implementation becomes mere "coding". Coding is the near-mechanical translation of a design into code. Solution design and coding should not occur simultaneously (in an ideal world, where all projects are on time, and dot-coms never fail).

QUESTION 18:

(Thought question:) Can a structured assembly program sometimes be speeded up by using non-structured control?

A good answer might be:

Yes. But you should write the structured version first. Keep it around, so that when the speedy version turns out to be buggy you have something that works.

End of Chapter

You have reached the end of this chapter. Click on a blue subject to make a smooth, structured transition to where it was discussed. To return here, click on the "back arrow" button of your browser.

- The computing power of a processor.
- The minimum set of instructions needed for full computing power.
- The throughput of a computer.
- CISC and RISC
- Rule 1 of structured programming
- Rule 2 of structured programming
- Rule 3 of structured programming
- Rule 4 of structured programming
- Rule 5 of structured programming

You have reached the end of the chapter.

created: 07/26/01

CHAPTER 20 — Loop and Branch Examples

This chapter contains two example programs that illustrate loops and branches. It also discusses how character strings and integer arrays are represented in assembly language.

Chapter Topics:

- *Null terminated character strings.*
- *String length program.*
- *Integer arrays.*
- *Array summing program.*

QUESTION 1:

A character *string* is a sequence of bytes containing ascii patterns. What is at the end of a *null-terminated* string?

What is at the end of a *null-terminated* string?

A good answer might be:

A zero byte (also called a null byte).

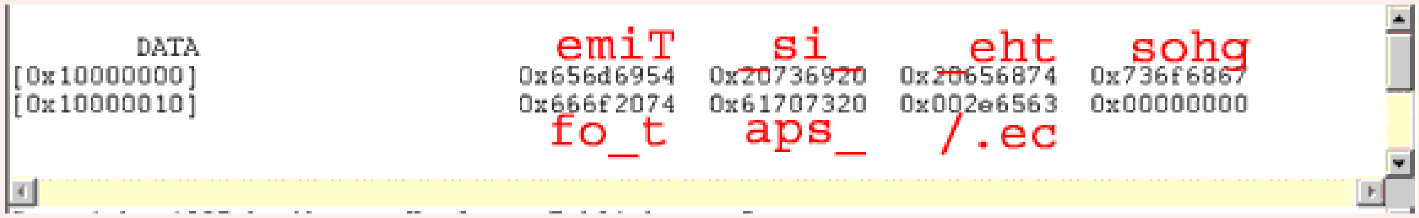
Null-terminated String

A null terminated string is a sequence of ASCII characters, one to a byte, with the last byte zero (null). Null terminated strings are common in C and C++. In the following picture, the blank bytes are filled with ASCII space (0x20). The last byte is filled with 0x00, represented with a slash.

Here is how this string is declared in assembly language:

```
        .data
str:    .asciiz "Time is the ghost of space."
```

Here is what it looks like in the data display of SPIM running on a Windows PC. The bytes are in ascending sequence beginning with 'I' (at address 0x10000000). However the display is aimed at displaying 32-bit little-endian integers. Every group of four bytes is displayed from right to left. The 'e' is at address 0x10000003. Space is 0x20, shown in red as '_'. The null byte is at address 0x1000001B



```
            DATA
[0x10000000]                 0x656d6954  0x20736920  0x20656874  0x736f6867
[0x10000010]                 0x666f2074  0x61707320  0x002e6563  0x00000000
```

QUESTION 2:

Why (do you suppose) null was picked as the byte to put at the end of a string?

A good answer might be:

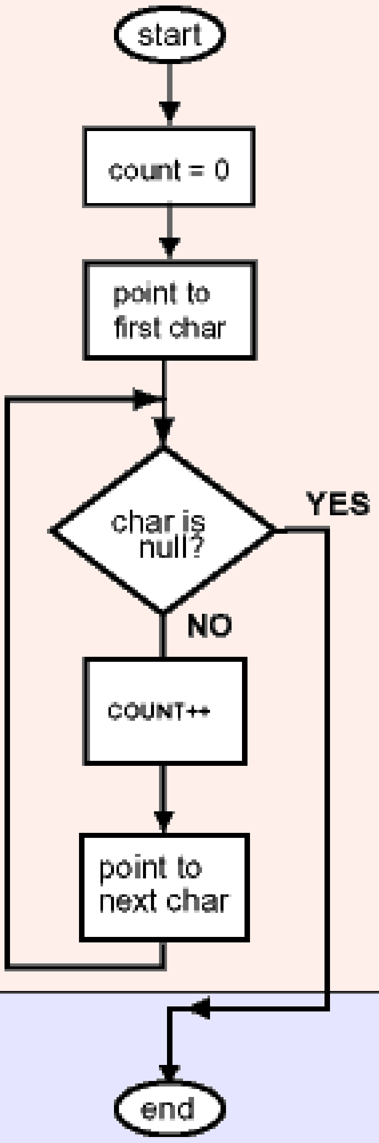
Null bytes (bytes containing 0x0) occur often in memory. A string that was accidentally not terminated would run on for longer than intended, but usually would soon encounter a null.

String Length

The length of a null-terminated string is defined as the number of characters it contains not counting the null. To calculate this start the count at one with the first character. Increment the count for each successive non-null byte.

The flow chart shows this procedure. It describes the algorithm in general terms. Assembly language details are left to the coding stage. Here is an outline of the program:

```
## strlen.asm
##
## Count the charaters in a string
##
## Registers:
## $8 --- count
##

        .text
        .globl  main
# Initialize
main:   ori     $8,$0,0         #  count = 0

# while not ch==null do
loop:


        j    loop
# finish
done:   sll   $0,$0,0

        .data
string: .asciiz  "Time is the ghost of space."
## End of file
```



QUESTION 3:

Why is the null not counted in the length of a string? (Hint: consider concatenation of two strings.)

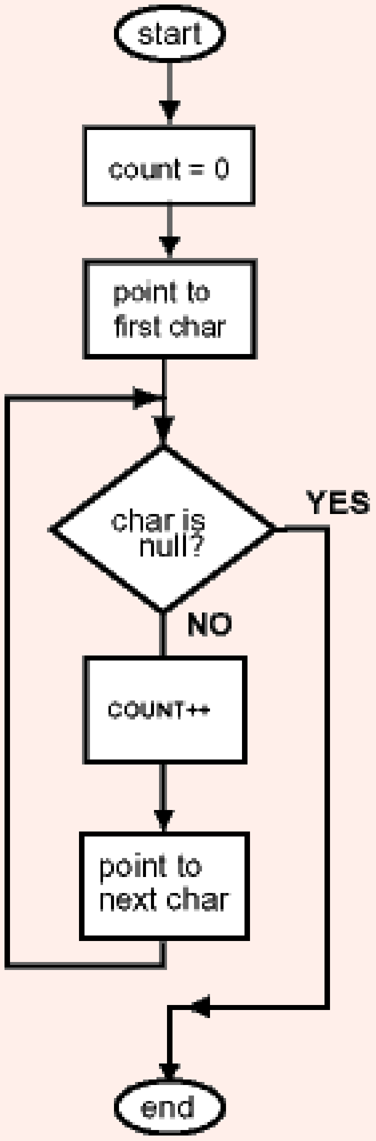
A good answer might be:

You would like length(string1) + length(string2) = length(concat(string1, string2)). If the two nulls of string1 and string2 were counted this would be false, because the contatenation has only one null (at the end).

Program continued

There are several ways to do what the second box of the chart calls for: "point at the first character". In a few chapters doing this will be much easier. But, for now, remember that the data section starts at the address 0x10000000. The top halfword of this is 0x1000. This is loaded into the top halfword of the base register with lui

```
## strlen.asm
##
## Count the charaters in a string
##
## Registers:
## $8 --- count
## $9 --- pointer to the char
## $10 --- the char (in low order byte)

        .text
        .globl  main
# Initialize
main:   ori     $8,$0,0       #  count = 0
        lui     $9,0x1000     #  point at first char

# while not ch==null do
loop:   lbu   $10,____($9)     # get the char

        _____   $10,$0,done    # exit loop if char == null



        j       loop
# finish
done:   sll   $0,$0,0          # target for branch

        .data
string: .asciiz  "Time is the ghost of space."
## End of file
```



QUESTION 4:

A few more statements have been added to get and test the current character. Fill in the blanks.

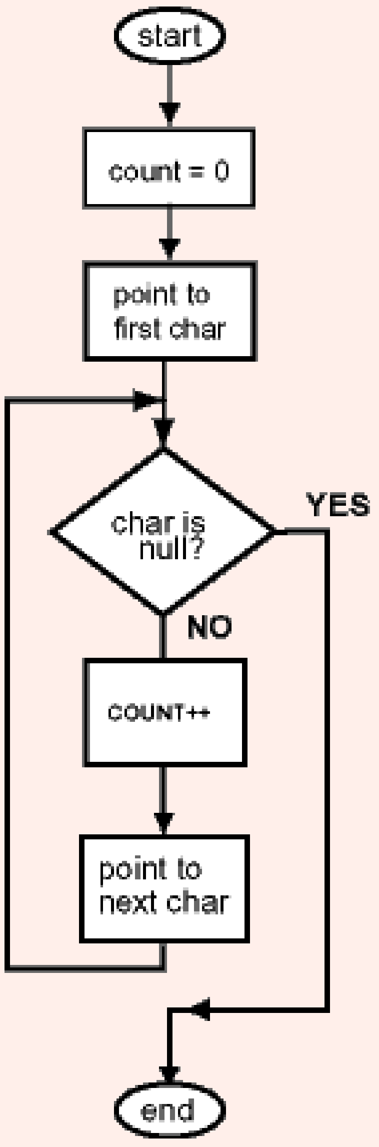
A good answer might be:

The answer is below.

Loop Body

The address of the byte in the lbu instruction is computed by displacement + $9. Since the base, $9, has the address of (points at) the character, the diplacement is zero. The lbu loads the character into the low-order byte of $10. The upper three bytes are zero. The beq instruction tests if the entire register is zero, but, of course, that also tests if the low-order byte is zero.

```
## strlen.asm
##
## Count the charaters in a string
##
## Registers:
## $8 --- count
## $9 --- pointer to the char
## $10 --- the char (in low order byte)

        .text
        .globl  main
# Initialize
main:   ori     $8,$0,0       #  count = 0
        lui     $9,0x1000     #  point at first char

# while not ch==null do
loop:   lbu   $10,0($9)       # get the char

        beq   $10,$0,done     # exit loop if char == null

        addiu    ___,___,___   # count++

        addiu    ___,___,___   # point at the next char

        j       loop
# finish
done:   sll   $0,$0,0         # target for branch

        .data
string: .asciiz  "Time is the ghost of space."
## End of file
```



Next, the program must increment the count, then move the base register (the character pointer) to the next byte of the string.

QUESTION 5:

Fill in the blanks.

A good answer might be:

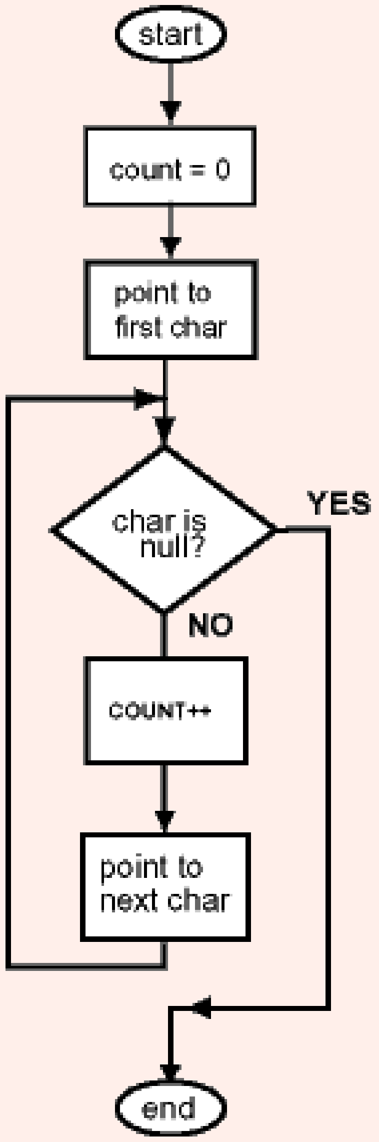
The complete program is below.

Complete Program

The base register is "moved" through the string by increasing the address by one for each byte. This is also called "moving a pointer".

This program has an important concept on every line. Using these concepts is how you program! Play with the program so that these concepts become 0x10nd nature!

```
## strlen.asm
##
## Count the charaters in a string
##
## Registers:
## $8 --- count
## $9 --- pointer to the char
## $10 --- the char (in low order byte)

        .text
        .globl  main
# Initialize
main:   ori     $8,$0,0       #  count = 0
        lui     $9,0x1000     #  point at first char

# while not ch==null do
loop:   lbu     $10,0($9)     # get the char

        beq     $10,$0,done   # exit loop if char == null
        addiu   $8,$8,1       # count++
        addiu   $9,$9,1       # point at the next char
        j       loop

# finish
done:   sll     $0,$0,0       # target for branch

        .data
string: .asciiz  "Time is the ghost of space."
## End of file
```



The program is very close to the C standard library function *int strlen(char*)*. As it is written, the program has its own data, and it is not invoked by another function. We will do both of those things in a few chapters.

QUESTION 6:

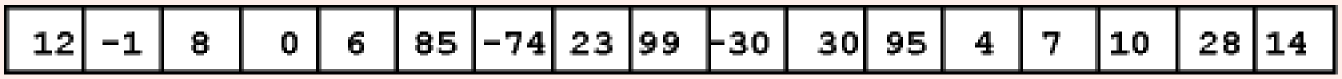
Does the program work correctly if the string is the null string (the string of length zero that consists of just a null byte)?

A good answer might be:

Yes.

Integer Array

An array of integers is a sequence of integers in successive words of memory. Usually the number of integers of the array is kept in a separate location in memory.



In assembly language an array of integers is declared using the directive .word followed by a list of comma separated integers.

```
        .data
size:   .word  17
array:  .word  12, -1, 8, 0, 6, 85, -74, 23, 99, -30, 30, 95, 4, 7, 10, 28, 14
```

The .word directive puts the data in word aligned locations.

QUESTION 7:

Why can't an array end with a null word, like with strings?

A good answer might be:

Because a null word, 0x00000000, is a legitimate (and common) integer.

Example Program: sum of an array

Sometimes programmers try to end an array with a pattern that they think will never occur in the data, like 0x10000000. This is an invitation for bugs. The pattern might truely never occur in correct data, but faulty data is common, and your program should deal with it gracefully.

The next example program uses an array of integer. The length of the array is given. The program calculates three sums:

- The sum of all integers.
- The sum of the positive integers.
- The sum of the negative integers.

You have probably seen this example before, done in Java or C.

QUESTION 8:

Sketch out the flowchart for the program (or write it in pseudo code).

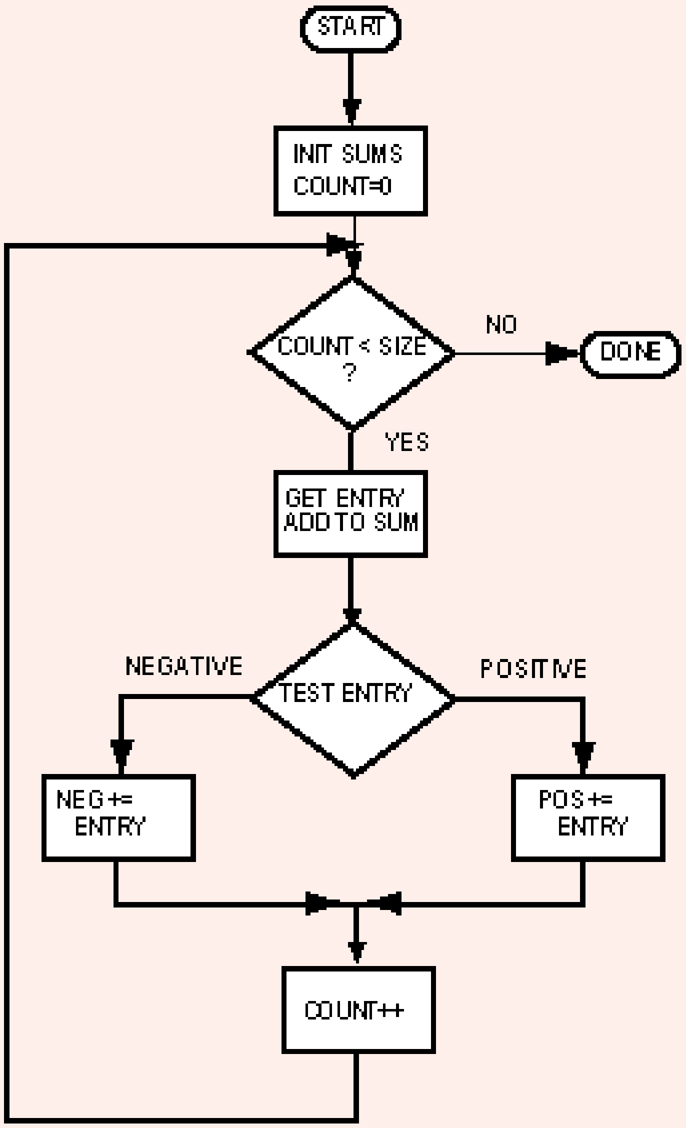
A good answer might be:

See Below

Flowchart

The flow chart could work for any language. Specific assembly language details are left out. The chart is a structured flowchart. It is nice to have a structured design before coding in an unstructured language like assembly. Here is an outline of the program:

```
## addIntArray.asm
##
## Sum all integers, the positive integers,
## and the negative integers in an array.

## Registers:
## $8 --- count
## $9 --- pointer to the array entry
## $10 -- current array entry
## $11 -- sum of all integers
## $12 -- sum of negative integers
## $13 -- sum of positive integers
## $14 -- pos. or neg. flag
## $15 -- SIZE of the array

        .text
        .globl  main
# Initialize
main:   ori     $____,$0,0      #  count = 0
        ori     $____,$0,0      #  sum = 0
        ori     $____,$0,0      #  neg = 0
        ori     $____,$0,0      #  pos = 0

# while  count < SIZE do
loop:

        j       loop
        sll     $0,$0,0         #  branch delay
```



```
        .data
size:   .word  17
array:  .word  12, -1, 8, 0, 6, 85, -74, 23, 99, -30, 30, 95, 4, 7, 10, 28, 14

## End of file
```

SPIM initializes registers to zero, but it is good practice to explicitly zero accumulators.

QUESTION 9:

Fill in the blanks.

A good answer might be:

See below

Loading the Array Size

Next, load the size of the array into register $15. The size is the first word of the data section. Recall that the data section starts at the address 0x10000000. Load the top half of that address into the base register.

```
## Registers:
## $8 --- count                    $12 -- sum of negative integers
## $9 --- pointer to the array entry  $13 -- sum of positive integers
## $10 -- current array entry       $14 -- pos. or neg. flag
## $11 -- sum of all integers       $15 -- SIZE of the array

## Initialize
main:   ori     $8,$0,0       #  count = 0
        ori     $11,$0,0      #  sum = 0
        ori     $12,$0,0      #  neg = 0
        ori     $13,$0,0      #  pos = 0

        ____    $9,0x1000     #  point at SIZE
        ____    $15,0($9)     #  get SIZE
        addiu   $9,$9,____    #  point to first entry

# while  count < SIZE do
loop:

        j       loop
        sll     $0,$0,0       #  branch delay

        .data
size:   .word  17
array:  .word  12, -1, 8, 0, 6, 85, -74, 23, 99, -30, 30, 95, 4, 7, 10, 28, 14
```

Next, load SIZE into register $15. Finally, point the base register at the first word of the array.

QUESTION 10:

Fill in the blanks.

A good answer might be:

See below.

Building the Loop

Perhaps you added one to the base register, rather than four. The base register $9 needs to be increased by the size of a full word, four. Now work on getting the loop correct. The beq instruction branches out of the loop if the count has reached the array size.

```
## Registers:
## $8 --- count                        $12 -- sum of negative integers
## $9 --- pointer to the array entry  $13 -- sum of positive integers
## $10 -- current array entry         $14 -- pos. or neg. flag
## $11 -- sum of all integers         $15 -- SIZE of the array

main:    ori      $8,$0,0        #  count = 0
         ori      $11,$0,0       #  sum = 0
         ori      $12,$0,0       #  neg = 0
         ori      $13,$0,0       #  pos = 0
         lui      $9,0x1000      #  point at SIZE

         lw       $15,0($9)      #  get SIZE
         addiu    $9,$9,4        #  point to first entry

# while  count < SIZE do
loop:    beq      $____,$____,done
         sll      $0,$0,0        #  branch delay


         addiu    $8,$8,____     # count++
         addiu    $9,$9,____     # point at next entry
         j        loop
         sll      $0,$0,0        #  branch delay

done:    sll      $0,$0,0        #  target for branch

        .data
size:   .word  17
array:  .word  12, -1, 8, 0, 6, 85, -74, 23, 99, -30, 30, 95, 4, 7, 10, 28, 14
```

At the bottom of the loop, the count is incremented and the base register is move to the next array entry.

QUESTION 11:

Fill in the blanks.

A good answer might be:

See below.

Add Entry to Sum

The count is increased by one. The pointer is increased by four. Next get the current array element and add it to the sum of all integers.

```
## Registers:
## $8 --- count                   $12 -- sum of negative integers
## $9 --- pointer to the array entry  $13 -- sum of positive integers
## $10 -- current array entry        $14 -- pos. or neg. flag
## $11 -- sum of all integers        $15 -- SIZE of the array

main:   ori     $8,$0,0       #  count = 0
        ori     $11,$0,0      #  sum = 0
        ori     $12,$0,0      #  neg = 0
        ori     $13,$0,0      #  pos = 0
        lui     $9,0x1000     #  point at SIZE

        lw      $15,0($9)     #  get SIZE
        addiu   $9,$9,4       #  point to first entry

# while  count < SIZE do
loop:   beq     $8,$15,done
        sll     $0,$0,0       #  branch delay


# get entry, add to sum
        lw      $10,____($____) #  get entry

        sll     $0,$0,0       #  load delay

        addu    $11,$____,$____ #  add to sum

        addiu   $8,$8,1       # count++
        addiu   $9,$9,4       # point at next entry
        j       loop
        sll     $0,$0,0       #  branch delay

done:   sll     $0,$0,0       #  target for branch

       .data
```

```
size:   .word  17
array:  .word  12, -1, 8, 0, 6, 85, -74, 23, 99, -30, 30, 95, 4, 7, 10, 28, 14
```

QUESTION 12:

Fill in the blanks.

A good answer might be:

See below.

Is the Entry Negative?

Next, set a flag (register $14) if the array entry is negative. Branch to the symbolic address neg if it is.

```
## Registers:
## $8 --- count                        $12 -- sum of negative integers
## $9 --- pointer to the array entry  $13 -- sum of positive integers
## $10 -- current array entry         $14 -- pos. or neg. flag
## $11 -- sum of all integers         $15 -- SIZE of the array

# while  count < SIZE do
loop:   beq     $8,$15,done
        sll     $0,$0,0        #  branch delay

# get entry, add to sum
        lw      $10,0($9)      #  get entry
        sll     $0,$0,0        #  load delay
        addu    $11,$11,$10    #  add to sum

#  test neg. or pos.
        slti    $14,$10,______  #  set flag if entry is neg
        bne     $___,$___,neg  #  branch if negative
        sll     $0,$0,0        #  branch delay

neg:
        addiu   $8,$8,1        # count++
        addiu   $9,$9,4        # point at next entry
        j       loop
        sll     $0,$0,0        #  branch delay

done:   sll     $0,$0,0        #  target for branch

        .data
size:   .word  17
array:  .word  12, -1, 8, 0, 6, 85, -74, 23, 99, -30, 30, 95, 4, 7, 10, 28, 14
```

QUESTION 13:

Fill in the blanks.

A good answer might be:

See below

True and False Branches

Next implement the two branches; one adds to the sum of negatives, the other adds to the sum of positives. Check that you don't have the branches confused.

```
## Registers:
## $8 --- count                    $12 -- sum of negative integers
## $9 --- pointer to the array entry  $13 -- sum of positive integers
## $10 -- current array entry        $14 -- pos. or neg. flag
## $11 -- sum of all integers        $15 -- SIZE of the array

# while  count < SIZE do
loop:   beq      $8,$15,done
        sll      $0,$0,0        #  branch delay


# get entry, add to sum
        lw       $10,0($9)      #  get entry
        sll      $0,$0,0        #  load delay
        addu     $11,$11,$10    #  add to sum

#  test neg. or pos.
        slti     $14,$10,0      #  set flag if entry is neg
        bne      $14,$0,neg     #  branch if negative
        sll      $0,$0,0        #  branch delay
        ____     $13,$13,$10    #  positive: add to PLUS
        j        _______
        sll      $0,$0,0        #  branch delay

neg:    ____     $12,$12,$10    #  negative: add to NEG
ifend:  addiu    $8,$8,1        #  count++
        addiu    $9,$9,4        #  point at next entry
        j        loop
        sll      $0,$0,0        #  branch delay

done:   sll      $0,$0,0        #  target for branch

        .data
size:   .word  17
```

```
array:  .word  12, -1, 8, 0, 6, 85, -74, 23, 99, -30, 30, 95, 4, 7, 10, 28, 14
```

QUESTION 14:

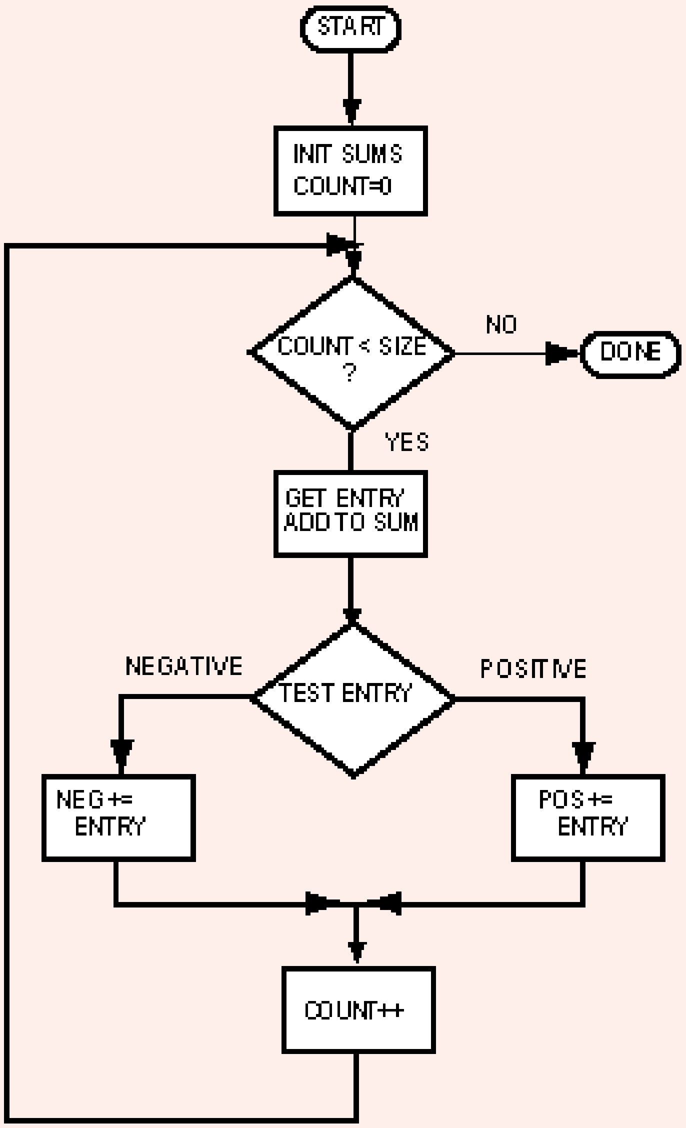
Fill in the blanks.

A good answer might be:

See below.

Complete Program

Here is the complete program suitable for running. The data in the array has been simplified to make testing easier. Study how the code matches the flow chart.



```
## addIntArray.asm
##
## Sum all integers, the positive integers,
## and the negative integers in an array.

## Registers:
## $8 --- count
## $9 --- pointer to the array entry
## $10 -- current array entry
## $11 -- sum of all integers
## $12 -- sum of negative integers
## $13 -- sum of positive integers
## $14 -- pos. or neg. flag
## $15 -- length of the array


        .text
        .globl  main
# Initialize
main:   ori     $8,$0,0        #  count = 0
        ori     $11,$0,0       #  sum = 0
        ori     $12,$0,0       #  neg = 0
        ori     $13,$0,0       #  pos = 0
        lui     $9,0x1000      #  point at SIZE

        lw      $15,0($9)      #  get SIZE
        addiu   $9,$9,4        #  point to first entry

# while  count < SIZE do
loop:   beq     $8,$15,done
        sll     $0,$0,0        #  branch delay
```

```
# get entry, add to sum
        lw      $10,0($9)     #  get entry
        sll     $0,$0,0       #  load delay
        addu    $11,$11,$10   #  add to sum

#  test neg. or pos.
        slti    $14,$10,0x0   #  set $14 if entry is neg
        bne     $14,$0,neg    #  branch if negative
        sll     $0,$0,0       #  branch delay
        addu    $13,$13,$10   #  positive: add to PLUS
        j       ifend
        sll     $0,$0,0       #  branch delay

neg:    addu    $12,$12,$10   #  negative: add to NEG

ifend:  addiu   $8,$8,1       # count++
        addiu   $9,$9,4       # point at next entry
        j       loop
        sll     $0,$0,0       #  branch delay

# finish
done:   sll     $0,$0,0       # target for branch

        .data
size:   .word  4
array:  .word  1, 2, -2, -1

## End of file
```

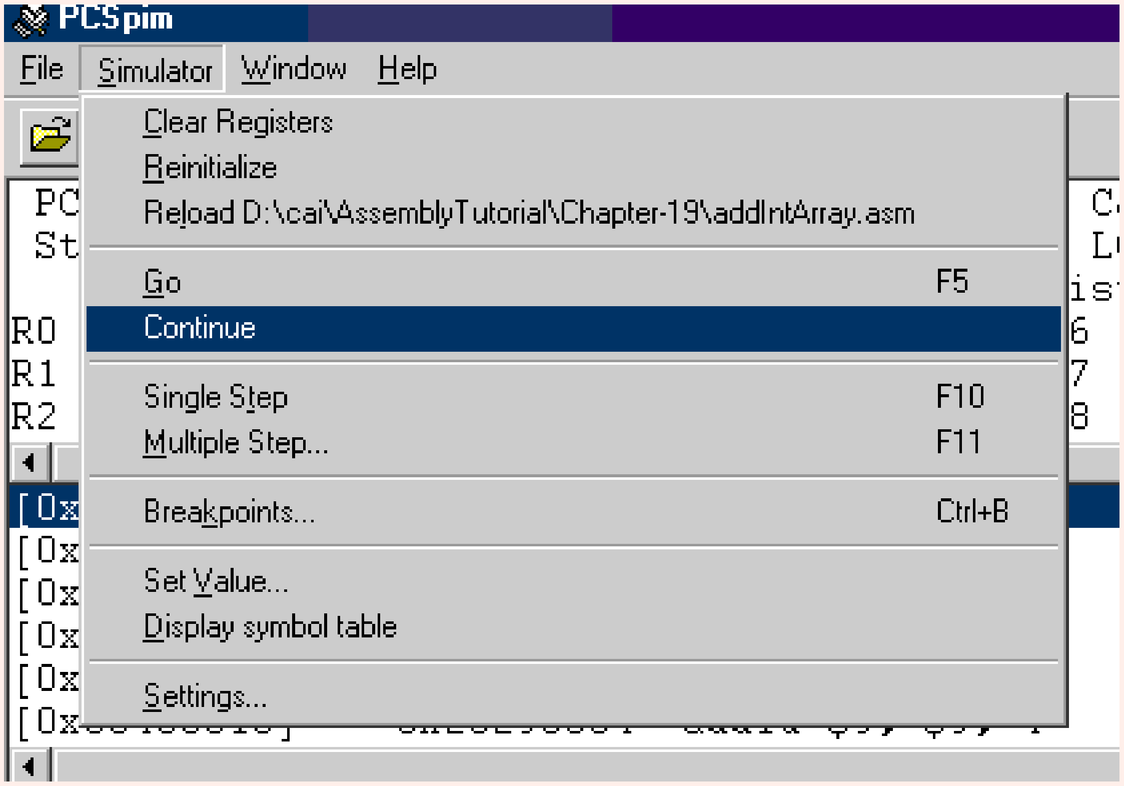
QUESTION 15:

Could the code be rearranged to make better use of the delay slots?

A good answer might be:

Yes. You might try this as an exercise.

Running the Program



You can run the program by initializing the PC (to 0x00400000) and repeatedly hitting single step (F10). But when the array has many entries this is tedious. On the version of SPIM for windows you can do the following to start the program and run it to completion:

1. Initialize the PC (as usual).
2. Hit F10 once (or more, if you want).
3. Click on "Continue" in the "Simulator" menu.
4. The program will execute until the PC goes beyond the end of the program.
5. The simulator halts.

This procedure is not very elegant. Set a breakpoint at the last instruction if you want. Or set a breakpoint at the top of the loop so that the loop body executes once per "Continue".

If you click "Go" in the menu, the simulator tries to start with some code that is not there and halts. "Bare Machine" is set in the options menu, so the extra code is not there. Keep things this way for now, unless you want to experiment.

QUESTION 16:

Is a single run of the program enough to be certain that it is correct?

A good answer might be:

Even a simple program like this might have some bugs that only careful testing can uncover.

End of Chapter

You have reached the end of this chapter. Click on a blue subject to go to where it was discussed. To get back here, click on the "back arrow" button of your browser.

- Null-terminated strings.
- String length.
- The Initializing a base register.
- Moving a base register through a string.
- Implementing an integer array in assembly.
- Moving a base register through an array of integers.

contents

You have reached the end of the chapter.

created: 07/30/01; revised 11/09/01

CHAPTER 21 — Pseudoinstructions

On the MIPS, the assembler creates a view of the processor a level above raw machine code. It implements an abstract machine that has a much richer instruction set than the MIPS hardware.

Chapter Topics:

- *Register mnemonic names.*
- *Pseudoinstructions and the pseudocomputer.*
- *move pseudoinstruction.*
- *li pseudoinstruction.*
- *la pseudoinstruction.*
- *lw pseudoinstruction.*
- *sw pseudoinstruction.*
- *nop pseudoinstruction.*

The previous chapters of these notes did not use these extended features of the asembler so that the actual machine could be seen. This chapter makes greater use of the extended assembler features.

QUESTION 1:

(Review:) What is a *general purpose register*?

Next

A good answer might be:

A general purpose register is a register that is visible to assembly language. The *floating point resgisters*, although visible to assembly language, are not regarded as general purpose registers. (A register is an electronic device that holds bit patterns.)

Register Mnemonics

Although the registers are called "general purpose", it is conventional in software for each register to be used for a specific purpose. The *extended assembler* uses mnemonic names for registers that groups them according to use. The following table shows this. Registers $8 through $15 are conventionally used to hold temporary values. The mnemonic name for these registers is $t0 through $t7.

Register Number	Mnemonic Name	Conventional Use		Register Number	Mnemonic Name	Conventional Use
$0	zero	Permanently 0		$24, $25	$t8, $t9	Temporary
$1	$at	Assembler Temporary		$26, $27	$k0, $k1	Kernel
$2, $3	$v0, $v1	Value returned by a subroutine		$28	$gp	Global Pointer
$4-$7	$a0-$a3	Arguments to a subroutine		$29	$sp	Stack Pointer
$8-$15	$t0-$t7	Temporary		$30	$fp	Frame Pointer
$16-$23	$s0-$s7	Saved registers		$31	$ra	Return Address

As far as *hardware* is concerned, the only registers $0 and $31 are different from the rest ($0 is always full of zeros and $31 is automatically used by some subroutine linkage instructions to hold the return address). The remaining registers are electronically identical. It is *software convention* to use different sets of registers for different purpose.

QUESTION 2:

Have new hardware registers been added to the processor with the use of register mnemonic names?

Next

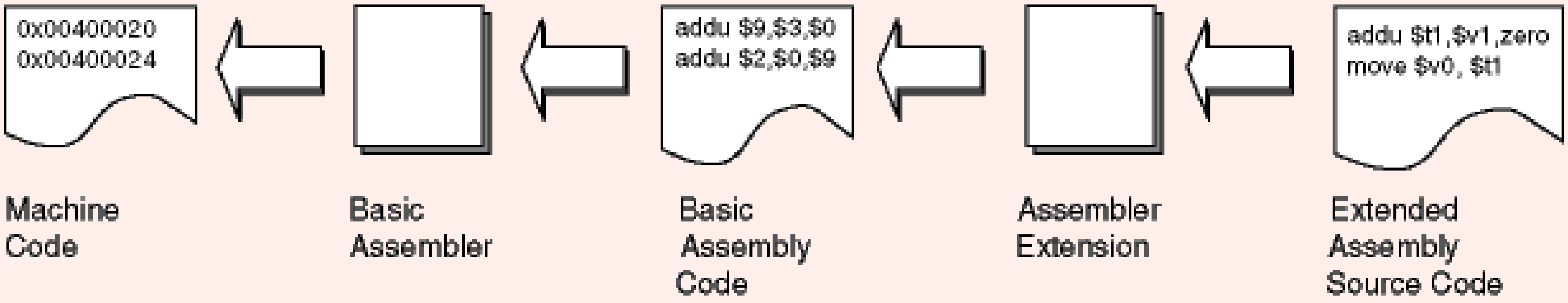
A good answer might be:

No--register mnemonic names are a feature implemented by the assembler.

Extended Assembler

The assembler software is written so that (if you ask for it) mnemonic register names can be used in place of hardware register numbers. The assembler replaces register mnemonics with register numbers. Of course, the assembly instruction is ultimately translated into machine code.

SPIM implements many features that are not provided by MIPS hardware. When it does this it is an **extended assembler**. The extended assembler accepts source code that uses extentions to basic assembly language. The extentions in the source code are replaced with basic assembly language, and then translated into machine code.



By using the extended assembler the programmer may program in a more convenient language, but the underlying hardware remains unchanged. Of course *high-level* languages such as C go even further: rather than merely extending the assembly language of the machine a completely new programming language is supported by the underlying hardware.

QUESTION 3:

The programs in the previous chapters used registers $8 through $15. What are the mnemonic names for these registers? See the register table.

Next

A good answer might be:

$8 through $15 == $t0 through $t7

Registers for Temporary Values

Most assembly programs, like most higher-level language programs, are collections of subroutines. Subroutines use *local variables* that are not visible outside the subroutine. The **temporary** registers $t0-$t7,$t8, and $t9 and the **saved** registers $s0-$s7 are used for local variables (or for the same idea in assembly language).

When you program a subroutine you assume that the temporary registers and saved are yours to use however you want. However, by software convention, the temporary registers may change across a subroutine call. The saved registers don't changed across a subroutine call.

```
        ori  $t0,$0,32    # load 32 into temp register
        ori  $s0,$0,13    # load 13 into saved register

        jal  subrout      # subroutine call
        sll  $0,$0,0      # delay slot

 back:  addu $s0,$s0,$t0  # return from sub;
                          # temp regs may have changed.
                          # saved regs have not changed.
```

The statement at back is in error. The temporary register t0 may have been changed by the subroutine subrout.

QUESTION 4:

Could the main program be considered a subroutine?

Next

A good answer might be:

Yes---to the operating system `main` is a subroutine.

Other Registers

The registers are grouped into those used as variables and arguments in subroutines ($0, $v0-$v1, $a0-$a3, $t0-$t9, $s0-$s7), and those used for other purposes. Subroutines are discussed in a future chapter. The other uses are discussed as the topic arises.

The core of an operating system is the **kernel**. The kernel contains those routines that are the most fundamental to the OS. The rest of the OS is mostly implemented in terms of the functions provided by the kernel. Two registers, $k0 and $k1, are reserved for the kernel. Applications programs (and most OS subroutines) should not touch them.

Of course, SPIM does not come with an OS. It loads a tiny monitor, if requested. But don't use $k0 and $k1.

The registers $gp, $sp, and $fp are base registers for access to various parts of memory.

QUESTION 5:

What does the following assembly language instruction do?

```
addu $t5,zero,$t7
```

It ________ the contents of register ________ into register ________.

A good answer might be:

What does the following assembly language instruction do?

```
addu $t5,zero,$t7
```

It copies the contents of register $7 into register $5.

Pseudoinstructions

The addu instruction, used as above, copies a bit pattern from the source register into the destination register. This operation is usually called a **move** operation, although the source register does not change. It is often convenient to move a value from one general purpose register to another.

It is awkward to say "add" when you mean "move". You might wish that there was a mnemonic for "move". In fact, the *extended assembler*, allows you to use the mnemonic move instead of "addu". This mnemonic does not correspond to a new machine instruction. It is a **pseudoinstruction** that the assembler translates into the appropriate basic assembly instruction.

A **pseudoinstruction** is an instruction used in assembly language that the extended assembler replaces with one or several basic assembly instructions. The extended assembler has many more pseudoinstructions than the hardware has opcodes.

```
move d,s        #  copy the contents of
             #  the source register s
             #  to the destination register d
             #  (pseudoinstruction)
```

QUESTION 6:

(Review:) What is in register $t5 after the following instruction: ori $t5,$0,74

A good answer might be:

ori $t5,$0,74 puts the two's complement representation of 74_{10} into register $t5.

Load Immediate

The ori instruction, used as above, copies a bit pattern from the instruction into the destination register. This operation is usually called a **load immediate** operation---it loads a register with a value that is immediately available (without going to memory).

You might wish that you could a mnemonic for "load immediate". With the extended assembler, you can. Use the li mnemonic. This is a pseudoinstruction that the assembler translates into the appropriate basic assembly instruction.

```
li   d,value       #  load register $d with the
                 #  positive or negative integer
                 #  "value"
```

QUESTION 7:

Translate the following *pseudoinstruction* into the corresponding basic assembly instruction (use mnemonic register names):

Next

li $v0,12
==
ori ____,____,____

A good answer might be:

```
li $v0,12
==
ori $v0,zero,12
```

Several Translations

The **load immediate** pseudoinstruction is useful for loading a register with an integer value. Here is another example:

```
li  $t2,-153
```

This puts the two's complement representation of -156$_{10}$ into register $t2. However, this can not be done with ori because the immediate field of that instruction must be a 16-bit unsigned integer. The immediate operand is *zero-extended* into the 32 bits of the destination register. Because the immediate value of the li is negative, a different translation is used:

```
li $v2,-153     ==     addiu $v2,$0,-153
```

(Remember that addiu does a two's complement addition with its operands. The u means that overflow does not cause a trap). The extended assembler translates li into different basic instructions, depending on the immediate value.

QUESTION 8:

Can either basic instruction, ori or addiu use an immediate value greater than 16 bits?

Next

A good answer might be:

No. The machine instruction has a 16-bit field for the immediate value.

Two Basic Instructions

Machine instructions have fixed-sized fields. Immediate operands are always 16 bits. However, two machine instructions can be used to load a 32-bit register with the top half and bottom half of a 32-bit integer:

```
li $v2,0x12345678 == lui $v2,0x1234
                     ori $v2,$v2,0x5678
```

In this situation, li translates into two basic instructions. The first loads the top half of $v2; the second ors-in the bottom half. Of course, the same is done for any integer (positive or negative) that requires seventeen to thirty-two bits. So, from the extended assembly language programmer's point of view there is a very useful instruction:

The extended assembler automatically translates this pseudoinstruction into the most efficient sequence of actual instructions.

QUESTION 9:

Translate the following *pseudoinstruction* into the corresponding basic assembly instruction: li $t7,0xFF001234

Next

A good answer might be:

```
li $t7,0xFF001234 == lui $t7,0xFF00
                     ori $t7,$t7,0x1234
```

The immediate operand of the pseudoinstruction is the representation of a negative integer (notice that the "sign bit" is set). You don't have to use hex with li. The programmer could have written: li $t7,-16772555.

Example Program

Here is a tiny example program using load immediate. There will be more realistic examples in pages to come.

```
## liEg.asm
##
        .text
        .globl  main

main:
        li    $t0,43        #  first alue
        li    $t1,-96       #  second value
        li    $t7,-16772555 #  third
        addu  $t0,$t0,$t1   #  add the values
        addu  $t0,$t0,$t7

## end of liEg.asm
```

You might wish to run the program with SPIM, mostly to see how the three li instructions translate into different basic instructions. To run it in SPIM, first look in the settings menu and check "allow pseudo instructions" and remove the check from "bare machine".

QUESTION 10:

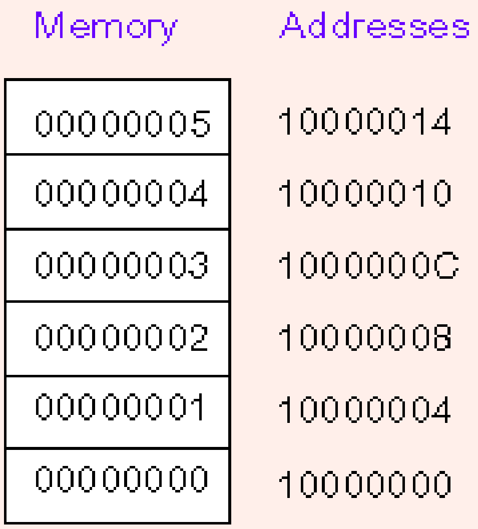
What is a *symbolic address*?

Next

A good answer might be:

A symbolic address is the name used in source code for a location in memory.

Unknown Addresses



Memory	Addresses
00000005	10000014
00000004	10000010
00000003	1000000C
00000002	10000008
00000001	10000004
00000000	10000000

A symbolic address does more than just a name a memory location. It stands for whatever location in memory a particular datum or instruction ends up at, *even if this is not known in advance.* Often a program is built using several object modules, library modules, and assembler source files. (See Chapter One on separate assembly.) The various modules refer to locations in each other using symbolic addresses. The actual addresses are determined by system software when all the modules are linked together and loaded into memory. It is very hard (and not necessary) for a programmer to know what addresses various items will eventually have.

The assembler enables a programmer to refer to an address by a symbol, even when the address will not be known until much later in the process of building an executable. The assembler and subsequent systems software (linker and loader) keep track of the symbolic address until the address it represents is finally determined. At run time, everything is resolved. A machine language program and data are loaded and executed.

QUESTION 11:

(Review:) In the picture, what data is in the word starting at address 0x1000000C? _________ What is the address of the word containing 0x00000002? _________

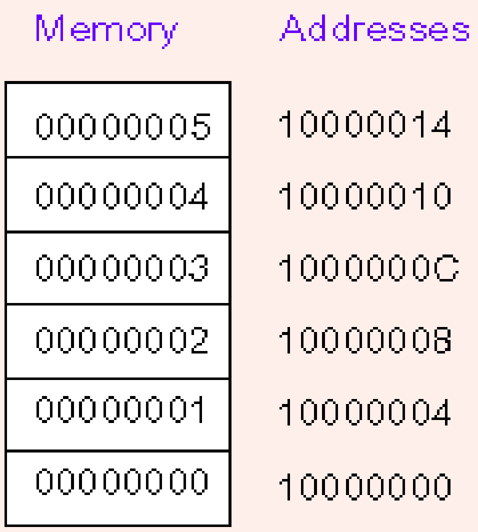
Next

A good answer might be:

(Review:) The picture is a run-time snapshot of memory while some program is running. What data is in the word starting at address 0x1000000C? `0x00000003` What is the address of the word containing 0x00000002? `0x10000008`

Address and Contents of an Address

The contents of a word of memory may be any 32-bit pattern. The address of a word of memory is also a 32-bit pattern. Both types of data fit into a 32-bit register.



Sometimes you want the contents of memory word to be loaded into a register. For example, you may wish to load the value 0x00000002. This is done with the lw instruction.

Sometimes you want to load the run-time address of a word in memory. For example, you may wish to load the address 0x10000008. The past chapters have done this when setting up a base register. This was done with code similar to the following:

```
main:                   #  Load memory address:
        lui   $10,0x1000      #  top half
        ori   $10,$10,0x0008  #  bottom half
        lw    $11,0($10)      #  Load contents of memory at the address

        . . .

        .data
val0:   .word   0
val1:   .word   1
val2:   .word   2         # base register points at "2"
val3:   .word   3
val4:   .word   4
val5:   .word   5
```

QUESTION 12:

Is there a problem with this code?

Next

A good answer might be:

Yes: the programmer must know the address in advance.

Load Address Pseudoinstruction

It is not always possible to know the address in advance. But systems software and symbolic addresses make this unnecessary. The way this feature is accessed in assembly language is by using the la mnemonic. This mnemonic does not correspond to a single machine instruction. It is a *pseudoinstruction* that the assembler translates into to several machine instructions.

```
la   d,exp         #  load register $d with the address
                   #  described by the expression "exp"
                   #  "exp" is often a symbolic address
```

In this pseudoinstruction, exp is an expression that evaluates to a memory address. There are several possible forms for exp. Often it is a symbolic address. The la pseudoinstruction translates into different machine code instructions depending on exp.

QUESTION 13:

Does the la instruction access memory?

Next

A good answer might be:

No. At run time, the one or few machine instructions it translates into ensure that the designated register is loaded with the required address.

LA Example

Here is the previous example code done using la:

```
main:                     #  Load memory address:
        la    $10,val2       #  complete 32-bit address into $10
        lw    $11,0($10)     #  Load contents of memory at the address

        . . .

        .data
val0:   .word   0
val1:   .word   1
val2:   .word   2
val3:   .word   3
val4:   .word   4
val5:   .word   5              # base register points at "5"
```

QUESTION 14:

This la instruction translates into two machine instructions. What two instructions does it translate into? (Answer with two instructions mnemonics.)

Next

A good answer might be:

This la translates into a lui followed by a ori (just as we previously did by hand).

SPIM Example

Here is a SPIM example using la. To run it in SPIM, first look in the settings menu and check "allow pseudo instructions" and remove the check from "bare machine".

```
## addEm.asm
## program to add two integers
##
        .text
        .globl  main

main:
        la    $t0,val2     #  complete 32-bit address into $t0
        lw    $t1,0($t0)   #  load first value, 2
        lw    $t2,4($t0)   #  load second value, 3
        addu  $t1,$t1,$t2  #  calc. sum


        .data
val0:   .word   0
val1:   .word   1
val2:   .word   2
val3:   .word   3
val4:   .word   4
val5:   .word   5

## end of addEm.asm
```

Execute the program by pushing F10 for each machine instruction.

QUESTION 15:

How many times will F10 be pushed?

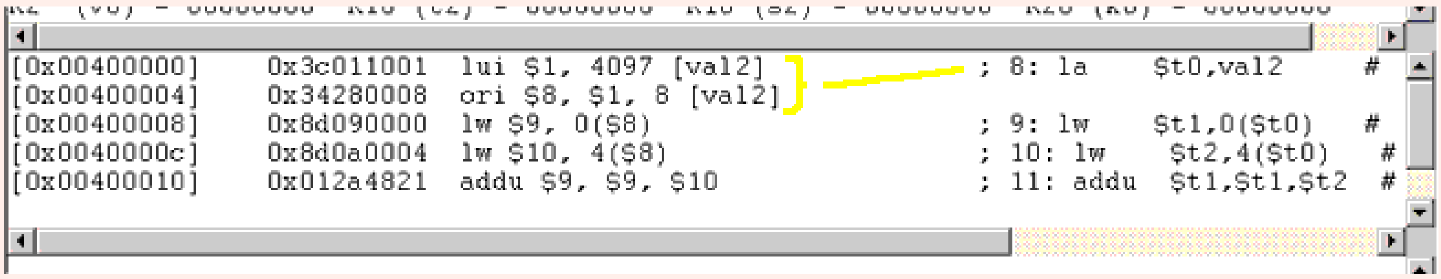
Next

A good answer might be:

Five: twice for the two instructions the la is translated into.

SPIM Run

Here is a run of the program. The SPIM listing shows the two machine instructions that the assembler used for the la pseudoinstruction. The assembler uses different machine instructions in different contexts.



```
[0x00400000]    0x3c011001  lui $1, 4097 [val2]         ; 8: la    $t0,val2      #
[0x00400004]    0x34280008  ori $8, $1, 8 [val2]
[0x00400008]    0x8d090000  lw $9, 0($8)                 ; 9: lw    $t1,0($t0)    #
[0x0040000c]    0x8d0a0004  lw $10, 4($8)                ; 10: lw    $t2,4($t0)   #
[0x00400010]    0x012a4821  addu $9, $9, $10             ; 11: addu  $t1,$t1,$t2  #
```

Look carefully at the two instructions that SPIM used for la. They use register $1 as the base register.

QUESTION 16:

In the register table. what is the mnemonic name for register $1?

Next

A good answer might be:

$at, the *assembler temporary* register.

Assembler Temporary Register

The *assembler temporary register* is reserved for the assembler to use in translating pseudoinstructions. Often a pseudoinstruction is implemented as a short sequence of machine instructions that use $at. Here is how la was translated in the example:

```
la $t0,val5  <==>  lui $1,4097
                   ori $8,$1,8
```

Sometimes a pseudoinstruction has the same mnemonic as the non-extended assembler instruction. For example, lw (load word) is a (non-extended) assembler instruction. As an assembler instruction, it is used like this:

```
lw    $t1,8($t0)   #  load the word at address $t0+8
```

The assembler instruction designates the address with a displacement (eight, in this case) to add to a base register ($t0, in this case). This corresponds exactly to one machine instruction. There is no other form for the non-extended assembler instruction. However, with the extended assembler, the following *psuedo*instruction can be used:

```
lw    $t1,exp    #  load value at  address exp
```

QUESTION 17:

The *assembler instruction* lw uses a base register. When the *pseudoinstruction* lw is translated, what base register is used?

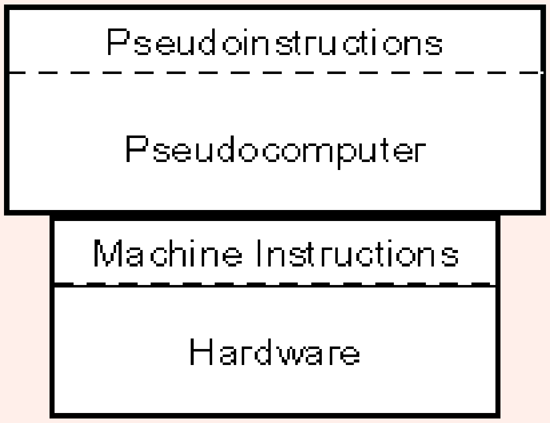
Next

A good answer might be:

The assembler temporary register, $at which is also $1.

Load Word PseudoInstruction

Here is a possible translation the pseudoinstruction lw. Say that the symbol data stands for the address 0x10000004



```
lw $t0,data  <==>  lui $1,0x1000
                   lw $8,4($1)
```

The extended assembler and its pseudoinstructions makes it look like you are programming a *psuedo*computer---a computer system that is richer in features and more convenient to program than the actual hardware. (Often this is called a "virtual computer"; but to avoid confusion I'll say that pseudoinstructions run on the pseudocomputer.) This idea of implementing a virtual machine on top of another machine (which may also be virtual) is very important in computer science.

The pseudocomputer built ontop of the hardware is a consistent *layer of abstraction*. When you write programs using the extended assembler and its pseudoinstructions you think in terms of the pseudocomputer and its capabilities. You can pretend that the pseudocomputer will directly execute your prograjm.

One of the most useful instructions of the pseudocomputer is the lw p-instruction. It copies a word of data from memory into a register seemingly in one step:

```
lw    d,exp      #  (pseudoinstruction) Load register $d
                 #  with the value at address exp.  exp can be any
                 #  of several expression types that evaluate to
                 #  an address
```

QUESTION 18:

Should the `lw` p-instruction be followed by a load delay slot? (Hint: look at its translation.)

Next

A good answer might be:

Yes. The second instruction in its translation is a non-extended load instruction, which requires a delay of one instruction before the loaded value can be used.

Store Word (pseudoinstruction)

The nop pseudoinstruction translates into sll $0,$0,0, the "no operation" instruction we have been using all along.

```
nop          #  (pseudoinstruction) no operation
             #  do nothing for one machine cycle.
```

The sw mnemonic is another that is used both by an unextended assembler instruction and a pseudoinstruction. With it the pseudocomputer can store a register to memory with one instruction.

```
sw    d,exp     #  (pseudoinstruction) Store register $d
                #  into the word at address exp.  exp can be any
                #  of several expression types that evaluate to
                #  an address
```

Of course, this is implemented as several non-extended instructions.

QUESTION 19:

Play Pseudocomputer: translate the sw instruction into non-extended instructions:

```
sw   $t0,someWord     ==     lui  _____,0x1000

                             sw   _____,12( ____ )
```

A good answer might be:

```
sw   $t0,someWord     ==      lui  $1,0x1000

                              sw   $8,12( $1 )
```

It would be OK to say $t0 for $8 and $at for $1.

Example Program

Here is an example of programming the pseudocomputer using the lw and sw instructions. The nop (no-operation) instructions are used to follow the rule that a mult instruction should not be started until two instructions after a mflo or mfhi instruction.

```
## pseudoPoly.asm
## evaluate the polynomial ax2 +bx + c
##
        .text
        .globl  main

main:
        lw   $t3,x          # get x
        lw   $t0,a          # get a
        lw   $t1,bb         # get bb
        lw   $t2,c          # get c

        mult $t3,$t3        # x2
        mflo $t4            # $t4 = x2
        nop
        nop
        mult $t4,$t0        # low  = ax2
        mflo $t4            # $t4  = ax2
        nop
        nop

        mult $t1,$t3        # low  = bx
        mflo $t5            # $t5  = bx
        addu $t5,$t4,$t5    # $t5  = ax2 + bx

        addu $t5,$t5,$t2    # $t5 = ax2 + bx + c
        sw   $t5,value      # value = polynomial
        .data
x:      .word   4
value:  .word   1
a:      .word  20
bb:     .word  -2
c:      .word   5

## end of pseudoPoly.asm
```

This is a straightforward evaluation of the polynomial. By using Horner's method, and by cleverly filling some of the no-ops with instructions, the program could be greatly improved.

QUESTION 20:

What big assumption was made in writing this program?

Next

A good answer might be:

The program assumes that all calculations fit into 32 bits.

End of Chapter

You have reached the end of this chapter. Click on a blue subject to go to where it was discussed. To get back here, click on the "back arrow" button of your browser.

- Mnemonic register names.
- The assembler temporary register.
- The kernel of an operating system.
- Systems software and symbolic addresses.
- Pseudoinstructions and the pseudocomputer.
- The move pseudoinstruction.
- The la pseudoinstruction.
- lw pseudoinstruction.
- sw pseudoinstruction.
- nop pseudoinstruction.

contents

You have reached the end of the chapter.

created: 08/05/01

CHAPTER 22 — SPIM Trap Handler

Up until now, the example programs run on SPIM simulating a "bare machine"--a computer with no machine code in it but your own. Most computer systems run under the direction of an operating system. Application programs use the services of the operating system to do input and output and other system tasks.

This chapter discusses the SPIM trap handler--a very small set of services that is a great help in writing assembly programs.

Chapter Topics:

- *The* syscall *instruction.*
- *SPIM trap handler.*
- Trap handler services:
  - *halt program*
  - *print string*
  - *read string*
  - *print integer*
  - *read integer*
- *Hello World* example.
- *Library Fine* example.

QUESTION 1:

In an actual (hardware) bare machine do you suppose that it is easy to write a character to the terminal?

A good answer might be:

No.

syscall

In even the simplest computer, putting a character on the screen involves many instructions and a detailed knowledge of the video card. Let's leave this topic until later (perhaps years later). SPIM includes a "trap handler" that simulates a tiny operating system that can do input and output to a console.

Assembly language programs request operating system services using the syscall instruction. (There is, unfortunately, no Ebert instruction). The syscall instruction (at run time) transfers control to the operating system, which then performs the requested service. Then control (usually) returns to the program. (This description leaves out many details).

```
        syscall          # invoke the operating system
```

Different OSes use this instruction in different ways. For SPIM, it is used like this:

```
        li      $v0,code  # "code" is the number of an OS service.

        .......         # Put parameters for the service in
        .......         # registers $a0, $a1 or $f12 (as appropriate).

        syscall          # Invoke the operating system.

                        # Return value (if any) is in $v0 or $f0
                        #
```

Different services use different registers, and not all return values.

QUESTION 2:

Is syscall a *pseudoinstruction*?

A good answer might be:

No. It is a 32-bit machine instruction.

Trap Handler Services

Here are the services of the SPIM trap handler. Details about how to use them are in the following pages. The *print* services write characters to the simulated monitor of SPIM. The *read* services read characters from the keyboard and (for numeric reads) convert them into the appropriate type.

Service	Code	Arguments	Returned Value
print integer	1	$a0 == integer	
print float	2	$f12 == float	
print double	3	($f12, $f13) == double	
print string	4	$a0 == address of string	
read integer	5		$v0 <- - integer
read float	6		$f0 <- - float
read double	7		($f0, $f1) <- - double
read string	8	$a0 == buffer address $a1 == buffer length	
allocate memory	9	$a0 == number of bytes	$v0 <- - address
exit	10		

The **exit** service stops the program. (Until now we have been single stepping or crashing into the bytes beyond the end of the program).

QUESTION 3:

What does the **exit** service of a real operating system do?

A good answer might be:

Control returns to the OS and does not return to the program. The OS then goes about its own tasks, possibly starting another user program.

Print String

The **print string** SPIM trap handler service prints a **null terminated** string to the simulated monitor. The address of the string is loaded into register $a0. Typically the string is in the data section of memory.

```
li      $v0,4       # code 4 == print string
la      $a0,string  # $a0 == address of the string
syscall             # Invoke the operating system.
 . . .

        .data
string: .asciiz     "Hello SPIM!\n"
```

The service sends bytes to the simulated monitor starting with the byte pointed to by $a0 until it hits the null byte. It does not check that the bytes are ascii, you will print garbage if you point $a0 at the wrong location.

If you want to advance to a new line, use the newline character '\n' inside or at the end of the string.

QUESTION 4:

Can the print string service be used to print out 32-bit integers?

A good answer might be:

No, not directly. Integers must be converted to characters before they are printed.

The Traditional Example

You are now ready to see the program you probably expected to see in Chapter One. The following program prints the string, then calls the **exit** service.

```
# hello.asm
#
        .text
        .globl  main
main:
        li      $v0,4       # code 4 == print string
        la      $a0,string  # $a0 == address of the string
        syscall             # Invoke the operating system.

        li      $v0,10      # code 10 == exit
        syscall             # Return to OS.

        .data
string: .asciiz      "Hello SPIM!\n"
# end of file
```

The string is printed to the monitor window of the simulator. The program is complete, and can be copied into a source file and run as is. But see the next page first.

QUESTION 5:

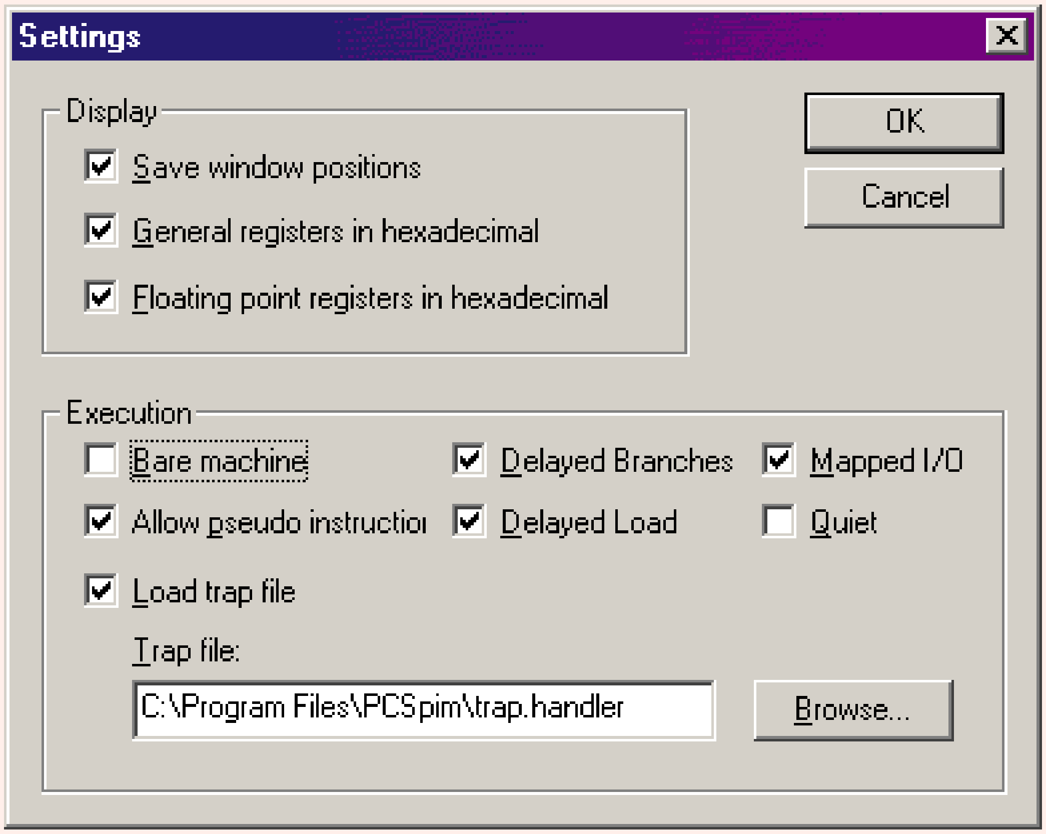
On a real (hardware) computer, what is the trap handler?

A good answer might be:

It is a machine language program in memory (possibly in ROM).

Starting SPIM

On SPIM, the trap handler is also a program that is optionally loaded into SPIM memory. On a real machine a trap handler might be permanently in ROM, might be loaded from a "boot sector" of the hard disk, or (in the good old days) might be entered into memory by hand.



Settings

Display
- [✓] Save window positions
- [✓] General registers in hexadecimal
- [✓] Floating point registers in hexadecimal

Execution
- [ ] Bare machine
- [✓] Delayed Branches
- [✓] Mapped I/O
- [✓] Allow pseudo instructio
- [✓] Delayed Load
- [ ] Quiet
- [✓] Load trap file

Trap file: C:\Program Files\PCSpim\trap.handler

To use the trap handler with SPIM ask for it in the Simulator Settings panel. Set the options "Allow pseudoinstructions" and "Load trap file" as well as the other options shown at right.

Now to assemble and load your program go to File Open and select the source file (just as we have been doing). Your program is loaded along with some initialization code. The initialization code starts at address 0x00400000, the address that has until now been where your programs started.

To run the program click on Simulator Go and then OK in the pop-up panel.

QUESTION 6:

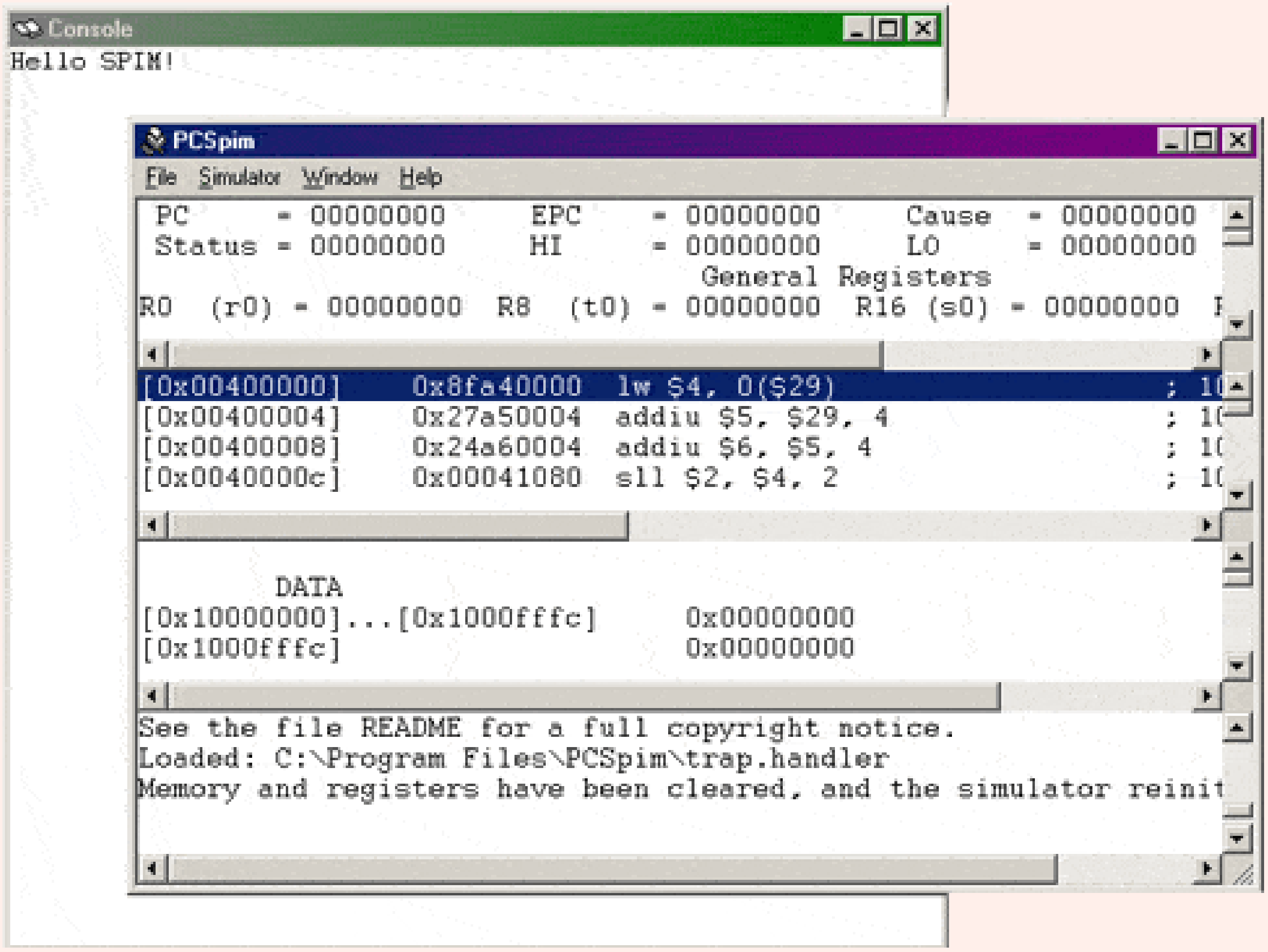
Can you single step starting at address 0x00400000 as before?

A good answer might be:

Yes, but since much of the code is not from your program single stepping is less interesting.

Example Output

Here is SPIM with the example "hello.asm" program. The simulated console is in the background, and the only code you can see is the initialization code. If (in your actual SPIM window) you scroll down the data window you will see the ascii codes for the string.



```
Hello SPIM!
```

```
 PC     = 00000000     EPC    = 00000000     Cause  = 00000000
 Status = 00000000     HI     = 00000000     LO     = 00000000
                             General Registers
R0  (r0) = 00000000  R8  (t0) = 00000000  R16 (s0) = 00000000  I
```

```
[0x00400000]    0x8fa40000  lw $4, 0($29)                    ; 1
[0x00400004]    0x27a50004  addiu $5, $29, 4                 ; 1
[0x00400008]    0x24a60004  addiu $6, $5, 4                  ; 1
[0x0040000c]    0x00041080  sll $2, $4, 2                    ; 1
```

```
        DATA
[0x10000000]...[0x1000fffc]     0x00000000
[0x1000fffc]                    0x00000000
```

```
See the file README for a full copyright notice.
Loaded: C:\Program Files\PCSpim\trap.handler
Memory and registers have been cleared, and the simulator reinit
```

QUESTION 7:

What happens if the program were slightly modified, as follows:

```
main:
        li      $v0,4       # code 4 == print string
        la      $a0,string  # $a0 == address of the string
        addiu   $a0,1       # add one to the address
        syscall             # Invoke the operating system.
        . . .
        .data
string: .asciiz      "Hello SPIM!\n"
```

A good answer might be:

The program prints "ello SPIM!"

Read Integer, Print Integer

The **read integer** service reads an entire line of input from your keyboard---all the characters you type up to the `newline` character. These characters are expected to be ASCII digits '0', '1', .., '9' with an optional leading '-' or '+'. The characters are converted into a 32-bit two's complement representation of the integer, returned in $v0.

```
li      $v0,5     # code 5 == read integer
syscall           # Invoke the operating system.
                  # $v0 <-- two's comp. int.
```

The **print integer** service prints the integer represented by the 32 bits in $a0 to the SPIM terminal. Of course, there are many ways that the integer can be placed in $a0, not just `lw`.

```
li      $v0,1     # code 1 == print integer
lw      $a0,int   # $a0 == the integer
syscall           # Invoke the operating system.
```

QUESTION 8:

The *print integer* prints an integer on the simulated monitor. But what must it do before printing?

A good answer might be:

The integer in $a0 is first converted from 32-bit two's complement representation into ascii characters.

Example Program

The example program reads in an integer, presumed to be a number of ounces, then writes out the equivalent number of pounds and ounces.

```
# ounces.asm
#
# Convert ounces to pounds and ounces.

        .text
        .globl  main

main:   li      $v0,4       # print prompt
        la      $a0,prompt  #
        syscall
        li      $v0,5       # read in ounces
        syscall

        li      $t1,16      # 16 oz. per pound
        divu    $v0,$t1     # lo = pound; hi = oz.

        mflo    $a0
        li      $v0,1       # print
        syscall             # pounds
        li      $v0,4       # print "pounds"
        la      $a0,pout
        syscall

        mfhi    $a0         # print
        li      $v0,1       # ounces
        syscall             #
        li      $v0,4       # print
        la      $a0,ozout   # "ounces"
        syscall

        li      $v0,10      # exit
        syscall
```

```
          .data
prompt: .asciiz "Enter ounces: "
pout:   .asciiz " Pounds\n"
ozout:  .asciiz " Ounces\n"
# end of file
```

As is often the case, input and output dominate the program. The actual calculation is hard to find.

QUESTION 9:

The program prints "Pounds" and "Ounces" on separate lines. Change the program so that they are printed on one line.

A good answer might be:

Remove the "\n" from " Ounces\n".

Read String

The trap handler can also read in a string from the keyboard.

```
li      $v0,8       # code 8 == read string
lw      $a0,buffer  # $a0 == address of buffer
li      $a1,16      # $a1 == buffer length
syscall             # Invoke the operating system.

 . . . .

        .data
buffer: .space 16   # reserve 16 bytes
```

Look at these details when you need them: Usually $a1 contains the length (in bytes) of the input buffer. Up to ($a1)-1 characters are read from the keyboard and placed in *buffer* as a **null terminated** string. The user ends the string by hitting "enter". The "enter" character appears at the end of the string as '\n' (0x0a) unless the string is ($a1)-1 characters long. No matter what, there is a null at the end of the string.

QUESTION 10:

Is the string that is read in immediately suitable for output using the *print string* service?

A good answer might be:

Yes.

A Fine Example

Sometimes you will have to remove the null at the end of the input string, perhaps because it is intended to be part of a larger string. Other times, as in the following example, you can use two *print string* operations.

The example program asks the user to enter a name followed by comma and "enter". Then the program writes out a personalized letter using the name. The body of the letter is printed using only one syscall. The personalized greeting is printed first in a separate syscall because it is null terminated.

```
# overdue.asm

        .text
        .globl  main

main:
        # get patron name
        li      $v0,4           # print prompt
        la      $a0,prompt      #
        syscall
        li      $v0,8           # code 8 == read string
        la      $a0,name        # $a0 == address of buffer
        li      $a1,24          # $a1 == buffer length
        syscall                 # Invoke the operating system.

        # print the letter
        li      $v0,4           # print greeting
        la      $a0,letter      #
        syscall
        li      $v0,4           # print body
        la      $a0,body        #
        syscall

        li      $v0,10      # exit
        syscall

        .data
```

```
prompt: .asciiz "enter name, followed by comma-enter: "
letter: .ascii "\n\nDear "
name:   .space 24

body:   .ascii "\nYour library books are way\n"
        .ascii "overdue.  Please return them\n"
        .ascii "before we give your name\n"
        .ascii "to the enforcement squad.\n\n"

# end of file
```

It is somewhat crude to make the user enter a comma at the end of the name. It would be nicer to have the program detect the end of the name and automatically put a comma after it. The program would be much longer.

QUESTION 11:

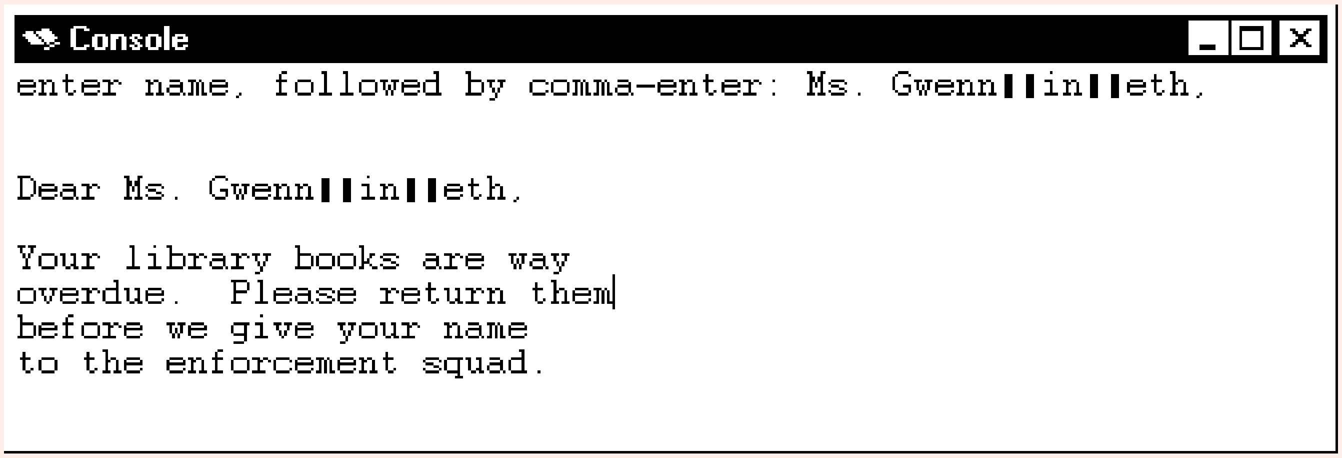
Is "backspace" a character?

A good answer might be:

Yes --- it is ascii code 0x08

No Line-buffer Correction

The user might make a mistake in entering the name and try to correct it by hitting the "backspace" key to erase the mistake. But this does not work. The ascii value of the key 0x08 is included in the string just as any character value. Odd byte values show up (in SPIM) as a small black rectangle.



```
enter name, followed by comma-enter: Ms. Gwenn▮▮in▮▮eth,


Dear Ms. Gwenn▮▮in▮▮eth,

Your library books are way
overdue.  Please return them
before we give your name
to the enforcement squad.
```

Most operating system user interfaces allow the user to edit a string (with BS, DEL, arrow and other keys) before it is sent to an application program. (This is sometimes called "cooked" input mode). But the SPIM service does not do this. Characters are sent as they were typed. (This is called "raw" input mode).

You could, of course, write your program so that it looks at the raw characters and corrects the input string. This is a messy task, and often skipped, but should not be.

QUESTION 12:

In a famous computer blunder, a radiation therapy device was controlled by an assembly program that did not used cooked input, and did not bother to check for valid input. What do you think happened?

A good answer might be:

Instead of the input, the patients were cooked.

End of the Chapter

You have reached the end of this chapter. Click on a blue subject to go to where it was discussed. To get back here, click on the "back arrow" button of your browser.

You have reached the end of the chapter.

created: 08/05/01; revised 11/09/01

CHAPTER 23 — Logic and Math Pseudoinstructions

This chapter further discusses pseudoinstuctions. When you write a program using *pseudoinstructions* it looks to you as if the computer has a rich set of convenient machine instructions. In fact, these convenient instructions are each implemented by using basic instructions. The extended assembler replaces each pseudoinstruction with one or several basic instructions. The basic instructions are then translated into machine code.

Chapter Topics:

- *Bitwise logic:*
  - *not pseudoinstruction.*
- *Integer Arithmetic:*
  - *addu pseudoinstruction.*
  - *subu pseudoinstruction.*
  - *negu pseudoinstruction.*
  - *mul pseudoinstruction.*
  - *div pseudoinstruction.*
  - *divu pseudoinstruction.*
  - *remu pseudoinstruction.*
- *Rotate Instructions:*
  - *rol rotate left pseudoinstruction.*
  - *ror rotate right pseudoinstruction.*

QUESTION 1:

What is the bit-wise **not** of this pattern: 0011 1001

Next

A good answer might be:

```
NOT( 0011 1001 ) == 1100 0110
```

not Pseudoinstruction.

The not pseudoinstruction changes each bit in a register. Each 1 is changed to a 0 and each 0 is changed to a 1.

```
not   d,s  # load register d with the
           #  bitwise not of register s
           # (pseudoinstruction)
```

NOR Operation on Bits				
first operand	0	0	1	1
second operand	0	1	0	1
	--	--	--	--
OR	0	1	1	1
not OR	1	0	0	0

This is a pseudoinstruction. The assembler translates it into a single basic assembly instruction, a nor instruction. The nor operation is given in the table. The **nor** of two bits is equivalent to applying **not** to the **or** of the bits. The nor instruction uses two registers as operands and writes the result to a third register:

```
nor  d,s,t  # $d <-- bitwise NOR $s with $t.
```

Look at the first two columns of the table where the first operand is zero. The **nor** of the two operands is the **not** of the second operand.

QUESTION 2:

How does the extended assembler translate:
not $s1,$t5
into:
nor ______, ______, ______

hint

Next

A good answer might be:

```
not $s1,$t5    =    nor $s1, $t5, $0
```

The order of the two operand registers could be reversed.

Implicit Immediate Operand

Some pseudoinstructions translate into different basic instructions depending on the type of operands used with it. A pseudoinstruction used with an immediate operand translates into different basic instructions than one used with all register operands.

Sometimes a mnemonic (*"or" for example*) is used for a basic instruction and for a pseudoinstruction. For example:

```
or $s0,$s1,0x00ff   ==> ori $s0,$s1,0x00ff
```

Here, the extended assembler notices the immediate operand and translates the instruction into an ori instruction. The resulting machine code is a 32-bit or immediate. However, the following is a basic instruction:

```
or $s0,$s1,$t1
```

This assembly instruction is not changed. The resulting machine code is a 32-bit and instruction.

The mmenomic and is similar. It might call for the basic instruction and, or the basic instruction andi, or other basic instructions depending on the operands.

QUESTION 3:

Is there both a basic add instruction and an add immediate instruction?

Next

A good answer might be:

yes

Arithmetic Pseudoinstructions

With the non-extended assembler there is an `addu d,s,t` instruction where d, s, and t are registers, and an `addiu d,s,im` instruction where im is an immediate operand.

With the extended assembler the `addu d,s,x` instruction translates into different basic instructions depending on **x**. **x** can be a register, a 16-bit immediate operand, or a larger immediate operand.

Examine the following instruction (for the extended assembler):

```
addu $t0,$s0,40000
```

The decimal value 40000 is too large for 16 bits, so the instruction cannot be translated into an addiu instruction. Two instructions must be used to add \$s0 to 40000_{10} and to place the result in \$t0.

QUESTION 4:

Fill in the blanks:

```
addu $s1,$s0,40000  ==

li   ______,40000    # use pseudoinstruction li
addu $s1,$s0,______  # perform 32-bit addition
```

Next

hint

A good answer might be:

```
addu $s1,$s0,40000  ==

li   $at,40000    # use pseudoinstruction li
addu $s1,$s0,$at  # perform 32-bit addition
```

Subtraction

The li in the above is itself a pseudoinstruction. It will be further translated into a basic instruction.

There is also (with the extended assembler) a subu d,s,x instruction where the last operand can be a register, a 16-bit immediate, or a 32-bit immediate. This instruction translates into the pseudoinstruction addu d,s,-x (which is then translated into basic instructions).

The negu d,s pseudoinstruction calculates the two's complement negation of register $s and puts it in register $d.

QUESTION 5:

Fill in the blanks:

```
negu $v2,$s3  ==  sub  ____,$0,____
```

Next

A good answer might be:

```
negu $v2,$s3  ==  sub  $v2,$0,$s3
```

Multiplication

The mult and the multu instructions (which you have seen before) put the result into MIPS registers **hi** and **lo**. The results may be moved out of these registers using mfhi and mflo.

The three operand pseudoinstruction mul d,s,t uses those basic instructions. It is used when the product of registers $s and $t is a value that fits into the 32 bits of **lo**.

QUESTION 6:

Finish the translation:

```
mul $v2,$s3,$t0  ==  multu  ____, ____
                     mflo   ____
```

Next

A good answer might be:

```
mul $v2,$s3,$t0  ==  multu  $s3,$t0
                     mflo   $v2
```

mul Pseudoinstruction

The mul pseudoinstruction makes it look as if MIPS has a 32-bit multiply instruction that places its 32-bit result in a register without checking for overflow:

```
mul  d,s,t # multiply $s by $t. put the
           # result in $d
           # (pseudoinstruction)
```

There is no overflow checking. The bits of **hi** are not examined nor saved. There are several other pseudoinstructions that examine **hi** and perform various actions on overflow. For now we won't use them.

A similar instruction exists for division. The basic instructions div s,t and divu s,t put their results in MIPS registers **hi** and **lo**. The 32-bit quotient goes in **lo** and the 32-bit remainder goes in **hi**. To move the quotient into a register, mflo is used.

QUESTION 7:

Finish the translation. The operand registers are s and t, the destination register is d.

```
div d,s,t  ==  div   ____, ____
               mflo  ____
```

A good answer might be:

```
div d,s,t  ==  div  s,t
               mflo d
```

div and divu Pseudoinstruction

The SPIM extended assembler includes these instructions:

```
div  d,s,t # divide $s by $t. Put the
           # quotient in $d. Operands are
           # two's complement.
           # (pseudoinstruction)
```

and

```
divu d,s,t # divide $s by $t. Put the
           # quotient in $d. Operands are
           # unsigned.
           # (pseudoinstruction)
```

Neither instruction checks for overflow. Other pseudoinstructions (not discussed here) do check for overflow.

QUESTION 8:

Is the remainder of an integer division sometimes useful?

A good answer might be:

Of course.

remu (Pseudoinstruction)

After a 32-bit integer division, the quotient is in the lo register and the remainder is in the hi register. The following pseudoinstruction uses several basic instructions to compute the remainder:

```
remu d,s,t # divide $s by $t. Put the
           # remainder in $d. Operands are
           # unsigned.
           # (pseudoinstruction)
```

Another useful pseudoinstruction creates the negative of a two's complement integer:

```
negu d,s   # d <-- -s
           # (pseudoinstruction)
```

There is a pseudoinstruction for creating the absolute value of an integer:

```
abs d,s    # d <-- |s|
           # (pseudoinstruction)
```

QUESTION 9:

What total amount is given to a waiter if the bill is $25, the tax rate is 8%, and the tip is 15% ? (Use integer math).

Next

What total amount is given to a waiter if the bill is $25, the tax rate is 8%, and the tip is 15% ? (Use integer math).

A good answer might be:

```
total = 25 + (25*(8+15))/100 = 25 + (25*23)/100 = 25 + 575/100 = 30
```

If the waiter grumbles about getting $30 instead of $31, just explain that in integer math 575/100 = 5.

Example Program

Write a program that calculates the total amount given the amount for the meal, plus 8% tax, plus 15% tip. The meal amount is input from the user. The tax and tip rates are fixed. Here is a run of the program:

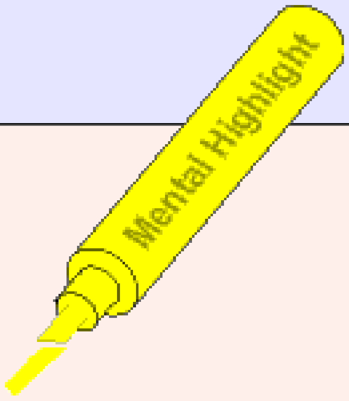
```
Enter food cost: 25
  Tax plus top:  5
    Total cost: 30
```

```
## rest.asm
##
## Total restaurant bill calculator
##
## Register Use:
##
## $s0 tip+tax
## $s1 tip rate
## $s2 tax rate
## $s3 meal cost
## $a0 tax+tip dollars

        .globl  main

        # Get meal cost
main:   li      $v0,4       # print a prompt (code 4 put in $v0)
        la      $a0,prompt  # address of prompt put in $a0
        syscall

        li      $v0,5       # input an integer  (code 5 put in $v0)
        syscall             # 32-bit result returned in $v0
        move    $s3,$v0     # save it in $s3

        . . . . .

        .data
tip:    .word   15          # tip rate in percent
tax:    .word    8          # tax rate in percent

prompt: .asciiz "Enter food cost: "
head1 : .asciiz "   Tax plus tip: "
head2 : .asciiz "     Total cost: "
# end of file
```

The only information needed from the user is the cost of the food. This is prompted for and then read in (as text) and converted (by part of the trap handler) into a 32-bit two's complement integer. The SPIM trap handler expects the service code to be in register $v0. Arguments are passed into and returned from the services using various other registers. See Appendix G.

QUESTION 10:

With your mental highlighter, mark the mnemonics in the code fragment which are *pseudoinstructions*.



Next

A good answer might be:

See Below.

Calculation Section

After reading the food cost, the program can do the calculations. It is convenient to do them all at once and then to print out the results.

```
## rest.asm
##
## Total restaurant bill calculator
##
## Register Use:
##
## $s0 tip+tax
## $s1 tip rate
## $s2 tax rate
## $s3 meal cost
## $a0 tax+tip dollars

        .globl  main

        # Get meal cost
main:   li      $v0,4       # print a prompt (code 4 put in $v0)
        la      $a0,prompt  # address of prompt put in $a0
        syscall

        li      $v0,5       # input an integer  (code 5 put in $v0)
        syscall             # 32-bit result returned in $v0
        move    $s3,$v0     # save it in $s3

        # Calculations
        lw      $s1,_____       # get tip rate

        lw      $s2,_____       # get tax rate

        addu    $s3,$s1,$s2     # total rate

        _____   $s4,$s0,$s3     # 100*(tax + tip)

        _____   $s4,$s4,100     # tax + tip
        _____  $s5,$s0,$s4      # total bill

       . . . . .

        .data
tip:    .word   15          # tip rate in percent
tax:    .word    8          # tax rate in percent

prompt: .asciiz "Enter food cost: "
head1 : .asciiz "   Tax plus tip: "
head2 : .asciiz "     Total cost: "
# end of file
```

QUESTION 11:

Fill in the blanks to complete the calculation section.

Next

A good answer might be:

```
# Calculations
lw      $s1,tip         # get tip rate
lw      $s2,tax         # get tax rate
addu    $s3,$s1,$s2     # total rate
mul     $s4,$s0,$s3     # 100*(tax + tip)
div     $s4,$s4,100     # tax + tip
addu    $s5,$s0,$s4     # total bill
```

Writting out the Results

The final section of the program writes the results to the monitor.

```
## Register Use:
##
##    $s0 meal cost
##    $s1 tip rate
##    $s2 tax rate
##    $s3 total rate
##    $s4 tax+tip dollars
##    $s5 total bill

        # Output

        ____    $v0,4           # print string

        ____    $a0,head1       # "tax plus tip"
        syscall

        ____    $a0,$s4         # get tax+tip
        li      $v0,1           # print integer
        syscall                 #

        ____    $v0,4           # print string

        ____    $a0,head2       # "total cost"
        syscall

        ____    $a0,$s5         # get total

        ____    $v0,1           # print integer
        syscall                 #
        li      $v0,10          # exit
        syscall

        .data
tip:    .word   15              # tip rate in percent
tax:    .word    8              # tax rate in percent

prompt: .asciiz "Enter food cost: "
head1 : .asciiz "   Tax plus tip: "
head2 : .asciiz "\n     Total cost: "
```

QUESTION 12:

Fill in the blanks to complete the program

Next

A good answer might be:

See below.

Complete Program

```
## rest.asm
##
## Total restaurant bill calculator
##
## Register Use:
##
##    $s0 meal cost
##    $s1 tip rate
##    $s2 tax rate
##    $s3 total rate
##    $s4 tax+tip dollars
##    $s5 total bill

        .globl  main

        # Get meal cost
main:   li      $v0,4           # print prompt
        la      $a0,prompt
        syscall
        li      $v0,5           # input meal cost
        syscall
        move    $s0,$v0         # save it in $s0

        # Calculations
        lw      $s1,tip         # get tip rate
        lw      $s2,tax         # get tax rate
        addu    $s3,$s1,$s2     # total rate
        mul     $s4,$s0,$s3     # 100*(tax + tip)
        div     $s4,$s4,100     # tax + tip
        addu    $s5,$s0,$s4     # total bill

        # Output
        li      $v0,4           # print string
        la      $a0,head1       # "tax plus tip"
        syscall

        move    $a0,$s4         # get tax+tip
        li      $v0,1           # print integer
        syscall                 #

        li      $v0,4           # print string
        la      $a0,head2       # "total cost"
        syscall

        move    $a0,$s5         # get total
        li      $v0,1           # print integer
        syscall                 #

        li      $v0,10          # exit
        syscall

        .data
tip:    .word   15              # tip rate in percent
tax:    .word    8              # tax rate in percent

prompt: .asciiz "Enter food cost: "
head1 : .asciiz "   Tax plus tip: "
head2 : .asciiz "\n     Total cost: "
# end of file
```

The program is complete and can be copied into your program editor for running and play. Remember to indicate in the SPIM settings to allow pseudoinstructions and to load the trap file.

QUESTION 13:

What is the bit pattern in register $t1 after the following:

```
li   $t0,0x12345678
sll  $t1,$t0,8
```

Next

What is the bit pattern in register $t1 after the following:

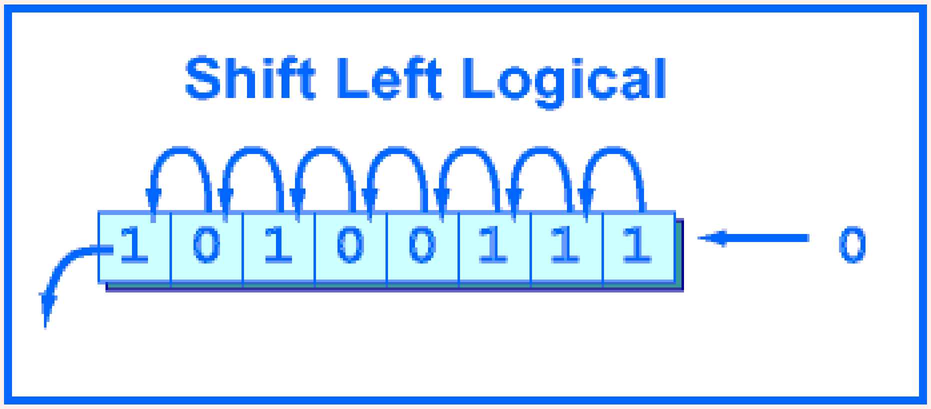
```
li   $t0,0x12345678
sll  $t1,$t0,8
```

A good answer might be:

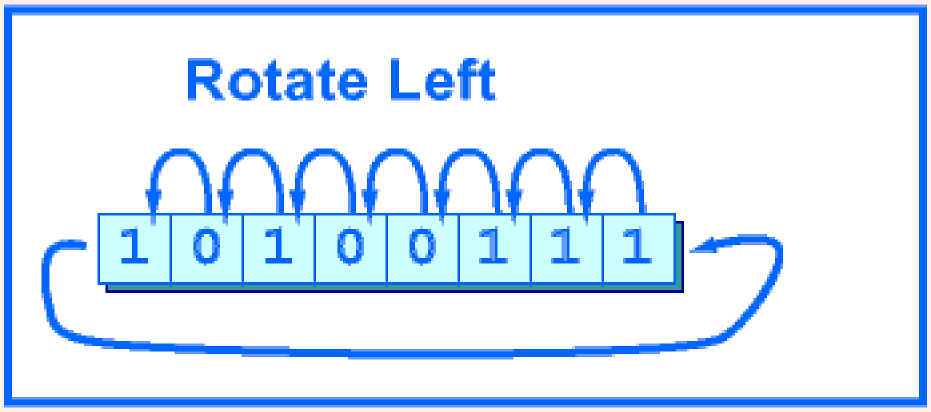
Register $t1 will contain 0x34567800

Bit Pattern Rotation

With a shift left instruction, the bits at the left end of the the register "fall off". The bits shifted into the right end are always zero bits.



With a rotate left instruction, the bits at the left end of the the register are fed into the right end of the register. The 32-bit register always contains the same 32-bit values it started with, but they are rotated into new positions.



In the illustration, after rotating left one bit the bit pattern changes from **10100111** to **01001111**. An additional left rotate changes the pattern to **10011110**.

Of course, registers have 32 bits, not the eight bits shown in the illustration.

A rotate right instruction, Works about the same, except that bit values in the register are moved right. Bits at the right end of the register are fed back into the left end of the register.

QUESTION 14:

Rotate right the following bit pattern by **two** bit positions:

11001100

Rotate right the following bit pattern by **two** bit positions:

```
11001100
```

A good answer might be:

```
00110011
```

Rotate Instructions

The rotate instructions are both pseudoinstructions. Each one takes four basic instructions to implement. The shift amount is given in a register, the third operand of the instruction.

```
rol d,s,t    # d <-- s rotated left by t
             # (pseudoinstruction)

ror d,s,t    # d <-- s  rotated right by t
             # (pseudoinstruction)
```

QUESTION 15:

Would a ror instruction ordinarily be used for arithmetic? For example, would...

```
li  $t2,2
ror $t0,$t1,$t2
```

...be a good way to divide $t1 by four?

```
li  $t2,2
ror $t0,$t1,$t2
```

A good answer might be:

No, because a rotate instruction is likely to rotate some low-order one bits into high order positions, which would greatly affect the result.

End of Chapter

You have reached the end of this chapter. Unless your head is rotating, click on a blue subject to go to where it was discussed. To get back here, click on the "back arrow" button of your browser.

- not pseudoinstruction
- Implicit immediate operands
- addu pseudoinstruction
- subu pseudoinstruction
- mul pseudoinstruction
- div pseudoinstruction
- remu pseudoinstruction
- negu pseudoinstruction
- abs pseudoinstruction
- Bit rotation.
- Rotate pseudoinstructions.

contents

You have reached the end of the chapter.

created: 06/05/2002

CHAPTER 24— Branch and Comparison Pseudoinstructions

The MIPS hardware has several instructions for branching and looping. These fundamental instructions are used in to create many pseudoinstructions that give added flexibility.

Chapter Topics:

- *Branch instructions*
- *Immediate operands in branch instructions*
- *Set instructions*
- *Indexed Addressing*
- *Byte arrays*
- *Integer arrays*

QUESTION 1:

(Review:) What is an conditional branch?

Next

A good answer might be:

An conditional branch tests a condition then changes the PC if the condition is true (for example, the instruction beq label).

Branch Equal to Zero

The extended assembler implements several conditional branch instructions beyond the basic ones. For example:

```
    beqz    rs,label          # branch to label if register rs==0
```

The hardware does not have a machine instruction for this operation. It does, however, have a zero register and the beq instruction (branch equal).

QUESTION 2:

Fill in the blanks so the branch is taken if $rs is zero.

```
    ______    rs,________,label         # branch to label if register rs==0
```

A good answer might be:

```
beq    rs,$0,label          # branch to label if register rs==0
```

Unconditional Branch Instruction

The pseudocomputer has an **unconditional branch** instruction that always causes a branch. The mnemonic for the instruction is b:

```
b     label    # branch to label
```

This instruction behaves like the jump instruction: it unconditionally loads the PC with the address specified by label. Unlike the jump instruction, the branch target label must be relatively close to the branch instruction, as is true with all branch instructions. The 32-bit address is computed using a 16 bit offset in the instruction and the current value of the PC.

QUESTION 3:

The extended assembler implements the b instruction using the beq instruction. Fill in the blanks to show what it does:

```
b  label        == beq  _____,_____,offset
```

Offset is the 16-bit offset that is added to the PC to calculate label (if possible).

A good answer might be:

```
b  label        == beq  $0,$0,offset
```

Table of Branch Instructions

Here is a table of some branch instructions. There are other branch instrucions concerned with subroutine linkage. Some instructions assume 32-bit two's complement data; others assume 32-bit unsigned data. Some instructions don't assume any data format.

The first operand s must be a register. The second operand t may be a register or a immediate operand (the extended assembler will generate the correct basic instructions depending on which you choose).

Mnemonic	Operands	Description	signed or unsigned?
b	label	branch	
beq	s,t,label	branch if s==t	
beqz	s,label	branch if s==0	
bge	s,t,label	branch if s>=t	signed
bgeu	s,t,label	branch if s>=t	unsigned
bgez	s,label	branch if s>=0	signed
bgt	s,t,label	branch if s>t	signed
bgtu	s,t,label	branch if s>t	unsigned
bgtz	s,label	branch if s>0	signed
ble	s,t,label	branch if s<=t	signed
bleu	s,t,label	branch if s<=t	unsigned
blez	s,label	branch if s<=0	signed

blt	s,t,label	branch if s<t	signed
bltu	s,t,label	branch if s<t	unsigned
bltz	s,label	branch if s<0	signed
bne	s,t,label	branch if s=/=t	
bnez	s,label	branch if s=/=0	

Some of the pseudoinstructions use the assembler temporary register ($at, or register $1) in the one or two basic instructions they are expanded to.

QUESTION 4:

What format of data is assumed for the bne (branch not equal) instruction?

Next

A good answer might be:

No assumptions are necessary. It tests if two bit patterns are identical.

Natural If-Else Structure

When you use only the basic branching instructions sometimes you have to implement an if-else structure with the false branch immediately following the conditional branch. This is not how high level programming languages do it and this sometimes is awkward. Here is a program fragment that is to add the value in register $t0 to register $t2 (if $t0 is even) and to $t1 (if $t0 is odd):

```
        lw    $t0,val         # $t0 has the value
        andi  $t8,$t0,1       # one's place is zero or one
        ____  $t8,odd         # if even
        addu  $t2,$t0,$t0     #     add to even sum
        b     endif
odd:                          # else
        addu  $t1,$t0,$t0     #     add to odd sum
endif:
```

The unconditional branch instruction is used at the bottom of the true branch of the if-else.

QUESTION 5:

Fill in the blank by choosing the correct branch instruction.

Next

A good answer might be:

```
        lw    $t0,val          # $t0 has the value
        andi  $t8,$t0,1        # one's place is zero or one
        bnez  $t8,odd          # if even
        addu  $t2,$t0,$t0      #     add to even sum
        b     endif
odd:                           # else
        addu  $t1,$t0,$t0      #     add to odd sum
endif:
```

Immediate Operand in Branch

If a branch instruction has a second operand, it can be an immediate operand or a register. For example, from the table:

bge	s,t,label	branch if s>=t	signed

Here are examples:

```
    bge    $t1,$t2,spot     # if ( $t1 >= $t2 ) goto spot

    bge    $t1,23,spot      # if ( $t1 >= 23  ) goto spot

    bge    $t1,-98,here     # if ( $t1 >= -98 ) goto here

    bge    12,$t1,oops      # WRONG: first op must be a register

    bge    $t1,value,there  # WRONG: second op can't be a symbolic address
```

Sometimes the mnemonic corresponds to a basic instrucion (for example beq). However if the second operand is an immediate operand then the result is a pseudoinstruction.

QUESTION 6:

Is the following instruction correct?

Next

```
        bge     $t1,-67,spot     # if ( $t1 >= -67 ) goto spot
```

A good answer might be:

Yes.

Example Program

Let us write a program that reads integers from the user and adds up those integers x that are in the range -32 <= x <= +32 and discards the rest. The user signals the end by entering -999 (which is not added to the sum). Here is the basic outline:

```
main:
        li    $v1,0          # zero the sum
loop:
        . . . .
        li    $v0,5          # read the integer
        syscall              # into $v0

        ____  $v0,____9,done   # while ( $v0 =/= -999 )
                             #
        ____  $v0,____,out     # less than -32

        ____  $v0,____,out     # greather than 32

        addu  $v1,$v1,$v0      # if in range add
                             # else skip
out:    b     loop

done:   . . . .
```

QUESTION 7:

Fill in the blanks

Next

A good answer might be:

See below.

Blanks Filled

The outline is completed below. Detecting the end of input (by testing for a sentinel value) is usually regarded as poor style.

```
main:
        li    $v1,0           # zero the sum
loop:
        . . . .
        li    $v0,5           # read the integer
        syscall               # into $v0

        beq    $v0,-999,done   # while ( $v0 =/= -999 )
                        #
        blt    $v0,-32 ,out        # less than -32
        bgt    $v0,32 ,out         # greather than 32
        addu  $v1,$v1,$v0         # if in range add
                              # else skip
out:    b     loop

done:   . . . .
```

QUESTION 8:

(Memory Test:) What do **set** instructions do on MIPS?

Next

A good answer might be:

Set instructions set a register to 1 or clear it to 0 to show the outcome of a comparison between two values.

Set on Less Than

Here (for review) is the slt from chapter 18:

```
                # $s and $t contain
                # two's comp. integers
                #
slt   d,s,t     # if ( $s < $t )
                #    d <-- 1
                # else
                #    d <-- 0
```

However, now that we are using the extended assembler, the operand t can be an immediate operand.

QUESTION 9:

Is the following correct?

```
slt   $t5,$v0,87       # if ( $v0 < 87 ) $t5 <-- 1
```

Next

A good answer might be:

Yes

Table of Set Instructions

Here is a table of set instructions. Most of them are pseudoinstructions. The t operand can be an immediate operand. The Imm operand must be an immediate operand.

The extended assembler outputs the correct basic instructions depending on the mnemonic and the operands. Sometimes there are several ways in which the same basic instructions can be specified.

Some of the instructions are intended for integers expressed in unsigned binary, other instructions are for integers in two's complement, and for others it makes no difference.

Mnemonic	Operands	Description	signed or unsigned?
seq	d,s,t	set d if s==t	
sge	d,s,t	set d if s>=t	signed
sgeu	d,s,t	set d if s>=t	unsigned
sgt	d,s,t	set d if s> t	signed
sgtu	d,s,t	set d if s> t	unsigned
sle	d,s,t	set d if s<=t	signed
sleu	d,s,t	set d if s<=t	unsigned
slt	d,s,t	set d if s< t	signed
slti	d,s,Imm	set d if s< Imm	signed
sltu	d,s,t	set d if s< t	unsigned
sltiu	d,s,Imm	set d if s< Imm	unsigned
sne	d,s,t	set d if s< t	

QUESTION 10:

Do you think that the following two pseudoinstructions translate into the same basic instructions?

Next

sltu $t4,$v0,45 and sltui $t4,$v0,45

sltu $t4,$v0,45 and sltui $t4,$v0,45

A good answer might be:

Yes. The first translates into the second.

Indexed Addressing

In addition to implementing new instructions, the extended assembler implements a new addressing mode. This is **indexed addressing**, a mode of addressing very useful for arrays. Here is an example:

```
        li    $t1,2                 # index 2
        lb    $v0,data($t1)         # $v0 = data[$t1]
        . . .

    data: .byte  6,34,12,-32, 90      # index zero is first
```

Think of the words beginning at data as an array of five bytes. Then the lw instruction loads the element of the array at index 2 (the byte that contains 12) into $v0.

The extended assembler does this the same way we have done in in past programs: basic instructions are used to add the index value in $t1 to the address symbolized by data. Here is what the assembler generates for the above code:

```
        ori   $t1,$0,2          # index 2
        lui   $at,4097          # $at register gets address "data"
        addu  $at,$at,$t1       # add index to $at
        lb    $v0,0($at)        # $v0 = data[$t1]
        . . .

    data: .byte  6,34,12,-32, 90
```

The assembler generates code that uses register $at to calculate the address of the correct byte. Then, using that address, the byte is loaded into $v0.

QUESTION 11:

When does the actual address calculation take place?

Next

A good answer might be:

When the code that the assembler generates is actually executing.

Four Levels

This is a very "Computer Science"-like idea. It takes some careful thought to see what is going on. Here is a table:

What the programmer writes	**What the extended assembler translates it into**	**What the basic assembler does**	**What happens at run time**

`li    $t1,2`
`lb    $v0,data($t1)`
	`ori   $t1,$0,2`
	`lui   $at,4097`
	`addu  $at,$at,$t1`
	`lb    $v0,0($at)`
	
	(4097 is the upper half of the address of data.)
		The basic assembly language is translated into machine code.
		
		`34090002`
		`3c011001`
		`00290821`
		`80220000`
			The first three machine instructions execute, placing the address of the third byte of the array into register $1. The fourth machine instruction load the byte at that address into register $2 ($v0).

QUESTION 12:

At what index do "C" and Java arrays begin?

A good answer might be:

Index zero

Indexes start at Zero

Experience has shown that indexing arrays starting at zero works best. The first element of an array is at a displacement of zero from the beginning of the array. To move through an array start the index at zero and increment it by the element size to move to the next element.

Here is a program fragment that adds up all the bytes in the array:

```
        li    $v1,0             # zero the sum
        li    $t1,0            # init index to 0
        li    $t2,0            # init loop counter
for:    beq   $t2,5,endfor      # for ( i=0; i < 5 ;i++ )
        lb    $v0,data($t1)
        addu  $v1,$v0,$v0       #    sum = sum+data[i]
        addi  $t1,$t1,1       #    increment index
        addi  $t2,$t2,1       #    increment counter
        b     for

endfor:
        . . .

data: .byte  6,34,12,-32, 90
```

QUESTION 13:

When indexed addressing is used with an array of 32-bit integers, by how much should the index be incremented to move to the next array element?

Next

A good answer might be:

By four.

Integer Array

Here is nearly the same program as before, except that now the program adds up the integers in an array of full words (32 bits). The logic is the same as before.

```
        .globl  main

main:
        li    $v1,0             # zero the sum
        li    $t1,_____         # init index to ???

        li    $t2,0             # init loop counter
for:    beq   $t2,5,endfor      # for ( i=0; i < 5 ;i++ )

        l___  $v0,array($t1)

        addu  $v1,$v0,$v0       #     sum = sum+array[i]

        addi  $t1,$t1,_____     #     increment index

        addi  $t2,$t2,_____     #     increment counter
        b     for

endfor:
        li    $v0,10            # exit
        syscall

        .data
array:  .word  1,2,3,-5,1
```

QUESTION 14:

Just fill in the blanks and you have a working program!

Next

A good answer might be:

See below.

Complete Program

Here is a complete program. It is not too distant from what a "C" for loop might be compiled into. If you copy it to a file and run it with SPIM make sure that pseudoinstrucions are allowed and that delayed load and delayed branches are turned off.

```
        .globl  main

main:
        li    $v1,0             # zero the sum
        li    $t1,0             # init index to 0
        li    $t2,0             # init loop counter
for:    beq   $t2,5,endfor      # for ( i=0; i < 5 ;i++ )
        lw    $v0,array($t1)
        addu  $v1,$v0,$v0       #    sum = sum+array[i]
        addi  $t1,$t1,4         #    increment index
        addi  $t2,$t2,1         #    increment counter
        b     for

endfor:
        li    $v0,10            # exit
        syscall

        .data
array: .word  1,2,3,-5,1
```

QUESTION 15:

Some languages (like Pascal) allow the first index of an array to be any integer. Can such a language be compiled for MIPS?

Next

A good answer might be:

Of course. The compiler might not use the indexed addressing mode of the extended assembler, however.

End of Chapter

You have reached the end this chapter. You may wish to review the following. Click on a subject that interests you to go to where it was discussed.

contents

You have reached the end of the chapter.

created: 06/06/02

CHAPTER 25 — The Run-time Stack

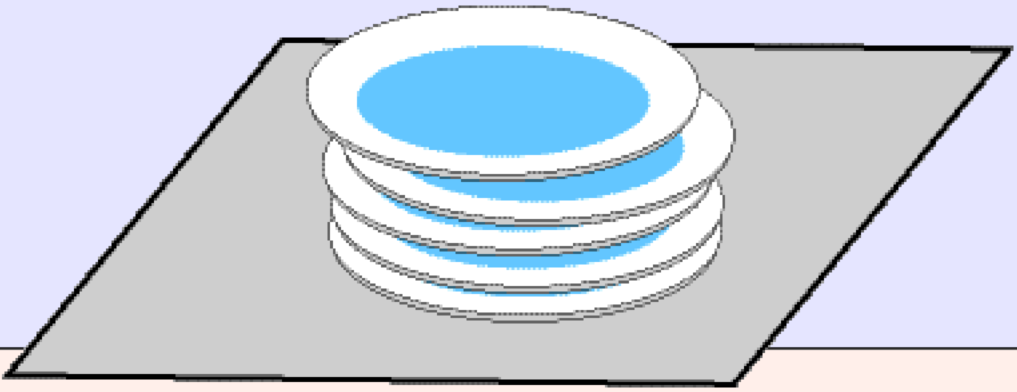
This chapter discusses the run-time stack and the stack pointer register $sp.

Chapter Topics:

- *Stacks*
- *The stack pointer register $sp*
- *Push and Pop stack operations.*
- *The MIPS runtime stack.*
- *Compiler use of stacks.*
- *String reversal example.*

QUESTION 1:

Say that you need to get a plate from a stack of dinner plates. Which one do you normally get?

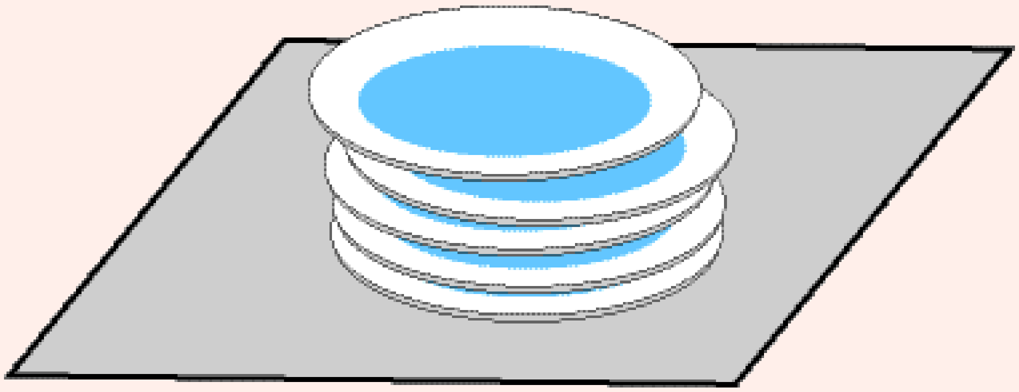


Next

A good answer might be:

The top plate.

Stack



A **stack** is a way of organizing data in memory. Data items are visualized as behaving like a stack of physical items. Often a stack is visualized as behaving like a stack of dinner plates. Data are added and removed from the stack data structure in a way analogous to placing plates on the stack of plates and removing them.

Normally with a stack of dinner plates all operations take place at the top of the stack. If you need a plate, you take the one at the top of the stack. If you add a plate to the stack, you place it on top of the stack.

QUESTION 2:

Which plate of a stack was the last one added that was added? Which plate will be the first one removed?

Next

A good answer might be:

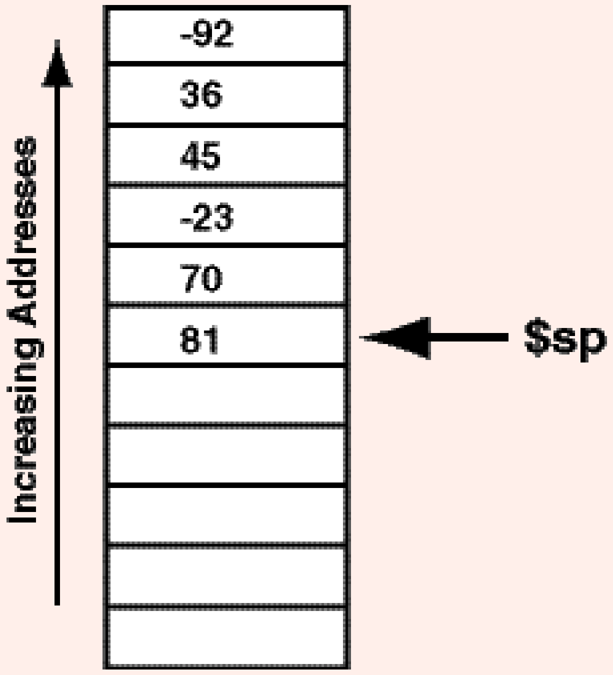
The plate at the top of the stack was the last one added. It will also be the first one removed.

Upsidedown MIPS Stack

Stack-like behavior is sometimes called "LIFO" for Last In First Out.

Our stacks are stacks of 32-bit words. In a course on data structures, stacks are used for all types of data. But for these chapters, stacks contain only 32-bit MIPS full words.

The picture shows a stack of MIPS full words. The **stack pointer register** $sp by convention points at the top item of the stack. The stack pointer is register $29 by software convention. The mnemonic register name $sp is used by the extended assembler.



In the usual way of drawing memory the stack is upside down. In the picture, the top item of the stack is 81. The bottom of the stack contains the integer -92.

Before the operating system starts your program it ensures that there is a range of memory for a stack and puts a suitable address into $sp.

QUESTION 3:

If an item of data (say the value 103) is added to the stack, where will it go?

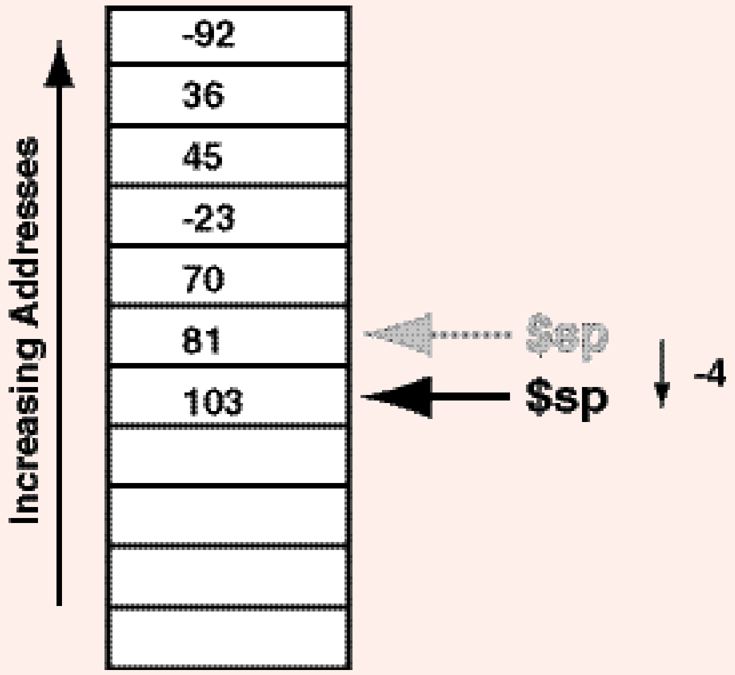
Next

A good answer might be:

After the 81. The 103 is now the top of the stack.

Push

By software convention, $sp always points to the top of the stack. Also by convention, the stack grows downward (in terms of memory addresses). So, for our stack of 4-byte (full word) data, adding an item means subtracing four from $sp and storing the item in that address. This operation is called a **push** operation.



> To **push** an item onto a stack, first **subtract** 4 from the stack pointer, then store the item at the address in the stack pointer.

Here is what that looks like in code. Say that the value to push on the stack is in register $t0:

```
                 # PUSH the item in $t0
sub $sp,$sp,4      # point to place for new item
sw  $t0,($sp)      # store $t0 as the new top
```

QUESTION 4:

In the new stack (above) if one item is removed, which will it be?

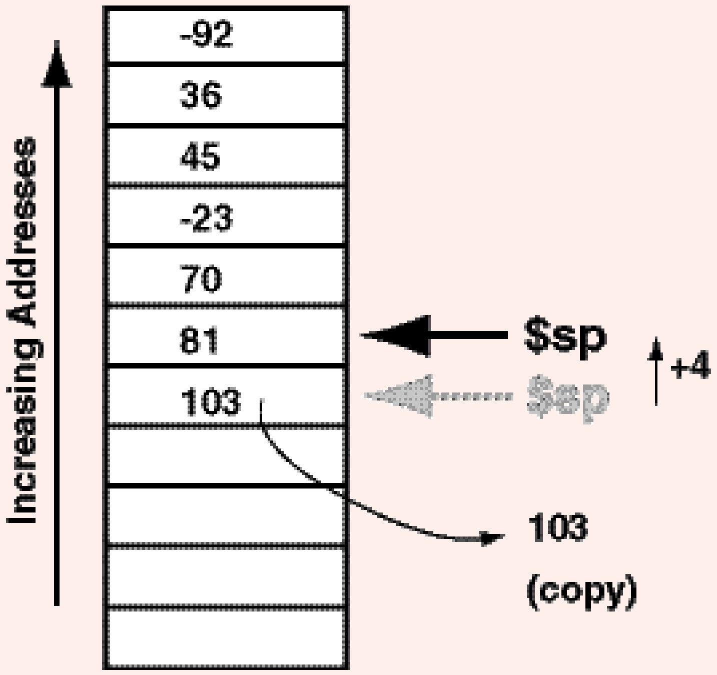
Next

A good answer might be:

103

Pop

Removing an item from a stack is called a **pop** operation. In the real-world analogy an item is actually removed: a dish is physically moved from the stack. In a software stack, "removal" of an item means it is copied to another location and the stack pointer is adjusted.



The picture shows a pop operation. The data is first copied from the top of stack to the new location and then the stack pointer is increased by four.

> To **pop** the top item from a stack, copy the item pointed at by the stack pointer, then **add** 4 from the stack pointer.

Here is what that looks like in code. Say that we want the value to be popped into $t0:

```
                    # POP the item into $t0
lw  $t0,($sp)       # copy top item to $t0
add $sp,$sp,4       # point item beneath old top
```

QUESTION 5:

When a software stack is popped, does the popped item remain in memory?

Next

A good answer might be:

Yes. The data is copied to a new location, but the old location is not changed. However, since the stack pointer is moved, "logically" the data is no longer on the stack.

Example

The stack is often used to hold temporary values when there are not enough registers. This is common in the code that a compiler outputs for evaluating arithmetic expressions. Here is an example of this. The program evaluates the expression ab - 12a + 18b - 7. Pretend that only $t0 and $t1 are available, and that the stack pointer $sp has been initialized properly by an operating system.

Here is the start of the program:

```
# Evaluate the expression ab - 12a + 18b - 7

main:
        lw      $t0,a          # get a
        lw      $t1,bb         # get b
        mul     $t0,$t0,$t1    # a*b

        sub     $sp,$sp,_____  # push a*b onto stack

        sw      $t0,______

        . . . . .

        .data
a:      2
bb:     3
```

QUESTION 6:

Fill in the blanks.

Next

A good answer might be:

See below:

Program Continued

Terms of the expression are pushed onto the stack as they are evaluated. Then the sum is initialized to -7 and the terms on the stack are popped and added to the sum.

```
# Evaluate the expression ab - 12a + 18b - 7

        .globl  main
        lw      $t0,a          # get a
        lw      $t1,bb         # get b
        mul     $t0,$t0,$t1    # a*b
        sub     $sp,$sp,4      # push a*b onto stack
        sw      $t0,($sp)

        lw      $t0,a          # get a
        li      $t1,-12        #
        mul     $t0,$t0,$t1    # -12a
        sub     $sp,$sp,4      # push -12a onto stack
        sw      $t0,($sp)

        lw      $t0,bb         # get b
        li      $t1,18         #
        mul     $t0,$t0,$t1    # 18b
        sub     $sp,$sp,4      # push 18b onto stack
        sw      $t0,($sp)

        li      $t1,-7         # init sum to -7

        lw      $t0,_____      # pop 18b

        add     $sp,$sp,______

        add     $t1,$t1,$t0    # 18b -7

        . . . .
```

QUESTION 7:

Fill in the blanks to pop the 18b term.

A good answer might be:

See below

Finished Program

Here is the finished program. It is not how you would write a practical program to evaluate the arithmetic expression. But it is similar to a section of a larger program (with many registers in use) that needs to do that and only has a few registers available.

```
# Evaluate the expression ab - 12a + 18b - 7
# SPIM settings: allow pseudo; load trap file;
#               no delayed branch; no delayed load

        .globl  main

main:
        lw      $t0,a          # get a
        lw      $t1,bb         # get b
        mul     $t0,$t0,$t1    # a*b
        sub     $sp,$sp,4      # push a*b onto stack
        sw      $t0,($sp)

        lw      $t0,a          # get a
        li      $t1,-12        #
        mul     $t0,$t0,$t1    # -12a
        sub     $sp,$sp,4      # push -12a onto stack
        sw      $t0,($sp)

        lw      $t0,bb         # get b
        li      $t1,18         #
        mul     $t0,$t0,$t1    # 18b
        sub     $sp,$sp,4      # push 18b onto stack
        sw      $t0,($sp)

        li      $t1,-7         # init sum to -7
        lw      $t0,($sp)      # pop 18b
        add     $sp,$sp,4
        add     $t1,$t1,$t0    # 18b -7

        lw      $t0,($sp)      # pop -12a
        add     $sp,$sp,4
        add     $t1,$t1,$t0    # -12a + 18b -7
        lw      $t0,($sp)      # pop ab
        add     $sp,$sp,4
        add     $t1,$t1,$t0    # ab - 12a + 18b -7

done:   li      $v0,1          # print sum
        move    $a0,$t1
        syscall
        li      $v0,10         # exit
        syscall

        .data
a:      .word  0
bb:     .word  10
```

QUESTION 8:

(Thought Question:) Is it possible to run out of memory if too many things are pushed onto the stack?

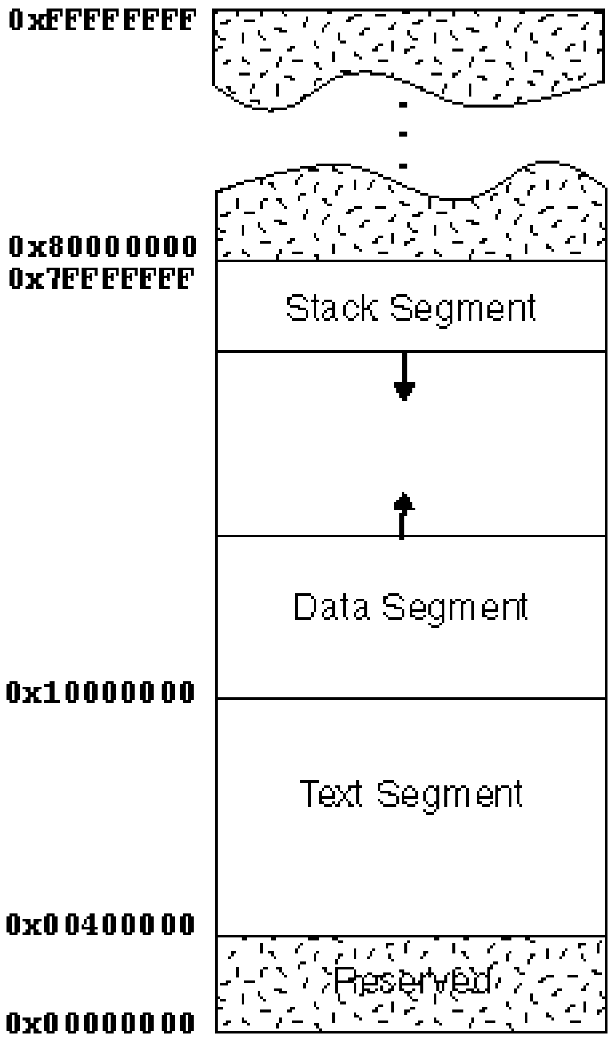
Next

A good answer might be:

Yes.

Run-time Stack

There is a finite amount of memory, even in the best computer systems. So it is possible to push more words than there are words of memory. Usually this would be the result of an infinite loop because when a program is first entered the operating system gives it space for a very large stack.



The picture shows how a typical operating system arranges memory when a program starts. There are four gigabytes of (virtual) memory available in total. The section of memory from 0x10000000 to 0x7FFFFFFF is available for the data segment and the stack segment. This is 1.8 Gigabytes of space.

When the program is started the stack pointer (`$sp`) is initialized to 0x7FFFFFFF. As the program runs, the stack grows downward into the available space. The data segment grows upward as the program runs. Of course, in a dynamic program, the segments grow and shrink. If the combined size of the segments exceeds the available space their boundaries will meet somewhere in the middle of the range. When this happens there is no memory left.

Another (inferior) way of arranging memory might be to have half the space 0x10000000 to 0x7FFFFFFF allocated to the stack and half the space allocated to data. But now the stack could grow too large for its allocated space, even if there was a tiny data segment that used little of its space.

QUESTION 9:

Which segment is for the machine instructions?

Next

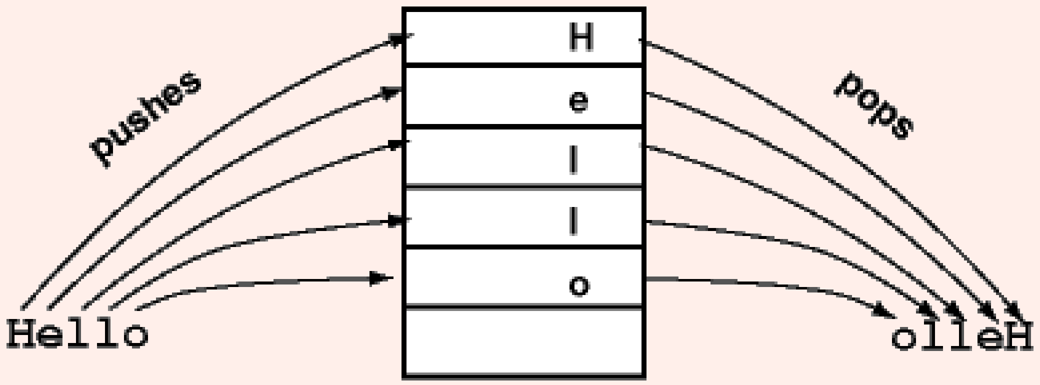
A good answer might be:

The text segment.

Reversing a String

"Text" is traditionally what a segment of machine instructions is called. It becomes a "process" when it starts executing. (This is analogous to the phrase "text of a play" and "performance of a play").

Almost all text books that describe stacks use this example program: the user enters a string. The program reverses the string and writes it out. To understand how the program works inspect the following diagram. The string "Hello" is pushed onto the stack, character by character. Then the characters are popped from the stack back into the original string buffer. This reverses the order of the characters.



We will always push and pop full words (four bytes). Each character on the stack will be contained in the low order byte of a fullword.

QUESTION 10:

(Review:) What does the following instruction do? lb $t0,string

Next

lbu $t0,string

A good answer might be:

It loads one byte (located at string) into the low order byte of register $t0. The other bytes are filled with zero.

Outline

Here is the outline of the program. The comments are for the major sections of the program.

```
        .text
        .globl  main
main:
        # ___________________________

        # ___________________________

        # ___________________________

        # ___________________________

        .data
str:    .space  128           # character buffer
```

Not much of an outline. Luckily, here are some phrases you can use to fill in the blanks: *print the reversed string*, *push each character onto the stack*, *input the string into a buffer*, and *pop chars from stack back into the buffer*.

QUESTION 11:

Fill the blanks with the correct phrases.

Next

A good answer might be:

```
main:
        # input the string

        # push each character onto the stack

        # pop chars from stack back into the buffer

        # print the reversed string
```

First Section

Here is the first secion of the program filled in. It merely reads in a line from the terminal in the usual way. There is no user prompt.

```
# Reverse and output a user-supplied string
#
# $t0 --- character pushed or popped
# $t1 --- index into string buffer str

        .text
        .globl  main

main:   #input the string
        li      $v0,8         # service code
        la      $a0,str       # address of buffer
        li      $a1,128       # buffer length
        syscall
                              # initialize the stack:
        li      $t0,_____     # push a null

        ____    $sp,$sp,4     # onto the stack

        sw      $t0,(___)     # to signal its bottom
        li      $t1,0         # index of first char in str buffer

        # push each character onto the stack

        # pop chars from stack back into the buffer

        # print the reversed string

        .data
```

```
str:    .space  128          # character buffer
```

Next the stack is initialized. Null is pushed onto the stack. Later on the stack will be popped until this null is encountered.

QUESTION 12:

Fill in the blanks of the program.

Next

A good answer might be:

See below.

Pushing Characters

```
# Reverse and output a user-supplied string
#
# $t0 --- character pushed or popped
# $t1 --- index into string buffer str

        .text
        .globl  main

main:   #input the string
        li      $v0,8          # service code
        la      $a0,str        # address of buffer
        li      $a1,128        # buffer length
        syscall

        li      $t0,0          # push a null
        sub     $sp,$sp,4      # onto the stack
        sw      $t0,($sp)      # to signal its bottom
        li      $t1,0          # index of first char in str buffer

        # push each character onto the stack
pushl:
        lbu     $t0,str($t1)   # get current char into
                               # a full word
        _____   $t0,stend      # null byte: end of string

        sub     $sp,$sp,4      # push the full word
        ___     $t0,($sp)      # holding the char

        add     $t1,1          # inc the index
        j       ______         # loop

stend:
        . . . . .

        # pop chars from stack back into the buffer

        # print the reversed string
```

```
          .data
str:    .space  128           # character buffer
```

In the next stage, characters from the character buffer are pushed one by one onto the stack. The first instruction (at pushl:) uses indexed addressing to load the current character from the buffer (str:) into the least significant byte of $t0.

Next, the current character is tested. If it is null (zero) then control branches out of the loop. Otherwise the character is pushed onto the stack. Then the process is repeated.

QUESTION 13:

Fill in the blanks.

Next

A good answer might be:

See below.

Popping Characters

```
        . . . . .
        # push each character onto the stack
pushl:
        lbu     $t0,str($t1)   # get current char into
                               # a full word
        beqz    $t0,stend      # null byte: end of string

        sub     $sp,$sp,4      # push the full word
        sw      $t0,($sp)      # holding the char

        add     $t1,1          # inc the index
        j       pushl          # loop

        # pop chars from stack back into the buffer
stend:  li      $t1,0          # index of first byte of str buffer
popl:
        ____    $t0,($sp)      # pop a char off the stack
        ____    $sp,$sp,4
        beqz    $t0,done       # null means empty stack

        ____    $t0,str($t1)   # store at string[$t1]
        add     $t1,1          # inc the index
        j       popl           # loop

        # print the reversed string
        . . . . .

        .data
str:    .space  128            # character buffer
```

When the null byte of the null-terminated input string is encountered, the first loop exits and the next loop begins. This next loop pops characters (contained in full words) off of the stack until the null at the bottom of the stack is encountered. Each character popped off the stack is placed into the string buffer, overwriting the character originally there.

The null at the end of the input string is not overwritten. It will remain there as part of the null-terminated result string.

QUESTION 14:

You know the drill: fill in those blanks.

Next

A good answer might be:

See below.

Final Phase

```
        . . . . .

        # pop chars from stack back into the buffer
stend:  li      $t1,0         # index of first byte of str buffer
popl:
        lw      $t0,($sp)     # pop a char off the stack
        add     $sp,$sp,4
        beqz    $t0,done      # null means empty stack

        sb      $t0,str($t1)  # store at string[$t1]
        add     $t1,1         # inc the index
        j       popl          # loop

        # print the reversed string
done:   li      $v0,4         # service code
        la      $a1,str       # address of string
        syscall
        li      $v0,10        # exit
        syscall

        .data
str:    .space  128           # character buffer
```

The last phase of the program prints out the result string. There is nothing new here. If you want to see the complete program, copy and past the several above sections into a text file.

QUESTION 15:

Would it be easier to do this program with arrays?

Next

A good answer might be:

Probably not. You would need two arrays. You would need to know the size of the input string before you begin copying from the input array to the output array.

End of the Chapter

You have reached the end this chapter. You may wish to review the following. Click on a subject that interests you to go to where it was discussed.

- Concept of stack.
- LIFO data structure.
- Stack pointer register and upsidedown MIPS stack.
- Push
- Pop
- Run-ime stack
- Text segment of MIPS memory.

contents

You have reached the end of the chapter.

created: 06/10/02

CHAPTER 26 — Simple Subroutine Linkage

All high level languages have the concept of a **subroutine** (sometimes called *module*, *procedure*, *function*, or *method*). A subroutine is a logical division of the code that may be regarded as a self-contained operation. A subroutine may be executed several times with different data as the program that contains it runs.

This chapter looks at a simple implementation in assembly language of this idea. The simple implementation is not adequate for the full power of subroutines (as implemented in high level languages), but is a good starting point. It corresponds to the type of subroutines implemented in early languages such as FORTRAN, BASIC, and COBOL.

Chapter Topics:

- *Subroutine call.*
- *Caller and Callee routines.*
- jal *and* jr *instructions*
- *The* $ra*register.*
- *The simple linkage calling convention.*
- *Register use in subroutines*

QUESTION 1:

In an assembly program, if you wish to unconditionally transfer control from one address to another, what instruction can you use?

Next

A good answer might be:

The j (jump) or the b (branch) instruction.

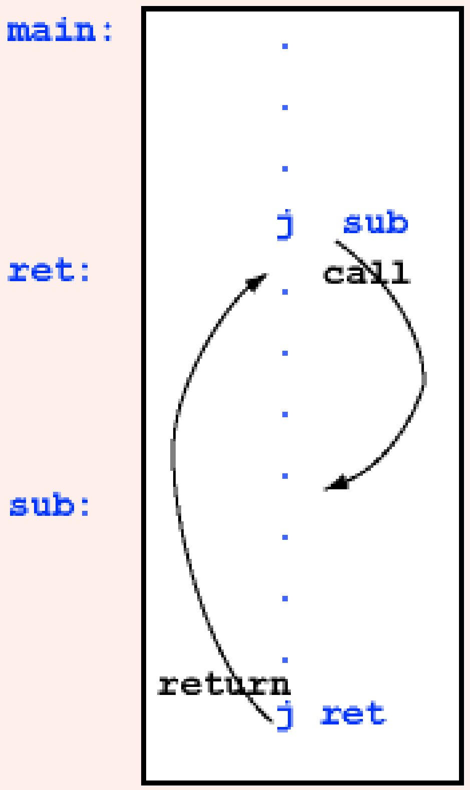
Callers and Callees

At right is a sketch of what you can do now. If the main routine needs to start up ("call") a subroutine sub it can jump to it with a j instruction. At the end of the subroutine, control can be returned with another j instruction.

In this type of linkage there must be a label in main where the subroutine is to return. The subroutine is written with the expectation that it is called at just one point in main and that it returns to an address a few words after that point.

The subroutine cannot be called from several locations in the main program because it must always return to the same location.

A subroutine **call** is when a main routine (or other routine) passes control to a subroutine. The main routine in this context is said to be the CALLER and the subroutine is said to be the CALLEE. A **return** from a subroutine is when a subroutine passes control back to its CALLER.



Subroutine Called Once

With modern software, when the CALLEE finishes execution it nearly always returns control to its CALLER.

QUESTION 2:

With modern software, during its execution might a CALLEE pass control to a subroutine other than its CALLER?

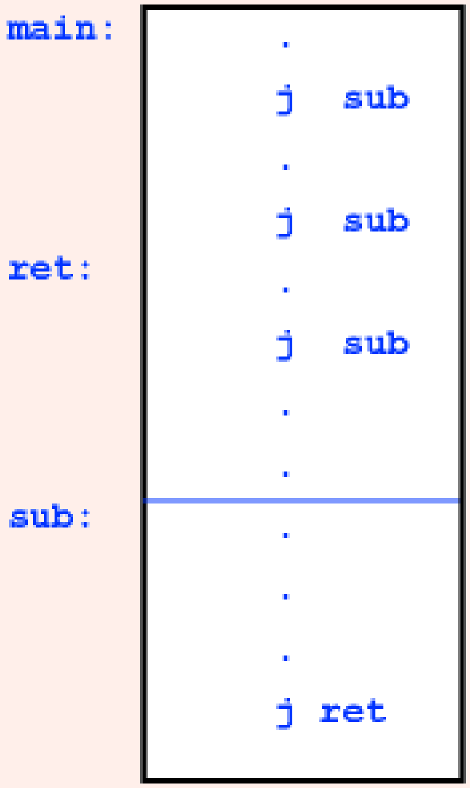
Next

A good answer might be:

Yes. Subroutines typically call other subroutines which call other subroutines.... and so on.

Many Calls but One Return

The problem is illustrated at right. The main routine is written to call a useful subroutine sub at several locations in the code. But sub is written to return to only one location. Usually this will not work.



Subroutine Called Three Times

In the past, before the concept was completely understood, hardware support for subroutines was missing. Various nasty tricks were used to implement the idea.

What is needed is a method that send the **return address** to the subroutine. When the subroutine finishes, it passes control to that return address.

Of course, passing control to a return address means to load the PC (program counter) with the return address. The next instruction fetch of the machine cycle will get the instruction from that address.

QUESTION 3:

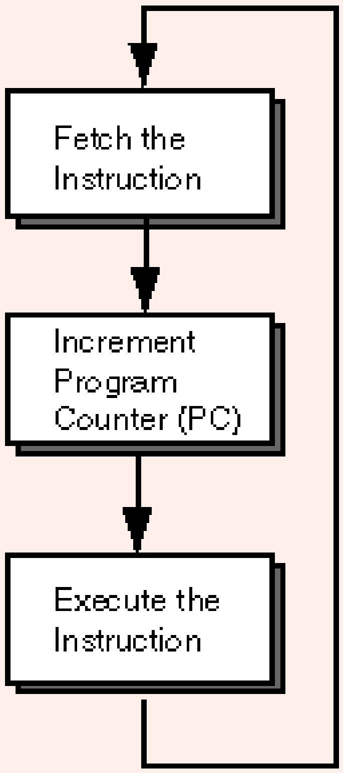
(Hardware Design Question:) How should the return address be passed to the subroutine? (i) By placing in in main memory somewhere, or (ii) By placing it in a register designated for this purpose.

Next

A good answer might be:

(ii) By placing it in a register designated for this purpose.

jal Instruction



The register that is used for linkage is register $31, which is called $ra by the extended assembler. It holds the *return address* for a subroutine. The instruction that puts the return address into $ra is (usually) the jal instruction.

To understand how jal works, review the machine cycle. The MIPS endlessly cycles through three basic steps. Each cycle executes one machine instruction. (This is a somewhat simplified view, but sufficient for now).

The jal instruction does the following (during the execute phase of the machine cycle):

```
jal sub:  $ra <• PC+4  # $ra <• address 8 bytes away from the jal
          PC <• sub   # load the PC with the entry point
```

Tricky: the middle step of the machine cycle has already incremented the PC by four (so that it points at the instruction after the jal instruction. Executing the jal instruction load this address plus four so that the $ra register points at the second instruction after the jal instruction.

The correct return address is "address of the jal plus four". This is because: (i) returning from the subroutine to the jal instruction would be a disaster (since it would execute again, sending control back to the subroutine), and (ii) the instruction following the jal in the branch delay slot.

QUESTION 4:

What instruction is usually placed in the branch delay slot?

Next

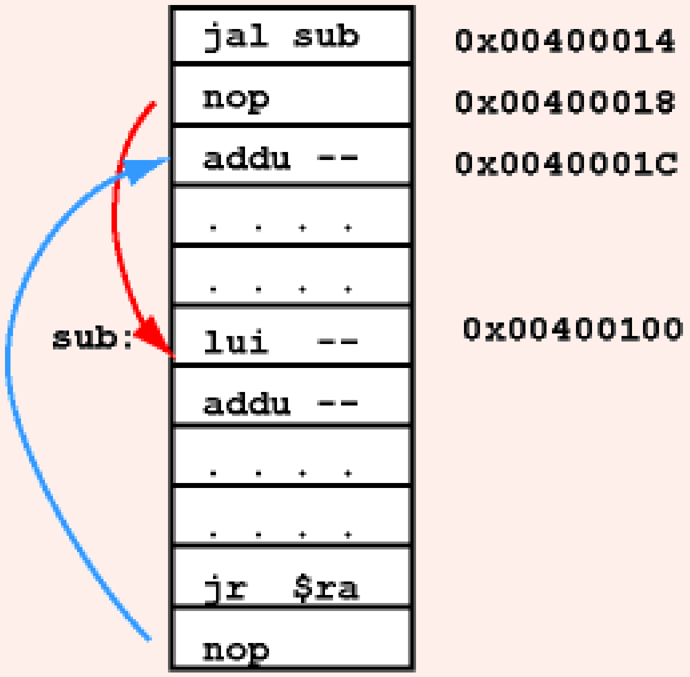
A good answer might be:

Usually the branch delay slot is filled with a nop instruction. This does nothing.

Example jal Instruction

It would not be a disaster to return to an instruction that does nothing. It is a slight optimization to return to the instruction after it. But sometimes clever programmers or clever compilers put something unexpected in the branch delay slot, so it is best not to pass control to it.

The diagram shows the execution of a jal instruction. The jal is at address 0x00400014. The return address is 0x0040001C which is the address of the jal plus eight. (The addu instruction is just there as an example of what might be at the return address).



Here is how the jal instruction works in general:

```
jal sub:  $ra <• PC+4  # $ra <• address 8 bytes away from the jal
          PC <• sub   # load the PC with the entry point
```

Here is how it works in this example. The entry point of sub: is 0x00400100

```
                    # when the jal is loaded the PC has 0x00400014
                    # next (in the machine cycle) the PC is increased
                    # to 0x00400018
jal sub:  $ra <• PC+4  # $ra <• 0x0040018+4
          PC <• sub   # load the PC with 0x00400100
```

QUESTION 5:

In the subroutine, the return address is in $ra. Can an ordinary jump instruction be used to return to the caller?

Next

A good answer might be:

No. An ordinary jump instruction has its one jump target encoded as an unchanging part of the instruction. (like j someSpot).

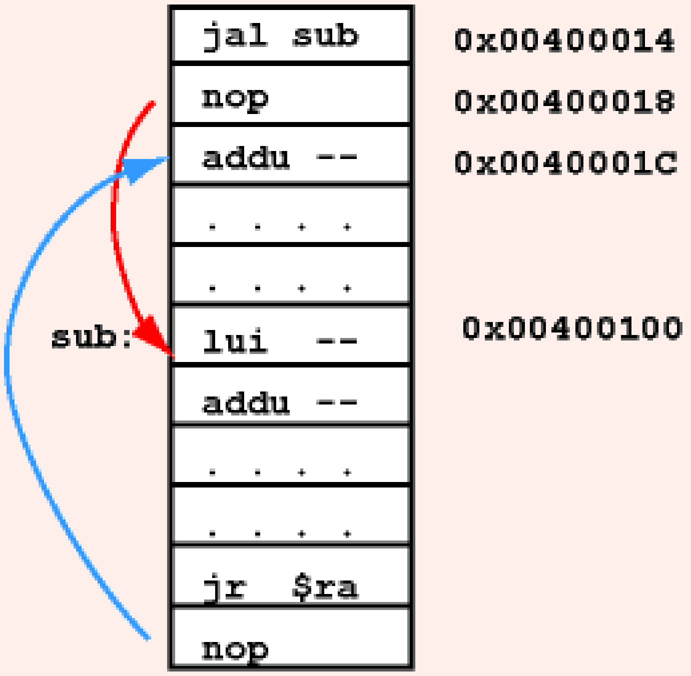
The jr Instruction

The jr instruction is used to return to the callee. It copies the contents of $ra into the PC:

```
jr  $ra      # PC <• $ra
```

To make the instruction more general, it can be used with any register, not just $ra. Like all jump and branch instructions, the jr instruction is followed by a branch delay.

The diagram shows the subroutine returning to the return address following the subroutine call.



QUESTION 6:

Do we now have a mechanism that enables us to call the same subroutine from many points in a program?

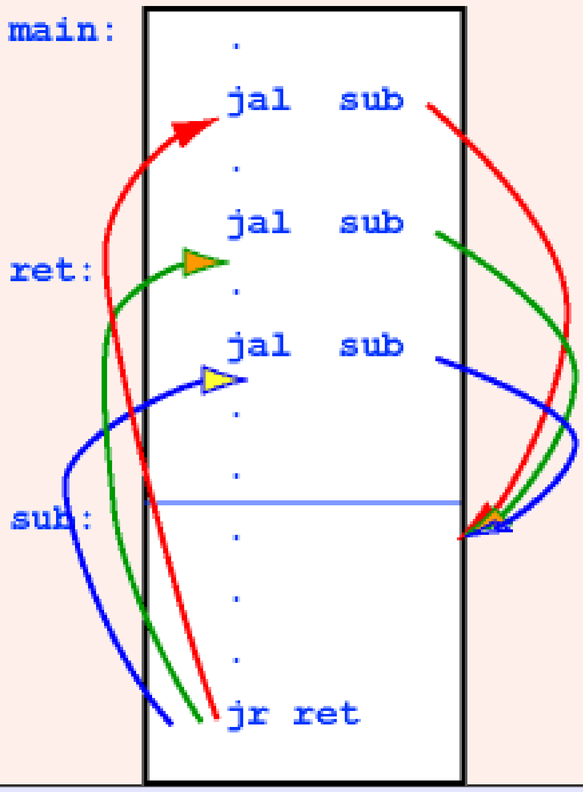
Next

A good answer might be:

Yes.

Calling Convention

The diagram shows the subroutine sub being called from three different points in the main routine. Each time the subroutine is called it returns to the correct return address.



Correct Subroutine Linkage

An agreement about how subroutines are called and how control is returned to the caller is called a **calling convention**. Mostly (as we will later see) this is an agreement about how software will work. In processors (like MIPS) that support subroutines, the convention says how those support features are used.

There is more to a calling convention than how to pass control, as the next question shows:

QUESTION 7:

What registers is the subroutine sub free to change?

Next

A good answer might be:

I don't know.

Register Use

To answer the question, you would have to look at the code for main and determine which registers could not be altered. Worse, you would have to look at the context surrounding each call to sub. One of the goals of subroutines is to create a module that is independent of the rest of the code. So far, that is not what we have achieved.

Another issue is how data is passed into and out of the subroutine. Often data is in registers, and the results are in registers. Which registers?

By agreement between programmers (not by hardware) registers have been assigned different roles with subroutine linkage:

- $t0 - $t9 — The subroutine is free to change these registers.
- $s0 - $t7 — The subroutine must not change these registers.
- $a0 - $a3 — These registers contain arguments for the subroutine. The subroutine can change them.
- $v0 - $v1 — These registers contain values returned from the subroutine.

QUESTION 8:

Is the following code fragment correct?

```
add   $t0,$s5,$s3   # calculate an important sum
jal   somesub       # call some subroutine
mul   $s4,$t0,$v1   # multiply the sum by the result
```

Next

A good answer might be:

No. The value in $t0 might have been changed by somesub, since $t0 (according to convention) is a register that a subroutine is free to use.

```
add    $t0,$s5,$s3   # calculate an important sum
jal    somesub       # call some subroutine
mul    $s4,$t0,$v1   # multiply the sum by the result
```

Simple Linkage

Let us agree on a calling convention. The convention is named *Simple Linkage*. (This is not the actual convention used by most MIPS software). Most of the rules of this convention you have already seen:

1. A subroutine is called using jal.
2. A subroutine will NOT call another subroutine.
3. The subroutine returns to its caller using jr $ra.
4. Register use is as follows:
   - $t0 - $t9 — The subroutine is free to change these registers.
   - $s0 - $t7 — The subroutine must not change these registers.
   - $a0 - $a3 — These registers contain arguments for the subroutine. The subroutine can change them.
   - $v0 - $v1 — These registers contain values returned from the subroutine.

QUESTION 9:

(Thought Question:) Rule number 2 is new. Why must not a subroutine call another subroutine?

Next

A good answer might be:

Because there is noplace to put the return address. The register $ra is already in use.

Pictoral Summary

The picture shows main linking to mySub. Two arguments are passed, in $a0 and #a1. The subroutine reads the arguments from those registers but should not change them.

There are some obvious limitations in the *Simple Linkage Convention*. They will be corrected in a complicated calling convention that is discussed in the next chapter.

Ideally, the caller and the callee should communicate only through arguments and return values. Both should be written without using any of the symbolic addresses in the other.

QUESTION 10:

Should a code module know about the inner workings of another module?

A good answer might be:

No. In assembly language this means routines should not use each other's symbolic addresses.

Example Program

Let us write a program that uses the *Simple Linkage* convention. The program is to read three integers from the user and compute the sum. The outline of the program is:

```
# read first integer

# read second integer

# read third integer

# compute the sum

# write out the result
```

Of course, the user will enter integers as characters from the keyboard. The program uses the trap handler service number five to read the characters and convert it to a full word.

QUESTION 11:

Examine the outline for the program. What do you think would be a useful subroutine?

A good answer might be:

It would be useful to have a *read integer* subroutine.

Prompt and Read Integer

The subroutine prompts the user for an integer and read it in. Here is a start on the subroutine:

```
# pread -- prompt for and read an integer
#
# on entry:
#    $ra -- return address
#
# on exit:
#    $v0 -- the integer

pread:
        la    $a0,prompt       # print string
        li    $v0,4            # service 4
        syscall

        li    $v0,5            # read int
        syscall                # service 5

        ____  ____             # return

        .data
prompt:
        .asciiz "Enter an integer"
```

QUESTION 12:

Fill in the blanks.

Next

A good answer might be:

```
# pread -- prompt for and read an integer
#
# on entry:
#    $ra -- return address
#
# on exit:
#    $v0 -- the integer

pread:
        la    $a0,prompt        # print string
        li    $v0,4             # service 4
        syscall

        li    $v0,5             # read int
        syscall                 # service 5

        jr    $ra               # return

        .data
prompt:
        .asciiz "Enter an integer: "
```

Main Program

Assume that the subroutine is correct. Now write the main program.

```
        .text
        .globl  main

main:
        jal     ______          # read first integer
        move    $s0,$v0         # save it in $s0
        jal     ______          # read second integer
        move    $s1,$v0         # save it in $s1
        jal     ______          # read third integer
        move    $s2,$v0         # save it in $s2

        addu    $s0,$s0,$s1     # compute the sum
        addu    $a0,$s0,$s2

        li      $v0,1           # print the sum
        syscall

        li      $v0,10          # exit
        syscall
```

QUESTION 13:

Fill in the blanks.

Next

A good answer might be:

```
        .text
        .globl  main

main:
        jal     pread           # read first integer
        move    $s0,$v0         # save it in $s0
        jal     pread           # read second integer
        move    $s1,$v0         # save it in $s1
        jal     pread           # read third integer
        move    $s2,$v0         # save it in $s2

        addu    $s0,$s0,$s1     # compute the sum
        addu    $a0,$s0,$s2

        li      $v0,1           # print the sum
        syscall

        li      $v0,10          # exit
        syscall
```

Global Symbols

Recall that modules (for us, subroutines) should not know about each other's symbolic addresses. It would violate the idea of modularity for main to do something to pread's prompt, for example.

But some symbolic addresses need to be used between modules. For example, pread is a symbolic address, and main must know about it and use it in the jal instruction.

A symbol that a subroutine makes visible to other subroutines is a **global** symbol. Global symbols usually label entry points used when other routines pass control (call) a subroutine. Symbols that are not global are called **local** symbols. In MIPS assembly and in SPIM global a symbol is made global by placing it in a list of symbols following the .globl directive:

```
        .globl  main
```

QUESTION 14:

What global symbols are in the subroutine pread?

Next

A good answer might be:

The entry point for the subroutine: pread.

Complete Program

Here is the complete example program. The global symbols have been correctly declared. Study how each module uses the directives .text and .data to describe its sections.

```
# addthree.asm --- read in three integers and print their sum
#
# This program uses simple linkage.  In SPIM, allow pseudoinstructions,
# disable delayed branches and disable delayed load, load trap file.
#
#
         .text
         .globl  main

main:
         jal     pread            # read first integer
         move    $s0,$v0          # save it in $s0
         jal     pread            # read second integer
         move    $s1,$v0          # save it in $s1
         jal     pread            # read third integer
         move    $s2,$v0          # save it in $s2

         addu    $s0,$s0,$s1      # compute the sum
         addu    $a0,$s0,$s2

         li      $v0,1            # print the sum
         syscall

         li      $v0,10           # exit
         syscall


# pread -- prompt for and read an integer
#
# on entry:
#    $ra -- return address
#
# on exit:
#    $v0 -- the integer
```

```
        .text
        .globl  pread
pread:
        la    $a0,prompt        # print string
        li    $v0,4             # service 4
        syscall

        li    $v0,5             # read int
        syscall                 # service 5

        jr    $ra               # return

        .data
prompt:
        .asciiz "Enter an integer: "
```

QUESTION 15:

Could pread be used as-is in other programs?

Next

A good answer might be:

Yes.

End of Chapter

You have reached the end this chapter. You may wish to review the following. Click on a subject that interests you to go to where it was discussed.

You have reached the end of the chapter.

created: 06/11/02

CHAPTER 27 — Stack-based Calling Convention

The Simple Linkage convention of the previous chapter lacked some features of high level languages. Some of these features are added in the Stack-based subroutine linkage explained in this chapter.

Chapter Topics:

- *Saving registers on the stack.*
- *The Stack-based Calling Convention.*
- *The prolog and epilog of the called subroutine.*
- *The call and return of the caller.*
- *Nested subroutine calls and the chain of activation.*
- *History of linkage conventions.*
- *Example program: converting user input to upper case.*

QUESTION 1:

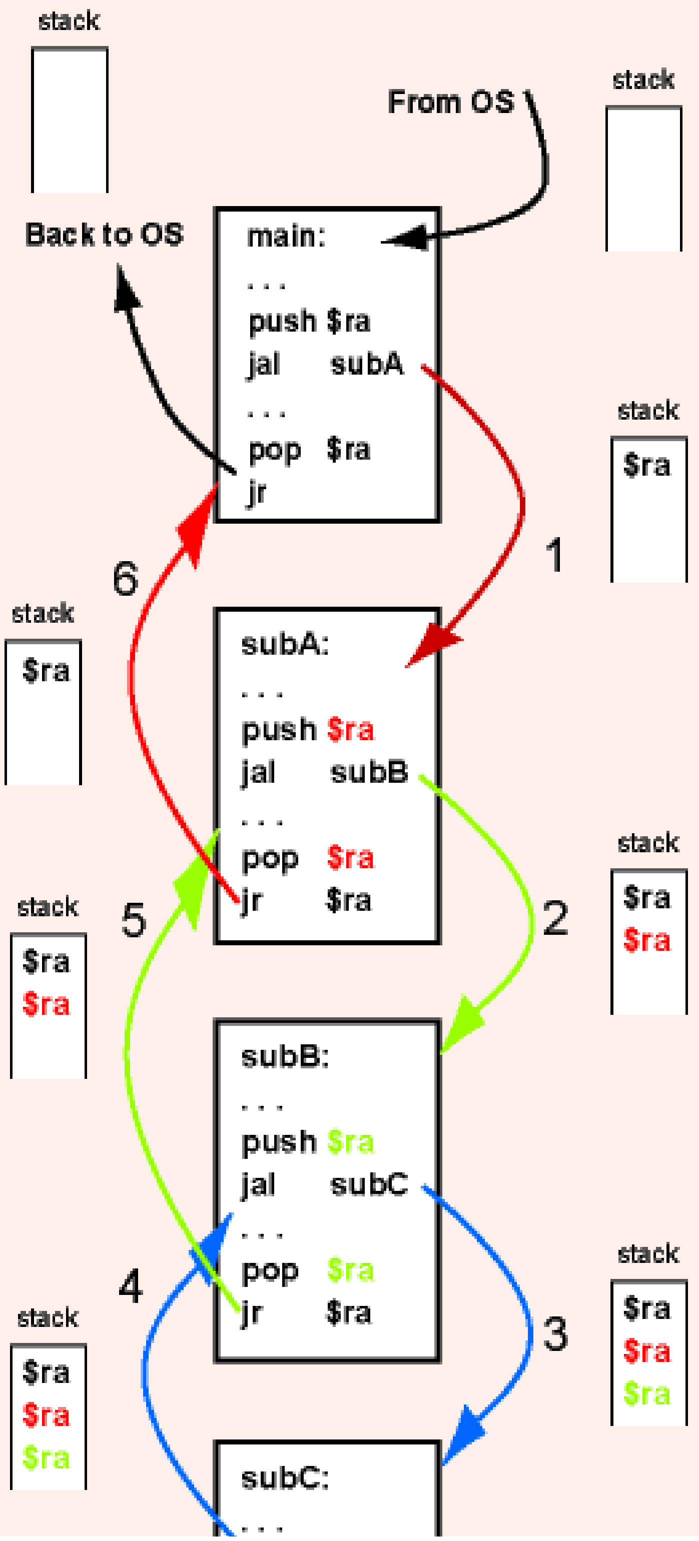
In the Simple Linkage convention of the previous chapter, can a subroutine call another subroutine?

Next

A good answer might be:

No.

Pushing the Return Address



To return to the caller a subroutine must have the correct address in $ra when the jr instruction is performed. But this address does not have to remain in $ra for all the time the subroutine is running. It works fine to save the value of $ra and then restore it when needed.

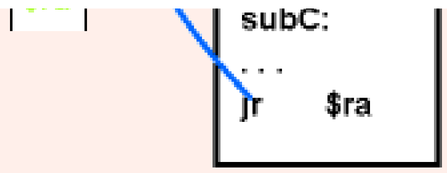
The picture shows main calling subA (step 1), which calls subB (step 2), which calls subC (step 3). Just before each call the current contents of $ra are pushed on the stack. Just before each return (using jr $ra) that value is popped back into $ra.

For this to work, each subroutine must return to its caller with the stack in the same state as when the caller did the jal instruction.

For example, main pushes its return address ($ra) onto the stack and calls subA. When main gets control again, the stack should have that return address on the top of the stack.

When main starts execution it has been called like a subroutine by the operating system. As far as the OS is concerned, main is a subroutine. So it has a return address (in $ra) to use when in finished execution (and returns control to the OS).

As another example, consider subB. When it starts execution the top of the stack is subA's return address ($ra for returning to main). When subB



calls subC, subB pushes its return address on the stack ($ra for returning to subA). Later on it pops this address when it returns to subA.

The push and pop operations are done in the usual way. There are no explicit push and pop operations as the diagram might imply. But some processors have explicit push and pop machine instructions.

QUESTION 2:

Why does not subC push its return address on the stack?

A good answer might be:

Because it calls no subroutine, $ra will not be changed, so it does not have to be saved on the stack.

Register Problem

In the Simple Linkage convention, registers $s0•$s7 could not be altered in a subroutine. This is a problem when subroutines call subroutines. Say that main calls subA and that subA calls subB. subA can't save any values in $s0•$s7 (because it is not allowed to alter them). But any values is saved in $t0•$s9 might be clobbered by subB (because subB is allowed to alter them). In effect, subA can't use any registers! Not good.

The solution is to allow a subroutine to use $s0•$s7. However, it must return them to their initial state when the subroutine returns to its caller.

QUESTION 3:

If a subroutine is to restore $s0•$s7 to their initial state, it must save the initial values somewhere. Where do you think those values are saved?

A good answer might be:

On the stack.

Pushing and Popping Registers

Here is a rule: if a subroutine alters any of the "S" registers, it must first push their values on the stack. Just before returning to the caller it must pop values from the stack into the proper registers. Of course, you have to be careful to do things in the correct order.

Here is an example program fragment. Subroutine subC calls subD which will use two of the "S" registers.

```
subC:
         . . . .

         sub    $sp,$sp,4    # push $ra
         sw     $ra,($sp)
         jal    subD         # call subD

         . . . .

# subD expects to use $s0 and $s1
subD:
         sub    $sp,$sp,4    # push $s0
         sw     $s0,($sp)
         sub    $sp,$sp,4    # push $s1
         sw     $s1,($sp)

         . . . .             # statements using $s0 and $s1

         lw     $___,($sp)   # pop _____
         add    $sp,$sp,4
         lw     $___,($sp)   # pop _____
         add    $sp,$sp,4

         jr     $ra          # return to subC
```

QUESTION 4:

Fill in the blanks so that subC sees its "S" registers when it regains control.

A good answer might be:

```
subC:
          . . . .

          sub    $sp,$sp,4    # push $ra
          sw     $ra,($sp)
          jal    subD         # call subD

          . . . .

# subD expects to use $s0 and $s1
subD:
          sub    $sp,$sp,4    # push $s0
          sw     $s0,($sp)
          sub    $sp,$sp,4    # push $s1
          sw     $s1,($sp)

          . . . .             # statements using $s0 and $s1

          lw     $s1,($sp)    # pop s1
          add    $sp,$sp,4
          lw     $s0,($sp)    # pop s0
          add    $sp,$sp,4

          jr     $ra          # return to subC
```

Notice that the registers are popped in the oppostite order that they were pushed.

Stack-based Calling Convention

The Simple Linkage Convention is easily extended into a *Stack-based Calling Convention*. This is not an official convention. However it is what you or a group of programmers might agree to use because it is not very complicated and does nearly everything you need. If you want to link assembly language routines to "C" or "C++" you need to use the full, official, linkage rules. Here are our simpler rules:

Calling a Subroutine (done by the caller):

1. The caller pushes onto the stack any registers $t0-$t9 that contain values that must be saved. The subroutine is free to change those registers.

2. The caller puts argument values into $a0-$a3.
3. Call the subroutine using jal.

Subroutine Prolog (at the start of the subroutine):

1. If this subroutine calls other subroutines, push $ra on the stack.
2. Push onto the stack any registers $s0-$s7 that the subroutine may alter.

Subroutine Body:

1. The subroutine may alter any "T" or "A" register or any "S" register that it saved in the prolog.
2. If the subroutine calls another subroutine, then it does so by following these rules.

Subroutine Epilog (at the end of the subroutine):

1. Return values are put in $v0-$v1
2. Pop from the stack (in reverse order) any registers $s0-$s7 that it previously pushed.
3. If it was pushed in the prolog, pop $ra from the stack.
4. Return to the caller using jr $ra.

Return from a Subroutine (done by the caller):

1. The caller pops from the stack (in reverse order) any registers $t0-$t9 that it previously pushed.

QUESTION 5:

Why do you think there are both "T" and "S" registers? Why not just have "S" registers and make it a rule that a subroutine must save each one it uses?

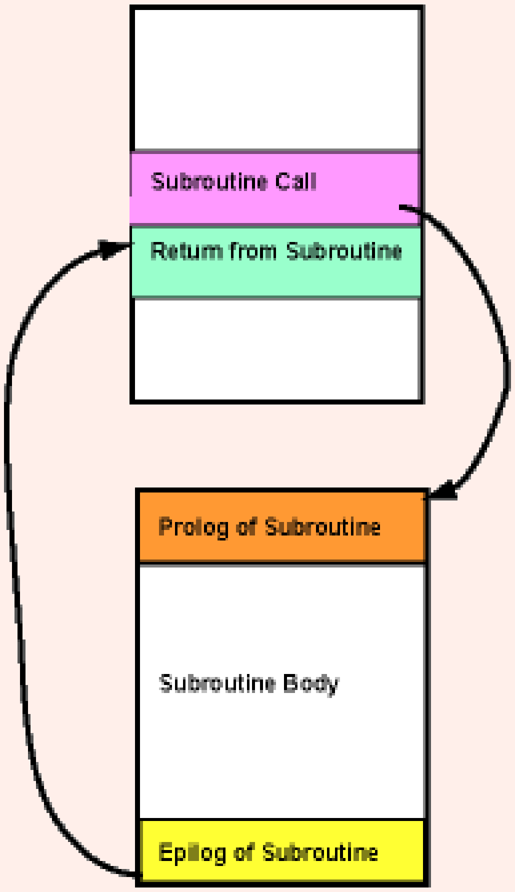
Next

A good answer might be:

This is an optimization for speed. Many subroutines neeed to use only a few registers. They can use the "T" registers without the added expense of saving and restoring them.

Diagram

Those rules are somewhat complicated. Here is a picture. It shows the four sections of subroutine linkage. The basic tasks of each section are:



Subroutine Call: Push any "T" registers that values that are needed. Put arguments in "A" registers. jal to the subroutine.

Prolog: If this subroutine calls other subroutines, push $ra. Push any "S" register the subroutine will alter.

Body: Normal code, except it must follow these conventions if it calls another subroutine. "T" and "A" registers can be used freely, as can any "S" registers that were saved in the prolog.

Epilog: Put return values in "V" registers. Pop any "S" registers. Pop $ra if it was pushed in the prolog. jr $ra back to the caller.

Return: Pop any "T" registers.

QUESTION 6:

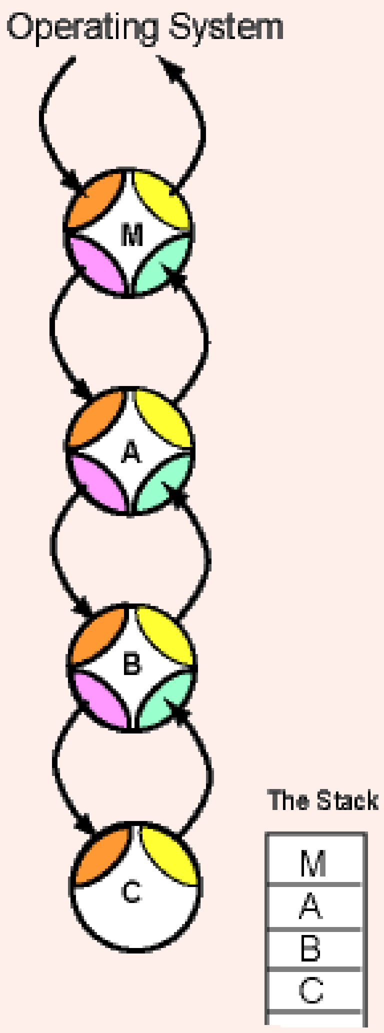
Is there any limit in these rules about how many levels deep subroutine calls may be?

Next

A good answer might be:

No.

Nested Subroutine Calls



The (somewhat silly) diagram at right shows the main routine linking to subroutine A, which links to subroutine B, which links to subroutine C. The subroutines link together like beads on a double string. Control is passed from the call to prolog, and from epilog to return. All but the subroutine at the bottom of a calling chain has all five sections (call, prolog, body, epilog, and return).

Each time another subroutine is added to the chain, more data is pushed onto the run-time stack. At the end of the chain of calls the run-time stack has a section of data (saved register values) from each of the subroutines (including main). The currently active subroutine is the one whose data is at the top of the stack (subroutine C, in our upside-down stack).

As each subroutine returns to its caller, its section of data is popped from the stack.

A subroutine does not "know" anything about the stack other than its section. It pushes its data onto the stack, and later on pops exactly that data back into registers. It does not look at any other data in the stack. It does not even know how deep the stack is.

Sometimes instead of saying "calling a subroutine" people say "activating a subroutine." Each bead in the picture and each section of the stack corresponds to one subroutine activation.

QUESTION 7:

After subroutine B returns control to subroutine A, might subroutine A call another subroutine (say subroutine X)?

A good answer might be:

Yes.

Linear Activation Chain

A particular subroutine (say "A") may call several other subroutines in succession (say "B", then "X", then "Y", then "Z"). But at any moment, it will have only one subroutine linked to it in the chain of activation. The calls in most modern programming languages follow this stack-based behavior. The currently executing subroutine (procedure, method, function, module, ...) is at the end of a linear chain of activations that link back to the operating system that first started "main".

When a subroutine reaches its end, it returns to its caller, and the chain is shortened. Its caller might call another subroutine, and the chain is lengthened again.

As a program made up of many subroutines executes, the activation chain grows up and down like a yo-yo. Ultimately the chain is of length zero when the main routine returns control to the operating system.

QUESTION 8:

Does your brain feel like a yo-yo?

Next

A good answer might be:

Uhh..

Programming Language History

Well,... this is not easy stuff. As proof of that statement, look to computer history. It took several decades before modern high level languages were established. The first one of any importance was Algol, created about 1960. Algol established stack-based subroutine linking. But Algol never quite caught on. Pascal (created about 1974) was a milestone. It became highly popular and used this stack-based idea of subroutines (which it called procedures and functions). Programming languages can be classified as "Before Pascal" and "After Pascal."

But let us return to the MIPS processor (created in the year 10 AP).

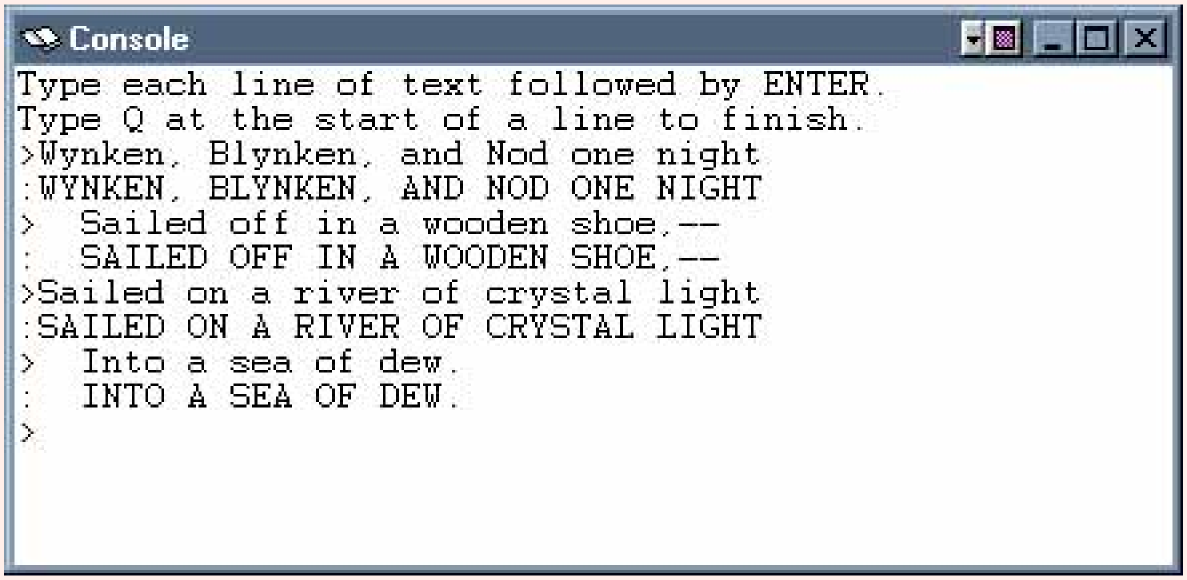
QUESTION 9:

Is the MIPS instruction set (its machine instructions) optimized for subroutine call and return?

A good answer might be:

Yes.

Example Program



```
Type each line of text followed by ENTER.
Type Q at the start of a line to finish.
>Wynken, Blynken, and Nod one night
:WYNKEN, BLYNKEN, AND NOD ONE NIGHT
>  Sailed off in a wooden shoe,--
:  SAILED OFF IN A WOODEN SHOE,--
>Sailed on a river of crystal light
:SAILED ON A RIVER OF CRYSTAL LIGHT
>  Into a sea of dew.
:  INTO A SEA OF DEW.
>
```

Some computers, such as the Digital Equipment Corporation (DEC) Vax, have specialized call and return machine instructions. But experimentally these add little to performance. The general purpose instructions of a reduced instruction set proved to be faster.

Here is an example program: the program is to read in lines of text from the user. Lower case characters from each line are converted to upper case. The user quits the program by entering a single character 'Q' at the start of a line.

QUESTION 10:

How can character 'a' be converted to character 'A'?

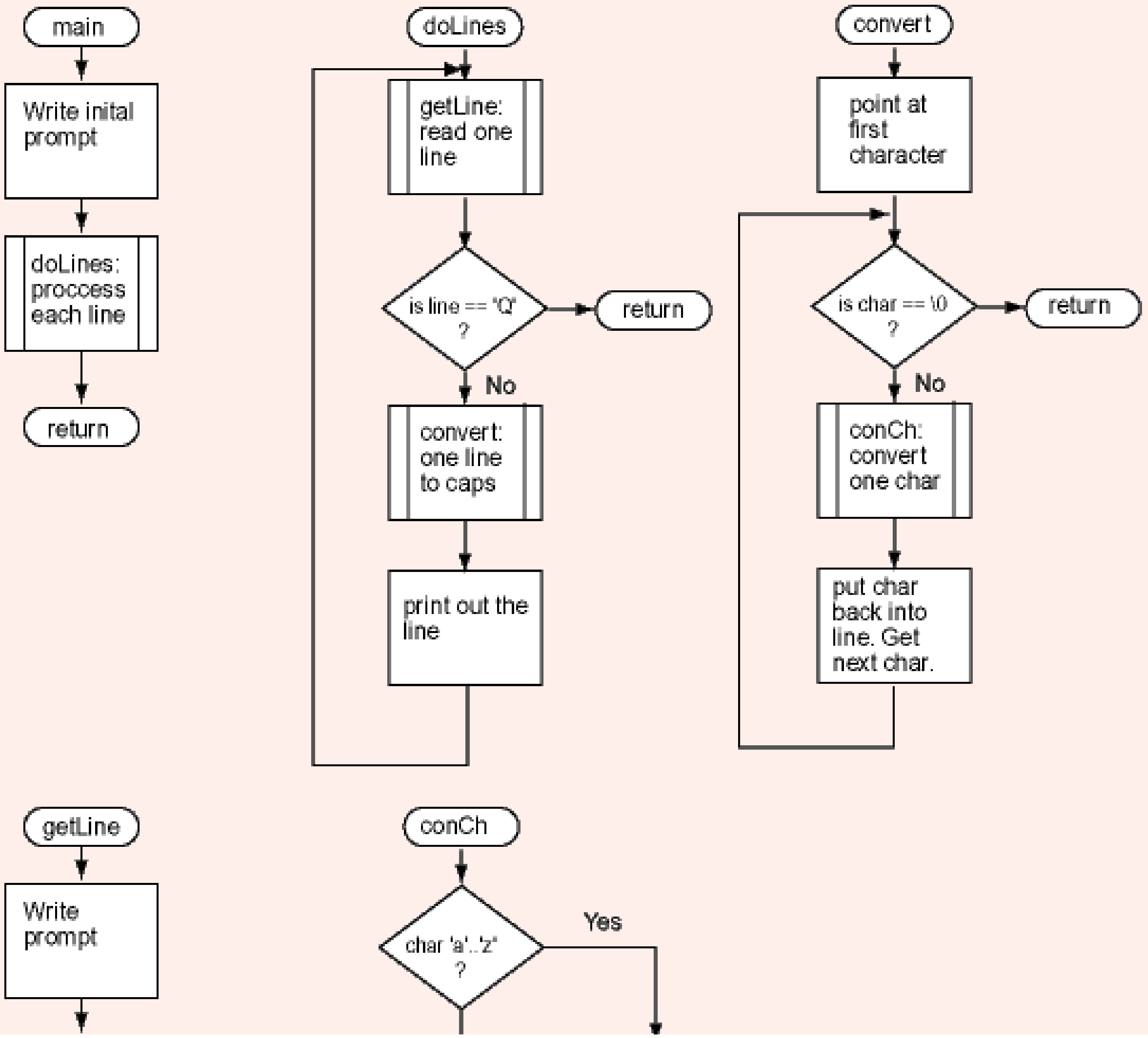
Next

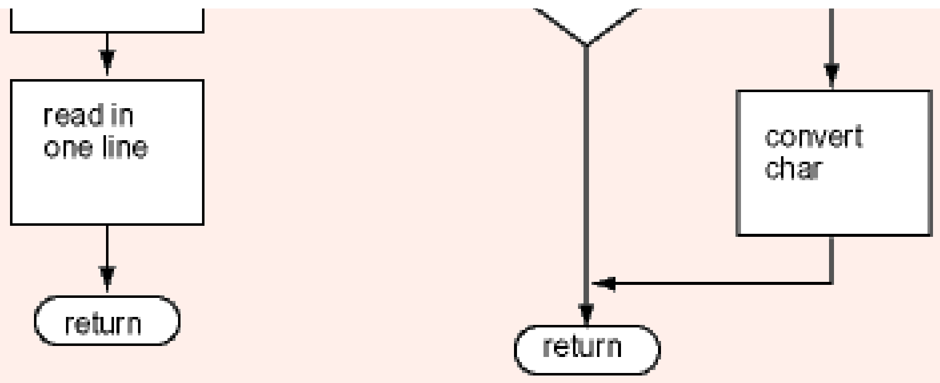
A good answer might be:

'a' - ('a' - 'A') == 'A', or, 'a' - 32 == 'A'. Subtracting 32 from a lower case character results in the corresponding upper case character (in ASCII).

Complete Program Design

Here is the complete design of the program. Glance over it to get the general idea. Its individual routines are explaind in the following pages. The design calls for more subroutines than usual, because the purpose is to show subroutine linkage.





A subroutine starts with a pill-shaped box that shows the name of the subroutine. A box with double vertical lines for its sides (like "doLines:" in main) designates a subroutine call. The program starts execution with "main".

QUESTION 11:

At the maximum, how many levels deep is subroutine nesting in this program?

Next

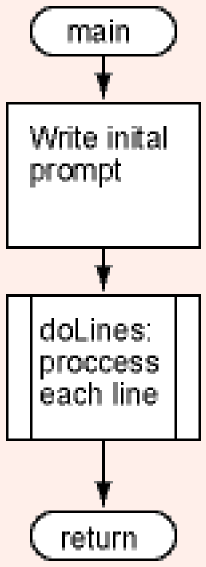
A good answer might be:

Four levels deep: main calls doLines which calls convert which calls conCh.

main Subroutine

An advantage of modular programming is that each subroutine can be displayed and explained independently of the others. At right is the design of main. Below is the code for it.

```
          .text
          .globl  main
main:
          ??????                  # what goes here?

          la      $a0,mainPr      # prompt the user
          li      $v0,4           # service 4
          syscall

          jal     doLines         # process lines of input

          ??????                  # what goes here?

          jr      $ra             # return to OS

          .data
mainPr:   .ascii  "Type each line of text followed by ENTER.\n"
          .asciiz "Type Q at the start of a line to finish.\n"
```



QUESTION 12:

According to the Stack-based Calling convention does main need to push and later pop the return address?

Next

A good answer might be:

Yes, because main calls a subroutine. The completed code for main is below.

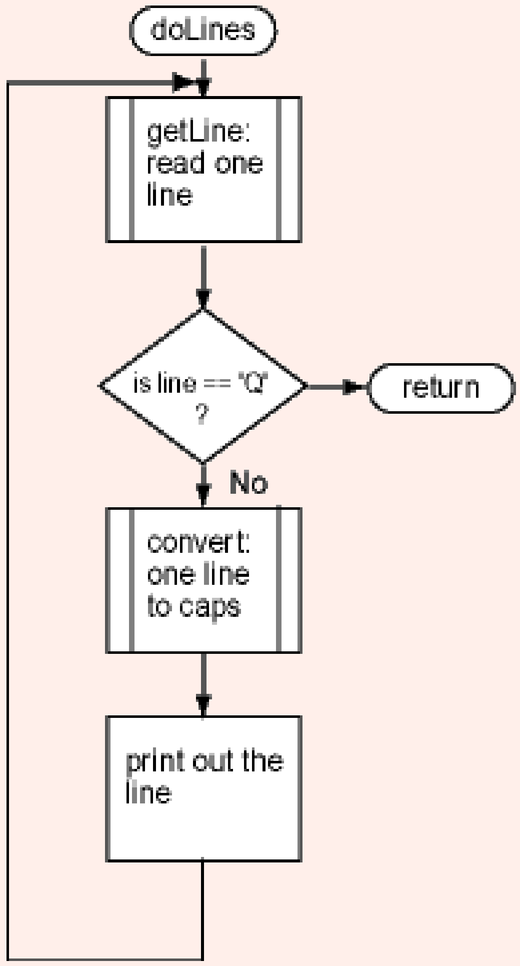
```
         .text
         .globl  main
main:
         sub     $sp,$sp,4        # push the return address
         sw      $ra,($sp)

         la      $a0,mainPr       # prompt the user
         li      $v0,4            # service 4
         syscall

         jal     doLines          # process lines of input

         lw      $ra,($sp)        # pop return address
         add     $sp,$sp,4

         jr      $ra              # return to OS

         .data
mainPr:  .ascii  "Type each line of text followed by ENTER.\n"
         .asciiz "Type Q at the start of a line to finish.\n"
```

Subroutine doLines

The main routine calls doLines. At right is the design for that routine. Below is its (incomplete) code.

```
# doLines -- read in and process each line of user input
#
# on entry:
#    $a0 -- address of the prompt text
#    $ra -- return address
#
# on exit:
#    no return values
```

```
        .text
        .globl  doLines
doLines:
        sub     $sp,$sp,4        # push the return address
        sw      $ra,($sp)

loop:                            # get a line
        la      $a0,_________    # argument: address of buffer
        li      $a1,____         # argument: length of buffer
        jal     ________         # get line from user

        la      $a0,line         # if "Q"
        jal     testEnd          # return to caller
        beqz    $v0,endloop

                                 # convert to capitals
        la      $a0,_________    # argument: address of buffer
        la      $a1,____         # argument: length of buffer
        jal     ________         # convert

        la      $a0,outline      # print out the result
        li      $v0,4
        syscall

        b       loop             # continue with next line

endloop:
        lw      $ra,($sp)        # pop return address
        add     $sp,$sp,4
        jr      $ra              # return to caller

        .data
outline: .ascii  ":"             # padding so output lines up with input
line:    .space  132             # input buffer
```



QUESTION 13:

No time like the present to fill in those blanks.

Next

A good answer might be:

The relevant section is filled in, below.

```
loop:                            # get a line
          la      $a0,line         # argument: address of buffer
          li      $a1,132          # argument: length of buffer
          jal     getline          # get line from user

          la      $a0,line         # if "Q"
          jal     testEnd          # return to caller
          beqz    $v0,endloop

                                 # convert to capitals
          la      $a0,line         # argument: address of buffer
          li      $a1,132          # argument: length of buffer
          jal     convert          # convert
```

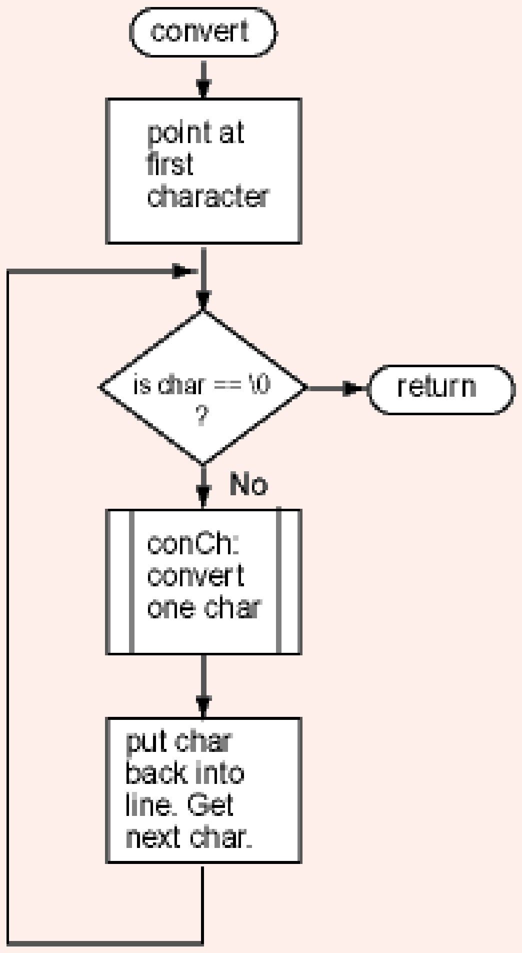
Subroutine convert

The first subroutine that doLines calls, getLine, is easy enough. Let us look at the more interesting convert. convert calls conChar for each character in the buffer. Since it calls a subroutine, it can't use a "T" register to hold the character pointer. But it must return the "S" registers to its caller in exactly the same state it got them!

This is the situation that calls for pushing an "S" register on the stack, using it in the subroutine body, and then popping it from the stack before returning to the caller.

```
# convert -- convert a line to all capitals
#
# on entry:
#    $a0 -- address of input buffer
#    $a1 -- length of input buffer
#
# register use:
#    $s0 -- pointer into character buffer
#
# on exit:
#    no return values
```

```
         .text
         .globl  convert
convert:
         sub     $sp,$sp,4        # push the return address
         sw      $ra,($sp)
         #   What should be done HERE??

                            # for ( p=buffer; *p!=0; p++ )
         move    $s0,$a0          # p=buffer

cloop:   lbu     $a0,($s0)        # get a char from the string
         beqz    $a0,endC         # exit if null byte
                            # argument a0: char to convert
         jal     conChar          # convert character
         sb      $v0,($s0)        # put converted char into string
         addu    $s0,$s0,1        # p++
         b       cloop

endC:
         #   And hat should be done HERE??
         lw      $ra,($sp)        # pop return address
         add     $sp,$sp,4
         jr      $ra              # return to caller
```



QUESTION 14:

Fill in the missing code.

Next

A good answer might be:

See below.

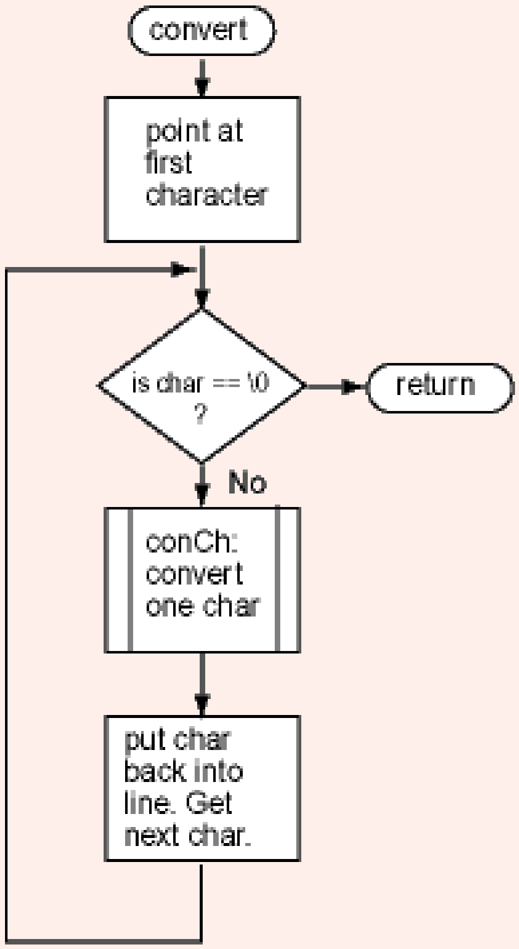
Complete Convert

convert uses $s0 as a character pointer because $a0 might be changed by conChar. You might think that it would be a good idea to look inside conChar to see if, in fact, it actually changes $a0. But this is a violation of modularity. It is much better to have a calling convention, and to follow it, than to make modules interdependant.

For example, conChar doesn't actually alter $a0. At least not now, but later on, conChar might be changed. Then we (or some unfortuante coder that got the job) would have to look everywhere conChar was used to see what assumptions were made.

```
# convert -- convert a line to all capitals
#
# on entry:
#    $a0 -- address of input buffer
#    $a1 -- length of input buffer
#
# register use:
#    $s0 -- pointer into character buffer
#
# on exit:
#    no return values
```

```
        .text
        .globl  convert
convert:
        sub     $sp,$sp,4        # push the return address
        sw      $ra,($sp)
        sub     $sp,$sp,4        # push $s0
        sw      $s0,($sp)

                             # for ( p=buffer; *p!=0; p++ )
        move    $s0,$a0          # p=buffer

cloop:  lbu     $a0,($s0)        # get a char from the string
        beqz    $a0,endC         # exit if null byte
                             # argument a0: char to convert
        jal     conChar          # convert character
        sb      $v0,($s0)        # put converted char into string
        addu    $s0,$s0,1        # p++
        b       cloop

endC:
        lw      $s0,($sp)        # pop $s0
        add     $sp,$sp,4
        lw      $ra,($sp)        # pop return address
        add     $sp,$sp,4
        jr      $ra              # return to caller
```



QUESTION 15:

Would it be a good idea to look into doLines to see if $s0 is important and actually needs to be saved?

Next

A good answer might be:

No.

Complete Capitalize Program

Here is the complete program. It would be useful to copy it to a file and to run it with SPIM.

```
# capitalize.asm --- convert user input to capitals and discard punctuation
#
# This program uses stack-based linkage.  SPIM Settings: allow pseudoinstructions,
# disable delayed branches and disable delayed load, load trap file.
#
        .text
        .globl  main

main:
        sub     $sp,$sp,4       # push the return address
        sw      $ra,($sp)

        la      $a0,mainPr      # prompt the user
        li      $v0,4           # service 4
        syscall

        jal     doLines         # process lines of input

        lw      $ra,($sp)       # pop return address
        add     $sp,$sp,4

        jr      $ra             # return to OS

        .data
mainPr: .ascii  "Type each line of text followed by ENTER.\n"
        .asciiz "Type Q at the start of a line to finish.\n"

# doLines -- read in and process each line of user input
#
# on entry:
#    $a0 -- address of the prompt text
#
# on exit:
#    no return values
```

```
        .text
        .globl  doLines
doLines:
        sub     $sp,$sp,4        # push the return address
        sw      $ra,($sp)

loop:                            # get a line
        la      $a0,line         # argument: address of buffer
        li      $a1,132          # argument: length of buffer
        jal     getLine          # get line from user

        la      $a0,line         # if "Q"
        jal     testEnd          # return to caller
        beqz    $v0,endloop

                                 # convert to capitals
        la      $a0,line         # argument: address of buffer
        la      $a1,132          # argument: length of buffer
        jal     convert

        la      $a0,outline      # print out the result
        li      $v0,4
        syscall

        b       loop             # continue with next line

endloop:
        lw      $ra,($sp)        # pop return address
        add     $sp,$sp,4
        jr      $ra              # return to caller

        .data
outline: .ascii  ":"             # padding so output lines up with input
line:    .space  132             # input buffer

# getLine -- read in a line of user input
#
# on entry:
#    $a0 -- address of input buffer
#    $a1 -- length of buffer
#
# on exit:
#    no return values

        .text
        .globl  getLine
getLine:
        move    $t0,$a0          # save buffer address
        la      $a0,prompt       # prompt the user
        li      $v0,4           # service 4
        syscall

        move    $a0,$t0         # restore buffer address
        li      $v0,8           # service 8
        syscall                 # read in a line to the buffer

        jr      $ra             # return to caller

        .data
prompt:
        .asciiz ">"

# testEnd -- check if a line is 'Q'
#
# on entry:
#    $a0 -- address of input buffer
#
# on exit:
#    $v0 -- 0 if line is equal to Q, 1 if not

        .text
        .globl  testEnd
testEnd:
        li      $v0,1           # assume not equal

        lbu     $t0,0($a0)      # get first char of line
        li      $t1,'Q'         # get 'Q'
        bne     $t0,$t1,endT    # if not equal, end the test

        lbu     $t0,1($a0)      # get second char of line
        li      $t1,'\n'        # it should be CR
        bne     $t0,$t1,endT    # if not equal, end the test

        li      $v0,0           # 'Q' has been found

endT:
        jr      $ra             # return to caller

# convert -- convert a line to all capitals
#
# on entry:
#    $a0 -- address of input buffer
#    $a1 -- length of input buffer
#
# register use:
#    $s0 -- pointer into character buffer
#
# on exit:
#    no return values
```

```
        .text
        .globl  convert
convert:
        sub     $sp,$sp,4       # push the return address
        sw      $ra,($sp)
        sub     $sp,$sp,4       # push $s0
        sw      $s0,($sp)

                                # for ( p=buffer; *p!=0; p++ )
        move    $s0,$a0         # p=buffer

cloop:  lbu     $a0,($s0)       # get a char from the string
        beqz    $a0,endC        # exit if null byte
                                # argument a0: char to convert
        jal     conChar         # convert character
        sb      $v0,($s0)       # put converted char into string
        addu    $s0,$s0,1       # p++
        b       cloop

endC:
        lw      $s0,($sp)       # pop $s0
        add     $sp,$sp,4
        lw      $ra,($sp)       # pop return address
        add     $sp,$sp,4
        jr      $ra             # return to caller

# conChar -- convert a character to a capital
#
# on entry:
#    $a0 -- character
#
# on exit:
#    $v0 -- converted character

        .text
        .globl  convert
conChar:
        move    $v0,$a0         # assume no change
                                # is ch in 'a' .. 'z' /
        li      $t0,'a'         # ch < 'a' ?
        blt     $a0,$t0,outc
        li      $t0,'z'         # 'z' < ch ?
        blt     $t0,$a0,outc
        sub     $v0,$a0,32      # convert to upper case

outc:   jr      $ra             # return to caller
```

QUESTION 16:

In which subroutine is the buffer for the input characters? Is a copy of those characters made when other subroutines operate on them?

Next

A good answer might be:

The string buffer line is defined in doLines. No copy is made of the characters in the buffer. However, other subroutines have access to the buffer since its address is passed as an argument (in $a0) to them. This technique is very common in "C" although, of course, the syntax for doing it is different.

End of the Chapter

You have reached the end this chapter. You may wish to review the following. Click on a subject that interests you to go to where it was discussed.

contents Go to Table of Contents

You have reached the end of the chapter.

created: 06/22/02

CHAPTER 28 — Frame-based Calling Convention, Variables, and Recursion

This chapter builds further on the stack-based calling convention to create a frame-based calling convention. A **stack frame** is the section of the run-time stack that holds the data of a subroutine.

Chapter Topics:

- *Stack frames and the frame pointer.*
- *Local variables.*
- *Simplified Frame-based Calling Convention.*
- *Compiler optimizations.*
- *Recursion.*
- *Storage classes: static, automatic, dynamic.*
- *Entry points.*

QUESTION 1:

(Thought Question:) What is a *local variable* in a higher level language?

Next

A good answer might be:

A *local variable* is a software entity that holds values for a subroutine while the subroutine is active.

Implementation of Local Variables

In a high-level language a **local variable** is implemented as a location on the part of the run-time stack the subroutine is using. Each activation of a subroutine pushes new data on the stack. The section of the stack for each activation is called a **stack frame** or an **activation record.** The **frame pointer** holds the address of the stack frame for a subroutine.

The format of a stack frame used by MIPS language processors is complicated. There are many situations that must be handled and many optimizations. It takes a compiler to do it correctly. These notes describe a much simplified stack frame.

The important part is to understand what a local variable is: a location on the run-time stack. This is an important idea in Computer Science, one you will run into repeatedly as you study advanced topics.

QUESTION 2:

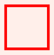
In a high-level language are there variables that are not local?

Next

A good answer might be:

Yes.

Picture of a Stack Frame



Programming languages have global variables. Also, an inner program block can use variables (that are not its local variables) which are defined in a containing block. Let us skip these messy details and implement local variables, only. The details may be convered in a course on compilers or in a course on programming languages.

The picture shows what a stack frame looks like when a subroutine is active (in our simplified format). As always (for us), each item on the stack is four bytes long.

As previously, the caller saves those "T" registers that contain values it needs, and the callee (the subroutine) saves those "S" registers it will change. But now room is pushed onto the stack for (in the example) four local variables `a`, `b`, `i` and `j`.

While the subroutine is active, the frame pointer, register $30, points to the top of the stack.

QUESTION 3:

Write the instruction that loads register `$t5` with the value held in the variable `a`:

```
lw $t5,____( ____ )
```

A good answer might be:

lw $t5,12($fp)

Frame Pointer

Register $30 is reserved, by software convention, for use as a **frame pointer**. In the extended assembler it has the mnemonic name . When a subroutine starts running the frame pointer and the stack pointer contain the same address. But the stack (and the stack pointer) may be involved in arithmetic expression evaluation. To make things easy for compilers (and for human assembly programmers) it is convenient to have a frame pointer that does not change its value while a subroutine is active.

When a routine calls a subroutine, the routine's frame pointer is pushed onto the stack along with the other caller-saved registers. Now the subroutine sets the frame pointer to the address of its stack frame.

QUESTION 4:

When a subroutine first starts executing, is the address in the frame pointer equal to the address of the top of the stack?

Next

A good answer might be:

Yes. However, the stack may grow as the subroutine executes. But the frame pointer for the subroutine does not change. The stack pointer ($sp) is always equal to the top of stack.

Sample Code

Imagine that the following statement is part of the subroutine whose stack frame is at right:

```
a = b + i + j;
```

This is how a compiler might implement that statement:

```
lw    $t0,8($fp)     # get b
lw    $t1,4($fp)     # get i
lw    $t2,0($fp)     # get j
addu  $t3,$t0,$t1    # b + i
addu  $t3,$t3,$t2    # b + i + j
sw    $t3,12($fp)    # a =
```

QUESTION 5:

Play compiler: translate the following statement into assembly language:

```
a = a + 1;
```

a = a + 1;

A good answer might be:

```
lw      $t0,12($fp)    # get a
addiu   $t0,$t0,1      # a + 1
sw      $t0,12($fp)    # a =
```

Our Frame-based Calling Convention

A real-world calling convention allows variables other than integers, array variables, parameters of various sorts, parameters on the stack, and other situations. Our rules are simpler:

Calling a Subroutine (done by the caller):

1. The caller pushes any registers $t0-$t9 that contain values that must be saved. Push the registers in numerical order.
2. Put argument values into $a0-$a3.
3. Call the subroutine using jal.

Subroutine Prolog (at the start of the subroutine):

1. Push $ra.
2. Push the caller's frame pointer $fp.
3. Push any of the registers $s0-$s7 that the subroutine might alter.
4. Initialize the frame pointer: $fp = $sp - space_for_variables. The "space for variables" is four times the number of local variables. (Remember that using "-" grows the stack).
5. Initialize the stack pointer: $sp = $fp.

Subroutine Body:

1. At this point the stack looks like the picture at right.
2. The subroutine may alter any "T" or "A" register or any "S" register that it saved in the prolog.

3. The subroutine refers to local variables as disp($fp)
.
4. The subroutine may push and pop values on the stack using $sp.
5. If the subroutine calls another subroutine, then it does so by following these rules.

Subroutine Epilog (at the end of the subroutine):

1. Put return values in $v0-$v1
2. $sp = $fp + space_for_variables.
3. Pop any registers $s0-$s7 that were previously saved in the frame.
4. Pop the caller's frame pointer into $fp.
5. Pop $ra.
6. Return to the caller using jr $ra.

Return from a Subroutine (done by the caller):

1. Pop any registers $t0-$t9 that the caller previously pushed.

QUESTION 6:

When the caller gets control back, are its frame pointer and stack pointer the same as when it called the subroutine?

Next

A good answer might be:

Yes.

Diagram

Those rules are nasty. In broad outline it works the same way as the previous chapter's stack-based calling convention.

Here is a picture. It shows the four sections of subroutine linkage. The basic tasks of each section are:

Subroutine Call: Push any "T" registers that values that are needed. Put arguments in "A" registers. jal to the subroutine.

Prolog: Push $ra, and caller's $fp. Push any "S" register the subroutine will alter. Initialize subroutine's $fp and $sp.

Body: Normal code, except it must follow these conventions if it calls another subroutine. "T" and "A" registers can be used freely, as can any "S" registers that were saved in the prolog. Variables are accessed using disp($fp).

Epilog: Put return values in "V" registers. Reset $sp. Pop any "S" registers. Pop the caller's $fp and $ra. jr $ra back to the caller.

Return: Pop any previously pushed "T" registers.

QUESTION 7:

Is there a limit to how many variables a subroutine may have?

Next

A good answer might be:

No.

Example Program

The number of registers of MIPS (or other processor) does not limit the number of variables that subroutines can have. As many variables as you want can be allocated on the stack. Here is an example program.

```
main()
{
  int a;
  a = mysub( 6 );
  print( a );
}

int mysub( int arg )
{
  int b,c;

  b = arg*2;
  c = b + 7;

  return c;
}
```

To the operating system, main() is a subroutine. When it first gets control it must follow the rules under "subroutine prolog".

QUESTION 8:

How much space on the stack is needed for main()'s variable?

Next

A good answer might be:

Four bytes.

main()

Here is the code for main(), with some blanks. The rules for the subroutine prolog are copied from above.

```
# main()
# {
#   int a;
#   a = mysub( 6 );
#   print( a );
# }
         .text
         .globl  main
main:
                                   # prolog
         sub     $sp,$sp,4        #   1. Push return address
         sw      $ra,($sp)
         sub     $sp,$sp,4        #   2. Push caller's frame pointer
         sw      $fp,($sp)
                                   #   3. No S registers to push
         sub     ___,_____,_____     #   4. $fp = $sp - space_for_variables

         ______  $sp,$fp           #   5. $sp = $fp

         # subroutine call

         . . . .

         # return from subroutine

         . . . .
         jr      $ra              # return to OS
```

QUESTION 9:

Fill in the blanks as the comments suggest.

A good answer might be:

See below.

Subroutine Call

```
#  main()
#  {
#    int a;
#    a = mysub( 6 );
#    print( a );
#  }
         .text
         .globl  main
main:
                            # prolog
         sub     $sp,$sp,4        #   1. Push return address
         sw      $ra,($sp)
         sub     $sp,$sp,4        #   2. Push caller's frame pointer
         sw      $fp,($sp)
                            #   3. No S registers to push
         sub     $fp,$sp,4        #   4. $fp = $sp - space_for_variables

         move    $sp,$fp          #   5. $sp = $fp

                            # subroutine call
                            #   1. No T registers to push
         li      $a0,___          #   2. Put argument into $a0
         jal     ________         #   3. Jump and link to subroutine

         # return from subroutine

         . . . .
         jr      $ra              # return to OS
```

QUESTION 10:

Fill in the blanks.

A good answer might be:

```
    #  main()
    #  {
    #    int a;
    #    a = mysub( 6 );
    #    print( a );
    #  }
             .text
             .globl  main
    main:
                                  # prolog
             sub     $sp,$sp,4       #   1. Push return address
             sw      $ra,($sp)
             sub     $sp,$sp,4       #   2. Push caller's frame pointer
             sw      $ra,($sp)

                                  #   3. No S registers to push

             addiu   $fp,$sp,4       #   4. $fp = $sp + space_for_variables

             move    $sp,$fp         #   5. $sp = $fp

                                  # subroutine call
                                  #   1. No T registers to push
             li      $a0,6           #   2. Put argument into $a0
             jal     mysub           #   3. Jump and link to subroutine

             # return from subroutine

             . . . .
             jr      $ra             # return to OS
```

Prolog for mysub()

Of course, mysub starts with a subroutine prolog. There are two variables, so space is assigned to them on the stack.

```
    #  int mysub( int arg )
    #  {
    #    int b,c;                    // b: 0($fp)
    #                                // c: 4($fp)
```

```
#    b = arg*2;
#    c = b + 7;
#
#    return c;
#  }
         .text
         .globl  ______
______:
                          # prolog
         sub     $sp,$sp,4       #   1. Push return address
         sw      $ra,($sp)
         sub     $sp,$sp,4       #   2. Push caller's frame pointer
         sw      $fp,($sp)

         ____    ___,___,___      #   3. Push register $s1

         ____    ___,_____

         sub    $fp,$sp,____     #   4. $fp = $sp - space_for_variables

         move   $sp,$fp          #   5. $sp = $fp

         . . . .

         jr     $ra              # return to caller
```

The subroutine could be written without using $s1. It is used to show how linkage works.

QUESTION 11:

Fill in those blanks.

Next

A good answer might be:

See below.

Using Variables

The subroutine uses two variables so there is eight bytes of space on the stack frame for them.

```
#  int mysub( int arg )
#  {
#    int b,c;                    // b: 0($fp)
#                                // c: 4($fp)
#    b = arg*2;
#    c = b + 7;
#
#    return c;
#  }
         .text
         .globl  mysub
mysub:
                                 # prolog
         sub     $sp,$sp,4       #   1. Push return address
         sw      $ra,($sp)
         sub     $sp,$sp,4       #   2. Push caller's frame pointer
         sw      $fp,($sp)

         sub     $sp,$sp,4       #   3. Push register $s1
         sw      $s1,($sp)

         sub     $fp,$sp,8       #   4. $fp = $sp - space_for_variables

         move    $sp,$fp         #   5. $sp = $fp

                                 # body of subroutine
         mul     $s1,$a0,2       # arg*2
         sw      $s1,___(___)    # b = "   "

         lw      $t0,___(___)    # get b
         add     $t0,$t0,___     # b+7
         sw      $t0,___(___)    # c = "  "

         . . . . .
```

```
        jr      $ra             # return to caller
```

The program is not very efficient, as written. There is no need to store and then load b. A non-optimizing compiler might do just that, however.

QUESTION 12:

Fill in the blanks. Assume that b is at displacement 0 and that c is at displacement 4.

Next

A good answer might be:

See Below

Subroutine Epilog

The epilog of a subroutine is the part that prepares the values that are returned to the caller and restores the registers and stack to where they were when the caller made the call.

```
#  int mysub( int arg )
#  {
#    int b,c;                     // b: 0($fp)
#                                // c: 4($fp)
#    b = arg*2;
#    c = b + 7;
#
#    return c;
#  }
         .text
         .globl  mysub
mysub:
                                  # prolog
         sub     $sp,$sp,4        #   1. Push return address
         sw      $ra,($sp)
         sub     $sp,$sp,4        #   2. Push caller's frame pointer
         sw      $ra,($sp)

         sub     $sp,$sp,4        #   3. Push register $s1
         sw      $s1,($sp)

         addiu   $fp,$sp,8        #   4. $fp = $sp + space_for_variables

         move    $sp,$fp          #   5. $sp = $fp

                                  # body of subroutine
         mul     $s1,$a0,2        #     arg*2
         sw      $s1,0($fp)       # b = "   "

         lw      $t0,0($fp)       # get b
         add     $t0,$t0,7        #      b+7
         sw      $t0,4($fp)       # c = "  "

                                  # epilog
```

```
lw     $v0,___(___)    #   1. Put return value in $v0

add    $sp,$fp,_____   #   2. $sp = $fp +space_for_variables

lw     $s1,(____)      #   3. Pop register $s1
add    ___,___,___     #

lw     $fp,(____)      #   4. Pop $fp
add    ___,___,___     #

lw     $ra,(____)      #   5. Pop $ra
add    ___,___,___     #

jr     $ra             #   6. return to caller
```

QUESTION 13:

Those blanks need filling.

Next

A good answer might be:

See Below.

Complete mysub()

```
#  int mysub( int arg )
#  {
#    int b,c;                    // b: 0($fp)
#                                // c: 4($fp)
#    b = arg*2;
#    c = b + 7;
#
#    return c;
#  }
         .text
         .globl  mysub
mysub:
                                 # prolog
         sub     $sp,$sp,4       #   1. Push return address
         sw      $ra,($sp)
         sub     $sp,$sp,4       #   2. Push caller's frame pointer
         sw      $fp,($sp)
         sub     $sp,$sp,4       #   3. Push register $s1
         sw      $s1,($sp)
         sub     $fp,$sp,8       #   4. $fp = $sp - space_for_variables
         move    $sp,$fp         #   5. $sp = $fp

                                 # body of subroutine
         mul     $s1,$a0,2       #     arg*2
         sw      $s1,0($fp)      # b = "   "

         lw      $t0,0($fp)      # get b
         add     $t0,$t0,7       #     b+7
         sw      $t0,4($fp)      # c = "  "

                                 # epilog
         lw      $v0,4($fp)      #   1. Put return value in $v0
         add     $sp,$fp,8       #   2. $sp = $fp + space_for_variables
         lw      $s1,($sp)       #   3. Pop register $s1
         add     $sp,$sp,4       #
         lw      $fp,($sp)       #   4. Pop $fp
         add     $sp,$sp,4       #
```

```
lw      $ra,($sp)      #   5. Pop $ra
add     $sp,$sp,4      #
jr      $ra            #   6. return to caller
```

QUESTION 14:

What must the caller do when control returns to it?

Next

A good answer might be:

The caller must restore any "T" registers it saved.

Back to main()

In this example, main() did not save any "T" registers. There is nothing to do for the return from the subroutine, other than use the value that was returned in $v0.

```
# main()
# {
#   int a;               // a: 0($fp)
#   a = mysub( 6 );
#   print( a );
# }
        .text
        .globl  main
main:
                         # prolog
        sub    $sp,$sp,4      #  1. Push return address
        sw     $ra,($sp)
        sub    $sp,$sp,4      #  2. Push caller's frame pointer
        sw     $fp,($sp)
                         #  3. No S registers to push
        sub    $fp,$sp,4      #  4. $fp = $sp - space_for_variables
        move   $sp,$fp        #  5. $sp = $fp

                         # subroutine call
                         #  1. No T registers to push
        li     $a0,6          #  2. Put argument into $a0
        jal    mysub          #  3. Jump and link to subroutine

                         # return from subroutine
                         #  1. No T registers to restore

        sw     $v0,__(___)    # a = mysub( 6 )

                         # print a
        lw     $a0,__(___)    # load a into $a0
        li     $v0,1          # print integer service
        syscall
```

```
                        # epilog
                        #   1. No return value
add     $sp,$fp,4       #   2. $sp = $fp + space_for_variables
                        #   3. No S registers to pop
lw      $fp,($sp)       #   4. Pop $fp
add     $sp,$sp,4       #
lw      $ra,($sp)       #   5. Pop $ra
add     $sp,$sp,4       #
jr      $ra             # return to OS
```

QUESTION 15:

Fill in the blanks.

Next

A good answer might be:

See below.

Complete main()

Here is the complete main() routine. Nearly all of the code is concerned with subroutine linkage. This is typical. Optimizing compilers produce much more efficient code than this.

```
#  main()
#  {
#    int a;                    // a: 0($fp)
#    a = mysub( 6 );
#    print( a );
#  }
         .text
         .globl  main
main:
                              # prolog
         sub     $sp,$sp,4      #   1. Push return address
         sw      $ra,($sp)
         sub     $sp,$sp,4      #   2. Push caller's frame pointer
         sw      $fp,($sp)
                              #   3. No S registers to push
         sub     $fp,$sp,4      #   4. $fp = $sp - space_for_variables
         move    $sp,$fp        #   5. $sp = $fp

                              # subroutine call
                              #   1. No T registers to push
         li      $a0,6          #   2. Put argument into $a0
         jal     mysub          #   3. Jump and link to subroutine

                              # return from subroutine
                              #   1. No T registers to restore

         sw     $v0,0($fp)      # a = mysub( 6 )

                              # print a
         lw     $a0,0($fp)      # load a into $a0
         li     $v0,1           # print integer service
         syscall
                              # epilog
```

```
                         #   1. No return value
add     $sp,$fp,4        #   2. $sp = $fp + space_for_variables
                         #   3. No S registers to pop
lw      $fp,($sp)        #   4. Pop $fp
add     $sp,$sp,4        #
lw      $ra,($sp)        #   5. Pop $ra
add     $sp,$sp,4        #
jr      $ra              # return to OS
```

If you wish to run the program, copy and paste main() and mysub() into one file.

QUESTION 16:

Rewrite the following code so that it does the same thing as the original but without using a variable.

Next

```
main()
{
  int a;
  a = mysub( 6 );
  print( a );
}
```

A good answer might be:

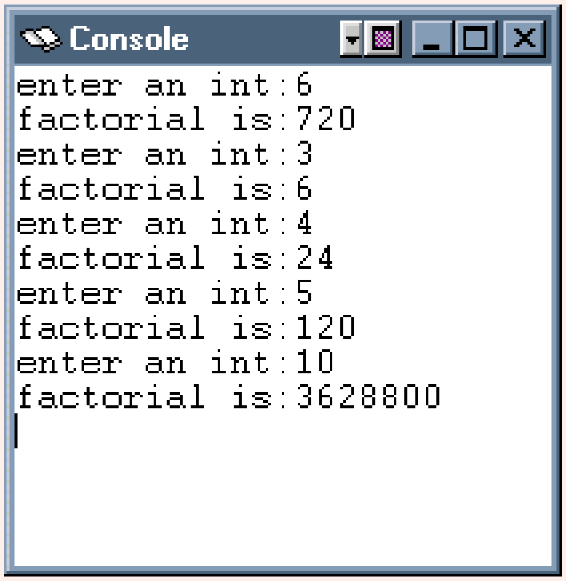
Optimizing compilers sometimes examine the source code and rewrite it into something more efficient, such as:

```
main()
{
  print(  mysub( 6 ) );
}
```

Example Program: Factorial(N)

The next example program prompts the user for an integer, reads in the integer, and prints the factorial. The SPIM console window is at right. The pseudo-code for the program is:

```
#  main()
#  {
#    int a, b;                    // a: 0($fp),  b: 4($fp)
#    write("enter an int:")
#    read( a );
#    b = fact( a );
#    write("factorial is:")
#    print( b );
#  }

#  int fact( int n )
#  {
#    if ( n<=1)
#      return 1;
#    else
#      return n*fact(n-1);
#  }
```



Console

```
enter an int:6
factorial is:720
enter an int:3
factorial is:6
enter an int:4
factorial is:24
enter an int:5
factorial is:120
enter an int:10
factorial is:3628800
```

QUESTION 17:

You have very likely seen the factorial function before (and are very likely sick of it!) But just in case:

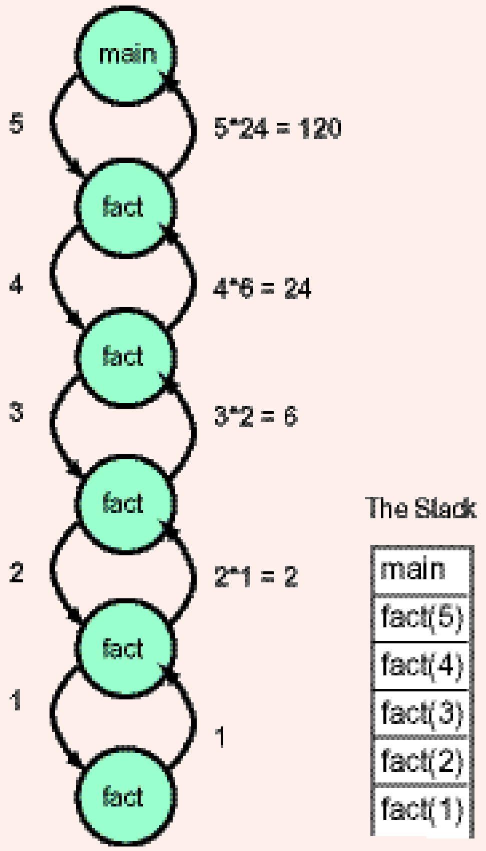
What is the factorial of 5?

A good answer might be:

```
fact(5) == 5*fact(4)
        == 5*( 4*fact(3) )
        == 5*( 4*( 3*fact(2)) )
        == 5*( 4*( 3*(2*fact(1))) )
        == 5*( 4*( 3*(2*1)) )
        == 5*4*3*2*1
        == 120
```

Activation Chain

If the subroutine fact() is called with an argument greater than one, it calls itself, fact(), with a new argument. This works because the data needed for the first activation offact() is pushed onto the stack. When that activation gets control again that data is popped from the stack. Often this process is illustrated as at right.



Each bead on the activation chain represents an activation of a subroutine. The label on a downward arc is the argument to the subroutine. The label on an upward ard is the returned value.

Each beads on the activation chain corresponds to one stack frame. The picture of the stack shows what it looks like when the activation fact(1) is running.

QUESTION 18:

A downward arc corresponds to a _______ of one stack frame. An upward arc corresponds to a _______ of one stack frame.

Next

A good answer might be:

A downward arc corresponds to a push of one stack frame. An upward arc corresponds to a pop of one stack frame.

Prolog of main()

Here is the main routine's pseudocode and prolog. Notice that there are two variables.

```
# main()
# {
#   int a, b;                  // a: 0($fp),  b: 4($fp)
#   write("enter an int:")
#   read( a );
#   b = fact( a );
#   write("factorial is:")
#   print( b );
# }
         .text
         .globl  main
main:
                             # prolog
         sub    $sp,$sp,4      #   1. Push return address
         sw     $__,($sp)
         sub    $sp,$sp,4      #   2. Push caller's frame pointer
         sw     $__,($sp)
                             #   3. No S registers to push
         sub    $fp,$sp,_____    #   4. $fp = $sp - space_for_variables
         move   $sp,$fp        #   5. $sp = $fp
```

QUESTION 19:

Fill in the blanks.

Next

A good answer might be:

```
#  main()
#  {
#    int a, b;                    // a: 0($fp),  b: 4($fp)
#    write("enter an int:")
#    read( a );
#    b = fact( a );
#    write("factorial is:")
#    print( b );
#  }
         .text
         .globl  main
main:
                                  # prolog
         sub     $sp,$sp,4        #   1. Push return address
         sw      $ra,($sp)
         sub     $sp,$sp,4        #   2. Push caller's frame pointer
         sw      $fp,($sp)
                                  #   3. No S registers to push
         sub     $fp,$sp,8        #   4. $fp = $sp - space_for_variables
         move    $sp,$fp          #   5. $sp = $fp
```

Guts of main()

The next part of main() is straightforward. The SPIM trap services for writing a string and reading an integer are used. The integer is returned in $v0. It is saved in the variable a (on the stack).

```
                          # write("enter an int:")
        li     $v0,4          #   print string service
        la     $a0,prompt1    #   address of prompt
        syscall
                          # read( a )
        li     $v0,5          #   read integer service
        syscall               #   $v0 gets the integer
        sw     $v0,0($fp)     #   save in variable a
```

Next the code implements b = fact(a). This is done by following the protocol for a subroutine call, then storing the returned value into the variable b:

```
                    # subroutine call: b = fact( a )
                    #   1. No T registers to push
lw     $a0,_____($fp)   #   2. Put argument into $a0
jal    _____            #   3. Jump and link to subroutine

                    # return from subroutine
                    #   1. No T registers to restore

sw     $v0,_____($fp)   # b = fact( a )
```

QUESTION 20:

Fill in the blanks.

Next

A good answer might be:

```
                           # subroutine call: b = fact( a )
                           #   1. No T registers to push
         lw     $a0,0($fp)      #   2. Put argument into $a0
         jal    fact            #   3. Jump and link to subroutine

                           # return from subroutine
                           #   1. No T registers to restore

         sw     $v0,4($fp)      # b = fact( a )
```

More main()

Next main() does some utterly routine things:

```
                           # print( b )
         lw     $a0,4($fp)      # load a into $a0
         li     $v0,1           # print integer service
         syscall
                           # end the print line
         li     $v0,4           #   print string service
         la     $a0,lf          #   address of line feed
         syscall
```

Finally, main() ends with a subroutine epilog. The data for the prompts is not stored on the stack.

```
                           # epilog
                           #   1. No return value
         add    $sp,$fp,8       #   2. $sp = $fp + space_for_variables
                           #   3. No S registers to pop
         lw     $fp,($sp)       #   4. Pop $fp
         add    $sp,$sp,4       #
         lw     $ra,($sp)       #   5. Pop $ra
         add    $sp,$sp,4       #
         jr     $ra             # return to OS

         .data
```

```
prompt1: .asciiz "enter an int:"
prompt2: .asciiz "factorial is:"
lf:      .asciiz "\n"
```

QUESTION 21:

How do the variables `a` and `b` differ from the data `prompt1` and `prompt2`?

A good answer might be:

a and b: (1) Storage for them exists only while the subroutine is active (while it has not yet returned control to its caller). (2) When it exists, the storage is on the run-time stack.

prompt1 and prompt2: (1) Storage for them exists while the program is in main storage. (2) Storage is in the data section of memory.

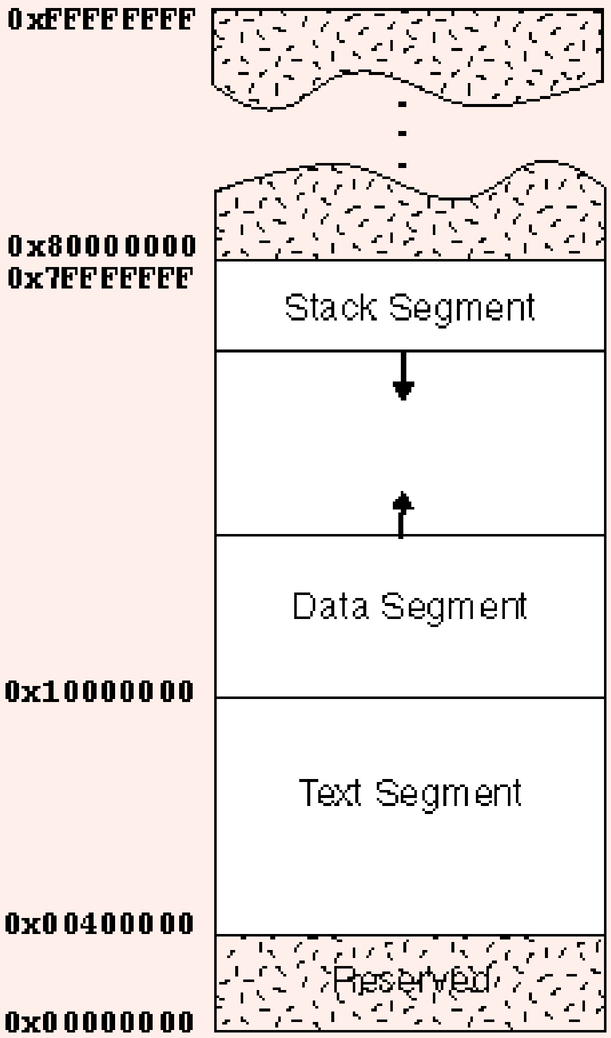
Storage Classes

There are three places in memory where data may be placed: in the data section (declared with .data in assembly language), on the run-time stack, and on the heap.

A subroutine other than main() can have data in the .data section. In high-level programming languages, such as "C", this type of storage is called **static**.

Variables whose storage is allocated on the run-time stack are (sometimes) called **automatic** variables. This is because their storage is "automatically" pushed and popped as a subroutine is entered and exited. Usually the word "variable" means "automatic variable".

A variable whose memory is located in the heap is called a **dynamic variable**. These notes only briefly deal with the heap. The heap is where memory for objects is found (using the new operation in Java or C++). In "C" dynamic memory is allocated using the malloc operation (or similar).



The **heap** is on top of the data segment. As dynamic variables are created it grows upward (towards the stack)

QUESTION 22:

(Review:) What happens if the stack and heap get larger and larger?

Next

A good answer might be:

If the combined size of the the stack, the data, and the heap is less than the total available memory, then there is no problem.

Complete main()

Back the the example program (you were probably hoping that I'd forget). Here is the complete main(). There is nothing new in it; its listed here so you can see all the parts in place.

```
#  main()
#  {
#    int a, b;                  // a: 0($fp),  b: 4($fp)
#    write("enter an int:")
#    read( a );
#    b = fact( a );
#    write("factorial is:")
#    print( b );
#  }
         .text
         .globl  main
main:
                           # prolog
         sub     $sp,$sp,4      #   1. Push return address
         sw      $ra,($sp)
         sub     $sp,$sp,4      #   2. Push caller's frame pointer
         sw      $fp,($sp)
                           #   3. No S registers to push
         sub     $fp,$sp,8      #   4. $fp = $sp - space_for_variables
         move    $sp,$fp        #   5. $sp = $fp

                           # write("enter an int:")
         li     $v0,4           #   print string service
         la     $a0,prompt1     #   address of prompt
         syscall
                           # read( a )
         li     $v0,5           #   read integer service
         syscall                #   $v0 gets the integer
         sw     $v0,0($fp)      #   save in variable a

                           # subroutine call
```

```
                        #   1. No T registers to push
        lw      $a0,0($fp)      #   2. Put argument into $a0
        jal     fact            #   3. Jump and link to subroutine

                        # return from subroutine
                        #   1. No T registers to restore

        sw      $v0,4($fp)      # b = fact( a )

                        # write("factorial is:")
        li      $v0,4           #   print string service
        la      $a0,prompt2     #   address of prompt
        syscall
                        # print( b )
        lw      $a0,4($fp)      # load a into $a0
        li      $v0,1           # print integer service
        syscall

                        # end the print line
        li      $v0,4           #   print string service
        la      $a0,lf          #   address of line feed
        syscall
                        # epilog
                        #   1. No return value
        add     $sp,$fp,8       #   2. $sp = $fp + space_for_variables
                        #   3. No S registers to pop
        lw      $fp,($sp)       #   4. Pop $fp
        add     $sp,$sp,4       #
        lw      $ra,($sp)       #   5. Pop $ra
        add     $sp,$sp,4       #
        jr      $ra             # return to OS

        .data
prompt1: .asciiz "enter an int:"
prompt2: .asciiz "factorial is:"
lf:      .asciiz "\n"
```

QUESTION 23:

What subroutine does main() call?

Next

A good answer might be:

fact

Entry Point

The first address in this subroutine is called fact. Of course, fact will correspond to a main storage address at run-time. The exact address is determined by the assembler, the linker, and the loader.

```
#  int fact( int n )
#  {
#    if ( n<=1)
#      return 1;
#    else
#      return n*fact(n-1);
#  }
         .text
         .globl  fact
fact:
                                  # prolog
         sub     $sp,$sp,4        #   1. Push return address
         sw      $ra,($sp)
         sub     $sp,$sp,4        #   2. Push caller's frame pointer
         sw      $fp,($sp)
         sub     $sp,$sp,4        #   3. Push register $s1
         sw      $s1,($sp)
         sub     $fp,$sp,0        #   4. $fp = $sp - space_for_variables (==0)
         move    $sp,$fp          #   5. $sp = $fp

                                  # body of subroutine
        . . . . . .

epilog:                           # epilog
                                  #   1. Return value is already in $v0
         add     $sp,$fp,0        #   2. $sp = $fp + space_for_variables (==0)
         lw      $s1,($sp)        #   3. Pop register $s1
         add     $sp,$sp,4        #
         lw      $fp,($sp)        #   4. Pop $fp
         add     $sp,$sp,4        #
         lw      $ra,($sp)        #   5. Pop $ra
         add     $sp,$sp,4        #
        jr      $ra             #  6. return to caller
```

The symbol fact is a global symbol (also called an external symbol) so that the assembler, linker, and loader can use that symbol to refer to the same place in memory.

A location such as fact that is a target of a subroutine call is called an **entry point**. Sometimes a subroutine has several entry points, one for each of several related functions.

QUESTION 24:

(Thought Question:) Does a global symbol *always* correspond to an entry point?

Next

A good answer might be:

No. Sometimes a global symbol is used to label data that several modules may refer to.

Body of fact()

Here is part of the body of the subroutine:

```
#  int fact( int n )
#  {
#    if ( n<=1)
#      return 1;
#    else
#      return n*fact(n-1);
#  }
         .text
         .globl  fact
fact:
                                  # prolog
         . . . . . .
                                  # body of subroutine
         move    $s1,$a0          # save argument in $s1
         li      $t1,1            # get a 1
         bgt     $s1,$t1,recurse  # if ( n<=1)
         li      $v0,1            #   return 1
         b       ________

recurse:                          # else
                                  #   return n*fact(n-1);
         . . . . . .
epilog:                           # epilog
                                  #   1. Return value is already in $v0
         . . . . . .
         jr      $ra              #
```

The argument in $a0 is saved in register $s1 because later on $a0 may be altered. (Since this subroutine uses $s1 the contents of $s1 is saved on the stack in the prolog).

The if statement checks if the argument (in $a0) is 1 or less. If so, it loads register $v0 with the value to return to the caller, one. Otherwise, the other branch is taken.

QUESTION 25:

Fill in the blank.

A good answer might be:

See below.

More fact()

```
#  int fact( int n )
#  {
#    if ( n<=1)
#      return 1;
#    else
#      return n*fact(n-1);
#  }
         .text
         .globl  fact
fact:
                              # prolog
         . . . . . .
                              # body of subroutine
         move    $s1,$a0          # save argument in $s1
         li      $t1,1            # get a 1
         bgt     $s1,$t1,recurse  # if ( n<=1)
         li      $v0,1            #   return 1
         b       epilog

recurse:                         # else
                              #   return n*fact(n-1);
         sub     $a0,$s1,1        #     argument0 = n-1

                              # subroutine call
                              #   1. No T registers to push
                              #   2. Argument is in $a0
         jal     fact             #   3. Jump and link to subroutine

         mul     $___,$___,$___   # n*fact(n-1)

epilog:                          # epilog
                              #   1. Return value is already in $v0
         . . . . . .
         jr      $ra              #
```

The alternate branch of the if statement has the job of calculating n*fact(n-1). It does this by first calculating the argument n-1.

Then it calls the subroutine fact() in the normal way. It does not hurt for fact() to call fact() because each activation has its own data on the stack.

On return from the (inner) call to fact(), register $v0 has the returned value, and register $s1 has the argument n. Now the return value from the current activation must be placed in $v0 to be returned to the caller.

QUESTION 26:

Fill in the blanks. (Hint: study the last paragraph).

Next

A good answer might be:

See Below.

Recursive Call

```
          . . . . . .

     recurse:                   # else
                             #   return n*fact(n-1);
          sub     $a0,$s1,1       #     argument0 = n-1

                             # subroutine call
                             #   1. No T registers to push
                             #   2. Argument is in $a0
          jal     fact         #   3. Jump and link to subroutine

          mul     $v0,$v0,$s1      # n*fact(n-1)

     epilog:                   # epilog
                             #   1. Return value is already in $v0
          . . . . . .
          jr      $ra          #
```

Recursion has been implemented using: (1) the normal machine operations of sequential execution, testing, and branching, and (2) the run-time stack.

This is (yet another) example of a new level of abstraction being build upon a foundation level. Have I mentioned that this is one of the most stunningly important ideas of Computer Science?

QUESTION 27:

The programming language FORTRAN IV did not have support for recursion. Was it possible to write a recursive program in FORTRAN IV?

Next

A good answer might be:

Yes. FORTRAN had sequential execution, testing, and branching, as do all languages. To write a recursive program the programmer had to create and manage a run time stack, just as in assembly language. Modern high level languages do this automatically.

Complete fact()

Here is the complete code for fact():

```
# int fact( int n )
# {
#   if ( n<=1)
#     return 1;
#   else
#     return n*fact(n-1);
# }
         .text
         .globl  fact
fact:
                              # prolog
         sub     $sp,$sp,4       #   1. Push return address
         sw      $ra,($sp)
         sub     $sp,$sp,4       #   2. Push caller's frame pointer
         sw      $fp,($sp)
         sub     $sp,$sp,4       #   3. Push register $s1
         sw      $s1,($sp)
         sub     $fp,$sp,0       #   4. $fp = $sp - space_for_variables (==0)
         move    $sp,$fp         #   5. $sp = $fp

                              # body of subroutine
         move    $s1,$a0         # save argument in $s1
         li      $t1,1           # get a 1
         bgt     $s1,$t1,recurse  # if ( n<=1)
         li      $v0,1           #   return 1
         b       epilog

recurse:                         # else
                              #  return n*fact(n-1)
         sub     $a0,$s1,1       #     n-1

                              # subroutine call
```

```
                                #   1. No T registers to push
                                #   2. Argument is in $a0
         jal     fact           #   3. Jump and link to subroutine

         mul     $v0,$v0,$s1    # n*fact(n-1)

epilog:                         # epilog
                                #   1. Return value is already in $v0
         add     $sp,$fp,0      #   2. $sp = $fp + space_for_variables (==0)
         lw      $s1,($sp)      #   3. Pop register $s1
         add     $sp,$sp,4      #
         lw      $fp,($sp)      #   4. Pop $fp
         add     $sp,$sp,4      #
         lw      $ra,($sp)      #   5. Pop $ra
         add     $sp,$sp,4      #
         jr      $ra            #   6. return to caller
```

QUESTION 28:

Is a different subroutine linkage convention followed for recursive subroutines than for non-recursive subroutines?

Next

A good answer might be:

No. The same convention is followed by both. There is nothing special about a recursive subroutine except that it calls itself.

End of the Chapter

You have reached the end this chapter. You may wish to review the following. Click on a subject that interests you to go to where it was discussed.

contents

You have reached the end of the chapter.

created: 05/27/2002

CHAPTER 29 — Binary Fractions

Up until now all of the arithmetic we have done on the MIPS has been with integers, both signed (two's complement) and unsigned. This chapter starts the discussion of how floating point arithmetic is done on the MIPS.

Chapter Topics:

- *Positional notation with fractions.*
- *Converting fractions between base two and base ten.*
- *Fixed point representation.*
- *Limited precision of binary fractions (fixed point and floating point).*
- *How "one tenth" cannot be expressed in binary.*
- *Dangerous program loops.*

QUESTION 1:

Is is possible to represent a **negative** integer with a **positive** number of bits?

Next

A good answer might be:

Of course. Bit patterns can represent anything symbolic.

Bit Patterns

The question is silly. There is nothing mysterious about representing signed numbers with bit patterns. Here is another silly question:

> Is it possible to represent a **fraction** with
> a whole number of bits?

Of course. It is just a matter of mapping fractions to the bit patterns that represent them. There are several systematic ways in which this can be done. This chapter discusses some of them.

QUESTION 2:

In the decimal fraction 12.6 what power of 10 is associated with:

Next

- The digit 1?
- The digit 2?
- The digit 6?

A good answer might be:

In the decimal fraction 12.6 what power of 10 is associated with:

- The digit 1? 10^{1}
- The digit 2? 10^{0}
- The digit 6? 10^{-1}

Positional Notation

This is just **base 10 positional notation** which you know about already. The **decimal point** is put between the 10^{0} position and the 10^{-1} position:

3	5	7	**.**	2	8	4
$3\times10^{2}+$	$5\times10^{1}+$	$7\times10^{0}+$	**.**	$2\times10^{-1}+$	$8\times10^{-2}+$	4×10^{-3}

QUESTION 3:

Express 3/10 + 1/100 using positional notation.

Next

A good answer might be:

0.301

When a particular power of ten is not needed, the digit '0' is used for that position.

Base Two Positional Notation

Positional notation can also be used with a base other than ten. Here is a number written using base two:

1	0	0	**.**	1	0	1
$1\times2^{2}+$	$0\times2^{1}+$	$0\times2^{0}+$	**.**	$1\times2^{-1}+$	$0\times2^{-2}+$	1×2^{-3}

Now the '**.**' is called a **binary point**. In general, the '**.**' is called a **radix point**.

QUESTION 4:

What is 1×2^{-1} ? (Express the answer in base ten notation).

Next

A good answer might be:

$1 \times 2^{-1} = 1/(2^1) = 1/2 = 0.5$

Base Two to Base Ten

Recall that X^{-n} means $1/X^n$. So, $2^{-2} = 1/4$ and $2^{-3} = 1/8$.

To convert an expression in base two notation to base ten notation, just do the arithmetic that the expression calls for:

1	0	0	.	1	0	1
1×2^2 +	0×2^1 +	0×2^0 +	.	1×2^{-1} +	0×2^{-2} +	1×2^{-3}
1×4 +	0×2 +	0×1+	.	1×0.5 +	0×0.25 +	1×0.125
4 +	0 +	0 +	.	0.5 +	0 +	0.125
		4	.	625		

As you work, keep track of what parts of the expression are in base two and what parts are in base ten. In the above, the first row is entierly in base two, the bottom row is entierly in base ten, and the middle rows are mixed.

QUESTION 5:

Write 0.11_2 as a base 10 expression.

Next

A good answer might be:

$0.11_2 = 0\times1 + 1\times(0.5) + 1\times(0.25) = 0.75$

Include or Exclude?

In a binary representation a particular power of two is either included in the sum or not, since the digits are either "1" or "0". A "1" bit in a position means to include the corresponding power of two in the sum. In converting representations, it is convenient to have a table.

Power of 2	**3**	**2**	**1**	**0**	**.**	**-2**	**-3**	**-4**	**-5**
Decimal	**8**	**4**	**2**	**1**	**.**	**0.5**	**0.25**	**0.125**	**0.0625**
Include?					**.**				

QUESTION 6:

What is 100.01_2 expressed in decimal?

Next

A good answer might be:

Power of 2	3	2	1	0	.	-2	-3	-4	-5
Decimal	8	4	2	1	.	0.5	0.25	0.125	0.0625
Include?		1	0	0	.	0	1		

= 4.25

Fixed Point Notation

With paper-and-pencil arithmetic you can use as many digits or bits as are needed. But computers (usually) use a fixed number of bits for a particular data type. For example, MIPS integers are 32 bits. Can a fixed number of bits be used to express fractions?

Yes. Let us look briefly at an older method, not much used anymore. In the past, some electronic calculators and some computers used **fixed point** notation for expressing fractional number. This is when a number is expressed in a number of bits (say eight) and the binary point is assumed be be fixed between two of them. For example, let us say that fixed point numbers use eight bits and that the binary point is fixed between the middle two bits, like in the table. Now to interpret an eight-bit expression, just copy the bits to the table.

QUESTION 7:

In this scheme, what does the bit pattern 01101001 represent in decimal?

Next

A good answer might be:

Power of 2	3	2	1	0	.	-2	-3	-4	-5
Decimal	8	4	2	1	.	0.5	0.25	0.125	0.0625
Include?	0	1	1	0	.	1	0	0	1

= 4 + 2 + 0.5 +0.0625 = 6.5625

Adding fixed point Expressions

Here is another number: 00010100 represents decimal 1.25. Here is the familiar *binary addition algorithm* performed with the two bit patterns, and the usual decimal addition performed with their decimal equivalent.

fixed point	as decimal
01101001	6.5625
00010100	1.2500
————	————
01111101	7.8125

QUESTION 8:

Of course, the question is, does the sum of the fixed point expressions (01111101) represent the sum of the decimal expressions (7.8125)?

Next

Power of 2	3	2	1	0	.	-2	-3	-4	-5

Decimal	**8**	**4**	**2**	**1**	**.**	**0.5**	**0.25**	**0.125**	**0.0625**
Include?	**0**	**1**	**1**	**1**	**.**	**1**	**1**	**0**	**1**

You can satisfy your burning curiosity by adding up the included decimals in the table.

A good answer might be:

Power of 2	3	2	1	0	.	-2	-3	-4	-5
Decimal	8	4	2	1	.	0.5	0.25	0.125	0.0625
Include?	0	1	1	1	.	1	1	0	1

= 4 + 2 + 1 + 0.5 + 0.25 + 0.0625 = 7.8125

Binary Addition Algorithm Works!

The binary addition algorithm works for our eight-bit fixed point notation! In early computers and calculators this enabled the same circuits to be used for integer arithmetic and for fixed point arithmetic. The two's complement method of representing a negative number also works with fixed point. This looks very promising.

However, fixed point is not as useful as **floating point** representation which has replaced it. Floating point will be discussed in the next chapter.

QUESTION 9:

How many different values is it possible to represent using our 8-bit fixed point representation?

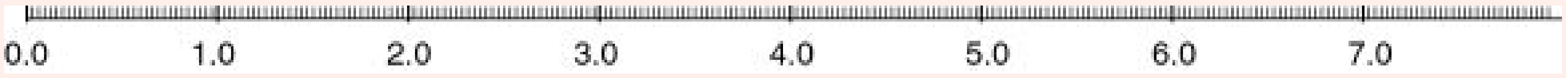
Next

A good answer might be:

$2^8 = 256$

Limited Precision

This is an easy question. If N bits are used, then only 2^N things can be represented, no matter what type of things they are. Eight bits can represent 256 positive integers (unsigned binary), 256 positive and negative integers (two' complement), 256 fractional numbers (our fixed point method), and so on. Here is a number line showing the 256 values that can be represented with this fixed notation:



The smallest value is zero (00000000); the largest value is 7.9375 (11111111). The smallest non-zero value is 0.0625. Every represented value is a multiple of 0.0625. Another way to think of fixed point notation is that the eight bits represent an integer from 0 to 255. The integer is a multiplier of the value 0.0625.

The important point: The point of all this is that on the number line from zero to eight not all real numbers can be represented. This is called a limit in **precision**. Floating point representation has the same problem.

QUESTION 10:

(Calculus Review:) How many **real numbers** lie between 0.0 and 0.0625?

Next

(Calculus Review:) How many **real numbers** lie between 0.0 and 0.0625?

A good answer might be:

An infinite number.

Shifting Left One Bit

The limited precision of fixed point (and floating point) expressions is a problem in programming. Even with 64-bit double precision floating point (such as Java data type double) precision is limited. Only 2^{64} numbers can be represented. This may seem like a lot, but between any two represented numbers there are an infinite number of real numbers that are not represented!

QUESTION 11:

How many digits does the value pi have?

Next

A good answer might be:

An infinite number of digits (or bits).

More about Positional Notation

Let us return to paper-and-pencil positional notation. Here is a binary expression:

$0001.0100 = 1.25_{10}$

Here is the same pattern, shifted left by one bit:

$0010.1000 = 2.50_{10}$

QUESTION 12:

What does shifting left by one bit do?

Next

What does shifting left by one bit do?

A good answer might be:

Multiplies by two. This is the same as with unsigned integers. Also, shifting one bit right is equivalent to division by two.

Converting Representation from Decimal to Binary

Often you need to convert a decimal expression like 7.625 into a binary expression. To do this, first convert the whole number (32 in this case) to binary (111 in this case), append a binary point, and convert the decimal fraction to binary.

> **To convert a decimal fraction to Base 2:** Repeatedly multiply the decimal fraction by two. After each multiplication, copy the bit (0 or 1) that moves into the one's place (the place just left of the radix point) to the right of the binary fraction, then erase it from the decimal. Stop when the decimal fraction is zero. (Note: if this never happens, stop when you have the number of places you want.)

For our example: 7.625 is 111.101_2. In this case the conversion stopped when the decimal fraction changed to zero. This may not always happen.

	Decimal	**Binary so far**
Start	0.625	0.
×2	**1**.250	0.**1**
	.250	0.1
×2	**0**.500	0.1**0**
	.500	0.10
×2	**1**.000	0.10**1**
Result	.000	0.101

QUESTION 13:

(Thought Question:) Why will no other digit but 0 or 1 move into the one's place? (Hint: consider the biggest possible decimal fraction).

Next

A good answer might be:

You might fear that a '2' or higher digit might move into the one's place. Try this: multiply the biggest decimal fraction 0.99999... by two. This yields 1.99999... Even with this fraction a '1' moves into the one's place.

Practice

Here is an table set up to convert 0.75 to binary.

	Decimal	Binary so far
Start	0.750	0.
×2		0.
		0.
×2		0.
Result		0.

QUESTION 14:

Perform the conversion.

Next

A good answer might be:

	Decimal	Binary so far
Start	0.750	0.
×2	1.50	0.1
	.50	0.1
×2	1.00	0.11
Result		0.11

So $0.75_{10} = 0.11_2$.

To check this, go in the other direction: $0.11_2 = 2^{-1} + 2^{-2} = 0.5 + 0.25 = 0.75$

Non-terminating Result

Here is the algorithm used to convert 0.1_{10} to binary.

The algorithm does not end. After it has started up, the same pattern 0.2, 0.4, 0.8, 1.6, 0.6, 1.2, 0.2 repeats endlessly. The pattern 0011 is appended to the growing binary fraction for each repitition.

Unexpected Fact: The value "one tenth" cannot be represented precisely using a binary fraction.

This is true in the base two positional notation used here, and also in floating point representation used in programming languages. This is sometimes an important consideration when high accuracy is needed.

	Decimal	Binary so far
Start	0.1	0.
×2	0.2	0.0
×2	0.4	0.00
×2	0.8	0.000
×2	1.6	0.0001
	.6	0.0001
×2	1.2	0.00011
	0.2	0.00011

×2	0.4	0.000110
×2	0.8	0.0001100
×2	1.6	0.00011001
	.6	0.00011001
×2	1.2	0.000110011
	0.2	0.000110011
×2	0.4	0.0001100110
×2	0.8	0.00011001100
Result		0.00011001100...

QUESTION 15:

Can "one third" be represented accurately in decimal?

Next

Can "one third" be represented accurately in decimal?

A good answer might be:

No. Its representation continues indefinitely: 0.3333333....

The problem with "one tenth" is not a special flaw of binary. All bases have fractions that cannot be represented accurately.

Dangerous Code

Here is a program fragment in Java. (It could also be in C or C++). Is there something you should worry about in it?

```
float x;

for ( x = 0.0; x != 10.0; x += 0.1 )
{
  System.out.println("Crimsom and Clover");
}
```

QUESTION 16:

How many times will "Crimsom and Clover" be printed?

Next

A good answer might be:

Over and over.

Hidden Infinite Loop

In base 10 arithmetic, 0.1 is added to x 100 times until x reaches exactly 100.0. But the arithmetic is actually done in base two floating point which does not represent 0.1 exactly. The variable x may miss hitting 100.0 and the program will loop indefinitely.

```
float x;

for ( x = 0.0; x != 10.0; x += 0.1 )
{
  System.out.println("Crimsom and Clover");
}
```

Smart compilers look out for this problem, and may fix it, so this loop might actually work as intended. But is not wise to use it.

QUESTION 17:

Fix the code.

A good answer might be:

See below.

Count with Integers

If the purpose of the program fragment is to loop 100 times, then a counting loop using an integer variable should be used. Here is the best fix:

```
int x;

for ( x = 0; x < 100; x++ )
{
  System.out.println("Crimsom and Clover");
}
```

(It is conventional to write a counting loop this way, even though testing for an "exact hit" with x == 99 is logically equivalent.) Here is a poor solution:

```
float x;

for ( x = 0.0; x < 10.0; x += 0.1 )
{
  System.out.println("Crimsom and Clover");
}
```

This will probably work as intended, but there is a risk it will loop 101 times.

QUESTION 18:

Say that you need 100 values, 0.0, 0.1, up to 9.9? Can an integer-controlled counting loop be used?

Next

A good answer might be:

Yes.

Calculate Each Float

Use the for-loop structure to correctly and safely count. Calculate a new value of the floating point variable each time.

```
double x;
int    j;

for ( j = 0; j < 100; j++ )
{
  x = j/10.0;

  // do something with x
}
```

Almost always, floating point work should be done with double precision, as above. An even more accurate version of the loop is:

```
double x;
int    j;

for ( j = 0; j < 160; j++ )
{
  x = j/16.0;

  // do something with x
}
```

This is better because (1/16) is accurately represented in binary.

QUESTION 19:

Is floating point representation (as used in computers) the same as base two radix notation (as discussed in this chapter)?

Next

A good answer might be:

No. They are related, but floating point is more sophisticated. The next chapter discusses it.

End of the Chapter

You have reached the end this chapter. If you are still afloat, you may wish to review the following. Click on a subject that interests you to go to where it was discussed. To get back here, click on the "back arrow" button of your browser.

- Base two positional notation.
- Converting base 2 to base 10.
- Fixed point notation.
- Binary addition algorithm, with fixed point
- Limited precision of fixed point (and floating point).
- Shifting left.
- Converting base 10 to base 2
- 0.1 in binary
- Dangerous counting loops

contents Back to the main menu.

You have reached the end of the chapter.

created: 05/29/2002

CHAPTER 30 — IEEE 754 Floating Point

This chapter is about the **IEEE 754 floating point standard.** This is a standard for floating point numbers used in nearly all computers and programming languages since 1985. Jave floating point (on its virtual processor) and MIPS floating point (on real hardware) conform to this standard.

Chapter Topics:

- *Why use floating point?*
- *History of floating point?*
- *Scientific notation*
- *Precision of floating point*
- *IEEE 754 floating point standard*
- *Converting to floating point notation*

QUESTION 1:

(Review:) Is it absolutely essential that a processor support floating point data?

Next

A good answer might be:

No. All a processor needs for data are basic bit-manipulating instructions. Everything else can be built up from them.

Why Floating Point?

Floating point representation makes numerical computation much easier. You could write all your programs using integers or fixed-point representations, but this is tedious and error-prone. For example, you could use integers and program with the "understanding" that the integer in each variable is 100 times bigger than the number it represents. As long as you are consistent, every thing works.

This is actually the same as using fixed point notation. In fixed point binary notation each variable holds an integer that is, say 64 times bigger than the number it represents. (This is what our fixed point method of the previous chapter did).

But it is very hard to stay consistent. The programmer must remember where the decimal (or binary) point "really is" in each number. Sometimes one program needs to deal with several different ranges of numbers. Consider a program must deal with both the measurements that describe the dimensions on a silicon chip (say 0.000000010 to 0.000010000 meters) and also the clock speeds it uses (say 500000000.0 to 2000000000.) It is hard to find a single non-floating point method that can represent both ranges.

Notice that in writing those numbers (0.000000010, 0.000010000, 500000000.0, 2000000000.) I was able to put the decimal point where it was needed in each number.

QUESTION 2:

(Take a guess:) When the decimal point moves to where it is needed in an expression, the decimal point is said to ____________.

A good answer might be:

Float.

Floating Point

The essential idea of floating point representation is that there a fixed number of bits is used (usually 32 or 64) and that the binary point "floats" to where it is needed in the expression. Of course there is no actual binary point; some of the bits are used to say where the binary point is considered to lie.

Floating point expressions can represent numbers that are very small and very large. When a floating point calculation is performed, the binary point floats to the correct position in the result. The programmer does not need to explicitly keep track of it.

QUESTION 3:

Is floating point a very modern or a very old idea?

Next

A good answer might be:

Old.

Early Floating Point

The first digital computer (Konrad Zuse's Z1 built in 1938) was also the first computer to use floating point representation. Early computers were built to do engineering and scientific calculation so it is no surprise that the invention of floating point happened at the same time. In later years it was realized that computers are very useful things, and not just for calculation. Often floating point was not supported.

Many early minicomputers and microprocessors did not directly support floating point in hardware. For example, Intel processor chips before the 80486 did not directly support floating point. A floating point operation in a program (written in C, say) was compiled into a sequence of bit-manipulation instructions that did the required operations.

MIPS processors are very strong on floating point, and have supported the IEEE standard from their (and its) beginning. MIPS are for high-end engineering and graphics workstations and are famous for their fast floating point.

QUESTION 4:

(Not very hard thought question:) Do you imagine that in (say) 1975 that the floating point methods used on IBM mainframes was the same as on DEC minicomputers?

Next

A good answer might be:

No.

Floating Point Compatibility

Up until 1985 each hardware manufacturer had its own type of floating point. Worse, different machines from the same manufacturer might have different types of floating point! And when floating point was not supported in the hardware, the different compilers emulated different floating point types.

The situation was awful. Consider a magnetic tape full of data written by an IBM mainframe. Now the tape must be read by a DEC minicomputer. Assume that the bits on the tape can be read in correctly. But DEC used different type of floating point than IBM interpretting thos bits is hard! This problem was very common, and with "legacy data" is still common. NASA (the USA National Aeronautical and Space Administration) has warehouses full of tapes from various space missions written in many different formats.

QUESTION 5:

Should the same numerical calculation, performed on two different computers, give the same result on each?

Next

A good answer might be:

Yes. But before a floating point standard this was sometimes not true.

Scientific Notation

To address this problem the IEEE (Institute of Electrical and Electronics Engineers) created a standard for floating point. This is the IEEE 754 standard, released in 1985 after many years of development. All hardware and software since then follow this standard (many processors also support older standards as well).

The idea of floating point comes from **scientific notation** for numbers. You have probably seen this before (and may already be shuddering). Here is a number in scientific notation:

```
1.38502 × 10^03
---+---     |
   |        +-- exponent
   |
   +--- mantissa
```

The mantissa is always has a decimal point after the first digit. The above expression means:

$1.38502 \times 10^{03} = 1385.02$

The decimal point "floats" to where it belongs. In effect, the mantissa gives the number and the exponent says where the decimal point should go.

QUESTION 6:

Write the following in scientific notation: 243.92

Next

A good answer might be:

243.92 = 2.4392 × 10^{02}

Negative Exponents

The exponent shows by what power of 10 to multiply the mantissa by, or (the same thing) how far to float the decimal point. 10^{+n} means shift the decimal point n places **right**. 10^{-n} means shift the decimal point n places **left**.

1.82371	×	10^{1}	=	1.82371	×	10	=	18.2371	(shift right 1 place)
1.82371	×	10^{-1}	=	1.82371	×	.1	=	.182371	(shift left 1 place)
1.82371	×	10^{2}	=	1.82371	×	100	=	182.371	(shift right 2 places)
1.82371	×	10^{-2}	=	1.82371	×	.01	=	.0182371	(shift left 2 places)

QUESTION 7:

Write 0.000456 in scientific notation.

Next

A good answer might be:

0.000456 = 4.56×10^{-4}

There must be only one digit to the left of the decimal place, so an answer like 45.6×10^{-5} is not in the correct format.

Negative Values

The notation for a negative number has a negative sign in front of the first digit:

$-1.38502 \times 10^{03} = -(1.38502 \times 10^{03}) = -1385.02$

The notation for a small negative number has two negative signs: one to negate the entire number and one (in the exponent) to shift the decimal point left:

$-1.38502 \times 10^{-5} = -(1.38502 \times 10^{-3}) = -0.0000138502$

QUESTION 8:

Match the numbers in column I with the scientific notation in column II.

Next

I	click the button		II
2.034	matches		(a) -2.034×10^{0}
-2.034	matches		(b) 2.034×10^{2}
203.4	matches		(c) 2.034×10^{0}
.002034	matches		(d) -2.034×10^{1}

-20.34	matches		(e) -2.034×10^{-2}
-.02034	matches		(f) 2.034×10^{-3}

A good answer might be:

Click!

e

Writing the cross × and the superscript exponent is tedious, and not suitable for programs. A slighltly altered notation is used in computer programs and in input and output text:

$1.38502 \times 10^{+03} =$ 1.38502e+03
$1.38502 \times 10^{-03} =$ 1.38502e-03
$-1.38502 \times 10^{+03} =$ -1.38502e+03
$-1.38502 \times 10^{-03} =$ -1.38502e-03

Caution: For SPIM, only a small "e" is allowed in floating point constants. (Some software systems, such as Java, allow both upper and lower case e's).

QUESTION 9:

How many digits are in each of the above expressions? How many digits are in the mantissa? How many in the exponent?

Next

A good answer might be:

How many digits are in each of the above expressions? 8 How many digits are in the mantissa? 6 How many in the exponent? 2

Fixed Number of Digits

This is getting close to the idea of floating point. Each expression has eight digits. Two of the digits are used to "float" the decimal point. The remaining six show the digits of the number.

The number of digits in the mantissa is called the **precision** of the floating point number. When we start talking about binary floating point the precision will be the number of bits in the mantissa. With a 32-bit single-precision floating point value the mantissa represents 24 bits, so single precsion floats have 24 bits of precision.

Caution: sometimes in casual speach people say that single precision values have 32 bits of precision, but this is not actually correct.

Another Caution: the mantissa uses 23 bits to represent its 24 bits of precision. This trick will be discussed shortly.

QUESTION 10:

How many possible mantissas can be represented in single precision?

Next

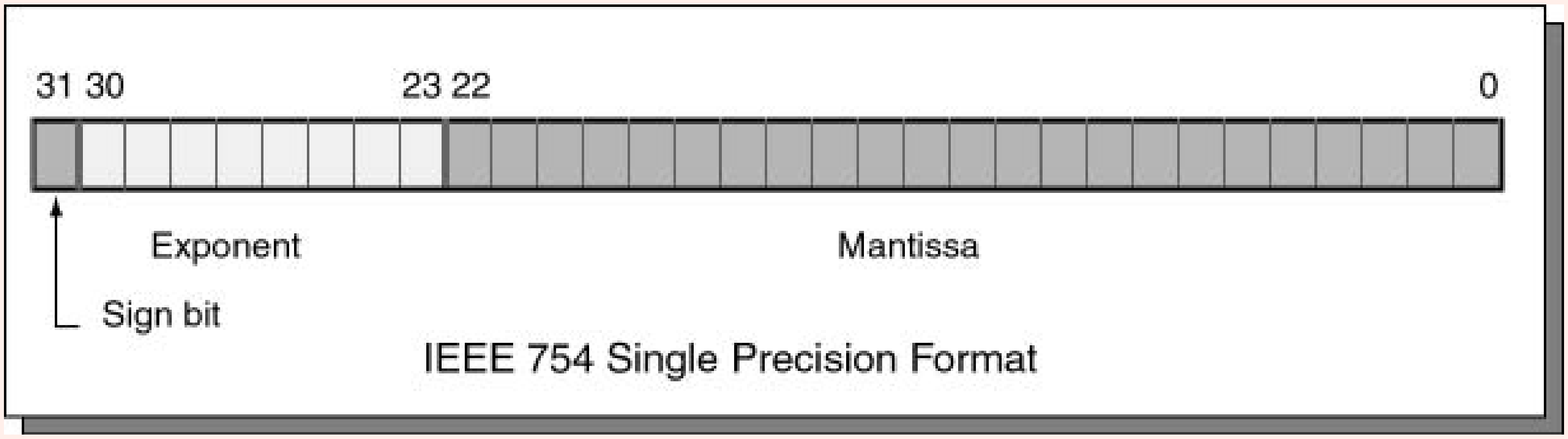
How many possible mantissas can be represented in single precision?

A good answer might be:

A mantissa has 23 bits so there are 2^{23} patterns possible.

IEEE 754 Floating Point

An IEEE 754 single-precision floating point value is 32 bits long. The bits are divided into fixed-sized fields as follows:



Bits 0 through 22 are for the mantissa; bits 23 through 30 are for the exponent; and bit 31 is the sign bit. The mantissa and exponent fields work like the similar parts in scientific notation (details follow). The sign bit gives the sign of the entire expression: a 0 bit means positive and a 1 bit means negative.

QUESTION 11:

You have seen the MIPS instruction `bltz` which branches if the 32-bit integer in a register is less than zero (negative). It does this by looking at bit 31 of the two's complement integer and branching if it is set.

Next

Does this instruction work with single precision values?

A good answer might be:

Yes. The sign bit of 32-bit integers and of 32-bit floats is set (to one) when the value is negative and clear (to zero) when the value is positive or zero.

More on Ms. Mantissa

The mantissa represents a number in 24-bit base two positional notation that looks like this:

1.xxxxxxxxxxxxxxxxxxxxxxx (x = 0 or 1; there are 23 x's)

Very tricky: The mantissa represents the 23-bit binary fraction part of a 24-bit number that starts with a "1". The 2^0 place (the one's place) is presumed to contain a 1 and *is not present in the mantissa.* This trick gives us 24 bits of precision with only 23 bits.

For example, the binary number 1.11110000101101101010001 is represented as 11110000101101101010001

QUESTION 12:

If you had a daughter would you name her Mantissa?

Next

A good answer might be:

What a sweet name!

How the Mantissa Works

Since leading zeros of a binary number can be dropped, it is safe to not actually include the first 1. There will always be one, so there is no need to show it. The exponent is adusted so that the most significant 1 is in the one's place of the mantissa. For example: say that you wish to represent 00011.00101101 as a float.

- Drop the leading zeros: 11.00101101 with a leading one.
- Shift the binary point so that the leading one is in the one's place: 1.100101101×2^1 (the exponent shows the shift)
- Drop the leading one: 100101101×2^1
- Add zero bits so that there are 23 bits in total: $10010110100000000000000 \times 2^1$
- The mantissa is: 10010110100000000000000

QUESTION 13:

Is the mantissa always zero or positive?

Next

A good answer might be:

Yes. (The sign bit is not part of the mantissa).

Practice

You must be eager to try this yourself!! Here is an opportunity. The number 7.46875_{10} written in base two positional notation is 00111.01111 (the leading zeros are not needed). What is the mantissa of this number?

Represent this number:		00111.01111
Drop the leading zeros:	?	
Shift the binary point:	?	
Drop the leading one:	?	
Right-pad with zero:	?	
The Mantissa is:	?	

QUESTION 14:

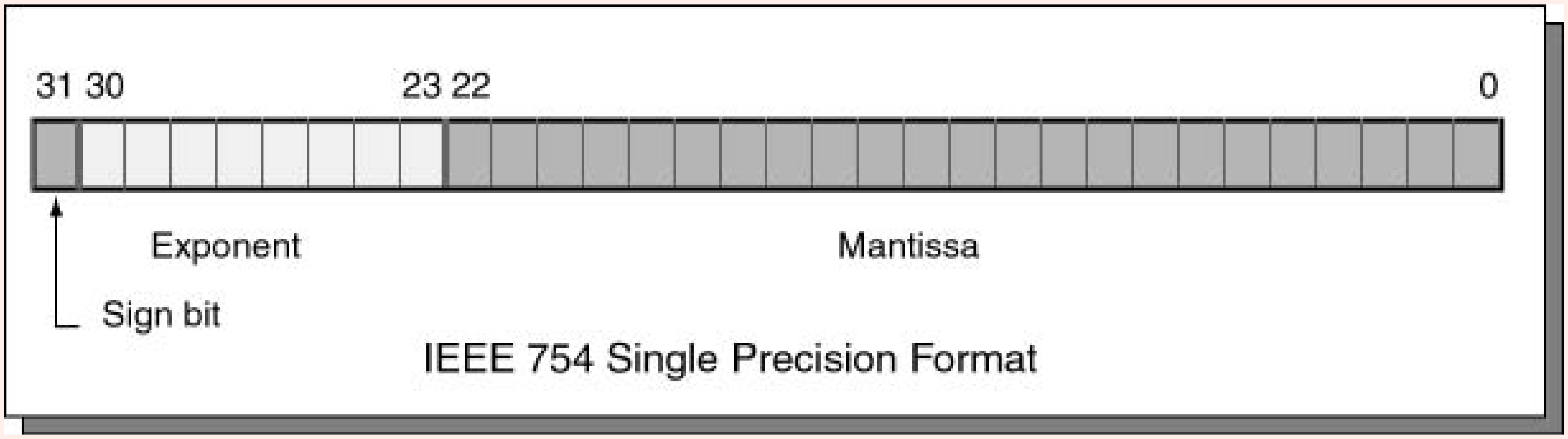
What part of the single precision float represents the shifted binary point?

Next

A good answer might be:

The exponent.

The Exponent



The eight bits 23 through 30 contain the exponent. The exponent is an integer, but may be negative, zero, or positive. You might guess that an 8-bit two's complement integer would work perfectly for this, but a different type of notation is used.

The exponent is expressed using a **biased integer**. This is an *unsigned* binary integer that has +127 added to it.

- A biased exponent of +127 represents the actual exponent 0.
- A biased exponent of +128 represents the actual exponent 1.
- A biased exponent of +121 represents the actual exponent -1.

Exceptions: (1) the biased exponent of 0 is used when the entire float is zero. (2) the biased exponent of 255 is used to signal various problems such as division by zero.

QUESTION 15:

What is the mantissa when the entire float is zero?

Next

What is the mantissa when the entire float is zero?

A good answer might be:

Zero.

Exponent Practice

Of course the sign bit is zero when the number is zero. A **floating point zero** consists of 32 zero bits. The MIPS instructions that test for integer zero can test for floating point zero.

actual exponent		biased exponent	eight bit exponent
0	?		
1	?		
10	?		
-10	?		
127	?		
-127	?		

QUESTION 16:

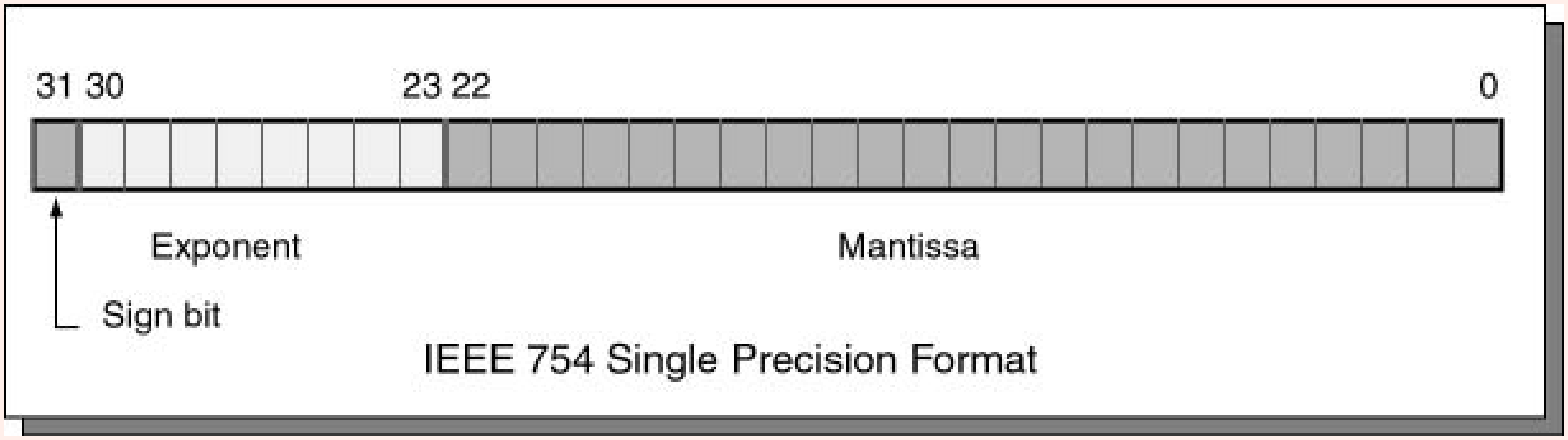
How is the exponent -127 represented?

Next

A good answer might be:

It is not represented. (Remember, the biased exponent 0000 0000 is used for the value zero).

Float Formula



Here is a formula that summarizes the past several pages. In it, s is the sign bit (0 or 1), M is the mantissa (.000...000 to .1111...111) and E is the exponent (-126 to +127).

$$\text{value} = (-1)^{s} \times 1.M \times 2^{E-127}$$

To convert a paper-and-pencil number into IEEE floating point, fill in each piece of the picture. Let us represent 1.0 to float.

QUESTION 17:

- What is the sign bit of 1.0?
- What is the mantissa of 1.0? (Hint: remember the assumed "1." preceding the mantissa)
- What is the actual exponent of 1.0?
- What is the biased of 1.0? (Hint: add the actual exponent to 127)

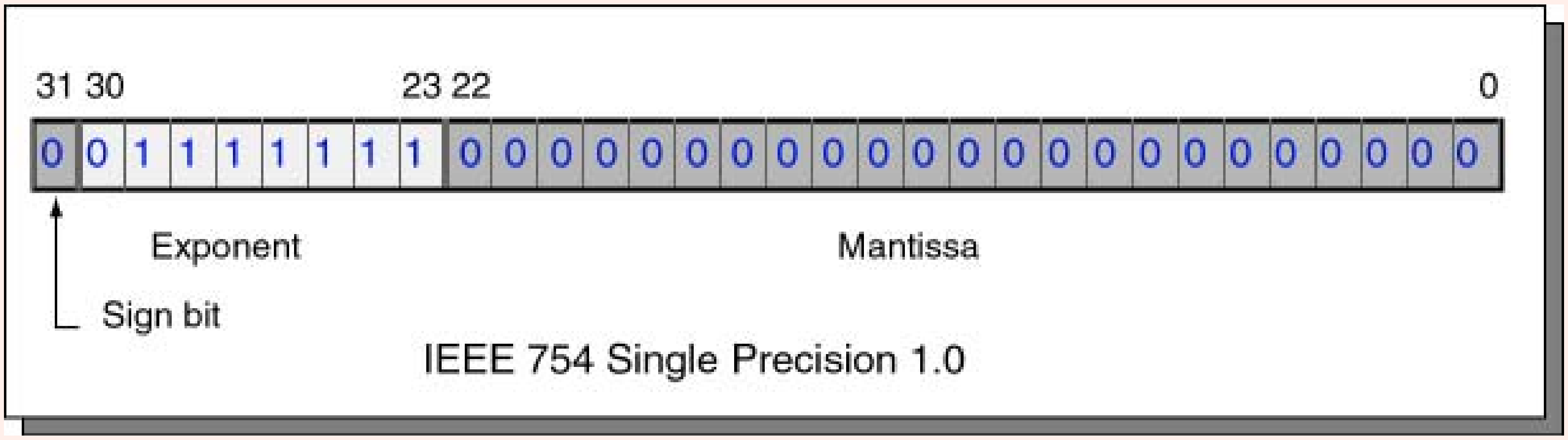
Next

A good answer might be:

- What is the sign bit of 1.0? 0 (for positive)
- What is the mantissa of 1.0? 000 0000 0000 0000 0000 0000
- What is the actual exponent of 1.0? 0
- What is the biased of 1.0? 127_{10} = $0111\ 1111_2$

1.0

Fitting each of these pieces into their proper fields gives us the full 32 bits:



If the bit pattern is written in hexadecimal it is 0x3F800000. To check that this is correct, use SPIM. Write a program that declares a floating point 1.0, then run SPIM.

```
## Program to represent 1.0
        .text
        .globl  main
main:
        .data
val:    .float    1.0
## End of file
```

QUESTION 18:

When you load this program in SPIM, where will the 1.0 be?

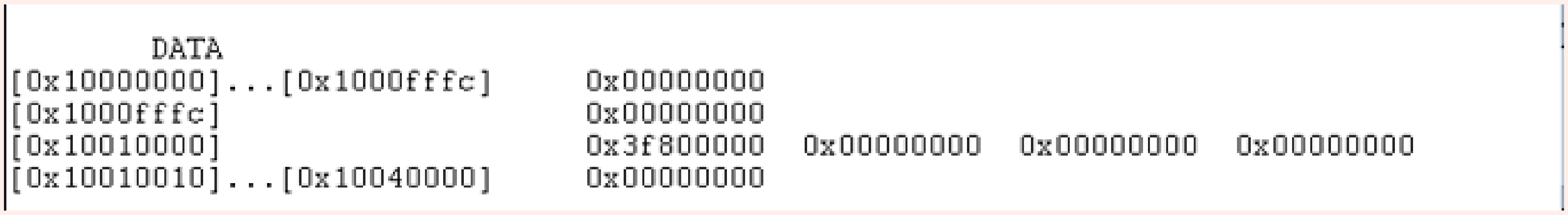
Next

A good answer might be:

The 0x3F800000 will be in the DATA section in the 32 bits starting at address 0x10010000.

SPIM Data

Of course, to the memory system a 32-bit floating point value is just another 32 bit pattern. Here is (part) of SPIM's display showing this:



```
         DATA
[0x10000000]...[0x1000fffc]      0x00000000
[0x1000fffc]                     0x00000000
[0x10010000]                     0x3f800000  0x00000000  0x00000000  0x00000000
[0x10010010]...[0x10040000]      0x00000000
```

QUESTION 19:

(Review:) Can -0.1 be represented precisely as a 32-bit float?

Next

A good answer might be:

No. Remember from the last chapter that 0.1 can't be represented precisely in binary. Using floating point representation does not change anything. It does not matter that the number is negative.

-0.1

But let us proceed to see how -0.1 is (imprecisely) represented.

- The sign bit of -0.1 is 1 (for negative).
- The binary fraction for 0.1 (from the previous chapter is) 0.0001100110011001100110011001100...
- The mantissa of 0.1 is:
  - Shift the leading bit into the one's place: 1.1001100110011001100110011001100...
  - The shift was 4 places left, for an exponent of -4
  - Drop bit in the one's place: .1001100110011001100110011001100...
  - Retain 23 bits: 100 1100 1100 1100 1100 1100
- The actual exponent is -4
- The biased exponent is 127-4 = 123 = 0111 1011

QUESTION 20:

Here are the bits written out:

+-	exponent								Mantissa																						
1	0	1	1	1	1	0	1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0

Write out the bit pattern as hexadecimal.

A good answer might be:

+-	**exponent**								**Mantissa**																						
1	0	1	1	1	1	0	1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0
B				D				C				C				C				C				C				C			

End of the Chapter

On SPIM, a constant of -0.1 is represented as 0xBDCCCCCD because the value is rounded up one bit to form the "D" at the end.

You have reached the end this chapter. You may wish to review the following. Point your mouse to a subject that interests you to go to where it was discussed.

- The advantages of floating point.
- History of floating point.
- Scientific notation for numbers.
- Precision of a floation point number.
- IEEE 754 Floating Point Standard
- The mantissa.
- The exponent.
- Floating point zero.
- Formula for floating point.

contents Back to the main menu.

You have reached the end of the chapter.

created: 05/29/2002

CHAPTER 31 — Floating Point Arithmetic on MIPS

The MIPS chips use the IEEE 754 floating point standard, both the 32 bit and the 64 bit versions. However these notes cover only the 32 bit instructions. The 64 bit versions are similar.

Chapter Topics:

- *Floating point registers.*
- *Loading and storing floating point registers.*
- *Single and (some) double precision arithmetic.*
- *Data movement instructions.*
- *Reading and writing floating point.*

QUESTION 1:

The 64 bit (double precision) floating point numbers are better than 32 bit (single precision). But how are they better?

- Increased Precison?
- Greater Range of values convered?
- Both?

Next

A good answer might be:

Both. Double precision numbers have more bits for the exponent (so the range of values is increased) and more bits in the mantissa (so the precision is increased).

MIPS Floating Point

Floating point on MIPS was originally done in a separate chip called **coprocessor 1** (also called the FPA for Floating Point Accelerator). Modern MIPS chips include floating point operations on the main processor chip. But the instructions sometimes act as if there were still a separate chip.

MIPS instructions use the IEEE 754 standard. MIPS has 32 single precision (32 bit) floating point registers.

- The registers are named $f0 – $f31
- $f0 is not special (it can hold any bit pattern, not just zero).
- Single precision floating point load, store, arithmetic, and other instructions work with these registers.

QUESTION 2:

(Memory Test:) Why is $f0 not hard-wired to floating point zero, like register $0 is hard-wired to integer zero?

Next

A good answer might be:

Because floating point zero has the same representation as integer zero, 32 zero-bits.

Double Precision

MIPS also has hardware for double precision (64 bit) floating point operations. For this, it uses pairs of single precision registers to hold operands. There are 16 pairs, named $f0, $f2, — $f30. Only the even numbered register is specified in a double precision instruction; the odd numbered register of the pair is included automatically.

Some MIPS processors allow only even-numbered registers ($f0, $f2,...) for single precision instructions. However SPIM allows all 32 registers in single precision instructions. These notes follow that usage.

QUESTION 3:

Remember the load delay, where it takes an extra machine cycle after a load instruction before integer data reaches a general purpose register? Do you think there is a load delay for floating point load operations?

A good answer might be:

Yes.

Single Precision Load

Actual hardware has a delay between a load instruction and the time when the data reaches the register. In SPIM there is an option that disables the load delay. For this chapter, set this option. (Floating point is tricky enough already).

Loading a single precision value is done with a pseudoinstruction:

```
l.s   fd,addr     --- load register fd from addr
```

This instruction loads 32 bits of data from address addr into floating point register $fd (where d is 0, 1, 2, ..., 15). Whatever 32 bits are located at addr are copied into $fd. If the data makes no sense as a floating point value, that is OK for this instruction. Later on the mistake will be caught, when floating point operations are attempted.

QUESTION 4:

Every one of the 2^{32} patterns that 32 bits can form is a legal two's complement integer.

Are some of there patterns not legal as a floating point value? Hint: think about the exponent and a zero mantissa.

A good answer might be:

It is not legal to have a non-zero exponent with a matissa that is zero. There are other nonsense combinations that are also illegal.

Single Precision Store

Sometimes the floating point registers are used as temporary registers for integer data. For example, rather than storing a temporary value to memory, you can copy it to an unused floating point register. This is OK, as long as you don't try to do math with them.

The single precision store pseudoinstruction is similar:

```
s.s   fd,addr     --- store register fd to addr
```

Whatever 32 bits are in fd are copied to addr

In both of these pseudoinstructions the address addr can be an ordinary symbolic address, or an indexed address. (As of this writing, these notes have not covered indexed addresses).

QUESTION 5:

Why would you want to copy a temporary integer value to a floating point register?

Next

A good answer might be:

To free up a general purpose register for some other use.

Floating Point Load Immediate

There is a floating point **load immediate** instruction. This instruction allows you to load a register with a constant value using only one line of assembly language. At the machine code level it corresponds to several machine instructions.

```
li.s   fd,val     # (pseudoinstruction)
                  # load register $fd with val
```

Here is a code snippet showing this:

```
li.s    $f1,1.0           # $f1 = constant 1.0
li.s    $f2,2.0           # $f2 = constant 2.0
li.s    $f10,1.0e-5       # $f10 = 0.00001
```

QUESTION 6:

Could first instruction be written li.s $f1,1# ?

Next

A good answer might be:

No. The decimal point is needed in the constant (at least for SPIM).

Example Program

Here is a program that exchanges (swaps) the floating point values at valA and valB. Notice how the two floating point values are written. The first in the ordinary style; the second in scientific notation.

```
## swap.asm
##
## Exchange the values in valA and valB

        .text
        .globl  main

main:
        l.s     $f0,valA       # $f0  <-- valA
        l.s     $f1,valB       # $f1  <-- valB
        s.s     $f0,valB       # $f0  --> valB
        s.s     $f1,valA       # $f1  --> valA

        li      $v0,10         # code 10 == exit
        syscall                # Return to OS.

        .data
valA:   .float  8.32           # 32 bit floating point value
valB:   .float  -0.6234e4      # 32 bit floating point value
                               # small 'e' only
## End of file
```

QUESTION 7:

Could this program have been written using general purpose registers?

Next

A good answer might be:

Yes. The floating point registers in this program were used only as a temporary holding space.

Full-Word Aligned

For both single precision load and store instructions the memory address must be full-word alligned. It must be a multiple of four. Ordinarily this is not a problem. The assembler takes care of this.

QUESTION 8:

Do you think that the SPIM system calls that print integers can be used for floating point values?

Next

A good answer might be:

No. They expect a different type of representation.

Floating Point System Calls

Here is the complete list of SPIM services. The floating point services expect the data to be in floating point format. Notice that the calls for double use an even-odd pair of registers.

Service	Code	Arguments	Returned Value
print integer	1	$a0 == integer	
print float	2	$f12 == float	
print double	3	($f12, $f13) == double	
print string	4	$a0 == address of string	
read integer	5		$v0 <- - integer
read float	6		$f0 <- - float
read double	7		($f0, $f1) <- - double
read string	8	$a0 == buffer address $a1 == buffer length	
allocate memory	9	$a0 == number of bytes	$v0 <- - address
exit	10		

QUESTION 9:

(Review:) Into what register do you put the service code before executing a syscall

Next

A good answer might be:

Register $v0

Mistake

Depending on the service, you may have to place arguments in other registers as well. The following example program prints out a floating point value. It first does this correctly (using system call 2). Then does it incorrectly uses the integer print service (system call 1). Of course, the 32 bits of the floating point value can be interpretted as an integer, so system call 2 innocently does what we asked it to do.

```
## print.asm
##
## Print out a 32 bit pattern, first as a float,
## then as an integer.

        .text
        .globl  main

main:
        l.s     $f12,val        # get the float argument
        li      $v0,2           # code 2 == print float
        syscall                 # (correct)

        li      $v0,4           # print
        la      $a0,lfeed       # line separator
        syscall

        lw      $a0,val         # use the float as a int
        li      $v0,1           # code 2 == print int
        syscall                 # (mistake)

        li      $v0,10          # code 10 == exit
        syscall                 # Return to OS.

        .data
val :   .float  -8.32           # negative value
lfeed:  .asciiz "\n"
## End of file
```

QUESTION 10:

Will the assembler catch this error? Will the program bomb when it runs?

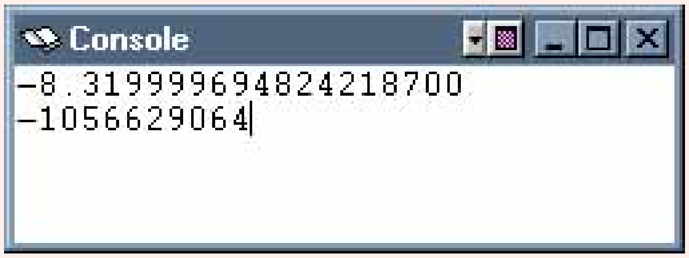
Next

A good answer might be:

The assembler will not catch this mistake, and the program will run fine.

No Type Checking

This type of mistake often happens when programming in "C" where type checking is weak. Sometimes the wrong type can be passed to a function (such as printf) and odd things happen. Here is the output of the program:



```
-8.319999694824218700
-1056629064
```

Compilers that keep track of the data types of values and that make sure that the correct types are used as arguments do **strong type checking**. Java is strongly typed. In assembly language type checking is largely absent.

QUESTION 11:

The first line of output shows the 32 bits interpretted as a single precision floating point value. Is this output correct? (Hint: look at the program on the previous page).

A good answer might be:

Maybe Not. The source file for the program specified the constant -8.32 but the program printed -8.3199996948242187OO

Precision of Single Precision Floats

There are two things wrong: (1) the value -8.32 can not be represented exactly in binary, and (2) SPIM prints out way too many digits.

Single precision floats have (recall) only 24 bits of precision. This is the equivalent of 7 to 8 decimal digits. SPIM prints out many more digits than are actually represented. This is incorrect. SPIM should have printed -8.319999 to the window.

The 7 or 8 decimal digits of precision is much worse than most electronic calculators. It is usually unwise to use single precision floating point in programs. (But these chapters use it since the goal is to explain concepts, not to write production grade programs). Double precision has 15 or 16 decimal places of precision.

QUESTION 12:

How many decimal places of precision does Java's primitive type float have?
How many decimal places of precision does C's data type float have?

A good answer might be:

The answer is the same for both: 7 or 8 places, same as MIPS. All three use the same IEEE standard for single precision float. (But be careful: some C compilers allow you to specify how many bits of precision you want for various data types. Also, C implemented on a weird machine might be non-standard).

Single Precision Arithmetic

Here are some single precision arithmetic instructions. All of these correspond to a single machine instruction. The double precision version of these instructions has a "d" in place of the "s". So `add.s` becomes `add.d` and corresponds to the machine code that adds double precision.

Instruction	Operation
abs.s fd,fs	$fd = \|$fs|
add.s fd,fs,ft	$fd = $fs + $ft
sub.s fd,fs,ft	$fd = $fs - $ft
mul.s fd,fs,ft	$fd = $fs * $ft
div.s fd,fs,ft	$fd = $fs / $ft
neg.s fd,fs	$fd = -$fs

The first instruction computes the absolute value (makes a positive value) of the value in register `$fs`

If the data in an operand register is illegal or an illegal operation is performed (such as divide by zero) an exception is raised. The IEEE 752 standard describes what is done in these situations. This will be discussed in a later chapter, not yet written (June 2002).

QUESTION 13:

(Thought Question:) How does the `abs.s` instruction alter the 32-bit pattern of the float?

Next

A good answer might be:

It clears (makes zero) the sign bit, bit 31. All the other bits stay the same.

Data Movement

And the neg.s instruction toggles the sign bit. Here are three data movement instructions:

Instruction	Operation
mov.s fd, fs	copy 32 bits from float register $fs to float register $fd
mtc1 rs, fd	move to coprocessor 1 copy 32 bits from general register $rs to float register $fd. No data conversion is done.
mfc1 rd, fs	move from coprocessor 1 copy 32 bits from float register $fs to general register $rd. No data conversion is done.

These instructions merely copy bit patterns between registers. The pattern is not altered. With the mftc1 instruction, the IEEE formatted float in a floating point register is copied as is to a general purpose register. So a complicated calculation with integers can use float registers for intermediate results. And a complicated calculation with floats can use general purpose resisters the same way.

QUESTION 14:

Can an arithmetic instruction use both floating point and general purpose registers?

Next

A good answer might be:

No.

Example Program

The example program computes the value of $ax^2 + bx + c$. It starts by asking the user for x:

```
        .text
        .globl main

main:   # read input
        la      $a0,prompt          # prompt user for x
        li      $v0,4               # print string
        syscall

        li      $v0,6               # read single
        syscall                     # $f0 <-- x

        # evaluate the quadratic
        . . . . .

        .data
        . . . . .
prompt: .asciiz "Enter x: "
```

After the syscall the user's value is in $v0. The next section of the program does the calculation.

QUESTION 15:

Should a, b, and c be integer constants or floating point constants?

A good answer might be:

Floating point constants

Evaluation

Here is the part of the code that evaluates the polynomial. Remember that x is in $f0.

```
        . . . . .
        # evaluate the quadratic
        l.s     $f2,a             # sum = a
        mul.s   $f2,$f2,$f0       # sum = ax
        l.s     $f4,bb            # get b
        add.s   $f2,$f2,$f4       # sum = ax + b
        mul.s   $f2,$f2,$f0       # sum = (ax+b)x = ax^2 +bx
        l.s     $f4,c             # get c
        add.s   $f2,$f2,$f4       # sum = ax^2 + bx + c
        . . . . . .

        .data
a:      .float  1.0
bb:      .float  1.0
c:      .float  1.0
```

The assembler objected to the symbolic address "b" (because there is a mnemonic instruction "b" for branch) so I used "bb" instead.

The polynomial is evaluated from left to right. First ax + b is calculated. Then that is multiplied by x and c is added in, giving axx + bx + c.

The value x^2 is not explicitly calculated. This way of calculating a polynomial is called **Horner's Method**. It is useful to have in your bag of tricks.

QUESTION 16:

Why (do you suppose) are the constants a, b, and c set to 1.0?

Next

A good answer might be:

To make debugging easy.

Printing Out the Result

Here is the final section of the code.

```
        . . . . .
        # print the result
        mov.s   $f12,$f2          # $f12 = argument
        li      $v0,2             # print single
        syscall

        la      $a0,newl          # new line
        li      $v0,4             # print string
        syscall

        li      $v0,10            # code 10 == exit
        syscall                   # Return to OS.

        .data
        . . . . .

    blank:  .asciiz " "
    newl:   .asciiz "\n"
        . . . . .
```

There is nothing new in this code.

QUESTION 17:

If $x = 0.1$ what is $ax^2 + bx + c$ when $a = b = c = 1$?

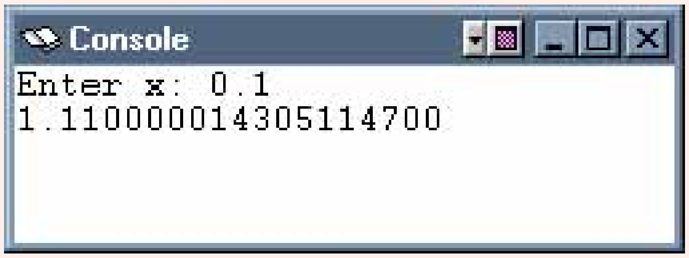
Next

A good answer might be:

0.01 + 0.1 + 1.0 = 1.11, you would hope.

Complete Program

Again, the SPIM simulator outputs too many digits. But recall, 0.1 can't be represented exactly, so that may be part of the problem.



```
Enter x: 0.1
1.11000001430511470O
```

Here is the complete program, suitable for you to copy into a text editor and to play with:

```
## float1.asm -- compute ax^2 + bx + c for user-input x

        .text
        .globl main

main:   # read input
        la      $a0,prompt        # prompt user for x
        li      $v0,4             # print string
        syscall

        li      $v0,6             # read single
        syscall                   # $f0 <-- x

        # evaluate the quadratic
        l.s     $f2,a             # sum = a
        mul.s   $f2,$f2,$f0       # sum = ax
        l.s     $f4,bb            # get b
        add.s   $f2,$f2,$f4       # sum = ax + b
        mul.s   $f2,$f2,$f0       # sum = (ax+b)x = ax^2 +bx
        l.s     $f4,c             # get c
        add.s   $f2,$f2,$f4       # sum = ax^2 + bx + c

        # print the result
        mov.s   $f12,$f2          # $f12 = argument
        li      $v0,2               # print single
        syscall

        la      $a0,newl            # new line
        li      $v0,4               # print string
        syscall

        jr      $ra                 # return to OS

##
##  Data Segment
##
        .data
a:      .float  1.0
bb:     .float  1.0
c:      .float  1.0

prompt: .asciiz "Enter x: "
blank:  .asciiz " "
newl:   .asciiz "\n"

## end of file
```

QUESTION 18:

Had enough?

Next

A good answer might be:

Precisely.

End of Chapter

You have reached the end this chapter. You may wish to review the following. Click on a subject that interests you to go to where it was discussed.

contents

You have reached the end of the chapter.

created: 05/31/2002

CHAPTER 32 — Floating Point Comparison Instructions

The MIPS FPA chip (now, usually part of the processor chip) has a condition bit that is set to 0 or 1 to indicate if a condition is false or true. Serval FPA instructions affect this bit and several CPU instructions test this bit.

Chapter Topics:

- *Comparison instructions: c.eq.s, c.lt.s, c.le.s*
- *Branch instructions:* bc1f, bc1t
- *The Condition bit*
- *Newton's method*

QUESTION 1:

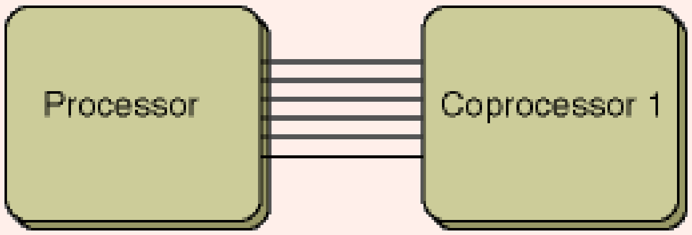
Are the FPA (floating point accelerator) and coprocessor 1 the same?

Next

A good answer might be:

Yes.

FPA Chip



In the early days of MIPS, floating point was done on a separate chip. Special instructions were used to bridge the gap between the two chips (such as the two move instructions mtc1 and mfc1 we have already seen). Although these days there is usually just one processor chip, some of the instructions act as if the chips were separate.

QUESTION 2:

(Thought Question:) Do you think that testing if two floating point values are equal is a common thing to do?

Next

A good answer might be:

Maybe.

Comparison Instructions

Testing if two floating point numbers are exactly equal is sometimes NOT a good idea. Floating point calculations are not exact. Sometimes values are not equal even though mathematically they should be. It is best to use "less than" or "less than or equal" instead of testing for exact equality. Here are some MIPS comparison instructions:

Instruction	Operation
c.eq.s fs, ft	set condition bit if $fs == $ft
c.lt.s fs, ft	set condition bit if $fs < $ft
c.le.s fs, ft	set condition bit if $fs <= $ft

The condition bit is set (made equal to one) if the condition is true. Otherwise the condition bit is cleared (made equal to zero). There are more comparison operations than these. But for these notes use just these three. The others involve the various IEEE 754 codes for exceptional results.

QUESTION 3:

There is no "greater than" instruction! What can we do?

Next

A good answer might be:

Use "less than" with the operations reversed.

Branch Instructions

The floating point branch instructions inspect the condition bit in the coprocessor. The bc1t instruction takes the branch if the bit is true (==1). The bc1t instruction takes the branch if the bit is false (==0). These are the only two floating point branch instructions.

Instruction	Operation
bc1t label	branch to label if the coprocessor 1 condition bit is true
bc1f label	branch to label if the coprocessor 1 condition bit is false

The branch instructions have a one instruction branch delay. A real machine has a nop instruction after each branch instruction (automatically put there by the assembler or by the compiler). For this chapter, let us not bother with this. When running the programs with SPIM, disable branch delays.

QUESTION 4:

Where does execution of the bc1t instruction take place? In the FPU or the CPU?

Next

A good answer might be:

The CPU. Although the instruction tests a bit in the FPU, it affects the program counter of the CPU and takes place in the CPU.

Example

The example program looks at two floating point values, A and B, and writes a message saying which one is smallest. First the program loads the two values A and B into registers:

```
main:   # get the values into registers
        l.s     $f0,A
        l.s     $f2,B
        . . . .

A:      .float  4.830
B:      .float  1.012
        . . . .
```

Then the code tests if A<B, or B<A. If neither of these is true, then it must be that B==A.

```
        . . . .
        c.lt.s  $f0,$f2         # is A < B?
        bc1t    printA          # yes -- print A

        c.lt.s  $f2,$f0         # is B < A?
        bc1t    printB          # yes -- print B

        la      $a0,EQmsg       # otherwise
        li      $v0,4           # they are equal
        . . . .
```

QUESTION 5:

The three outcomes can be detected by other conditionals. Here is another arrangement of the code:

Next

```
            c.______.s  $f0,$f2      # is A = B?
```

```
        bc1t    printEQ          # yes -- print equal

        c.______.s  $f2,$f0       # is B < A?
        bc1t    printB           # yes -- print B

        bc1t    printA           # otherwise A < B
```

Fill in the blanks. You may wish to consult the list of floating point comparison instructions.

A good answer might be:

```
c.eq.s  $f0,$f2         # is A = B?
bc1t    printEQ         # yes -- print equal

c.lt.s  $f2,$f0         # is B < A?
bc1t    printB          # yes -- print B

bc1t    printA          # otherwise A < B
```

The Complete Program

Here is the complete program. There is nothing new in the remainder of the code.

```
## min.asm --- determine the min of two floats
##
        .text
        .globl main

main:   # get the values into registers
        l.s     $f0,A
        l.s     $f2,B

        c.lt.s  $f0,$f2         # is A < B?
        bc1t    printA          # yes -- print A
        c.lt.s  $f2,$f0         # is B < A?
        bc1t    printB          # yes -- print B

        la      $a0,EQmsg       # otherwise
        li      $v0,4           # they are equal
        syscall
        mov.s   $f12,$f0        # print one of them
        b       prtnum

printA: la      $a0,Amsg        # message for A
        li      $v0,4
        syscall
        mov.s   $f12,$f0        # print A
        b       prtnum

printB: la      $a0,Bmsg        # message for A
        li      $v0,4
        syscall
        mov.s   $f12,$f2          # print A

prtnum: li      $v0,2             # print single
        syscall
        la      $a0,newl
        li      $v0,4             # print new line
        syscall
        jr      $ra               # return to OS

        .data

A:      .float  4.830
B:      .float  1.012
Amsg:   .asciiz "A is smallest: "
Bmsg:   .asciiz "B is smallest: "
EQmsg:  .asciiz "They are equal: "
newl:   .asciiz "\n"
```

QUESTION 6:

Do you think that the condition bit (of the FPU) holds its value until it is altered by another comparision instruction?

Next

A good answer might be:

Yes.

Condition Bit Holds its Value

The condition bit is like a one bit register that continues to hold the result of a comparison until another one is made. The following code ensures that register $f12 has the minimum of $f0 or $f2. If they are equal, then $f12 gets the value of both.

```
main:   l.s     $f0,A           # get the values
        l.s     $f2,B           # into registers

        c.lt.s  $f0,$f2         # is A < B?
        mov.s   $f0,$f12        # move A to $f12
                                # (condition bit holds its value)
        bc1t    common          # otherwise
        mov.s   $f2,$f12        # move B to $f12
common:
```

The above code is contrived; it would be better to reverse the third and fourth statements. However, sometimes it is very useful to hold the condition bit's value for several instructions before using it.

QUESTION 7:

(Review:) should c.eq.s be used to implement a while loop?

A good answer might be:

No, not usually. With the inherent inaccuracy of floating point this risks creating a non-terminating loop. This next example shows a while loop.

Newton's Method

Newton's method is a common way to compute the square root of a number. Say that n is the number and that x is an approximation to the square root of n. Then:

$$x' = (1/2)(x + n/x)$$

x' is an even better approximation to the square root. The reasons for this are buried in your calculus book (which you, perhaps, have buried in the darkest corner of your basement). But, to make the formula plausible, look what happens if the approximation x happens to be exactly the square root:

$$x' = (1/2)(x + n/x) = (1/2)(n^{0.5} + n/n^{0.5})$$
$$= (1/2)(n^{0.5} + n^{0.5}) = n^{0.5}$$

If x gets the exact value, it stays fixed at that value.

QUESTION 8:

Try it. Say that n == 4 and that our first appoximation to the square root is x == 1. Use the formula to get the next approximation:

$$x' = (1/2)(1 + 4/1)$$

Next

A good answer might be:

x = 2.5, a better approximation

Start of the Program

Now using x = 2.5 in the formula yields:

x' = (1/2)(2.5 + 4/2.5) = (1/2)(2.5 + 1.6) = (1/2)(4.1) = 2.05

The program will repeatedly use the formula to calculate increasingly accurate appoximations to the square root. Here is the start of the program:

```
## newton.asm -- compute sqrt(n)

## given an approximation x to sqrt(n),
## an improved approximation is:

## x' = (1/2)(x + n/x)

## $f0  ---  n
## $f1  ---  1.0
## $f2  ---  2.0
## $f3  ---  x  : current approx.
## $f4  ---  x' : next approx.
## $f8  ---  temp
## $f10 ---  small value

        .text
        .globl main

main:

        l.s     $f0,n              # get n
        li.s    $f1,1.0            # constant 1.0
        li.s    $f2,2.0            # constant 2.0
        li.s    $f3,1.0            # x == first approx.
        li.s    $f10,1.0e-5        # accuracy limit
loop:
```

The program will be written so that it stops looping when no further improvement is possible. We can't expect to get the answer exactly correct.

QUESTION 9:

(Memory test:) How many decimal places of precision can be expected with floating point?

A good answer might be:

Six or seven. To be safe, count on no better than six digits of accuracy. This program uses five place accuracy.

Apply the Formula

The value in f10 is used to test if the five digits of accuracy have been reached. We will do this in a moment. Here is the part of the program that uses the formula:

```
## x' = (1/2)(x + n/x)

## $f0  ---  n
## $f1  ---  1.0
## $f2  ---  2.0
## $f3  ---  x  : current approx.
## $f4  ---  x' : next approx.
## $f8  ---  temp
## $f10 ---  small value

loop:
        ___.s   $f4,$f0           #  x' = n
        ___.s   $f4,$f4,$f3       #  x' = n/x
        ___.s   $f4,$f3,$f4       #  x' = x + n/x
        ___.s   $f3,$f4,$f2       #  x = (1/2)(x + n/x)
```

QUESTION 10:

But part of each menemonic is missing! Fill them in. You may wish to look at floating point operations.

A good answer might be:

See below.

Ending the Loop

```
## x' = (1/2)(x + n/x)

## $f0  ---  n
## $f1  ---  1.0
## $f2  ---  2.0
## $f3  ---  x  : current approx.
## $f4  ---  x' : next approx.
## $f8  ---  temp
## $f10 ---  small value

loop:
        mov.s   $f4,$f0           #  x' = n
        div.s   $f4,$f4,$f3       #  x' = n/x
        add.s   $f4,$f3,$f4       #  x' = x + n/x
        div.s   $f3,$f4,$f2       #  x = (1/2)(x + n/x)

        # test if loop should end
```

After calculting a new approximation it is time to see if the loop should end. This is more work than the actual calculation. We can't test if we have the *exact* answer yet, because that may never happen. Instead let us test if the current x is close to the square root of n

QUESTION 11:

Say that x is very close to $n^{0.5}$. What do you think will be the value of n/(x*x)?

Next

Say that x is very close to $n^{0.5}$. What do you think will be the value of $n/(x*x)$?

A good answer might be:

It should be very close to 1.0

Ending Test

But, due to the lack of precision with single precision, the best we can hope for is to calculate an x such that:

$$0.999999 < n/x*x < 1.00001$$

The left and the right bound in the above have 6 decimal digits of precision. If 1.00000 is subtracted from this value the result is:

$$-0.000001 < n/x*x - 1 < 0.00001$$

To be safe, terminate the loop when:

$$|n/x*x - 1| < 0.00001$$

... where $|x|$ means the absolute value of x.

QUESTION 12:

Is there a floating point absolute value instruction?

Next

A good answer might be:

Yes.

Code for Ending Test

Here is the code for ending the loop when |n/x*x - 1|< 0.00001.

```
## $f0  ---  n
## $f1  ---  1.0
## $f2  ---  2.0
## $f3  ---  x  : current approx.
## $f4  ---  x' : next approx.
## $f8  ---  temp
## $f10 ---  small value

        mul.s   $f8,$f3,$f3        # x^2
        div.s   $f8,$f0,$f8        # n/x^2
        sub.s   $f8,$f8,$f1        # n/x^2 - 1.0
        abs.s   $f8,$f8            # |n/x^2 - 1.0|
        c.lt.s  $f8,$f10           # |x^2 - n| < small ?
        bc1t    done               # yes: done

        j       loop

done:
```

QUESTION 13:

Is this program commercial quality code?

Next

A good answer might be:

Nowhere close. Numerical analysis is tricky. You should use a square root method from a package of mathematical subroutines if you really need one.

Complete Program

But it is good enough for a demonstration on the SPIM simulator. Here is the complete program. Copy it to your editor, save it to a file, and play with it in SPIM.

```
## newton.asm -- compute sqrt(n)

## given an approximation x to sqrt(n),
## an improved approximation is:

## x' = (1/2)(x + n/x)

## $f0  ---  n
## $f1  ---  1.0
## $f2  ---  2.0
## $f3  ---  x  : current approx.
## $f4  ---  x' : next approx.
## $f8  ---  temp

        .text
        .globl main

main:

        l.s     $f0,n             # get n
        li.s    $f1,1.0           # constant 1.0
        li.s    $f2,2.0           # constant 2.0
        li.s    $f3,1.0           # x == first approx.
        li.s    $f10,1.0e-5       # five figure accuracy

loop:
        mov.s   $f4,$f0           # x' = n
        div.s   $f4,$f4,$f3       # x' = n/x
        add.s   $f4,$f3,$f4       # x' = x + n/x
        div.s   $f3,$f4,$f2       # x    = (1/2)(x + n/x)

        mul.s   $f8,$f3,$f3       # x^2
        div.s   $f8,$f0,$f8         # n/x^2
        sub.s   $f8,$f8,$f1         # n/x^2 - 1.0
        abs.s   $f8,$f8             # |n/x^2 - 1.0|
        c.lt.s  $f8,$f10            # |x^2 - n| < small ?
        bc1t    done                # yes: done

        j       loop                # next approximation

done:
        mov.s   $f12,$f3            # print the result
        li      $v0,2
        syscall

        jr      $ra                 # return to OS

##
##  Data Segment
##
        .data
n:      .float  3.0

## end of file
```

QUESTION 14:

What result do you expect if this program calculated the square root of 4.000000e+6 ?

Next

What result do you expect if this program calculated the square root of 4.000000e+6 ?

A good answer might be:

You should expect 2.0000e+3 or so: five places of precision.

End of the Chapter

In fact, SPIM prints out a 25 digit answer. Most of the digits are meaningless.

You have reached the end this chapter. You may wish to root among the following topics. Click on a subject that interests you to go to where it was discussed.

- The floating point accelerator chip.
- Floating point comparison instructions.
- Floating point branch instructions.
- The condition bit.
- Newton's method.

contents

You have reached the end of the chapter.