7-24-2018

Computer Organization with MIPS

Seth D. Bergmann
Rowan University

DOI: 10.31986/issn.2689-0690_rdw.oer.1008

Computer Organization with MIPS

Seth D. Bergmann

August 9, 2019

Preface

This book is intended to be used for a first course in computer organization, or computer architecture. It assumes that all digital components can be constructed from fundamental logic gates.

The book begins with number representation schemes and assembly language for the MIPS architecture, including assembler directives, pseudo-operations, and floating point instructions. It then describes the machine language instruction formats, and shows the student how to translate an assembly language program to machine langauge.

This is followed by a chapter which describes how to construct an assembler for MIPS. This chapter may be omitted without loss of continuity.

There is then an introduction to boolean algebra and digital logic, followed by a design of the MIPS datapath. The book concludes with a description of the memory hierarchy, including cache memory, RAM, and virtual memory.

Each section concludes with a list of exercises (solutions are available to instructors who have adopted this text in a course).

This book is an open source book. That means that not only is the pdf version available (to potential students and teachers) for free download, but that the original (LaTeX) source files are also available (to potential authors and contributors). Based on the model of open source software, open source for textbooks is a relatively new paradigm in which many authors and contributors can cooperate to produce a high quality product, for no compensation. For details on the rationale of this new paradigm, and citations for other open source textbooks, see the journal *Publishing Research Quarterly*, Vol. 30, No. 1, March 2014. The source materials and pdf files of this book are licensed with the Creative Commons NonCommercial license, which means that they may be freely used, copied, or modified, but not for financial gain.

The source files for this book are available at `rdw.rowan.edu` (search for Bergmann) and at `cs.rowan.edu/∼bergmann/books`

Secondary Authors

Contributors

Contents

Chapter 1

Computers and Computer Programs

In driver education classes students are taught not only how to drive and the rules of the road, but they are also taught some fundamentals of the inner workings of the car - the four cycle engine, the distributor, the electrical system, etc. Strictly speaking it is not necessary to know these things in order to drive the car, but they are generally considered important enough for every driver to have a rudimentary understanding. When something fails, a mechanic may not be immediately available, and the driver who has some knowledge of what is under the hood will be better prepared to deal with the problem than the driver who is clueless.

Some computer scientists work with computer hardware (often in conjunction with software), but many work exclusively with software. Like automobile drivers they will be better prepared to deal with failures if they have some knowledge of what is 'under the hood'. In addition software developers who are hardware-savvy can produce more efficient software than those who are not.

For these reasons most computer science curricula include at least one hardware course. This book includes topics such as CPU design, datapath, the memory hierarchy, and assembly language. All of these are essential to a broad understanding of computer organization and design.

As a prototypical example of a computer, we use the MIPS [1] architecture. This architecture is complex enough that it is used in some real devices, yet it is simple enough to be understood and programmed by novices.

The computer is but one example of a *digital* device. By this we mean that at its most fundamental level it stores and works with binary values - zeros and ones. There is no other value in a digital device; all other information (numbers, keyboard characters, sound, images) are made up of sequences of zeros and ones, i.e. binary values. This is true of all digital devices - computers,

[1] Microprocessor without Interlocked Pipeline Stages. MIPS has been used primarily in embedded systems, routers, game consoles, etc.

tablets, phones, cameras, music players, game consoles, and electronic devices embedded in appliances, automobiles, medical equipment, etc. Each binary value is called a *bit* (binary digit).

1.1 Hardware Components

Here we provide a simplified description of the hardware components of a typical general purpose computer, such as those which are called 'desktop' or 'laptop' computers. These are not tablet systems, nor smart phones, which have a somewhat different design. Most of the components described here reside in the *system unit* which is the box housing the desktop or laptop computer.

1.1.1 Central Processing Unit

The *Central Processing Unit* (CPU) is probably the most important part of the computer. This is where computations take place, and this is where decisions are made concerning the sequence in which computations are made. The CPU consists of registers (described below), an arithmetic logic unit (ALU) capable of arithmetic and logical operations, a control unit, and other components which are connected with buses and wires.

Registers

The CPU registers are storage elements with fast access time. The time to access the contents of a value in the computer's memory can be over 1000 times slower than the time to access a CPU register. A register consists of a fixed number of bits, usually 32 or 64. In the MIPS architecture which we will be studying there are 32 bits in a register, and there are 32 general purpose registers. These registers can store intermediate results from arithmetic and logical computations, for which the operands must also be stored in registers. They can also be used to move information from one location in memory to another, and to store memory addresses (see the section on memory below).

In this book we will be diagramming registers as shown in Fig 1.1 Each register is designated by a unique number: 0..31 and stores 32 bits. To save space on the page only the first 8 of 32 general registers are shown. In the MIPS architecture register 0 will always contain all zeros. The other registers in Fig 1.1 contain randomly selected values with no particular intent or purpose. We will be using these diagrams to explain the various operations which the CPU is capable of performing, by showing the contents of registers before and after the operation is performed.

Program Counter

The 32 general purpose registers described above may be referred to as *programmable* registers, i.e. the values which they contain can be explicitly altered at the programmer's discretion. There are other registers in the CPU which are

0	00000000000000000000000000000000
1	11011000010101010100010101010100
2	00010101000111101010101010010101
3	11010000011100010010101010101010
4	00000000000000001111111111111111
5	11111111111111110000000000000000
6	00000000000000000000000000000001
7	00000000000000000000000000000000

Figure 1.1: Possible values for 8 of the 32 CPU registers in the MIPS architecture

necessary for the correct sequence of operations to take place. One such register is called the *program counter* register (PC). It contains the location (i.e. memory address) of the next instruction to be executed.

Datapath

The *datapath* is the name which we give to the components in the CPU which enable data to move between memory and the registers. The datapath also contains hardware which can execute fundamental arithmetic and logical operations. The following components are included in the datapath: the registers, the memory, the arithmetic/logic unit (ALU), the PC, the control unit, and the connections necessary for these components to work together.

1.1.2 Memory

Closely associated with the CPU is the *memory*, also known as *main memory* or *random access memory* (RAM). The memory stores data which is needed for CPU operations. For example, if a program is working with an array of numbers, those numbers would be stored in memory, where the CPU would have immediate access to them. The instructions making up a program, coded in binary, are also stored in memory. Each memory location has a unique *address*, much like the houses on a street have unique, sequential, addresses. When the CPU needs to access a particular memory value, it uses the address of that value to access it.

The bits (binary digits) of memory are normally viewed in groups of 8 bits. Each 8-bit group is called a *byte*. In the MIPS architecture which we study in this book, each byte of memory has a unique address; we say the memory is *byte addressable*. Recalling that registers are 32 bits, every 4 bytes constitute a *full word* of memory; we say that the *word size* for the MIPS architecture is 4 bytes, or 32 bits. This means that calculations and memory access are normally done with 32-bit values. Fig 1.2 is a diagram of the MIPS memory structure. This diagram shows only two words of memory, each with its own address. The 32 bits in each word are shown with a space between bytes to show the 4 bytes in a word (in an actual memory there is no such space). Since there are 4 bytes

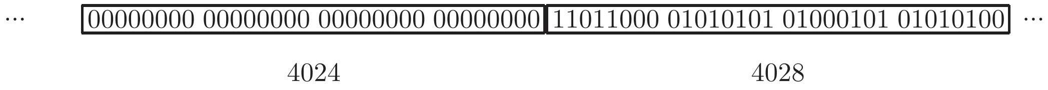


Figure 1.2: A portion of the MIPS memory, showing word addresses

in a word, and the memory is byte addressable, the word addresses increase by 4 (from 4024 to 4028 in this example).

1.1.3 Peripheral Devices

Peripheral Storage Devices

The memory described in the previous section is said to be *volatile*. This means that when power is switched off, all values in memory are lost. The term volatile is taken from chemistry in which a volatile liquid is one which evaporates readily and seemingly disappears. You may have experienced a loss of data when entering text into a word processor and there was a loss of electrical power to the computer. If the document was not saved, it would be lost because it is stored in memory.

This is but one reason for the need for *peripheral* devices such as disks or flash storage systems. These devices are capable of retaining data when power is shut off. Disks can be either magnetic or optical. The fixed disk (non-removable) in your computer's system unit is typically a magnetic disk. In addition to data files it stores all the system and application software which is needed by your computer. Optical disks are removable; examples of optical disk formats are CD, CD-ROM, DVD, etc. Data written to optical disks are typically not removed, whereas data on magnetic disks are commonly over-written or removed.

Another important device for permanent storage is *flash* memory, also known as a USB stick, a USB drive, thumb drive, etc. Here the word 'drive' is a misnomer because there are no moving parts in flash memory; it is a semiconductor material which is not volatile. Most mobile devices such as tablets and phones contain a substantial amount of (non-removable) flash memory.

Peripheral Devices for Input and Output

The terms *input* and *output* are to be understood from the memory's perspective. Output occurs when information is transferred from memory, out to an external device. Input occurs when information is transferred in to memory, from an external device.

A printer is an example of a device which is primarily an output device. Information is sent from the computer's memory to the printer, presumably in a form which is familiar to the user.

A monitor (or display) is also an output device. Information is constantly being sent from memory to the monitor. In most computers a section of memory is set aside specifically for this purpose, in which case the monitor is said to be *memory mapped*.

The keyboard and mouse are input devices. When the user types a key, moves the mouse, or presses a mouse button, signals are sent into the computer's memory, and the operating program is capable of responding appropriately.

Input and output can also take place through a wireless adapter and/or an ethernet port. This is useful for communication with other computers and the internet.

1.2 Machine Language

The CPU is capable of executing only the most basic and fundamental operations such as the addition of two values, the subtraction of two values, comparison of two values, etc. Furthermore these operations must be properly encoded in binary in order for the CPU to 'understand' their meaning. Such an operation is called an *instruction*. The instruction generally consists of a an operation code (for add, subtract, compare, etc) and locations of the operands (typically registers). An instruction which refers to a memory location would have to specify the address as a binary number. A sequence of binary-coded instructions in memory is called a *program*. The CPU executes the instructions of a program sequentially in the order in which they are stored in memory, unless it encounters an instruction which specifically orders it to alter that sequence. This language of binary coded instructions is usually called *machine language*; it is the only language which the CPU understands.

1.3 Assembly Language

Programming a computer in a binary machine language has, historically, been done; however it is obviously a tedious and error-prone way to proceed. For this reason we have developed a language called *assembly language* in which the operations are represented by plain text such as `add` and `sub`, and memory locations may be referenced with symbolic names, such as `salary` instead of a binary memory addresses. Since the CPU is not capable of executing assembly language programs, they must first be translated into machine language; this is done by software known as an *assembler*. We will be using an assembler developed at Missouri State University known as MARS [2]. In chapter 5 we will attempt to develop our own assembler.

1.4 Programming Languages

The assembly language instructions correspond, for the most part, with machine language operations. In assembly language it is not possible to specify many operations in a single statement, such as $a * 32 + b * 8$. For this feature (and many others) we rely on *high-level* or *programming* languages. Examples of

[2] MIPS Assembler and Runtime Simulator

programming languages are Java, C++, Visual Basic, and Python. Programs written in these languages must also be translated to machine language in order to be executed; this is done by a software translator known as a *compiler*. Compilers are not covered in this book, but a free introductory textbook on compiler design can be found at `cs.rowan.edu/~bergmann/books`.

1.5 Exercises

1. What is a computer program?

2. (a) What does CPU stand for?

   (b) What are the two primary purposes of the CPU?

3. In the CPU where are intermediate results of calculations stored?

4. In the MIPS archtitecture:

   (a) How many registers are there?

   (b) What is the size (in bits) of each register?

5. What is the purpose of the Program Counter (PC) register?

6. In the MIPS architecture:

   (a) How many bits are in a byte?

   (b) How many bytes are in a word?

   (c) How many bits are in a word?

7. (a) If the address of a particular byte in memory is 4321, what is the address of the next byte?

   (b) If the address of a particular word in memory is 70324, what is the address of the next word?

8. Which of the following are *volatile* storage?

   (a) Main memory

   (b) Flash memory

   (c) Magnetic disk

   (d) Optical disk

9. Many game consoles utilize a *joystick* to control the game. Is the joystick considered an input or output device?

10. Match the words with the correct descriptions:

1. Assembly Language	(a) May contain many operations in a single statement.
2. Machine Language	(b) Utilizes plain text to name instructions, such as 'add' and 'sub'.
3. Programming Language	(c) Primitive instructions are encoded in binary.

Chapter 2

Number Systems

In chapter 1 we introduced the notion of binary numbers. Here we generalize the notion and look at more convenient ways of describing large binary numbers. We also show how some operations on binary values can be simplified or facilitated.

2.1 Base Two - Binary

Whereas in a base ten (decimal) number each digit represents a power of 10, in base two each digit represents a power of 2, as shown in Fig 2.1 which depicts the base two representation of 19. $19 = 1 \cdot 16 + 0 \cdot 8 + 0 \cdot 4 + 1 \cdot 2 + 1 \cdot 1 = 10011_2$. If you add the positional values for which there is a 1 in the binary number, the sum is the value of the binary number. Other examples of binary numbers are shown in Fig 2.2.

2.1.1 Binary Arithmetic

Binary arithmetic is easy to learn; it is just like decimal arithmetic, but the only numerals permitted are 0 and 1. Simply remember that $1_2 + 1_2 = 10_2$ and that $1_2 + 1_2 + 1_2 = 11_2$. An example showing the addition of two 8-bit numbers is shown in Fig 6.41.

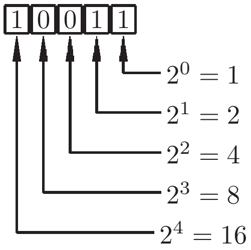


Figure 2.1: The binary representation of 19 (19 = 16 + 2 + 1)

$101_2 = 5$
$0101_2 = 5$
$10101_2 = 21$
$10000_2 = 16$
$1111_2 = 15$
$1000000000000_2 = 4096$
$1000000000001_2 = 4097$
$111111111111_2 = 4095$

Figure 2.2: Some examples of binary numbers (base 2)

```
  0 0 1 0 1 0 1 1   = 43
+ 0 0 0 0 1 1 1 0   = 14
 -----------------------
  0 0 1 1 1 0 0 1   = 57
```

(a)

```
        1 1 1
  0 0 1 0 1 0 1 1  = 43
+ 0 0 0 0 1 1 1 0  = 14
  ---------------------
  0 0 1 1 1 0 0 1  = 57
```

(b)

Figure 2.3: (a) Addition of 43 + 14 in binary using 8-bit values and (b) The same operation showing *carry* bits

Note that in any column where the result is 10_2, the 0 is written and the 1 is a *carry* into the next column. In any column where the result is 11_2, the (low-order) 1 is written and the (high-order) 1 is a *carry* into the next column.

Subtraction is similar to addition. When attempting to subtract $0_2 - 1_2$ we will need to *borrow* a 1 from the its (high-order) neighbor. If that neighbor is a 0, it will become 1 by borrowing from *its* neighbor, and so on. An example of a binary subtraction is shown in Fig 2.4 in which we subtract 70 - 13. Fig 2.5 shows the same operation, with the borrow digits at the top. In our example we are subtracting a small number from a larger number, ensuring that we get a positive result. If we were to subtract a large number from a smaller number, the result would be negative. This implies that we need a way to represent negative numbers, which is described in the section on Twos Complement Representation.

```
  0 1 0 0 0 1 1 0   = 70
- 0 0 0 0 1 1 0 1   = 13
 -----------------------
  0 0 1 1 1 0 0 1   = 57
```

Figure 2.4: Subtraction of 70 - 13 in binary using 8-bit values

$$\begin{array}{cccccccccccc}
 & & 0 & 1 & 1 & 10 & & 0 & 10 & & \\
 & 0 & \cancel{1} & \cancel{0} & \cancel{0} & \cancel{0} & 1 & \cancel{1} & \cancel{0} & = & 70 \\
- & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & = & 13 \\
\hline
 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & = & 57
\end{array}$$

Figure 2.5: Subtraction of 70 - 13 in binary using 8-bit values and showing a *borrow* from the neighboring column

2.1.2 Exercises

1. Show each of the following numbers as an 8-bit binary value: 15, 3, 0, 64, 63, 127

2. Show the following numbers in binary using only as many bits as are needed: 15, 3, 0, 128, 255, 256

3. Show the following binary numbers in decimal: 0101, 0111, 0111111, 01010101, 0100000

4. Show how to do the following operations in binary, using 8-bit words (show the carry bits for additions as shown in Fig 6.41(b) and the borrows for subtractions as shown in Fig 2.5): 12+3, 64+64, 64+63, 63+63, 12-4, 17-3, 128-127

5. Read parts (a) and (b) aloud so that they make sense.

   (a) There are 10 kinds of people in the world: those who know binary and those who do not.

   (b) There are 10 kinds of people in the world: those who know base 3, those who do not know base 3, and those who do not know what I'm talking about.

   (c) Make up a statement similar to the ones in parts (a) and (b) above, using a base in the range [4..9].

6. Show how to count from 0 to 31 using only the fingers on one hand (try not to offend anyone when you get to 4).

2.2 Base 8 - Octal

We wish to explore other number bases, primarily because they can be used as a *shorthand* for binary numbers. In this section we look at base 8, or octal, numbers.

In base 8 the numerals are 0..7, and each position represents a power of 8, as shown in Fig 2.6. We multiply the numeral in each position with the number represented by the position and add the results. $543 = 1 \cdot 512 + 0 \cdot 64 + 3 \cdot 8 + 7 \cdot 1 = 1037_8$.

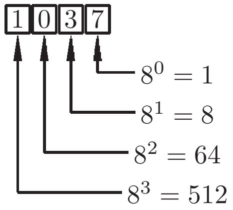


Figure 2.6: The octal representation of 543 (543 = 512 + 24 + 7)

$23_8 = 19$
$205_8 = 69$
$1000_8 = 512$
$3012_8 = 1546$
$1001_8 = 513$
$777_8 = 511$

Figure 2.7: Some examples of octal numbers (base 8)

Other examples of octal numbers are shown in Fig 2.7 in which numbers shown without subscripts are assumed to be base ten.

Why are we concerned with base 8? The best reason is that it provides us with a shorthand for binary numbers. Each octal digit represents 3 binary digits, as shown in Fig 2.8.

This means that we have a convenient way to represent long strings of bits - simply group them into groups of 3 bits, and represent each 3-bit group with an octal digit, as shown in Fig 2.9. Notice the last line in Fig 2.9 in which the number of bits provided is 16 (not a multiple of 3). This means we have one bit 'left over'. It must be the left-most bit, not the right-most bit, in order that the octal result represents the same number as the given binary value.

Conversely, we have a more common situation: we are given the octal representation of an n-bit field. If n is not a multiple of 3, the high order octal digit does not represent 3 bits. For example, if we are describing a 10-bit field in octal

octal	binary
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Figure 2.8: Each octal digit represents 3 bits

binary	groups of 3	octal
010101	010 101	25_8
111011l001	111 011 001	731_8
111111111111	111 111 111 111	7777_8
1011101011000111	1 011 101 011 000 111	135307_8

Figure 2.9: Each octal digit represents 3 bits

as 1234_8, then the 10-bit field is 1 010 011 100 = 1010011100. The high order (leftmost) octal digit represents just one bit. Another example, if describing a 5-bit field as 32_8, the 5-bit field must be 11 010 = 11010. In this case the high order octal digit represents just 2 bits. It will be important to remember this when dealing with the MIPS architecture in which the word size is 32 bits (not a multiple of 3), and many of the field widths in MIPS instructions are not multiples of 3.

2.2.1 Exercises

1. Show each of the following decimal numbers in base 8, using only as many octal digits as are necessary: 7, 9, 23, 100, 511, 512

2. Show each of the following octal numbers in decimal (base 10): 12_8, 32_8, 77_8, 777_8, 1000_8, 1010_8

3. Show each of the following binary values in base 8, using only as many octal digits as are necessary: 111_2, 110_2, 100000000_2, 100000001_2, 111111111_2, 10101011_2, 1111111_2
   Hint: There is no need to convert to decimal.

4. Show each of the following octal values in binary (base 2): 10_8, 37_8, 73_8, 234_8, 7150_8
   Hint: There is no need to convert to decimal.

5. An 8-bit field is storing the value 10101011_2. Show the 8-bit field in octal, using no more digits than are necessary.

6. What is the largest (decimal) value that can be represented with 4 octal digits?

2.3 Base 16 - Hexadecimal

In this section we look at base 16, or hexadecimal, numbers which are more useful and more commonly used than octal numbers as a shorthand for binary.

Recall that in base two there are only two numerals: 0 and 1. In base 8 there are 8 numerals: 0,1,2,3,4,5,6,7. In base 10 there are 10 numerals: 0,1,2,3,4,5,6,7,8,9. This implies that in base 16 we will need 16 numerals, though

hexadecimal	binary	decimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
a	1010	10
b	1011	11
c	1100	12
d	1101	13
e	1110	14
f	1111	15

Figure 2.10: Base 16: Each hexadecimal digit represents 4 bits. The 16 hex numerals have values ranging from 0 to 15.

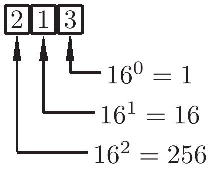


Figure 2.11: The hexadecimal (base 16) representation of 541 (541 = 512 + 16 + 3)

the character set provides only 10 numeric characters; we will have to supplement these numeric characters with 6 more characters to represent the decimal values 10, 11, 12, 13, 14, 15. We use the first 6 letters of the alphabet for this purpose, as shown in Fig 2.10.

Fig 2.11 shows how the decimal number 541 can be represented in hexadecimal. $541 = 2 \cdot 16^2 + 1 \cdot 16^1 + 3 = 213_{16}$.

Other examples of hexadecimal numbers are shown in Fig 2.12.

Note that each hexadecimal digit represents 4 bits, thus providing a somewhat more efficient representation for long bit strings. Fig 2.13 shows some examples of bit strings which can be represented much more easily in hexadecimal. Note in the last line of Fig 2.13 that the number of bits (19) is not a multiple of 4. We have 3 bits left over. As in the case with octal numbers, these left over bits must be the high order (leftmost) bits, in order for the number represented by the hex digits to be equal to the number represented by the given binary digits.

$a3_{16} = 10 \cdot 16 + 3 = 163$
$20d_{16} = 2 \cdot 16^2 + 0 \cdot 16 + 13 = 512 + 0 + 13 = 525$
$1000_{16} = 1 \cdot 16^3 + 0 \cdot 16^2 + 0 \cdot 16 + 0 = 4096$
$3012_{16} = 3 \cdot 16^3 + 0 \cdot 16^2 + 1 \cdot 16 + 2 = 12288 + 16 + 2 = 12306$
$1001_{16} = 1 \cdot 16^3 + 0 \cdot 16^2 + 0 \cdot 16 + 1 = 4097$
$fff_{16} = 15 \cdot 16^2 + 15 \cdot 16 + 15 = 4095$

Figure 2.12: Some examples of hexadecimal numbers (base 16)

binary	groups of 4	hexadecimal
01010111	0101 0111	57_{16}
1110101l0001	1110 1011 0001	$eb1_{16}$
1111111111111111	1111 1111 1111 1111	$ffff_{16}$
1011011101011001111	101 1011 1010 1100 1111	$5bacf_{16}$

Figure 2.13: Each hexadecimal digit represents 4 bits

We will often show hexadecimal values with a subscript of 'x' instead of 16 to indicate base 16. $321_x = 321_{16} = 801$. When using the MARS software, there are no subscripts, so base 16 constants will be designated with a prefix of '0x'. $0x321 = 321_{16} = 801$. The student may often see numbers written without any base indicated. These are usually intended to be base 10, but at times the base is evident from the context. For example, *the 6-bit opcode is 101001* is obviously binary, and *the result in register 2 is 4a56bf0f* is obviously base 16.

2.3.1 Hexadecimal Values in the MIPS Architecture

In the MIPS architecture which we are examining, the word size is 32 bits, but often words are broken into groups of bits called *fields*. When we describe the contents of a word (whether it be a register or a word in memory) we will generally use hexadecimal as a shorthand for the actual binary digits in a word.

For example, a word could contain the 32 bits
10010110111100011010001100011000
However it is easier to describe this word in hexadecimal than in binary, and it is easier for someone to grasp this word in hexadecimal than in binary. Can you look at the 32 bit word above and memorize it? We will now convert it to hexadecimal, first by grouping the 32 bits into fields of 4 bits:

1001 0110 1111 0001 1010 0011 0001 1000

It is then easy to convert each 4-bit field to a hexadecimal digit:

96f1a318

You will probably have more success memorizing the word in this form.

In later chapters we will be examining the fields in a MIPS instruction. In some instructions the bits are grouped in fields of size 6,5,5,5,5,6. Thus an instruction, in binary, could be:

101001 10001 00000 11111 01000 000111

To describe each field separately we would need two hex digits for each field, whether it represents 5 or 6 bits (for a 6-bit field, the high order hex digit represents two bits, and for a 5-bit field the high order digit represents just one bit):

29 11 00 1f 08 07

To describe the entire word in hexadecimal, we need to regroup the 32 bits into fields of 4 bits each:

1010.01 10.001 0.0000.1111.1 010.00 00.0111
1010 0110 0010 0000 1111 1010 0000 0111

This can now be shown in hexadecimal by substituting the correct hex digit for each group of 4 bits:

a620 fa07

2.3.2 Exercises

1. Show the following decimal values in hexadecimal (base 16): 13, 25, 170, 4095, 4096

2. Show the following hexadecimal values in decimal (base 10): 12_{16}, 20_{16}, $2e_{16}$, ff_{16}, 100_{16}, abc_{16}, fff_{16}, 1000_{16}

3. Convert the following hexadecimal values to binary: 92_{16}, $b9_{16}$, $2bf_{16}$, fff_{16}, 1000_{16}, $ffff_{16}$
Hint: Do not convert to decimal. Do not multiply.

4. Convert the following binary values to hexadecimal: 10110001_2, 11111101_2, 000100111110_2, 111111111111_2, 11111111111_2, 1111111111_2
111111111_2
Hint: Do not convert to decimal.

5. A 15-bit field is storing the value 101111101111100_2. Show this field in hexadecimal, using no more digits than necessary.

6. What is the largest value (in decimal) which can be represented with 4 hexadecimal digits?

7. A MIPS register contains the value 0xab3c401f. Show the 32 bits stored in that register.

8. A MIPS instruction with 6 fields (in hexadecimal) is 13 13 0f 1d 03 3a.

   (a) Show that instruction in binary (32 bits).

   (b) Show that instruction as a full word, using 8 hexadecimal digits.

2.4 Twos Complement Representation

We have seen that it is possible to represent non-negative whole numbers in binary (base 2), but we have not said anything about representing negative whole numbers. Recall that there are no minus signs ('-') in the computer's memory - only zeros and ones.

Number	binary
+7	0111
+6	0110
+5	0101
+4	0100
+3	0011
+2	0010
+1	0001
0	0000
-1	1111
-2	1110
-3	1101
-4	1100
-5	1011
-6	1010
-7	1001
-8	1000

Figure 2.14: Twos complement representation of positive and negative whole numbers, for a 4-bit word

Over the years various schemes have been used to represent negative whole numbers. At the present time almost every computer architecture uses a representation scheme called *twos complement* representation. With this scheme it becomes possible to add and subtract whole numbers without worrying about whether they are positive or negative; we always get the correct result.

To describe twos complement in an easy to understand way, we will work with a 4-bit word instead of a 32-bit word, but the same concepts will apply to a 32-bit word. Fig 2.14 shows the positive and negative values assigned to a 4-bit word. Note that:

- Zero is neither positive nor negative
- The largest positive value is 0111 = +7
- The smallest negative value is 1000 = -8
- There are more negative numbers than positive
- Every negative number begins with a 1 in the high-order (leftmost) bit
- All ones represents -1 (for any word size).

It should be clear why the positive numbers are as shown in Fig 2.14, but the negative numbers may require some explanation. Think back to the days when a car's odometer[1] was mechanically connected to the wheels, so that when the car backs up, the number on the odometer decreases. Now imagine that a brand new car, with an odometer reading of 000000 backs up 1 mile. What would you see on the odometer? Most likely 999999, which is equivalent to -1. For this reason -1 is represented in twos complement by all 1's.

Now try adding -1 to some other value, in binary (discard the bit carried out from the high order position):

```
  0110 = +6
+ 1111 = -1
-----------
  0101 = +5
```

[1] An odometer measures a car's mileage; don't confuse it with a speedometer

Students often ask, given a binary value, how do you know whether it is intended to be twos complement representation? For example, does the binary value 1100 represent 12 or -4? The answer is that given no other information about this binary value, it is impossible to know what it is supposed to represent. If you are told that it is two's complement representation, then you know it represents -4. But if you are told that it is *unsigned*, then you know it represents 12. You will see this concept again when we look at the instructions in the MIPS architecture. There are two `add` instructions, one of which is called `add unsigned`. The first `add` instruction assumes twos complement representation, and the `add unsigned` instruction assumes all values are non-negative.

How can we negate a value in twos complement representation? Here are three fairly easy algorithms for negating (or complementing) a number in binary:

- Subtract from 0. 0 - x = -x
- 1. Change all zeros to ones, and change all ones to zeros (this is called the *ones complement*).
  2. Add 1
- Scan the bits from right to left.
  1. As you scan, copy the low-order zeros
  2. Copy the first 1 digit
  3. Complement all remaining digits

As an example, we wish to negate (i.e. complement) the 4-bit binary value 0100 = +4. Using the first algorithm, we subtract from 0:

```
  0000 =  0
- 0100 = +4
-----------
  1100 = -4
```

When attempting to borrow from the high-order bit, we assume there is an extra 1 at the high-order end of the number, if needed.

If we negate that result, we should obtain the value we started with (- -x = x).

```
  0000 =  0
- 1100 = -4
-----------
  0100 = +4
```

We will now negate the same value, +4 = 0100, using the second algorithm. First, form the ones complement:
1011
Next, add 1:

```
  101l
+ 0001 = +1
-----------
  1100 = -4
```

Using the same algorithm to negate that result, we should obtain +4 = 0100. First, form the ones complement of 1100, to obtain 0011. Then add 1:

```
  0011
+ 0001 = +1
-----------
  0100 = +4
```

We will now negate the same value, +4 = 0100, using the third algorithm. Scanning from right to left we copy the low-order zeros:

```
0100
  00
```

We then copy the first 1 digit:

```
0100
 100
```

Finally we complement the remaining digits:

```
0100
1100 = -4
```

Using the same algorithm to negate that result, we should obtain +4 = 0100. Scanning from right to left we copy the low-order zeros:

```
1100
  00
```

We then copy the first 1 digit:

```
1100
 100
```

Finally we complement the remaining digits:

```
1100
0100 = +4
```

In summary, we have seen that negative as well as positive numbers can be represented using the twos complement representation. This scheme allows for easy implemenation of addition and subtraction, and it is used by virually every chip maker in the world.

2.4.1 Exercises

1. Show the following numbers using 8-bit twos complement representation: +6, -1, -2, -6, +22, -15, +127, -127, -128

2. (a) What are the largest and smallest numbers which can be represented using 8-bit twos complement representation?

   (b) What are the largest and smallest numbers which can be represented using an n-bit twos complement representation?

3. Show each of the following in twos complement representation, using only as many bits as are necessary: 15, 23, -15, -23, 2, 1, 0, -1, -2, 511, 512, -512

   Hint: A twos complement number is negative if and only if the high order bit is 1.

4. Show the decimal value of each of the following assuming (a) unsigned (b) twos complement representation:

   0111_2
   1111_2
   0101_2
   1010_2
   011_2
   11_2
   11111111_2
   11111110_2
   11010101_2
   10_2
   1_2
   01_2

5. Use any of the three algorithms given to negate each of the following, showing the solution in binary (binary numbers are twos complement representation).

   +75
   -76
   +15
   1111_2
   01000_2
   10001_2
   1111_2

6. A java int is a 32-bit whole number in twos complement representation. The class variable Integer.MAX_VALUE is the largest possible int, and the class variable Integer.MIN_VALUE is the smallest possible int.

(a) What is Integer.MAX_VALUE + Integer.MIN_VALUE?

(b) What is Integer.MIN_VALUE - Integer.MAX_VALUE?

(c) What is Integer.MAX_VALUE + Integer.MAX_VALUE?

(d) What is Integer.MIN_VALUE + Integer.MIN_VALUE?

Hint: See Fig 2.14. Use a java compiler to check your solutions if you are not sure.

2.5 Powers of Two

Computers are binary machines; consequently it is important that we be familiar with the powers of two. These numbers will also be cropping up in other areas of computer science, such as analysis of algorithms. In this section we will see how to find any power of 2 up to 2^{49}; moreover we will be able to do this mentally, without help from pencil, paper, nor digital device. We will also see some easy ways of doing arithmetic with powers of two.

n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1024

Figure 2.15: The powers of two

Fig 2.15 shows the powers of two up to $2^{10} = 1024$. Students should memorize these numbers; this will make it easy to do important calculations mentally.

Next we introduce some standard notation for large powers of two.[2]

$2^{10} = 1024 = 1K$

1K is the same as 1024; in this book 1K always represents 1024, and never 1000. 1K is simply a number and could be used to specify a quantity of memory (1K bytes or 1K words or 1K bits) as well as anything else (1K rabbits). Using this notation we have larger powers of two:

$2^{11} = 2 \cdot 2^{10} = 2K$
$2^{12} = 4 \cdot 2^{10} = 4K$
$2^{13} = 8 \cdot 2^{10} = 8K$
$2^{19} = 512 \cdot 2^{10} = 512K$

We have standard notation for larger powers of two.

$2^{20} = 2^{10} \cdot 2^{10} = 1024 \cdot 1K = 1M$

Some examples are:

$2^{21} = 2 \cdot 2^{20} = 2M$
$2^{22} = 4 \cdot 2^{20} = 4M$
$2^{23} = 8 \cdot 2^{20} = 8M$
$2^{29} = 512 \cdot 2^{20} = 512M$

[2] We observe the convention that capital letters are used for powers of 2, and lower case letters for powers of ten. Thus $1K = 2^{10} = 1024$ but $1k = 10^3 = 1000$.

n	2^n
3	8
13	8K
17	128K
21	2M
25	32M
32	4G
38	256G
40	1T
49	512T

Figure 2.16: Examples of large powers of 2

And yet larger powers of two:
$2^{30} = 2^{10} \cdot 2^{20} = 1024 \cdot 1M = 1G$
Some examples are:
$2^{31} = 2 \cdot 2^{30} = 2G$
$2^{32} = 4 \cdot 2^{30} = 4G$
$2^{33} = 8 \cdot 2^{30} = 8G$
$2^{39} = 512 \cdot 2^{30} = 512G$

Using the notation, $2^{40} = 2^{10} \cdot 2^{30} = 1024 \cdot 1G = 1T$
we can now easily describe any power of 2 up to 2^{49}

A few examples are shown in Fig 2.16. How are these values obtained? The letter - K,M,G, or T - is determined by the the first digit in the exponent of 2. 1=K 2=M 3=G 4=T. The number before the letter is two raised to the power of the second digit. $2^0 = 1$ $2^1 = 2$ $2^2 = 4$ $2^3 = 8$ (these are shown in Fig 2.15, and you've memorized them).

The next time you are at a party with friends you can impress people by announcing that you know all the powers of two, up to 2^{49}!

2.5.1 Arithmetic With Powers of Two

You may recall the following properties of exponents from your math classes:

- $x^y \cdot x^z = x^{y+z}$
- $x^y / x^z = x^{y-z}$
- $x^{y^z} = x^{y \cdot z}$

These properties will make arithmetic with powers of two much easier:

- $2^2 \cdot 2^3 = 2^{2+3} = 2^5 = 32$
- $2^9 / 2^3 = 2^{9-3} = 2^6 = 64$
- $2^{2^3} = 2^{2 \cdot 3} = 2^6 = 64$
- $1K \cdot 4K = 2^{10} \cdot 2^{12} = 2^{10+12} = 2^{22} = 4M$
- $64K \cdot 32K = 2^{16} \cdot 2^{15} = 2^{16+15} = 2^{31} = 2G$
- $8G/32K = 2^{33}/2^{15} = 2^{33-15} = 2^{18} = 256K$
- $2K^4 = (2^{11})^4 = 2^{11 \cdot 4} = 2^{44} = 16T$

We will be working extensively with powers of two in chapter 8 and will find it much easier using what we have learned here.

2.5.2 Exercises

1. Complete the following table, using the KMGT notation introduced in this section:

n	2^n
3	
7	
11	
15	
20	
24	
36	
48	
	16
	512
	8K
	128K
	1M
	64M
	4G
	32G
	32T
	512T

2. Evaluate each of the following:

   (a) 4K * 32K

   (b) 16M * 16M

   (c) 32M * 64G / 2T

   (d) 16T * 32G * 128M / 4T / 8T

   (e) $(32K)^3$

   Hint: See the identities in this section, and use powers of two.

3. Use the definitions provided in this section.

   (a) If there are 128K protozoa in a liter of pond water, and there are 4M liters of water in the pond, how many protozoa are in the pond?

   (b) If a ROM (read-only memory) consists of 4G bits, and there are 8 hits in a byte, how many bytes are in the ROM?

Chapter 3

Assembly Language for MIPS

In chapter 1 we described the fundamental components of a computer, including the CPU. We also described the notion of a program as a sequence of binary coded instructions stored in the computer's memory. We also introduced the notion of *assembly* language, in which we can use *mnemonics* and symbolic names rather than binary codes. Software known as an *assembler* is then needed to translate programs written in assembly language into equivalent machine language programs. In this chapter we deal exclusively with assembly language for a particular CPU known as MIPS.

3.1 Registers and Register Names

In the MIPS architecture there are 32 general registers, each of which stores a 32-bit full word. In order to perform an instruction, each operand must be in one of these registers (possibly the same register); the result of the instruction is placed in one of these registers (possibly the same register as an operand). Almost every instruction in the MIPS architecture will involve one or more registers.

In assembly language these registers can be referred to by their (numeric) addresses (in decimal), preceded by a dollar sign: $0 .. $31. Examples of register numbers are $5, $0, $30. Many of these registers have predefined purposes which *must* be observed, and others have predefined purposes which *should* be observed. The registers have names according to these predefined purposes, and these names are shown in Fig 3.1. At the present time we will be using only a few of these registers, and can safely ignore the others. We should note, however, that register $0, $zero, must *always* contain the value 0. It is a mistake to attempt to store any other value in register $0.

number	name	usage
$0	$zero	0
$1	$at	assembler temp
$2-$3	$v0-$v1	function return values
$4-$7	$a0-$a3	function arguments
$8-$15	$t0-$t7	temp
$16-$23	$s0-$s7	saved temp
$24-$25	$t8-$t9	temp
$26-$27	$k0-$k1	OS Kernel
$28	$gp	global pointer
$29	$sp	stack pointer
$30	$fp	frame pointer
$31	$ra	function return address

Figure 3.1: MIPS Register Names and Usages

```
[label:]    mnemonic   operand, operand, operand   [# comment]

[label:]    mnemonic   operand, operand            [# comment]

[label:]    mnemonic   operand                     [# comment]
```

Figure 3.2: Format of an Assembly Language Statement (optional fields are shown in brackets)

3.1.1 Exercises

1. Briefly describe the purpose of each of the following general registers

   (a) $a2

   (b) $v1

   (c) $ra

   (d) $29

2. How many bits are in a general register? How many bytes?

3. If a binary number is to select a general register by specifying its number, how many bits, in general, would be needed?

3.2 Assembly Language Statements

An assembly language statement takes the form shown in Fig 3.2. The square brackets indicate that the label and the comment are optional. The components of a statement are:

- The optional label will be discussed in the section on branches and jumps (transfer of control)
- The *mnemonic* is the operation to be performed; examples are `add` (for addition) and `sub` (for subtraction). The word 'mnemonic' means 'to remember'; mnemonics are easier to remember than their machine language equivalents (binary operation codes).
- There may be 0, 1, 2, or 3 operands separated by commas. At this point the operands will simply be general registers. Later in this chapter we will cover symbolic memory references as operands.
- The operands may be followed by an optional comment. Comments begin with a # and extend to the end of the line. There are no multi-line comments. Comments are strictly for the programmers use and are ignored by the assembler. Comments are especially important in assembly language, which is typically more difficult to read and understand than a high level language. Comments may also appear on a line with no statements, but must always begin with a # character.

An example of an assembly language statement with three operands is:

```
add  $t0, $t1, $zero       # this is a statement
```

In the example above the mnemonic is `add`, and the three operands are `$t0`, `$t1`, and `$zero`. That statement has a comment, but no label.

3.2.1 Exercises

1. A comment
   (a) must begin with what character?
   (b) has what use in a statement?
   (c) is always required: true or false.
   (d) is terminated with what character?
2. How many operands may an assembly language statement have?
3. What is a mnemonic?

3.3 Arithmetic Instructions

In this section we introduce some of the arithmetic[1] instructions available in the MIPS architecture. At this point we restrict our attention to instructions which are in the *Register* format, otherwise known as R format. Other formats will be covered in the sections on Immediate instructions and Jump instructions.

[1] pronounced a-rith-me'-tic

(a)

```
[label:]    add   $rd, $rs, $rt   [# comment]
```

(b)

$Reg[\$rd] \leftarrow Reg[\$rs] + Reg[\$rt]$

(c)

```
     add   $s0, $t3, $a0
```

(d)

```
     add   $t0, $t0, $t0      # $t0 = $t0 + $t0
```

Figure 3.3: Add Statement: (a) Format (b) Meaning (c) Example, which puts the sum of registers $t3 and $a0 into register $s0. (d) Example which doubles the value stored in register $t0
.

3.3.1 Add Instruction

The *add* instruction is a fundamental instruction in every CPU. It's purpose is to perform addition of 32-bit full word values (in general registers), placing the result also in a general register. A carry out of the high order bit is ignored (as in Java), so one may think of the operands as twos complement values (we'll have more to say about this in the section on overflow).

The add instruction is described in Fig 3.3 which shows:

- (a) The general format of an add statement. It must have three operands, all of which must be general registers. It may have an optional label and an optional comment.

- (b) The meaning of the add statement. The first operand is the destination register ($rd) and specifies which register is to receive the result of the addition. The second and third operands ($rs and $rt) are the registers which contain the values to be added. The notation `Reg[$reg]` means to select the general register with name `reg`.

- (c) An example of an add statement which adds the values stored in registers $t3 and $a0, and puts the sum into register $s0.

- (d) An example of an add statement which adds the value stored in register $t0 to itself, putting the sum back into register $t0, effectively doubling the value in register $t0. This example shows that the operand registers need not be different registers.

(a)

```
[label:]    sub   $rd, $rs, $rt   [# comment]
```

(b)

$Reg[\$rd] \leftarrow Reg[\$rs] - Reg[\$rt]$

(c)

```
     sub   $s0, $t3, $a0
```

(d)

```
     sub   $t0, $t0, $t0      # $t0 = $t0 - $t0
```

Figure 3.4: Subtract Statement: (a) Format (b) Meaning (c) Example, which puts the value of register $a0 subtracted from register $t3 into register $s0. (d) Example which puts the value 0 into register $t0
.

3.3.2 Subtract Instruction

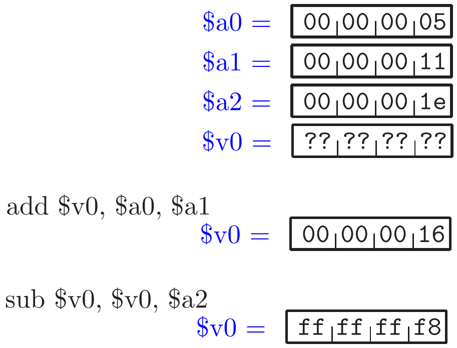
The `sub` (subtract) instruction is very similar to the `add` instruction. It must specify three general registers as operands; the first operand specified is the destination register. This instruction will subtract the value in the $rt register from the value in the $rs register and place the result in the $rd register as described in Fig 3.4.

How does this instruction work if the $rt register value is larger than the the $rs register? Think twos complement! The sub instruction works by negating the third operand and adding it to the second operand:
`a - b = a + -b`

3.3.3 Examples of Add and Subtract Instructions

In our first example we will perform the following simple calculation: `5 + 17 - 30`. This will require two assembly language statements: an `add` instruction and a `sub` instruction. We will assume that register $a0 contains the value 5, register $a1 contains 17, and register $a2 contains 30 (in the section on immediate instructions we will see how to put a constant value into a register). Our objective is to put the final result (which should be -8) into register $v0. Fig 3.5 is a diagram showing the relevant register contents (in hexadecimal) before and after each instruction. Note that the contents of register $vo is shown as question marks. This is our way of saying any of the following:

- We do not know what value is in register $v0.
- We do not care what value is in register $v0.



```
add $v0, $a0, $a1
sub $v0, $v0, $a2
```

Figure 3.5: Example to calculate 5 + 17 - 30, leaving the result in register $v0. The contents of the relevant registers are shown in hexadecimal before and after each instruction is executed.

- Register $v0 contains *garbage*.
- It is ok to store a new value in register $v0; the current value is not needed.

We do not assume that register $v0 contains 0. The `add` instruction will overwrite any value that is in register $v0, so its initial contents is irrelevant here.

In Fig 3.5 the `add` instruction will add the contents of register $a0 (5) to the contents of register $a1 (17) and store the result (22) in register $v0. The `sub` instruction will subtract the contents of register $a2 (30) from the value in register $v0 (22) and store the result (-8) in register $v0.

3.3.4 Set If Less Than

Here[2] we discuss an instruction which will not have any apparent utility; its usefulness will become evident in chapter 4. The instruction is *set if less than*, and the mnemonic is `slt`. It is an R format instruction which compares two registers, the $rs and $rt registers. It stores a 1 in the $rd register if the value of the $rs register is strictly less than the value of the $rt register. Otherwise it stores a 0 in the $rd register. This instruction is described
more formally in Fig 3.6.

Note that the comparison is an *arithmetic* comparison, meaning that the operands are assumed to be twos complement representation; negative numbers are smaller than positive numbers.[3] We will discuss this instruction further in connection with conditional branching in chapter 4.

As an example, we show a trace of the following sequence of instructions in Fig 3.7:[4]

[2] This section may be omitted without loss of continuity, but it should be covered before chapter 4

[3] There is another instruction, sltu, which performs an *unsigned* comparison.

[4] the `li` instruction loads a constant value into a register

(a)

```
[label:]    slt   $rd, $rs, $rt   [# comment]
```

(b)

$Reg[\$rd] \leftarrow\ 1\ if\ Reg[\$rs] < Reg[\$rt]$
$Reg[\$rd] \leftarrow\ 0\ if\ Reg[\$rs] \geq Reg[\$rt]$

(c)

```
     slt   $s0, $t3, $a0
```

Figure 3.6: Set If Less Than Statement: (a) Format (b) Meaning (c) Example, which stores 1 in register $s0 if register $t3 is less than register $a0, and which clears register $a0 if register $t3 is not less than register $a0

```
li     $t0, 5
li     $t1, -7
slt    $t2, $t0, $t1      # compare $t0 with $t1
slt    $t3, $t1, $t0      # compare $t1 with $t0
slt    $t4, $t0, $t0      # compare $0 with itself
```

3.3.5 Exercises

1. Show a diagram similar to Fig 3.5 for the following sequence of instructions:

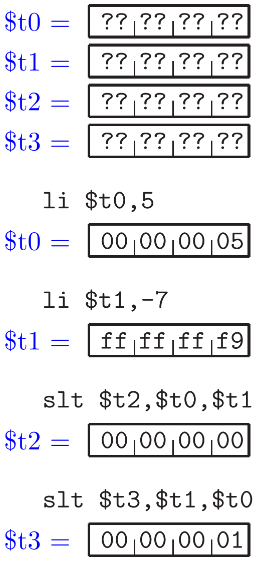
   ```
   add   $s3, $a0, $a1
   sub   $s2, $t0, $t0
   sub   $v0, $s3, $a0
   ```

   Assume that register $a0 initially contains +37 and that register $a1 contains -12. All other registers contain *garbage*.

2. Show a sequence of `add` and/or `sub` instructions which can be used to multiply the contents of register $t0 by 5. Use register $t1 for temporary storage, if helpful.

3. Show a sequence of `add` and/or `sub` instructions which can be used to multiply the contents of register $t0 by 8 (this can be done with only three instructions, and no other temporary register is needed).

4. Show a sequence of instructions which will put a 1 in register $v0 if the value in register $a0 is negative; otherwise it should clear register $v0.



```
$t0 = ??|??|??|??
$t1 = ??|??|??|??
$t2 = ??|??|??|??
$t3 = ??|??|??|??

      li $t0,5
$t0 = 00|00|00|05

      li $t1,-7
$t1 = ff|ff|ff|f9

      slt $t2,$t0,$t1
$t2 = 00|00|00|00

      slt $t3,$t1,$t0
$t3 = 00|00|00|01
```

Figure 3.7: Trace of a program which uses slt to compare register values.

5. Show a program trace, similar to Fig 3.7, for the following sequence of instructions:

```
li    $t0, -13
slt   $v0, $0, $t0
slt   $v1, $t0, $v0
```

3.4 Logical Instructions

Logical operations are the fundamental building blocks of computers. Complex arithmetic operations are all implemented using logical operations. In this section we discuss some of the common logical operations and their inclusion in the MIPS architecture. As in the previous section we will be looking at Register (R) format instructions only; other logical instruction formats will be covered later in this chapter.

3.4.1 Logical Operations

Logical operations are operations for which each operand must have one of the values: `true` or `false`. The result of a logical operation must also be `true` or `false`. These values and operations are often called *Boolean*[5].

The primary logical operations are *AND*, *OR*, *NOT*, and *EXCLUSIVE OR*. We define each of these separately:

[5] Named for the British mathematician George Boole

x	y	x AND y	x OR y	x XOR y	NOT x
		$x \wedge y$	$x \vee y$	$x \oplus y$	$\sim x$
false	false	false	false	false	true
false	true	false	true	true	true
true	false	false	true	true	false
true	true	true	true	false	false

Figure 3.8: Definition of 4 logical operations

- The AND operation results in `true` only if both operands are `true`. We use the notation $\wedge$ for the AND operator (many logic textbooks use the $\cdot$ symbol).

- The OR operation results in `false` only if both operands are `false`. This operation is sometimes called *INCLUSIVE* OR, to distinguish it from *EXCLUSIVE* OR. We use the notation $\vee$ for the OR operator (many logic textbooks use the + symbol).

- The EXCLUSIVE OR operation (XOR) results in `true` only if the two operands are `different`. It is called 'exclusive' because it is similar to INCLUSIVE OR but excludes the case where both operands are true from those having a true result. We use the notation $\oplus$ for the EXCLUSIVE OR operator.

- The NOT operation has only one operand (i.e. it is a *unary* operation). Its result is the complement of its operand. We use the notation $\sim$ for the NOT operator (many logic textbooks use x' or $\overline{x}$ to designate NOT x). For example, $\sim true$ is $false$ and $\sim false$ is $true$.

These definitions are summarized in Fig 3.8.[6] In common English usage, we often use 'or' to mean EXCLUSIVE OR: "Tonight I will go to the movies OR do my homework", implying that both will not occur. To express the meaning of INCLUSIVE OR, we sometimes use a slash as in: "For spring break I am going to Clearwater and/or Ft Lauderdale", meaning that both destinations are a possibility.

Having defined some fundamental logical operations, we can now form expressions using these operations. An example of a logical expression is: $true \wedge (false \vee true)$. This expression evaluates to `true` because the subexpression in parentheses evaluates to `true`, and we are left with $true \wedge true$, which is `true`. Examples of other expressions are shown in Fig 3.9 in which parentheses are used to specify the order in which the operations are applied.

A logical (or boolean) *identity* is an expression which may involve logical variables which is always true, regardless of the values of the variables. Some examples of identities are shown in Fig 3.10.

[6] The `not` instruction is actually a pseudo-op which uses a NOR operation, which will be covered in chapter 4

Logical expression	Value
$true \vee (true \wedge false)$	true
$false \vee (true \wedge false)$	false
$(false \vee true) \oplus (true \wedge true)$	false
$\sim ((false \vee true) \oplus (true \wedge true))$	true

Figure 3.9: Examples of logical expressions

$x \vee false = x$	$x \wedge true = x$	$x \vee true = true$
$x \wedge false = false$	$x \vee \sim x = true$	$x \wedge \sim x = false$
$x \oplus false = x$	$x \oplus true = \sim x$	$x \oplus x = false$
$x \oplus \sim x = true$	$(x \oplus y) \oplus z = x \oplus (y \oplus z)$	$y \oplus x \oplus y = x$
$\sim (x \wedge y) = (\sim x) \vee (\sim y)$	$\sim (x \vee y) = (\sim x) \wedge (\sim y)$	$x \oplus y = y \oplus x$

Figure 3.10: Some logical identities

The first two identities in the last row of the table are known as *deMorgan's Laws*. The identity $y \oplus x \oplus y = x$ is used extensively in private key cryptography.

Each of these identities can be proven with a simple truth table, in which we show that the identity holds for every possible value of the variables. As an example, Fig 3.11 shows a proof of deMorgan's first law.

In computer architecture, or logic design, the binary value 0 correspoonds to `false`, and 1 corresponds to `true`. In what follows we will make use of this.

3.4.2 MIPS Logical Instructions

The logical operations described in the preceding section are available as instructions in the MIPS architecture. In this section we will be covering only the Register (R) format instructions. As with the arithmetic instructions, each instruction has 3 operands the destination ($rs) register, and the two operand ($rs and $rt) registers. Fig 3.12 describes each of the logical instructions.

In these logic instructions (and in logic design in general) a 0 bit always represents `false`, and a 1 bit always represents `true`.

Note that the first three instructions in Fig 3.12 have three operands, but the *not* instruction has only two operands: the destination register and the source register. All three of these operations are *bitwise* logical operations, which means that each implies 32 logical operations, one for each bit in a register.

x	y	$\sim (x \wedge y)$	$(\sim x) \vee (\sim y)$
false	false	true	true
false	true	true	true
true	false	true	true
true	true	false	false

Figure 3.11: A truth table comprising a proof of deMorgan's first law

Mnemonic	Format	Meaning
and	`and $rd, $rs, $rt`	$Reg[\$rd] \leftarrow Reg[\$rs] \wedge Reg[\$rt]$
or	`or $rd, $rs, $rt`	$Reg[\$rd] \leftarrow Reg[\$rs] \vee Reg[\$rt]$
xor	`xor $rd, $rs, $rt`	$Reg[\$rd] \leftarrow Reg[\$rs] \oplus Reg[\$rt]$
not	`not $rd, $rs`	$Reg[\$rd] \leftarrow\sim Reg[\$rs]$

Figure 3.12: Four Logical instructions in the MIPS Architecture

$a0 = $\boxed{00|00|00|05}$

$a1 = $\boxed{00|00|00|0c}$

$v0 = $\boxed{??|??|??|??}$

`and $v0,$a0,$a1`

$v0 = $\boxed{00|00|00|04}$

`or $v0,$a0,$a1`

$v0 = $\boxed{00|00|00|0d}$

`xor $v0,$a0,$a1`

$v0 = $\boxed{00|00|00|09}$

`not $v0,$a0`

$v0 = $\boxed{ff|ff|ff|fa}$

Figure 3.13: Examples of Logical Instructions

Thus, for example, the `and` instruction will perform the logical AND operation on corresponding bits of the operand registers, comprising a total of 32 AND operations, with 32 results.

Some examples of logical instructions are shown in Fig 3.13, in which the contents of registers are shown before and after an instruction is executed.

To understand Fig 3.13, we must view the values in binary. The `and` instruction will perform the logical AND operation on all 32 bits of those two registers, putting the result into register $v0, as shown below (recalling that 0 represents false, and 1 represents true):

```
     0000 0000 0000 0000 0000 0000 0000 0101 = 0x00000005
AND  0000 0000 0000 0000 0000 0000 0000 1100 = 0x0000000c
     ----------------------------------------------------
     0000 0000 0000 0000 0000 0000 0000 0100 = 0x00000004
```

The logical OR, XOR, and NOT operations from Fig 3.13 are shown below:

```
     0000 0000 0000 0000 0000 0000 0000 0101 = 0x00000005
```

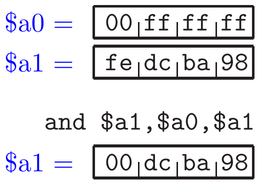
```
OR   0000 0000 0000 0000 0000 0000 0000 1100 = 0x0000000c
     ----------------------------------------------------
     0000 0000 0000 0000 0000 0000 0000 1101 = 0x0000000d


     0000 0000 0000 0000 0000 0000 0000 0101 = 0x00000005
XOR  0000 0000 0000 0000 0000 0000 0000 1100 = 0x0000000c
     ----------------------------------------------------
     0000 0000 0000 0000 0000 0000 0000 1001 = 0x00000009


NOT  0000 0000 0000 0000 0000 0000 0000 0101 = 0x00000005
     ----------------------------------------------------
     1111 1111 1111 1111 1111 1111 1111 1010 = 0xfffffffa
```

Masks

We conclude this section with some useful examples of logical instructions. The first such example is called a *mask*. A masks can be used to change or sense individual bits of a register, while leaving other bits unchanged (or unsensed).



```
and $a1,$a0,$a1
```

Figure 3.14: Example of a mask (in register $a0), to clear the high order byte of a register ($a1)

The first example of a mask will use the `and` instruction. Recall from Fig 3.10 that $x \wedge 0 = 0$ and that $x \wedge 1 = x$. Now suppose that we would like to force the high order byte of a register to all zeros, leaving the low order 3 bytes unchanged. We would use a mask of 0x00ffffff. This is shown in Fig 3.14 in which it is assumed that the appropriate mask, 0x00ffffff, has been loaded into register $a0 (we will see how this can be done in the next section), and the high order byte of register $a1 is to be set to all zeros.

We can also use a mask with an `or` instruction to *set* certain bits to 1. In this case we rely on two identities from Fig 3.10: $x \vee 0 = x$ and $x \vee 1 = 1$. Fig 3.15 shows an example where a negative number (-53) has been (somehow) loaded into the low order two bytes of register $a1. In order for this to be a valid 32-bit negative number, the high order two bytes must be set to all ones, leaving the low order two bytes unchanged. This can be done with a mask of 0xffff0000 (we assume this value has been loaded into register $a0).

We can also use a mask with an `xor` instruction to *complement* certain bits. In this case we rely on two identities from Fig 6.3: $x \oplus 0 = x$ and $x \oplus 1 = \sim x$.

Fig 3.16 shows an example in which we are interested in complementing alternate bits in a register. The value 0xff009876 has (somewhow) been loaded into register $a1. In binary this is
1111 1111 0000 0000 1001 1000 0111 0110.

$$\$a0 = \boxed{\texttt{ff|ff|00|00}}$$
$$\$a1 = \boxed{\texttt{00|00|ff|cb}}$$

```
or $a1,$a0,$a1
```

$$\$a1 = \boxed{\texttt{ff|ff|ff|cb}}$$

Figure 3.15: Example of a mask (in register $a0), to set the high order 2 bytes of a register ($a1)

$$\$a0 = \boxed{\texttt{55|55|55|55}}$$
$$\$a1 = \boxed{\texttt{ff|00|98|76}}$$

```
xor $a1,$a0,$a1
```

$$\$a1 = \boxed{\texttt{aa|55|cd|23}}$$

Figure 3.16: Example of a mask (in register $a0), to complement alternate bits in a register ($a1)

If we complement alternate bits, beginning with the low order bit, we should get
1010 1010 0101 0101 1100 1101 0010 0011 = 0xaa55cd23
This can be done with a single `xor` instruction, and is depicted in Fig 3.16. Since $5_{16} = 0101_2$ we can use a mask of 0x55555555 in register $a0.

3.4.3 Exercises

1. Find the value of each of the following expressions:
   (a) $(true \wedge false) \oplus (true \vee false)$
   (b) $(true \oplus false) \oplus (true \oplus false)$
   (c) $(true \oplus false) \oplus \sim (true \oplus false)$

2. Show a proof of deMorgan's second law.

3. Show a proof of the associativity of XOR: $(x \oplus y) \oplus z = x \oplus (y \oplus z)$

4. Show a proof of the identity: $y \oplus x \oplus y = x$
   (a) Using a truth table.
   (b) Using other identities from Fig 6.3.

5. The identity $k \oplus m \oplus k = m$ can be used to encrypt, and decrypt, a digital message, m, using a secret key, k. Let a 0 bit represent false and a 1 bit represent true.

   Bob wishes to send an 8-bit message to Alice so that she and only she will be able to read it. He will encrypt the message by applying a *bitwise* XOR operation with a secret 8-bit key.

(a) Show how Bob can encrypt the message m = 01101100 by applying the XOR operation with the secret key, k = 11010001. Show the ciphertext which he sends to Alice.

(b) Show how Alice can decrypt the ciphertext to obtain the original message, m, using the same secret key, k.

6. Show a diagram similar to Fig 3.13 For the following sequence of instructions. Assume that register $a0 contains `0x0011abcd` and that register $a1 contains `0xffab0123`.

```
and  $v0, $a0, $a0
and  $v0, $a0, $a1
or   $v0, $a0, $a1
xor  $v0, $a0, $a1
not  $v0, $a0
```

7. Show how the contents of $a0 can be copied into $v0 using:

   (a) An `and` instruction

   (b) An `or` instruction

   (c) An `xor` instruction

8. Show an instruction which will put the value -1 into register $v0, using only one (R format) instruction covered in this section.

9. In each of the following show the value of the mask in hexadecimal, and the instruction which will accomplish the given task. Assume the bits in a register are numbered, with the low order bit as bit 0, and the high order bit as bit 31.

   (a) Clear bits 0,1, and 31 of register $t0 using a mask in register $a1.

   (b) Set bits 6, 7, 9, 12 of register $a0 using a mask in register $a1.

   (c) Complement bits 0,1,2,3,28,29,30,31 of register $t0 using a mask in register $t1.

10. An iPod control system uses a 32-bit word in register $a0 to determine, and change, the state of the iPod according to the following table:

bit number	state
0	playing
1	paused
2	searching
3	stopped
4..31	[unused]

Only one of these bits should be set at any time.

For example, if bit 2 is set, the iPod is searching.

   (a) Registers $t0, $t1, $t2, $t3 are to be used as masks to control the 4 states. Show the values of these registers.

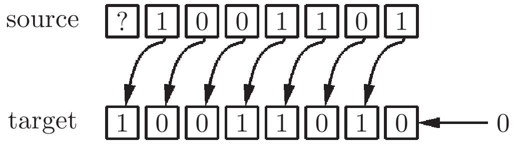


Figure 3.17: Diagram of a left shift instruction on an 8-bit register

(b) Using your response to part (a), show an instruction which will put a 0 into register $v0 if the iPod is not in the searching state, and some non-zero value into register $v0 if it is in the searching state.

(c) Show instruction(s) which will put the iPod into the stopped state, regardless of the state it is currently in.

11. Assume that a crypto-system has a secret key in register $s0.

    (a) Show an instruction which will encrypt the value in register $a0, putting the result into register $v0.

    (b) Show an instruction which will decrypt the value in register $v0, putting the result into register $v1.

    Hint: A previous exercise in this section described how this can be done.

3.5 Shift Instructions

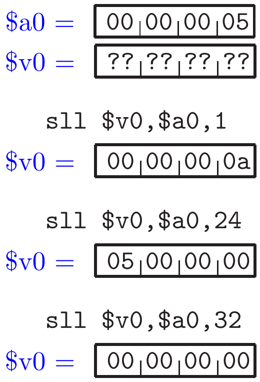
3.5.1 Logical Shift Instructions

In this section we cover bit *shifting* which is the process of moving each bit in a word to its right (or left) neighbor. We will ultimately distinguish between *logical* shift operations and *arithmetic* shift operations (to be discussed in the next section). With all shift operations there is a source register and a target register. The target register receives the result, and the source register is *unchanged*.

Fig 3.17 shows a diagram of a left shift operation on an 8-bit register. Each bit value is shifted to the left neighbor in the target register. Note that a zero is shifted into the low order bit of the target register, and the high order bit of the source register (shown as a `?` in the diagram) does not appear in the target register. In Fig 3.17 only 1 bit is shifted, but in the MIPS architecture several bits can be shifted with one instruction.

The mnemonic for a logical left shift instruction is `sll`. As with all shift instructions a *shiftamount* is used to specify the number of bits to be shifted. The format of the instruction is:

```
sll   $rd, $rt, shamt
```



```
$a0 = 00 00 00 05
$v0 = ?? ?? ?? ??

   sll $v0,$a0,1
$v0 = 00 00 00 0a

   sll $v0,$a0,24
$v0 = 05 00 00 00

   sll $v0,$a0,32
$v0 = 00 00 00 00
```

Figure 3.18: Example showing three left shift operations; shift 1 bit position, 24 positions, and 32 positions.

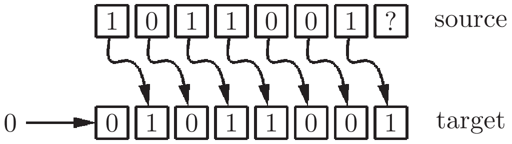


Figure 3.19: Diagram of a right shift instruction on an 8-bit register

$rd is the destination, or target, register. $rt is the source register, and `shamt` is the shiftamount, or number of bits to be shifted. Examples of shift instructions are shown in Fig 3.18.

A logical *right* shift goes in the other direction, as shown in Fig 3.19. A zero is shifted in at the high order bit, and the low order bit of the source register (shown with a `?`) is ignored. Examples of logical right shift instructions are shown in Fig 3.20, in which we shift by 1, 8, and 32 bits, respectively.

3.5.2 Arithmetic Shift Instructions

The logical shift instructions which we discussed above operate on *unsigned* quantities (the high order bit is not considered a sign bit, and the quantities are not treated as twos complement). When we are working with twos complement values, we wish to preserve the sign, particularly when shifting right. For this reason the architecture provides us with an *arithmetic* shift when shifting right: `sra`. Figure 3.21 depicts this operation on a register containing a non-negative value (the high order bit is 0).

Figure 3.22 depicts this operation on a register containing a negative value (the high order bit is 1). In each case the high order bit of the source register is copied to the high order bit of the target rigister, preserving the sign of the number.

```
$t3 = f3 00 00 05
$v0 = ?? ?? ?? ??

      srl $v0,$t3,1
$v0 = 79 80 00 02

      srl $v0,$t3,8
$v0 = 00 f3 00 00

      srl $v0,$t3,32
$v0 = 00 00 00 00
```

Figure 3.20: Example showing three right shift instructions; shift 1 bit position, 8 positionss, and 32 positions.

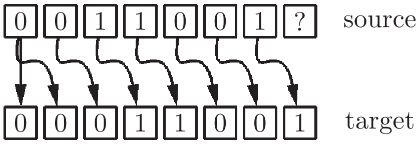


Figure 3.21: Diagram of an arithmetic right shift instruction on an 8-bit register. The source register is not negative.

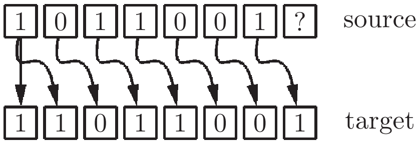


Figure 3.22: Diagram of an arithmetic right shift instruction on an 8-bit register. The source register is negative.

3.5.3 Common Applications of Shift Instrucrtions

Appending zeros at the end of a decimal number is the same as multiplying that number by a power of 10:
$123000 = 123 \cdot 10^3$

Consequently shifting a binary value to the left, shifts in zeros from the right, and is the same as multiplying the number by a power of 2:

```
1101 =     13
11010 =    2 * 13 = 26
110100 =   4 * 13 = 52
1101000 =  8 * 13 = 104
```

We frequently make use of this fact to obtain fast multiplication of a number by a power of 2. To multiply the number in a register by 2^n, shift it left n bit positions:

```
sll   $t3, $t3, 7      #  $t3 = $t3 * 128
```

By similar reasoning, shifting right performs a *division* by a power of two (recall we are working with whole numbers only):

```
00011010 =     26
00001101  =    26 / 2 = 13
00000110   =   26 / 4 = 6
00000011   =   26 / 8 = 3
```

The result of the shift provides the quotient only (no remainder). When shifting right we may wish to use an *arithmetic* shift to preserve the sign of the number[7]:

```
11110100 =     -12
11111010 =     -12 / 2 = -6
11111101 =     -12 / 4 = -3
```

3.5.4 Exercises

1. Show a diagram similar to Fig 3.18 for the following sequence of instructions. Assume that register \$t2 initially contains `80a3001f`$_{16}$ and that register \$t3 initially contains `ffffff0b`$_{16}$.

```
            sll   $v0, $t2, 4
            sll   $v0, $t2, 1
            sll   $t2, $t2, 3
            srl   $v0, $t2, 7
            sra   $v0, $t3, 3
```

[7] Caution: When shifting a negative number right, we get a valid division by a power of 2 only if no ones are shifted out!

2. In each case, show an instruction which will accomplish the given task:
   (a) Multiply the contents of register $a0 by 2, leaving the result in register $t0.
   (b) Multiply the contents of register $a3 by 128, leaving the result in register $a3.
   (c) Divide the (unsigned) contents of register $a3 by 1024, leaving the result in register $v0.

3. Show two examples of shift instructions which can be used to clear register $v0.

4. Show a diagram similar to Fig 3.18 for the following sequence of instructions. Assume that register $a0 initially contains `00000011`$_{16}$ = 17 and register $a1 initially contains `ffffffef`$_{16}$ = -17.

```
srl   $v0, $a0, 3       # divide 17/8, unsigned
sra   $v0, $a1, 3       # divide -17/8, signed (?)
```

5. Does an arithmetic right shift always yield a correct quotient for a division by a power of 2?

3.6 Immediate Instructions

All of the instructions that we have seen so far are classified as R (Register) format instructions. These instructions lack the ability to:

- Put a constant value into a register (other than zero)
- Add a constant value to a register
- Perform a logical operation with a constant value
- Transfer a number from RAM into a register
- Transfer a number from a register into RAM

These tasks can be accomplished by *Immediate* (I) format instructions.

3.6.1 Add, Load, Move, and PsuedoOps

The first immediate instruction we will study is *Add Immediate*. The mnemonic is `addi`, and its purpose is to add a register's contents to a constant, and store the result in a destination register. The constant is *part of the instruction itself.* The format and meaning of an `addi` instruction is shown in Fig 3.23. The permitted range of values for the constant, in hexadecimal is `0x0000..0xffff`. This corresponds to (twos complement) decimal values of `-32,768` through `32,767`.

(a)

```
[label:]    addi  $rd, $rs, constant   [# comment]
```

(b)

$Reg[\$rd] \leftarrow Reg[\$rs] + constant$

(c)

```
     addi  $s0, $t3, 17     # $t0 = $t3 + 17
```

(d)

```
     addi  $t0, $0, -8      # $t0 = -8
```

Figure 3.23: Add Immediate Instruction: (a) Format (b) Meaning (c) Example, which puts the sum of register $t3 and 17 into register $s0. (d) Example which puts the value -8 in register $t0.

This range applies not only to the addi instruction, but to all immediate format instructions.

Note that there is no immediate instruction for subtraction, but this is not a problem, since we can use a negative constant with an addi instruction. For example, to decrement the value in register $a0 by 3, we could use:

```
     addi   $a0, $a0, -3        # decrement reg $a0 by 3
```

Pseudo operations

Before introducing any more immediate format instructions, we digress briefly to introduce *pseudo operations*. Strictly speaking, these are not part of the MIPS instruction set architecture; however, they are permitted by the assembler, which translates them into actual instructions. A simple example of a pseudo-op is *load immediate*, for which the mnemonic is li. The purpose of li is simply to load a constant value into a register. The word *load* generally means to move data into a register, overwriting the data previously stored in the register. The format and meaning of an li pseudo-op is shown in Fig 3.24.

The assembler will translate an li instruction into an equivalent addi instruction, which makes use of the fact that register $0 always contains 0, as shown in part(e) of Fig 3.24.

Another useful pseudo-op is the *move* instruction, which will copy the contents of one register into another register. The format and meaning of the move instruction are shown in Fig 3.25. Notice that the target, or destination, for the move is the first operand, and the source, or origin, is the second operand. Fig 3.25 also shows that the assembler will translate a move pseudo-op to an actual MIPS instruction, such as an add instruction.

(a)

```
[label:]    li  $rd, constant   [# comment]
```

(b)

$Reg[\$rd] \leftarrow constant$

(c)

```
     li  $v0, 1023           # $v0 = 1023
```

(d)

```
     li  $t0, -1             # $t0 = -1
```

(e)

```
     addi  $t0, $0, -1       # $t0 =  0 + -1
```

Figure 3.24: Load Immediate Instruction: (a) Format (b) Meaning (c) Example, which puts 1023 into register \$v0 (d) Example which puts -1 in register \$t0 (e) addi instruction which is equivalent to example (d)

(a)

```
[label:]    move   $rd, $rs        [# comment]
```

(b)

$Reg[\$rd] \leftarrow Reg[\$rs]$

(c)

```
     move    $v0, $a0            # $v0 = $a0
```

(d)

```
     add  $v0, $a0, $0           # $v0 =  $a0 + 0
```

Figure 3.25: Move Instruction: (a) Format (b) Meaning (c) Example, which copies the value from register \$a0 into register \$v0 (d) An add instruction which is equivalent to example (c)

```
$v0 = | ??|??|??|?? |

   li $v0,2456
$v0 = | 00|00|09|98 |

   addi $v0,723
$v0 = | 00|00|0c|6b |

   addi $v0,-412
$v0 = | 00|00|0a|cf |

   sll $v0,$v0,6
$v0 = | 00|02|b3|c0 |
```

Figure 3.26: Trace of a program which calculates `(2456 + 723 - 412) * 64 = 177,088 = 2b3c0`$_{16}$

At this point we are able to write a somewhat meaningful program. Suppose we wish to do the following calculation, leaving the result in register $v0:
`(2456 + 723 - 412) * 64`

```
li     $v0, 2456         #   $v0 = 2456
addi   $v0, 723          #   $v0 = 2456 + 723
addi   $v0, -412         #   $v0 = 2456 + 723 - 412
sll    $v0, $v0, 6       #   $v0 = (2456 + 732 - 412) * 64
```

Note that these instructions are executed sequentially, beginning with the `li` instruction and ending with the `sll` instruction. Figure 3.26 shows a trace of the execution of this short program. At this point it would be advisable to run a small program such as this on a real computer. To do that you will need the software package known as MARS which is available free on the internet. Instructions on downloading, installing, and using MARS are in the Appendix.

3.6.2 Logical Immediate Instructions

The MIPS architecture also provides us with logical instructions in the Immediate format. The operations are AND, OR, and XOR; the formats are shown in Fig 3.27. In all cases these instructions perform a bit-wise operation using the $rs register and the constant as operands (the constant is limited to 16 bits, but is zero-extended to a full 32-bit word for purposes of executing the instruction).

Fig 3.28 shows a trace of the execution of the following instructions:

```
ori    $t0, $0, 23       # $t0 = $t0 OR 23
andi   $v0, $t0, 42      # $v0 = $t0 AND 42
xori   $v1, $t0, 42      # $v1 = $t0 XOR 42
```

Mnemonic	Format	Meaning
andi	`andi $rd, $rs, constant`	$Reg[\$rd] \leftarrow Reg[\$rs] \wedge constant$
ori	`ori $rd, $rs, constant`	$Reg[\$rd] \leftarrow Reg[\$rs] \vee constant$
xori	`xori $rd, $rs, constant`	$Reg[\$rd] \leftarrow Reg[\$rs] \oplus constant$

Figure 3.27: Three logical immediate instructions in the MIPS Architecture

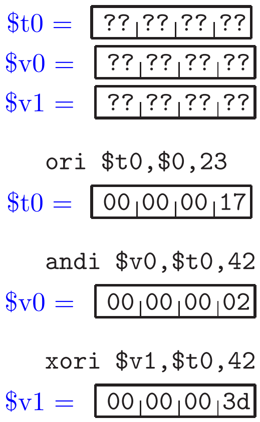


Figure 3.28: Examples of logical instructions, immediate format

In Fig 3.28, note that $23 = 00000017_{16}$ and $42 = 0000002a_{16}$.

As a further example, we show below a program to clear the low order 16 bits of register $a0, and complement the high order 16 bits of register $a0.

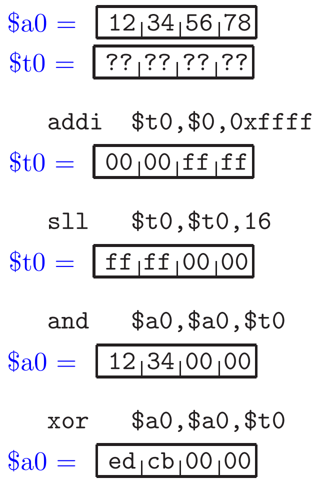
```
# Program to clear the low order 16 bits of register $a0
# and complement the high order 16 bits of register $a0.
   addi   $t0, $0, 0xffff       # mask:  $t0 = 0x0000 ffff
   sll    $t0, $t0, 16          # mask:  $t0 = 0xffff 0000
   and    $a0, $a0, $t0         # clear low order 16 bits
   xor    $a0, $a0, $t0         # complement high order 16 bits
```

Assuming that register $a0 initially contains 0x12345678, Fig 3.29 shows a trace of this program.

3.6.3 Load Upper Immediate

As we will see in chapter 4, there is a significant restriction on the size of an immediate operand: it must fit in a half word, i.e. 16 bits. For two's complement representation, the largest value would be $2^{15} - 1 = 32,767$, and the smallest value would be $-2^{15} = -32,768$. For operations which expect an unsigned immediate value, the largest possible value would be 0xffff = 65,535.

If we wish to load a constant that is not in this range, we must use the `lui` (load upper immediate) instruction. This instruction will load a 16-bit value



```
$a0 = 12|34|56|78
$t0 = ??|??|??|??

   addi  $t0,$0,0xffff
$t0 = 00|00|ff|ff

   sll   $t0,$t0,16
$t0 = ff|ff|00|00

   and   $a0,$a0,$t0
$a0 = 12|34|00|00

   xor   $a0,$a0,$t0
$a0 = ed|cb|00|00
```

Figure 3.29: Example to clear the low order 16 bits of register $a0, and complement the high order 16 bits of register $a0.

into the *high-order* half of a register. It will also clear the *low-order* half of the register, as shown in Fig 3.30.

Note that the `rs` field is not used by this instruction.

As an example, we show a sequence of instructions which will put a large constant, 65,539 = 0x00010003, into register $t3:

```
lui     $t3, 0x0001           # $t3 = 0x00010000
ori     $t3, $t3, 0x0003      # $t3 = 0x00010003
```

In this example we have used the `ori` (or immediate) instruction to get the desired result in register $t3. We will see extensive use of the `li` instruction in chapter 4.

3.6.4 Exercises

1. Show a diagram similar to Fig 3.20 for the following sequence of instructions:

```
li    $v0, 23
li    $a0, 17
addi  $v0, $a0, 9
move  $v1, $v0
srl   $v0, $a0, 3
```

2. (a) Show a sequence of instructions which will perform the calculation shown below, leaving the result in register $v0.
   `(127 + 43) / 8`

(a)

```
[label:]    lui   $rd, imm   [# comment]
```

(b)

$Reg[\$rt]_{16..31} \leftarrow\ imm$
$Reg[\$rt]_{0..15} \leftarrow\ 0$

(c)

```
     lui   $s0, 0x3001
```

Figure 3.30: Load Upper Immediate Statement: (a) Format. The `imm` field is 16 bits. (b) Meaning (c) Example, which loads the value 30010000_{16} into register $s0.

(b) Show a sequence of instructions which will perform the calculation shown below, leaving the result in register $v0. Use the t registers, if necessary, to store intermediate results.
`(127 + 59) * 4 + (700 - 659) * 16`

3. The valid range of values for a constant in an Immediate instruction is -32,768..32,767. This is the same as -32K..32K-1. How many bits are used for the constant in an immediate instruction?

4. Show a trace of the following program:

```
      li     $t0, 75
      li     $t1, 0x23
      addi   $v0, $t0, 23
      ori    $v0, $t1, 0xf070
      andi   $v1, $t1, 7
      xori   $v1, $t0, 7
      move   $a0, $v1
```

5. Show a program which will clear bits 0, 2, 3, and 31 of register $a0, leaving the other bits unchanged. Bit 0 is the low order bit of the register. You may use the t registers for temporary storage if necessary.

6. Show a program which will set bits 10, 11, 12, and 29 of register $a0, leaving the other bits unchanged. Bit 0 is the low order bit of the register. You may use the t registers for temporary storage if necessary.

7. Show a single MIPS statement which will complement bits 4, 5, and 7 of register $a0, leaving the other bits unchanged. Bit 0 is the low order bit of the register.

8. An iPod control system uses a 32-bit word in register $a0 to determine, and change, the state of the iPod according to the following table:

bit number	state
0	playing
1	paused
2	searching
3	stopped
4..31	[used for other purposes]

The iPod can be in one state only at any time. For example, if bit 2 is set, the iPod is searching.

   (a) Show the statement(s) which can be used to put the iPod into the `stopped` state. Do not change bits 4 through 31 of register $a0.

   (b) Show the statement(s) which will put 0 into register $v0 if the iPod is currently in the `searching` state and some non-zero value into register $v0 if the iPod is in some other state.

9. Show a sequence of instructions which will load the value 100,000 into register $t0.

10. Show the values of register $s1 and $s2, in hexadecimal, after the following instructions have executed:

```
    lui   $s1, 25
    lui   $s2, -5
```

3.7 Memory Reference Instructions

All the instructions we have dealt with thus far have involved operations on register contents, and possibly constants in the instructions. No attempt has been made to access data in main memory (RAM), yet this is typically where most data will reside during program execution.

There are two fundamental operations for memory reference:

- Transferring data from a full word in memory into a CPU register. This is called a *load* operation.

- Transferring data from a CPU register into a full word of memory. This is called a *store* operation.

The instructions which accomplish the load and store operations are, technically, Immediate format instructions. The reason for this will be more clear in the section on explicit memory addresses.

3.7.1 Symbolic Memory Addresses

Assembler directives

To include data in our programs we will need a few *assembler directives*. An assembler directive does not correspond to any machine language instruction, but provides the assembler with information which it will need to perform the translation to machine language.

The assembler directives which we need here are *.text* and *.data*:

- `.text` - This directive tells the assembler that all subsequent statements are to be translated as instructions. This is the default.
- `.data` - This directive tells the assembler that all subsequent statements are to be translated as data, and stored separately in the machine language program.

Note that directives begin with a decimal point, to distinguish them from instructions. There may be several .text and .data directives in a program, but in the resulting machine language program, all the instructions are stored in one contiguous area of memory and all the data are stored in a separate contiguous area of memory.

We will tend to put our instructions first, and data last, though many programmers use the opposite convention.

In the data area values can be included in the format:

```
[label:]    type    value(s)     # comment
```

The label is optional, but may be used by an instruction to access that particular data value. The `type` is a directive which specifies whether the data is a whole number, a floating point number, a string of characters, etc. To specify a whole number, the type should be `.word`; this corresponds to an `int` in java or C++ .[8] The values should be separated with commas.

As an example we examine a program which has 6 full word whole numbers as data values:

```
             .data
   x:        .word   7
   y:        .word   -1
   z:        .word   9, 10, 0xff     # three words
             .word   11              # 11 follows the 0xff in memory
```

Fig 3.31 shows these values in the computer's memory after being translated to machine language. Note that the memory address as well as the memory values are shown (this kind of diagram is often called a *memory dump*). The data memory area begins at address 0x1001000 because this is the location used by our assembler/simulator, MARS. The address and values are all shown in

[8] C++ does not specify the particular storage used for its data types, and may vary from one platform to another. Java is a portable language.

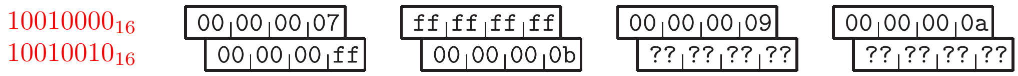


Figure 3.31: Six full words of data in contiguous memory locations, beginning at memory address 10010000_{16}

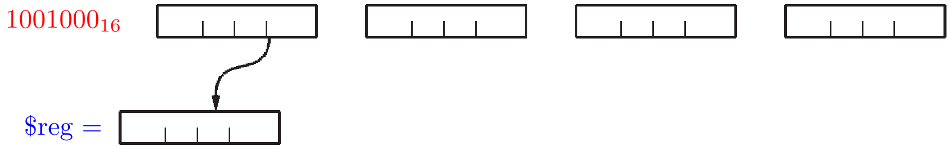


Figure 3.32: The load word (lw) instruction copies a full word from memory into a register

hexadecimal. Recall that each full word consists of four bytes, and the memory is byte adressable. Thus, the second data word is at address 0x10010004, and the fifth data word is at address 0x10010010. In Fig 3.31 the words at addresses 0x10010008 and 0x1001000c are shown as garbage because our program has not initialized them. [9]

Load and store

Now that we have data in our program we can utilize that data in our calculations. The relevant instructions are `lw` (load word), to load a full word from memory into a register, and `sw` (store word), to store a register's value into a full word of memory. The load word (lw) instruction and store word (sw) instructions are depicted in Figures 3.32 and 3.33, respectively. Note that the `lw` instruction will 'clobber' the existing value in the register being loaded, and the `sw` instruction will clobber the memory word into which data is being stored.[10]

The format and meaning of these two instructions is shown in Fig 3.34. Note that the memory location is specified by a label, which should match the label of a word in the .data section of the program.

[9] In actual practice, MARS will initialize all such data memory to zeros, but we prefer not to rely on this.

[10] We use the word *clobber* to imply that the existing value is overwritten and no longer available.

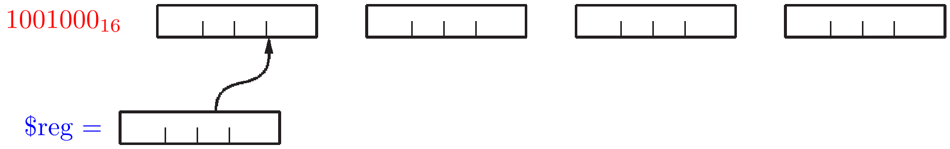


Figure 3.33: The store word (sw) instruction copies a full word from a register into memory

mnemonic	format	meaning	example
lw	lw $rt, label	$reg[\$rt] \leftarrow memory[label]$	lw $t3, x
sw	sw $rt, label	$memory[label] \leftarrow reg[\$rt]$	sw $v0, result

Figure 3.34: Format of the load word (lw) and store word (sw) instructions, using symbolic memory addresses (i.e. labels)

An example of a program which adds two values from memory, and stores the result into memory is shown below:

```
           .text
           lw    $t0, x
           lw    $t1, y
           add   $v0, $t1, $t0    # $v0 = x + y
           sw    $v0, result      # result = x + y

          .data
x:        .word    17
y:        .word    3
result:   .word    0              # store sum here
```

When this program executes, the value of x ($17 = 11_{16}$) is loaded from memory into register $t0, then the value of y (3) is loaded from memory into register $t1. The add instruction puts the sum of those values ($20 = 14_{16}$) into register $v0, which is then stored into the memory location labeled result. A trace of the execution of this program is shown in Fig 3.35 in which all data values are shown in hexadecimal.

Some computer architectures permit arithmetic and/or logical operations directly on memory locations. However, in the MIPS architecture all operands must be loaded into registers.

We conclude this section with an example to increment the value in a memory word, modulo 256. This means that the value will be reset to 0 when incrementing 255. We do this by using a mask to clear the high order 24 bits. The program is shown below:

```
                .text
                lw      $t0, x           # mod 256 counter
                addi    $t0, $t0, 1      # increment by 1
                andi    $t0, $t0, 0xff   # clear high order 24 bits
                sw      $t0, x           # store back to memory

                .data
x:              .word   33
```

In this program we load the value of x into register $t0, add 1, and then we use a mask of $\texttt{00000000ff}_{16}$ to clear the high order 24 bits, leaving the low order

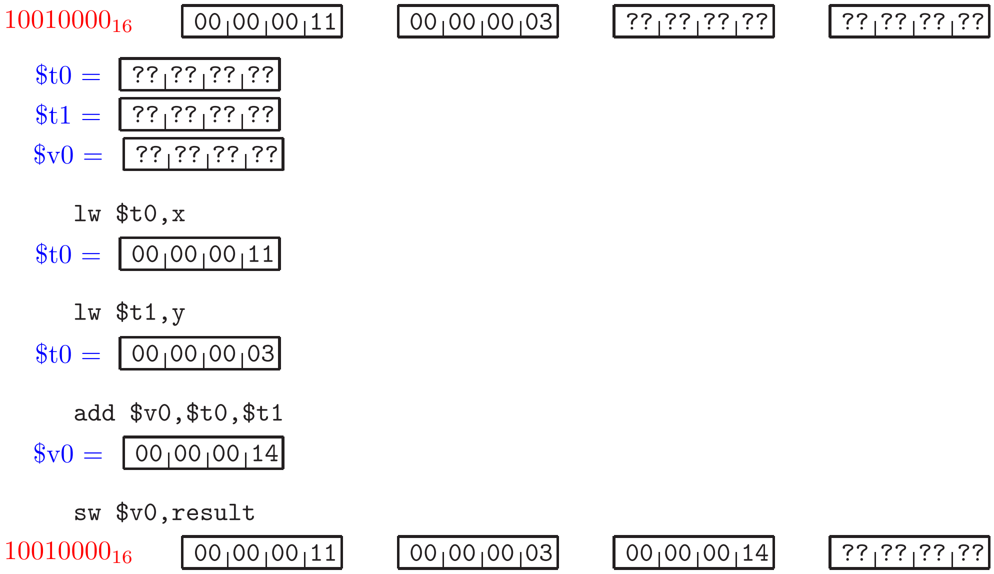


Figure 3.35: Trace of a program which stores the sum of two values in memory (x and y) into a third memory location (result)

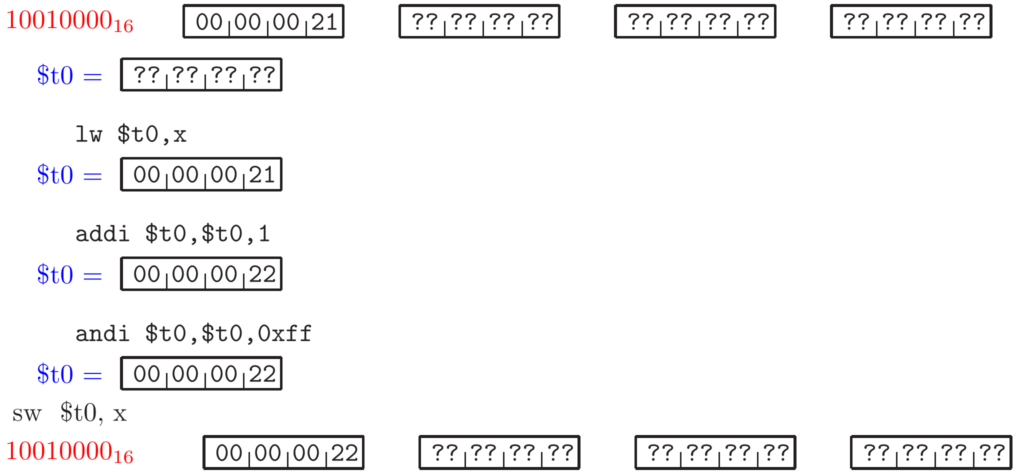
8 bits unchanged. We then store the result, 34, back into the memory location labeled `x`. Fig 3.36 shows a trace of the execution of this program.

If the value of `x` had been 255 instead of 33, the result would be 0, as shown in Fig 3.37. A modulo 256 counter resets to 0 when incrementing 255.

We can also add or subtract a fixed number of bytes to a symbolic address in assembly language. For example, if we have a label, `x`, on a data value, the expression `x+12` represents the address 12 bytes (or 3 words) larger; i.e. the address of data 12 bytes away from `x`. In the example below, we store the difference of the two words beginning at `start` into the location named `diff`.

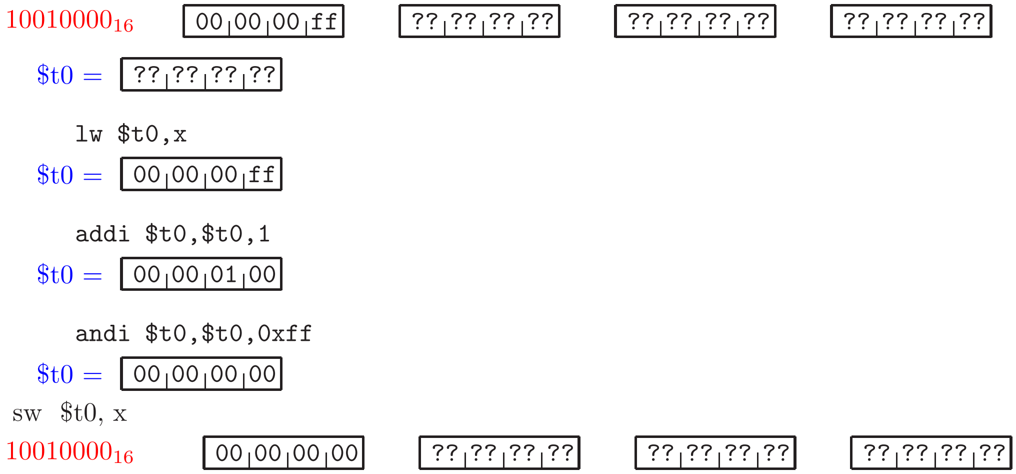
```
           .text
           lw   $t0, start
           lw   $t1, start+4        # next word after start
           sub  $t1, $t0, $t1       # difference
           sw   $t1, diff

           .data
start:     .word  23
           .word  17
diff:      .word  0                 # should be 6 when done
```



```
10010000₁₆   00|00|00|21   ??|??|??|??   ??|??|??|??   ??|??|??|??
  $t0 =  ??|??|??|??
    lw $t0,x
  $t0 =  00|00|00|21
    addi $t0,$t0,1
  $t0 =  00|00|00|22
    andi $t0,$t0,0xff
  $t0 =  00|00|00|22
 sw  $t0, x
10010000₁₆    00|00|00|22   ??|??|??|??   ??|??|??|??   ??|??|??|??
```

Figure 3.36: Trace of a program which increments a memory value, 33, modulo 256



```
10010000₁₆   00|00|00|ff   ??|??|??|??   ??|??|??|??   ??|??|??|??
  $t0 =  ??|??|??|??
    lw $t0,x
  $t0 =  00|00|00|ff
    addi $t0,$t0,1
  $t0 =  00|00|01|00
    andi $t0,$t0,0xff
  $t0 =  00|00|00|00
 sw  $t0, x
10010000₁₆    00|00|00|00   ??|??|??|??   ??|??|??|??   ??|??|??|??
```

Figure 3.37: Trace of a program which increments a memory value, 255, modulo 256

mnemonic	format	meaning	example
lw	lw $rt, imm($rs)	$reg[\$rt] \leftarrow memory[\$rs + imm]$	lw $t3, 20($a0)
sw	sw $rt, imm($rs)	$memory[\$rs + imm] \leftarrow reg[\$rt]$	sw $v0, -12($t3)

Figure 3.38: Format of the load word (lw) and store word (sw) instructions, using explicit memory addresses

3.7.2 Non-symbolic Load and Store

We have seen how the load and store instructions can use labels to refer to memory locations. However, it is also possible to refer to memory locations without using labels; instead we use explicit memory addresses, stored in registers. This form of memory reference, which is known as *explicit addressing*, will be essential when we start using arrays of data in memory. It is also important in understanding machine langauge.[11]

The lw and sw instructions are Immediate (I) format instructions. The format of lw and sw, using explicit memory addresses is shown in Fig 3.38, in which the immmediate field is shown as `imm`. The address of the desired memory word is calculated by adding the immediate value to the contents of the `$rs` register. The immediate value is often referred to as a *displacement* because it displaces the desired address by a fixed amount (positive or negative) from the address in the $rs register. In the first example, the memory address would be 20 bytes (5 words) beyond the address in regster $a0. The second example shows that the immediate field may be negative; the word of memory referred to is 12 bytes (3 words) *prior* to the address in register $t3.

As an example of a memory reference instruction with explicit addressing, we show below a program which will obtain the memory word whose address is in register $a0, negate it, and store it back into the succeeding word of memory. For the moment we ignore how the memory address of the desired word is placed into register $a0 initially.

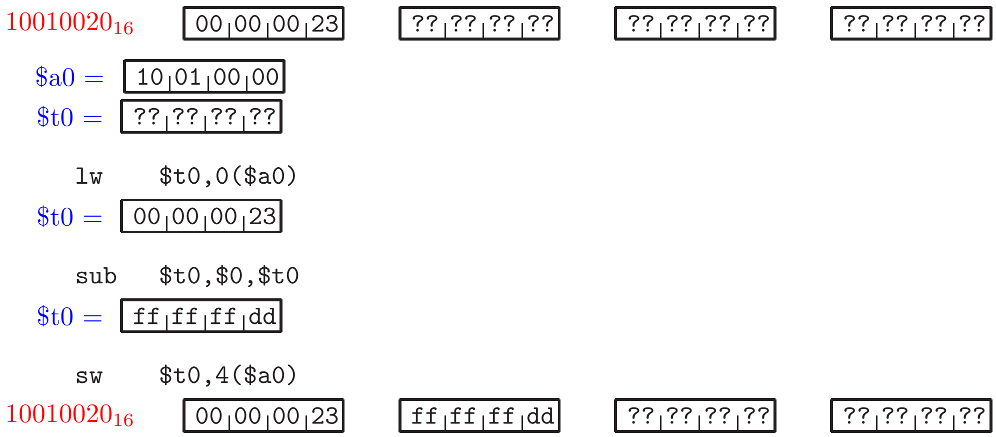
```
.text
lw    $t0, 0($a0)      # load word whose address is in $a0
sub   $t0, $0, $t0     # $t0 = 0 - $t0
sw    $t0, 4($a0)      # store into next word of memory
```

Note that the memory word adjacent to the one whose address is in register $a0 is obtained with a displacement of 4, because there are 4 bytes in a word. If we assume that memory location 10010020_{16} contains the value $00000023_{16} = 35$, and that register $a0 contains 10010020_{16}, then Fig 3.39 shows an execution trace of this program.

Load Address

In the previous example we assumed that register $a0 contained the desired memory address. We now show how a memory address can be placed in a

[11] A memory address is called a *pointer* in C++ or a *reference* in java.



```
10010020_16   00|00|00|23   ??|??|??|??   ??|??|??|??   ??|??|??|??
   $a0 =   10|01|00|00
   $t0 =   ??|??|??|??

      lw    $t0,0($a0)
   $t0 =   00|00|00|23

      sub   $t0,$0,$t0
   $t0 =   ff|ff|ff|dd

      sw    $t0,4($a0)
10010020_16   00|00|00|23   ff|ff|ff|dd   ??|??|??|??   ??|??|??|??
```

Figure 3.39: Trace of a program which stores the negation of the memory word whose address is in register $a0 in the next adjacent memory location

mnemonic	format	meaning	example
la	la $rd, label	$reg[\$rd] \leftarrow label$	la $t3, x

Figure 3.40: Format of the load address (la) instruction, with an example

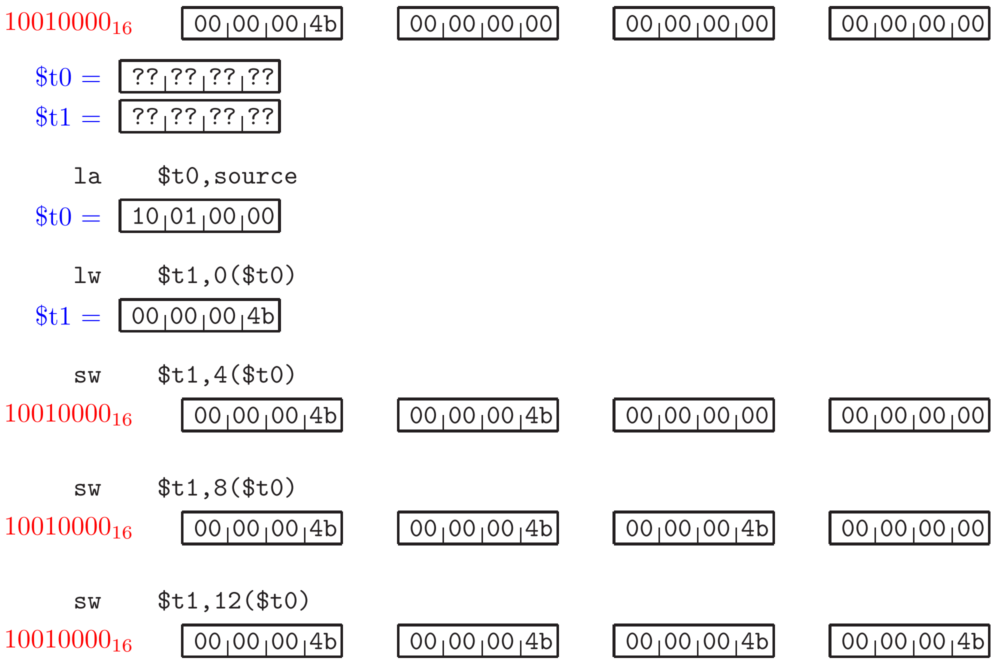
register. This is done with the *load address* (la) instruction. At this point we consider only the symbolic form of this instruction (it also has an explicit form, which is rarely used). The `la` instruction is actually a pseudo-operation, and in chapter 4 we will see how it is translated to machine language. The format of the `la` instruction is shown in Fig 3.40.

In this figure, compare the meaning of the load address instruction with the meaning of the load word instruction (in Fig 3.34). Instead of putting the *value* of the memory word into the destination register, it puts the *address* of the memory word into the destination register.

As an example we show below a program which will copy the value of the memory word labeled `source` to the next three adjacent words of memory (which have no labels).

```
           .text
           la     $t0, source     # address of source
           lw     $t1, 0($t0)     # value of source
           sw     $t1, 4($t0)     # store into source + 4
           sw     $t1, 8($t0)     # store into source + 8
           sw     $t1, 12($t0)    # store into source + 12

           .data
source:    .word  75
           .word  0,0,0
```



```
10010000_16   00|00|00|4b   00|00|00|00   00|00|00|00   00|00|00|00
   $t0 =   ??|??|??|??
   $t1 =   ??|??|??|??

     la     $t0,source
   $t0 =   10|01|00|00

     lw     $t1,0($t0)
   $t1 =   00|00|00|4b

     sw     $t1,4($t0)
10010000_16   00|00|00|4b   00|00|00|4b   00|00|00|00   00|00|00|00

     sw     $t1,8($t0)
10010000_16   00|00|00|4b   00|00|00|4b   00|00|00|4b   00|00|00|00

     sw     $t1,12($t0)
10010000_16   00|00|00|4b   00|00|00|4b   00|00|00|4b   00|00|00|4b
```

Figure 3.41: Trace of a program which copies the memory word labeled source to the next three adjacent memory words

An execution trace of this program is shown in Fig 3.41. Note that when execution finishes, all four words of memory store the same value ($4b_{16} = 75$).[12]

3.7.3 Exercises

1. Show an execution trace of the following program:

```
          .text
          lw      $t0, x
          lw      $t1, y
          sub     $v0, $t0, $t1      # v0 = x - y
          sw      $v0, x             # store result in x
          sw      $t0, y

          .data
x:        .word   0xff
y:        .word   127
```

2. Show an execution trace of the following program:

[12] This program can be done without using the la instruction, by storing into label+4, label+8, and label+12

```
         .text
         lw      $t0, x
         lw      $t1, y
         xor     $v0, $t0, $t1     # v0 = x xor y
         sw      $v0, x            # store result in x
         sw      $t0, y

         .data
x:       .word   127
y:       .word   0xff
```

3. Show the contents of register $v0 after the program shown below has executed:

```
          .text
          lw      $v0, x
          lw      $v1, x+8
          add     $v0, $v1, $v0

          .data
foo:      .word   100, 50
          .word   40
x:        .word   12, 13
          .word   4, 5
```

4. Given the data values shown below, write a MIPS program to store the sum of the values labeled `mary, jim, sue` into the location labeled `result`.

```
          .data
mary:     .word   17
jim:      .word   -99
sue:      .word   10
result:   .word   0
```

5. Given the data values shown below, write a MIPS program to store the sum of the 3 full-word values beginning at the location labeled `junk` into the location labeled `result` (the result should be 23).

```
          .data
junk:     .word   17, 23
          .word  -17, 5
result:   .word   0
```

6. Show an execution trace of the following program:

```
          .text
          la     $t0, first
          la     $t1, second
          la     $t2, first+8
          lw     $v0, 0($t0)
          lw     $v1, 0($t1)
          lw     $t3, 8($t0)
          sw     $t3, 8($t1)

          .data
first:    .word   7
second:   .word   8
          .word   9
          .word  10
```

7. What value will be in register $v0 after the program shown below has executed? What value will be stored in the full word of memory labeled `result`?

```
          .text
          la     $t0, x
          addi   $t0, $t0, 12
          lw     $t1, 0($t0)
          lw     $v0, 0($t0)
          la     $t2, result
          sw     $v0, 0($t2)

          .data
x:        .word  17
          .word  1, 2, 3, 4
result:   .word  0
```

8. Write a program which will add the 5 contiguous full words of memory beginning with the word whose address is in register $a0, and leave the sum in register $v0. (Assume register $a0 has been loaded with the appropriate address.)

9. Write a program which will compute the number of data values in the full words labeled `array`, leaving the result in register $v0. Use the data section shown below. If the data values are changed, your program should work without any changes to your code.
   Hint: Find the difference between the addresses represented by the start and end labels, then shift right to divide by 4.

```
          .data
array:    .word  12, 3, -9, 0, 0, 55, -44, 0 99
end:      .word  0
```

3.8 Transfer of Control

The assembly language programs that we have seen thus far execute sequentially - each instruction is executed, exactly once, in turn beginning with the first instruction of the program and ending with the last instruction.

There are principally two main reasons that we may wish execution to proceed in some other way:

- We may wish execution to take one of two possible paths, depending on the current state of the program (i.e. the values currently stored in registers). This is called a *selection structure* and is usually implemented with an `if` statement in high level programming languages.

- We may wish an instruction, or group of instructions, to be executed repeatedly, until some condition is satisfied. This is called an *iteration structure* or *loop* and is usually implemented with a `while`, `for`, or `do while` statement in high level programming languages.

In assembly language both of these control structures can be implemented with a *transfer of control*, which departs from the usual sequential execution of the statements in a program. There are two kinds of transfer of control:

- A *conditional transfer* is one in which the transfer may or not take place; it depends on the current state of the program. A conditional transfer is implemented in assembly language with a *branch* instruction.

- An *unconditional transfer* is one in which the transfer must take place; it does not depend on the current state of the program. An unconditional transfer is implemented in assembly language with a *jump* instruction.

In assembly language a selection structure is implemented with a *forward* transfer (either branch or jump) to a subsequent (higher memory address) instruction. An iteration structure, or loop, is implemented with a *backward* transfer (either branch or jump) to a previous (lower memory address) instruction. This is depicted in Fig 3.42. This is a somewhat naive view of selection structures, to be clarified later in this chapter.

3.8.1 Conditional Transfer of Control: Branch

A conditional transfer of control is one which may or not take place, depending on the program state (i.e. contents of registers) during execution. In MIPS assembly language a conditional transfer of control is implemented with a *branch* instruction. The branch instructions are summarized in Fig 3.43. Each of these instructions will compare two registers and branch to the instruction with the specified label if the specified condition is satisfied. In each case there is a target for the branch, which must be a valid label on some instruction in the program.

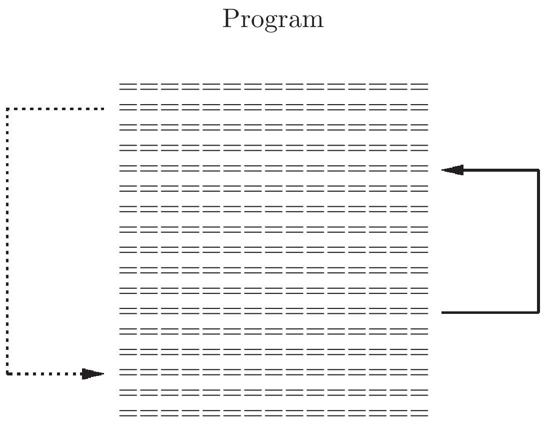


Figure 3.42: Diagram comparing a selection structure (left) with an iteration structure (right)

mnemonic	format	meaning	example
beq	beq $rs, $rt, label	$\rightarrow$ `label` $if\ reg[\$rs] = reg[\$rt]$	beq $t2, $a0, done
bne	bne $rs, $rt, label	$\rightarrow$ `label` $if\ reg[\$rs] \neq reg[\$rt]$	bne $t2, $0, lp
blt	blt $rs, $rt, label	$\rightarrow$ `label` $if\ reg[\$rs] < reg[\$rt]$	blt $t2, $t3, less
bgt	bgt $rs, $rt, label	$\rightarrow$ `label` $if\ reg[\$rs] > reg[\$rt]$	bgt $t0, $a0, big
ble	ble $rs, $rt, label	$\rightarrow$ `label` $if\ reg[\$rs] \leq reg[\$rt]$	ble $t2, $a0, start
bge	bge $rs, $rt, label	$\rightarrow$ `label` $if\ reg[\$rs] \geq reg[\$rt]$	bge $t0, $0, done

Figure 3.43: Format and meaning of the branch instructions, with examples

mnemonic	format	meaning	example
j	j label	→ `label`	j done

Figure 3.44: Format and meaning of the jump instruction, with example

To be accurate, only the `beq` and `bne` branch instructions are true MIPS instructions. The others are pseudo-ops; we will see how these are implemented in chapter 4.

Examples of programs with branch instructions will be given in the sections on Selection Structures and Iteration Structures.

3.8.2 Unconditional Transfer of Control: Jump

An unconditional transfer of control is one which takes place regardless of the current state of the program; no comparisons are done. In the MIPS architecture an unconditional transfer of control is implemented with a Jump (j) statement. The Jump statement is actually a J Format instruction (more on this in chapter 4). In assembly language it simply requires a label for the target, i.e. the statement to which control is transferred. The format and meaning of the Jump statement is shown in Fig 3.44.

3.8.3 Selection Structures

In this section we will describe how to implement selection structures in assembly language. These correspond to `if` statements in high level programming languages; there are two types of selection structures with which we are concerned:

- A *one-way selection* is a selection structure in which a block of statements is either executed or not executed. A one-way selection corresponds to an `if` statement without an `else` part.
- A *two-way selection* is a selection structure in which exactly one of two separate blocks of statements is executed. A two-way selection corresponds to an `if` statement with an `else` part.

One-way selection structures

A one-way selection structure corresponds to an `if` statement with no `else` part in a high level programming language. In assembly language it is implemented with a (conditional) branch to a subsequent statement, as depicted in Fig 3.45.

In that diagram, and subsequent diagrams, a dashed arrow will represent a (conditional) branch, and a solid arrow will represent an (unconditional) jump.

An example of a program with a one-way selection is shown below. It will move the value from either register \$a0 or register \$a1, which ever is smaller, into register \$v0, and also store it to the memory location labeled `result`.

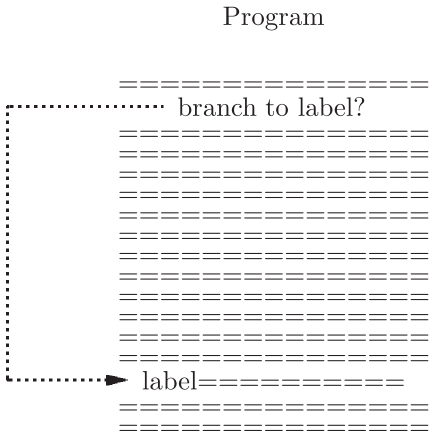


Figure 3.45: Diagram of a one-way selection structure

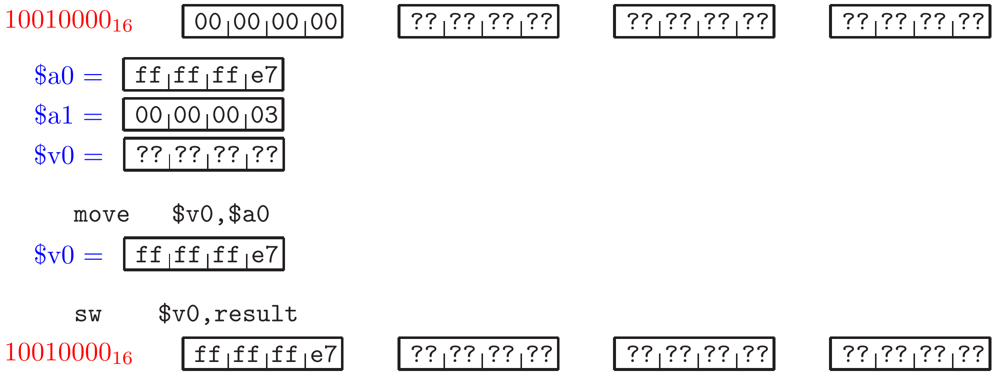
```
        .text
        move     $v0, $a0          # assume $a0 is smaller
        blt      $a0, $a1, done    # If $a0 is smaller, branch to done
                                   # If not, fall thru to next instr
        move     $v0, $a1          # $a1 is smaller (or equal)
done:
        sw       $v0, result
        .data
result: .word    0
```

The logic is as follows: We move the value in register $a0 into register $v0, whether it is the smaller or not. We then compare the registers $a0 and $a1; if register $a0 is smaller, we do not wish to change register $v0, so we branch around the second `move` instruction, to the `sw` instruction. In this program we could have put the label `done` on the same line with the `sw` instruction; normally we will put a label on a line by itself, for clarity, with the intention that it labels the following instruction. An execution trace of this program is shown in Fig 3.46. In that example, register $a0 initially contains -25, which is the smaller value, and register $a1 initially contains 3.

Two-way selection structures

A two-way selection structure corresponds to an `if` statement with an `else` part in a high level programming language. Exactly one of two separate blocks of statements is to be executed, depending on the program state. In assembly language it is implemented with a (conditional) branch instruction and an (unconditional) jump instruction, as depicted in Fig 3.47.

The conditional branch to the `else` part is executed when the `if` condition



```
10010000_16   00|00|00|00   ??|??|??|??   ??|??|??|??   ??|??|??|??
    $a0 =   ff|ff|ff|e7
    $a1 =   00|00|00|03
    $v0 =   ??|??|??|??

     move   $v0,$a0
    $v0 =   ff|ff|ff|e7

     sw    $v0,result
10010000_16   ff|ff|ff|e7   ??|??|??|??   ??|??|??|??   ??|??|??|??
```

Figure 3.46: Trace of a program which loads the smaller of registers $a0 and $a1 into register $v0, and also stores the smaller into the memory location labeled `result`

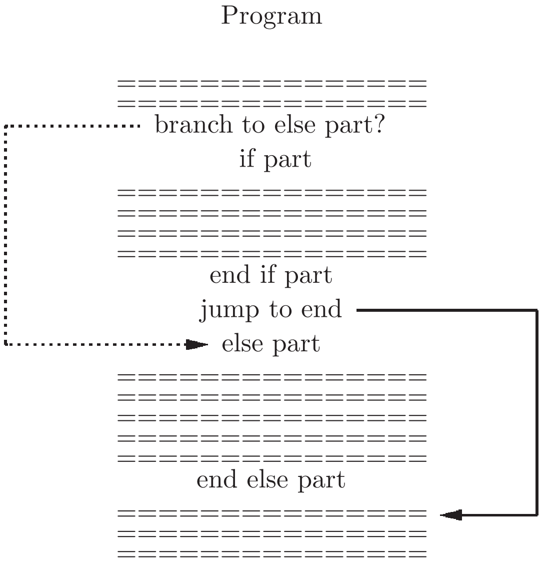


Figure 3.47: Diagram of a two-way selection structure

is *false*. This is the case where we wish to execute the else part and not the if part. In the case where we wish to execute the if part, the branch is not taken and execution falls through into the if part; at the conclusion of the if part there is an (unconditional) jump to avoid execution of the else part.

As an example, we wish to code the following java statement in assembly language:[13]

```
if ($a0 > 6)
    {  $v0 = 0;
       $a1++;
    }
else
   {  $v0 = $a0;
      $a1 = 0;
   }
```

The assembly language version of this statement uses register $t0 for temporary storage:

```
   li     $t0, 6
   ble    $a0, $t0, else    # branch if $a0 is NOT greater than 6
   li     $v0, 0            # $v0 = 0
   addi   $a1, $a1, 1       # $a1++
   j      done
else:
   move   $v0, $a0         # $v0 = $a0
   li     $a1, 0           # $a1 = 0
done:
```

We load the constant 6 into the temporary register $t0 because the conditional branch compares the contents of two registers. Note that the condition `$a0 > 6` is implemented with the logical complement: branch if less or equal (`ble`).

We show a few execution traces for this program. In both of these register $a1 initially contains the value 15. Fig 3.48 shows an execution trace in which register $a0 initially contains 6 (the conditioon is false because $a0 is not strictly greater than 6). Fig 3.49 shows an execution trace in in which register $a0 is initially 7 (the condition is true).

3.8.4 Iteration Structures - Loops

Without loops, computers are essentially powerless. Loops permit us to write short simple programs which can do extensive computations. High level programming languages such as Java and C++ use the while (pretest loop) and

[13] In java examples such as this, the use of registers correspond to java variables declared as int.

```
$a0 = 00|00|00|06
$a1 = 00|00|00|0f
$v0 = ??|??|??|??

      li    $t0,6
$t0 = 00|00|00|06

      ble   $a0,$t0,else

      move  $v0,$a0
$v0 = 00|00|00|06

      li    $a1,0
$a1 = 00|00|00|00
```

Figure 3.48: Trace of a program which implements a two-way selection; only the `else` part is executed.

```
$a0 = 00|00|00|07
$a1 = 00|00|00|0f
$v0 = ??|??|??|??

      li    $t0,6
$t0 = 00|00|00|06

      ble   $a0,$t0,else

      li    $v0,0
$v0 = 00|00|00|00

      addi $a1,$a1,1
$a1 = 00|00|00|10

      j     done
```

Figure 3.49: Trace of a program which implements a two-way selection; only the `if` part is executed.

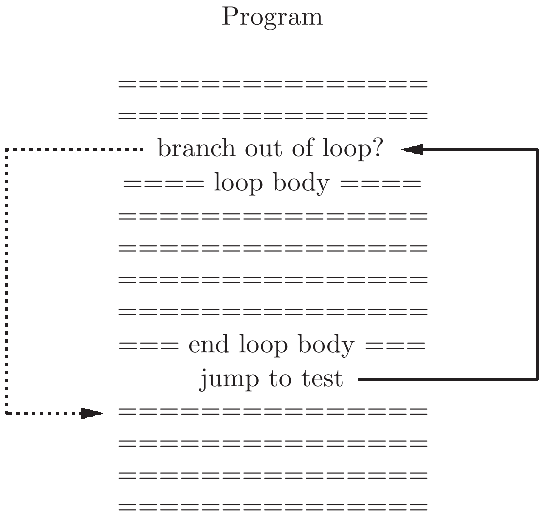


Figure 3.50: Diagram of a pretest loop iteration structure

`do while` (posttest loop) statements to implement loops.[14] In this section we will show how to implement each of these loops in assembly language. In both cases we refer to the sequence of statements to be repeated as the *body* of the loop.

Pretest loops - while statement

A pretest loop is one in which the test for termination occurs *before* the body of the loop is executed (even once). This allows for a loop in which the body is not executed at all (if the termination condition is initially false). A pretest loop is implemented in many high level languages with a `while` statement:

while (*condition*)
        Statement // loop body

In assembly language this can be implemented with a (conditional) branch out of the loop, and an (unconditional) jump back to the termination test as depicted in Fig 3.50.

As an example of a pretest loop, we have the following program to add the first 100 whole numbers, leaving the sum in register $v0:

```
          li     $t0, 100        # counter
          li     $v0, 0          # accumulator for sum
lp:
          beq    $t0, $0, done   # test for termination of loop
          add    $v0, $v0, $t0   # acc = acc + counter
```

[14] High level languages also use `for` statements to implement loops. Every `for` statement has an equivalent `while` statement.

```
          addi   $t0, $t0, -1    # counter--
          j      lp              # repeat body of loop
done:
```

In this program register \$t0 is used as a *counter*, initialized to 100, and decremented by 1 each time the loop repeats. Register \$v0 is used as an *accumulator*, increasing by the value of the counter each time the loop repeats; regiser \$v0 accumulates the sum of the first 100 whole numbers. Note that this program is careful to initialize the accumulator, \$v0, to 0; it is not advisable to assume that any register contains any particulare initial value, including 0.

Fig 3.51 shows a partial trace of this program, starting with the value 100 = 64_{16} in register \$t0, and ending with 0 in that register. The final sum, 5050 = $11ba_{16}$ is left in register \$v0.[15]

Posttest loops - do while

A posttest loop is one in which the test for termination occurs at the *end* of the body of the loop. This implies that the loop body must be executed at least once, even if the termination condition is initially false. The posttest loop is implemented in many high level language with the `do while` statement:

`do` *Statement* `while` (*Condition*);

Below we show the same example to sum the first 100 whole numbers, this time using a posttest loop:

```
          li     $t0, 100        # counter
          li     $v0, 0          # accumulator for sum
lp:
          add    $v0, $v0, $t0   # acc = acc + counter
          addi   $t0, $t0, -1    # counter--
          bne    $t0, $0, lp     # repeat body of loop?
```

Note that the body of the loop is the same as for pretest loops, but no (unconditional) jump is needed in the loop control for this program. When the `bne` branch fails, control falls through to the next instruction, terminating the loop.

Pretest versus posttest loops

How does one know whether to use a pretest or a posttest loop? They are very similar, and in most cases either will suffice; choose the one which is more natural for you.

[15] This sum can be verified using the formula

$$\sum_{i=1}^{n} i = n \cdot (n+1)/2$$

```
$t0 = |??|??|??|??|
$v0 = |??|??|??|??|

   li $t0,100
$t0 = |00|00|00|64|

   li $v0,0
$v0 = |00|00|00|00|

   beq $t0,$0,done

   add $v0,$v0,$t0
$v0 = |00|00|00|64|

   addi $t0,$t0,-1
$t0 = |00|00|00|63|

   j lp

   beq $t0,$0,done

   add $v0,$v0,$t0
$v0 = |00|00|00|c7|

   addi $t0,$t0,-1
$t0 = |00|00|00|62|

   j lp

   ...

   add $v0,$v0,$t0
$v0 = |00|00|11|ba|

   addi $t0,$t0,-1
$t0 = |00|00|00|00|

   j lp

   beq $t0,$0,done
```

Figure 3.51: Partial trace of a program which uses a pretest loop to sum the first 100 whole numbers, leaving the result in $v0

A posttest loop requires fewer statements for loop control, however if there are cases where the loop body should not be executed at all, you *must* use a pretest loop. We generally favor pretest loops, and most of our examples which require loops will use pretest loops. The most frequent error made by programmers with regard to loops is an *off by one* error - a loop which misses the correct number of iterations by just one. Be sure to check this by hand simulating a simple example.

3.8.5 Exercises

1. Show an execution trace of the following program:

```
           li   $t0, 3
           li   $v0, 1
           bgt  $t0, $0, positive
           li   $v0, -1
positive:
```

2. Show an execution trace of the following program:

```
           li   $t0, 3
           blt  $t0, $0, negative
           li   $v0, 1
           j    done
negative:
           li   $v0, -1
done:
```

3. Write a program which will implement the following in assembly language:

```
  if ($a0 == 17)
     {  $v0 = 0;
        $v1 = 3;
     }
```

4. What value will be left in register $v0 after the program shown below has executed?

```
         li     $t0, -7
         li     $v0, 6
         blt    $t0, $v0, skip
         li     $v0, 0
skip:
```

5. Write a program which will put the larger value (first or second) into register $v0. Use the following data section. Your program should work without change if the values in the data section are changed.

```
        .data
first:  .word   23
second: .word  -23
```

6. Write a program which will compare the values in registers $a0, $a1, and $a2 and put the smallest of the three values into register $v0.

7. Write an assembly language program which is equivalent to the program shown here:

```
if ($a0 > $a1)
   {  $v0 = 17;
      $v1 = 0;
   }
else
   $v0 = $v1;
```

8. Show an execution trace of the following program:

```
         .text
         lw     $t0, x
         lw     $t1, y
         ble    $t0, $t1, skip
         li     $v0, 0
         li     $v1, 1
         j      done
skip:
         li     $v0, 1
         li     $v1, 0
done:

         .data
x:       .word  15
y:       .word  -17
```

9. Write a program which will determine the sign of a full word in memory labeled `x`. If x is negative, $v0 should be -1; if x is positive, $v0 should be +1; if x is zero, $v0 should be 0.

10. What value will be left in register $v0 after the program shown below has executed?

```
         .text
         li      $v0, 3
         lw      $t0, count
lp:
```

```
         ble     $t0, $0, done
         addi    $t0, $t0, -1
         add     $v0, $v0, $v0
         j       lp
done:

         .data
count:   .word  4
```

11. Show an execution trace for the program in the preceding exercise.

12. What value will be left in register $v0 after the program shown below has executed?

```
         .text
         li      $v0, 3
         lw      $t0, count
lp:
         addi    $t0, $t0, -1
         add     $v0, $v0, $v0
         ble     $t0, $0, lp

         .data
count:   .word  4
```

13. Show an execution trace for the program in the preceding exercise.

14. Write a program which will find the sum of the whole numbers from 20 through 40, inclusive, leaving the result in register $v0. Use a loop.

15. Write a program which will multiply data values x and y, leaving the product in register $v0. Assume x is not negative.
Hint: Use repeated addition in a loop. Use x as a counter, and add the value of y into $v0, used as an accumulator.

16. Write a program which will calculate 2^x, where x is a non-negative data value. It should leave the result in register $v0.
Hint: Use x as a counter, and starting with 1 in $v0, shift left in a loop.

3.9 Memory Arrays

High level programming languages have an important feature known as the *array.* An array of 100 floats, in java, can be declared and created as shown below:

```
float [] numbers = new float [100];
```

Each element of the array can be accessed directly, with an int *subscript* in square brackets;

```
numbers[7] = 14.5;
numbers[3] = numbers[7] + 1.0;     // 15.5
```

An array is simply a sequence of *contiguous (i.e. consecutive)* locations in memory. In assembly language we cannot enforce a *type* on the values of an array. I.e. the assembler will permit an array to consist of different data types, though this is not normally recommended.

We will normally assign a name to an array by naming the first element. Then other values in the array can be accessed by loading the address of the first element, and adding a displacement to arrive at the desired array element.

The following program will sum the five values in the array named `grades`, leaving the result in register $v0.

```
             .text
             li    $t0, 5          # counter = 5
             la    $t1, grades     # pointer to array
             li    $v0, 0          # accumulator = 0
lp:
             beq   $t0, $0, done   # exit loop?
             lw    $t2, 0($t1)     # grades[i]
             add   $v0, $v0, $t2   # acc = acc + grades[i]
             addi  $t1, $t1, 4     # pointer = pointer + 4
             addi  $t0, $t0, -1    # counter--
             j     lp              # repeat loop
done:

             .data
grades:      .word 25, 63, -45, 0, 12
```

In this example we use register $t0 as a counter. It is initialized to 5, because we know the length of the array is 5 words. Each time the loop repeats we decrement the counter by 1. Register $t1 is initialized with the address of the first word in the array; we call this a *pointer*[16] to the array. Each time the loop repeats we increment the pointer by 4 (because there are 4 bytes in a word) to obtain the address of the next word in the array. Register $t2 is used to load a word from the array, so that it can be added to the accumulator, register $v0. A partial execution trace of this program is shown in Fig 3.52.

3.9.1 Exercises

1. The following program should find the sum of the positive values in an array of whole numbers (the length is 5). Show an execution trace of this program.

[16] A pointer in C/C++ is essentially the same as a *reference* in java - a memory address. The difference is that you can do arithmetic with a pointer but not with a reference.

```
10010000₁₆  [00|00|00|19]  [00|00|00|3f]  [ff|ff|ff|d3]  [00|00|00|00]
10010010₁₆    [00|00|00|0c]  [??|??|??|??]  [??|??|??|??]  [??|??|??|??]
  $t0 = [??|??|??|??]
  $t1 = [??|??|??|??]
  $t2 = [??|??|??|??]
  $v0 = [??|??|??|??]

      li    $t0,5
  $t0 = [00|00|00|05]

      la    $t1,grades
  $t1 = [10|01|00|00]

      li    $v0,0
  $v0 = [00|00|00|00]

      beq   $t0,$0,done

      lw    $t2,0($t1)
  $t2 = [00|00|00|19]

      add   $v0,$v0,$t2
  $v0 = [00|00|00|19]

      addi  $t1,$t1,4
  $t1 = [10|01|00|04]

      addi  $t0,$t0,-1
  $t0 = [00|00|00|04]

      j     lp
```

Figure 3.52: Partial trace of a program which sums the values in an array of five whole numbers, beginning at `grades`.

```
           .text
           la    $t0, numbers     # pointer to array
           li    $t1, 5           # counter
           li    $v0, 0           # accumulator
loop:
           ble   $t1, $0, done
           lw    $t2, 0($t0)
           ble   $t2, $0, notPos
           add   $v0, $v0, $t2
notPos:
           addi  $t0, $t0, 4     # pointer to next word
           addi  $t1, $t1, -1    # decrement counter
           j     loop
done:

          .data
numbers:  .word   -12, 32, 0, -3, 4
```

2. Modify the example in this section so that it will find the sum of the values in an array of any length. Assume the array is followed by a label: `end`.

   ```
            .data
   array:   .word 25, 63, -45, 0, 12, -25, 66, 99
   end:     .word 0     # Marks the end of the array.
   ```

3. Write a program which will scan the values in an array named `array` and leave the smallest value in register $v0. Assume the length of the array is not 0. Assume the end of the array is marked by the label `end` as in the preceding exercise.
   Hint: Use the first value in the array as a temporary result, then replace it if you find a value which is smaller.

   ```
            .data
   array:   .word 25, 63, -45, 0, 12, -25, 66, 99
   end:     .word 0     # Marks the end of the array.
   ```

4. Write a program to determine if the values in an array are in ascending order. If so, leave 1 in register $v0. If not, leave 0 in register $v0. Assume the length of the array is not 0. Assume the end of the array is marked by the label `end` as in the preceding exercise.

   ```
            .data
   array:   .word -25, 63, 450, 450, 512
   end:     .word 0     # Marks the end of the array.
   ```

5. Write a program which will find the vector sum of two arrays having the same length, leaving the sum in an array named `result`. It should add corresponding elements of the two arrays, producing an array of the same length as the result.

```
          .data
array1:   .word  16, 17, 0 -2, 4, 5
end1:     .word  0
array2:   .word  0, 3, 5, 2, -7, 16
end2:     .word  0
result    .word  0
```

3.10 Functions

In mathematics a *function* is a mapping from a set of values, called the domain, to a set of values, called the range. Functions may have 0 or more *arguments* (also known as parameters). The arguments may be thought of as the input(s) to the function, which produces a single range value as its result. Examples of mathematical functions are:

- $f(x) = sin(x) \cdot cos(x)$
- $g(x, y, z) = x^2 + y \cdot z - 4 + f(0.5)$
- $exp(x) = 1 + x + x^2/2 + x^3/6 + x^4/24$

High level programming languages provide for sub-programs in one form or another. In C/C++ these are called *functions*, and in Java they are called *methods*.[17] We will use the term 'function' here. We note that an assembly language function may have up to four *parameters* which are considered input to the function, and an assembly language function, unlike a mathematicsl function or a java method, may have as many as two explicit return values which may be thought of as the output of the function (though it may also have *side effects*).

3.10.1 Function Calls - jal and jr

Two additional instructions will be needed to implement function calls in assembly language. The *jump and link*, `jal` instruction is used to *call*, or *invoke*, a function. When executed, the jal instruction will do two things:

1. It will load the address of the next instruction into the $ra register. This is the *return address*, or the address to which the function should return control when it terminates.

[17] Some languages use the term *procedure* for a function which has no explicit return value, and some use the general term *sub* or *subprogram*.

mnemonic	format	meaning	example
jal	jal label	$\$ra \leftarrow address\ of\ next\ instr$ $\rightarrow$ `label`	 jal myFunction

Figure 3.53: Format and meaning of the jump and link instruction, with example

mnemonic	format	meaning	example
jr	jr $reg	$\rightarrow reg[\$reg]$	jr $ra

Figure 3.54: Format and meaning of the jump register instruction, with example

2. It will perform an (unconditional) jump to the specified label, which should label the beginning of the function being called.

The format and meaning of the `jal` instruction is shown in Fig 3.53.

The other instruction needed to implement functions involves a *return* to the calling function. The instruction is *jump register*, or `jr`. This instruction will perform an (unconditional) jump to the instruction whose address is in a given register, normally the $ra register. Assuming that the function was called with a `jal` instruction, this will effectively return control to the instruction following the function call. The format and meaning of the `jr` instruction is shown in Fig 3.54.

Fig 3.55 depicts the flow of control when a function is called (arrow to function), and when it returns to the calling function (arrow back to main).

As an example of a function and function call, we use the following program which includes a function to rearrange two contiguous words of memory so that the smaller is first. For example if the two words are 12 and 3, they would be swapped, to be 3 and 12. However if the two words were -9 and 3, they would not be swapped since -9 is smaller than 3. We will name this function `order2`, and it will assume that the address of the first word is in register $a0.

```
######################### Begin main program
           .text
main:
           la      $a0, words
           jal     order2
           addi    $a0, $a0, 0       # no effect
           .data
words:     .word   17, 3
######################### End main program

######################### Begin order2 function
           .text
### Arrange two contiguous words of memory in ascending order.
### Pre: Register $a0 contains the address of the first word.
### Author: sdb
```

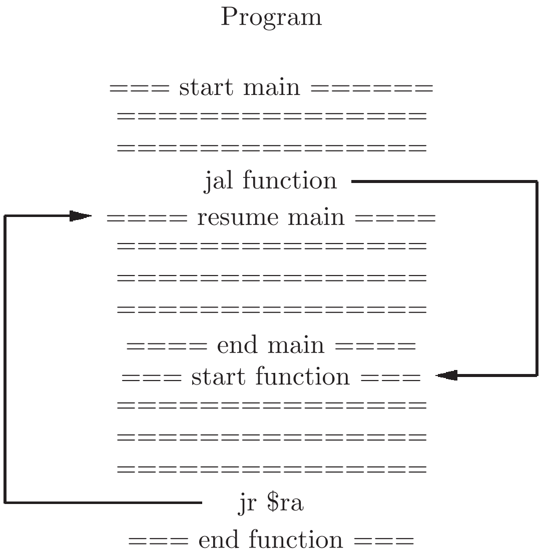


Figure 3.55: Diagram of a function call and return

```
order2:
           lw     $t0, 0($a0)    # load first word
           lw     $t1, 4($a0)    # load second word
           ble    $t0, $t1, done # already in ascending order?
           sw     $t0, 4($a0)    # store first word
           sw     $t1, 0($a0)    # store second word
done:
           jr     $ra            # return to calling function
######################### End order2 function
```

We note a few important points in regard to this example:

- The function is clearly delineated with comments, showing the beginning and end, as well as the purpose of the function.
- The part shown as the `main` program is included simply to test the function. It contains the call to the function (`jal order2`). It is given the name `main`, though this label is not used, except for documentation.
- The instruction `addi    $a0, $a0, 0` has no effect. It is included simply so that we can run the main program to completion by stepping through the statements one at a time, using MARS (we have not yet learned how to terminate a program normally).
- When the main program calls the `order2` function, the `jal` instruction puts the address of the next instruction (`addi    $a0, $a0, 0`) into register $ra.

- Since the order2 function is expecting the address of the memory words to be swapped in register $a0, it is the responsibilty of the main program to ensure that this is the case. This is done with a la instruction. This is known as a *precondition*; the function will work correctly only if the precondition is satisfied.
- This function has no explicit result. However, it does have side effects: it may change data in memory.

If you run the above program using MARS, you should single-step one instruction at a time and stop at the addi instruction in the main program. If you run it at full speed, control will fall through into the function after the last line of the main program, which is not desired. We will find a way to improve on this below.

We emphasize the importance of separating the function itself from the code that is used to test it - the main program. This testing code is often called a *Driver*; it serves no other purpose. Once we are sure the function is working correctly it can be extracted and used in other programs, where needed.

Terminating a program with the MARS simulator

At this point we are single-stepping through a program to verify its correctness. We might wish to run it at full speed, and view the final results; but we will need a way of terminating the program when it is finished. To do this, we make use of a *system call*. The system call provided by MARS is syscall. This is not a MIPS instruction, but is a special call to the MARS system. There are several uses of syscall, which we shall examine later in this chapter. A syscall uses the contents of register $v0 to determine the desired action. To terminate the program, $v0 should contain 10. An example is shown below, in which we add two values from memory and store the result back to memory:

```
          .text
          lw    $t0, first
          lw    $t1, second
          add   $t0, $t0, $t1
          sw    $t0, result
          li    $v0, 10         # return code for syscall
          syscall               # terminate the program
          .data
first:    .word 5
second:   .word 9
result:   .word 0
```

Note that this program will terminate with no error messages when finished; this is a much better way to terminate a program.

We could also make this change in the driver for our order2 function above. However there is a potential problem with our order2 function, which we examine next.

Ordering three contiguous words

Suppose we wish to write a function which will place three contiguous words of memory in ascending order. We could do this in three steps, using our `order2` function as follows:

1. Place the first and second words in order, if necessary.
2. Place the second and third words in order, if necessary.
3. Place the first and second words in order, if necessary.

Here is the program:

```
          .text
main:
          la    $a2, words
          jal  order3     # invoke order3
          li    $v0, 10
          syscall          # terminate
          .data
words:    .word 9, 0, -12

          .text

######################### Begin order3 function
###  Place 3 (signed) memory words in ascending order
###  Pre: Address of first word is in register $a0
### Author: sdb
order3:
          jal    order2           # Arrange first and second words
          addi   $a0, $a0, 4      # address of second word
          jal    order2           # Arrange second and third words
          addi   $a0, $a0, -4     # address of first word
          jal    order2           # Arrange first and second words
          jr     $ra              # return to main program
######################### End order3 function


######################### Begin order2 function
        (Not shown here)
######################### End order2 function
```

Here is the problem: when the order3 program is called, the return address to the main program is in the $ra register. This address is then clobbered by the `jal order2` instruction in `order3`. The `order2` function will work correctly, and return correctly to the `order3` function all three times that it is called. However, when `order3` attempts to return to the main program, it will fail because the return address in the $ra register has been clobbered by the `jal order2` instruction. This will cause the `jr $ra` instruction to jump to itself - an infinite loop! This is clearly not what is desired.

To rectify this problem in the `order3` function, we will make a copy of the $ra register contents in another register, then move it back to $ra when ready to return to the main program. This modified `order3` function is shown below:

```
######################### Begin order3 function
###  Place 3 (signed) memory words in ascending order
###  Pre: Address of first word is in register $a0
###  Author: sdb
order3:
         move   $t2, $ra         # copy return address
         jal    order2           # Arrange first and second words
         addi   $a0, $a0, 4      # address of second word
         jal    order2           # Arrange second and third words
         addi   $a0, $a0, -4     # address of first word
         jal    order2           # Arrange first and second words
         move   $ra, $t2         # return address
         jr     $ra              # return to main program
######################### End order3 function

######################### Begin order2 function
         (Not shown)
######################### End order2 function
```

Note that we could have avoided moving the return address back to $ra by changing the last instruction to `jr $t2`, but for reasons made clear later, this is not a good practice. Also, we were careful not to use registers $t0 nor $t1 for this purpose, because they would be clobbered by the `lw` instructions in `order2`.

What would happen if the `order2` function was later modified to use $t2 for some other purpose? What would happen if the `order2` function was modified to call some other function? When working as a member of a team, how can your functions call the functions of other team members without conflicting usages of registers? These questions will be resolved in the section on register conventions.

3.10.2 Function Parameters

Methods and functions in high level programming languages allow for optional parameters. An example of a java method declaration with three parameters in the signature is:

```
int myMethod (int x, float y, char z)
```

This means that the method, when called, must be provided three values - an int, a float and a char. In a MIPS assembly language program we can use registers to pass parameter information to a function. Though we could use any registers for this purpose, we suggest using only the registers $a0, $a1, $a2, and $a3.[18] This is called a *register convention*, and is discussed further below.

Does this mean that a function is limited to only four parameters? No, one of these registers could contain the memory address of several contiguous words of memory constituting an unlimited number of additional parameters. This is what we did above in the `order3` function.

API for functions

You may be familiar with the concept of an API, which stands for *Application Program Interface*. An API provides all the information needed to make use of a software module, such as a function. Some high level languages, such as Java, provide the capability of generating readable web pages, with links, from the API.

For our purposes the API in an assembly language function will consist of several comments which serve to delineate the function, and will contain information such as the following;

- Name of the function
- Author, date, etc.
- Overall purpose of the function
- Preconditions: Conditions which must be satisfied in order for this function to work correctly.
- Registers containing parameters, and their intended purpose (think of this as the input to the function)
- Side effects: memory locations which are affected, registers which are changed, output produced, etc.
- Explicit values returned (in the $v0 and/or $v1 registers, for example)
- Revision history, or version number

With a well-written API, other programmers will be able to call your function without having to read through the detailed statements in the function. Everyhing they need to know should be in the API. This is extremely important when using assembly language because it can be extemely difficult to gain a full understanding of a program simply by reading the constituent statements.

3.10.3 Register Conventions and the Call Stack

We now return to some of the questions raised in the preceding section. When developing larger programs consisting of many functions, it will be extremely

[18] The 'a' stands for *argument*, which is essentially the same as a parameter.

difficult to avoid conflicts in register usage. For example, a function may incorrectly clobber a register value which is needed by some other function. Consequently register values which must be preserved across function calls need to be saved (somewhere) in memory when a function is called, and reloaded before the function teminates.

Moreover, the problem of clobbering return addresses was raised in the previous section. When a function is called, the return address is loaded into the $ra register. If that function then calls some other function, the $ra register is clobbered, so it would have to be saved and reloaded as described above. Where in memory should it be saved? This problem becomes worse if the called function calls another function, which in turn calls another function, ... And suppose that a function calls itself?. [19]

These problems become even more serious in an environment where several people are working as a team on one software project. They must come to some agreement as to how function calls will work. In cases where registers need to be saved across a function call, the programmers need to decide which is responsible for saving register values - the calling function or the called functions. The MIPS architecture defines such an agreement, which we call *register conventions*, some of which we have seen already. We define these below:

- Function parameters should be in registers $a0 .. $a3.
- Functions should return explicit result(s) in registers $v0 and/or $v1.
- If any of the registers $s0 .. $s7[20] are changed in a function, that function should reload their original values before returning to the calling function. Their values should be saved on the *call stack* (to be defined below). The $s registers correspond to local variables in a java method or C/C++ function.
- If any of the registers $t0 .. $t9[21] need to be preserved when a function is called, it is the responsibility of the calling function to save them, preferably on the call stack.
- The return address (i.e. the $ra register) must be saved on the call stack.
- The $sp register (*stack pointer*) is used to access the memory stack.
- The $at register (*assembler temporary*) is reserved for use by the assembler, and must not be used by the assembly language programmer.
- The $fp (frame pointer) and $gp (global pointer) are used by compilers to implement function calls. We will not be concerned with them here.
- The $k0 and $k1 registers are reserved for use by the kernel (i.e. operating system), and must not be used by the assembly language programmer.

[19] A function recursive calls instead

[20] The 's' stands for *saved*

[21] The 't' stands for *temporary*

registersi	purpose	responsible for saving
\$a0..\$a3	function parameters	[see API]
\$v0..\$v1	function result	[see API]
\$s0..\$s7	temporary results	called function
\$t0..\$t9	temporary results	calling function
\$ra	return address	called function
\$sp	stack pointer	
\$at	assembler temporary	[do not use]

Figure 3.56: MIPS register conventions

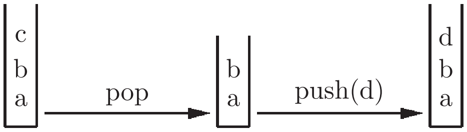


Figure 3.57: Diagram of a stack containing the values a,b,c, showing the effects of a *pop* operation followed by a *push(d)* operation

In general, functions should not expect parameters ($a registers) to be preserved when a function terminates.[22] If a function wishes to use a parameter as a result or side effect, this must be explicit in the API. These conventions are summarized in Fig 3.56.

Stacks

Finally we address the question: Where in memory should the $ra register and other registers be saved? In order for function calls to work in a general way (no matter how deeply function calls may be nested) and to handle recursive function calls, we will need a *software stack*. A *stack* is a last-in first-out (LIFO) structure. When extracting an item from a stack, it must be the most recent item that was added to the stack. The process of adding an item to a stack is usually called a *push* operation, the process of determining the last item added is called a *peek* operation, and the process of removing an item is called a *pop* operation. Fig 3.57 depicts these operations on a stack, which is shown vertically, with the last item added on the top.

MIPS stack

We will implement a software stack using the $sp register. This register contains the memory address of the top value on the stack. When we wish to push a value onto the stack we will first decrement the $sp register, then store the value to be pushed at the address specified by the $sp register. Thus if we wish to push the value of the $ra register, we would use the following two instructions

[22] This is known as *call by reference*.

```
        addi    $sp, $sp, -4       # decrement stack pointer
        sw      $ra, 0($sp)        # store onto stack
```

To peek at the top value on the stack (i.e. put its value into a register), simply use the $sp register to access that memory location:

```
      lw   $ra, 0($sp)        #  peek
```

To pop a value off the stack, simply increment the $sp register:

```
      addi   $sp, $sp, 4      #  pop
```

Note that we *decrement* the stack pointer when doing a *push* operation, and we *increment* the stack pointer when doing a *pop* operation. This means that the stack grows toward a *lower* memory address, which may seem counter-intuitive.

Using the stack for function calls

We have seen how to push the $ra register onto the memory stack when a function is called. According to the MIPS register conventions, we must also save the 8 $s registers. These must be pushed onto the stack, because in general we do not know whether the function call is (indirectly) recursive. Each $s register which is modified somewhere in a function must be pushed onto the stack when the function is entered, and loaded from the stack when the function terminates. The $s registers are pushed onto the stack the same way the $ra register is pushed - by using the stack pointer register, $sp.

For example, if a function modifies the $s2, $s5, and $s7 registers then it would push them, along with the $ra register onto the stack when the function is entered as shown in Fig 3.58. It would also pop them from the stack when the function is to return to the calling function.

In Fig 3.58 note that we need to decrement the $sp register by 16 instead of by 4 because we are pushing 4 registers onto the stack instead of 1, and there are 4 bytes in a word. As usual, the stack grows toward low-address memory.

For a more complete example, we return to our `order2` function, which arranges two contiguous words of memory in ascending order. However, this time we will use registers $s0 and $s1 instead of $t0 and $t1 for temporary storage. The example is shown below:

```
######################### Begin order2 function
### Arrange two contiguous words of memory in ascending order.
### Pre: Register $a0 contains the address of the first word.
### Post: Register $a0 is unchanged
### Author: sdb
#   This version uses $s registers which must be pushed
#   onto the stack.
           .text
order2:
           addi   $sp, -12($sp)  # pushing 3 regs
           sw     $ra, 0($sp)    # return address
           sw     $s0, 4($sp)    # push 2 s regs
           sw     $s1, 8($sp)
```

```
#################   Begin function
###  API
###  This function modifies registers $s2, $s5, and $s7
###  Author:  sdb

name:      addi   $sp, $sp, -16   # pushing 4 regs
           sw     $ra, 0($sp)     # return address
           sw     $s2, 4($sp)     # push 3 s regs
           sw     $s5, 8($sp)
           sw     $s7, 12($sp)

...
done:                             # return to calling function
           lw     $s7, 12($sp)    # pop 3 s regs
           lw     $s5, 8($sp)
           lw     $s2, 4($sp)
           lw     $ra, 0($sp)     # pop return address
           addi   $sp, 16, $sp    # original stack pointer
           jr     $ra             # return to calling function
#################   End function
```

Figure 3.58: A function which modifies registers $s2, $s5, and $s7 needs to push them on the stack on entry to the function, and pop them from the stack when exiting the function

```
           lw     $s0, 0($a0)    # load first word
           lw     $s1, 4($a0)    # load second word
           ble    $s0, $s1, done # already in ascending order?
           sw     $s0, 4($a0)    # store first word
           sw     $s1, 0($a0)    # store second word
done:
           lw     $s1, 8($sp)    # pop 2 s regs
           lw     $s0, 4($sp)
           lw     $ra, 0($sp)    # pop return address
           addi   $sp, 12($sp)
           jr     $ra            # return to calling function
######################### End order2 function
```

It may seem that this version of `order2` is unnecessariy complicated by the fact that we are using $s registers. However, in more complex software systems, it will be essential that we use the $s registers, especially if our functions are (potentially) recursive.

We now rewrite our order3 function so that it also agrees with the MIPS register conventions:

```
######################### Begin order3 function
###  Place 3 (signed) memory words in ascending order
###  Pre: Address of first word is in register $a0
###  Author: sdb
order3:
          addi   $sp, $sp, -4
          sw     $ra, 0($sp)      # push return address on stack

          jal    order2           # Arrange first and second words
          addi   $a0, $a0, 4      # address of second word
          jal    order2           # Arrange second and third words
          addi   $a0, $a0, -4     # address of first word
          jal    order2           # Arrange first and second words

          lw     $ra, 0($sp)      # pop return address from stack
          addi   $sp, $sp, 4
          jr     $ra              # return to main program
######################### End order3 function
```

Name conflicts

At this point we have been using labels in our program text sections, without regard to the possibility of duplicate labels. The assembler will not permit duplicate labels; all labels in a source file must be unique.

Consider writing a program in which we wish to find the range of an array; i.e., we want to find the difference between the largest and smallest values in an array. We will do this with two separate functions, one to find the smallest and one to find the largest, then subtract to get the range. The two functions to find the smallest and largest values are shown below:

```
#####################   Begin smallest function
### Return the smallest value of an array in register $v0
### Pre: Register $a0 points to first word in the array
### Pre: Register $a1 points to word after the last word in the array
### Pre: Length of the array is at least 1
### Post: Registers $a0, and $a1 are unchanged
          .text
smallest:
          addi   $sp, $sp, -12
          sw     $ra, 0($sp)     # push return address on stack
          sw     $s0, 4($sp)     # save s registers on stack
          sw     $s1, 8($sp)

          move   $s0, $a0        # ptr to array
          lw     $v0, 0($s0)     # smallest seen so far
lp:
          ble    $a1, $s0, done
          lw     $s1, 0($s0)
          bge    $s1, $v0, ok
          move   $v0, $s1     # new candidate for smallest
ok:
          addi   $s0, $s0, 4
          j      lp
done:
          lw     $s1, 8($sp)     # pop s registers
          lw     $s0, 4($sp)
          lw     $ra, 0($sp)     # pop return address
          addi   $sp, $sp, 12
          jr     $ra
#####################   End smallest function

#####################   Begin largest function
### Return the largest value of an array in register $v0
### Pre: Postz Register $a0 points to first word in the array
### Pre: Register $a1 points to word after the last word in the array
### Pre: Length of the array is at least 1
### Post: Registers $a0, and $a1 are unchanged
          .text
largest:
          addi   $sp, $sp, -12
          sw     $ra, 0($sp)     # push return address on stack
          sw     $s0, 4($sp)     # save s registers on stack
          sw     $s1, 8($sp)

          move   $s0, $a0        # ptr to array
          lw     $v0, 0($s0)     # largest seen so far
lp:
          ble    $a1, $s0, done
          lw     $s1, 0($s0)
          ble    $s1, $v0, ok
          move   $v0, $s1        # new candidate for largest
ok:
          addi   $s0, $s0, 4
          j      lp
done:
          lw     $s1, 8($sp)     # pop s registers
          lw     $s0, 4($sp)
          lw     $ra, 0($sp)     # pop return address
          addi   $sp, $sp, 12
          jr     $ra
#####################   End largest function
```

Each of the functions shown above is fine, independently. However there is a problem when we include them both in the same source file: there are name conflicts. The assembler will tell us that the labels `lp`, `ok`, and `done` are duplicated; each is defined more than once. The assembler will not allow us to run this program until these errors are corrected.

We could solve this problem by changing each duplicated label to something else; for example, the `lp` label in the function `largest` could be changed to `lp2`, in which case the jump instruction would be changed to `j      lp2`. But suppose we had several functions, each with a `lp` label; we would have to carefully distinguish them in some way. We propose a more systematic solution to the problem of name conflicts. Each label in a function will have the function's name appended to it (beginning with an underscore character). Thus in function `smallest` we would have the label `lp_smallest`, and in the function `largest` we would have the label `lp_largest`. If we do this consistently with all labels in all functions, and all function names are distinct, we should never have to worry about name conflicts. A revised version of the `smallest` function which eliminates the possibility of name conflicts is shown below.

```
#####################   Begin smallest function
### Return the smallest value of an array in register $v0
### Pre: Register $a0 points to first word in the array
### Pre: Register $a1 points to word after the last word in the array
### Pre: Length of the array is at least 1
### Post: Registers $a0, and $a1 are unchanged
          .text
smallest:
```

```
          addi   $sp, $sp, -12
          sw     $ra, 0($sp)     # push return address on stack
          sw     $s0, 4($sp)     # save s registers on stack
          sw     $s1, 8($sp)

          move   $s0, $a0        # ptr to array
          lw     $v0, 0($s0)     # smallest seen so far
lp_smallest:
          ble      $a1, $s0, done_smallest
          lw     $s1, 0($s0)
          bge    $s1, $v0, ok_smallest:
          move   $v0, $s1        # new candidate for smallest
ok_smallest:
          addi   $s0, $s0, 4
          j      lp_smallest:
done_smallest:
          lw     $s1, 8($sp)     # pop s registers
          lw     $s0, 4($sp)
          lw     $ra, 0($sp)     # pop return address
          addi  $sp, $sp, 12
          jr     $ra
#####################   End smallest function
```

Similar changes made to the `largest` function, though not strictly necessary, would be consistent with our new convention on labels. Our goal is to be able to copy a tested and trusted function, and paste it into any source file which may have a need for it. We should not have to make any changes to the function.[23]

Local data for functions

Thus far all of our functions have been able to use registers for temporary storage. Suppose a function has a need for so much temporary storage that memory, rather than registers, must be used. We would like this memory area to be considered 'local' to the function. It is needed by the function, and is not needed elsewhere. MARS will permit multiple .data sections and .text sections in one source file.

As an example, consider a function which will count the number of positive, negative, even, and odd values in an array of unlimited length. We call this a *distribution*. This function will use local data which we call a *buffer*.

```
######################  Begin distr function
### Find the distribution of positive, negative, even, odd, and zero
### values in an array of unlimited length.
```

[23] We will see later in Chapter 5 that MARS provides an `.include` directive which eliminates the need for copying and pasting, thus eliminating the problem of duplicated code in multiple source files.

```
### Pre: Register $a0 points to first word of the array
### Pre: Register $a1 points to the word following the last word of the array
### Return, in register $v0, the address of a buffer in which:
###   Word    contains
###    0      number of even positive values
###    1      number of even negative values
###    2      number of odd  positive values
###    3      number of odd  negative values
###    4      number of zero values
### Author:  sdb
                .text
distr:
                addi    $sp, $sp, -8
                sw      $ra, 0($sp)     # push return address on stack
                sw      $s0, 4($sp)     # save s registers on stack

lp_distr:
                ble     $a1, $a0, done_distr # reached end of array?
                lw      $t0, 0($a0)     # array element
                addi    $a0, $a0, 4
                blt     $t0, $0, neg_distr
                bgt     $t0, $0, pos_distr
                li      $a2, 16         # increment zero ctr
                jal     incr
                j       lp_distr
neg_distr:
                andi    $t0, $t0, 1     # test low order bit
                beq     $t0, $0, negEven_distr
                li      $a2, 12               # increment neg odd ctr
                jal     incr
                j       lp_distr
negEven_distr:
                li      $a2, 4          # increment neg even ctr
                jal     incr
                j       lp_distr
pos_distr:
                andi    $t0, $t0, 1     # test low order bit
                beq     $t0, $0, posEven_distr
                li      $a2, 8          # increment pos odd ctr
                jal     incr
                j       lp_distr
posEven_distr:
                li      $a2, 0          # increment pos even ctr
                jal     incr
                j       lp_distr
done_distr:
```

```
                la      $v0, buffer_distr  # return ptr to buffer
                lw      $s0, 4($sp)     # pop s register
                lw      $ra, 0($sp)     # pop return address
                addi    $sp, $sp, 8
                jr      $ra             # return

                .data                   # local data
buffer_distr:
                .word    0,0,0,0,0      # counters
#####################   End distr function

 ######################  Begin incr function
### Increment one of the words in the buffer of function distr
### Pre: Register $a2 holds displacement from start of buffer
                .text
incr:
                la     $t2, buffer_distr
                add    $t2, $t2, $a2   # add displacement
                lw     $t3, 0($t2)
                addi   $t3, $t3, 1
                sw     $t3, 0($t2)
                jr     $ra
#######################  End incr function
```

Note that the local data is called `buffer_distr` because it is in the function `distr` (to distinguish it from local buffers in other functions). Also, it might have been a good idea to name our called function `incr_distr` instead of `incr`, to indicate that it is used as a 'local' function by `incr`, and to distinguish it from other functions which may have a similar purpose.

3.10.4 Recursive Functions

A recursive function is a function which calls itself. In order for this to work, the function must satisfy two properties:

- There must be a *base case* which does not involve a recursive call.
- When the function calls itself, the input to the function (usually contained in the parameters) must be reduced in some way.

If these two properties are satisfied, we will avoid 'infinite recursion' and it should work as intended. In many cases a recursive function call can be replaced by a loop, but not always. There are some tasks which require recursive calls (which in turn require the use of a stack). Note that it is possible for a function to be indirectly recursive. If function A calls function B, and function B calls function A, both function A and function B are (indirectly) recursive. Thus it is not always evident at first glance whether a function is recursive, and that is one reason that we should save registers on the stack.

As an example we choose the fibonacci sequence, a well known sequence of numbers which is found in various natural phenomena. The sequence is:

1 1 2 3 5 8 13 ...

Note that after the first two numbers, each number in the sequence is the sum of the previous two numbers, so the next number after 13 would be 8 + 13 = 21.

We can state this more formally with a recursive definition of the fibonacci sequence, where fib(n) is the nth number in the sequence:

$fib(n) = 1\ if\ n < 3$

$fib(n) = fib(n-1) + fib(n-2)\ if\ n > 2$

We will now use this definition to implement the fib function with a recursive call. Note that the first part of the definition is the base case (does not involve a call to fib). The second part involves two calls to fib, but the input is reduced for each of those recursive calls. The function is shown in Fig 3.59.

Note that we are using $s registers here to store the values of n-1 and n-2. These are values which need to be pushed onto the stack, just as local variables in a recursive java method are pushed onto the runtime stack.

Fig 3.59 shows a rather inefficient way to find the nth Fibonacci number. For example, to find fib(5) we add fib(4) + fib(3). To find fib(4) we add fib(3) + fib(2); thus in finding fib(5) there are at least two separate calls to fib(3). This function can be programmed much more efficiently with a loop. We chose to use recursion merely to demonstrate the correct usage of the stack and recursive functions.

3.10.5 Exercises

1. The function shown below is supposed to return the sum of three contiguous words of memory. Modify the function (including its API) so that it correctly observes the MIPS register conventions. It should use registers $s0 and $s1 for temporary storage.

   ```
   ################### Begin function example
   ###  Find the sum of the three numbers stored in memory
   ###  at the address in register $t0.
   ###  Return the sum in register $t1.
   ###  Author:
                .text
   example:
                lw      $s0, 0($t0)
                lw      $s1, 4($t0)
                add     $t1, $s0, $s1
                lw      $s0, 8($t0)
                add     $t1, $t1, $s0
   #################### End function example
   ```

```
######################## Begin function fib
## Find the nth number in the fibonnaci sequence
## Pre: n is in $a0
## Return nth fibonnaci number in $v0
        .text
fib:
        addi   $sp, $sp, -16
        sw     $ra, 0($sp)
        sw     $s0, 4($sp)
        sw     $s1, 8($sp)
        sw     $s2, 12($sp)

        li     $s0, 2         # check for base case
        bgt    $a0, $s0, rec_fib  # if not, go to recursive case
        li     $v0, 1         # result is 1
        j      done_fib           # return to calling function
rec_fib:
        addi   $a0, $a0, -1   # n-1 for recursive case
        addi   $s1, $a0, -1   # n-2
        jal    fib
        move   $s2, $v0       # fib(n-1)
        move   $a0, $s1       # n-2
        jal    fib
        add    $v0, $v0, $s2  # fib(n-1) + fib(n-2)
done_fib:
        lw     $ra, 0($sp)    # load saved regs
        lw     $s0, 4($sp)
        lw     $s1, 8($sp)
        lw     $s2, 12($sp)
        addi   $sp, $sp, 16
        jr     $ra            # return to calling function

######################## End   function fib
```

Figure 3.59: Function to find the nth number in the Fibonacci sequence

2. Write and test a MIPS function named reverse3 to reverse the order of 3 contiguous words in memory. It can expect that the address of the first word is in register $a0. Be sure to include an appropriate API. To test your solution you will need a main program which calls reverse3.

3. The function named 'sum', shown below, is supposed to return the sum of its two arguments, registers $ao and $a1 in register $v0. What is wrong with this function?

```
            .text
sum:
            addi    $sp, $sp, -4
            sw      $ra, 0($sp)       # push return address
            add     $v0, $a0, $a1
            lw      $ra, 0($sp)       # pop return address
            addi    $sp, $sp, 4
            jr      $ra               # return
```

4. Write the following java method as a MIPS function:

```
/** @return first parameter plus second parameter multiplied
 *  by 16
 */
int meth (int x, int y)
{   return x + y * 16; }
```

5. Write and test a MIPS function, named reverseArray, to place the words of an array in reverse order. For example, if the array is 43, -12, 5, 6 then when the function terminates, the array should be 6, 5, -12, 43. The arguments to the function should be the start address of the array, in register $a0, and the ending address (the address of the word *after* the last word in the array) in register $a1. Use at least one $s register for temporary storage. Don't forget to include the API and observe MIPS register conventions.

6. (a) Write and test a MIPS function named addrSmallest which will return the address of the smallest word in a memory array. The arguments to the function should be the start address of the array, in register $a0, and the ending address in register $a1. You may assume the length of the array is at least 1.

   (b) Write and test a MIPS function named 'sort' to sort the words of a memory array in ascending order. Use the following algorithm (known as Selection Sort):

   - For each position in the array, find the address of the smallest value beginning at that position. (Call your addrSmallest function from part (a))

- Swap the word at that position with the smallest.

7. (a) Write and test a function named 'pal' which will determine whether a memory array is a palindrome: It reads the same backwards as it does forward. Each of the following arrays is a palindrome:

   4, 9, -3, -3, 9 4

   2, 0, 2

   17

   Return a one in register $v0 only if the array is a palindrome, and zero if it is not a palindrome. Your function should use a loop. The arguments to the function should be the start address of the array, in register $a0, and the ending address in register $a1.

   (b) Repeat part (a) but your function should use a recursive call rather than a loop. Name the function palRecursive.

   Hints:

   - Base case: The length of the array is less than two. It must be a palindrome, so put zero into register $v0 and return.
   - Base case: Compare the first and last words of the array. If they are not equal, the array could not be a palindrome, so put zero into register $v0 and return.
   - Recursive case: Determine whether the rest of the array is a palindrome by calling palRecursive with different arguments.

3.11 Strings and String Functions

All the examples we have dealt with thus far involved calculations with whole numbers. Here we will see how non-numeric information can be represented and manipulated. To represent non-numeric information, such as the characters on the keyboard, we use a numeric code in which each character is represented by an 8-bit code (i.e. one byte). This code is called ASCII.[24] For example, the code for the character 'k' is $01101011_2 = 6b_{16} = 107$. The full range of ASCII codes is shown in the appendices; many of these characters are not on your keyboard (and many are not viewable as characters on a typical display).

A *string* is simply an array of characters. I.e. the characters of a string are in contiguous memory locations, with one character per byte (i.e. 4 characters per word). Each byte has a unique memory address. The address of the first character in the string is the string's address.

3.11.1 Initializing Memory with Strings

In assembly language there are two directives which allow us to store a string in memory:

[24] This code is actually a subcode of a much larger and more general code called Unicode, which is a 16-bit code, and includes foreign alphabets.

- The `.ascii` directive allows one to initialize the data memory with the characters of a string:

```
        .data
name:   .ascii    "harry"
```

- The `.asciiz` directive is the same as the `.ascii` directive but the string is terminated with a *null byte*. This null byte is a character whose code is 0. (Not the character '0', but the binary value 0, which is 00000000_2):

```
        .data
name:   .asciiz   "harry"
```

These two definitions may look alike, but the first occupies 5 bytes of memory, and the second occupies 6 bytes, because of the null byte at the end. One of the issues to be addressed when working with strings is the need to determine when we have reached the end of a string. Some high level languages, such as Java, store a numeric length, along with the characters of a string. Other languages, such as C store a null byte at the end of the string (and calculate the length if needed). The MIPS directive `.asciiz` is in agreement with C strings - terminated by a null byte. With the `.asciiz` directive, the null byte serves as a sentinel, or terminating character.

In what follows, we will generally use the `.asciiz` form and rely on the null byte to terminate the string. When processing the characters of a string, the logic will be very similar to the logic we used when processing the elements of an array, with two main distinctions:

- Since each element of the string consists of one byte rather than one word, we will increment the address register by 1 instead of 4, to advance to the next character of a string.
- We will normally wish to load a single character into a register; thus the `lw` and `sw` instructions are not what we want. Instead we will use *byte* instructions, introduced in the next section.

3.11.2 Byte Instructions - `lbu` and `sb`

To load a single byte into the low order 8 bits of a register, we will use the `lbu` instruction (load byte unsigned).[25] This instruction copies one byte from memory into the low order byte of the register. It also zeros out the high order 24 bits (3 bytes) of the register.

To store a byte from a register into memory, we use the `sb` instruction (store byte), which will copy the low order 8 bits of the instruction to the specified memory location.

[25] There is also a `lb` (load byte) instruction which extends the sign bit of the byte being loaded through the high order 24 bits of the register; this will not normally be used when working with strings.

mnemonic	format	meaning	example
lbu	lbu \$rt, label lbu \$rt, d(\$rs)	$reg[\$rt]_{31:8} \leftarrow 0$ $reg[\$rt]_{7:0} \leftarrow mem[label]$ $reg[\$rt]_{31:8} \leftarrow 0$ $reg[\$rt]_{7:0} \leftarrow mem[\$rs + d]$	lbu \$t3, x lbu \$t3, 0(\$a0)
sb	sb \$rt, label sb \$rt, d(\$rs)	$mem[label] \leftarrow reg[\$rt]_{7:0}$ $mem[\$rs + d] \leftarrow reg[\$rt]_{7:0}$	sb \$t3, x sb \$t3, 0(\$a0)

Figure 3.60: Format of the load byte unsigned (lbu) and store byte (sb) instructions, using symbolic memory addresses (i.e. labels), and explicit addressessing

Both the `lbu` and the `sb` instructions may be given in either symbolic or explicit format, as shown in Fig 3.60. In that figure, note that we use subscripts to designate specific bit locations within a register - 0 is the position of the low order bit and 31 is the position of the high order bit.[26]

3.11.3 String Processing

We now have everything we need to process strings. Recall that a string is simply an array of characters (terminated with a null byte). Suppose we are given the memory address of the first character in a string, and we wish to find the number of characters, i.e. the length, of the string. All we need to do is use a loop to move from each character to the next, checking for the null byte, and increment a counter in the loop. The null byte is not considered part of the string, and should not be included in the length. A function named `strlen` which does this is shown in Fig 3.61.

In that function the first thing that is done in the loop is to check for termination (i.e. have we loaded the null byte?). This means that we could even process a string of length 0, in which the very first, and only, character is the null byte.

If you are using MARS to inspect data memory which contains strings, a word of caution is in order. The characters of a string are stored in reverse order *within each word*, though their addresses are in sequential order. For example, the string "Good Morning" will store "dooG" in the first word, "roM " in the second word, "gnin" in the third word, and the null byte in the low order byte of the fourth word, as shown in Fig 3.62. The address of the "G" is 10010000_{16} and the address of the "M" is 10010005_{16}.

We conclude this section with one more example of a string function - a function which will compare two strings. When comparing two strings, str1 and str2, this function will produce a result as follows:

- Return some negative number if $str1 < str2$. Intuitively, str1 is smaller than str2 if str1 precedes str2 alphabetically.

- Return 0 if $str1 = str2$.

[26] People have not always agreed on this bit numbering convention. See the well known article on this in *IEEE Computer*, Oct 1981, by Danny Cohen.

```
######################## Begin function strlen
## Return the length of a string in $v0.
## Pre: The string is terminated with a null byte, which
## is not included in the character count.
## Pre: Address of the string is in register $a0.
## Author:  sdb
       .text
strlen:
       addi  $sp, $sp, -4
       sw    $ra, 0($sp)        # push return address

       li    $v0,0              # counter
lp_strlen:
       lbu   $t0, 0($a0)        # next char of string
       beq   $t0, $0, done_strlen # end of string?
       addi  $v0, $v0, 1        # increment counter
       addi  $a0, $a0, 1        # address of next char
       j     lp_strlen
done_strlen:
       lw    $ra, 0($sp)
       addi  $sp, $sp, 4
       jr    $ra                # return
######################## End function strlen
```

Figure 3.61: Function to find the length of a string. Address of the string is in register $a0.

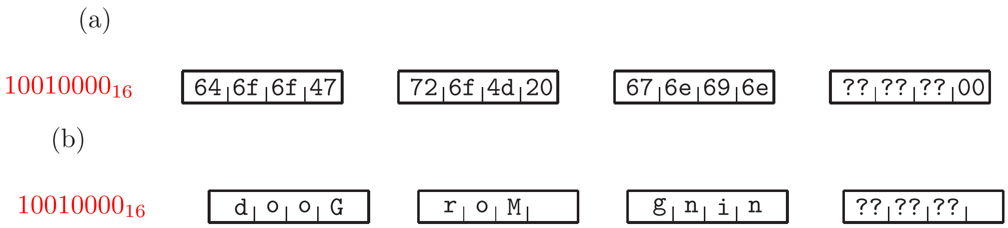


Figure 3.62: Diagram of the characters of the string “Good Morning” in data memory, showing the hex codes (a), and the characters(b)

- Return some positive number if *str*1 > *str*2. Intuitively, str1 is greater than str2 if str1 follows str2 alphabetically.

When comparing strings, we are *not* comparing the lengths of the strings. Some examples of string comparisons are:

```
"batter" <  "cat"
"zoo"    >  "zonk"
"john"   <  "johnson"
"foo"    =  "foo"
```

Note that if a string is a prefix of the string with which it is being compared, the prefix is smaller.

The logic we use to compare two strings is to compare corresponding characters of the two strings, beginning at the left end. If the characters are different, we know which string is smaller. If the characters are equal, we advance to the next position and repeat until we reach the end of one (or both) of the strings. When comparing two characters, we will load each into a register and compare the two registers; hence, we are actually comparing the ASCII codes of the characters. The function `strcmp` is shown in Fig 3.63.

In the `strcmp` function note that if we determine that the two characters are equal, and one of them is the null byte (zero), then both characters are the null byte, and the two strings must be equal. At this point register $v0 must be 0, which is the desired result when the two strings are equal.

3.11.4 Exercises

1. Write the API for the function shown below. In the API describe the purpose of the function, its parameters, precondiitions, if any, explicit result(s) if any, and side effects, or post conditions, if any. Give the function an appropriate name.

```
###############   Begin function foo
###  Put API here

             .text
foo:
             addi  $sp, $sp, -4
             sw    $ra, 0($sp)
lp_foo:
             lbu   $t0, 0($a0)
             beq   $t0, $0, done_foo
             sb    $a1, 0($a0)
             addi  $a0, $a0, 1
             j     lp_foo
done_foo:
             lw    $ra, 0($sp)
```

```
######################## Begin function strcmp
# Compare two strings.
# Pre: $a0 contains address of first source string, str1
# Pre: $a1 contains address of second source string, str2
# Pre: Each string is terminated with a null byte.
# Return in $v0:
#    A negative number if str1 precedes str2
#    A positive number if sr2 precedes str1
#    Zero if equal
# Author: sdb

strcmp:
     addi    $sp, $sp, -4        # push return address
     sw      $ra, 0($sp)

lp_strcmp:
     lb     $t0, 0($a0)          # load byte from str1
     lb     $t1, 0($a1)          # load byte from str2
     sub    $v0, $t0, $t1        # v0 = t0 - t1
     bne    $v0, $0, done_strcmp # different chars, finished

     #   The two chars are equal
     beq    $t0, $0, done_strcmp # end of str1 ?

     # Advance to next byte of each string
     addi   $a0, $a0, 1          # Go to next char of str1
     addi   $a1, $a1, 1          # Go to next char of str2
     j      lp_strCmp            # repeat the loop
done_strcmp:
       lw    $ra, 0($sp)
       addi  $sp, $sp, 4
       jr    $ra                 # return
######################## End function strcmp
```

Figure 3.63: Function to compare two strings. A negative result means the first string is smaller. A positive result means the second string is smaller.

```
            addi  $sp, 4($sp)
            jr    $ra
###############   End function foo
```

2. Write and test a function named `toUpper` which will convert all the alphabetic characters in a given string to upper case. Any uppercase characters or non-alphabetic characters should be unchanged. For example, the string "fooBaR!@" should be changed to "FOOBAR!@". Assume that register $a0 points to the string, and that it is terminated by a null byte.

   Hint: In Appendix .6.1 compare the binary ascii codes of lower case alphabetic characters with the binary ascii codes of the corresponding upper case characters.

3. Write and test a function named `isNumeric` which will determine whether a string consists entirely of numeric characters, i.e. the characters '0'..'9'. Your function should return a 1 in register $v0 if this is so; otherwise it should return a 0 in register $v0. Assume that register $a0 points to the string, and that it is terminated by a null byte. The string of length 0 is not a numeric string.

4. (a) Write and test a function named `toInt` which will convert a given string of numeric characters to a binary full word (i.e. an int), which is returned in register $v0. Assume that register $a0 points to the string, and that it is terminated by a null byte. Assume the given string is a valid positive int, and that its length is not 0.

   Hint: 10x = 8x + x + x

   (b) Modify your solution to part (a) to allow for negative numbers. The given string could begin with a '-' character.

5. Write and test a function which will extract a substring from a given string. Use the API shown below:

```
###############   Begin function substr
### Return the string which is a substring of a given string.
### Pre: Register $a0 points to the given string, which is
###    null-terminated.
### Pre: Register $a1 contains the starting position of the substring
###   (First position is 0)
### Pre: Register $a2 is the length of the substring.
### Pre: Register $a3 is the address of a memory buffer for the result,
###   which should be a null-terminated substring.
### Author:
```

6. Write and test a function which will concatenate two strings to produce a string result. For example, if the strings "Holy" and "Cow" are concatenated, the result would be "HolyCow".

```
###############   Begin function concat
### Return the string which is the concatenation of two
###   given strings.
### Pre: Register $a0 points to the first string.
### Pre: Register $a1 points to the second string.
### Pre: Both of the given strings are terminated by a null byte.
### Pre: Register $a2 points to a memory buffer for the result,
##    which should be a null-terminated string.
### Author:
```

Either, or both, of the given strings could have a length of 0.

3.12 Multiplication of Whole Numbers

In this section we will examine several ways of multiplying whole numbers in assembly language. Initially, we will assume there is no MIPS instruction for multiplication, and we will implement multiplication with software. Later we will expose the MIPS multiply instruction.

3.12.1 Multiplication with Software

Perhaps the easiest way to multiply two whole numbers is to use repeated addition: 5*7 = 7 + 7 + 7 + 7 + 7 = 35. In general, to multiply x*y we will use a loop. In the loop body we will add the operand y to an accumulator, which has been initialized to zero. The operand x will be used to control the loop. A function to accomplish this is shown in Fig 3.64.

There are better ways (i.e. faster ways) to multiply whole numbers, as we have implied in preceding sections. Recall that shifting the binary digits of a register to the left is the same as multiplying by a power of 2. By using the shift instruction, in conjunction with an add instruction, we can speed up the multiplication considerably. For example, note that
$200 * x = 128 * x + 64 * x + 8 * x = 2^7x + 2^6x + 2^3x$.

Instead of performing 200 add instructions, we need to perform 3 shift instructions and 2 add instructions, which is a huge improvement in efficiency. This implies that we can use a repeated shift-and-add algorithm for fast multiplication. This shift-and-add algorithm corresponds to the way we learned to multiply decimal numbers in elementary school.[27] A binary version of multiplication is shown in

$$
\begin{array}{ccccccccc}
 & & & & 1 & 0 & 1 & = & 5 \\
 & & \mathrm{x} & 1 & 0 & 1 & 1 & = & 11 \\
\hline
 & & & & 1 & 0 & 1 & 1 & \\
 & & & 1 & 0 & 1 & & 1 & \\
 & & 0 & 0 & 0 & & & 0 & \\
+ & 1 & 0 & 1 & & & & 1 & \\
\hline
 & 1 & 1 & 0 & 1 & 1 & 1 & = & 55
\end{array}
$$

Figure 3.65: Multiplication - the multiplier is $11 = 1011_2$ and the multiplicand is $5 = 101_2$.

[27] Is this still taught in elementary schools?

```
       .text
######################## Begin multLoop function
### Multiply two whole numbers using repeated addition
### Pre: The values to be multiplied are in registers
###      $a0 and $a1
### Pre: $a0 is not negative
### Return product in $v0
multLoop:
       addi       $sp, $sp, -4
       sw       $ra, 0($sp)

       li       $v0, 0                    # Accumulator for product
lp_multLoop:

       ble       $a0, $0, done_multLoop
       add       $v0, $v0, $a1
       addi      $a0, $a0, -1             # Decrement loop counter
       j         lp_multLoop              # Repeat the loop
done_multLoop:
       lw        $ra, 0($sp)              # return to calling function
       addi      $sp, $sp, 4
       jr        $ra
########################## End function multLoop
```

Figure 3.64: Function to multiply two whole numbers, using repeated addition

Fig 3.65 in which the multiplier is $11 = 1011_2$ and the multiplicand is $5 = 101_2$, producing a product of $55 = 110111_2$. Note in Fig 3.65 that the multiplicand, 1101, is shown vertically at the right, to show which rows include the multiplier, 101_2, and which rows include 0. Each row is shifted one bit to the left (these are called *partial products*).

To generalize, we describe the shift-and-add algorithm below (the first operand is called the *multiplier* and the second operand is called the *multiplicand*):

1. Initialize an accumulator to zero.
2. If the multiplier is zero, terminate with the product in the accumulator.
3. If the multiplier is odd, add the multiplicand to the accumulator.
4. Shift the multiplicand 1 bit to the left.
5. Shift the multiplier 1 bit to the right.
6. Repeat from step 2 above.

Fig 3.66 depicts this algorithm for multiplication of 5 * 6. The algorithm is implemented in Fig 3.67.

3.12.2 Multiplication with a MIPS Instruction

Now that we have some understanding of the process of multiplication, we turn our attention to a MIPS instruction which can multiply 32-bit whole numbers. In the preceding section we made the tacet assumption that the product would fit in a register ($v0). Suppose we were to multiply two 4-bit numbers: $9 = 1001_2 \cdot 8 = 1000_2$ produces a product of $72 = 1001000_2$, which clearly does not fit in 4 bits. In general, when multiplying an n-bit number by an m-bit number, we should allow for the result to occupy m+n bits. Thus, if we are multiplying whole numbers in 32-bit general registers, we should allow for a 64-bit result.

The MIPS architecture accommodates this requirement by providing a pair of 32-bit registers with the names `Hi` and `Lo`. The `Hi` register is used to store the high order 32 bits of a product, and the `Lo` register is used to store the low order 32 bits of a product. These registers are separate from the general registers in the CPU and are normally used only for multiplication and division.

In addition to the multiply instruction itself, there are also instructions which provide access to the `Hi` and `Lo` registers.

The instruction mnemonic for multiply is `mult`, and it is an R format instruction which multiplies the whole numbers in two general registers (assumed to be twos complement[28]), placing the result in the `Hi,Lo` register pair, as shown in Fig 3.68. This figure also shows the format of the instructions which permit access to the `Hi,Lo` register pair - Move from Hi (`mfhi`) and Move from Lo (`mflo`).

[28] There is another instruction which presumes the values are unsigned - `multu`

```
1. Accumulator = 0
2. Multiplier is not 0
3. Multiplier is odd, Accumulator = Accumulator + Multiplicand
= 0 + 110 = 110
4. Shift Multiplicand left
Multiplicand = 1100
5. Shift Multiplier right
Multiplier = 10
2. Multiplier is not 0
3. Multiplier is even
4. Shift Multiplicand left
Multiplicand = 11000
5. Shift Multiplier right
Multiplier = 1
2. Multiplier is not 0
3. Multiplier is odd, Accumulator = Accumulator + Multiplicand
= 110 + 11000 = 11110
4. Shift Multiplicand left
Multiplicand = 11000
5. Shift Multiplier right
Multiplier = 0
6. Terminate, result is Accumulator = 11110 = 30
```

Figure 3.66: Multiplication of 5 * 6 using a shift-and-add algorithm. The Multiplier is $5 = 101_2$, and the Multiplicand is $6 = 110_2$.

```
######################## Begin mult function
### Multiply two whole numbers using
###   a shift-and-add algorithm
### Pre: The values to be multiplied are in registers
###      $a0 and $a1
### Pre: $a0 is not negative
### Return product in $v0
mult:
      addi      $sp, $sp, -4
      sw        $ra, 0($sp)

      li        $v0, 0                 # Accumulator for product
lp_mult:
      ble       $a0, $0, done_mult
      andi      $t0, $a0, 1            # Test low order bit
      beq       $t0, $0, even_mult
      add       $v0, $v0, $a1          # Add in multiplicand
even_mult:
      sll       $a1, $a1, 1            # Shift multiplicand
      srl       $a0, $a0, 1            # Shift multiplier
      j         lp_mult                # Repeat the loop

done_mult:
      lw        $ra, 0($sp)            # return to calling function
      addi      $sp, $sp, 4
      jr        $ra
########################## End function mult
```

Figure 3.67: Function to multiply two whole numbers using a sift-and-add algorithm.

(a)

```
[label:]    mult   $rs, $rt       [# comment]
[label:]    mfhi   $rd            [# comment]
[label:]    mflo   $rd            [# comment]
```

(b)

$\{Hi, Lo\} \leftarrow Reg[\$rs] \cdot Reg[\$rt]$
$Reg[\$rd] \leftarrow Hi$
$Reg[\$rd] \leftarrow Lo$

(c)

```
     mult    $t3, $a0
     mfhi    $v0
     mflo    $v1             # $v0,$v1 = $t3 * $a0
```

Figure 3.68: Multiply Statement: (a) Format of Multiply, Move from Hi, and Move from Lo (b) Meaning of Multiply, Move from Hi and Move from Lo (c) Example, which puts the product of registers $t3 and $a0 into the register pair $v0, $v1.

We now provide an example of a function which uses a multiply instruction. The problem is to find the volume of a cube, with a given edge length. The volume is simply edge * edge * edge, so two multiply instructions will be needed. The function is careful to stipulate a precondition in the API - the edge length cannot exceed $32,768 = 8000_{16}$. If the edge length exceeds this value, the result of the second multiplication will not fit in the `Hi,Lo` register pair. The function is shown in Fig 3.69.

3.12.3 Exercises

1. (a) Use pencil and paper to multiply in binary $10111_2 \cdot 110_2$.

   (b) Use pencil and paper to multiply in binary $1011_2 \cdot 11011_2$.

2. You wish to use one of the algorithms given to multiply 4023 (given in register $a0) times 201 (given in register $a1). How many times will the loop repeat if using:

   (a) The repeated addition algorithm given in Fig 3.64

   (b) The shift and add algorithm given in Fig 3.67

3. Show a trace (similar to Fig 3.46) of the mult function given in Fig 3.67 when the value in register $a0 is 25, and the value in register $a1 is 13.

```
       .text
#####################  Begin function volume
### Find the volume of a cube
### Edge length is in $a0
### Pre:  Edge length must not exceed 0x8000
###       (32,768)
volume:
       addi     $sp, $sp, -4
       sw       $ra, 0($sp)

       mult     $a0, $a0       # product should fit in lo
       mflo     $t0            # edge * edge
       mult     $a0, $t0       # edge * edge * edge
       mflo     $v1
       mfhi     $v0

       lw       $ra, 0($sp)
       addi     $sp, $sp, 4
       jr       $ra
######################  End function volume
```

Figure 3.69: Function to find the volume of a cube

4. Write a MIPS function named 'block' to find the volume and surface area of a right rectangular prism (i.e. a cube for which the angles are all 90, but the edges may have different lengths). The API is shown below:

   ```
   ################## Begin function block
   ### Find the volume and surface area of a right rectangular prism
   ### The three edge lengths, length, width, and height are given
   ###    in registers $a0, $a1, and $a2, respectively.
   ### The volume should be returned in register $v0.
   ### The surface area should be returned in register $v1
   ### Pre:  The results will be small enough to fit in a general
   ###       register.
             .text
   block:
   ```

   Use the MIPS mult instruction for multiplication.

5. Write a MIPS function named scalarMult to multiply a given array of whole numbers (i.e. a vector) by a given whole number (i.e. a scalar), leaving the result in a separate memory area. The API is shown below:

   ```
   ################ Begin function scalarMult
   ### Multiply a vector by a scalar.
   ### Register $a0 contains the scalar
   ### Register $a1 contains the address of the vector in memory
   ### Register $a2 contains the length of the given vector
   ### Register $a3 contains the address of the result in memory
             .text
   scalarMult:
   ```

   Use the MIPS mult instruction for multiplication.

6. Write a MIPS function named 'dot' to find the *dot product* of two vectors. The dot product is the vector which contains the products of corresponding values from the two given vectors. The API is shown below:

   ```
   ################ Begin function dot
   ### Multiply two vectors, forming the dot product
   ### Register $a0 contains the address of the first vector
   ### Register $a1 contains the address of the second vector
   ### Register $a2 contains the length of the two vectors
   ### Register $a3 contains the address of the result in memory
   ### Pre:  The two given vectors have the same length
             .text
   dot:
   ```

   Use the MIPS mult instruction for multiplication.

7. Write a MIPS function named ‘matrixMult’ to find the *matrix product* of two matrices. The matrix product of two matrices, A and B, can be calculated if the number of columns in A equals the number of rows in B (call that number `max`). The value at row r and column c of the result matrix is computed as

   $$\sum_{i=0}^{max} A[r,i] \cdot B[i,c]$$

   The API is shown below:

```
################ Begin function matrixMult
### Find the matrix product of two matrices
### Register $a0 contains the address of the first matrix, A,
###      stored in row-major order
### Register $a1 contains the address of the second matrix, B,
###      stored in row-major order
### Register $a2 contains the memory address of three memory
###      words storing the following:
###      - The number of columns in matrix A, which is equal to
###        the number of rows in matrix B
###      - The number of rows in matrix A
###      - The number of columnss in matrix B
###  - Address for result matrix.
### Rows = rows in A.
### Cols = cols in B.
          .text
matrixMult:
```

   Use the MIPS mult instruction for multiplication.
   Hint: Program this problem in a high level language, such as Java or C++, then translate your program to MIPS assembly language.

3.13 Division

In this section we consider the division of whole numbers. Division is interesting in that there are actually two results for division of integers - a quotient and a remainder, both of which are also whole numbers. For example 39 divided by 5 produces a quotient of 7 and a remainder of 4. The remainder is often called a ‘modulus’, or simply ‘mod’. Many high level languages provide an operator for mod (usually the % symbol). Thus in a language such as Java or C++, 39/5 is 7 but 39%5 is 4.[29]

[29] Be careful when using the % operator with negative numbers; high level languages do not agree on the results

```
       .text
######################## Begin divLoop function
### Perform division using repeated subtraction
### The dividend is in register $a0
### The divisor is in register $a1
### The quotient is left in $v0
### The remainder is left in $v1
### Pre:  The divisor is positive
### Pre   The dividend is not negative
### Post  Both $a0 and $a1 are unchanged
divLoop:
       addi       $sp, $sp, -4
       sw         $ra, 0($sp)

       li         $v0, 0                # Counter for quotient
       move       $v1, $a0              # Dividend
lp_divLoop:
       blt        $v1, $a1, done_divLoop
       sub        $v1, $v1, $a1
       addi       $v0, $v0, 1           # Decrement loop counter
       j          lp_divLoop            # Repeat the loop
done_divLoop:
       lw         $ra, 0($sp)           # return to calling function
       addi       $sp, $sp, 4
       jr         $ra
########################## End function divLoop
```

Figure 3.70: Function to divide whole numbers, using repeated subtraction

3.13.1 Division Implemented in Software

Just as multiplication can be implemented with repeated addition, division can be implemented with repeated subtraction. For example, to divide 39 by 7, we count the number of times that we can subtract 7 from 39, before it becomes negative; this is the quotient. The result of the final subtraction is the remainder. The two results can be produced with one loop. Fig 3.70 shows a MIPS function which performs the division operation using repeated subtraction.

There are better ways (i.e. faster ways) to divide whole numbers, Just as we used a shift and add algorithm for multiplication, we can use a shift and subtract algorithm for division, which is significantly faster.

Our division algorithm will treat registers $v1 (the remainder) and $a0 (the dividend) as a register pair. We will need to shift the register pair one bit to the left, which means that the high order bit of $a0 is to be shifted into the low order bit of $v1, as shown in Fig 3.71.

To implement this shift in a register pair, we can use three instructions (the

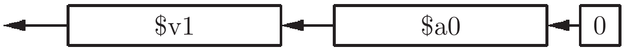


Figure 3.71: Diagram of a left shift on a pair of registers, $v1 and $a0

left register is $v1, and the right register is $a0):

1. Shift the left register left
   ```
   sll     $v1, $v1, 1
   ```

2. Determine whether the high order bit of the right register is a 1
   ```
   bge     $a0, $0, notNeg
   ```

3. If not, add 1 to the left register
   ```
   addi     $v1, $v1, 1
   ```

4. Shift the right register left
   ```
   notNeg:
   sll     $a0, $a0, 1
   ```

Our shift and subtract algorithm for division is shown below:

1. Iniitalize the quotient and remainder to 0

2. Shift the quotient left.

3. Treating the remainder (left) and the divisor (right) as a register pair, shift left.

4. If the remainder is greater than or equal to the divisor,

   (a) Subtract the divisor from the remainder, leaving the result in the remainder.

   (b) Increment the quotient

5. Repeat from 2, once for each bit in the word (i.e. 32 times).

This algorithm is implemented in Fig 3.72.

As an example, we show the division 27/5 for 8-bit registers in Fig 3.73. The resulting quotient is 5, and the remainder is 2. In this example the step number from the algorithm given above is shown at the right (the shifting of a zero quotient is not shown in this example).

```
       .text
######################## Begin div function
### Perform division using shift and suubtract algorithm
### The dividend is in register $a0
### The divisor is in register $a1
### The quotient is left in $v0
### The remainder is left in $v1
### Pre:  The divisor is positive
### Pre   The dividend is not negative
div:
       addi     $sp, $sp, -4
       sw       $ra, 0($sp)

       li       $v0, 0                   # quotient
       li       $v1, 0                   # remainder
       li       $t0, 32                  # loop counter
lp_div:
       beq      $t0, $0, done_div
       sll      $v0, $v0, 1              # shift quotient
       sll      $v1, $v1, 1              # shift remainder,dividend
       bge      $a0, $0, notNeg_div
       addi     $v1, $v1, 1
notNeg_div:
       sll      $a0, $a0, 1              # shift dividend
       blt      $v1, $a1, noSubtr_div    # subtract?
       sub      $v1, $v1, $a1
       addi     $v0, $v0, 1
noSubtr_div:
       addi     $t0, $t0, -1
       j        lp_div                   # Repeat the loop
done_div:
       lw       $ra, 0($sp)              # return to calling function
       addi     $sp, $sp, 4
       jr       $ra
########################## End function div
```

Figure 3.72: Function to divide whole numbers, using shift and subtract algorithm

```
quotient        remainder   dividend    divisor    step
--------        --------    --------    -------    ----
00000000        00000000    00011011    00000101
                                                   3 shift remainder,dividend
                00000000    00110110
                                                   3 shift remainder,dividend
                00000000    01101100
                                                   3 shift remainder,dividend
                00000000    01101100
                                                   3 shift remainder,dividend
                00000000    11011000
                                                   3 shift remainder,dividend
                00000001    10110000
                                                   3 shift remainder,dividend
                00000001    10110000
                                                   3 shift remainder,dividend
                00000011    01100000
                                                   3 shift remainder,dividend
                00000110    11000000
                                                   4(a) subtract divisor from remainder
                00000001
                                                   4(b) increment quotient
00000001
                                                   2 shift quotient
00000010
                                                   3 shift remainder,dividend
                00000011    10000000
                                                   2 shift quotient
00000100
                                                   3 shift remainder,dividend
                00000111    00000000
                                                   4(a) subtract divisor from remainder
                00000010
                                                   4(b) increment quotient
00000101
```

Figure 3.73: Division 27/5, yielding a quotient of 5 and a remainder of 2, using 8-bit registers

(a)

```
[label:]    div    $rs, $rt       [# comment]
[label:]    mfhi   $rd            [# comment -  Remainder]
[label:]    mflo   $rd            [# comment -  Quotient]
```

(b)

$Hi \leftarrow Reg[\$rs] \% Reg[\$rt]$
$Lo \leftarrow Reg[\$rs] / Reg[\$rt]$
$Reg[\$rd] \leftarrow Hi$
$Reg[\$rd] \leftarrow Lo$

(c)

```
    div     $t3, $a0
    mfhi    $v0             # $v0 = $t3 % $a0
    mflo    $v1             # $v1 = $t3 / $a0
```

Figure 3.74: Divide Statement: (a) Format of Divide, Move from Hi, and Move from Lo (b) Meaning of Divide, Move from Hi and Move from Lo (c) Example, which divides using register $t3 as the dividend and register $a0 as the divisor, leaving the quotient in register $v1 and the remainder in register $v0.

3.13.2 Division with a MIPS Instruction

Now that we understand how division works, we will examine the MIPS instruction which divides whole numbers. Like the multiply instruction, it is an R format instruction. The dividend is the first operand, and the divisor is the second operand. The quotient is stored in the LO register, and the remainder is stored in the HI register (see the section on multiplication for an explanation of the LO and HI registers). The format, meaning, and example of the division instruction are shown in Fig 3.74.

This divide instruction assumes the operands are signed (i.e. twos complement representation). The quotient is negative if either operand is negative, and the quotient is positive when the operands are both positive or both negative.[30] There is also a divide instruction which assumes the operands and results are unsigned: (`divu`).

We conclude this section with an example of a function which makes use of division of whole numbers. In this example we wish to develop a function which will convert a whole number of seconds to an equivalent number of hours, minutes, and seconds. For example, if the input to the function is 4713 seconds, the result should be three numbers representing 1 hour, 18 minutes, 33 seconds.

Our function will accomplish this using a few divide instructions.

[30] The sign of the remainder is a bit more complicated, and is not standard across platforms, and over time; we will not address it here.

1. Divide the given number by 60. The remainder is the the number of seconds in the result, and the quotient will be the total minutes remaining. For example, 4713 % 60 is 33, so the number of seconds is 33. And 4713 / 60 is 78. Save this for the next step.

2. The quotient from the previous step is then divided by 60. The remainder is the number of minutes, and the quotient is the number of hours. For example, the number of minutes from the previous step is 78. Divide 78 / 60. The remainder, 18 is the number of minutes, and the quotient, 1, is the number of hours.

This algorithm is implemented in a MIPS function in Fig 3.75. The number of seconds is provided in register $a0. Register $a1 contains the memory address for three full words - the number of hours, minutes, and seconds, in that order.

3.13.3 Exercises

1. Using pencil and paper, perform the following division operations:

   (a) 29 / 3 = ? 29 % 3 = ?

   (b) 290 / 17 = ? 290 % 17 = ?

   (c) 4098 / 256 4096 % 256

2. You wish to use one of the algorithms given to divide 4023 by 21. How many times will the loop repeat if using:

   (a) The repeated subtraction algorithm given in Fig 3.70

   (b) The shift and subtract algorithm given in Fig 3.72

3. Show a trace (similar to Fig 3.46) of the div function given in Fig 3.72 when the dividend (in register $a0) is 4023 and the divisor (given in register $a1) is 210.

4. Write a function which will take a distance measurement, given in inches, and produce the same distance in yards, feet, and inches. For example, if the given distance is 86 inches, the result should be 2 yards, 1 foot, and 2 inches. The API is shown below:

```
### Author:  sdb

### Convert given number of inches to inches, feet, and yards
### Example:  122 inches =>  3 yards, 1 foot, 2 inches
### Register $a0 contains the given number of inches
### Register $a1 points to memory area for results:
###        Number of yards
###        Number of feet
###        Number of inches
```

```
       .text
######################## Begin hms function
### Convert a whole number of seconds to
### hours, minutes, and seconds.
###     3701 seconds =>  1 hour 1 minute, 41 seconds
### Total seconds is provided in $a0
### Register $a1 points to a memory area for
###         three results:  hours, minutes, seconds
### Pre:  All values are non-negative.
hms:
       addi     $sp, $sp, -4
       sw       $ra, 0($sp)

       li       $t0, 60
       div      $a0, $t0
       mfhi     $t1
       sw       $t1, 8($a1)             # seconds
       mflo     $a0                     # total minutes
       div      $a0, $t0
       mfhi     $t1
       sw       $t1, 4($a1)             # minutes
       mflo     $t1
       sw       $t1, 0($a1)             # hours

       lw       $ra, 0($sp)             # return to calling function
       addi     $sp, $sp, 4
       jr       $ra
########################## End function hms
```

Figure 3.75: Function to convert total number of seconds to hours, minutes, and seconds

5. Write a MIPS function named scalarDiv to divide a given array of whole numbers (i.e. a vector) by a given whole number (i.e. a scalar), leaving the quotients in a separate memory area. The API is shown below:

   ```
   ################ Begin function scalarDiv
   ### Divide a vector by a scalar.
   ### Register $a0 contains the scalar
   ### Register $a1 contains the address of the vector in memory
   ### Register $a2 contains the length of the given vector
   ### Register $a3 contains the address of the result in memory
   ```

   Use the MIPS div instruction for division. Disregard the remainders.

6. Write a MIPS function named ‘dotDiv’ to find the *dot quotient* of two vectors. The dot quotient is the vector which contains the quotients of corresponding elements. For example, if the two vectors are A = (5, 0, 7) and B = (3, 3, 2), then the quotient vector is (1, 0, 3). The API is shown below:

   ```
   ################ Begin function dotDiv
   ### Form the dot quotient of two vectors
   ### Register $a0 contains the address of the dividend vector
   ### Register $a1 contains the address of the divisor vector
   ### Register $a2 contains the length of the two vectors
   ### Register $a3 contains the address of the result in memory
   ### Pre:  The two given vectors have the same length
   ```

   Use the MIPS div instruction for division. Disregard the remainders.

3.14 Floating Point Instructions

Thus far we have dealt exclusively with whole numbers. The arithmetic instructions that we have used all assume that the operands are 32-bit integers, stored in general-purpose registers (these integers correspond to the `int` data type in Java).

MIPS also allows for computations using non-integers, as well as integers which are too big to fit in a 32-bit register. This data type is called *floating point*. MIPS allows for two floating point types, a 32-bit type and a 64-bit type, corresponding to the types `float` and `double` in Java. The 64-bit type, double precision, is exactly like the 32-bit type, single precision, except that it allows for greater accuracy and larger magnitudes. Hence, we will focus our attention on the 32-bit single precision format.

Some examples of floating point values are:

- 2.075

- -3.0
- $6.02 \cdot 10^{23}$
- 0.000001

To enter the third example as a MIPS data value, we would write it as `6.02e23`, which is the same way it would be written in a high-level programming language. Before we go on, we should emphasize that most floating point data values are merely approximations, i.e. they do not represent the intended value exactly.[31] We will explore this in more depth in chapter4.

3.14.1 Floating Point Registers

The MIPS architecture has an additional set of 32 registers, each of which is 32 bits, which are dedicated specifically for storing floating point values. These registers are located in a separate part of the MIPS architecture called `coprocessor 1`. The names of these registers are $f0, $f1, $f2, ... $f31. The double precision instructions make use of these registers as even-odd neighboring pairs to obtain 16 64-bit registers. For example, a double precision add instruction could add the 64-bit quantity in registers ($f6, $f7) to the 64-bit quantity in registers ($f2, $f3). But it could not refer to the register pair ($f9, $f10) since that is not an even-odd pair.

The single precision floating point instructions may refer to any of the 32 floating point registers. In our examples, however, we will refer to even numbered registers. This would allow for a fairly simple revision that would work with double precision instructions instead if more accuracy is desired.

3.14.2 Floating Point Instructions

In this section we expose some MIPS instructions which perform floating point calculations. These instructions are all similar in that they operate on two operand registers, placing the result in a target register (which could be the same as one of the operand registers).

Since we are dealing with single precision floating point here, the instructions will have a `.s` suffix. To use the corresponding double precision instruction, use a `.d` suffix. For example, to add the contents of two floating point registers we use the `add.s` instruction, but to add the contents of two double precision floating point register pairs we would use the `add.d` instruction.

The basic four floating point arithmetic instructions are shown in Fig 3.76.

For all four of these instructions, the first operand is the target register, i.e. the register which is to receive the result of the calculation. For subtraction and division, be careful to place the operands in the correct order, since these operations are not commutative. A few comments on these floating point instructions:

[31] To see this, print the sum 0.1 + 0.1 + 0.1 in a Java program.

(a)

```
[label:]    add.s   $rt, $rs, $rt     [# comment]
[label:]    sub.s   $rt, $rs, $rt     [# comment]
[label:]    mul.s   $rt, $rs, $rt     [# comment]
[label:]    div.s   $rt, $rs, $rt     [# comment]
```

(b)

$Reg[\$rd] \leftarrow Reg[\$rs] + Reg[\$rt]$
$Reg[\$rd] \leftarrow Reg[\$rs] - Reg[\$rt]$
$Reg[\$rd] \leftarrow Reg[\$rs] \cdot Reg[\$rt]$
$Reg[\$rd] \leftarrow Reg[\$rs] / Reg[\$rt]$

(c)

```
     add.s   $f2, $f3, $f6
```

(d)

```
     sub.s   $f2, $f2, $f6
```

(e)

```
     mul.s   $f2, $f2, $f2
```

(f)

```
     div.s   $f22, $f22, $f0
```

Figure 3.76: Single Precision Floating Point Instructions: (a) Format of arithmetic instructions; (b) Meaning of each instruction from part (a); (c) Example, which adds the values in floating point registers $f3 and $f6, plaicng the result in floating point register $f2; (d) Example which subtracts the value in floating point register $f6 from register $f2; (e) Example, which squares floating point register $f2; (f) Example, which divides the value in floating point register $f22 by the value in floating point register $f0, leaving the result in floating point register $f22

(a)

```
[label:]    mov.s   $rd, $rs          [# comment]
```

(b)

$Reg[\$rd] \leftarrow Reg[\$rs]$

(c)

```
     mov.s   $f2, $f4
```

Figure 3.77: Floating Point Move Instruction: (a) Format of move instruction; (b) Meaning of move instruction; (c) Example, which copies the floating point value from floating point register $f4 into floating point register $f2.

- There is no floating point register which always contains the value zero, as there is with the general registers.
- The programmer may wish to put the value 0.0 into a floating point register, by subtracting a register from itself:
  ```
  sub.s       $f2, $f4, $f4       # Put zero into reg $f2
  ```
  However, because of the inexact nature of floating point representations, this should be avoided. Instead load a zero constant from memory.
- The programmer may wish to put the value 1.0 into a floating point register, by dividing a register by itself:
  ```
  div.s       $f2, $f4, $f4       # Put 1.0 into reg $f2
  ```
  However, because of the inexact nature of floating point representations, this should be avoided. Instead load a 1.0 constant from memory.
- For division there is only one result, the quotient (unlike the fixed point division instruction which produces two results).

To transfer the contents of one floating point register into another floating point register, there is a floating point *move* instruction. It is `mov.s` (or `mov.d` for double precision). The format and definition of a floating point move instruction is shown in Fig 3.77.

3.14.3 Floating Point Data in Memory

We have seen earlier how to initialize memory with data values, using the .data assembler directive. There are directives which provide the capability of initializing memory with floating point values. The `.float` directive initializes a full word of memory to a particular single precision value, and the `.double` directive initializes two consecutive full words of memory to a particular double precision value:

```
pi:          .float      3.141592653
pi:          .double    3.14159265358979324
```

An array (i.e. a vector of single precision floating point values) can be initialized:

```
numbers:     .float       2.3, 0.00001, 45, 6.02e23
```

The last value in the array named `numbers` is Avogadro's number, which is $6.02 \cdot 10^{23}$.

3.14.4 Loading and Storing Floating Point Registers

As with the general registers, we also have the capability of loading a floating point register from memory, and storing a value from a floating point register into memory. These instructions are called `lwc1` and `swc1`, respectively[32]. These instructions are both I format instructions (for the same reason that the lw and sw instructions are I format - they reference memory locations). Fig 3.78 shows the format and meaning of these instructions, as well as a few examples.

Fig 3.79 shows an example of a function which will compute the area and circumference of a circle. This function uses what we call *local data*, i.e. it has its own data section which stores an approximation to the value of π. Thus this function can be pasted into any program where it may be needed and called usign the instruction: `jal   circle`.

3.14.5 Floating Point Comparisons

In the section on transfer of control we discussed conditional branches. We will certainly have a need for conditional branches when using floating point arithmetic. Hence, we will need to be able to compare the values in floating point registers, and we will need to be able to branch conditionally on the result of the comparison.

For floating point comparisons this will be a two-step process:

1. Compare floating point registers, specifying the desired comparison (equality, less, less or equal). This step will set a 1-bit *condition code* to 1 or 0, depending on whether the desired comparison is true or false, respectively.

2. Branch to another instruction. This step will use the condition code, set in the previous step, to determine whether the branch should take place.

The comparison instructions are `c.eq.s`, for *compare floats for equality*, `c.lt.s` for *compare floats for strictly less than*, and `c.le.s` for *compare floats for less than or equal*. These instructions are defined in Fig 3.80.

Note the `.s` suffix indicates that these comparison instructions operate on single precision values. For double precision comparisons the suffix would be `.d`, and the 64-bit values being compared would be in even-odd register pairs.

The branch instructions are `bc1t`, for *branch on floating point true* and `bc1f`, for *branch on floating point false*.[33] These instructions are defined in

[32] The `c1` stands for *coprocessor 1*

[33] As previously noted, the `c1` is for coprocessor 1.

(a)

```
[label:]    lwc1    $rt, symbol       [# comment]   symbolic address
[label:]    lwc1    $rt, imm($rs)     [# comment]   explicit address
[label:]    swc1    $rt, symbol       [# comment]   symbolic address
[label:]    swc1    $rt, imm($rs)     [# comment]   explicit address
```

(b)

$Reg[\$rt] \leftarrow Memory[symbol]$
$Reg[\$rt] \leftarrow Memory[Imm + Reg[\$rs]]$
$Memory[symbol] \leftarrow Reg[\$rt]$
$Memory[Reg[\$rs]] \leftarrow Reg[\$rt]$

(c)

```
     lwc1    $f0, pi
```

(d)

```
     lwc1    $f4, 0($t4)
```

(e)

```
     swc1    $f4, result
```

(f)

```
     swc1    $f2, 16($s4)
```

Figure 3.78: Single Precision Floating Point Memory Reference Instructions: (a) Format of single precision load and store instructions, using symbolic and explicit memory addresses; (b) Meaning of each instruction from part (a); (c) Example which loads floating point register $f0 with the value of pi, from memory; (d) Example which loads floating point register $f4 from the memory location 1 word beyond the address in general register $t4; (e) Example, which stores floating point register $f4 into the memory location named `result`; (f) Example which stores floating point register $f2 into the memory location which is 4 words beyond the address in general register $s4

```
              .text
##################  Begin circle function ##########
#     Author sdb
#     Find the area and circumference of a circle having
#          a given radius.
#     Register $a0 contains the memory address of the circle’s
#          radius (single precision)
#     Register $a1 contains the memory address for the two results:
#          - area
#          - circumference
circle:
              addi       $sp, $sp, -4
              sw         $ra, 0($sp)

              lwc1       $f0, pi_circle      # PI
              lwc1       $f1, 0($a0)         # radius
              mul.s      $f2, $f0, $f1       # PI * r
              mul.s      $f3, $f2, $f1       # PI * r * r = area
              swc1       $f3, 0($a1)         # Store area in memory

              add.s      $f4, $f2, $f2       # PI * r + PI * r = 2*PI*r
                                             # = circumference
              swc1       $f4, 4($a1)         # store circumference in memory

              lw         $ra, 0($sp)
              addi       $sp, $sp, 4
              jr         $ra                 # return

              .data                          # data local to this function
pi_circle:    .float       3.1415926
######################  End circle function ##########
```

Figure 3.79: Function to compute the area and circumference of a circle having a given radius

(a)

```
[label:]    c.eq.s  $fs, $ft       [# comment]
[label:]    c.lt.s  $fs, $ft       [# comment]
[label:]    c.le.s  $fs, $ft       [# comment]
```

(b)

if $\$fs == \$ft$, $cc \leftarrow 1$ else $cc \leftarrow 0$
if $\$fs < \$ft$, $cc \leftarrow 1$ else $cc \leftarrow 0$
if $\$fs <= \$ft$, $cc \leftarrow 1$ else $cc \leftarrow 0$

(c)

```
     c.eq.s  $f4, $f6
```

(d)

```
     c.lt.s  $f14, $12
```

(e)

```
     c.le.s  $f22, $16
```

Figure 3.80: Single Precision Floating Point Comparison Instructions: (a) Format of comparison instructions; (b) Meaning of each instruction from part (a) in which cc is the 1-bit condition code; (c) Example which compares floating point registers $f4 and $f6 for equality; (d) Example which determines whether floating point register $f14 is strictly less than floating point register $f12; (e) Example which determines whether floating point register $f22 is less than or equal to floating point register $f16

(a)

```
[label:]    bc1t    symbol          [# comment]
[label:]    bc1f    symbol          [# comment]
```

(b)

if cc == true, branch to instruction of symbol
if cc == false, branch to instruction of symbol

(c)

```
     bc1t    lp
```

(d)

```
     bc1f    done
```

(e)

```
     c.le.s  $f22, $16
```

Figure 3.81: Single Precision Floating Point Branch Instructions: (a) Format of conditional branch instructions; (b) Meaning of each branch instruction from part (a) in which cc is the 1-bit condition code; (c) Example which branchs to `lp` only if the condition code is 1 (d) Example which branchs to `done` only if the condition code is 0

Fig 3.81.

For example, to branch to the instruction labeled `lp` only if the value in floating point register $f4 is smaller than the value in floating point register $f2, we would use the instructions

```
      c.lt.s    $f4, $f2       # set condition code
      bc1t      lp             # branch if condition code is true
```

To demonstrate the usage of these conditional branch instructions, we show an example of a function which will find the largest of three given floating point values in Fig 3.82.

At this point we have shown only three types of comparison: compare for equality, compare for strictly less, and compare for less or equal. The student may be wondering about other kinds of comparisons which may be useful. They can all be implemented using the three existing comparisons. They are described below:

1. Compare for strictly greater than

   Here we can use the logical identity: $x > y \equiv y < x$

```
       .text
#####################  Begin function largestOf3
### Author:  sdb
### Find the largest of three given floats
### Register $a0 points to three consecutive floats in memory
### Register $a1 points to memory word where largest is to be stored
largestOf3:
       addi     $sp, $sp, -4
       sw       $ra, 0($sp)

       lwc1     $f0, 0($a0)       #  first float
       lwc1     $f2, 4($a0)       #  second float
       c.le.s   $f2, $f0
       bc1t     float2Larger_largestOf3
       lwc1     $f0, 4($a0)
float2Larger_largestOf3:          # f0 is largest of first two floats
       lwc1     $f2, 8($a0)       #  third float
       c.le.s   $f2, $f0          #  compare with larger of first two
       bc1t     float3Larger_largestOf3
       lwc1     $f0, 8($a0)
float3Larger_largestOf3:
       swc1     $f0, 0($a1)       # store largest

       lw       $ra, 0($sp)
       addi     $sp, $sp, 4
       jr       $ra
######################  End function largestOf3
```

Figure 3.82: Function to find the largest of three given floating point values

For example, to branch to `lp` only if $f2 is strictly greater than $f4, we need only to reverse the operands, and test for stricly less than:

```
c.lt.s   $f4, $f2      # Is $f2 > $f4 ?
bc1t     lp            # branch if true
```

2. Compare for greater than or equal to

   Here we can use the logical identity: $x \geq y \equiv\ !x < y$

   For example, to branch to `done` only if $f2 is greater than or equal to $f4, we need only to test for strictly less than (branch if false):

   ```
   c.lt.s   $f2, $f4      # Is $f2 >= $f4 ?
   bc1f     done          # branch if false
   ```

3. Compare for not equal to

   Here we can use the logical identity: $x \neq y \equiv\ !x = y$

   For example, to branch to `next` only if $f2 is not equal to $f4, we need only to test for equality, and branch if false:

   ```
   c.eq.s   $f2, $f4      # Is $f2 != $f4 ?
   bc1f     next          # branch if false
   ```

As one final example for this section, we show a function which searches an array of floats for a particular target, with a given error tolerance. The reason for the error tolerance is that floats do not have perfect precision. Thus, for example, if searching for the target 17.01, we may wish to specify a tolerance of 0.000001 so that any value in the array which is sufficiently close to 17.01 qualifies as matching the target. Figures 3.83 and Figures 3.84 shows this *sequential search* function.

In order to determine whether a value from the array is sufficiently close to the target, we compute the absolute value of the difference between the value and the target, and compare with the tolerance, epsilon.

$|value - target| < epsilon$

For absolute value we use a local function which puts the absolute value of floating point register $f6 into floating point register $f6.

3.14.6 Exercises

1. Test the function which finds the area and circumference of a circle (Fig 3.79) using the MARS simulator. Write a Driver (i.e. a main program which calls the `circle` function), and check the results in memory.

2. Test the function which finds the largest of three floats (Fig 3.82) using the MARS simulator. Write a Driver (i.e. a main program which calls the `largestOf3` function), and check the result in memory.

```
#####################  Begin function seqSearch
### Author:       sdb
### Search a vector of floats for the first occurrence of a given
###        target, to within a given tolerance.
### $a0 contains the memory address of the vector of floats to be searched
### $a1 contains the length of the vector
### $a2 points to the target in memory
### $a3 points to the tolerance for comparisons
### Return the position of the target in $v0,
###       or -1 if not found.
       .text
seqSearch:
       addi     $sp, $sp, -4
       sw       $ra, 0($sp)

       lwc1     $f0, 0($a3)              # tolerance
       lwc1     $f2, target              # target
       li       $v0, -1                  # code for not found
       li       $t0, 0                   # loop counter
lp_seqSearch:
       beq      $t0, $a1, done_seqSearch
       lwc1     $f4, 0($a0)              # value from vector
       sub.s    $f6, $f2, $f4            # target - value
       jal      abs_seqSearch            # $f6 is absolute value of difference
       c.lt.s   $f6, $f0
       bc1t     hit_seqSearch
       addi     $a0, $a0, 4              # next word of vector
       addi     $t0, $t0, 1              # decrenent loop counter
       j        lp_seqSearch
hit_seqSearch:
       move     $v0, $t0
done_seqSearch:
       lw       $ra, 0($sp)
       addi     $sp, $sp, 4
       jr       $ra
######################  End function seqSearch
```

Figure 3.83: Function to search a given array of floats for a given target, to within a given tolerance. It calls an absolute value function - Fig 3.84

```
######################  Begin local function abs_seqSearch
### Author:   sdb
### Return absolute value of $f6 in $f6
### Post:  Clobbers $f8
       .text
abs_seqSearch:
       addi     $sp, $sp, -4
       sw       $ra, 0($sp)

       lwc1     $f8, zero_abs_seqSearch   #  0.0
       c.lt.s   $f8, $f6                  # f6 is positive?
       bc1t     done_abs_seqSearch
       sub.s    $f6, $f8, $f6             # if not, f6 = 0 - f6
done_abs_seqSearch:
       lw       $ra, 0($sp)
       addi     $sp, $sp, 4
       jr       $ra
       .data
zero_abs_seqSearch:       .float       0.0
######################  End function abs_seqSearch
```

Figure 3.84: Function to find the absolute value of a float, called by the sequential search function in Fig 3.83

3. Test the sequential search function (Fig **??**) using the MARS simulator. Write a Driver (i.e. a main program which calls the `seqSearch` function), and check the result in memory. Be sure to test the case where the target is not found in the array, and the case where there is more than one occurrence of the target in the array.

4. Show a trace (similar to Fig 3.52) of the largestOf3 function when the three floats are 2.0, 5.5, and -9.9. Show the contents of floating point registers in decimal. Do not show memory contents (we have not yet discussed how floating point values are represented).

5. Write and test a function which will return the volume and surface area of a sphere having a given radius. Store an approximaition of pi as local data in your function. The API is shown below:

   ```
   #####################  Begin function sphere
   ### Author:  sdb
   ### Find the volume and surface area of a sphere
   ### Register $a0 points to radius, in memory
   ### Register #a1 points to the volume and surface area in
   ###    consecutive memory locations.
   ###
   ### Volume = 4/3 pi r*r*r
   ### Area   = 4 pi r*r
   ```

6. Write and test a function named `maxFloat` which will return the largest value in an array of floats. Assume the array is not empty, i.e. the length is at least 1. Be sure to test the case where all the values are negative. The API is shown below.

   ```
   #####################  Begin function maxFloat
   ### Author: sdb
   ### Find the largest value in an array of floats
   ### $a0 contains the memory address of the vector of floats to be searched
   ### $a1 contains the length of the vector
   ### $a2 points to the memory location for the result
   ### Pre:  The array is not empty.
   ```

   Hint: Assume the first value is the largest; then use a loop to scan the rest of the array. Each time you find a value larger than the largest you've seen so far, save it in a floating point register.

7. The exponential function, $exp(x) = e^x$ where e is approximately 2.7181818284590 is the inverse of the natural log function, $ln(x)$. This function can be computed as an infinite sum of terms (this is called a Taylor series):
   $exp(x) = 1 + x^2/2! + x^3/3! + x^4/4! + ...$
   Define a MIPS function named `exp` to evaluate the exponential function

for a given value of x, and a given tolerance value, epsilon, such that the result is within epsilon of the correct result. The API is shown below:

```
###################### Begin  function exp ##############
# Author:  sdb
# $a0 contains address of x in memory (float)
# Desired precision is in the next word (epsilon)
# $a1 contains the address of the result
# Pre:  x is not negative
```

Hints:

- Each term, t, can be calculated from the previous term by multiplying by x and dividing by a counter, n:
  $t = t * x/n$
- Terminate the loop when a term's value is smaller than epsilon.

8. Write and test a *binary search* function to search a sorted array of floats for a given target, within a given tolerance.
The API is shown below:

```
#####################  Begin function binSearch
### Author:       sdb
### Search a sorted array of floats for a given target, within
###        a given tolerance
### $a0 contains the memory address of the vector of floats to be searched
### $a1 contains the length of the vector
### $a2 points to the target in memory
### $a3 points to the tolerance in memory
### Post: $v0 contains the position of the target, or -1 if not found
### Pre:  The array is sorted in ascending order.
```

Hint: Find the midpoint of the array, and compare it with the target. If equal (within the tolerance), terminate. If the value of the midpoint is less than the target, you know the target must be *after* the midpoint if it is in the array; repeat using the position after the midpoint as the left end. If the value of the midpoint is greater than the target, you know the target must be *before* the midpoint if it is in the array; repeat using the position before the midpoint as the right end. If the position of the left end exceeds the position of the right end, the target is not found.

3.15 Input, Output, and Other System Calls With MARS

The MARS assembler/simulator provides for input (reading information from an external source, such as the keyboard), output (writing information to an

\$v0	function	Args / Resdult
1	print int	\$a0 = integer to be printed
2	print float	\$f12 = float to be printed
4	print string	\$a0 = address of string
5	read int	\$v0 gets input from stdin
6	read float	\$f0 gets input from stdin
8	read string	\$a0 = address of string, \$a1 = max length
10	terminate	

Figure 3.85: Various uses of the syscall instruction

external destination, such as the monitor), and other useful system calls. We emphasize that these system calls are specific to MARS and may not apply with other MIPS simulators (such as SPIM).[34]

All MARS system calls are invoked with the `syscall` instruction. The particular function to be performed is specified by the value in register \$v0. Fig 3.85 shows some of the options available for a system call. For a complete list of options, see the MARS web site.

3.15.1 Normal Program Termination

To terminate a program which has completed execution successfully, we need only load the value 10 into register \$v0 before issuing the `syscall` instruction, as shown below:

```
li       $v0, 10
syscall              # normal termination
```

This means that if register \$v0 contains the final result of a calculation, it must be copied to another register, or saved in memory, before terminating (the return code of 10 will clobber the result in register \$v0).[35]

As an example, we show below a main function which calls the string comparison function shown in Fig 3.63. The main function stores the result in memory before terminating execution.

```
###  Test the string comparison function
main:
        la      $a0, str1
        la      $a1, str2
        jal     strcmp        # call function
        sw      $v0, result   # save function result
        li      $v0, 10       # code for normal termination
        syscall               # terminate program
```

[34] This section may be omitted without loss of continuity

[35] This seems to contradict the MIPS register convention that register \$v0 should be used to return function results.

```
        .data
str1:   .asciiz    "Good Morning"
str2:   .asciiz    "Good morning"
```

3.15.2 Input with syscall

With the syscall instruction it is possible to read an int, a float, or a string from the standard input stream (i.e. the keyboard).

To read an int from the keyboard, load the value 5 into register $v0 before executing the syscall instruction. Execution will then pause, waiting for the user to enter a whole number on the keyboard and press the Enter key. The whole number entered will be placed in register $v0, as shown below:

```
         li     $v0, 5      # code for input of an int
         syscall
                            # entered value is now in $v0
```

To read a string load the value 8 into register $v0. Also load a the address of a memory buffer for the input string into register $a0, and load a maximum length for the input into register $a1 before executing the syscall instruction. Execution will then pause, waiting for the user to enter any string of characters on the keyboard and press the Enter key. The string entered will be stored in the memory buffer, as shown below:

```
         la     $a0, buffer      # storage space for the string
         li     $a1, 1000        # max length of string
         li     $v0, 8           # code for input of a string
         syscall
                                 # string is now in the buffer
         .data
buffer:  .space 1001             # reserve 1001 bytes of memory
```

The .space directive reserves 1001 bytes (one extra byte for the null terminating character) of memory for the buffer.

To read a floating point value from the keyboard, load the value 6 into register $v0 before executing the syscall instruction. The value entered at the keyboard will stored in floating point register $f0:

```
        .li     $v0, 6          # code for input of a float
        syscall
                                # entered float is now in $f0
```

3.15.3 Output with syscall

With the syscall instruction it is possible to put out an int, a float, or a string to the standard output stream (i.e. the monitor) as indicated in Fig 3.85.

To display an int on the monitor, load the value 1 into register $v0, and load the value to be displayed into register $a0, before executing the `syscall` instruction, as shown below:

```
# we wish to display the int in register $t3
move     $a0, $t3      # move into register $a0
li       $v0, 1        # code to put out an int
syscall
                       # value of $t3 is displayed on monitor
```

To display a floating point value on the monitor, load the value 2 into register $v0, and load the value to be displayed into register $f12, before executing the `syscall` instruction, as shown below:

```
# we wish to display the float in register $f0
move     $f12, $f0     # move into float register $f12
li       $v0, 1        # code to put out a float
syscall
                       # value of $f0 is displayed on monitor
```

To display a (null terminted) string on the monitor, load the address of the string into register $a0, and load the value 4 into register $v0. Then when the `instruction` is executed, the string will be displayed on the monitor (in MARS' message window):

```
    # we wish to display the message
    la     $a0, message     # address of message to be displayed
    li     $v0, 4           # code to put out a string
    syscall                 # message is written out

          .data
message:  .asciiz   "Good morning"
```

3.15.4 Example for Input and Output

We conclude this section with an example of a program that does several kinds of input and output. This program will prompt the user for several strings and then display the average length of those strings.

The program is shown in Fig 3.86 and needs a few remarks:.

- The user MUST be prompted for input. Otherwise, when an input syscall is executed, the program will pause, and the user will not know that he/she is expected to type something on the keyboard.
- This program uses the `strlen` function from Fig 3.61 to find the length of a string.
- Computation of the sum of the string lengths, and the loop counter are done with ints (i.e. general registers). Avoid floating point arithmetic if possible.

- To convert the sum of the lengths (register $s1) to floating point, we use the `cvt.s.w` instruction. It expects a *fixed point* argument in its second operand, and produces the corresponding floating point value in its first operand.
- Likewise for the number of strings (register $s2).

We conclude this section by emphasizing that this is *not* a typical assembly language program. Programs designed to be used directly by the end user are typically coded in a high level language, such as Java or C++. Even for applications which are CPU intensive, today's compilers are so advanced that there is not a signicant loss in efficiency. Assembly language, if used at all, is generally used for low level operating system functions, device drivers, and embedded systems.

3.15.5 Exercises

1. Write a program to calculate 10! and display the result on the monitor (i.e. MARS' message window).
2. Write a program to obtain the radius of a circle from the user's keyboard. It should then display the area and circumference of the circle on the monitor (i.e. MARS' message window).
3. Write a program to input a string from the keyboard, eliminate all spaces from the string, and write the resulting string out to the monitor (MARS' message window).
4. Write a program to write 10 random ints in the range [0..1000] to the monitor. Use a code of 42 in register $v0, and specify the upper bound on the range with register $a1, with a `syscall` instruction. Register $a0 is a *seed*, or starting point, for the random number generator.
5. Write a program to play military 'taps' using the MIDI interface. See the `syscall` description on the MARS web site.

```
# Program to display the average length of several strings
# entered on the keyboard
      .text
avgLength:
       la       $a0, prompt1_avgLength       # Prompt for
       li       $v0, 4                       # number of strings.
       syscall

       li       $v0, 5                        # Read an int,
       syscall                                # number of strings.
       ble      $v0, $0, abend_avgLength      # Must be positive

       move     $s0, $v0                      # loop counter
       move     $s2, $v0                      # save count
       li       $s1, 0                        # sum
lp_avgLength:
       ble      $s0, $0, done_avgLength
       la       $a0, prompt2_avgLength       # Prompt for a
       li       $v0, 4                       # string.
       syscall
       la       $a0, buffer_avgLength
       li       $a1, 1000                    # max length
       li       $v0, 8                       # code to read a string
       syscall
       jal      strlen                       # find its length
       addi     $v0, $v0, -1                 # exclude newline
       add      $s1, $s1, $v0                # accumulate sum
       addi     $s0, $s0, -1
       j        lp_avgLength
done_avgLength:
       sw       $s1, word_avgLength
       lwc1     $f4, word_avgLength
       cvt.s.w  $f0, $f4                     # sum to floating pt
       sw       $s2, word_avgLength
       lwc1     $f4, word_avgLength
       cvt.s.w  $f2, $f4                     # count to floating pt
       div.s    $f12, $f0, $f2               # find average
       la       $a0, outMsg_avgLength
       li       $v0, 4
       syscall
       li       $v0, 2                       # code to print float
       syscall
abend_avgLength:
       li       $v0, 10                      # code to terminate
       syscall
                      .data
prompt1_avgLength:    .asciiz    "How many strings do you wish to enter? "
prompt2_avgLength:    .asciiz    "Enter a string\n"
outMsg_avgLength:     .asciiz    "The average length is: "
word_avgLength:       .float     0
buffer_avgLength:     .space     1002
```

Figure 3.86: Program to display the average length of several strings entered at the keyboard

Chapter 4

Machine Language for MIPS

In chapter 3 we described the fundamental MIPS instruction set at the assembly language level. Programs written in this form are not directly executable by the CPU. They must first be translated to *machine language*. In machine language:

- There is a binary operation code for each MIPS instruction.
- An instruction is coded into a single 32-bit word, in one of three different formats.
- Symbolic addresses are replaced by binary memory addresses.
- Some assembly language instructions, known as pseudo-operations, must be converted to one or more MIPS instructions.

This translation is done by software known as an *assembler*.

In this chapter we will describe the details of machine language and see how to translate, manually, from assembly language to machine language.

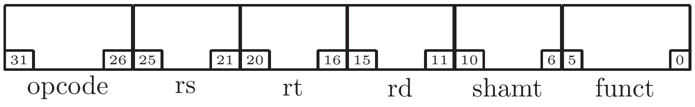
4.1 Instruction Formats

In the MIPS architecture there are three instruction formats: Register Format (R), Immediate Format (I), and Jump Format (J).

R format instructions are generally used for operations with two operand registers and a target register, such as the `add` and `or` instructions. R format is also used for *shift* instructions. The R format is described in Fig 4.1. The function code is actually part of the operation code. For example, both the `add` and `or` instructions have an opcode of 0. These instructions are distinguished by the function code, 20_x for `add` and 25_x for `or`.

I format instructions are generally used for instructions which have an *immediate* operand (i.e. a constant), such as `addi` or `andi`. I format can also

R Format



field name	bit positions	purpose
opcode	31..26	operation code
rs	25..21	left operand
rt	20..16	right operand
rd	15..11	destination
shamt	10..6	shift amount
funct	5..0	function code

Figure 4.1: Register Format (R) is used for instructions such as `add`, `or`, and `srl`. The diagram shows the bit positions for each field. A description of each field is shown in the table.

be used for memory reference instructions and conditional branch instructions. The I format is described in Fig 4.2. Notice that instead of the rd, shift amount, and function code fields we have an *immediate* field. The immediate field is used for:

- The right operand for arithmetic and logical operations for which the right operand is a constant.
- Memory address displacement for memory reference instructions.
- Relative branch address for conditional branch instructions.

We will elaborate further on this format in the sections which follow.

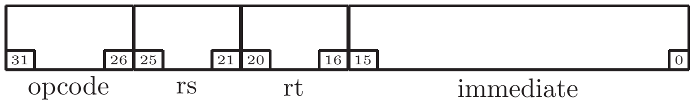
J format instructions are used for `j` (jump) and `jal` (jump and link) instructions. This is the simplest instruction format - it consists of only two fields, the operation code and the jump address. The format is shown in Fig 4.3.

4.1.1 Introduction to the Instruction Formats

It is important to understand the relationship between the size of a field in an instruction, and the number of different values which can be stored in that field. In a 1-bit field there can be only two values: 0 or 1. In a 2-bit field there can be four values: 0, 1, 2, or 3. In general an n-bit field can store 2^n different values.

The opcode field in the instruction formats is always 6 bits in length. This means we can have $2^6 = 64$ different instructions. R format instructions allow

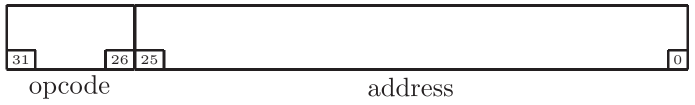
I Format



field name	bit positions	purpose
opcode	31..26	operation code
rs	25..21	left operand (or memory address)
rt	20..16	destination (or right operand)
immediate	15..0	right operand (or displacement)

Figure 4.2: Immediate Format (I) is used for immediate instructions such as `andi`, compare instructions such as `beq`, and memory reference instructions such as `lw`. The diagram shows the bit positions for each field. A description of each field is shown in the table.

J Format



field name	bit positions	purpose
opcode	31..26	operation code
address e	25..0	jump address

Figure 4.3: Jump Format (J) is used for unconditional jump instructions, such as `j` and `jal`. The diagram shows the bit positions for each field. A description of each field is shown in the table.

for an *expanded opcode* through the use of the function field, which is an additional 6 bits, allowing for an additional 64 instructions for each of the R format instructions.

The register fields, rs, rt, and rd are always 5 bits in length, allowing for $2^5 = 32$ different values; hence these fields can specify any one of the 32 registers, which are numbered 0 through 31.[1]

In designing an instruction set architecture, the widths of the fields in an instruction has an impact on the overal design of the CPU. For example, since the width of the register fields in a MIPS instruction is always 5 bits, the CPU cannot have more than 32 general registers.[2]

4.1.2 Exercises

1. Explain why the `shamt` field in an R format shift instruction is 5 bits in length.

2. A constant may be specified in the immediate field of an I format instruction. Assuming the constant represents an integer in twos complement representation,

   (a) What is the maximum value of the constant in an `addi` instruction?

   (b) What is the minimum value of the constant in an `addi` instruction?

3. You are given a MIPS instruction in binary:

   `001101 11001 00000 01010 00111 001111`

   (a) What is the value, in binary, of the opcode field?

   (b) Assuming this is an R format instruction,

      i. What is the value, in binary or decimal, of the `rs` field?

      ii. What is the name of the register specified in the `rd` field?

      iii. If this is a shift instruction, how many bit positions s are to be shifted?

   (c) Assuming this is an I format instruction,

      i. What is the name of the register specified by the `rt` field?

      ii. What is the value, in decimal or hexadecimal, of the immediate field?

      iii. Is the immediate field positive, negative, or zero?

   (d) Assuming this is a J format instruction, what is the value, in hexadecimal, of the `address` field?

4. A hypothetical architecture has:

[1] The correspondence between these register numbers and the register names was presented in Fig 3.1.

[2] Conversely, since the CPU has 32 general registers, the register fields in an instruction need not be more than 5 bits.

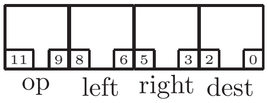


Figure 4.4: Instruction format for a hypothetical machine. The fields `left`, `right`, and `dest` specify registers.

- 64 general registers
- 16 different instructions
- Instructions which have two operands, both of which are registers.

Show a diagram, similar to Fig 4.1 of a possible instruction format for this machine.

5. An instruction format for a hypothetical machine is given in Fig 4.4.

   (a) How many different instructions (i.e. different opcodes) could this machine have?

   (b) How many general registers could this machine have? Assume the `left`, `right`, and `destination` fields specify register numbers.

4.2 Showing Binary Fields

In what follows we will often have a need to expose a machine language instruction in its raw form. While we could do this in binary (as in the exercises in the previous section) it may be more palatable to show the various fields of an instruction in hexadecimal.

This will be somewhat complicated by the fact that a hex digit represents 4 bits, but fields need not be a multiple of 4 bits. As we have seen, some fields are 5 bits and some are 6 bits in length. Hence we need to agree on how these are to be displayed in hexadecimal.

The convention which is normally followed for a field with a length not a multiple of 4, is to work from right to left (low order bit to high order bit), grouping the bits into groups of four. The remaining bits at the high order end can still be represented by a hex digit which may not correspond to four bits.

As an example we take binary value `01001011010`. In groups of four, we have `010 0101 1010` which is `25a` in hexadecimal. Note that the 2 represents only 3 bits, whereas each of the other hex digit represents 4 bits.

Another example: starting with the 9 bit value `111111111` we have
`1 1111  1111` which is `1ff` in hex.

We also will be interested in the opposite transformation: given an instruction in hexadecimal, find the values of the fields (in hex or decimal). For example, if we have the 32-bit instruction given in hexadeximal as `ba0af863` which can be written binary as:

`1011 1010 0000 1010 1111 1000 1100 0011`. If this is an R format instruction we can regroup the fields as:

```
101110 10000 01010 11111 00011 000011
opcode   rs    rt    rd  shamt  funct
```

We have decoded the instruction as:

- opcode $= 2e_x = 46$
- rs $= 10_x = 16$
- rt $= 0a_x = 10$
- rd $= 1f_x = 31$
- shamt $= 03_x = 3$
- funct $= 03_x = 3$

4.2.1 Exercises

1. Show each of the following binary fields in hexadecimal. Do not convert to decimal; rather, group the bits into groups of 4 bits.
   (a) 10111
   (b) 101010101010110
   (c) 110000110110
2. Show each of the following hex values, with an associated field size, in binary. Do not convert to decimal; rather, treat each hex digit as representing 4 or fewer bits.
   (a) 3fb Field size = 10
   (b) 3fb Field size = 11
   (c) 009c Field size = 13
3. Given the hexadecimal instruction, `4f3cffab`.
   (a) Assume this is an R format instruction.
      i. Show the value of the opcode field in hexadecimal.
      ii. Show the name of the register specified by the rs field.
      iii. Show the name of the register specified by the rt field.
      iv. Show the name of the register specified by the rd field.
      v. Show the shift amount in decimal.
      vi. Show the value of the function field in hexadecimal.
   (b) Assume this is an I format instruction.
      i. Show the value of the opcode field in hexadecimal.

ii. Show the name of the register specified by the rs field.

iii. Show the name of the register specified by the rt field.

iv. Show the value of the immediate field as four hexadecimal digits.

v. Show the value of the immediate field in decimal (It might be negative).

(c) Assume this is a J format instruction.

i. Show the value of the opcode field in hexadecimal.

ii. Show the value of the jump address in hexadecimal.

4.3 Pseudo Operations

In chapter 3 we discussed a few assembly language operations which do no correspond to any machine language instructions. These operations are called *pseudo-operations*, or pseudo-ops. When the assembler encounters a pseudo-op, it translates it to one or more actual machine instructions which serve an identical function. These pseudo-ops are provided simply for the convenience of the programmer, and are not essential to the MIPS architecture. We will discuss a few of them here, and others in a later section.

4.3.1 Load Immediate

The purpose of the `li` (load immediate) is to store a particular value into a register. However, there is no machine language instruction for `li`. Instead the assembler will substitue a different instruction which does the same thing, such as `addi` with a second operand of register \$0.[3]

For example, if you use the following instruction:

```
li    $t0, 3         # put 3 into $t0
```

the assembler will substitute:

```
addi  $t0, $0, 3
```

It has the same result - register \$t0 will be loaded with the value 3.

4.3.2 Move

The `move` operation in assembly language has no corresponding instruction in machine language; it is a pseudo-op. The assembler will replace each `move` operation with an equivalent `add` instruction, in which one of the operands is register \$0.[4]

For example, if you use the following instruction:

```
move    $t0, $t3        # move $t3 into $t0
```

the assembler will substitute:

[3] MARS actually uses `addiu` which we have not discussed; it avoids the possibility of an overflow exception.

[4] MARS actually uses `addu` which we have not discussed; it avoids the possibility of an overflow exception.

```
add  $t0, $0, $t3
```

It accomplishes the desired result - the value in register \$t3 will be stored into register \$t0.

As with all pseudo-ops, `move` was not included in the MIPS architecture because it is not essential - but the assembler provides it as a convenience for the programmer.

4.3.3 Not

The logical `not` operation introduced in chapter 3 is actually a pseudo-op. The assembler translates the `not` operation to a machine language `nor` instruction in which one of the operands is register \$0. The logical identity applied here is: $\sim x = \sim (x \vee 0)$.

For example, if the assembly language statement is:

```
not  $v0, $a0      # $v0 = complement of $a0
```

then the assembler will translate it to:

```
nor  $v0, $a0, $0
```

This accomplishes the desired result.

4.3.4 Load Address

The `la` (load address) operation discussed in chapter 3 is actually a pseudo-op. The purpose of the `la` operation is to load a memory address into a register. When the operand is given explicitly, the assembler could translate it to an `addi` instruction using the displacement as the immediate operand.[5]

For example, if the assembly language statement is:

```
la    $t0, 19($t1)    #  address in $t1 with offset 19
```

then the assembler could translate this to:

```
addi  $t0, $t1, 19
```

The `la` operation is used more typically with a symbolic memory address; this will be covered below in the section on symbolic memory references.

4.3.5 Other Pseudo Operations

MARS provides many more pseudo-ops, not discussed here. Many of them involve relaxed requirements on operands. For example, an `add` instruction can be given an immediate operand!

```
add    $t0, $t1, 100     # $t0 = $t1 + 100
```

The assembler translates it to:

```
addi   $t0, $t1, 100
```

Even more impressive, is the capability of using large constants in an immediate operation. As we have seen, the length of the immediate field is 16 bits, which would allow for a maximum positive value of $2^{15} - 1 = 32,767$. The MARS assembler will allow larger constant values, as long as they fit in 32

[5] MARS would actually translate this to an `ori` instruction followed by an `add` instruction.

bits. It will do this by translating an immediate operation to a `lui` (load upper immediate) operation followed by an `ori` (or immediate) operation followed by an `add` operation.

As an example, we choose the constant value 65539 which is 10003 in hexadecimal; it clearly will not fit in 16 bits. Nevertheless, MARS will permit it to be used as an immediate operand, as shown below:

```
addi   $t0, $t1, 65539      # $t0 = $t1 + 65,539
```

The MARS assembler will translate this statement to the following three statements:

```
lui    $at, 0x00000001
ori    $at, $at, 0x00000003
add    $t0, $t1, $at
```

We note the following about this translation:

- This example makes use of the $at (assembler temporary) register which is register 1. This register is normally reserved for use by the assembler. The programmer who attempts to use this register for temporary storage is asking for trouble.
- The three instructions which achieve the desired result are:
  1. The constant value, 0x00010003, is split into two halves. The high order half, 0x0001, is put into the high order half of register $at by the `lui` instruction.
  2. The low order half of the constant, 0x0003, is placed in the low order half of the $at register, without disturbing the high order half of that register. This is done with the `ori` instruction. The $at register is now storing the full constant, 0x00010003.
  3. The addition can now be done with an `add` instruction, using the $at register as an operand.

We have presented only some of the pseudo-ops provided by MARS. For a complete list see the `Help` menu provided with your MARS software.

4.3.6 Exercises

1. Show how each of the following assembly language statements is translated to equivalent machine language instruction(s).

   (a) `li    $v1, -23`

   (b) `move  $s3, $t8`

   (c) `not   $t1, $t0`

   (d) `la    $t4, 12($a0)`

   (e) `ori   $t1, $t1, 65537`

Instruction	opcode	function code	ignored fields	Example

add	00	20_x	shamt	`add $a1, $v1, $0`
				`$rs   $rt   $rd`
				`03    00    05`

sub	00	22_x	shamt	`sub $t0, $t1, $t2`
				`$rs   $rt   $rd`
				`09    0a    08`

Figure 4.5: Machine language for add and subtract instructions

2. The `move` pseudo-op is normally translated into an `add` instruction. Show another way to implement the `move` psuedo-op, without using any form of an `add` instruction. Use the example:
   ```
   move  $s3, $t8
   ```

4.4 R Format Instructions

In this section we expose several R format instructions in machine language. These instructions typically have two operand registers ($rs and $rt) and a destination register ($rd). The shift instructions are also included here.

4.4.1 Add and Subtract Instructions

The add and subtract instructions use the rs and rt fields to specify the left and right operands, respectively. The rd field is used to specify the destination register for the result. Both add and subtract make use of the expanded opcodes (i.e. the function code). They ignore the shift amount (shamt). Both of these instructions have an opcode of 0, but are distinguished by the function code: 20_x for add, and 22_x for subtract. These instructions are summarized in Fig 4.6.

Note that the order of the fields is different in machine language, as compared with assembly language. In assembly language we coded the destination register first, followed by the two operand registers.

```
add  $rd, $rs, $rt       # $rd = $rs + $rt
```

However, the $rd register is placed *last* in machine language, as shown in an earlier section.

Taking the two examples from Fig 4.6, we can now construct the 32-bit instruction, in three steps.

Starting with the first example:

```
add     $a1, $v1, $0
```

1. Show the fields in binary:

Instruction	opcode	function code	ignored fields	Example

add	00	20_x	shamt	`add $a1, $v1, $0`
				`$rs   $rt   $rd`
				`03    00    05`

sub	00	22_x	shamt	`sub $t0, $t1, $t2`
				`$rs   $rt   $rd`
				`09    0a    08`

Figure 4.6: Machine language for add and subtract instructions

000000	00011	00000	00101	?????	100000
31 26	25 21	20 16	15 11	10 6	5 0
opcode	rs	rt	rd	shamt	funct

Here we show the `shamt` field with question marks. The `add` instruction ignores this field, so it really doesn't matter what value is in that field. This is sometimes called a *don't-care* value.

2. Group the bits into groups of 4, as shown below.

```
0000 0000 0110 0000 0010 1??? ??10 0000
```

3. Convert to hexadecimal to get a more concise version of the machine language instruction. In doing so, we will assume all the don't-care values are zeros.

```
00 60 28 20
```

The second example from Fig 4.6 is:

```
sub     $t0, $t1, $t2
```

1. Show the fields in binary:

000000	01001	01010	01000	?????	100010
31 26	25 21	20 16	15 11	10 6	5 0
opcode	rs	rt	rd	shamt	funct

Again we show the `shamt` field as a don't care (i.e. question marks) because the `sub` instruction ignores this field.

2. Group the bits into groups of 4, as shown below.

```
0000 0001 0010 1010 0100 0??? ??10 0010
```

3. Convert to hexadecimal to get a more concise version of the machine language instruction. In doing so, we will assume all the don't-care values are zeros.

Instruction	opcode	function code	ignored fields	Example

and	00	24_x	shamt	`and $a1, $v1, $0`
				`$rs   $rt   $rd`
				`03    00    05`

or	00	25_x	shamt	`or $t0, $t1, $t2`
				`$rs   $rt   $rd`
				`09    0a    08`

xor	00	26_x	shamt	`xor $at, $ra, $a0`
				`$rs   $rt   $rd`
				`1f    10    01`

nor	00	27_x	shamt	`nor $t0, $t1, $t2`
				`$rs   $rt   $rd`
				`09    0a    08`

Figure 4.7: Machine language for the instructions `and, or, xor, nor`

```
01 2a 40 22
```

The student should verify both of these results by assembling the source statements with MARS to view the corresponding machine language instructions in hexadecimal.

4.4.2 Logical Instructions

In this section we discuss the logical instructions And, Or, Nor, and Exclusive Or, in machine language. Except for a different function code, they are identical to the `add` and `sub` instructions.

The Nor instruction is logically a composition of two operations, Or and Not:
$x\ Nor\ y\ =\ \sim(x \vee y)$
It is an R format instruction, as in the example below:

```
nor    $a0, $t0, $t1           # $a0 = $t0 nor $t1
```

These logical instructions are shown in Fig 4.7. We will take the `xor` example, and translate to machine language using the same three steps that were used in the previous section:

```
xor   $at, $ra, $a0
```

1. Show the fields in binary:

000000	11111	00100	00001	?????	100110
31 26	25 21	20 16	15 11	10 6	5 0
opcode	rs	rt	rd	shamt	funct

Again we show the `shamt` field as a don't care (i.e. question marks) because the logical R format instructions ignore this field.

2. Group the bits into groups of 4, as shown below.

```
0000 0011 1110 0100 0000 1??? ??10 0110
```

3. Convert to hexadecimal to get a more concise version of the machine language instruction. In doing so, we will assume all the don't-care values are zeros.

```
03 e4 08 26
```

The unary Not operation was covered in chapter 3 but it is actually a pseudo-operation making use of the identity:
$\sim x = x\ Nor\ 0$
The assembler will translate a Not statement into a Nor instruction in machine language; the second operand will be register $0. As an example, we take:

```
not   $v1, $t9      # $v1 = ~ $t9
```

For this example there will be an additional step to translate the `not` statement to an equivalent `nor` statement.

1. Translate to an equivalent `nor` statement:

```
nor   $v1, $t9, $t0    # ~ $t9 = $t9 nor 0
```

2. Show the fields in binary:

000000	11001	00000	00011	?????	100111
31 26	25 21	20 16	15 11	10 6	5 0
opcode	rs	rt	rd	shamt	funct

3. Group the bits into groups of 4, as shown below.

```
0000 0011 0010 0000 0001 1??? ??10 0111
```

4. Convert to hexadecimal to get a more concise version of the machine language instruction. In doing so, we will assume all the don't-care values are zeros.

```
03 20 18 27
```

4.4.3 Shift Instructions

In this section we discuss the left and right shift instructions which were introduced in chapter 3. There are two *logical* shift instructions: `sll` (shift left logical) and `srl` (shift right logical). There is one *arithmetic* shift instruction, which preserves the sign of the number by propagating the high order bit: `sra` (shift right arithmetic).

Instruction	opcode	function code	ignored fields	Example

srl	00	02_x	rs	`srl $a1, $v1, 17`
				`$rt   $rd   shamt`
				`03    05    11`

sll	00	00_x	rs	`sll $t0, $t0, 12`
				`$rt   $rd   shamt`
				`08    08    0c`

sra	00	03_x	rs	`sra $sp, $s0, 2`
				`$rt   $rd   shamt`
				`10    1d    02`

Figure 4.8: Machine language for shift instructions

Fig 4.8 shows the shift instructions. Since they are all R format instructions, they are similar to the previous instructions in this section. Note, however, that unlike the arithmetic and logical instructions, the shift instructions make use of the `shamt` field (number of bits to be shifted) and ignore the `$rs` field.

We take the `sra` example from that figure to translate to machine language:

```
sra   $sp, $s0, 2
```

1. Show the fields in binary:

000000	?????	10000	11101	00010	000011
31 26	25 21	20 16	15 11	10 6	5 0
opcode	rs	rt	rd	shamt	funct

2. Group the bits into groups of 4 as shown below.

```
0000 00?? ???1 0000 1110 1000 1000 0011
```

3. Convert to hexadecimal to get a more concise version of the machine language instruction. In doing so, we will assume all the don't-care values are zeros.

```
00 10 e8 83
```

4.4.4 Multiply and Divide Instructions

To conclude the section on R format instructions we consider the multiply and divide instructions. Since these instructions both put their results into the `lo` and `hi` registers, they ignore the $rd field, in addition to ignoring the `shamt` field. Fig 4.9 shows the machine language formats for multiply and divide.

		function	ignored	
Instruction	opcode	code	fields	Example

mult	00	18_x	\$rd, shamt	`mult $t0, $t1`
				`$rs   $rt`
				`08    09`

div	00	$1a_x$	\$rd, shamt	`div $a0, $a3`
				`$rs   $rt`
				`04    07`

Figure 4.9: Machine language for multiply and divide instructions

As an example we take the divide instruction from that figure, and translate to machine language:

```
div   $a0, $a3
```

1. Show the fields in binary:

000000	00100	00111	?????	?????	011010
31 26	25 21	20 16	15 11	10 6	5 0
opcode	rs	rt	rd	shamt	funct

2. Group the bits into groups of 4 as shown below.

```
0000 0000 1000 0111 ???? ???? ??01 1010
```

3. Convert to hexadecimal to get a more concise version of the machine language instruction. In doing so, we will assume all the don't-care values are zeros.

```
00 87 00 1a
```

This concludes our discussion of the R format instructions in machine language. Note that there are other R format instructions in the MIPS instruction set, but they will be described either in subsequent chapters or in the appendix. Also, there is an R format instruction, *set if less than* which will be discussed in connection with conditional branch instructions in a subsequent section.

4.4.5 Jump Register

The `jr` (Jump Register) instruction discussed in chapter 3 is an R format instruction. It has one operand, the register containing the memory address of the instruction to be executed next, in the `rs` field. The `rt, rd, shamt` fields are ignored, as shown in Fig 4.10

The example in Fig 4.10 is `jr   $ra` which we now translate to machine language in three steps:

Instruction	opcode	function code	ignored fields	Example

jr	00	08_x	rt, rd, shamt,	`jr $ra`
				`$rs`
				`31`

Figure 4.10: Machine language `jr` (Jump Register) instruction

1. Show the fields in binary:

000000	11111	?????	?????	?????	001000
31 26	25 21	20 16	15 11	10 6	5 0
opcode	rs	rt	rd	shamt	funct

2. Group the bits into groups of 4 as shown below.

```
0000 0011 111? ???? ???? ???? ??00 1000
```

3. Convert to hexadecimal to get a more concise version of the machine language instruction. In doing so, we will assume all the don't-care values are zeros.

```
03 e0 00 08
```

4.4.6 Exercises

1. Translate the following assembly language statements to machine language instructions. In each case, show your solution in hexadecimal.

   (a) `add  $v0, $t0, $t1`

   (b) `or   $ra, $v0, $s1`

   (c) `not  $t3, $a3`

   (d) `sll  $v1, $a0, 25`

   (e) `mult $a1, $s7`

   (f) `jr   $a3`

2. Translate each of the following machine language instructions to an equivalent assembly language statement. Note that some of the information provided should be ignored.

   (a) `00 0a 4c c2`

   (b) `00 85 11 20`

   (c) `00 a8 20 27`

   (d) `00 e2 ff 18`

4.5 I Format Instructions

The I format (immediate) instructions have many varied uses:

- Instructions which use a constant as an operand
- Memory reference instructions
- Conditional branch instructions

These instructions may ignore one or more fields in the I format, which we will point out.

4.5.1 Immediate instructions using constants

In chapter 3 we discussed several operations which involve a constant operand as part of the instruction. These included:

- `addi` (add immediate)
- Logical operations
  - `andi` (and immediate)
  - `ori` (or immediate)
  - `xori` (exclusive or immediate)

These will all be translated to machine language the same way. Fig 4.11 shows the machine language formats for these instructions. Note that although the destination operand, register `rt`, is first in assembly language, it is *not* first in machine language.

As an example we use the following:

```
xori    $a1, $a3, 1000
```

As in the previous section, we will use three steps to translate this assembly language statement to machine language.

1. We show the opcode and register fields in binary, but we find it easier to show the immediate field in hex:

001110	00101	00111	03e8
31 26	25 21	20 16	15 0
opcode	rs	rt	immediate

2. Group the binary fields into groups of 4:

```
0011 1000 1010 0111 03e8
```

3. Convert the binary fields to hexadecimal:

```
38 a7 03 e8
```

Instruction	opcode	ignored fields	Example

addi	08		`addi  $a1,  $v1, 27`
			`$rs   $rt   imm`
			`$v1   $a1    001b`

andi	0c		`andi  $t0,  $t1, -3`
			`$rs   $rt   imm`
			`$t1,  $t0   fffd`

ori	0d		`ori  $t0,  $0, 0`
			`$rs  $rt   imm`
			`$0,  $0,   0000`

xori	0e		`xori  $a3,  $a1, 1000`
			`$rs   $rt   imm`
			`$a1,  $a3,  3148`

Figure 4.11: Machine language formats for instructions which use the immediate field for a constant operand

Instruction	opcode	ignored fields	Example

lw	23		`lw    $a1, 12($a0)`
			`$rs   $rt   imm`
			`$a0   $a1    000c`

sw	2b		`sw    $t0, -24($a1)`
			`$rs   $rt   imm`
			`$a1   $t0    ffe8`

Figure 4.12: Machine language formats for memory reference instructions with explicit addressing

4.5.2 Memory Reference Instructions

Memory reference instructions enable the CPU to store information in main memory and retrieve information from main memory; these instructions are vital to software development. In the MIPS architecture the memory reference instructions are I (immediate) format instructions.

The two memory reference instructions which were discussed in chapter 3 were `lw` (load word) and `sw` (store word). In this section we cover assembly language statements with an explicit (i.e. non-symbolic) memory address.

For both the `lw` and the `sw` instructions, the `rs` operand is assumed to contain a memory address, and the immediate field is assumed to be an offset to, or displacement from, that address. The immediate field is twos complement representation, and consequently could be negative.

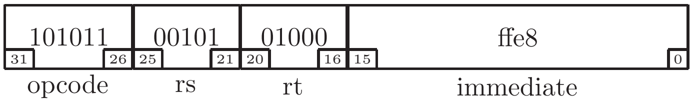
Fig 4.12 shows the format for both memory reference instructions.

We take the second instruction from Fig 4.12 for an example of translation to machine language:

```
    sw   $t0, -24($a1)
```

Again, we have a three step process:

1. We show the opcode and register fields in binary, but we find it easier to show the immediate field in hex:



2. Group the binary fields into groups of 4:
   `1010 1100 1010 1000 ffe8`

3. Convert the binary fields to hexadecimal:
   `ac a8 ff e8`

4.5.3 Memory Reference - Symbolic

Memory reference instructions often make use of symbolic addresses. A `lw` or `sw` instruction can refer to a label on a word in the data section. In this section we will see how such a memory reference can be translated to machine language.

First, we must understand what the assembler does with the values in the data section. These are the values which come after a `.data` directive. There can be several such directives in a program, but all the data values are placed in one memory area. The MARS assembler places all data in memory sequentially, beginning at memory address 0x1001000.

For example, the following statements:

```
               .data
salary:        .word     300
neg:           .word     -1
name:          .asciiz   "harry"
negByte:       .byte     -1
zero:          .word     0, 12
```

would cause the assembler to initialize memory as shown below:

10010000_{16}	00 00 01 2c	ff ff ff ff	72 72 61 68	00 ff 00 79
10010010_{16}	00 00 00 00	00 00 00 0c		

The labels in the data section have values corresponding to their memory addresses:

```
  salary =  0x10010000
  neg    =  0x10010004
  name   =  0x10010008
```

```
negByte = 0x1001000e
zero    = 0x10010010
```

We make a few observations on this memory diagram:

- The memory area begins at address 0x10010000.
- The characters of the string "harry" appear to be reversed in a memory word. The ascii codes, in hexadecimal are: 'h' = 68, 'a' = 61, 'r' = 72', y = '79'. Thus the byte addresses begin with the low order byte on the right:
  0x10010007, 0x10010006, 0x1001005, 0x1001004, in that order.
- A data value declared with the `.word` directive is a twos complement integer aligned on a *full word boundary.* This means that its hex address must end with a 0, 4, 8, or c. Thus the variable `zero` does not begin immediately after `negByte`; it begins at the next full word boundary, 0x10010010.

We can now explain memory references with symbolic addresses. Since the assembler placed the data values in a specific memory area, beginning at 0x10010000, and the assembler knows the address represented by each label, the assembler is able to calculate on offest from the start of the memory area to any given label's address.

For example, the label `name` corresponds to address 0x10010008. Therefore its offset from the start of the memory area is 0x8 bytes.

The assembler will convert a memory reference (`lw` or `sw`) with symbolic addresses to two machine language instructions:

1. A `lui` (load upper immediate) instruction to load the high order half word of the address into the `$at` (assembler temporary) register.
2. A `lw` (or `sw`) instruction, addressing the memory location explicitly using the address in the `$at` register, with an offset which brings the effective address to the desired location.

For example, the instruction

```
lbu    $t0, name      # first char of "harry"
```

will be translated to:

```
lui    $at, 0x1001    # $at = 0x10010000
lbu    $t0, 8($at)
```

The effective address obtained in the `lw` instruction will be 8 + 0x10010000 = 0x10010008, which is the memory address represented by the label `name`.

For another example, we will translate the instruction

```
lw     $a3, zero
```

to hexadecimal machine language. We do this in two steps:

1. Convert the `lw` statement to two statements:

```
lui    $at, 0x1001
lw     $a3, 0x0010
```

2. Translate each statement to hexadecimal machine language in three steps. First the `lui` statement:

   (a) Here we show all fields in binary:

001111	?????	00001	0001 0000 0000 0001
31 26	25 21	20 16	15 0
opcode	rs	rt	immediate

   (b) Group the binary fields into groups of 4, treating don't cares as zeros:

```
0011 1100 0000 0001 0001 0000 0000 0001
```

   (c) Convert the binary fields to hexadecimal:

```
3c 01 10 01
```

3. For the `lw` statement, we use the same three steps:

   (a) First we show all fields in binary:

100011	00001	00111	0000 0000 0001 0000
31 26	25 21	20 16	15 0
opcode	rs	rt	immediate

   (b) Group the binary fields into groups of 4:

```
1000 1100 0010 0111 0000 0000 0001 0000
```

   (c) Convert the binary fields to hexadecimal:

```
8c 27 00 10
```

Thus the assembly language statement

```
lw     $a3, zero
```

will be translated to the two machine language instructions:

```
3c011001
8c270010
```

4.5.4 Conditional Branches

In chapter 3 we learned of six conditional branch instructions:

- beq - branch if equal

- bne - branch if not equal
- blt - branch if less than
- ble - branch if less than or equal to
- bgt - branch if greater than
- bge - branch if greater than or equal to

The operands always consist of two registers, and a label on some other statement in the program, which is the destination for the branch. In each case the branch statement would compare the values in two registers, and transfer control to the statement with the given label if the comparison is true; otherwise, control would 'fall through' to the next statement.

Only the first two of these branch instructions `beq` and `bne` are actually MIPS instructions. The other four are pseudo-ops! The four pseudo-ops can all be implemented using a combination of the `slt` (set if less than) instruction and either `beq` or `bne`.

We'll begin by discussing the actual instructions first, then we'll handle the pseudo-ops. The branch statements always branch to a location which is some number of instructions *before* or *after* the instruction *following* the branch instruction. This is what we call a branch to a *relative* address.

Since all instructions are full words, they will always be located on full word boudaries, i.e. the address of every instruction will end with 00_2. This means that the field storing relative branch address can drop the last two bits! In other words, the branch instruction will store the number of *words* rather than the number of *bytes* in the immediate field. Also, for reasons to be explained in chapter 7, the branch is relative to the instruction *after* the branch instruction.

For example, if there is a branch instruction at memory address 0x00400030, then a branch to relative address +3 would branch to the instruction which is 3 instructions after the instruction after the branch instruction. This would be a branch to the instruction at address 0x00400040. A branch instruction at address 0x0040030 with a relative branch address of -4 would be a branch to the instruction which is 3 instructions *prior* to the branch instruction; it would be at address 0x0400034.

The branch instructions are I (immediate) format instructions. The two registers being compared are in the `rs` and `rt` fields of the instruction. The relative address is a twos complement value in the immediate field of the branch instruction. The branch instructions are described more formally in Fig 4.13.

The labels shown in Fig 4.13 are taken from the following example:

```
       .text
start:                          # Beginning of text area
       li    $t0, 0
       li    $t1, 1
a:     beq   $t0, $t1, forward    # forward branch
       li    $t2, 2
```

Instruction	opcode	ignored fields	Example

beq	04		`a:  beq  $t0, $t1, forward`
			` $rs   $rt   imm`
			` $t0   $t1   0003`

bne	05		`b:  bne  $t3, $t2, start`
			` $rs   $rt   imm`
			` $t3   $t2   fff8`

Figure 4.13: Machine language formats for actual MIPS branch instructions

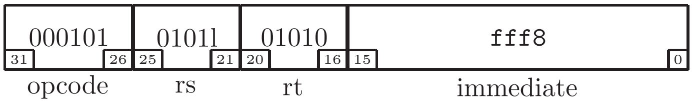
```
       li    $t3, 3
       li    $t4, 4
forward:
       li    $t5, 5
b:     bne   $t3, $t2, start      # backward branch
       li    $t6, 6
```

Note in Fig 4.13. that branch (`beq`)to `forward` is three instructions after the `li $t2, 2` instruction. Also the branch (`bne`)to `start` is eight instructions prior to the `li $t6, 6` instruction.

Translating the `bne` instruction to machine language, we use the same three steps used previously:

`b:   bne   $t3, $t2, start`

1. We show the opcode and register fields in binary, but we find it easier to show the immediate field in hex (-8 = 0xfff8):



2. Group the binary fields into groups of 4:
   `0001 0101 0110 1010 fff8`

3. Convert the binary fields to hexadecimal:
   `15 6a ff f8`

Next we discuss the implementation of the four pseudo-ops separately (in each case, the registers being compared are called $rs and $rt; the destination for the branch is called `dest`):

- `blt   $rs, $rt, dest   # branch if less than`

  1. Compare the two registers using `slt`, storing the result in the $at register:
     `slt   $at, $rs, $rt      # $rs < $rt ?`

Instruction	slt	branch
`blt  rs, rt, dest`	`slt  $at, rs, rt`	`bne  $at, $0, dest`
`ble  rs, rt, dest`	`slt  $at, rt, rs`	`beq  $at, $0, dest`
`bgt  rs, rt, dest`	`slt  $at, rt, rs`	`bne  $at, $0, dest`
`bge  rs, rt, dest`	`slt  $at, rs, rt`	`beq  $at, $0, dest`

Figure 4.14: Implementation of the four branch pseudo-ops

2. Branch if the result is 1:
   ```
   bne   $at, $0, dest      # branch if not 0
   ```

- `ble   $rs, $rt, dest   # branch if less than or equal`
  Here we make use of the identities: $a \leq b = \sim\ a > b$ and $a > b = b < a$.

  1. Compare the two registers using `slt` but *reverse* the operands, storing the result in the $at register:
     ```
     slt   $at, $rt, $rs      # $rt < $rs ?
     ```
  2. Branch if the result is not 1:
     ```
     beq   $at, $0, dest      # branch if 0
     ```

- `bgt   $rs, $rt, dest   # branch if greater than`
  Here we make use of the identity $a > b = b < a$.

  1. Compare the two registers using `slt` but *reverse* the operands, storing the result in the $at register:
     ```
     slt   $at, $rt, $rs      # $rt < $rs ?
     ```
  2. Branch if the result is 1:
     ```
     bne   $at, $0, dest      # branch if not 0
     ```

- `bge   $rs, $rt, dest   # branch if greater than or equal`
  Here we make use of the identity $a \geq b = \sim\ a < b$.

  1. Compare the two registers using `slt` but *reverse* the operands, storing the result in the $at register:
     ```
     slt   $at, $rs, $rt      # $rs < $rt ?
     ```
  2. Branch if the result is not 1:
     ```
     beq   $at, $0, dest      # branch if 0
     ```

Fig 4.14 summarizes the implementation of the four branch pseudo-ops.

Before taking an example, we must be very careful when counting the instructions before or after a branch; some of the statements might be psuedo-ops which expand to two or more MIPS instructions!

We take the following example to translate a branch pseudo-op to machine language:

```
bgt   $a0, $t0, done      # branch to done if $a0 > $t0
li    $t0, 0
```

```
       lw    $t1, x              # two instructions!
done:
       sub   $t3, $t4, $t5
```

Here we will translate only the branch instruction to machine language:

```
   bgt   $a0, $t0, done
```

First we expand the branch pseudo-op to two instructions:

```
      slt   $at, $t0, $a0
      bne   $at, $0, 3       # branch forward 3 instructions
```

Next we translate each instruction to machine language in three steps. The `slt` (R format) instruction translated:

1. We show the fields in binary, with don't cares shown as question marks.

000000	01000	00100	00001	?????	101010
31 26	25 21	20 16	15 11	10 6	5 0
opcode	rs	rt	rd	shamt	funct

2. Group the binary fields into groups of 4, assuming dont cares are zeros:
   `0000 0001 0000 0100 0000 1000 0010 1010`

3. Convert the binary fields to hexadecimal:
   `01 04 08 2a`

Next we translate the `bne` instruction:

1. We show the opcode and register fields in binary, but we find it easier to show the immediate field in hex:

000101	0000l	00000	0003
31 26	25 21	20 16	15 0
opcode	rs	rt	immediate

2. Group the binary fields into groups of 4:
   `0001 0100 0010 0000`

3. Convert the binary fields to hexadecimal:
   `14 20 00 03`

The two machine language instructions resulting from the conditional branch are:

```
0104082a
14200003
```

A final word of caution is in order here. When branching across another branch instruction, and calculating the relative address, be careful. If branching across a `beq` or `bne`, count it as one instruction. But if branching across any of the four branch pseudo-ops, count it as *two* instructions.

4.5.5 Exercises

1. Translate each of the following instructions to machine language:

   (a) `xori   $v0, $a3, 0xffff`

   (b) `addi   $ra, $ra, -8`

   (c) `lw     $s0, 4($sp)`

2. Given the program shown below:

```
        .text
start:
        lw     $a0, parm
        addi    $t0, $t0, 64
        ble    $t0, $s0, done
        sw     $t0, x
        bne    $t0, $t1, start
        bgt    $a0, $a1, start
done:
        .data
x:      .word     7
parm:   .word     3
```

   (a) Show a diagram of the hexadecimaldata in memory, with hexadecimal address(es).

   (b) Translate the `lw` instruction to hexadecimal machine language.

   (c) Translate the `addi` instruction to hexadecimal machine language.

   (d) Translate the `ble` instruction to hexadecimal machine language.

   (e) Translate the `sw` instruction to hexadecimal machine language.

   (f) Translate the `bne` instruction to hexadecimal machine language.

   (g) Translate the `bgt` instruction to hexadecimal machine language.

3. Given the following hexadecimal machine language program, translate it to assembly language (there could be many correct solutions):[6]

```
38 e2 ff ff
23 ff ff f8
3c 01 10 01
8c 30 00 04
```

[6] Translating from machine language to assembly language is called *disassembling* or *reverse engineering*; hackers and security analysts often do it.

Instruction	opcode	Example

j	02	`j foo`
		`address`
		0x010000e.

jal	03	`jal foo`
		`address`
		0x010000e.

Figure 4.15: Machine language for `j` (Jump) and `jal` (Jump And Link) instructions. `foo` labels the instruction at location 0x0040001c

4.6 J Format Instructions

As discussed in chapter 3 the jump instructions are used for unconditional transfer of control. The MIPS archtitecture has two such instructions, `j` (Jump) and `jal` (Jump And Link)[7] which are described in Fig 4.15

The J format has two fields:

- opcode - 6 bits
- jump address - 26 bits

The jump address is an *absolute* address, unlike the address in branch instruction, which is a *relative* address. Since the memory is byte addressable, and all instructions are one word (4 bytes) in length, the low order two bits in an instruction address will always be `00`. For this reason the address in the address field omits the two low order bits. For example, if a jump to the instruction at memory location 0x0040204c is desired, the address field should store the 26 bit value 0x0100813. There are two ways to arrive at this:

- Divide by 4:
  0040204c / 4 = 00100813
  Take the low order 26 bits (the first hex digit represents two bits):
  0x0100813
- Write out the bits of the full address:
  0000 0000 0100 0000 0010 0000 0100 1100
  Shift right, two bits:
  0000 0000 0001 0000 0000 1000 0001 0011
  Take the low order 26 bits:
  00 0001 0000 0000 1000 0001 0011
  Write in hex (the first hex digit represents 2 bits):
  0100813

We are now ready to translate a jump instruction to machine language. Here is the example, beginning at memory location 0x00400000:

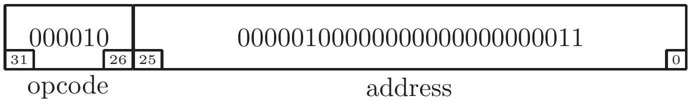
[7] The `jr` (Jump Register) instruction is R format, and was discussed earlier in this chapter.

```
start:
       j      done
       li     $t0, 3
       add    $t0, $t1, $t4
done:
       jr     $ra
```

We wish to translate the `j    done` statement to machine language. This can be done in four steps:

1. Find the 26-bit jump address:
   `done` labels the instruction at location 0x0040000c. Dividing by 4 and taking the low order 26 bits we get 0100003.

2. Show the fields in binary:



3. Group the bits into groups of 4:

```
0000 1000 0001 0000 0000 000 0000 0011
```

4. Convert to hexadecimal to get a more concise version of the machine language instruction.

```
08 10 00 03
```

4.6.1 Exercises

1. Given the program shown below, translate the `j    lp` statement to machine language.

```
start:
       add   $t0, $a0, $0
       sub   $t1, $a1, $a2
lp:
       bne   $t0, $0, done
       addi  $t0, $t0, -1
       xori  $t3, $t3, 0xffff
       j      lp
done:
```

2. Given the program shown below, translate the `jal    fn` statement to machine language.

```
start:
       jal   fn         # call the function
       bne   $t0, $0, done
       addi  $t0, $t0, -1
done:
       jr    $ra
fn:
       addi  $t3, $t3, 5
       jr    $ra
```

3. Translate the program shown below to machine language. Assume the starting location is 0x00400000.

```
start:
       li      $t3, 7
       li      $t4, 12
lp:
       bne     $t3, $t4, done
       jal     function
       addi    $t3, $t3, 1
       j       lp
done:
       jr      $ra

function:
       addi    $sp, $sp, 4
       mult    $a0, $a1
       mflo    $t0
       jr      $ra
```

4.7 Floating Point Data Representation

In this section and the following section we will examine floating point data representation and the floating point instructions at the machine language level. In order to have a good understanding of the floating point instructions, we will first need to have a good understanding of floating point data representation. Once we have accomplished this, we then move on to examine the floating point instructions in machine language.

In chapter 3 we learned of floating point instructions, but we did not look at the floating point data in any detail. In our programs we may have had a data value such as:

```
pi     .float   3.14159
```

In stepping through a program with MARS the reader may have looked at these floating point values, as represented in hexadecimal. At the time they seemed to bear no resemblance to the values they represent. In this section we will take a careful look at the details of floating point data.

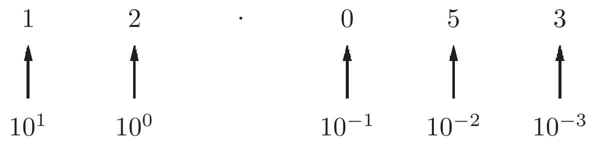


Figure 4.16: Decimal representation of the number 12.053

4.7.1 Fixed Point in Binary

The first step is to understand what we call *fixed point* representation for rational numbers. In base 10 this would be a number such as 3.14159, but here we are working in base 2.

Fig 4.16 shows in the form of a diagram the meaning of the value 12.053 in which the position of each digit represents a power of 10. The positions after the decimal point represent negative powers of 10, as shown below:

- $10^1 = 10$
- $10^0 = 1$
- $10^{-1} = 0.1$
- $10^{-2} = 0.01$
- $10^{-3} = 0.001$

Consequently 12.053 = 10 * 1 + 1*2 + 0.1*0 + 0.01*5 + 0.001*3.

The table below shows the same concept, in base two.

power of 2	in base 10	in base 2
2^2	4	100_2
2^1	2	10_2
2^0	1	1_2
2^{-1}	$1/2 = 0.5$	0.1_2
2^{-2}	$1/4 = 0.25$	0.01_2
2^{-3}	$1/8 = 0.125$	0.001_2

Now that we understand the meaning of places after the binary point, we can produce the binary fixed point representation for a rational number. For example, $3/4 = 0.5 + 0.25 = 0.75 = 0.11_2$ and $3/8 = 1/4 + 1/8 = 0.375 = 0.011_2$ A diagram of some binary fixed point numbers is shown in Fig 4.17.

In that diagram we show only values which can be represented with perfect precision. However, as with decimal numbers, there will be some rational numbers for which we do not have an exact representation. For example, 1/3 is not exactly equal to 0.333333, and no matter how many decimal places we write, it will never be perfectly accurate.

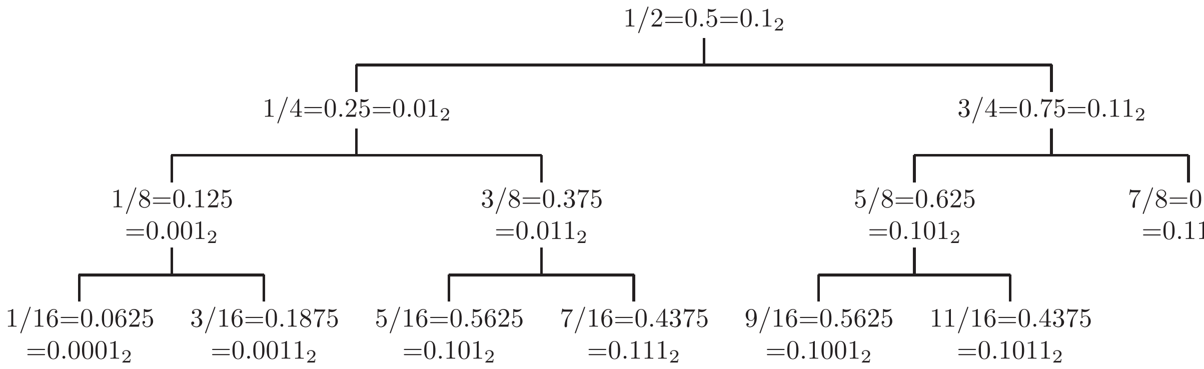


Figure 4.17: Diagram of fractions using binary fixed point representation

In base two, 1/3 could be approximated as follows:

$1/3 \approx 1/4 + 1/16 + 1/64 = 0.010101_2$

Interestingly there are numbers, such as one tenth, which can be represented exactly in decimal, but not in binary. This is why we can get apparently strange results when doing calculations with numbers which are not integers. For example, try the following in Java or C++:

```
if (0.1 + 0.1 + 0.1 == 0.3) ...
```

4.7.2 IEEE 754 Floating Point Data Representation

Over the years various different floating point representations have been used. In recent years most architectures, including MIPS have settled on a standard known as IEEE[8] 754, which is the one described here.

Before describing this format, we take a look at the *exponent* notation used in most high level languages. Avogadro's number, in chemistry, is 6.02×10^{23}. In most high level programming languages it can be written as `6.02e23`. There are other, equivalent, ways of writing this number:

`6.02e23 = 60.2e22 = 602.0e21 = 0.602e24`

There is more than one representation for the same number, but one of them can be chosen as a preferable representation, or *normal form* of the number. We choose the `6.02e23` as the normal form, and in general, we require the normal form to show one non-zero digit before the decimal point.[9].

Note that the exponent can also be negative:

`0.0402 = 4.02e-2 = 40.2e-3 = 0.402e-1 = 0.0402e0`

Again, we choose the representation with one non-zero digit before the decimal point, `4.02e-2` as the normal form representation. If we remove the decimal point from these representations, the part before the `e` is called the *mantissa* and the part after the decimal point is called the *exponent*. In this example the digits of the mantissa are `402` and the exponent is `-2`.

[8] Institute of Electrical and Electronic Engineers

[9] we'll need a special representation for 0.0, such as 0.0e0

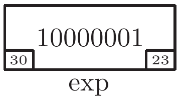
An IEEE 754 (single precision) floating point number is a 32-bit format. It consists of three parts:

- Sign (1 bit): This is the sign of the number (not to be confused with the sign of the exponent). 0 represents positive, 1 represents negative.
- Exponent (8 bits): This is an exponent of 2, in *excess-127 notation*. The value used for the exponent is the amount by which this (unsigned) field exceeds 127. For example, if this field is 10000011 = 131, then the exponent is 4, and if this field is 01111000 = 120, then the exponent is -7.
- Fraction (23 bits): The fraction is taken from the normalized mantissa; imagine a binary point with a single bit, always 1, before the binary point, and the exponent is adjusted accordingly. Since the high order bit of the mantissa is always 1, it is not stored as part the fraction!

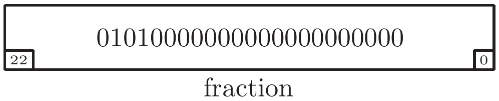
The number 0.0 is handled as a special case: 0x00000000

A few examples should help to clarify floating point data representation. Our first example is 5.25 which will be converted to floating point in eight steps:

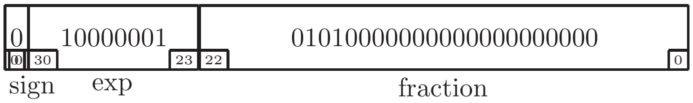
1. The number is positive so the sign field is 0.
2. Write the absolute value of the number as a fixed-point binary number. $5.25 = 101.01_2$
3. Normalize, so there is a single 1 in front of the binary point. $101.01 = 1.0101 \times 2^2$
4. The exponent is 2, which is 129 = 10000001 in excess-127 notation



5. The mantissa is 1.0101_2
   Obtain the 23-bit fraction by dropping the high order 1 :



6. Assemble the three fields in order:



7. Group in groups of 4:
   0100 0000 1010 1000 0000 0000 0000 000

Number	5.25	fraction	$\cancel{1}$01010000000000000000000
sign	0	binary result	0 10000001 01010000000000000000000
normalized	$101.01_2 = 1.0101 \times 2^2$	group by 4	0100 0000 1010 1000 0000 0000 0000 0000
exponent	$127 + 2 = 129 = 10000001_2$	hex result	40a80000
Number	0.375	fraction	$\cancel{1}$10000000000000000000000
sign	0	binary result	0 01111101 10000000000000000000000
normalized	$0.011_2 = 1.1 \times 2^{-2}$	group by 4	0011 1110 1100 0000 0000 0000 0000 0000
exponent	$127 - 2 = 125 = 01111101_2$	hex result	3ec00000
Number	-27.0	fraction	$\cancel{1}$10110000000000000000000
sign	1	binary result	1 10000011 10110000000000000000000
normalized	$127 + 4 = 131 = 10000011_2$	group by 4	1100 0001 1101 1000 0000 0000 0000 0000
exponent	$127 + 4 = 131 = 10000011_2$	hex result	c1d80000

Figure 4.18: Examples of IEEE 754 single precision floating point data: 5.25, 0.375, -27.0

8. Write in hex, for clarity:
   40 a8 00 00

Fig 4.18 shows a few other examples of floating point numbers in IEEE 754 format, specifically 0.375 and -27.0. As you read the table in Fig 4.18, read down the left column first, then read down the right column.

4.7.3 Exercises

1. Show each of the following numbers in binary fixed point notation. If the number cannot be represented precisely, show enough binary places to indicate a repeating sequence of bits after the binary point.

   (a) $4\frac{3}{4}$

   (b) 13/16

   (c) 7.0

   (d) 0.1

   (e) 13.6

2. Show each of the following numbers in IEEE 754 single precision floating point format. Show your final result in hexadecimal,

   (a) 17.0

   (b) 13.375

   (c) 0.15625

   (d) -5.0

   (e) 0.0

   (f) 3.6

3. (a) Run the following Java or C++ code, and explain why it appears to behave in an undesirable way:

```
for (double x = 0.0; x!=1.0; x = x + 0.1)
   System.out.println ("x is " + x);            \\ If using Java
   cout << "x is " << x;                        \\ If using C++
```

   (b) Show a better way to code the following statement:

```
if (x == 3.1)    x = 0.0;
```

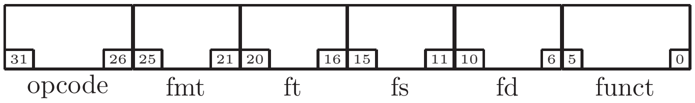
4.8 Floating Point Instructions

We now turn our attention to the machine language instructions which work with floating point data. The implementation of these instructions is rather complex. In the early years of computing all floating point arithmetic was done with software; when the floating point algorithms had reached a good level of efficiency and accuracy, and the hardware technology had improved, the floating point instructions were implemented in hardware (circa 1983).

4.8.1 MIPS Arithmetic Floating Point Instruction Formats

There are two floating point instruction formats in the MIPS archtitectrure: FR (Floating point Register format) and FI (Floating point Immediate format). Most floating point instructions are FR format. FI is used only for conditional branches.

The FR format is shown below:



Note that the three register operands are in reverse order, as compared with assembly language. For example, in the instruction
`sub.s   $f2, $f4, $f6`
the three register operands are fd = \$f2, fs = \$f4, and ft = \$f6. In machine language the three register operands are ft = \$f6, fs = \$f4, and fd = \$f2. Also note that there is an *expanding opcode* with the fields `opcode`, `fmt`, and `funct` all serving to identify the instruction.

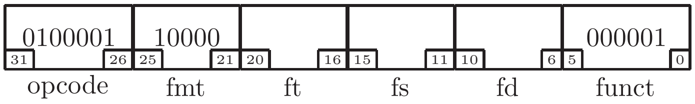
The machine language format for add, subtract, multiply, and divide are shown in Fig 4.19. The instructions are similar, but differ only in the funct field.

As an example, we take the same floating point instruction:
`sub.s   $f2, $f4, $f6`
and convert it to machine language in five steps:

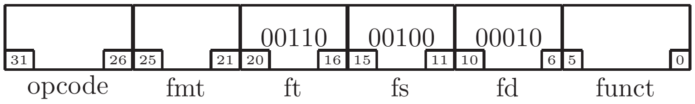
Instruction	opcode (6)	fmt (5)	funct (6)
add.s	11	10	00
sub.s	11	10	01
mul.s	11	10	02
div.s	11	10	03

Figure 4.19: Machine language for floating point instructions (all values are in hexadecimal)

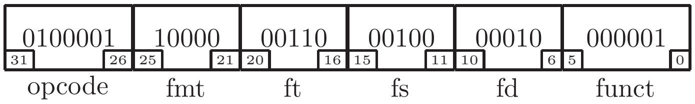
1. The opcode, fmt field, and funct field are taken from Fig 4.19.
   opcode = 11_{16}
   fmt = 10_{16}
   funct = 01_{16}



2. The register operands are
   ft = \$f6, fs = \$f4, and fd = \$f2.



3. We now have the complete instruction in binary



4. Grouping in groups of 4 bits, we have:
   `0100 0110 0000 0110 0010 0000 1000 0001`

5. And finally, we show the instruction in hexadecimal
   `46062081`

4.8.2 Floating Point Memory Reference Instruction Formats

The load and store instructions for floating point data actually use the I format (*not* the FI format), as covered previously. These are the `lwc1` (Load Word

Instruction	opcode	ignored fields	Example

lwc1	0x31		`lwc1    $f2, 12($a0)`
			`$rs   $rt   imm`
			`$a0   $f2    000c`

swc1	0x39		`swc1    $f4, -24($a1)`
			`$rs   $rt   imm`
			`$a1   $f4    ffe8`

Figure 4.20: Machine language formats for memory reference instructions for floating point data (single precision)

Coprocessor 1) and `swc1` (Store Word Coprocessor 1) instructions. They are analogous to the `lw` and `sw` instructions, and are summarised in Fig 4.20.

As an example, we translate the following instruction to machine language in three steps:

```
swc1   $f12, 8($a0)
```

1. Show the fields in binary:

111001	00100	01100	0000000000001000
31 26	25 21	20 16	15 0
opcode	rs	rt	immediate

2. Group the bits into groups of 4, as shown below.

```
1110 0100 1000 1100 0000 0000 0000 1000
```

3. Show the instruction in hexadecimal:
   `e4 8c 00 08`

The `lwc1` and `swc1` instructions work with symbolic addresses in exactly the same way as the `lw` and `sw` instructions described previously.

4.8.3 Floating Point Conditional Branch Instruction Formats

As we saw in chapter 3, conditional branching is significantly different when comparing floating point values. There is a separate instruction format, FI, for a floating point conditional branch.

Floating point comparison instructions

But first we will cover the comparison instructions. `c.eq.s`, `c.lt.s`, and `c.le.s`. These are all FR format instructions, and are shown in Fig 4.21. Note that these three instructions differ only in the function field.

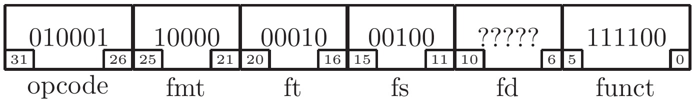
Instruction	opcode (6)	fmt (5)	funct (6)	fd (5)
c.eq.s	11	10	32	ignored
c.lt.s	11	10	3c	ignored
c.le.s	11	10	3e	ignored

Figure 4.21: Machine language for floating point comparison instructions, in FR format (all values are in hexadecimal)

As an example, we translate the following instruction to machine language in three steps:

```
c.lt.s   $f4, $f2
```

1. Show the fields in binary:



   The fd field is a don't care.

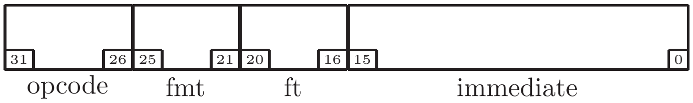
2. Group the bits into groups of 4, as shown below.

```
0100 0110 0000 0010 0010 0??? ??11 1100
```

3. Show the result in hexadecimal, assuming the don't cares are zeros.
   `46 02 20 3c`

Floating point conditional branch instructions

The floating point conditional branch uses the `FI` format:



These instructions make use of the floating point condition code, set by a floating point compare instruction, as discussed in chapter 3. Fig 4.22 shows the field values for these instructions in machine language. Note that they differ only in the funct field.

As an example, we translate the conditional branch instruction, taken from the code segment shown below, to machine lanuguage in five steps:

```
lp:
         add.s    $f2, $f4, $f2
         mul.s    $f6, $f8, $f2
         c.lt.s   $f6, $f2
         bc1f     lp
```

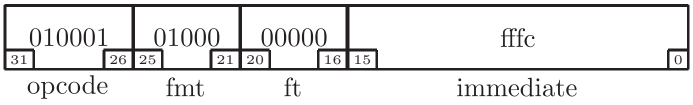
Instruction	opcode	fmt	ft	immediate	Example
bc1t	0x11	0x08	0x01	0x0018	`bc1t 24` Relative branch +24
bc1f	0x11	0x08	0x00	0xffe0	`bc1f -32` Relative branch -32

Figure 4.22: Machine language formats for floating point conditional branch instructions

1. Find the immediate field. The `lp` label is 4 instructions *prior* to the branch, so the immediate field is -4 = 0xfffc.

2. Find the other fields (see Fig 4.22).
   opcode = 010001
   fmt = 01000
   ft = 00000

3. Show the instruction in binary (we show the immediate field in hex)



4. Group the binary values in groups of 4 bits:
   `0100 0101 0000 0000 ff fc`

5. Show the result in hex: `45 00 ff fc`

4.8.4 Exercises

1. Given the following code:

```
         .text
         c.le.s   $f6, $f2
         bc1t     done
         sub.s    $f12, $f16, $f2
         lwc1     $f6, x
         div.s    $f0, $f28, $f2
done:
         swc1     $f0, y
         .data
x:       .float   3.45
y:       .float   0
```

   (a) Translate the `sub.s` instruction to machine language

(b) Translate the `div.s` instruction to machine language

(c) Translate the `lwc1` instruction to machine language

(d) Translate the `swc1` instruction to machine language

(e) Translate the `c.le.s` instruction to machine language

(f) Translate the `bc1t` instruction to machine language

Chapter 5

A MIPS Assembler

In chapter 4 we have described the purpose of an assembler: to translate an assembly language program to machine language instructions, and we learned to perform this translation manually for the MIPS architecture. In this chapter we will program the assembler, from the ground up. We intend to use MIPS assembly language to implement our assembler. It may seem problematic to implement an assembler using the assembly language that we are implementing, but in fact this is often done in a process known as *bootstrapping* - we construct a small version of the assembler, perhaps in machine language, then construct pregressively larger versions using the assembler already constructed.[1]

Intitially the input to our assembler will be a series of strings in memory, such as

```
"add   $2, $3, $4"
"sub   $12, $2, $3"
```

The output will be the binary instructions in memory (shown here in hexadecimal):
00641020_{16}
00436022_{16}
In a later stage we may wish to read the input from a source file.

Why are we building an assembler, when we already have a perfectly good one (i.e. MARS)? Our motivation here is educational. By building an assembler we will have a better understanding of its workings and of the MIPS architecture in general.

Our plan is to build the assembler incrementally, in a series of stages, or versions. We begin with a fairly simple version, with only a few useful features. Each version will add one or more features to our assembler - a rough outline is shown below:

1. R format instructions only

[1] Here we will not be bootstrapping our assembler, as we will be using the MARS assembler/simulator to develop and test it.

   (a) No symbolic registers
   (b) Include symbolic registers

2. Include I format instructions

   (a) Load, store (explicit addresses)
   (b) Allow symbolic addresses
   (c) Branch instructions

3. Include J format instructions

Working on each version of the assembler, we plan to build low-level supporting functions first, then work our way up to higher-level functions which make use of the lower-level functions. As we do this, we should test each function with its own specially designed driver. In most cases we will not show the driver in order to conserve space in the text.

To avoid name conflicts we will append the name of the function to the name of each label, as described in chapter 3.

In what follows we will use the word *statement* to refer to one line of assembly language (which could be a pseudo-op), and we will use the word *instruction* to refer to a full-word machine language instruction. One statement could result in one or more instructions.

5.1 Version 1 - R Format Instructions Only

In this section we will implement R format instructions, implementing a few common arithmetic and logical instructions (it should be clear how to include other R format instructions).

5.1.1 Version 1a - No Symbolic Registers

We begin by allowing assembly language statements with explicit register numbers, but no register names. Instead of

```
     add $v0, $v1, $t0
```

we require the register numbers:

```
     add $2, $3, $8
```

skipCommaWhite

Note that in an assembly language statement the mnemonic is separated from the operands by one or more spaces and/or tab characters (we call this *white space*[2] because that is what it looks like when printed on white paper). Note also that the operands are separated from each other by commas and possibly white space. The MARS assembler does not require the use of commas, i.e. white

[2] For our purposes here we exclude newline characters from white space.

space can be used interchangeably with commas, and we will take the same approach. Thus the above statement could conceivably (but not advisably) be written as:

```
add,$2      $3,,  ,     $8
```

Thus the first, lowest-level function, will be a function which scans a string from a given starting point until it finds a character which is neither white space nor comma (i.e. neither space nor tab nor comma). We call this function `skipCommaWhite`.

This function is shown in Fig 5.1. Note that the API for this function specifies that register $a0 points to a character in a statement (the start point for the scan). The post conditions are (1) register $a0 points to the first non-white, non-comma character found and (2) register $v0 is unchanged. The `skipCommaWhite` function uses local data to store the space, tab, and comma characters, which are loaded into registers $t1, $t2, and $t3, respectively. In the body of the loop it checks a character from the given string for one of these *delimiters*, terminating if not found. In order to satisfy the post condition, it decrements the pointer in $a0 when finished. This function does not need to save any registers because it does not use any `s` registers, and it does not call any functions (which would clobber the $ra register).

As we develop our assembler incrementally, it is important that we test each function as it is developed. For this purpose we should develop a *driver* for each function that we develop.[3] The sole purpose of the driver is to test a specific function. A driver for the `skipCommaWhite` function is shown in Fig 5.2.

To run the driver, we could copy and paste the `skipCommaWhite` function into the file containing the driver. However, that would give us two identical copies of `skipCommaWhite`. This is not a good idea - if we ever need to make a change to this function, we would need to make the change in each copy. For this reason, duplicated code should be avoided whenever possible.

The MARS assembler provides a way of avoiding duplicated code; it is the `include` directive. The operand is a file name, and the code from that file is included as the assembler processes the current source file. The `include` directive is after the `syscall` termination statement in Fig 5.2.

strcmp

We next turn our attention to the mnemonic in a statement. As noted in chapter 3 the mnemonic represents the operation to be performed. For example in the statement

```
add    $2, $3, $8
```

the mnemonic is `add`. As our assembler encounters a mnemonic in a statement, we will need to determine which operation it represents. To do this we will use a table of valid mnemonics, and compare against the entries in the table. In chapter 3 we devloped a function to compare strings, `strcmp`, but unfortunately it assumed the strings were terminated with null bytes. We will have to modify

[3] In most cases we leave the driver as an exercise.

```
################### skipCommaWhite function begin ####################

#  Pre:  $a0 points to a char in a statement
#  Post: $a0 contains address of first non-white and non-comma byte encountered
#        $vo is unchanged

###   local data
                 .data
space_skipCommaWhite:     .asciiz       " "
tab_skipCommaWhite:       .asciiz       "\t"
comma_skipCommaWhite:     .asciiz        ","

                 .text
skipCommaWhite:
       lbu       $t1, space_skipCommaWhite
       lbu       $t2, tab_skipCommaWhite
       lbu       $t3, comma_skipCommaWhite
lp_skipCommaWhite:
       lbu       $t0, 0($a0)                       # load byte from input string
       addi      $a0, $a0, 1
       beq       $t0, $t1, lp_skipCommaWhite       # check for space
       beq       $t0, $t2, lp_skipCommaWhite       # check for tab
       beq       $t0, $t3, lp_skipCommaWhite       # check for comma
done_skipCommaWhite:
       addi       $a0, $a0, -1
       jr        $ra
######################### skipCommaWhite function end  ##################
```

Figure 5.1: Function to scan past white space and commas in a statement

```
#  Test a function which reads past white space in a string

## Local data
              .data
input:        .asciiz  "  , ,              abc  "
              .word        -1
result:       .byte       0

              .text
main:
       la      $a0, input        # address of first byte in string
       jal     skipCommaWhite
       lbu     $t0, 0($a0)       # should be non-white char

       sb      $t0, result       # should be 'a'

       li      $v0, 10           # terminate the test
       syscall

       .include "skipCommaWhite.asm"
```

Figure 5.2: Driver to test the `skipCommaWhite` function

that function to use it here, because the strings we are comparing could be terminated with white space or commas.

The modified version of `strcmp` is shown in Fig 5.3. Note that we load the space and tab characters into registers $t2 and $t3, respectively. Then, in the loop, we load a character from the first string into register $t0 and the corresponding character from the second string into register $t1. After subtracting these characters, we know that if the result is not zero, the strings could not be equal, and we terminate the function. If we reach the end of both strings on the same iteration of the loop, then we know the strings are equal.

The driver for `strcmp` should test several cases:

- The strings are equal, with identical terminating characters
- The strings are equal, with different terminating characters
- The strings are not equal
- The strings are not equal, but one string is a prefix of the other string, e.g. “add” and “addi”

memonic

In the `mnemonic` function, the assembler scans the mnemonic in the statement, locates that mnemonic in a table of mnemonics, and starts to con-

```
################### strCmp function begin ####################

# Pre:  $a0 contains address of first source string
#       $a1 contains address of second source string
#        both strings terminated with a space or tab char!
#
# Post: Return in v0:
#    Zero if equal

####   local data
                .data
space_strCmp:   .asciiz       " "
tab_strCmp:     .asciiz       "       "

                .text
strcmp:
       lb       $t2, space_strCmp
       lb       $t3, tab_strCmp
lp_strCmp:
       lb       $t0, 0($a0)                   # load byte from src1
       lb       $t1, 0($a1)                   # load byte from src2
       sub      $v0, $t0, $t1                 # v0 = t0 - t1
       bne      $v0, $0, unequal_strCmp
       beq      $t0, $t2, done_strCmp         # space or tab?
       beq      $t0, $t3, done_strCmp

       # Advance to next byte of each string
       addi     $a0, $a0, 1                   # Go to next of src1
       addi     $a1, $a1, 1                   # Go to next byte of src2
       j        lp_strCmp                     # repeat the loop

unequal_strCmp:
       beq       $t0, $t2, white0_strCmp
       beq       $t0, $t3, white0_strCmp
       j         done_strCmp
white0_strCmp:
       beq       $t1, $t2, white1_strCmp
       beq       $t1, $t3, white1_strCmp
       j         done_strCmp
white1_strCmp:
       move       $v0, $0                     # Strings are equal

done_strCmp:
       jr       $ra                           # return
######################## strCmp function end  ##################
```

Figure 5.3: Function to compare strings for equality. The strings are terminated by white space or a comma

struct the machine language instruction. Once the mnemonic is recognized, we know the machine language opcode and function code. The name of the table is `ops_mnemonic`, and each entry consists of a 7-character string (for the mnemonic) and a full word for the instruction, with an opcode and function code. For example, the first entry is `"add    "`. It is directly followed by the hexadecimal word `0x00000020`, which is a machine language `add` instruction (opcode=0, function code=20). The table contains a few mnemonics which are not R format instructions; they will be used in the next version of the assembler, and they will be tested after we include I format instructions in the next version of the assembler.

The `mnemonic` function is shown in Figs 5.4 and 5.5 The API specifies that register $a0 contains the memory address of the statement, and register $a1 contains the memory address for the resulting machine language instruction. At this point (version 1) labels are not permitted in a statement, so the `mnemonic` function calls `skipCommaWhite` to arrive at the mnemonic. It then enters a loop in which it compares the mnemonic in the statement with the mnemonics in the table. If it finds the desired mnemonic, it fills in the instruction from the table. If it finds the statement `end` it sets a return code of 0. If it does not find the mnemonic in the table, it sets a negative return code, indicating an error has occurred.

The `mnemonic` function makes use of several `s` registers, which must be saved to (and reloaded from) the runtime stack, in additon to the $ra register. Also this function includes the `strcmp` function because `strcmp` is called from `mnemonic`. However, `mnemonic` also calls `skipCommaWhite`.

At this point we have the makings of a machine language instruction, with an opcode and function code. All that needs to be done is fill in the three register operands.

parseInt and isNumeric

In order to fill in the register operands, we will need to scan the register numbers in the statement, remembering that they are numeric characters, and convert each of them to a 5-bit binary field in the instruction. For example, in the statement

```
    add   $3, $31, $21
```

the three operands (in binary) are
`00011`, `11111`, and `10101`. This is called *parsing* a number. To perform this task we use a low-level function named `parseInt`, which is shown in Fig 5.6.

The `parseInt` function uses a loop to take a numeric character, subtract the code for the character '0', and multiply by 10. For example, if the string is "21", it sees the '2' first. Subtracting, '2' - '0' we get a binary 2. This is then multiplied by 10, to produce 20, and we are ready for the next character on the next iteration of the loop.

The `parseInt` function expects the numeric string to be terminated by some non-numeric character. We use a separate function, `isNumeric` for this purpose. This function is shown in Fig 5.7.

```
######################  mnemonic function begin ##########################
#  Pre:
#  Register $a0 contains address of assemb stmt.
#  Register $a1 contains address of instruction.
#  Mnemonic 'end ' indicates end of source program text
#  Other parts of instruction have not yet been filled in.
#
#  Post:
#  Instruction opcode and function code are initialized.
#  Return register       $v0=0  => no error, instruction initialized
#                      $v0 > 0  => end of program
#                      $v0 < 0  => mnemonic not found in table.
#  Register $a0 will contain address of first non-white char after mnemonic
#  $a1 is unchanged


###### local data for mnemonic
       .data

ops_mnemonic:
       .asciiz       "add   "              # mnemonic
       .word              0x00000020       # opCode, function.
       .asciiz       "sub   "
       .word              0x00000022
       .asciiz       "and   "
       .word              0x00000024
       .asciiz       "or    "
       .word               0x00000025
       .asciiz "slt   "
       .word              0x0000002a
       .asciiz       "beq   "
       .word              0x10000000       # opCode = 4 (shift left 2 bits)
       .asciiz       "bne   "
       .word              0x14000000       # opCode = 5
       .asciiz       "j     "
       .word              0x08000000       # opCode = 2
       .asciiz       "end   "
       .word               -1
opsEnd_mnemonic:
       .text
mnemonic:
       addi     $sp, $sp, -20
       sw       $s0, 0($sp)
       sw       $s1, 4($sp)
       sw       $s2, 8($sp)
       sw       $s3, 12($sp)
       sw       $ra, 16($sp)
       move     $s3, $a1                     # address of instruction
```

Figure 5.4: Function to search a table for a mnemonic from the assembly language statement, and initialize the machine language instruction with an opcode and a function code (continued in Fig 5.5)

```
       move     $s3, $a1                    # address of instruction
       la       $s1, opsEnd_mnemonic             # end of mnemonic data below
       jal      skipCommaWhite
       move     $s0, $a0                    # address of mnemonic
       la       $a1, ops_mnemonic             # address of ops data below
       move     $s2, $a1                    # address of ops data below
lp_mnemonic:
       bge       $a1, $s1, error_mnemonic
       move      $a0, $s0                    # address of mnemonic
       jal       strcmp
       addi      $s2, $s2, 12                    # next op
       move      $a1, $s2
       bne       $v0, $0 lp_mnemonic

#  Found the mnemonic, or end
       lw        $t0, -4($s2)             # instruction
       addi      $t1, $0, -1             # -1 => end of program
       beq       $t0, $t1, end_mnemonic
       sw        $t0, 0($s3)             # found the mnemonic

       jal       skipCommaWhite
       move      $v0, $0
       j         done_mnemonic

end_mnemonic:
       addi      $v0, $0, 1                    # end of program
       j         done_mnemonic

error_mnemonic:
       addi      $v0, $0, -1

done_mnemonic:
       move      $a1, $s3
       lw        $ra, 16($sp)
       lw        $s3, 12($sp)
       lw        $s2, 8($sp)
       lw        $s1, 4($sp)
       lw        $s0, 0($sp)
       addi      $sp, $sp, 20
       jr        $ra

       .text
####################  mnemonic function end #######################
       .include "strcmp.asm"
```

Figure 5.5: Function to search a table for a mnemonic from the assembly language statement, and initialize the machine language instruction with an opcode and a function code (continued from Fig 5.4)

```
################################ function parseInt #######
# Pre:  address of numeric string is in $a0
#       its length is not 0.
#       string is terminated by a non-numeric char.
# Post: $v0 contains the resulting int
#       $a0 points to next char after the numerics.

## Local data

       .data
zero_parseInt:   .asciiz        "0"
ten_parseInt:    .word              10

       .text
parseInt:
       addi     $sp, $sp, -16
       sw       $ra, 0($sp)          # push return address onto stack
       sw       $s0, 4($sp)
       sw       $s1, 8($sp)
       sw       $s2, 12($sp)

       li       $s0, 0               # accumulate result
       lw       $s1, ten_parseInt    #  a1 = 10
       lbu      $s2, zero_parseInt
lp_parseInt:
       lbu      $a1, 0($a0)          # load byte from nums
       jal      isNumeric
       beq      $v0, $zero, done_parseInt
       mult     $s0, $s1             #result = result * 10
       mflo     $s0
       sub      $t1, $a1, $s2        # t1 = byte - '0'
       add      $s0, $s0, $t1        # result = result + int value of char
       addi     $a0, $a0, 1          # next char
       j        lp_parseInt          # repeat the loop
done_parseInt:
       move     $v0, $s0
       lw       $ra, 0($sp)          # pop return address from stack
       lw       $ra, 0($sp)          # push return address onto stack
       lw       $s0, 4($sp)
       lw       $s1, 8($sp)
       lw       $s2, 12($sp)
       addi     $sp, $sp, 16
        jr      $ra
###################### end function parseInt ############
       .include "isNumeric.asm"
```

Figure 5.6: Function to parse a numeric string, producing a 5-bit binary field

```
######################  Function isNumeric
# Function to test for numeric char
# Pre:   $a1 contains the char in low order byte
# Post:  $v0 = 1 if numeric, else $v0 = 0

## Local data
                     .data
zero_isNumeric:      .asciiz       "0"
nine_isNumeric:      .asciiz       "9"
###################################### End isNumeric function
                     .text
isNumeric:
       addi           $sp, $sp, -4
       sw             $ra, 0($sp)

       li              $v0, 0
       lbu             $t0, zero_isNumeric
       blt             $a1, $t0, done_isNumeric     # too small
       lbu             $t0, nine_isNumeric
       bgt             $a1, $t0, done_isNumeric     # too big

       li              $v0, 1                       # char is numeric
done_isNumeric:
       lw              $ra, 0($sp)
       addi            $sp, $sp, 4
       jr              $ra
###################################### End isNumeric function
                     .text
```

Figure 5.7: Function to determine whether a given character is numeric, '0'..'9'

```
######################  reg function begin ##########################
#  Pre:
#  Register $a0 contains address of reg number in assemb stmt.
#  Register number is one or two decimal digits.
#
#  Post:
#  Return register number in $v0
#                     $v0<0  =>  error
#  $a0 points to char after last digit of register number

reg:
       addi     $sp, $sp, -4
       sw       $ra, 0($sp)

       jal      parseInt
       li       $t0, 32
       blt      $v0, $t0, done_reg       # check for valid reg number
       li       $v0, -1
done_reg:
       lw       $ra, 0($sp)
       addi     $sp, $sp, 4
       jr       $ra
######################  reg function end ##########################
              .include "parseInt.s"
```

Figure 5.8: Function to scan a register number in a statement, and obtain its binary value

reg

We are now ready to process a register number in a statement. The function `reg`, shown in Fig 5.8, will accept the address of the register number in a statement, and return the register number, in binary, in register $v0. Thus, if the statement is

```
   add $31, $20, $20
```

and register $a0 points to the '3' then the binary value, 31, will be returned in register $v0.

operand

We next consider a function to process a register operand in an R format statement. The operand could be any of the three operands: RD, RS, or RT. One of the inputs to the function will determine which of the three operands is being processed. The `operand` function will convert the register number to binary, and place it in the correct field of the instruction.

This function is shown in Figs 5.9 and 5.10, in which the API specifies a code in register $a2 to specify which operand is being processed. This code is actually a shift amount, to place the operand in the correct field of the machine language instruction. For example, if we are processing the RT register, then $a2 will contain 16. This is used to shift the register number 16 bits left, which is where it is to be placed in the instruction. To do this we use a *variable shift* instruction, `sllv`, in which the third operand is a register containing the number of bits to be shifted. Once it has been shifted it can be ORed into the instruction.

operandRD, operandRS, lineEnd, and operandRT

We now turn our attention to the functions which place a register operand into the instruction. There are three such functions, one for each of the operands, RD, RS, and RT. Since the RT operand comes last in the statement, we will also need function to check for the end of a line - `lineEnd`.

The `operandRD` function is shown in Fig 5.11. It calls the `operand` function, with the shift amount, 11, in register $a2, which will place the operand into bits 11..15 of the instruction. It then calls the function `skipCommaWhite` to move to the next operand in the statement.

The `operandRS` function is similar to the `operandRD` function; the only difference is that it uses a shift amount of 21 rather than 11, in order to place the operand into bits 21..25 of the instruction.

The `operandRT` function is similar to the two previous functions, but after processing the RT operand, the assembler should scan to the end of the line, in preparation for the next statement (we are assuming at this point that there are no comments to be scanned). Thus instead of calling the `skipCommaWhite` function we call the function `lineEnd`, shown in Fig 5.13, which scans to the start of the next line. The `operandRT` function is shown in Fig 5.12. If the assembly language program is a sequence of strings in memory, each statement is terminated by a null byte. However, if the program is a sequence of lines in a text file, each line is terminated by a newline character. Our `lineEnd` function will handle either of these cases.

asm

Finally, we have all the tools we need to process an assembly language statement, and produce the machine language instruction (for a limited subset of the MIPS architecture). We call this function `asm`, for *assembler*.

The `asm` function is shown in Fig 5.14. This function is fairly short; all the real work is done in other functions which are called by this function, specifically:

1. Call the `mnemonic` function to process the mnemonic

2. Call the `operandRD` function to process the RD operand

3. Call the `operandRS` function to process the RS operand

```
######################  operand function begin ##########################
#  Pre:
#  Register $a0 contains address of $regNumber in assemb stmt.
#  Register $a1 contains address of instruction.
#  Register $a2 contains shift amount, to place register address in instr
#                rs:        shamt = 21
#              rt:        shamt = 16
#              rd:       shamt = 11
#
#  Post:
#  Return register $v0=address of syntax error
#                     $v0=0  => no error
#  rd, rs, or rt register entered in instruction
#  Register $a0 will contain address of first non-white char after reg
#  $a1 is unchanged


###   local data
                      .data
dollar_operand:       .asciiz       "$"

       .text
operand:
       addi     $sp, $sp, -12
       sw       $s0, 0($sp)
       sw       $s1, 4($sp)
       sw       $ra, 8($sp)

       lbu      $t0, dollar_operand            #  ’$’
       lbu      $t1, 0($a0)                    #  address of ’$’ in statement
       bne      $t0, $t1, error_operand
```

Figure 5.9: Function to place an operand (i.e. register number) into the machine language instruction (continued in Fig 5.10)

```
       move     $s0, $a1                 # address of instruction
       move     $s1, $a2                 # shift amount
       addi     $a0, $a0, 1              # address of reg number
       jal      reg                      # get reg number
       blt      $v0, $0, error_operand
       sllv     $v0, $v0, $s1            # shift to appropriate field
       lw       $t0, 0($s0)              # instruction
       or       $t0, $t0, $v0            # rd into instruction
       sw       $t0, 0($s0)              # save instruction
       jal      skipCommaWhite
       move     $v0, $0
       j        done_operand

error_operand:
       move  $v0, $a0

done_operand:
       move     $a1, $s0
       lw       $s0, 0($sp)
       lw       $s1, 4($sp)
       lw       $ra, 8($sp)
       addi     $sp, $sp, 12
       jr       $ra

       .text
##########  operand function end  ########################################
       .include "reg.s"
       .include "skipCommaWhite.s"
```

Figure 5.10: Function to place an operand (i.e. register number) into the machine language instruction (continued from Fig 5.9)

```
######################  operandRD function begin ##########################
#  Pre:
#  Register $a0 contains address of $rd reg in assemb stmt.
#  Register $a1 contains address of instruction.
#
#  Post:
#  Return register $v0=address of syntax error
#                     $v0=0  => no error
#  RD register entered in instruction
#  Register $a0 will contain address of first non-white char after reg
#  $a1 is unchanged

operandRD:
       addi     $sp, $sp, -4
       sw       $ra, 0($sp)

       move     $s0, $a1
       li       $a2, 11                     # shift amount for rd
       jal      operand
       bne      $v0, $0, error_operandRD
       jal      skipCommaWhite
       beq      $v0, $0, done_operandRD
error_operandRD:
       move     $v0, $a0
done_operandRD:
       move     $a1, $s0

       lw       $ra, 0($sp)
       addi     $sp, $sp, 4
       jr       $ra
##########  operandRD function end  ########################################
       .include "operand.asm"
```

Figure 5.11: Function to place the RD operand (i.e. register number) into the machine language instruction

```
######################  operandRT function begin ##########################
#  Pre:
#  Register $a0 contains address of $rt reg in assemb stmt.
#  Register $a1 contains address of instruction.
#
#  Post:
#  Return register    $v0=0  => no error
#  RT register entered in instruction
#  Register $a0 will contain address of next statement

operandRT:
       addi     $sp, $sp, -8
       sw       $s0, 0($sp)
       sw       $ra, 4($sp)

       move     $s0, $a1
       addi     $a2, $0, 16       # shift amount for rt
       jal      operand
       bne      $v0, $0, error_operandRT
       jal      lineEnd           # scan to next statement
       j        done_operandRT
error_operandRT:
       move     $v0, $a0

done_operandRT:
       move     $a1, $s0
       lw       $s0, 0($sp)
       lw       $ra, 4($sp)
       addi     $sp, $sp, 8
       jr       $ra
##########  operandRT function end  ########################################
       .include "lineEnd.s"
```

Figure 5.12: Function to place the RT operand (i.e. register number) into the machine language instruction and scan to the beginning of the next statement

```
################### lineEnd function begin ####################
#  Pre:  $a0 contains address of current position in the statement
#              after all operands have been scanned.  Remainder of
#              stmt should be white space
#              Each stmt ends with null byte or newline char

#  Post: $a0 contains address of first byte in next statement
#              v0 is neg => error, non whitespace encountered before end
#              v0 is 0   => ok, proceed to next stmt
                   .data
newline_lineEnd:   .ascii "\n"
                   .text
lineEnd:
       addi     $sp, $sp, -4
       sw       $ra, 0($sp)

       jal      skipCommaWhite
       li       $v0, 0
       lbu      $t0, 0($a0)                  # load byte from input string
       beq      $t0, $zero, done_lineEnd     # null byte, end of stms
       lbu      $t1, newline_lineEnd
       beq      $t0, $t1, done_lineEnd       # newline character, end of stmt

       li       $v0, -1                      # error, some nonwhite char after operands

done_lineEnd:
       addi     $a0, $a0, 1
       lw       $ra, 0($sp)
       addi     $sp, $sp, 4
       jr       $ra
######################## lineEnd function end  ##################
                .text
```

Figure 5.13: Function to scan to the end of a line, to the start of the next statement in the assembly language program

4. Call the `operandRT` function to process the RT operand

The program repeats the above sequence once for each statement in the assembly language program.

We also show a driver for the `asm` function; this will tell us whether our assembler is working correctly. The driver is shown in Fig 5.15. In the driver there are four statements after the `syscall` statement which terminates the program. Since there are no labels on those four statements, they can never be reached during execution. Nevertheless they will be useful. They are the same four statements which we have placed in memory at the label `input_asmDriver`. We can then compare the machine language instructions produced by our assembler with the instructions produced by MARS, at the end of the driver.

At this point we emphasize the fact that the assembler function, `asm`, is relatively short and simple, because it simply calls other functions, none of which is excessively long or complex. This exposes a few important principles of software engineering:

- Use many small and simple components (functions in this case) as opposed to a few large and complex components.

- Design the low-level components appropriately so as to simplify the implementation of higher-level components.

- Be sure that the interfaces[4] are appropriate, and are clearly documented in the API for each component.

5.1.2 Version 1b - Allow Symbolic Registers

In what we have done thus far, we are restricting the assembly language statemens to use register *numbers*, such as $1, $8, $31, only. Register *names*, such as $at, $t0, and $ra have not been allowed. In version 1b of the assembler we accommodate the register names, i.e. symbolic registers, and translate them into the appropriate register numbers.

To accommodate register names, we will need to search a table of register names for the name provided in the statement. This is best done in the `reg` function, shown in Figs 5.16 and 5.17, in which the table shows all 32 register names, in order. When searching the table in a loop, the loop counter will give us the appropriate register number. Note that the `reg` function still needs to accommodate register numbers; to do this it calls `isNumeric` to determine whether the character after the dollar sign is a numeric character.

We can now modify the driver shown in Fig 5.15 to include register names in a statement, for example:

```
    add $v0, $v1, $a0
```

[4] By *interface* here we mean the pre and post conditions for a function.

```
######################  asm function begin ##########################
#  Translate lines of assembly language to machine code.
#  Version 1a.  (See Driver for specs)

#  Pre:  $a0 contains the memory address of the first line of asm code
#        Last asm statement followed by -1.
#        $a1 contains memory address for output.
#  Post: $v0<0   =>  syntax error
#        $a0 will contain address of error

asm:
       addi      $sp, $sp, -4
       sw        $ra, 0($sp)
lp_asm:
       jal       mnemonic
       bne       $v0, $0, done_asm              # end of source program
       jal       operandRD
       bne       $v0, 0, done_asm
       jal       operandRS
       bne       $v0, 0, done_asm
       jal       operandRT
       bne       $v0, 0, done_asm
       addi      $a1, $a1, 4                    # next instruction
       blt       $v0, $0, done_asm              # error
       beq       $v0, $0, lp_asm

done_asm:
       lw        $ra, 0($sp)
       addi      $sp, $sp, 4
       jr        $ra

#####################  asm function end ####################
       .include        "mnemonic.asm"
       .include       "operandRD.asm"
       .include       "operandRS.asm"
       .include       "operandRT.asm"
```

Figure 5.14: Function to scan an assembly language program and create the corresponding machine language program (version 1a)

```
# testing assembler
# version 1a:   R format instructions only (no shift)
#               No symbolic memory addresses
#               No symbolic register addresses
#               'end' mnemonic terminates input
#               Limited error checking
#               No comments nor Pseudo-ops
                .data
input_asmDriver:
              .asciiz        "    add    $2,$3,$4   "
              .asciiz        "    sub    $12, $2, $3"
              .asciiz        "         and       $9, $10,$11"
              .asciiz        " or       $21, $22, $23 "
              .asciiz        " end    "

output_asmDriver:
              .word          0, 0, 0, 0
returnCode:   .word          -1

              .text
main:
       la       $a0, input_asmDriver
       la       $a1, output_asmDriver
       jal      asm
       sw       $v0, returnCode

       li       $v0, 10
       syscall

       add      $2,$3,$4
       sub      $12, $2, $3
       and      $9, $10,       $11
       or       $21, $22, $23

       .include  "asm.s"
```

Figure 5.15: Driver to test the `asm` function (version 1a)

```
######################  reg function begin ##########################
#  Pre:
#  Modified for version 1b.  Permit symbolic registers.
#  Register $a0 contains address of reg number or reg name in assemb stmt.
#
#  Post:
#  Return register number in $v0
#                     $v0<0  =>  error
#  $a0 points to char after last char of register number or name
#  $a1 points to instr being written out


              .data
              .word               -1
regNames_reg:  .asciiz             "zero   "
              .asciiz             "at     "
              .asciiz             "v0     "
              .asciiz             "v1     "
              .asciiz             "a0     "
              .asciiz             "a1     "
              .asciiz             "a2     "
              .asciiz             "a3     "
              .asciiz             "t0     "
              .asciiz             "t1     "
              .asciiz             "t2     "
              .asciiz             "t3     "
              .asciiz             "t4     "
              .asciiz             "t5     "
              .asciiz             "t6     "
              .asciiz             "t7     "
              .asciiz             "s0     "
              .asciiz             "s1     "
              .asciiz             "s2     "
              .asciiz             "s3     "
              .asciiz             "s4     "
              .asciiz             "s5     "
              .asciiz             "s6     "
              .asciiz             "s7     "
              .asciiz             "t8     "
              .asciiz             "t9     "
              .asciiz             "k0     "
              .asciiz             "k1     "
              .asciiz             "gp     "
              .asciiz             "sp     "
              .asciiz             "fp     "
              .asciiz             "ra     "
regNamesEnd_reg: .word             -1
              .asciiz             ""
```

Figure 5.16: Function to scan a register name or number in a statement, and obtain its binary value (continued in Fig 5.17)

```
              .text
reg:
       addi     $sp, $sp, -20
       sw       $ra, 0($sp)
       sw       $s0, 4($sp)
       sw       $s1, 8($sp)
       sw       $s2, 12($sp)
       sw       $s3, 16($sp)

       move     $s3, $a1       # copy of pointer to instr
       lbu      $a1, 0($a0)       # first char of reg num or name
       jal      isNumeric
       bne      $v0, $zero, num_reg

       # reg is specified by a name
       la       $a1, regNames_reg       # First word in table
       move     $s0, $a1              # First word in table
       la       $s1, regNamesEnd_reg
       move     $s2, $a0
lp_reg:
       bge      $s0, $s1, error_reg       # end of table?
       jal      strcmp                     # Reg name found in table?
       beq      $v0, $zero, foundName_reg
       addi     $s0, $s0, 8              # Next row of table
       move     $a1, $s0
       move     $a0, $s2
       j        lp_reg
error_reg:
       li       $v0, -1                     # error code
       j        done_reg
foundName_reg:
       la       $t0, regNames_reg
       sub      $v0, $s0, $t0              # bytes into the table
       srl      $v0, $v0, 3              # 8 bytes per row
       j        done_reg
num_reg:
       jal      parseInt
       li       $t0, 31
       bgt      $v0, $t0, error_reg       # reg number should be 0..31
done_reg:
       move     $a1, $s3              # reload address of instr
       lw       $ra, 0($sp)
       lw       $s0, 4($sp)
       lw       $s1, 8($sp)
       lw       $s2, 12($sp)
       lw       $s3, 16($sp)
       addi     $sp, $sp, 20
       jr       $ra
######################  reg function end ##########################
              .include "parseInt.asm"
```

Figure 5.17: Function to scan a register name or number in a statement, and obtain its binary value (continued from Fig 5.16)

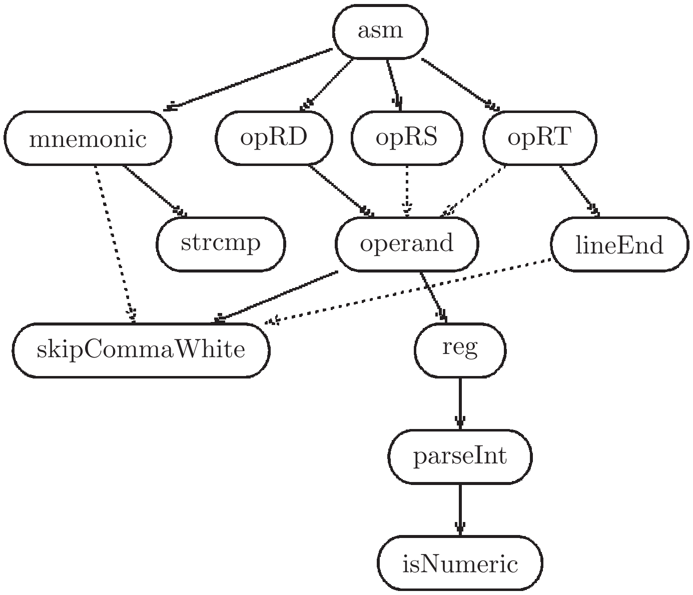


Figure 5.18: Call graph for version 1a of the assembler. Solid arrows represent a function call, with an `.include` directive. Dashed arrows represent function calls with no `.include` directive. `operandRD`, `operandRS`, and `operandRT` have been abbreviated to save space.

5.1.3 Include Directives

Most of the functions we have seen have used `.include` directives at the bottom to include source code from other files, typically for functions which are being called. As we mentioned earlier, we need to be careful not to include the same source file twice - this will give us multiply defined symbols, i.e. the same symbol defined twice. For example, the `operandRS` function calls the `operand` function; however, it would be a mistake for the `operandRS` function to include the `operand` function. The `operand` function has already been included by the `operandRD` function and if we include the `operand` function twice, we will get error messages: *multiply defined symbols*, i.e. duplicate definitions of the same symbol.

Fig 5.18 is a diagram showing which functions are called by other functions and have a `.include` directive for the called function (solid arrows). It also shows function calls for which the called function is *not* included, to avoid multiply defined symbols (dashed arrows). This diagram makes it clear that each source file is included exactly once. There is exactly one solid arrow leading to each function.

This diagram is especially useful when writing drivers for the various functions - it tells you when to include, and when not to include, the source file for a function called from the driver.

5.1.4 Exercises

1. Write a driver for each of the following functions:

   (a) `strcmp`

   (b) `mnemonic`

   (c) `isNumeric`

   (d) `parseInt`

   (e) `reg` (version 1a)

   (f) `reg` (version 1b)

   (g) `operand`

   (h) `operandRD`

   (i) `operandRS`

   (j) `operandRT`

   (k) `lineEnd`

2. Extend the driver for the assembler (Fig 5.15) to include at least three other statements. Compare your results with the program produced by the MARS assembler.

3. Extend the mnemonic table (Fig 5.4) to include the operations `xor`, `mult`, `div`, `mflo`, and `mfhi`,

4. Improve the `lineEnd` function to allow comments at the end of a statement.

5. Include the shift instructions `sll` and `srl` in your mnemonic table. You will need to implement changes to `mnemonic` , `asm` , `operandRT` , and `lineEnd` . Also write a function, `shamt` to process the shift amount instead of the RS register.
   Hint: This is tricky because the shift amount replaces the RS operand in the instruction, so the second operand in the statement is the RT register.

6. How would the call graph shown in Fig 5.18 be changed for version 1b of the assembler (i.e. register names are permitted)?

5.2 Version 2 - Include I and J Format Instructions

In this section we develop version 2 of our assembler. In this version we will introduce I and J format instructions; we will also introduce some simple pseudo operations, such as `move` and `li`.

All the code for this version is shown in appendix **??**.

5.2.1 Version 2a - I and J Format Instructions

In this section we will introduce the following instructions in our assembler:

1. Simple I (immediate) format instructions, such as `addi`, `andi`, and `ori` (but no memory reference instructions, nor branch instructions)

2. J (jump) format instructions, such as `j` and `jal`

3. Conditional branch instructions - `beq` and `bne`.

Most of the changes will be in our `mnemonic` function, which determines the instruction op code. We now include a one byte value in the table of instruction mnemonics to tell us whether the instruction is R, I, or J format. We use this as the return code for the mnemonic function.

We then make use of this return code in the `asm` function. If the statement is I format, it will process the $rt and $rs fields. Then instead of a $rd field (and shamt and function code fields) it will process the immediate field and store it directly in the low order 16 bits of the instruction. In doing so, the `asm` function will call the `parseInt` function to convert the immediate field from ascii to a binary 16-bit value. We used the `parseInt` function in version 1 to convert a register number to a binary value. Now we must update this function to accept negative numbers as well as positive. To do so, we merely check for a minus sign ('-') at the beginning. If so, we negate the returned result.

This version of our assembler is shown in Appendix **??**

5.2.2 Version 2b - Explicit Memory Addresses

In this version of our assembler we implement memory reference instructions with *explicit* addressing only. We also implement a few simple (one-to-one) pseudo operations, such as `move` and `li`.

Because we are now handling pseudo operations, we will now make our assembler a *two-pass* assembler. This means it will scan through the source file *twice*. On the first pass it will convert all pseudo-ops to actual instructions. Then the second pass will translate to machine code.

The first pass will also be useful when we introduce symbolic instruction addresses for branching and jumping. A forward branch or jump will need to know the instruction memory address of the target. We can build a table of these symbols on the first pass, and generate the machine code on the second pass.

Load and store with explicit addresses

We are working with two instructions which reference memory: `lw` (load word) and `sw` (store word). In this version of our assembler we permit explicit (i.e. non-symbolic) memory addresses only.

Thus we permit instructions such as:

pseudo op	Example	Equivalent
move	move \$t0,\$s3	add \$t0, \$0, \$s3
li	li \$a0, -17	addi \$a0, \$0, -17

Figure 5.19: Two simple pseudo operations, and their equivalent instructions

```
        lw      $t0, 0($sp)
        sw      $s3, -12($a0)
```

These are I (immediate) format instructions, in which the \$rt register is the register being loaded or stored, and the effective memory address is the sum of the \$rs register and the immediate field. To handle these changes we will need to work with the second pass of our assembler which we now call `asmPass2`. This function will call a function which determines the operation code, which we now call `mnemonic2`. It will determine the instruction type and return values as shown below:

- \$v0 = 0: R format
- \$v0 = 1: I format, not memory ref
- \$v0 = 2: I format, memory ref
- \$v0 = 3: J format

If the instruction is a memory reference instruction, we handle the \$rt register as in previous versions. Then we come to the explict address: `imm($rs)` We handle this in a function called `explicitAdd`. In this function we process the immediate field, with a call to the `parseInt` function (which now handles negative values), and store the low order 16 bits into the instruction. Then we process the register in parentheses, with a call to the `reg` function.

One-to-one pseudo operations

In this version we also introduce some simple pseudo operations: those which correspond to a single machine language instruction, such as `li` and `move`. Each of these pseudo operations can be replaced by one actual instruction, as shown in Fig 5.19.[5]

In each of these instructions, all we need to do is substitute an instruction, `add` for `move` and `addi` for `li` , then insert `$0` as the `$rt` register. This is done in the first pass of the assembler, function `asmPass1`, which writes the statements to a memory buffer called 'basic'. The second pass then reads the statements from the `basic` buffer.

[5] MARS actually uses `addu` and `addiu`, which are the *unsigned* versions of `add` and `addi`. We suspect this is done to avoid setting an overflow flag in the CPU.

5.2.3 Exercises

1. Include a `clear` pseudo operation in version 2b of the assembler. It will have just one operand, the register to be cleared. For example, the statement `clear $t0` will put the value 0 into register $t0.
   Hint: Use an `add` or `addi` instruction to clear the operand.

2. Does the assembler permit white space in the middle of an explicit address? For example, is the following handled correctly?
   `lw  $t0, 12 (    $a0 )` If not, make the necessary modifications to version 2b of the assembler.

3.

5.3 Version 3 - More Pseudo Operations

Chapter 6

Boolean Algebra and Digital Logic

This chapter begins our discussion of digital hardware. We start with some basic theory of boolean algebra. We will then show how boolean functions may be realized using some elementary building blocks - *logic gates*. We then use these to build more complex components, which are then used to build more complex components, and so on. Our goal is to build a small *central processing unit*, or CPU. At this point the student should be able to understand the execution of a machine language program. Various technologies are used to build the logic gates, but this text will treat the gates as atomic components; i.e. we will work from the gate level and up.

In what follows, a *true* value is represented by a binary 1, and a *false* value is represented by a binary 0, which is consistent with chapter 3.

6.1 Notation for Boolean Functions

At this point we introduce a more convenient notation for boolean functions. In previous chapters we used the operations AND, OR, and NOT. As we work with more complex boolean functions this notation becomes cumbersome, so we adopt a notation which is taken from algebra: A plus is used for OR, a raised dot is used for AND (the dot is sometimes omitted), and a quote mark is used for NOT:

x AND y = $x \cdot y$ = xy
x OR y = x + y
NOT x = x'

The x' is read "x prime".[1] The reader is cautioned not to confuse these operations with their arithmetic counterparts - addition and multiplication.

[1] Some textbooks use $\overline{x}$ instead of x', and it is read "x bar" or "not x".

x	y	xy + x	term
0	0	0	
0	1	0	
1	0	1	xy’
1	1	1	xy

Figure 6.1: Table representation of the boolean function xy + x.

Hopefully the context of boolean functions will always make it clear that we are working with booleans and not with numbers.

In an expression involving more than one operation, the order of operations is important. Parentheses may be used to specify the order of operaions. If parentheses are omitted, the NOT operation takes precedence over the AND operation:

$x \cdot y' = x \cdot (y')$

The AND operation takes precedence over the OR operation, consistent with algebra:

$x + y \cdot z = x + (y \cdot z)$

Thus we could write more complex expressions such as the one below:

$x + yz' + x'z$

which is equivalent to:

$(x + (y \cdot (z'))) + ((x') \cdot z)$

The Exclusive OR operation is designated by a $\oplus$ symbol. Its precedence level is the same as the +. When an expression contains two or more operations of the same precedence, the leftmost operation is performed first. For example:

$x + y \oplus z + w = ((x + y) \oplus z) + w$

6.1.1 Boolean Expressions

A boolean function may be specified by an expression, as we did previously, or it may be specified with a table. For example, the boolean function xy + x is shown in Fig 6.1, and the boolean function xy + x’yz’ is shown in Fig 6.2. For now, the reader may ignore the last column, labeled `term`, in these tables. Note that a function with two variables results in a table with 4 rows, a boolean function with three variables results in a table with 8 rows, and, in general, a boolean function with n variables results in a table with 2^n rows. This table is called a *truth table*.

Any boolean function can be implemented with an appropriate combination of logic gates.

Canonical Forms

The student may have noticed that in Fig 6.1 the entries in the column for x are the same as the entries in the column for xy+x. Thus we have the identity, for any variables, x and y:

xy + x = x

x	y	z	xy + x'yz'	term
0	0	0	0	
0	0	1	0	
0	1	0	1	x'yz'
0	1	1	0	
1	0	0	0	
1	0	1	0	
1	1	0	1	xyz'
1	1	1	1	xyz

Figure 6.2: Table representation of the boolean function xy + x'yz'.

It should now be clear that there is more than one expression for a given boolean function. There are many areas of computer science where there exist multiple representations for an entity. For example, in chapter 4 we saw multiple representations (mantissa and exponent) of a floating point number. In such situations we often wish to designate exactly one of those representations as preferred over the others. This is known as a *canonical form* or a *normal form*.

For any boolean expression we have a canonical form which is known as a *sum of products* or *disjunctive normal form*[2]. The sum of products normal form can be obtained from a truth table as follows (here we assume the variables are x, y, and z):

1. In each row of the truth table which has a 1 for the function's value, include a term involving an AND of all inputs, but for those inputs which have a 0 value in that row, negate the variable. For example, in the row for 0 1 0, the term would be x'yz'. These terms are shown in the last column of Figs 6.1 and 6.2.
2. Form the OR of all the above terms.

For the function of Fig 6.1 the sum of products normal form would be xy' + xy. For the function of Fig 6.2 the sum of products normal form would be x'yz' + xyz' + xyz.

6.1.2 Minimizing Boolean Expressions

In the previous section we saw that there may be several different boolean expressions for the same boolean function. In this section we explore this idea further.

Consider the boolean function given by the expression:
x'yz' + xyz' + xyz
If we were to implement this function in assembly language (or to construct hardware to implement it), we would need to perform six AND operationss,

[2] There is also a product of sums, or conjunctive normal form, which we do not discuss here.

Name	Identity	Dual identity
Identity	$x + 0 = x$	$x \cdot 1 = x$
Negation	$x + x' = 1$	$x \cdot x' = 0$
Idempotent	$x + x = x$	$x \cdot x = x$
Constant	$x + 1 = 1$	$x \cdot 0 = 0$
Involution	$x'' = x$	
Commutative	$x + y = y + x$	$x \cdot y = y \cdot x$
Associative	$x + (y + z) = (x + y) + z$	$x \cdot (y \cdot z) = (x \cdot y) \cdot z$
Distributive	$x(y + z) = xy + xz$	$x + yz = (x + y)(x + z)$
DeMorgan	$(x + y)' = x'y'$	$(xy)' = x' + y'$
Absorption	x + xy = x	$x(x + y) = x$
Absorption	x + x'y = x + y	$x(x' + y) = xy$
Consensus	xy + x'z + yz = xy + x'z	(x+y)(x'+z)(y+z) = (x+y)(x'+z)

Figure 6.3: Boolean algebraic identites

two OR operations, and two NOT operations. However since we know that the expression represents the same function as xy + x'yz' we can implement the function with only three AND operations, one OR operation, and three NOT operations.

As we will see later, minimizing a boolean expression leads to reduced hardware requirements, and consequently reduced production costs. For that reason we will be interested in finding ways of finding a minimal expression for a boolean function.

Boolean identities

One way to minimize a boolean expression is by applying various identities, such as x + xy = x. This identity, and many more, are shown in Fig 6.3[3]. For each identity, there is a corresponding *dual* identity, shown in the last column. The dual is obtained by changing all AND operations to ORs, changing all OR operations to ANDs, changing all 1's to 0's, and changing all 0's to 1's. Note in particular that, unlike arithmetic algebra, the OR operation distributes over the AND operation: x + yz = (x+y)(x+z).

As an example we take the boolean expression x'yz + x'yz' + xyz' and we attempt to find an equivalent expression which is, in some sense, minimal. Below we show each step in the derivation, with the algebraic identity as justification:

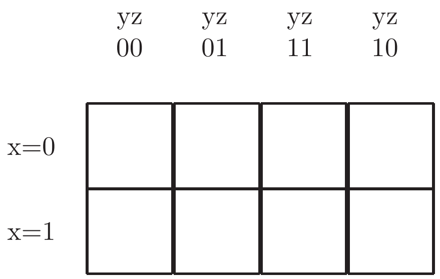
```
1.  x'yz + x'yz' + xyz'       Given

2.  x'(yz + yz') + xyz'       Distributive

3.  x'y(z+z') + xyz'          Distributive

4.  x'y(1) + xyz'             Negation
```

[3] In some cases the raised dot representing an AND operation has been omitted.



	yz 00	yz 01	yz 11	yz 10
x=0				
x=1				

Figure 6.4: An empty three variable K-map

5.	`x’y + xyz’`	`Identity`
6.	`yx’ + yxz’`	`Commutative`
7.	`y(x’ + xz’)`	`Distributive`
8.	`y(x’ + z’)`	`Absorption`

As far as we know there are no algorithms which can be applied to minimize a boolean expression using the given identities; we must rely on our own ingenuity.

Karnaugh maps with three variables

One common method for minimizing boolean expressions is known as the *Karnaugh Map*, or K-map, introduced by Maurice Kaurnagh in 1953 as a two-dimensional table.

We begin by looking at boolean functions with three variables. The table conssistes of two rows and four columns; for example, if the names of the variables are x, y, and z. then the (empty for now) map is shown in Fig 6.4. Note that the sequence of values for the columns are not what you might expect.[4]

We should now be able to fill in the cells of the K-map. Fig 6.5 shows the completed map for the boolean function x’y’z’ + x’yz + x’yz’ + xy’z’. Note that a 1 is placed in the map corresponding to each term. If a variable is negated in the expression, it’s position in the map corresponds to a 0 value. For example, the term xy’z’ is the term for which x=1, y=0, and z=0, so we place a 1 in row 1, column 00.

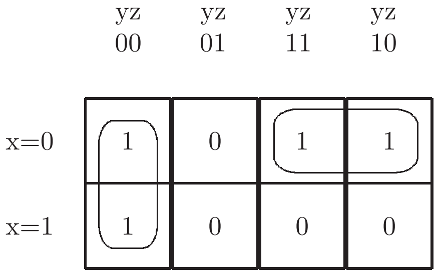
After we have filled in a 1 for each term in the expression, we need to group them in rectangular boxes. Two adjacent 1’s will be grouped in a 1x2 block, or in a 2x1 block, and 4 adjacent 1’s may be grouped in a 1x4 block or a 2x2 block. In our example we have a 1x2 block and a 2x1 block, as shown in Fig 6.6.

Each block of 1’s in the K-map will correspond to a term in the minimized expression. In a block of 1’s look at the variable(s) which correspond to equal values. Retain only those variables in a term of the boolean expression. Negate

[4] The sequence {00, 01, 11, 10} is known as a *Gray Code* sequence, in which exactly one bit is changed from one value to the next.

	yz 00	yz 01	yz 11	yz 10
x=0	1	0	1	1
x=1	1	0	0	0

Figure 6.5: A K-map for the boolean expression x'y'z' + x'yz + x'yz' + xy'z'



	yz 00	yz 01	yz 11	yz 10
x=0	1	0	1	1
x=1	1	0	0	0

Figure 6.6: A K-map for the boolean expression x'y'z' + x'yz + x'yz' + xy'z'. Two groups of two are identified. The minimized expression is y'z' + x'y.

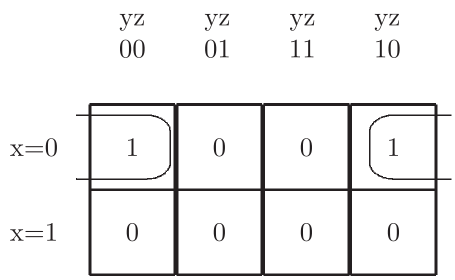
the variables for which the value is zero. For example, in Fig 6.6 there is a vertical group in the column for y=0, z=0. This means we will have the term y'z' in our result. There is also a horizontal group in the row for x=0 and the columns for which y=1. This means that we will have the term x'y in our result. Thus, our minimized expression is y'z' + x'y.

Next we attempt to minimize the boolean expression x'y'z' + x'yz'. There are only two 1's in the map, but they are actually adjacent, because the groups may *wrap around* from end to end, as shown in Fig 6.7. Thus the resulting term will have x=0 and z=0, producing x'z' as the minimal expression.

We can also have a 2x2 block of 1's in a K-map, and the blocks may overlap with other blocks to produce a minimal expression. Our example, shown in Fig 6.8, is the function x'yz + x'yz' + xy'z + xyz + xyz'. In this case we have a 2x2 block (y) and a 1x2 block (xz) which overlap, yielding the minimized expression y + xz.

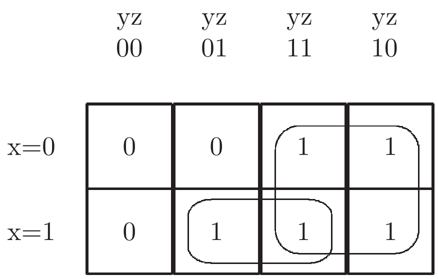
Karnaugh maps with four variables

Karnaugh maps can be extended to handle four variables; we will have four rows instead of two, as shown in the empty map in Fig 6.9. The variables are w,x,y, and z. The row labels for w and x follow the Gray code sequence {00,01,11,10}, consistent with the column labels for y and z.



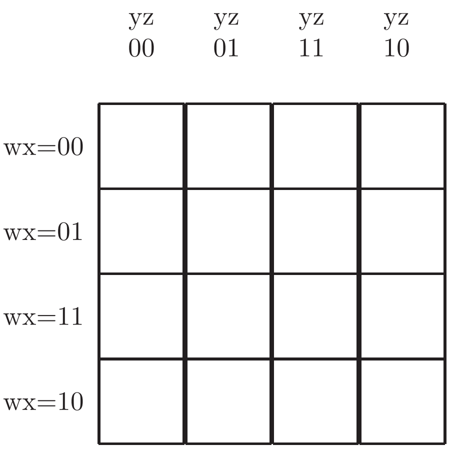
	yz 00	yz 01	yz 11	yz 10
x=0	1	0	0	1
x=1	0	0	0	0

Figure 6.7: A K-map for the boolean expression x'y'z' + x'yz'. The group wraps around from end to end. The minimized expression is x'z'.



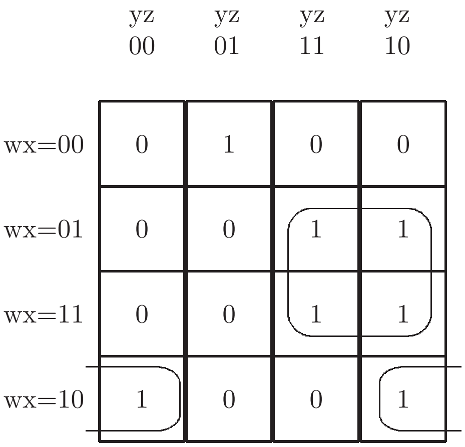
	yz 00	yz 01	yz 11	yz 10
x=0	0	0	1	1
x=1	0	1	1	1

Figure 6.8: A K-map for the boolean expression x'yz + x'yz' + xy'z + xyz + xyz'. The 2x2 block overlaps the 1x2 block. The minimized expression is y + xz.



	yz 00	yz 01	yz 11	yz 10
wx=00				
wx=01				
wx=11				
wx=10				

Figure 6.9: An empty K-map with four variables



	yz 00	yz 01	yz 11	yz 10
wx=00	0	1	0	0
wx=01	0	0	1	1
wx=11	0	0	1	1
wx=10	1	0	0	1

Figure 6.10: A K-map for the boolean expression w'x'y'z + w'xyz + w'xyz' + wxyz + wxyz' + wx'y'z' + wx'yz'. A 1x2 group and a 2x2 group are identified. The minimized expression is xy + wx'z' + w'x'y'z

The technique for minimizing a boolean function with four variables is very much like the technique used for three variables. However, with four variables we can now have 4x1 blocks, 2x4 blocks, and 4x2 blocks. We use the same rules for blocking the 1's and determining the terms of the minimized expression. Note that blocks can overlap, as with three variable maps, and that blocks can now wrap around vertically as well as horizontally.

As an example, we take the boolean function of four variables given by w'x'y'z + w'xyz + w'xyz' + wxyz + wxyz' + wx'y'z' + wx'yz'. The K-map is shown in Fig 6.10. Note that we have a 2x2 group (the term is xy) and a 1x2 group, wrapping around horizontally (the term is wx'z'). There is also a 1 which is not adjacent to any others, corresponding to the term w'x'y'z. That term will have to appear in the final result, which is xy + wx'z' + w'x'y'z.

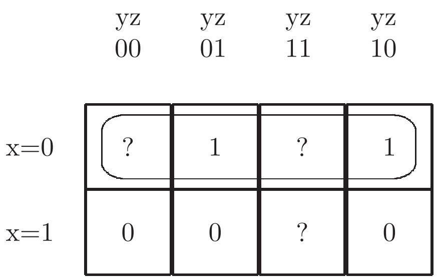
Don't cares

Up to this point all boolean expressions have been completely specified (the truth tables show a 0 or 1 for every possible row). There are often situations in which the problem does not require a complete specification, i.e. for some inputs we do not care what the output is. In many applications certain inputs are not expected, or disallowed, making the corresponding output irrelevant. In this case we call the output a *don't care* - it may be either a 0 or a 1. When minimizing the boolean expression, it is possible to improve the minimization by making use of don't cares.

For example, consider the truth table in Fig 6.11. The don't care outputs are shown as question marks. A canonical sum of products expression for this boolean function is x'y'z + xyz'. Looking at the K-map in Fig 6.12, at first

x	y	z	f(x,y,z)
0	0	0	?
0	0	1	1
0	1	0	1
0	1	1	?
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	?

Figure 6.11: Truth table defining a boolean function with 3 don't cares (shown as question marks)



	yz 00	yz 01	yz 11	yz 10
x=0	?	1	?	1
x=1	0	0	?	0

Figure 6.12: A K-map for the boolean function whose truth table is given in Fig 6.11 (question marks are don't cares)

glance it appears that it cannot be simplified. But if we assume that the two don't cares in the top row are 1's and the other don't care is a 0, then we have a group of 4 1's in the top row, as shown in Fig 6.12, and this boolean function simplifies to f(x,y,z) = x'.

6.1.3 Exercises

1. Show a truth table corresponding to each of the following boolean expressions:

   (a) x + yz'

   (b) xy + x'yz' + yz

   (c) x (y' + xz)

2. Show the disjunctive normal form (sum of products) expression for the boolean function corresponding to each of the following expressions:

   (a) x + yz'

   (b) xy + x'yz' + yz

   (c) x (y' + xz)

Hint: Form a truth table first; there will be one term for each 1 in the truth table.

3. Show how to minimize each of the following boolean expressions using the identities given in Fig 6.3. Justify each step in your derivation by naming the identity which is used.

   (a) x'y'z + x'yz + x'yz' + xyz'

   (b) x'y'z + x'yz + xy'z + xyz

   (c) x'y'z' + x'y'z + x'yz + xy'z' + xy'z + xyz

4. Minimize each of the following boolean expressions using a K-map (some may have more than one correct solution):

   (a) x'y'z + x'yz + x'yz' + xyz'

   (b) x'y'z + x'yz + xy'z + xyz

   (c) x'y'z' + x'y'z + x'yz + xy'z' + xy'z + xyz

   (d) x'y'z' + x'y'z + x'yz + xyz + xyz'

   (e) w'x'y'z' + w'x'y'z + w'xy'z' + w'xy'z + wxy'z' + wxy'z

   (f) w'x'yz + w'xy'z + w'xyz + w'xyz' + wxy'z + wxyz + wx'y'z + wx'yz

   (g) w'xy'z + w'xyz + w'x'y'z + w'x'yz + wxy'z + wxyz + wx'y'z + wx'yz

5. Minimize each of the following boolean functions, in which question marks represent don't cares in the result. In each case show the result as a minimal sum of products expression.

(a)

x	y	z	f(x,y,z)
0	0	0	1
0	0	1	?
0	1	0	0
0	1	1	1
1	0	0	?
1	0	1	1
1	1	0	0
1	1	1	0

(b)

x	y	z	f(x,y,z)
0	0	0	0
0	0	1	?
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	?
1	1	0	1
1	1	1	1

(c)

w	x	y	z	f(w,x,y,z)
0	0	0	0	0
0	0	0	1	?
0	0	1	0	?
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	?
1	0	1	0	0
1	0	1	1	1
1	1	0	0	?
1	1	0	1	1
1	1	1	0	0
1	1	1	1	?

6.2 Basic Logic Gates

The basic logic gates described in this section will be considered the fundamental, or *atomic* components from which all other digital components are created. Subsequent sections describe how these logic gates can be used to build increasingly sophisticated digital components.

6.2.1 AND Gates

The AND gate, shown in Fig 6.13, performs the logical AND operation, as described in chapter 3. Each input to the AND gate may have the value 0 or 1, and never anything else; think of an input (or output) line as a single bit of information. The output of the AND gate will be 1 only if both inputs are 1.

AND gates may have more than two inputs; the AND gate in Fig 6.14 has three inputs. In this case the output is 1 only if all three inputs are 1. In general, an AND gate may have any number of inputs (though we will try not

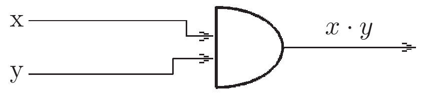


Figure 6.13: A simple AND gate with two inputs, x and y

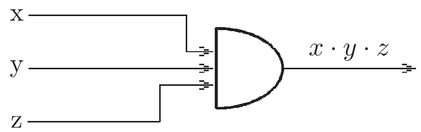


Figure 6.14: A simple AND gate with three inputs, x, y, and z

to use more than five inputs, to avoid cluttering our circuit diagrams). There is no problem with several inputs because the AND operation is associative: $(x \cdot y) \cdot z = x \cdot (y \cdot z)$

Thus the parentheses are not needed, and an AND gate may have any number of inputs.

6.2.2 OR Gates

The OR gate, shown in Fig 6.15, performs the logical OR operation, as described in chapter 3. The output of an OR gate is 0 only if all inputs are 0. In all other respects OR gates are exactly like AND gates. An OR gate with three inputs is shown in Fig 6.15.

When drawing circuit diagrams be sure that your AND gates have a straight base, and that your OR gates have a curved base. Also, your AND gates should have a rounded tip, and your OR gates should have a sharp tip.

6.2.3 Inverters

An inverter performs the NOT operation. Thus, an inverter has one input, and the output is the logical complement of the input:

- if the input is a 0, the output is a 1
- if the input is a 1, the output is a 0

An inverter is shown in Fig 6.16. Since the NOT operation is a unary operation (i.e. it has only one operand), the inverter must have just one input. Note that

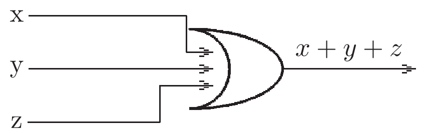


Figure 6.15: A simple OR gate with three inputs, x, y, and z

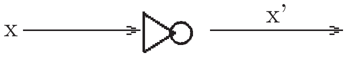


Figure 6.16: An Inverter, with input x

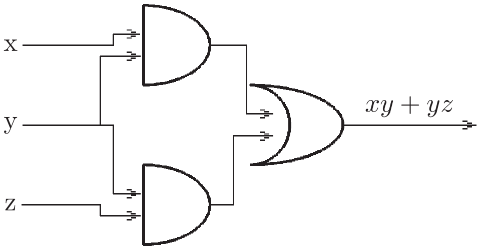


Figure 6.17: An implemenation of the boolean function $xy+yz$ using logic gates

the inverter symbol consists of a small triangle with a small circle, or 'bubble' on the tip. As we will see below, the bubble will usually signify a NOT operation.

6.2.4 Composition of Logic Gates

The logic gates described above can be *composed* (as with functional composition) to obtain logic circuits with output corresponding to a boolean function. To do this we simply connect the output of one gate to an input of another gate. As an example Fig 6.17 shows the diagram which implements the boolean function xy + yz.

Fig 6.18 shows the implementation of a slightly more complex function:
$(x + y) \cdot (y \oplus z) \cdot z'$
This implementation makes use of an *Exclusive OR* gate which is similar to an OR gate, but has a double arc at the base.

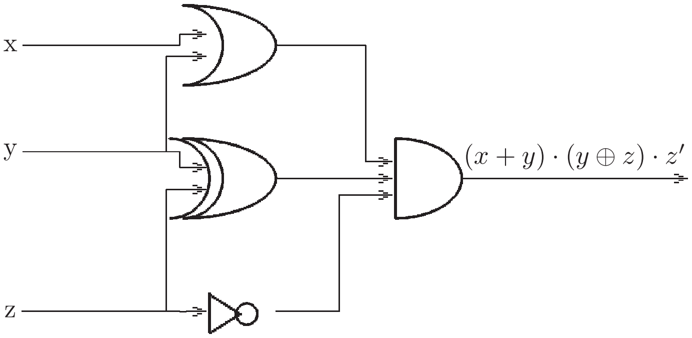


Figure 6.18: An implemenation of the boolean function $(x+y)\cdot(y\oplus z)\cdot z'$ using logic gates

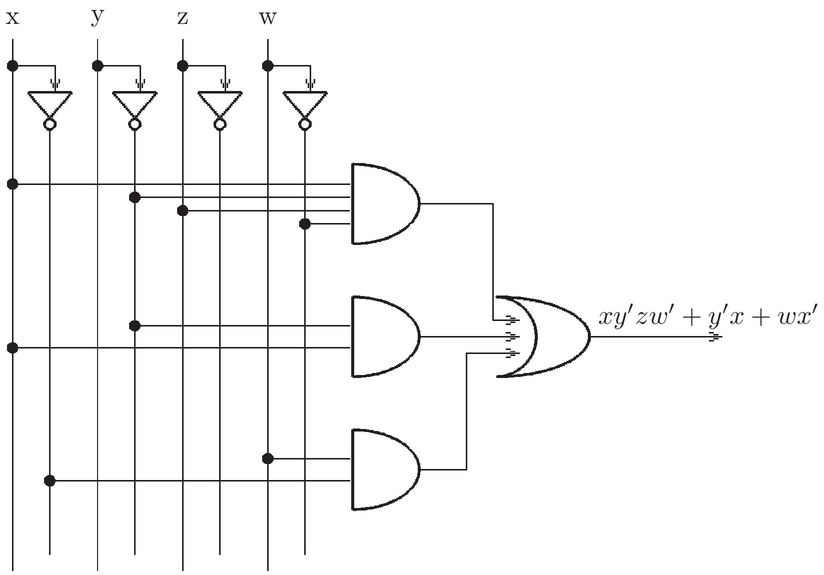


Figure 6.19: A sum of products logic diagram for the expression $xy'zw' + y'x + wx'$

6.2.5 Sum of Products Logic Diagrams

As our logic diagrams become larger, they will tend to become more difficult to read; they will be essentially a tangled morass of wires and gates. To alleviate this problem, we recommend using a more methodical approach. If the boolean expression we wish to implement can be expressed as a (not necessarily canonical) sum of products, we suggest organizing the diagram as shown in Fig 6.19.

In this diagram we show the inputs at the top, and we include an inverter for each input, though the inverter for the variable z is not needed in this example. Each term in the expression then corresponds to the inputs to an AND gate, and the outputs of the AND gates form the input to an OR gate. With this kind of diagram, large boolean expressions can be implemented with a clean, readable diagram.

6.2.6 Wires and Buses

Connecting wires

When drawing logic diagrams it is often inevitable that wires will cross over other wires. We need to be able to distinguish *connected* wires from crossed wires which are *not connected*. To do this we will mark the crosspoint with a dark dot to indicate that the wires are actually connected, as shown in Fig 6.19.

However, when connecting wires, we must be careful to avoid *contradictions*.

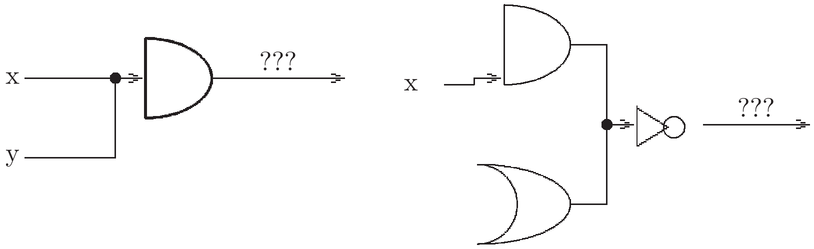


Figure 6.20: Contradictions: incorrect connections of wires

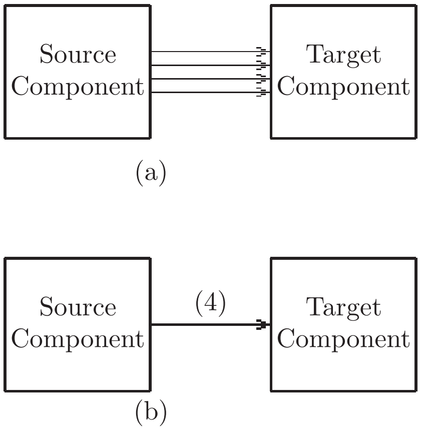


Figure 6.21: Two components connected by four wires, (a) The wires are shown explicitly, and (b) The wires are shown as a bus of width 4

This occurs when a variable, or gate output, is connected to another variable, or gate output, as shown in Fig 6.20.

Buses

In digital logic diagrams we will often need to send several bits from one component to another, all at the same time. We will need several parallel wires to do this; this is called a *bus*. The *width* of a bus is the number of wires which make up the bus. Fig 6.21 shows two components, a source and a target connected by 4 wires. For convenience we can use the diagram in part (b) instead of the diagram in part (a). Note that the width of the bus is always shown in parentheses.

Splitting and joining of buses

The wires of a bus can be split two or more separate buses, as shown in Fig 6.22. The wires of two or more buses can be joined into single bus, as shown in Fig 6.23.

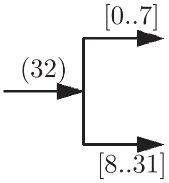


Figure 6.22: A 32-bit bus is split into an 8-bit bus and a 24-bit bus

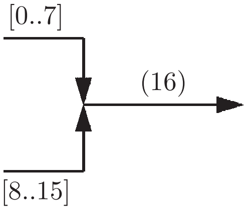


Figure 6.23: Two 8-bit buses are joined to form a 16-bit bus

We should note *how* the buses are split or joined. We note which are the low-order bits and which are the high-order bits by specifying bit locations in square brackets. This is done in both Fig 6.22 and Fig 6.23.

6.2.7 Exercises

1. Show a logic diagram corresponding to each of the following boolean expressions (do not attempt to minimize the expressions):

   (a) xy + z'

   (b) (x+y'+z)(x'+y)z

   (c) xy ⊕ y'z'

   (d) x(x+yz)

2. Show a logic diagram using only AND gates, OR gates, and Inverters, to implement the Exclusive OR operation.
   Hint: Build a truth table first.

3. (a) Assume you have no OR gates. Show how the function x+y can be implemented using only AND gates and Inverters.

   (b) Assume you have no AND gates. Show how the function xy can be implemented using only OR gates and Inverters.

   Hint: Use the DeMorgan identities from Fig 6.3

4. Show a sum of products logic diagram for each of the boolean expressions shown below (do not attempt to simplify the expressions):

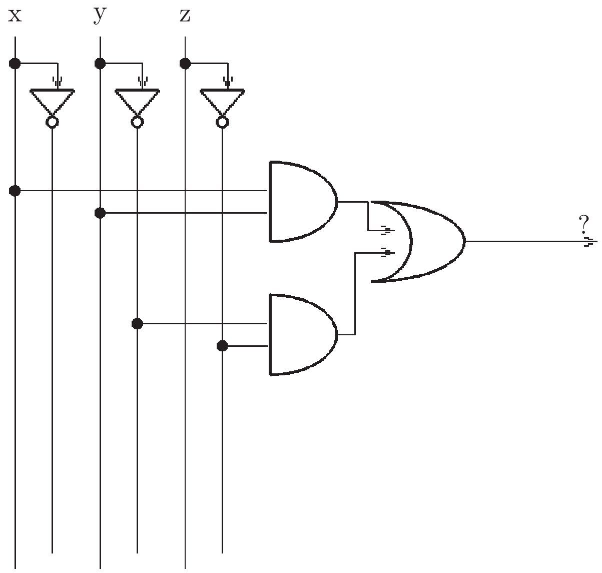
   (a) xy + y'z

(b) xy'z + xy + x'yz'

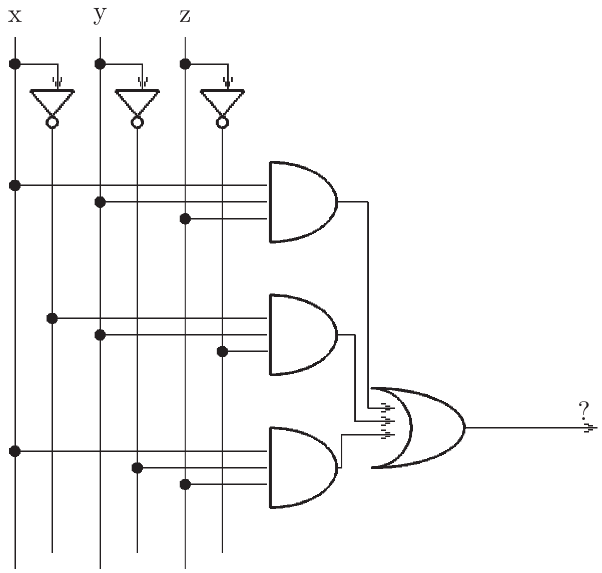
(c) w'x'y'z' + wx + yz

5. Show a boolean expression corresponding to each of the following logic diagrams:

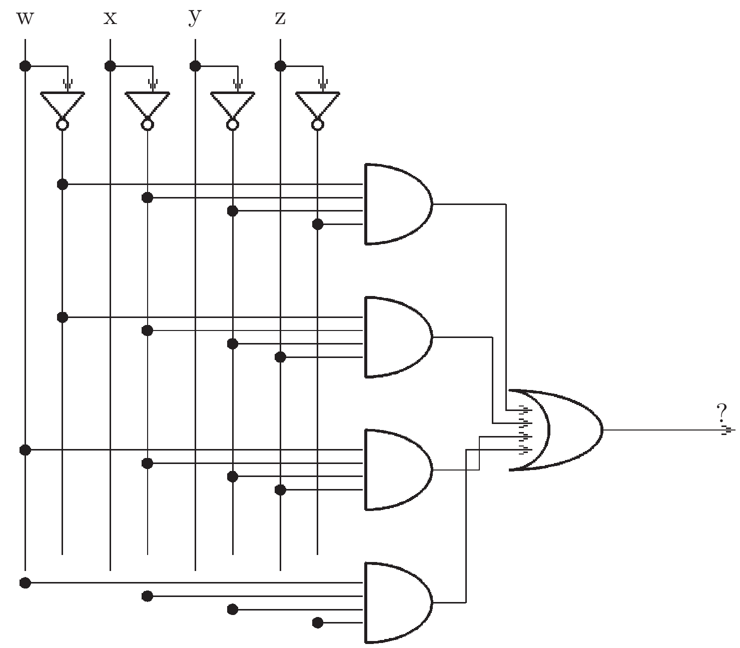
(a)



(b)



(c)



6. Show a Karnaugh map for each of the logic diagrams in the previous exercise, and show a simplified expression, if possible.

6.3 Combinational Logic Circuits and Components

We have seen how the fundamental digital building blocks, logic gates, can be used to implement any boolean function. We now explore the construction of more complex tools from these building blocks. These tools are called digital *components* and will be used to construct the CPU of a computer.

Each component which we will build will consist of:

- A name for the component
- A definition of its inputs
- A definition of its outputs

Once we understand how the component is constructed, and how its outputs are determined from its inputs, we can then represent the component as a plain box, showing simply its name, inputs, and outputs. In this way we can use it in a logic diagram without cluttering the diagram with the inner workings of our component. This process, known as abstraction, is also used in software development. A clearly defined and appropriate interface replaces the need to expose the inner workings of a component.

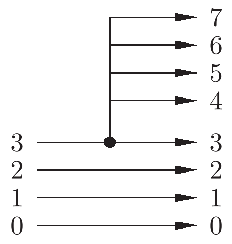


Figure 6.24: A sign-extend component for which the input is a 4-bit bus, and the output is an 8-bit bus, preserving the sign of the number

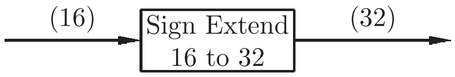


Figure 6.25: Block diagram of a 16-bit to 32-bit sign extend component

6.3.1 Sign Extend

Perhaps the simplest component that we may wish to use is called *sign extend*. It simply propagates the sign bit on a bus to yield a bus with more bits, preserving the sign of the number. For example, if we have a 4-bit bus as input to a sign extend which puts out an 8-bit bus, then the following example shows that the sign is preserved, whether it be positive, negative, or zero:

Number	Input bus	Output bus
6	0110	0000 0110
-6	1010	1111 1010

This can be accomplished easily by connecting the high order bit of the input bus to all the extended bits in the output bus, as shown in Fig 6.24.

A block diagram for a sign extend component is shown in Fig 6.25.

6.3.2 Decoders

A component which selects one of several output lines, based on the binary value of its input, is called a *decoder*. A decoder with n input lines will have 2^n output lines. For example, a decoder with 3 inputs will have 8 outputs; this would be known as a 3x8 decoder.

If the 8 outputs of a 3x8 decoder are labeled D_0 through D_7, then an input value of i will result in a value of 1 on output D_i and a value of 0 on all other outputs. For example, if the inputs to a 3x8 decoder are $101_2 = 5$, then the outputs will be 00100000 in which D_5 is 1 (here we show the low order bit, D_0 at the right).

Fig 6.26 shows the 3x8 decoder function as a truth table. The inputs are labeled I, and the outputs are labeled D. Fig 6.27 shows the logic diagram which implements that function. Think of the inputs to a decoder as *selectors*, because they act to select one output line.

I_2	I_1	I_0	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

Figure 6.26: Truth table defining a 3x8 decoder

Now that we understand how to build a decoder of any size, we can present a more *abstract* view of a decoder, showing only the name, inputs, and outputs. This is called a *block diagram* of a decoder and is shown in Fig 6.28, which shows a 3x8 decoder. For simplicity we show the three input lines as a bus of width 3, and the eight output lines as a bus of width 8.

How can a decoder perform a useful task? Consider a traffic signal with four lights: red, yellow, green, and left-turn arrow. Assume we have a mod-4 counter, i.e. a counter which repeatedly puts out the values 00, 01, 10, 11, 00, 01, 10, 11, ... on two output lines (mod-4 counters will be covered in the section on sequential circuits, below). We can then use a 2x4 decoder to send a 1 or 0 signal to each light, so that only one light is on at any one time. The logic diagram is shown in Fig 6.29.

6.3.3 Encoders

We now turn our attention to a component which performs the inverse function of the decoder; an *encoder* is a device with 2^n inputs and n outputs. Normally only one of the inputs will have a value of 1, and the others will all have a value of 0. If the i^{th} input is a 1, then the binary value of i will be on the output lines. For example, for an 8x3 encoder, if the inputs are 00010000 (i.e. I_4 is 1), then the output will be $100_2 = 4$. Fig 6.30 shows a truth table for an 8x3 encoder. encoder.

How can we build a 4x2 encoder using our basic logic gates? We take the first four rows of Fig 6.30, and form a K-map for each of the two outputs in which unexpected input combinations are shown as don't cares, as shown in Figures 6.31 and 6.32. In those figures, the inputs which we had previously been calling w,x,y,z are now I_3, I_2, I_1, I_0, respectively. The resulting minimized expressions are

$E_1 = I_3 + I_2$

$E_0 = I_3 + I_1$

Note that the input I_0 is not used! Fig 6.33 shows the logic diagram for a 4x2 encoder.

We could also have 2x1, 8x3, 16x4 encoders, etc. As an example of an appli-

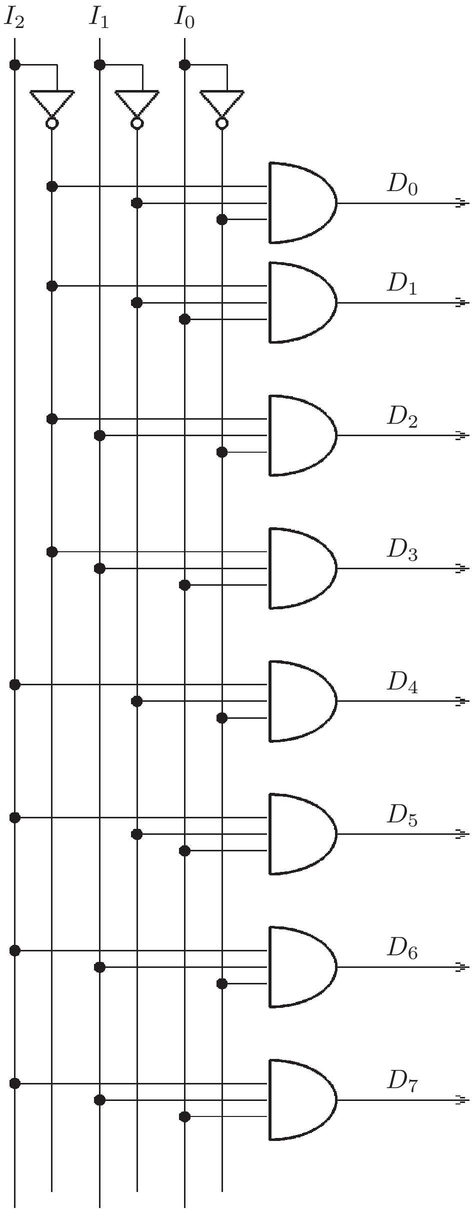


Figure 6.27: Logic diagram for a 3x8 decoder

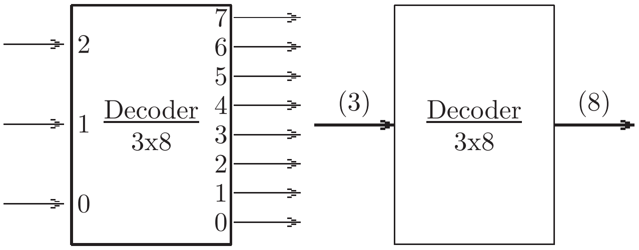


Figure 6.28: Block diagrams of a 3x8 decoder; left diagram shows the inputs and outputs as separate lines; right diagram shows the inputs and outpus as busses.

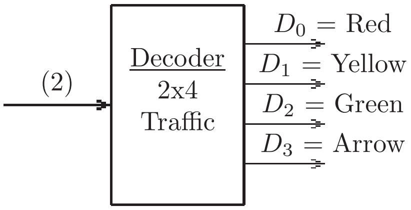
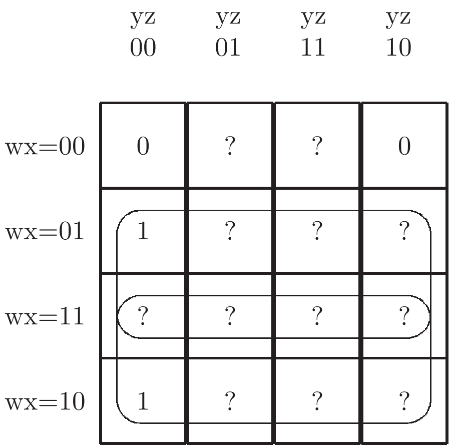


Figure 6.29: Traffic signal control using a 2x4 decoder. The inputs come from a mod-4 counter, the outputs go to the four lights on a traffic signal - Red, Yellow, Green, Left-turn arrow

I_7	I_6	I_5	I_4	I_3	I_2	I_1	I_0	E_2	E_1	E_0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

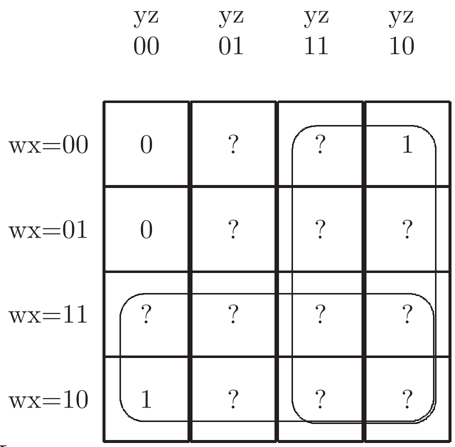
Figure 6.30: Truth table defining an 8x3 encoder; the inputs are labeled I, and the outputs are labeled E



	yz 00	yz 01	yz 11	yz 10
wx=00	0	?	?	0
wx=01	1	?	?	?
wx=11	?	?	?	?
wx=10	1	?	?	?

$E_1 = w + x = I_3 + I_2$

Figure 6.31: A K-map for the high order output bit, E_1 of a 4x2 encoder. Unexpected input combinations are shown as don't cares (question marks). The inputs wxyz = $I_3 I_2 I_1 I_0$.



	yz 00	yz 01	yz 11	yz 10
wx=00	0	?	?	1
wx=01	0	?	?	?
wx=11	?	?	?	?
wx=10	1	?	?	?

$E_0 = w + y = I_3 + I_1$

Figure 6.32: A K-map for the low order output bit, E_0 of a 4x2 encoder. Unexpected input combinations are shown as don't cares (question marks). The inputs wxyz = $I_3 I_2 I_1 I_0$.

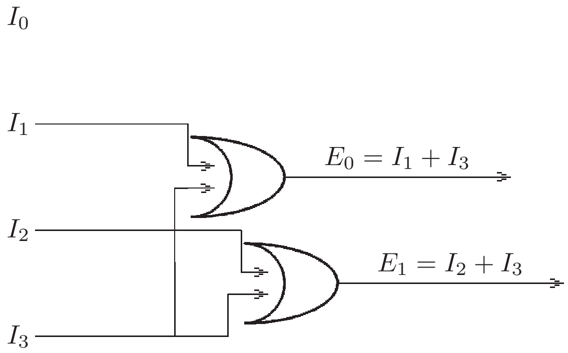


Figure 6.33: A logic diagram implementing a 4x2 encoder

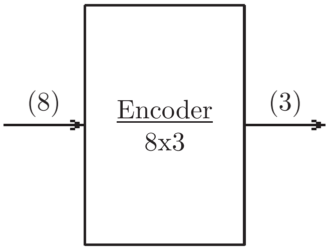


Figure 6.34: Block diagram of an 8x3 encoder

cation which could use an encoder, consider a building with a motion detector in each room. We wish to put out an alert signal if motion is detected in any room. The alert signal should indicate the room number in which motion is detected.[5] If the building has 32 rooms, we could use a 32x5 encoder, with an input coming from a motion detector in each room (0=no motion detected, 1=motion detected). Our output would then be the room number (in binary) of the room in which motion is detected.

A block diagram for an 8x3 encoder is shown in Fig 6.34.

6.3.4 Multiplexers

There are many applications in which we wish to select one of several input lines (or buses), and pass it on to the output. This kind of selector is generally called a *multiplexer*.[6] Thus in addition to the data inputs, the multiplexer will require *control inputs* which determine which of the input lines are put on the output lines. For example, a multiplexer with 8 data inputs will have one output line and 3 control lines to select one of the 8 data inputs. This would be called an 8x1 multiplexer. In an 8x1 multiplexer, if the three control bits are 101 (i.e. 5), then the value of the fifth input data line is copied to the output data line.

[5] We assume that motion may be detected in no more than one room at a time.

[6] Also spelled multiplexor

I_7	I_6	I_5	I_4	I_3	I_2	I_1	I_0	S_2	S_1	S_0	M
								0	0	0	I_0
								0	0	1	I_1
								0	1	0	I_2
								0	1	1	I_3
								1	0	0	I_4
								1	0	1	I_5
								1	1	0	I_6
								1	1	1	I_7

Figure 6.35: Truth table defining an 8x32 multiplexer. Each of the eight data inputs, I, is a 32-bit bus, and the output,M, is a 32-bit bus.

S	I_1	I_0	M
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Figure 6.36: Truth table defining a 2x1 multiplexer.

As with other components, the inputs and outpus may be buses. A multiplexer with 16 data inputs, each of which is a 32-bit bus, will have one output bus (also 32 bits) and 4 control bits to select one of the 16 data input busses. This would be called a 16x32 multiplexer. A truth table for an 8x32 multiplexer is shown in Fig 6.35.

In general, a multiplexer[7] with k control bits, and n-bit data buses would be called a 2^k x n multiplexer.[8]

How can we design a simple multiplexer, using basic logic gates? To build a 2x1 multiplexer, working from the truth table in Fig 6.36 we form the K-map shown in Fig 6.37.

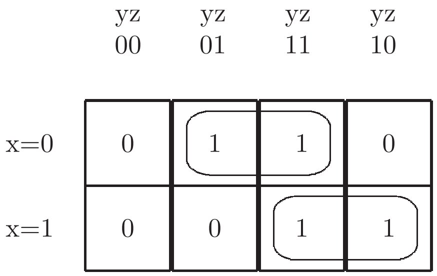
This will then give us the sum of products expression:
$M = S'I_0 + SI_1$
The logic diagram is shown in Fig 6.38

Block diagrams of an 8x1 multiplexer and a 4x16 multiplexer are shown in Fig 6.39.

As an example of an application which could use a multiplexer, consider a

[7] Terminology for multiplexers in the literature is not consistent. What we call an 8x4 multiplexer, others might call a quad 8-input multiplexer.

[8] For those who understand logarithms, an mxn multipexer would have $log_2(m)$ control lines.



	yz 00	yz 01	yz 11	yz 10
x=0	0	1	1	0
x=1	0	0	1	1

Figure 6.37: K-map used to build a 2x1 multiplexer, derived from the truth table in Fig 6.36

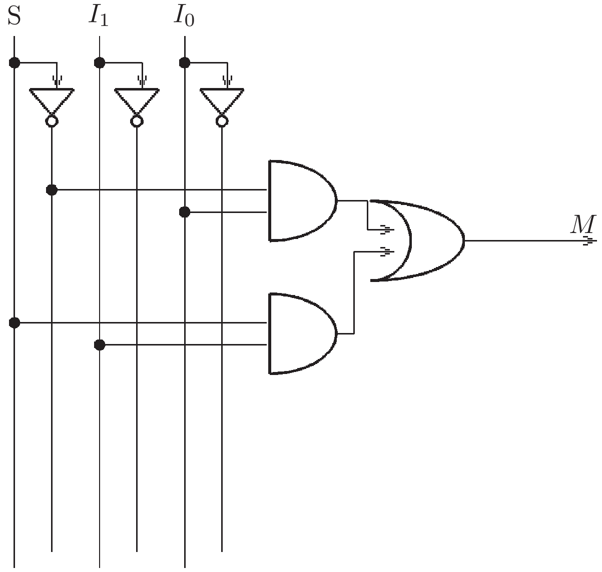


Figure 6.38: Logic diagram for a 2x1 multiplexer

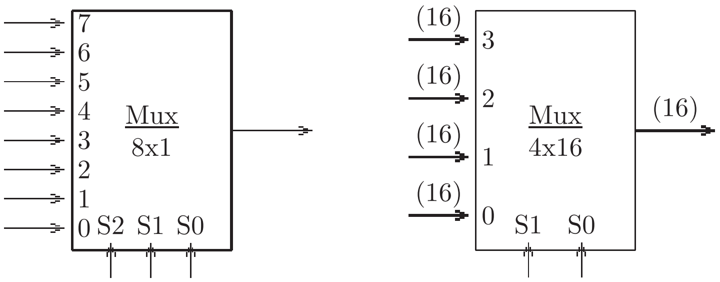


Figure 6.39: Block diagrams of an 8x1 multiplexer (3 control inputs), left, and a 4x16 multiplexer (2 control inputs), right

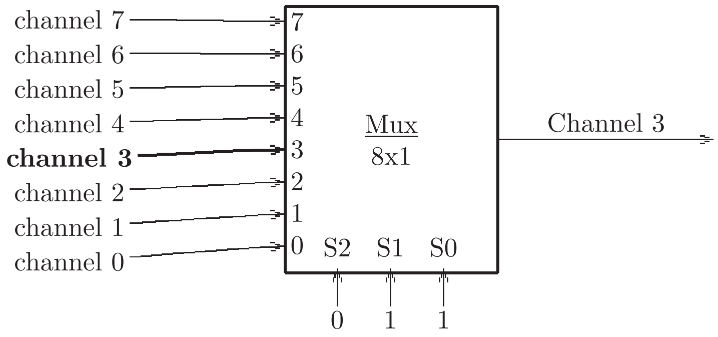


Figure 6.40: Application of a multiplexer: A digital radio or TV channel selector. Channel 3 is selected by the user.

radio or TV which needs to select one of several digital channels to be put out to the user. Each data input to the multiplexer would be the digital signal for a particular channel, and the control inputs would be used to select one of the channels, which is than sent to the output data line. A possible diagram is shown in Fig 6.40, in which the user has selected channel 3.

6.3.5 Binary Adders

Thus far we have basic logic gates and digital components that can be constructed using those gates. However, our goal is to design the hardware that can implement the MIPS instruction set. Thus, we will need to be able to perform *arithmetic* operations, such as add, subtract, multiply, divide, in addition to the logical operations. In this section we discuss how an adder can be built using basic logic gates. The other arithmetic operations can be derived from an adder: if we have a *negate* operation, we can implement subtraction by adding a negated operand.
$a - b = a + (-b)$
Multiplication and subtraction can then be implemented as repeated addition and subtraction.[9]

When adding binary values, we have the possiblity of a carry bit (1) when the sum in any column exceeds 1, as shown in Fig 6.41 where we add 11 + 14. Thus we will need to allow for the fact that the hardware needed to process any one column in that addition will need three inputs: the x value, the y value, and the carry from the previous column.

To build an adder, we suggest a two-step process:
1. Design a *Half Adder* which will take two bits as input and produce as output a sum bit and a carry bit.
2. Design a *Full Adder* which will take three bits as input and produce as output

[9] As we saw in chapter 3, multiplication can be implemented with a repeated shift-and-add; division and remainder can be implemented with a repeated shift-and-subtract.

carries	1	1	1		
x = 11	0	1	0	1	1
y = 14	0	1	1	1	0
sum = 25	1	1	0	0	1

Figure 6.41: Addition of binary numbers: 11 + 14 = 25

x	y	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Figure 6.42: Truth table defining the sum and carry outputs of a half adder

a sum bit and a carry bit. The full adder can be implemented using two half adders.

To design the half addder we refer to the truth table in Fig 6.42 which shows two outputs, a sum (S) and a carry (C). From the truth table we obtain boolean expressions for the outputs:
$S = x'y + xy' = x \oplus y$
$C = xy$

Using these expressions for the output we can build the logic diagram for a half adder; it is shown in Fig 6.43, and a block diagram for a half adder is shown in Fig 6.44

We now turn our attention to the full adder; it will have three inputs: x, y, and the carry from the previous stage. It will have two outputs: sum and carry to the next stage. The truth table for a full adder is shown in Fig 6.42 which shows the two outputs, a sum (S) and a carry (C). In this figure we distinguish between the two carries. c_{in}, or carry-in, is the carry from the previous column. C_{out}, or carry-out, is the carry out to the next coliumn. In general the carry-out from column i is the carry-in to column i+1, working from right to left.

At this point we could find minimal boolean expressions for the two outpus and construct the logic diagram. But there is an easier way: we propose using

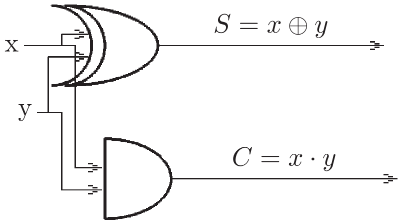


Figure 6.43: A logic diagram implementing a half adder. S is the sum, and C is the carry.

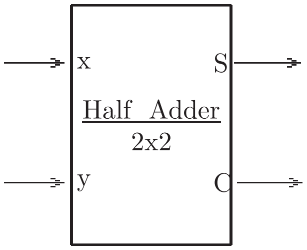


Figure 6.44: Block diagram for a Half Adder. S is the one-bit sum, x+y, and C is the carry.

x	y	c_{in}	S	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Figure 6.45: Truth table defining the sum and carry outputs of a full adder

x	y	c_{in}	S	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Figure 6.46: Truth table defining the sum and carry outputs of a full adder

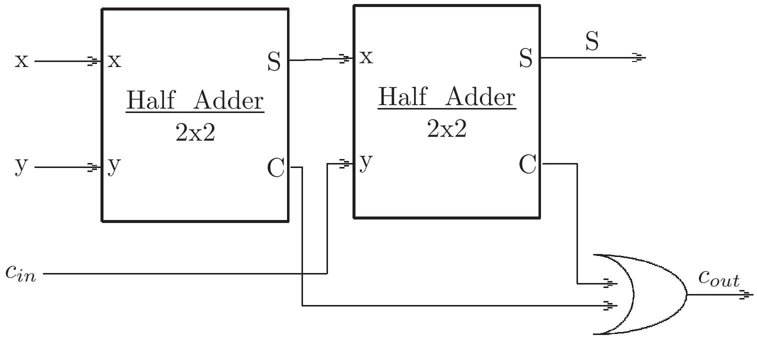


Figure 6.47: Logic diagram for a full adder, using two half adders

two half adders to construct the full adder.

The first half adder simply adds x and y. The S output of the first half adder forms the x input to the second half adder. The second input to the second half adder is c_{in}. The S output of the second half adder is the S output of the full adder. The two C outputs of the half adders are input to an OR gate, which produces the full adder's C output. This is all shown with block diagrams in Fig 6.47.

We can now show a block diagram for our full adder in Fig 6.48. It is important to point out that to add two numbers, we would need several full adders, one for each bit position. Thus to add two 32-bit values we would need 32 full adders, in which the carry-out of each stage is the carry-in to the next stage. For the very first stage, the carry-in is always 0. For the very last stage the carry-out can be used to detect overflow for unsigned addition.[10]

In the chapters which follow we will need to use special-purpose adders, apart from the ALU described in this chapter. We define such an adder at this point.

[10] For twos complement addition, overflow can be detected when the carry-in to the last stage is different from the carry-out of the last stage.

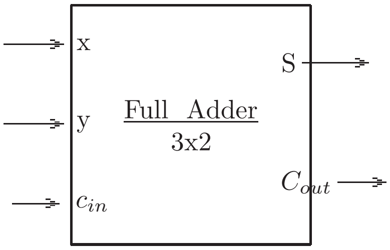


Figure 6.48: Block diagram for a Full Adder. S is the one-bit sum, $x + y + c_{in}$, and C_{out} is the carry-out.

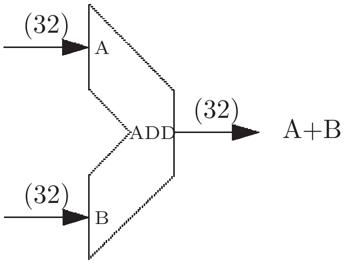


Figure 6.49: Block diagram of a 32-bit adder

It will have two input busses (the values being added) and one output bus (the sum). The adder will consist of n Full Adders, each of which is as shown in Fig 6.48, where n is the size of the busses. For example, an adder which adds two 16-bit numbers, producing a 16-bit result would be called a *16-bt adder*. A block diagram of a 32-bit adder is shown in Fig 6.49.

In this adder the S output of the i^{th} full adder is the i^{th} bit of the adder's output bus. The C_{out} output of the i^{th} full adder is the c_{in} to the $i + 1^{st}$ full adder. The carry out from the high order bit will not be used here.

6.3.6 Exercises

1. In each case show the 8-bit output of a 3x8 decoder if the inputs are as shown below (the low order bit is at the right end):

   (a) 000

   (b) 011

   (c) 110

2. In each case show the 3-bit output of an 8x3 encoder for the inputs given below

(a) 00000010

(b) 00100000

(c) 10000000

3. (a) Show the logic diagram for an 8x3 encoder using only AND gates, OR gates, and inverters.
   Hint: You will not be able to use kMaps because there are too many inputs. Instead write a boolean expression for each of the three outputs.

   (b) What would be the output of your 8x3 encoder if the input were 00101000 ? (This is not a valid input, but your encoder would still produce an 3-bit output)

4. Show a block diagram for each of the following:

   (a) A 4x1 multiplexer

   (b) A 4x4 multiplexer

   (c) An 8x32 multiplexer

5. Show a logic diagram for a 4x1 multiplexer.

6. What is the output of a 4x8 multiplexer if the inputs are $I_3 = 01010000$, $I_2 = 01011110$, $I_1 = 00101101$, $I_0 = 01010000$, and the control input is 10?

7. In each case show the output of a full adder if the input is:

   (a) 011

   (b) 101

   (c) 110

8. How many full adders are needed to implement the MIPS `add` instruction?

9. Construct a full adder using only basic logic gates (i.e. do not use a half adder). Try to minimize your design.

   (a) Show a Kmap for each of the two outputs. Work from the truth table in Fig 6.46.

   (b) Draw the logic diagram, or two separate logic diagrams for the two outputs. The Kmaps give you a minimal sum-of-products for the function. Perhaps using XOR gates provides a cheaper solution.

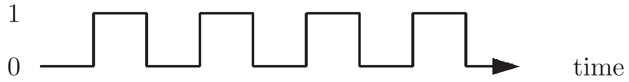


Figure 6.50: Clock signal

6.4 Sequential Circuits

Up to this point we have been working entirely with combinational circuits. Combinational circuits:

- Have no memory, or storage, capability
- Outputs change as soon as an input changes
- Cannot be used to implement CPU registers.

With *sequential circuits* components can maintain *state*, thus storing some representation of the inputs that it has received over some period of time. Generally, the components in a sequential circuit need to be synchronized in such a way that they all update their states at the same time. This is done with a *clock* signal. A clock signal is a 1-bit signal that varies periodically, and consistently, between 0 and 1. A diagram of a clock signal is shown in Fig 6.50 in which the vertical axis shows the value of the clock signal (0 or 1), and the horizontal axis is time.

The *period* of a clock signal is the time it takes to undergo a complete transition from 0 to 1 and back to 0. This is called a *cycle*. The *frequency* of a clock signal is the number of complete periods in a unit of time. Frequency is usually expressed with the unit *cycles per second*, or Hertz.[11] Period and frequency are multiplicative inverses of each other:
frequency = 1 / period
period = 1 / frequency

If the clock signal for a particular CPU has a period of 1 *nanosecond* $(1ns) = 10^{-9}$ sec, then its frequency would be $1/10^{-9} sec = 10^9$ *cycles per second* $= 10^9 Hertz = 1\ GigaHertz\ (1GHz)$

In this section we introduce a component known as a *flip-flop*, which is capable of storing 1 bit of information, for use at a later time, even if an input changes. Thus we will find that flip-flops will be useful in implementing CPU registers (each bit of the register requires one flip-flop). As with other components, flip-flops will be constructed from basic logic gates.

[11] Named after the German Physicist Heinrich Hertz, who proved the existence of electromagnetic waves (light) in the late 19th century

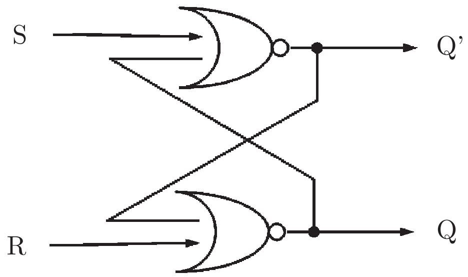


Figure 6.51: SR Latch: Does not change state when S=R=0.

6.4.1 SR Flip-Flops

Our first example of a flip-flop is known as an SR flip-flop, because the two inputs are Set and Reset. To begin we construct a one-bit storage device known as an *SR Latch*. Though it has some undesirable characteristics, it will lead us to the design of an SR flip-flop. A diagram of an SR latch is shown in Fig 6.51.

This is our first example of a combinational circuit with *feedback*. The output of the top NOR gate is used as the first input to the bottom NOR gate, and the output of the bottom NOR gate is used as the second input to the top NOR gate. This will require some careful thinking to analyze.

With a NOR gate, if one of the inputs is a 1, we know the output must be a 0, regardless of the value of the other input. Algebraically, (1+x)' = 0. This leads to the table shown below:

	S	R	Q'	Q	State
1	0	0	?	?	Unchanged
2	0	1	1	0	Reset (0)
3	1	0	0	1	Set (1)
4	1	1	0	0	Unknown

The Q output of the SR latch represents the state of the latch, and Q' represents the complement of the state. We explain rows 2, 3, 4, and 1 in this table:

- Row 2: State = Reset(0), S=0, R=1. Because R is 1, the the output of the bottom NOR gate must be (1+x)' = 0, thus Q is 0. The bottom input to the top NOR gate is 0, thus the output of the top NOR gate is (0+0)' = 1, and thus Q' is 1.

- Row 3: State = Set(1), S=1, R=0. Because S is 1, the output of the top NOR gate must be (1+x)' = 0, thus Q' is 0. The top input to the bottom NOR gate is 0, thus the output of the bottom NOR gate is (0+0)' = 1, and thus Q is 1.

- Row 4: State = Unknown, S=1, R=1. Because S and R are both 1, the outputs of both NOR gates must be 0. Thus, Q=0 and Q'=0, which is a

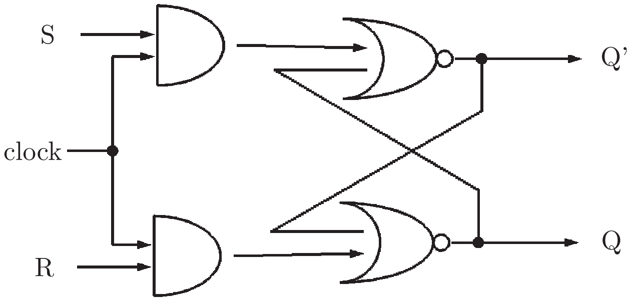


Figure 6.52: SR Flip-flop: Can change state only when clock=1

contradiction because Q' is supposed to be the complement of Q.

- Row 1: State = Unchanged, S=0, R=0. Here the current state of the flip-flop depends on its previous state. If the previous state had been Set (Q=1,Q'=0), then the current state would still be Set (Q=1, Q'=0). If the previous state had been Reset (Q=0,Q'=1), then the current state would still be Reset (Q=0, Q'=1).

To make effective use of the SR latch, the user must be careful not to set both inputs at 1.

We will extend the design of the SR latch to arrive at the design of a clocked SR flip-flop. This is done by adding two AND gates and a clock input, as shown in Fig 6.52.

The clock input insures that the flip-flop can change state only when the clock signal is 1. When the clock signal is 0, both inputs to the NOR gates are 0, and as we showed above the latch maintains its current state. This will make the SR flip-flop useful in digital circuits in which all components need to be synchronized (i.e. change state at the same time).

6.4.2 D Flip-Flops

Above we pointed out that the inputs to the SR flip-flop should not both be 1, because that yields an unknown state. The D flip-flop is similar to the SR flip-flop; however, it ensures that the S and R inputs are complements of each other. The D flip-flop is shown in Fig 6.53. As with the SR flip-flop, the clock input serves to synchronize the change of state with other devices. Aside from the clock input, there is only one input, D, and its complement is formed with an inverter. The D input determines the state of the flip-flop, as shown in the table, below:

	D	Q'	Q	State
1	0	1	0	Reset (0)
2	1	0	1	Set (1)

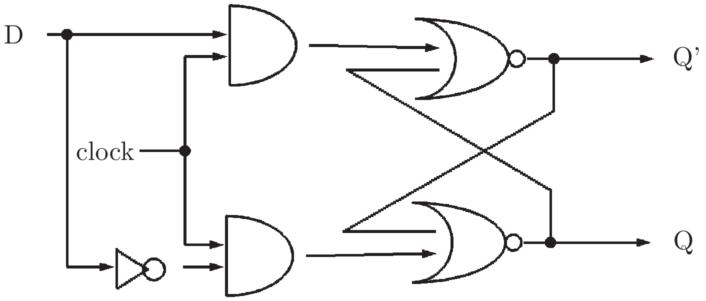


Figure 6.53: D Flip-flop: D input determines the state

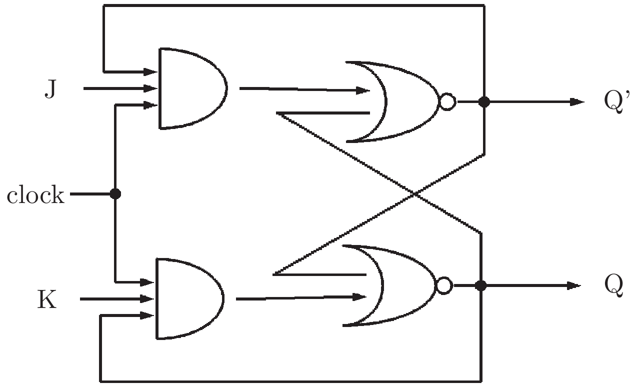


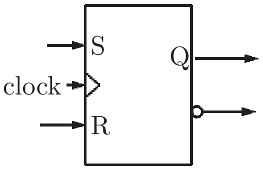
Figure 6.54: JK Flip-flop: Can be complemented

6.4.3 JK Flip-Flops

JK flip-flops are similar to SR flip-flops, with the addition of feedback from the NOR gates to the AND gates, as shown in Fig 6.54. Because of the feedback, the behavior of this flip-flop will clearly depend on the current state. In the table below, Q_n represents the current value of the Q output (which represents the state of the flip-flop), and Q_{n+1} represents the state of the flip-flop when the clock signal returns to 1, i.e. when the state is permitted to change.

The behavior of a JK flip-flop is analyzed using the table below:

	J	K	Qn	Q_{n+1}
1	0	0	0	0
2	0	0	1	1
3	0	1	0	0
4	0	1	1	0
5	1	0	0	1
6	1	0	1	1
7	1	1	0	1
8	1	1	1	0



S	R	State (Q)
0	0	No change
0	1	Reset (0)
1	0	Set (1)
1	1	Undefined

Figure 6.55: SR Flip-flop: Block diagram and function table

Referring to Fig 6.54, when the output of both AND gates are 0, the inputs to both NOR gates are 0, and the state of the flip-flop is unchanged. This is the case in rows 1, 2, 3, and 6 of the table. In row 4 the output of the top AND gate is 0, and the output of the bottom AND gate is 1; thus we essentially have inputs of S=0 and R=1 to the embedded latch (the two NOR gates). The state is changed to 0 (Q=0). In rows 5 and 7 the output of the top AND gate is 1, and the output of the bottom AND gate is 0; thus we essentially have inputs of S=1 and R=0 to the embedded latch (the two NOR gates). The state is changed to 1 (Q=1). In row 8 the output of the top AND gate is 0, and the output of the bottom AND gate is 1; thus we essentially have inputs of S=0 and R=1 to the embedded latch (the two NOR gates). The state is changed to 0 (Q=0).

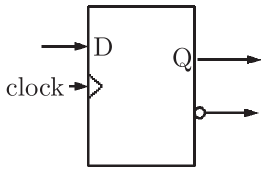
The important feature of JK flip-flops, as compared with the other flip-flops we've seen, is that it has a *toggle* feature: when J and K are both 1, the state of the flip-flop is complemented. This is like a toggle light control which is simply a button that turns the light off if it is on, and on if it is off.

6.4.4 Block Diagrams and Function Tables for Flip-Flops

In this section we introduce block diagrams for SR, D, andJK flip-flops. We also summarize the behavior of these flip-flops with *function tables*.

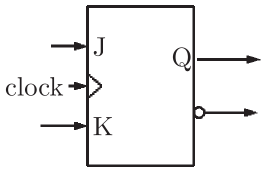
As with combinational circuit components, the block diagrams hide the details of a component, and show the input(s), the output(s), and a name to identify the component. The function tables show the exact input-output relation of the component, i.e. how the input affects the output.

The block diagram and function table of the SR, D, and JK flip-flops are shown in Fig 6.55, Fig 6.56, and Fig 6.57, respectively.



D	State (Q)
0	Reset (0)
1	Set (1)

Figure 6.56: D Flip-flop: Block diagram and function table



J	K	State (Q)
0	0	No change
0	1	Reset (0)
1	0	Set (1)
1	1	Complement

Figure 6.57: JK Flip-flop: Block diagram and function table

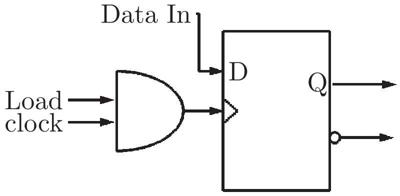


Figure 6.58: A 1-bit register, with a load signal

6.4.5 Registers

We have seen that a flip-flop is merely a 1-bit storage device. Thus, several flip-flops can be used to implement a CPU register. For example, a 32-bit register would consist of 32 flip-flops, one flip-flop for each bit of the register. We choose to use D flip-flops for this purpose. However, to ensure that the register's state changes only at the appropriate times (for example, during a store word instruction) we will use a *load* signal to tell the flip-flops that they should change state according to the D input. To do this all we need is an AND gate, taking as inputs the clock signal and the load signal. The output of the AND gate will then be the clock input to the D flip-flop, as shown in Fig 6.58

We can use four of these devices to build a 4-bit register with a load signal. It is shown in Fig 6.59. In this diagram the signal i_k is the k^{th} bit of the input bus to the register, and the signal reg_k is the k^{th} bit of the register's stored value.

6.4.6 State Machines

Using one or more flip-flops we can build machines that are said to be in a particular *state* at any time. For example, a machine with two flip-flops can have one of four possible states: 00, 01, 10, 11. A machine with n flip-flops can have at most 2^n states. Such machines are called *state machines.*[12]

State machines can be represented with graphs, also known as *state diagrams*. In such a graph, each state is a *node* and is represented by a circle; the name of the state, if it has one, would be inside the circle. At any given time the machine is said to be in one of its states, also known as the *current state*. When an input is provided to the machine, it can change to a different state; this is called a *transition*. A transition is shown in the graph by a directed arrow from one state to another (or to the same) state. The arrow is labeled by

[12] The reader may be familiar with theoretical *finite state machines*, abreviated FSM, or *deterministic finite automata*, abbreviated DFA.

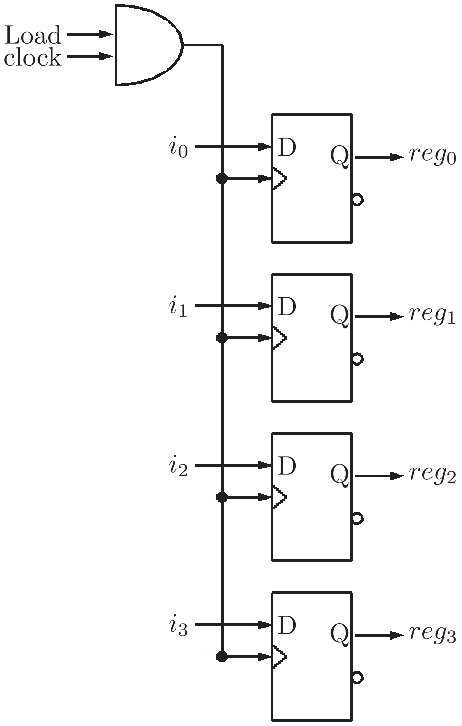


Figure 6.59: A 4-bit register, with a load signal

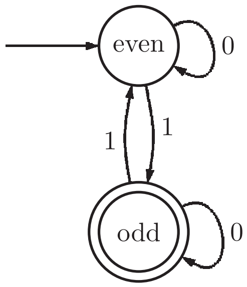


Figure 6.60: An example of a state machine: a parity checker

the input causing the transition to take place (0 or 1).

An example of a state machine is shown in Fig 6.60. The machine is represented as a state graph, or state diagram. We note the following about this state machine:

- This state machine has two states, labeled `even` and `odd`.
- When the machine is in the `even` state and the input is a 1, the machine makes a transition to the `odd` state. When the machine is in the `odd` state and the input is a 1, the machine makes a transition to the `even` state.
- When the machine is in the `even` state, it has seen an even number of 1's. When the machine is in the `odd`. state, it has seen an odd number of 1's. Such a machine is known as a *parity* checker.
- The unlabeled arrow pointing to the `even` state indicates that the `even` state is the *start* state. Every state machine should have exactly one start state. Before reading any input symbols, the machine is in the start state.
- The double circle on the `odd` state indicates that it is an *accepting* state.[13] A state machine may have zero or more accepting states.

State graphs are most useful when designing and analyzing state machines. To construct the state machine using flip-flops, it will be helpful to represent the machine as a *table*. In a state table, the columns are labeled by input symbols, and the rows are labeled by states. A state, `s`, in row `r` and column `c` indicates that if the machine is in state `r` and the input is `c`, then the machine makes a transition to state `s`.

The parity checker of Fig 6.60 is shown in table form, also known as a *transition table*, in Fig 6.61.

[13] If viewed as a terminating state, an accepting state provides the machine with the capability of defining a language as the set of strings which cause the machine to end up in an accepting state after the entire string has been read. The machine of Fig 6.60 will accept any string of 0's and 1's which has odd parity. This kind of machine is known as a *Moore* machine: the output is associated with the state; state machines which produce an explicit output on each transition are known as *Mealy* machines.

States	Inputs	
	0	1
even	even	odd
odd	odd	even

Figure 6.61: Transition table for the parity checker

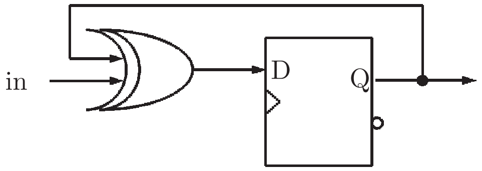


Figure 6.62: Parity checker, using a D flip-flop

We are now ready to build the parity checker logic circuit. We know that we will not need more than one flip-flop because the machine has only two states. Let 0 represent the `even` state, and let 1 represent the `odd` state. We will use a D flip-flop; the Q output of the flip-flop represents the state of the machine. To build the logic circuit we first show a truth table relating the input, the current state of the machine, and the next state of the machine. This will show us what logic gates are needed in a feedback loop from the output of the flip-flop.

i	Q	D = next State
0	0	0
0	1	1
1	0	1
1	1	0

In this truth table `i` represents the input symbol, `Q` represents the current state, and `D` represents the next state. We can write a boolean expression for `D`:

$D = i \oplus Q$

Using what we have developed above, we can now build the logic circuit for our parity checker. It is shown in Fig 6.62.[14]

6.4.7 Exercises

1. Implement the parity checker of Fig 6.60. using a JK flip-flop and no other logic gates.

2. Design a state machine to control a traffic signal. The state machine has outputs which enable the lights on the traffic signal: Red, Yellow, Green, and Left Turn Arrow. The sequence of states for the traffic signal are Green and Left Turn, Green, Yellow, Red. Each time a 'clock' signal is applied, the machine goes to the next state.

[14] We have chosen to use a D flip-flp because it generalizes well to more complex state machines. The parity checker can also be implemented with a JK flip-flop.

3. Design a state machine which will calculate n % 3 (i.e. n mod 3) for any unsigned binary number n provided as the input. Examples:

n	n % 3	n	n % 3
0000 = 0	00 = 0	0110 = 6	00 = 0
0001 = 1	01 = 1	0111 = 7	01 = 1
0010 = 2	10 = 2	1110 = 14	10 = 2
0011 = 3	00 = 0	1000000 = 64	01 = 1
0100 = 4	01 = 1	1010110 = 86	10 = 2
0101 = 5	10 = 2	1111111 = 127	01 = 1

On each clock signal, the machine reads one bit of the number, high order bit first. The flip-flop outputs represent the binary value of `n % 3`.

(a) Draw the state diagram.

(b) Derive the state table from the state diagram.

(c) Make the state assignments for two flip-flops; show the truth table relating the current state, current input, and next state.

(d) Derive the equations for the feedback loops to the flip-flop inputs.

(e) Draw the logic diagram using gates and flip-flops.

Hints:

- This machine will need 4 states; the machine will never return to the start state after reading the first input symbol. The other states represent the three possible results: 0=00, 1=01, and 2=10.
- After reading several bits of a binary number, if the next bit is a 0, the number is doubled.
- After reading several bits of a binary number, if the next bit is a 1, the number is doubled, plus 1.

6.5 An Arithmetic and Logic Unit - ALU

We now turn our attention to a hardware component which will be needed to execute the arithmetic MIPS instructions **add**, **addi**, and **sub** for addition and subtraction. This same component will also execute the logical instructions, such as **andi**, **and**, **or**, **ori**, and **nor**. Because this component executes both arithmetic and logical operations, it is called an *arithmetic and logic unit*, or *ALU*.

We will design our ALU so that it performs six different functions, as shown in Fig 6.63. It will have two data inputs, each of which is a 32-bit bus. It will have a data output bus, which also has 32 bits, and a Z output (1 bit). The ALU will also need control inputs to determine its function. We use 4 control inputs, even though we have only 6 different operations (this allows for future expansion of our ALU).

Operation Select	Data Out	Z Out
0000	A AND B	1 if A AND B is all 0's
0001	A OR B	1 if A OR B is all 0's
0010	A + B	1 if A + B is all 0's
0110	A - B	1 if A - B is all 0's
0111	[unspecified]	1 if A < B
1100	A NOR B	1 if A NOR B is all 0's

Figure 6.63: Function table for the MIPS ALU. Each of the inputs, A and B, is a 32-bit bus. Z is a 1-bit output, indicating a zero result

The Z output is 1 when the data output is all 0's. It tells us whether an operation resulted in zero. For example, if the A data input is 0000000123 and the B data input is 00000000009, and the control input is 0010 (meaning 'add'), then the data output bus will be the sum of the two inputs: 0000012c. The Z output will be 0, indicating that the result is not zero.

Note that in Fig 6.63 the function selected with Operation Select = 2 = 0010_2, the function is addition; here the plus symbol does not represent logical OR. Also, for Operation Select = 7 = 0111_2, the Data Out is unspecified. This function puts a 1 on the Z output if A < B (with a two's complement comparison); otherwise it puts a 0 on the Z ouput. This function will be used to implement the MIPS instruction `slt` (set if less than).

A block diagram for our ALU is shown in Fig 6.64. Unlike other components the ALU is not rectangular. Most people use this shape because this component is fundamentally important to the CPU design, and the shape serves to remind us that it is basically operating on two busses to produce data on an output bus.

6.5.1 Exercises

1. Assume the A and B inputs to the 32-bit ALU defined in Fig 6.63 are:
   $A = 01020304_{16}$
   $B = f0f100ff_{16}$
   Show the 32-bit output (in hex) and the 1-bit Z output for each of the following operation select inputs (show unspecified outputs as question marks):

   (a) Operation select = 0000_2

   (b) Operation select = 0001_2

   (c) Operation select = 0010_2

   (d) Operation select = 0110_2

   (e) Operation select = 0111_2

   (f) Operation select = 1100_2

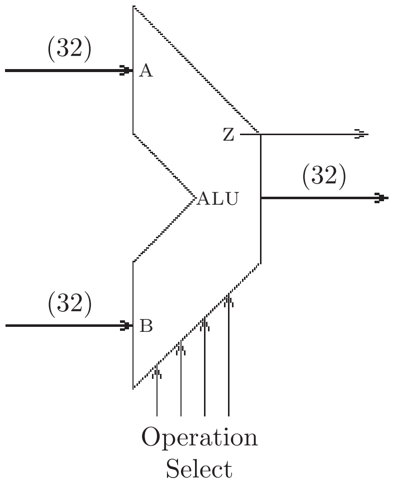


Figure 6.64: Block diagram of an ALU for the MIPS processor

(g) Operation select = 1111_2

2. Show a block diagram of an 16-bit ALU which has a maximum of 8 different functions that it can perform.

6.6 Construction of the ALU

In this section we get into the construction of a specific ALU - the one defined in Fig 6.63. This ALU will have 4 select inputs, though it has only 6 distinct functions. In our logic diagrams we show the gates needed to implement only one stage (i.e. 1 bit) of this 32-bit ALU. So, for example, the inputs to stage 3 will be A_3 and B_3, each of which is a single bit. We will also have a carry-in input to each stage shown as c_{in}. The carry-in to stage 0 will be the constant 0. We will also have a carry-out output from each stage, shown as c_{out}. The carry-out from stage i is connected to the carry-in of stage i+1.

Each stage will produce an output bit for each of the 6 functions. These 6 outputs are fed into a 16x1 multiplexer, which chooses one of its inputs based on the four operation select inputs. For the Z output the 32 output bits from the 32 stages are used as input to an OR gate which is then negated.

We show the implementation of each of the ALU functions separately, then combine them all into a single logic diagram.

6.6.1 ALU function AND: 0000

The implementation of the ALU AND function is simple. In each stage of the ALU we simply feed the A and B inputs into an AND gate, and ignore the c_{in} input, as shown in Fig 6.65.

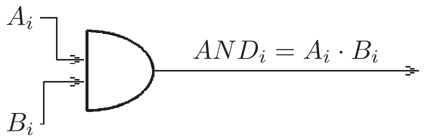


Figure 6.65: Implementation of the AND function for stage i of the ALU. Operation select is 0000 (raised dot indicates logical AND)

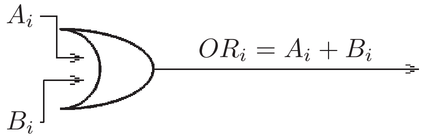


Figure 6.66: Implementation of the OR function for stage i of the ALU. Operation select is 0001 (+ indicates logical OR)

6.6.2 ALU function OR: 0001

The implementation of the ALU OR function is also simple. In each stage of the ALU we simply feed the A and B inputs into an OR gate, and ignore the c_{in} input, as shown in Fig 6.66.

6.6.3 ALU function Add: 0010

To add two numbers, whether considered unsigned or twos complement representation, the ALU function code is 0010. In each stage of the ALU we feed the A, B, and carry inputs into a Full Adder. For stage *i*, the carry-in is the carry-out of stage *i-1*, and the carry-out is the carry-in of stage *i+1*. For the very first stage, stage 0, the low order bit, the carry-in will be fixed at 0. The carry-out of the last stage, stage 31, the high order bit is the carry-out of the alu.[15]

In addition to the Sum output, which is the ALU output for add, it produces a c_{out} output which is to be connected to the c_{in} input of the next stage. This function is shown in Fig 6.67.

6.6.4 ALU function Subtract: 0110

To subtract the B input from the A input, we are assuming twos complement representation,[16] and the ALU function code is 0110. In order to implement subtraction, we utilize addition. Note that:

`A - B = A + (-B)`
`-B = B' + 1`   two's complement

[15] This can be useful when checking for an overflow condition, which exists when the carry-in to the high order bit is different from the carry-out of the high order bit.

[16] If assuming unsigned representation, one would have to make sure that the B input is smaller than the A input.

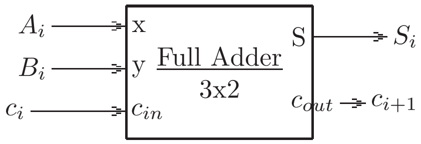


Figure 6.67: Implementation of the twos complement addition function for stage i of the ALU. Operation select is 0010. c_{out} is connected to c_{in} for stage i+1.

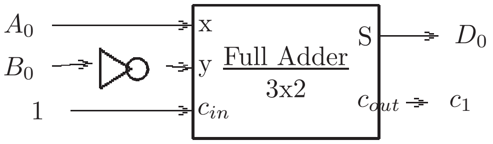


Figure 6.68: Implementation of the twos complement subtract function for stage 0 of the ALU. Operation select is 0110. c_{in} is the constant 1, and c_{out} is connected to c_{in} for stage 1.

Thus,
`A - B = A + B' + 1`
We complement the B input in each stage of the ALU. We choose to add in the 1 at the very first stage, since that stage will have no carry-in. In what follows the resulting difference is labeled D, i.e. `D = A - B` and we are implementing a 32-bit ALU:
$D_i = A_i + B_i'$ for i=1..31
$D_0 = A_0 + B_0' + 1$ for the low order bit

This means that the low order stage of the 32-bit ALU will be slightly different from the other 31 stages. Fig 6.68 shows the subtract operation for the low order stage of the ALU (i.e. stage 0) in which the carry-in to the full adder is 1 rather than 0. Fig 6.69 shows the subtract operation for all other stages of the ALU. In both of these figures we complement the B input to produce a subtraction rather than an addition.

6.6.5 ALU function NOR: 1100

The alu NOR function stands for Not OR, i.e. form the logical OR of the two operands, and negate the result. The ALU function code is 1100.
x NOR y = (x + y)'
We can do this easily by taking the output of the alu OR function and feeding it into an inverter.

In Fig 6.70 the input OR_i represents the i^{th} bit of the output for the OR

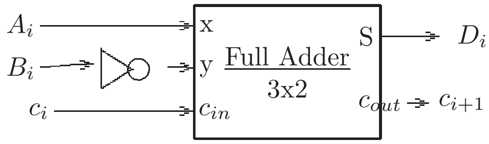


Figure 6.69: Implementation of the twos complement subtract function for stage i, where $i > 0$, of the ALU. Operation select is 0110. c_{in} is the carry-in, c_i, which is the carry-out of the previous stage. c_{out} is c_{i+1} and is the carry-in for the next stage

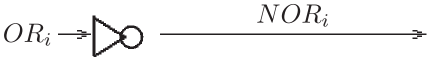


Figure 6.70: Implementation of the NOR function for stage i of the ALU. OR_i represents the output of the OR operation shown in Fig 6.66. Operation select is 1100.

operation (operation select 0001).

6.6.6 ALU: Putting it all together

We have shown how each function of the ALU can be implemented separately. All that remains is to combine all these logic diagrams into one logic diagram. In doing so, we will need logic to form the Z output of the ALU as specified in Fig 6.63. This includes setting the Z output for the *less than* operation (ALU operation code 0111).

The output for each function will form an input to a multiplexer which selects one of the inputs, based on the given ALU function code, as described in Fig 6.63. In Fig 6.71 we do not repeat all the logic shown in the preceding figures, but it should be understood that the outputs are as summarized below:

ALU Function Select	Operation	Name in logic diagram	See Fig
0000	And	AND	Fig 6.65
0001	Or	OR	Fig 6.66
0010	Add	S	Fig 6.67
0110	Subtract	D	Fig 6.69
1100	Nor	NOR	Fig 6.70

For example, the output labeled D_i in Fig 6.69 (the i^{th} bit of the result of a subtraction) is connected to the i^{th} bit of input 6 to the MUX in Fig 6.71.

Fig 6.71 is our first attempt at the ALU design. It handles all of the ALU operations except for the comparison operation (select code $7 = 0111_2$).

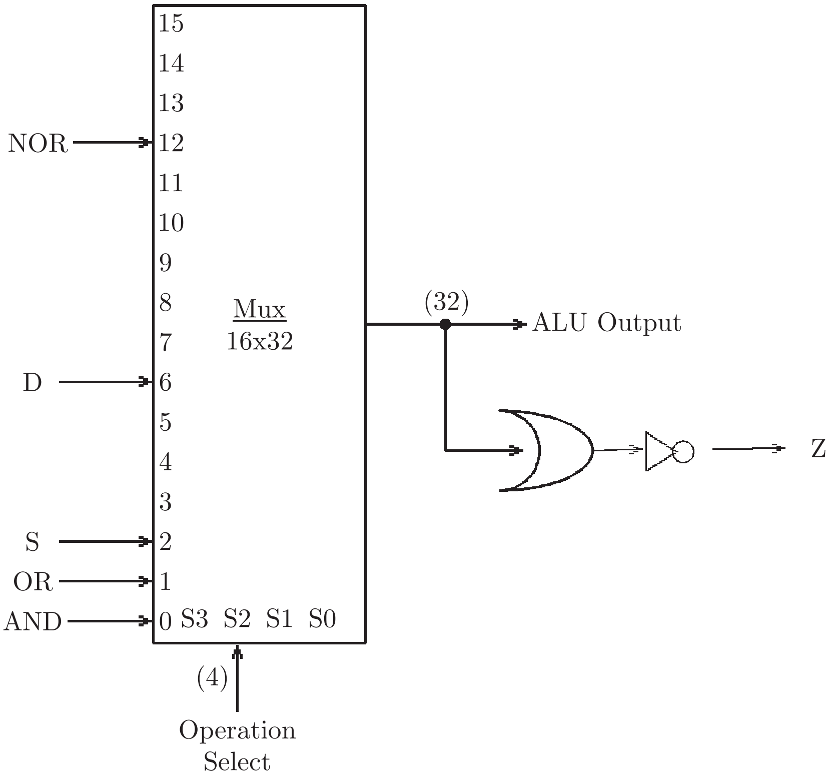


Figure 6.71: Using a multiplexer to select the desired output for the ALU. Each data input is a bus, and all data buses are 32 bits.

In addition to the data output, the ALU also has a 1-bit Z output. This output is 0 only when the data output bus is all zeros (see Fig 6.63). To accomplish this we need to OR all the output data bits, and complement the result:

$Z = (Out_{31} + Out_{30} + Out_{29} + ...Out_0)'$

as shown in Fig 6.72. This logic diagram shows a bus input to an OR gate. The intention is that the 32 bits of the bus form the 32 inputs to the OR gate. The output, when negated, will be 1 only if all 32 inputs are zeros.[17]

We now wish to complete the ALU by implementing the comparison operation. When the select code is 7 (0111_2), the data output bus is ignored, and the Z output should be a 1 only if A - B is negative, as described in Fig 6.63. This operation will be essential in implementing the MIPS `slt` (set if less than) instruction.

A - B will be negative if and only if the high order bit of the D (difference) input to the multiplexer is a 1 (two's complement representation). Thus for the Z output we wish to select either the Z output as shown in Fig 6.71 or the high order bit of the D input to the multiplexer (D_{31}), as determined by the operation select (not 7, or 7). To do this we will use a 2x1 multiplexer to choose either the output of the OR gate in Fig 6.71 or the high order bit of D. The select input to the 2x1 multiplexer is taken from the 4-bit operation select to the ALU (checking for 0111) with an AND gate. The result is shown in the final design of our ALU in Fig 6.72.

We make one final remark on the construction of the ALU. Note that if the operation select signal is 0001, for example, the ALU output is A OR B. But all the other ALU functions, shown in Fig 6.65 through Fig 6.70, are producing results at the same time. Then the multiplexer shown in Fig 6.71 selects the desired input to be sent to the output. This exposes a sharp distinction between hardware and software. If the ALU were to be implemented with software, it might be done with conditionals, i.e. `if` statements or a `switch` statement, to determine which computation is to be performed. Hardware is typically parallel, and software is typically sequential.[18]

6.6.7 Exercises

1. If an OR gate can have a maximum of 5 input lines, how many OR gates would be needed to generate the Z output in Fig 6.71?

2. How many Full Adders are needed in the implementation of the ALU?

3. Show how our ALU could produce the one's complement of the 32-bit word on the A input. Do not make any changes to the ALU designed

[17] There is generally a limit on the number of inputs to a single logic gate. In this case, several OR gates would be needed, with their outputs all fed into a single OR gate.

[18] With the development of computers with several cores, software is starting to become increasingly parallel, but much remains to be done to make effective use of multiple cores in a personal computer.

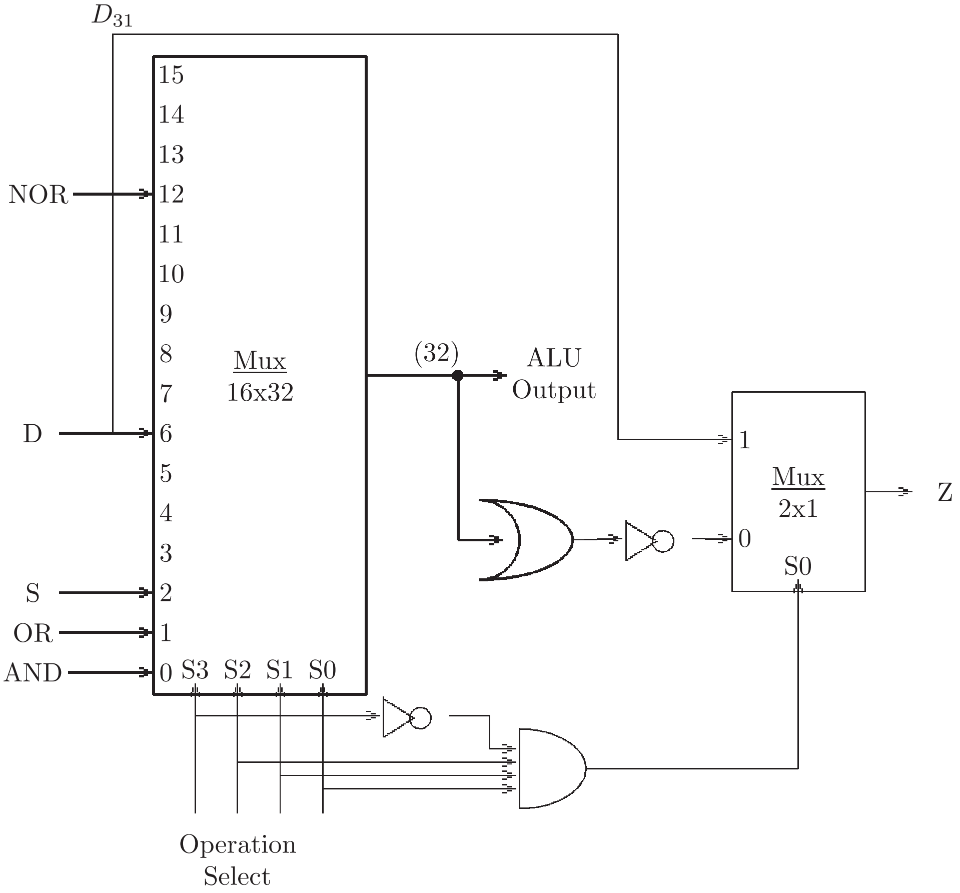


Figure 6.72: Completed implementation of the alu, showing Z output for the case where Operation Select = 7 ($A < B$).

in this section; instead show the appropriate operation select bits, and a possible value for the B input to the ALU.

4. Show how to modify the design of the ALU so as to include an XOR operation. Assume the operation select is 1011_2.

Chapter 7

MIPS Datapath

This chapter shows how digital components can be used to implement the MIPS instruction set. When the student has completed this chapter, it should be clear how software can be implemented and executed by hardware.

The name commonly given to the components, storage elements, and connections which accomplish this is the *datapath*. We first introduce the components which will be needed, and then show how they are connected to implement (a subset of) the MIPS instruction set.

7.1 Storage Components

In this section we take a look at the storage components in the MIPS datapath. This includes the *register file*, *data memory*, and the *instruction memory*. These storage components have a few things in common:

- They each store a sequence of 32-bit words.
- By providing an appropriate address, one of those 32-bit words can be extracted.
- They have *control signals* which, for example, can determine whether a word is to be written into the storage unit.
- They have at least one output bus.
- They have at least one input bus.

7.1.1 The Register File

The phrase *register file* is perhaps a misnomer because the word 'file' is usually associated with permanent storage, such as disk or flash storage. The MIPS register file consists of 32 registers, each of which is a full word (i.e. 32 bits). This is high-speed, but volatile[1] storage used by the CPU to execute MIPS

[1] Volatile storage requires power to operate, and all data is lost when power is lost.

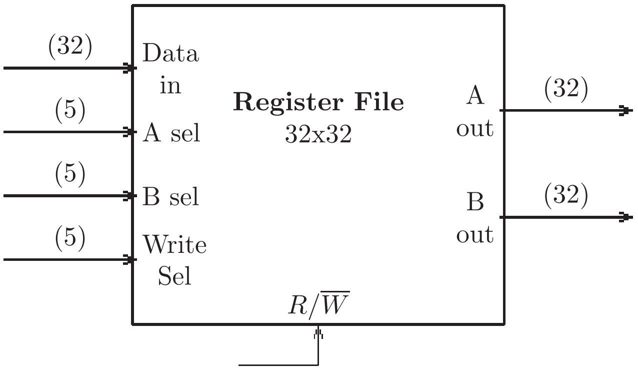


Figure 7.1: Register File for MIPS consists of 32 32-bit registers.

instructions. Each register has a unique address (or register number) in the range 0..31. Since there are 32 registers, the register addresses will be 5 bits in length. For example, the address 10111_2 specifies register number 23. A block diagram for the register file is shown in Fig 7.1.

The control signal to the register file labeled $R/\overline{W}$ determines whether a Read or Write operation is to take place. When the $R\overline{W}$ control signal is 1, the operation is Read, and when the $R/\overline{W}$ is 0, the operation is Write.[2]

The register file has two output busses, A and B. The register values which are placed on those busses are determined by the A Select and B Select inputs, respectively. For example, when the value of $R/\overline{W}$ is 1, signifying a Read operation, and A Select is $01101_2 = 13$, and B Select is $00000_2 = 0$, then the contents of register 13 is placed on the A output bus, and the contents of register 0^3 is placed on the B output bus. In this case the Data In and Write Select busses are ignored.

As another example, when the $R/\overline{W}$ is 0, signifying a Write operation, and Write Select is $01111_2 = 15$, the value on the Data In line is copied into Register 15. In this case the A Select, B Select busses are ignored.[4]

The table in Fig 7.2 shows examples of several other operations that are possible with the register file.

7.1.2 Data Memory and Instruction Memory

Since the Data Memory and Instruction Memory are similar in design, we cover both of them in this section. The Data Memory is used to store data which can be used as operands for the MIPS load and store instructions. The Instruction

[2] We follow the convention that the bar over the W in $R/\overline{W}$ serves as a reminder that the W operation is asserted with a 0 value.

[3] Recall that in the MIPS architecture register 0 always stores the value 0.

[4] Conceivably, the A Select and B Select inputs could select outputs for the A Out and B Out busses, respectively, even when $R/\overline{W}$ is 0, but that is beyond the scope of our current discussion.

$R/\overline{W}$	A select	B select	Write select	Operation
1	00111	10010		$Aout \leftarrow Reg[7], Bout \leftarrow Reg[18]$
1	11111	00010		$Aout \leftarrow Reg[31], Bout \leftarrow Reg[2]$
0			01111	$Reg[15] \leftarrow DataIn$
0			00011	$Reg[3] \leftarrow DataIn$

Figure 7.2: Examples of operations on the register file

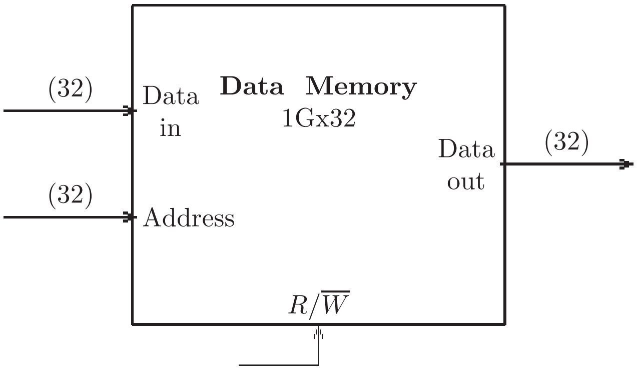


Figure 7.3: Data Memory unit for MIPS consists of 1G 32-bit words.

Memory is where the program instructions are stored. Each of these storage units consists of 1G x 32-bit words, where $1G = 2^{30}$.[5]

Each of these storage units has a 32-bit data input bus and a 32-bit output bus. Each also needs a 32-bit address input bus to select a word from the memory. The Data Memory also needs a $R/\overline{W}$ control signal. A block diagram of the Data Memory unit is shown in Fig 7.3. When the $R/\overline{W}$ control signal is 0, the word on the Data In bus is copied into the Data Memory at the address specified on the Address bus. The existing word at that address is clobbered (i.e. it is over-written). When the $R/\overline{W}$ control signal is 1, the word at the address specified on the Address bus is copied onto the Data Out bus.

The Instruction Memory differs from the Data Memory in that the Instruction Memory cannot be changed during program execution, i.e. a MIPS program cannot modify itself. Thus there is no `Instruction In` bus for the Instruction Memory. Also the Instruction Memory is always used in conjunction with two dedicated CPU registers, the Program Counter register (PC) and the Instruction Register (IR). A block diagram for the Instruction Memory, with its two dedicated registers is shown in Fig 7.4.

The PC register always stores the address of the next instruction to be executed; thus, it serves as an Address input to the Instruction Memory. Each time an instruction is executed by the CPU, the PC must be incremented by 4 to move to the next instruction in the program. When a branch or jump

[5] Recall that since a memory address is stored in a 32-bit word, the memory is *byte addressable*, and there are 4 bytes in a word we will have an address space of 2^{30} words.

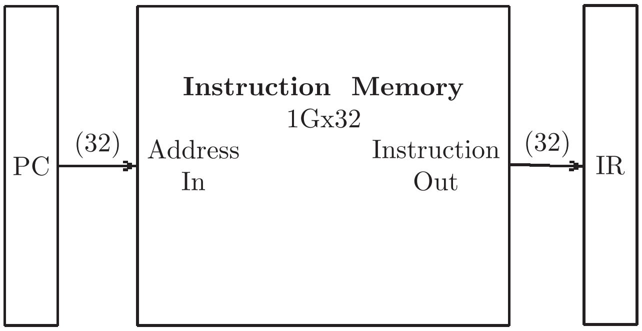


Figure 7.4: Instruction Memory unit for MIPS consists of 1G 32-bit instructions.

instruction is executed, the PC register is modified to store the target address for the branch or jump.

The IR stores the instruction currently being executed. Depending on the format of the instruction being executed, we imagine the bits of the IR being grouped into fields corresponding to the instruction fields. For example, if the instruction is an R format instruction, the fields of the IR are:

- bits 0-5: Function code
- bits 6-10: Shift amount
- bits 11-15: RD register
- bits 16-20: RT register
- bits 21-25: RS register
- bits 26-31: Op code

However, if the instruction is an I format instruction, the fields of the IR would be:

- bits 0-15: Immediate
- bits 16-20: RT register
- bits 21-25: RS register
- bits 26-31: Op code

And if the instruction is a J format instruction the fields of the IR would be:

- bits 0-25: Jump Address
- bits 26-31: Op code

Thus there will need to be busses connected to the IR for all of these fields, as shown in Fig 7.5. When we discuss the control unit in the MIPS datapath, we will show how these busses are selected, depending on the operation code of the instruction being executed.

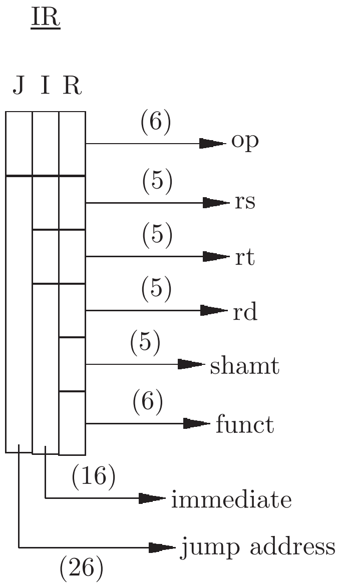


Figure 7.5: Instruction Register, showing the fields for the three instruction formats: J, I, and R.

7.1.3 Exercises

1. In an architecture with 16 registers in the register file, each of which is 8 bits:

   (a) What is the width of the Data In bus to the Register File?

   (b) What is the width of the A out and B out busses from the Register File?

   (c) What is the width of the A sel and B sel busses to the Register File?

   (d) What is the width of the Write sel bus to the Register File?

   (e) What is the width of the $R/\overline{W}$ signal to the Register File?

2. Draw a block diagram of the Register File described in the previous problem.

3. Assume that in the Register File described in Fig 7.1 each register has been initialized with its own address. Thus, register 0 contains 0, register 1 contains 1, register 2 contains 2, ... register 31 contains 31. Complete the table shown below by showing the register or bus which is changed (e.g. reg[8] = 00003212):

	$R/\overline{W} = 0$	$R/\overline{W} = 1$
DataIn= $00004c3f_x$		
A Sel = 0011_2		
B Sel = 0101_2		
Write Sel = 0001_2		

4. Refer to the Data Memory in Fig 7.3. Show the input signals and busses which are needed to:

   (a) Put the value of the word at location $04c00124_x$ onto the Data Out bus.

   (b) Change the word at location $4c001008_x$ to $ffffffff_x$.

   (c) Clear the word at location $4c001000_x$.

5. Show a block diagram for a byte addressable 16Gx64-bit Data Memory.

6. Refer to the Instruction Register (IR) which is loaded from the Instruction Memory (Figures 7.4 and 7.5). Show the values of the fields for each of the following 32-bit instructions. (Hint: The number of fields will depend on the instruction type.)

   (a) $03c42320_x$

   (b) $2d004501_x$

   (c) $0c000048_x$

7.2 Design of the Datapath

In this section we design the core component of the MIPS processor, which is called the *datapath*. We will make use of the components defined in the previous sections, as well as some of the logic devices described in chapter 6. Rather than presenting the entire datapath at once, we proceed in smaller increments, showing various portions of the datapath before putting it all together.

It is often desirable to synchronize the operation of several components in a combinational circuit. This is done with a *clock* signal. A clock signal is simply a (1-bit) control line which varies periodically between 0 and 1, as shown in Fig 7.6. The clock is then generally used as an input to an AND gate, for which the output goes into the component, as shown in Fig 7.7. In this way the component does not 'see' the input signal until the clock signal goes to 1. If all component inputs are connected in this way, all components generate their outputs at the same time, i.e. synchronously. For example, in Fig 7.7 we are using the clock signal to disable writing into the Register File until the clock signal rises to a 1. Since the Register File write operation takes place when $R/\overline{W}$ is 0, we must first negate the clock signal, then OR it with the input $r/\overline{w}$ signal. Thus the Register File is written only when the input $r/\overline{w}$ is 0 and the clock is 1. In this way all components which utilize the clock signal can be synchronized to be updated simultaneously.

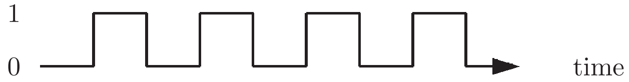


Figure 7.6: Clock signal

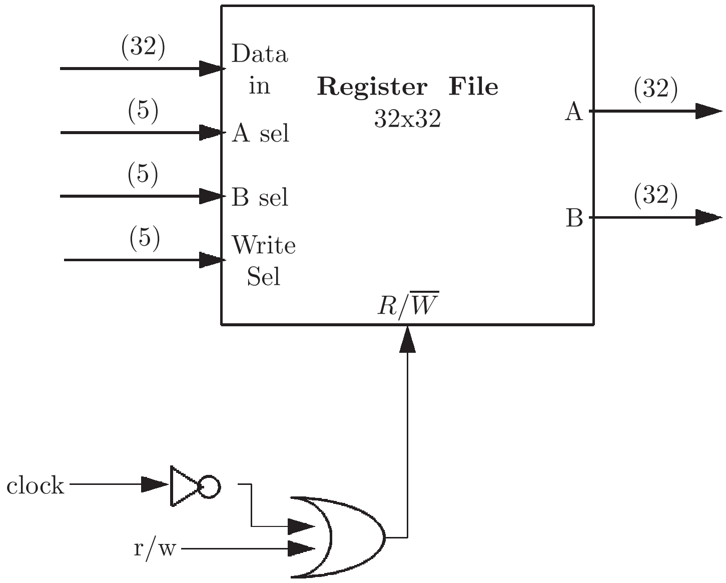


Figure 7.7: A clock signal is used to synchronize the effects of all components in the datapath

The period of the clock signal determines the speed of the processor. For example, a clock which goes through one complete cycle in 0.001 second has a frequency of 1000 cycles per second, or 1000 Herz = 1 KHz. A clock speed for a typical processor would be on the order of 10^9 cycles per second = 1 GHz.

In implementing the datapath, we could use more than one clock cycle to execute a single instruction. For example, an `add` instruction could load the ALU inputs from the register file during one clock cycle, and store the result back into the register file during the next clock cycle. This would be called a *two-cycle* data path. Alternatively it is possible to implement the register file in such a way that this instruction could be completed in one clock cycle, even if the destination register is the same as one of the operand registers. In this case we would have a *one-cycle* datapath (for which the clock speed would have to be slower to avoid clobbering an operand register before the result has been computed by the ALU). To simplify the exposition of the datapath we will be using a one-cycle design.

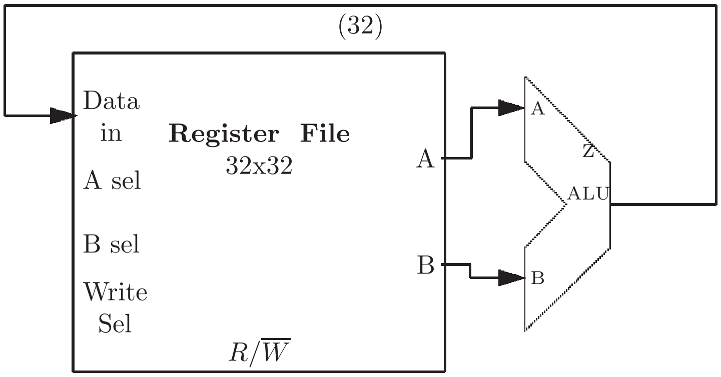


Figure 7.8: Connecting the Register File with the ALU in the datapath

7.2.1 Register File and ALU

We are now ready to begin development of the datapath, in a piecemeal fashion. We begin by showing how the Register File and the ALU are connected, in order to implement an R format instruction such as `add`. Fig 7.8 shows that the A and B output busses of the Register File form the two input busses to the ALU. Then the output of the ALU leads back into the Register File so that the operation result can be stored back into a register. Note that much is not shown in this diagram, including the A and B select inputs and the $R/\overline{W}$ input to the Register File; also the operation select input bus to the ALU. We will discuss these later.

7.2.2 Instruction Memory, Instruction Register, and Register File

We now turn our attention to the Instruction Memory, the Instruction Register, and the Register File. These three components are connected as shown in Fig 7.9. The output bus from the Instruction Memory is connected directly to the Instruction Register. This register stores the instruction which is currently executing. Fields of the Instruction Register are connected to the appropriate inputs of the Register File:

- The 5-bit `rs` field of the Instruction Register is connected to the `A select` input to the Register File. This is needed for all R format and I format instrtuctions.

- The 5-bit `rt` field of the Instruction Register is connected to the `B select` input to the Register File. This is needed for all R format and I format instrtuctions.

- The 5-bit `rd` field of the Instruction Register is connected to the `Write select` input to the Register File. This is needed for R format instrtuctions only.

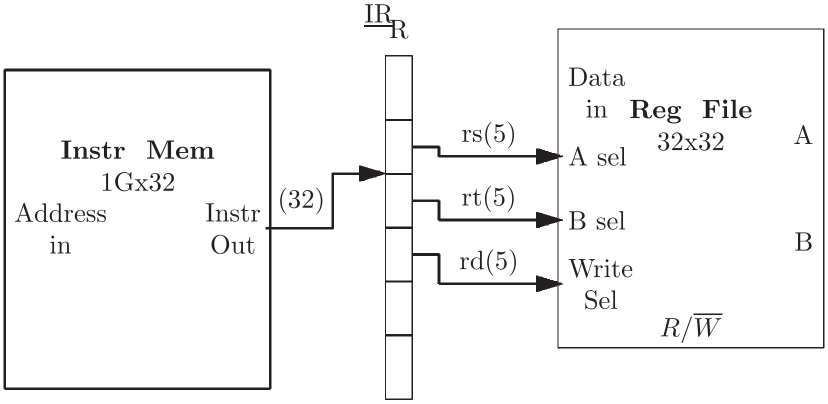


Figure 7.9: Connecting the Instruction Memory, Instruction Register and the Register File in the datapath, for an R format instruction, such as `add`

For example, if an `add` instruction is loaded into the Instruction Register, the `rs` and `rt` fields would specify which registers are to be added; their values would be placed on the A and B output busses of the Register File. Also, the `rd` field would specify which register is to store the result; thus it forms the `write select` input to the Register File.

7.2.3 Instruction Register, Register File, and Data Memory, for Load/Store

We now examine that part of the datapath which is responsible for *load* and *store* instructions, such as `lw` and `sw`. Recall from chapter 3 that these instructions copy a 32-bit word from Data Memory into a register, and copy a 32-bit word from a register into Data Memory, respectively. In chapter 4 we saw that in both cases the memory address was computed as the sum of the rs register and the 16-bit immediate field of the instruction. In the datapath we use the ALU to do this addition; the output of the ALU is connected to the `Address In` input of the Data Memory, as shown in Fig 7.10.

In the case of the `lw` instruction, the selected word from the Data Memory must be sent to the Register File; thus the `Data Out` output of the Data Memory must be connected to the `Data In` input to the Register File. Also the rt field specifies the register to be loaded, so it must be connected to the `Write Sel` input of the Register Files.

In the case of the `sw` instruction, the selected register from the Register File must be sent to the Data Memory; thus the rt field of the instruction is connected to the `B Sel` input of the Register File. The B output of the Register File bypasses the ALU and is connected to the `Data In` input to the Data Memory.

Note that the ALU is a 32-bit ALU and is expecting both inputs to be 32 bits; however, in Fig 7.10 the immediate field is only 16 bits. Thus we need a

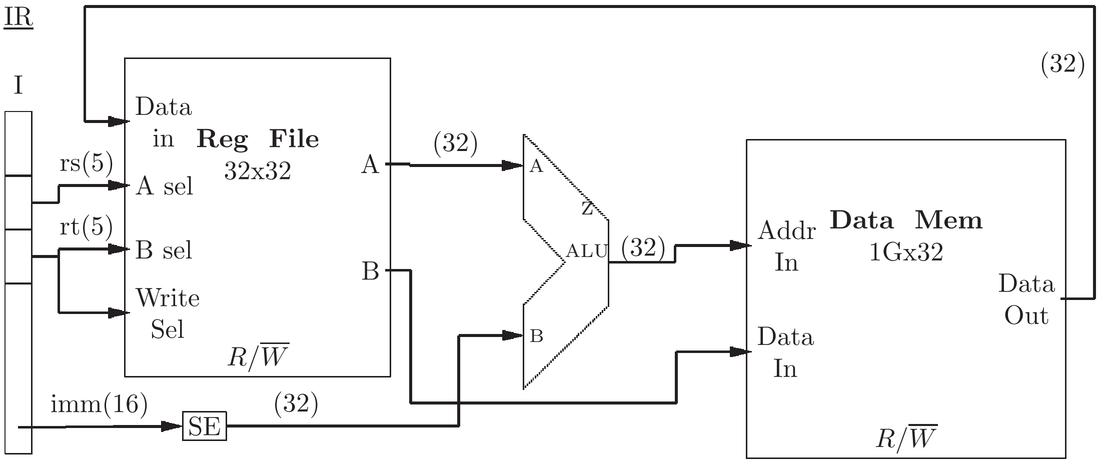


Figure 7.10: Connecting the Instruction Register, the Register File, the ALU, and the Data Memory in the datapath, for load/store instructions

sign-extend component (`SE`) to extend it to a full 32-bit bus, while preserving the sign.

Also note in Fig 7.10 that the rt field of the instruction is connected to both the `B Select` input and the `Write Select` input of the Register File. This is not a contradiction. A contradiction occurs when one target input is loaded from to different source outputs (as described in chapter 6). However, we should keep this in mind as we assemble the complete datapath, because there *will be* contradictions, which we will need to resolve.

7.2.4 Program Counter, Instruction Memory, and Transfer of Control

In this section we describe the function of the Program Counter (PC) register. This special purpose register stores the address (in the Instruction Memory) of the next instruction to be executed (as described above in the section on Instruction Memory and Data Memory). As each instruction is executed, the PC must be updated to contain the address of the next instruction to be executed. There are three possible ways that this happens:

- Normal flow of control is that instructions are executed sequentially, in the order in which they are stored in the Instruction Memory. I.e., the next instruction to be executed is the one which follows immediately in the Instruction Memory. In this case the datapath should increment the PC by 4 (since there are 4 bytes in an instruction, and the Instruction Memory is byte addressable). We will use a dedicated 32-bit adder, for which the A input is the constant 4, for this purpose. This aspect of the datapath is shown in Fig 7.11

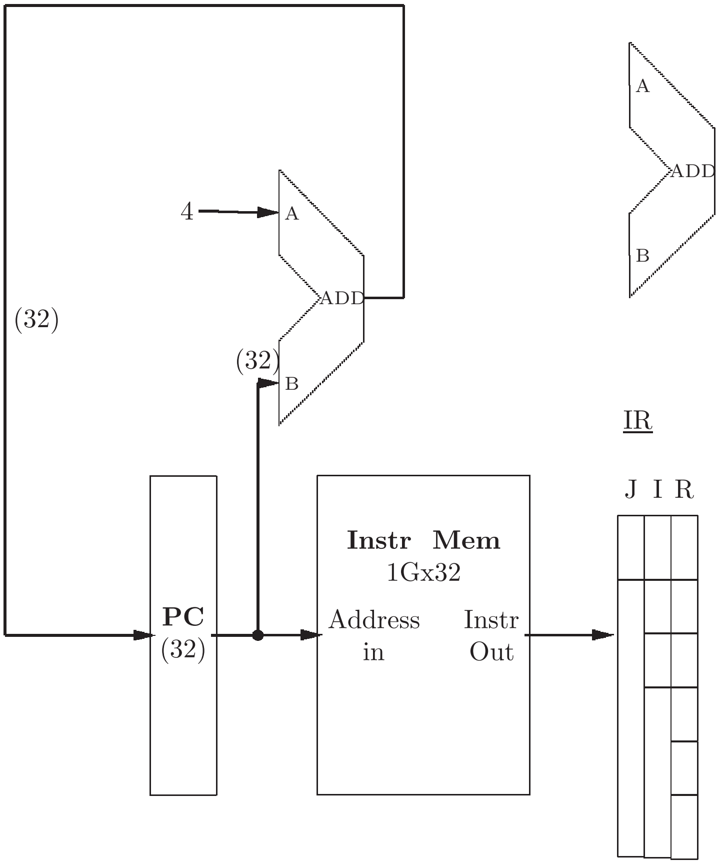


Figure 7.11: Connecting the PC, Instruction Memory, and Instruction Register for sequential transfer of control. A dedicated 32-bit adder is used to increment the PC by 4 (the second adder is not needed here).

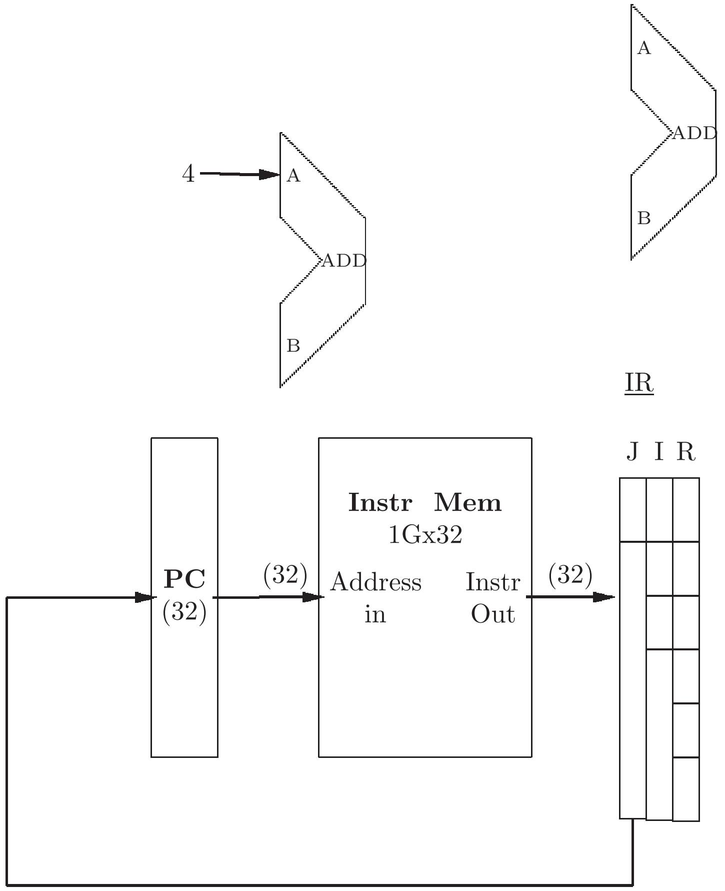


Figure 7.12: Connecting the PC, Instruction Memory, and Instruction Register for unconditional Jump instructions. The jump address is copied into the PC. (The dedicated adders are not used here)

- With an unconditional transfer of control, for a jump instruction (such as `j` or `jal`) we will need to alter the PC. The 26-bit address field, extended to a full 32 bits, is copied into the PC. The target of the jump is an *absolute* address. This aspect of the datapath is shown in Fig 7.12 (the 26-bit jump address would have to be padded with 6 0-bits to form a full 32 bit input to the PC).

- With a conditional transfer of control, a branch instruction such as (`beq` or `bne`) will need to alter the PC, but in this case the branch is to a *relative* address; thus we must add the current value of the PC to the immediate field, storing the result back into the PC. We will use another dedicated 32-bit adder for this purpose. This aspect of the datapath is shown in Fig 7.13. The 16-bit immediate field is a *relative* branch address, and it may be negative to allow for branching to a prior instruction. The adder is a 32-bit adder, and the immediate field is only 16 bits, thus the need to send the bus through a sign extend (`SE`) to produce a full 32-bit word, preserving the sign.

The three options described above are combined into a single diagram showing the connection of the PC, Instruction Memory, and Instruction Register (IR) in Fig 7.14. Note that when we combine all this logic into a single diagram, we produce *contradictions*, as described in chapter 6. These are points in the datapath where two or more sources come together, and are circled in Fig 7.14.

It is critical that we resolve these contradictions; each contradiction can be resolved with a multiplexer. Recall from chapter 6 that a multplexer (or MUX) with n select inputs can select one of 2^n input busses to be copied to a single output bus. In Fig 7.14 each contradiction can be resolved with a 2x1 MUX (i.e. a MUX with two 32-bit inputs, one 32-bit output, and a single select input). This is shown in Fig 7.15. In this diagram we label the two MUXes MUX BC (for BC instructions) and MUX J (for J or JAL instructions); in what follows we will need to refer to them individually.

Note that there is nothing connected to the select input for either of these multiplexers. They will both have to come from the Control Unit, to be described below.

7.2.5 Exercises

1. The speed of a CPU is determined by a clock signal.

   (a) For a 20 MHz clock ($1MHz = 10^6Hz$): How many clock cycles are there in 3.7 seconds?

   (b) What is the speed of a clock which issues 6,750 pulses every second?

2. The diagram in Fig 7.8 is designed to execute which of the following MIPS instructions?

   (a) `add`

   (b) `lw`

   (c) `or`

   (d) `bc`

   (e) `j`

3. (a) What are the names (and widths, in bits) of the unlabeled fields in the IR shown in Fig 7.9?

   (b) Briefly explain why they are not relevant in this diagram?

4. (a) In Fig 7.10 which of the ALU operations should be selected? Refer to Fig 6.63.

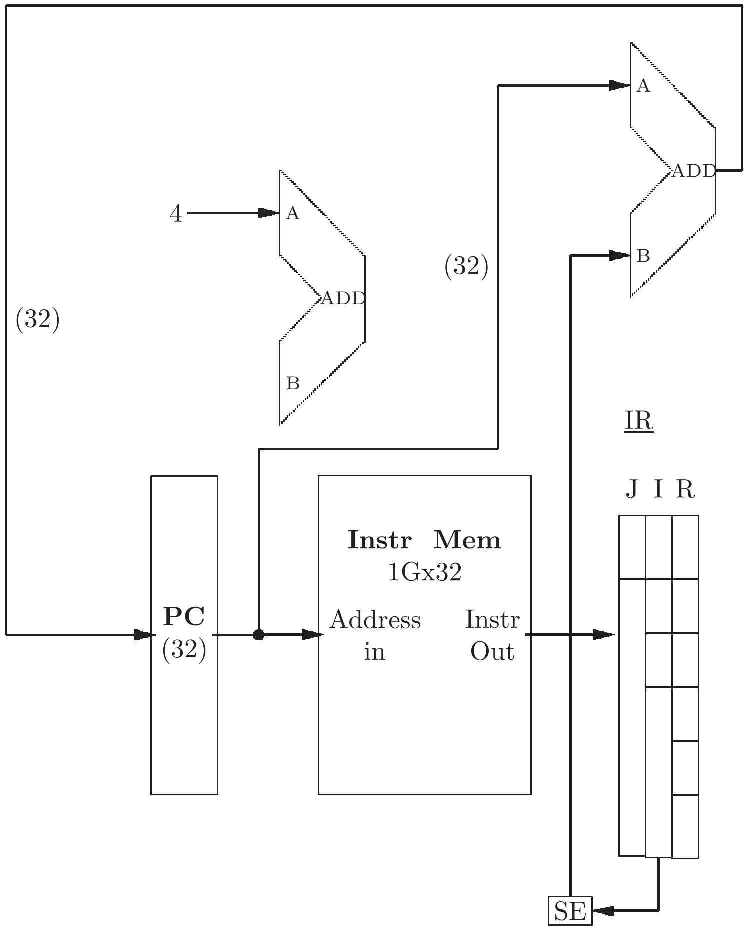


Figure 7.13: Connecting the PC, Instruction Memory, and Instruction Register for conditional branch instructions. A dedicated 32-bit adder is used to increment or decrement the PC by the relative branch address (the other adder is not used here).

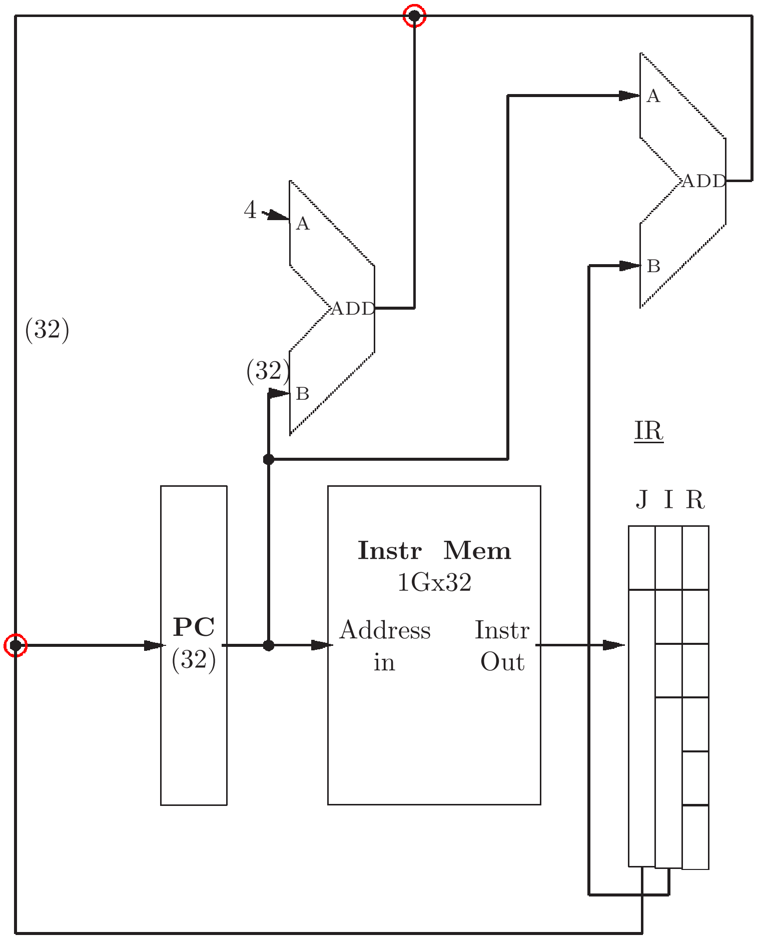


Figure 7.14: Connecting the PC, Instruction Memory, and Instruction Register. Dedicated 32-bit adders are used for transfer of control. Contradictions are circled.

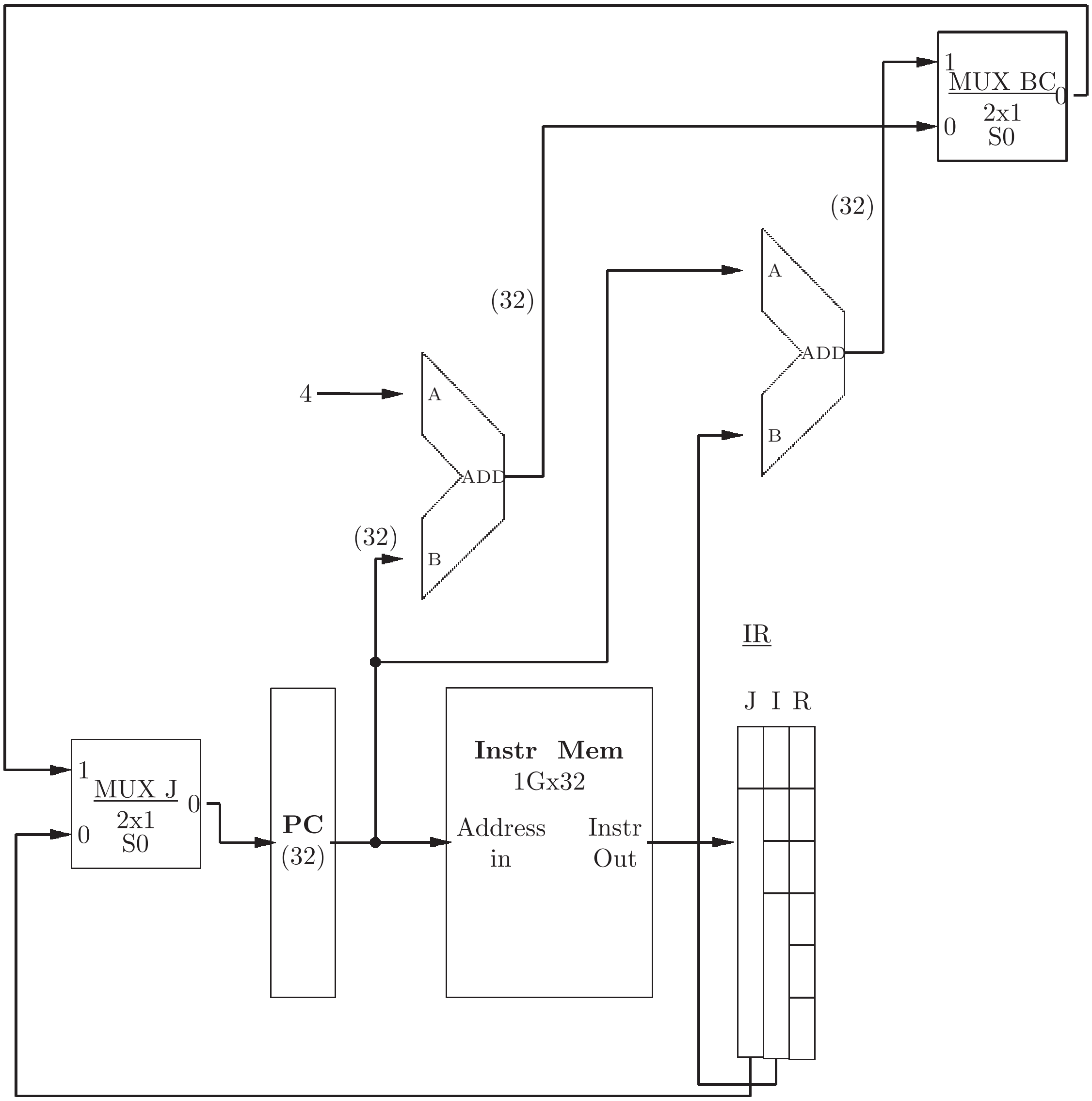


Figure 7.15: Connecting the PC, Instruction Memory, and Instruction Register. Dedicated 32-bit adders are used for transfer of control. Contradictions have been resolved with multiplexers.

(b) In Fig 7.10 there is an `SE` component. If the input to `SE` is $812a_x$, what is the output?

5. In Fig 7.10 there is a $R/\overline{W}$ signal to the Register File ane to the Data Memory.

   (a) What should be the value of each of those signals if an `add` instruction is being executed?

   (b) What should be the value of each of those signals if a `beq` instruction is being executed?

   (c) What should be the value of each of those signals if a `lw` instruction is being executed?

   (d) What should be the value of each of those signals if a `sw` instruction is being executed?

6. In Fig 7.12 the jump address field is only 26 bits, but the PC is 32 bits. Show a better version of this diagram to rectify this problem. (See the section on busses in chapter 6)

7. In Fig 7.10 explain briefly why the Sign Extend `SE` component is needed for conditional branches.

8. In Fig 7.16 Component 1 has one output, and Component 2 has two outputs. Identify the contradiction(s), if any, resolve using multiplexer(s) and redraw the diagram, if necessary. (See Fig 7.14 and Fig 7.15)

7.3 The Control Unit

As mentioned, there are a few critical missing signals in the datapath, as we have described it thus far. Specifically, there need to be:

- $R/\overline{W}$ signals for the Data Memory and the Instruction Memory.
- Control signals for the ALU
- Select signals for the multiplexers introduced so far.

The datapath logic for the production of these signals is called the *control unit.* Since these signals depend on the instruction being executed, the control unit will take as input the `op` field of the instruction; i.e. the operation code. In cases where several different R format instructions share the same op code, the control unit will also examine the `funct` field of the instruction. Using thses inputs the control unit will produce the necessary select and control signals for the datapath. We examine these output signals below for a subset of the MIPS instruction set. Specifically, we will handle the following:

- Load and store (lw and sw)

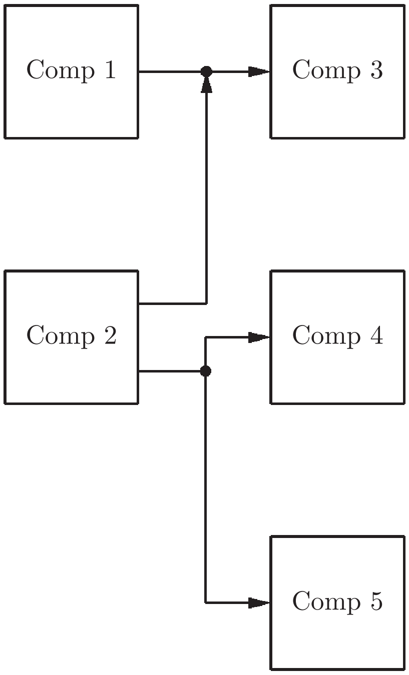


Figure 7.16: Exercise to identify and resolve contradiction(s), if any

- R format instructions: add, sub, and or, slt
- Conditional branch: beq, bne
- Unconditional jump: j

.

Control Unit Output to Data Memory $R/\overline{W}$

The Data Memory is written by a Store Word (`sw`) instruction. All other instructions should not write to the Data Memory. Thus the output should be 0 only when the opcode is $2b_x = 10\ 1011_2$. For all other opcodes, the output should be 1, to ensure that the Data Memory is not written. This signal from the control unit is called `DW`.
DW = ($op_5\ op'_4\ op_3\ op'_2\ op_1\ op_0$)' in which *op* represents the 6-bit opcode field of the instruction. This can be done with a single AND gate with 5 inputs, and three inverters.

Control Unit Output to Register File $R/\overline{W}$

The Register File is written by most R format instructions, such as `add` and `sub`. Also, the the Load Word (`lw`) instruction will need to write to the Register File.

We might be tempted to set this output to 1 for any R format instruction, but that would not be correct if we were to expand our subset of instructions. The `jr` instruction is R format, and there are multiply and divide instructions which are R format, none of which should write to the Register File. For our subset, we will set this output to 0 when the opcode is $23_x = 100011_2$ (lw) or the opcode is 0 and the function code is $20_x = 10\ 0000_2$ (add), $22_x = 10\ 0010_2$ (sub), $24_x = 10\ 0100_2$ (and), $25_x = 10\ 0101_2$ (or), or $2a_x = 10\ 1010_2$ (slt), and set the output to 1 in all other cases. We call this output from the control unit `RegW`. Thus for our subset

$$\begin{aligned}
&\text{RegW} = op_5\ op_4'\ op_3'\ op_2'\ op_1\ op_0\ + \\
&op_5'\ op_4'\ op_3'\ op_2'\ op_1'\ op_0'\ (f_5\ f_4'\ f_3'\ f_2'\ f_1'\ f_0'\ + \\
&\qquad f_5\ f_4'\ f_3'\ f_2'\ f_1\ f_0'\ + \\
&\qquad f_5\ f_4'\ f_3'\ f_2\ f_1'\ f_0'\ + \\
&\qquad f_5\ f_4'\ f_3'\ f_2\ f_1'\ f_0\ + \\
&\qquad f_5\ f_4\ f_3'\ f_2'\ f_1\ f_0'\ )
\end{aligned}$$

in which *op* represents the 6-bit opcode field of the instruction and *f* represents the 6-bit function code field of the instruction. This can be done with seven AND gates, five OR gates, and a multitude of inverters.

Control Unit Output to ALU Operation Select - 4 bits

Our control unit must also tell the ALU which operation is to be performed, the result of which is to be put onto the ALU output bus. The control unit will have 4 output bits for this purpose. There is an ALU operation for each R format instruction in our subset. The ALU is also used for the I format instructions `lw` and `sw`; the ALU is used to form an absolute address to the Data Memory as the sum of the rs register and the immediate field. For these instructions, the ALU must perform an `add` operation. are shown below:

Instruction	Op code	Function code	ALU Operation Code
add	0	$20_x = 10\ 0000_2$	0010_2
sub	0	$22_x = 10\ 0010_2$	0110_2
and	0	$24_x = 10\ 0100_2$	0000_2
or	0	$25_x = 10\ 0101_2$	0001_2
slt	0	$2a_x = 10\ 1010_2$	0111_2
lw	$23_x = 10\ 0011_2$		0010_2
sw	$2b_x = 10\ 1011_2$		0010_2

In the above table we see that the high order bit (bit 3) of the ALU operation code is always 0.

$ALU_3 = 0$

Bit 2 is 1 for the `sub` and `slt` instructions only:

$$\begin{aligned}
&ALU_2 = op_5'\ op_4'\ op_3'\ op_2'\ op_1'\ op_0' \\
&\qquad (\ f_5\ f_4'\ f_3'\ f_2'\ f_1\ f_0'\ \ + f_5\ f_4'\ f_3\ f_2'\ f_1\ f_0'\ \ )
\end{aligned}$$

For bit 1 of the ALU, it is 0 only for the `add`, `sub`, and `slt` instructions:

$$ALU_1 = op_5' \ op_4' \ op_3' \ op_2' \ op_1' \ op_0'$$
$$(\ f_5 \ f_4' \ f_3' \ f_2' \ f_1' \ f_0' + f_5 \ f_4' \ f_3' \ f_2' \ f_1 \ f_0' + f_5 \ f_4' \ f_3 \ f_2' \ f_1 \ f_0' \ )$$

Finally, bit 0 of the output to ALU operation select is 1 for the `or` and `slt` instructions only.

$$ALU_0 = op_5' \ op_4' \ op_3' \ op_2' \ op_1' \ op_0'$$
$$(\ f_5 \ f_4' \ f_3' \ f_2 \ f_1' \ f_0 \ + f_5 \ f_4' \ f_3 \ f_2' \ f_1 \ f_0' \ )$$

We have shown boolean expressions for each of the four output lines from the control unit to the ALU operation select. The logic diagram can be constructed from these four expressions.

Control Unit Output to Multiplexers

The only remaining control unit outputs are for the two multiplexers in our datapath; they are labeled MUX BC and MUX J in Fig 7.15. Each of these multiplexers requires one output signal from the control unit.

The multiplexer MUX BC takes input from two busses:

- Input bus 0 - the dedicated adder which adds 4 to the current PC value.

- Input bus 1 - the dedicated adder which adds the immediate field to the current PC value (for conditional branch to a relative address).

The control signal to this MUX should be 1 when a conditional branch is executing, and 0 for all other instructions except for an unconditional jump instruction. For a jump instruction we don't care about the output of the MUX BC; it is ignored. The conditional branch instructions are `beq` and `bne`.[6]

However, for these instructions the branch is *conditional*; the branch should take place only if the registers being compared are equal (for `beq`) or unequal (for `bne`). This comparison can be accomplished by instructing the ALU to do a subtract operation, and testing the Z output (for a zero result from the subtraction). If the ALU output is zero and the instruction being executed is `beq`, then the control signal to MUX BC should be a 1. If the ALU output is not zero, and the instruction being executed is `bne`, then the control signal to MUX BC should also be a 1; otherwise it should be a 0.

The select input to the MUX BC can thus be written as:
MUXBC = BEQ Z + BNE Z'
where

- BEQ is a signal from the Control Unit that the instruction being executed is `beq`

- BNE is a signal from the Control Unit that the instruction being executed is `bne`

[6] Other conditional branches, such as `ble` and `bgt` are actually pseudo-ops and are not in the MIPS instruction set (see chapter 3).

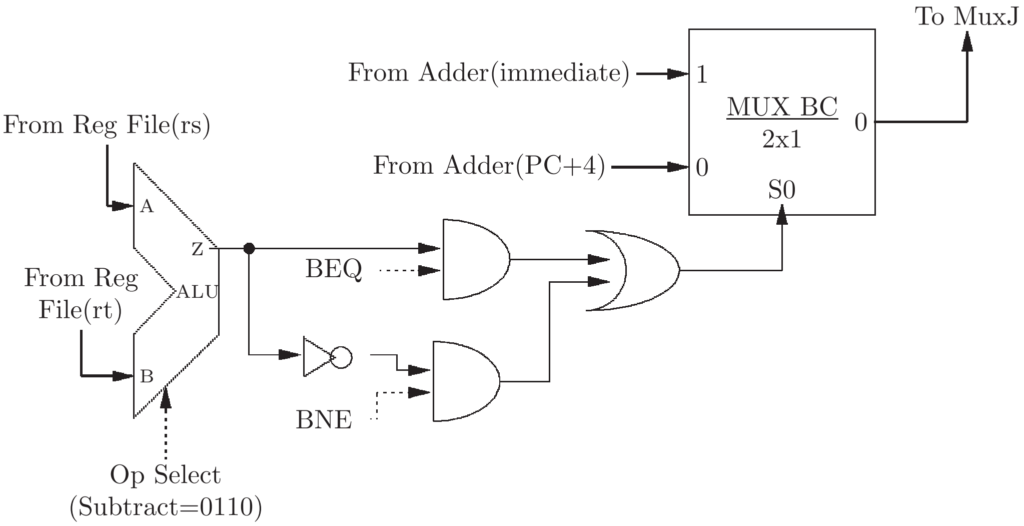


Figure 7.17: Generating the Select input for the BC Multiplexer, from the ALU Z output and the control unit. Signals from control unit are dashed arrows.

- Z is the Z output signal from the ALU indicating that the output of the ALU is 0.

This means we will need two AND gates,an Inverter, and an OR gate in our datapath, forming the input to the MUX BC multiplexer. The select input to the BC multiplexer is shown in Fig 7.17.

We can now write the boolean expressions for the Control Unit outputs described above. The op codes for the conditional branch instruction are `be` $= 04_x = 00\ 0100_2$ and `bne` $= 05_x = 00\ 0101_2$.

BEQ $= op_5'\ op_4'\ op_3'\ op_2\ op_1'\ op_0'$

BNE $= op_5'\ op_4'\ op_3'\ op_2\ op_1'\ op_0$

We consider the MUX J multiplexer next. It decides whether the PC should be loaded with the jump address from an unconditional jump (input 0), or loaded from the MUX BC output (input 1). The control unit will generate a signal, J=1, if the instruction being executed is an unconditional jump. Thus, all that is needed is to use that as the select signal to MUX J, as shown in Fig 7.18.

Data In to Register File

Another contradiction which needs to be resolved is at the Data In to the Register File. By comparing Fig 7.8 (for R format instructrions) and Fig 7.10 (for the `lw` instruction), we see two busses leading in to the Data In input for the

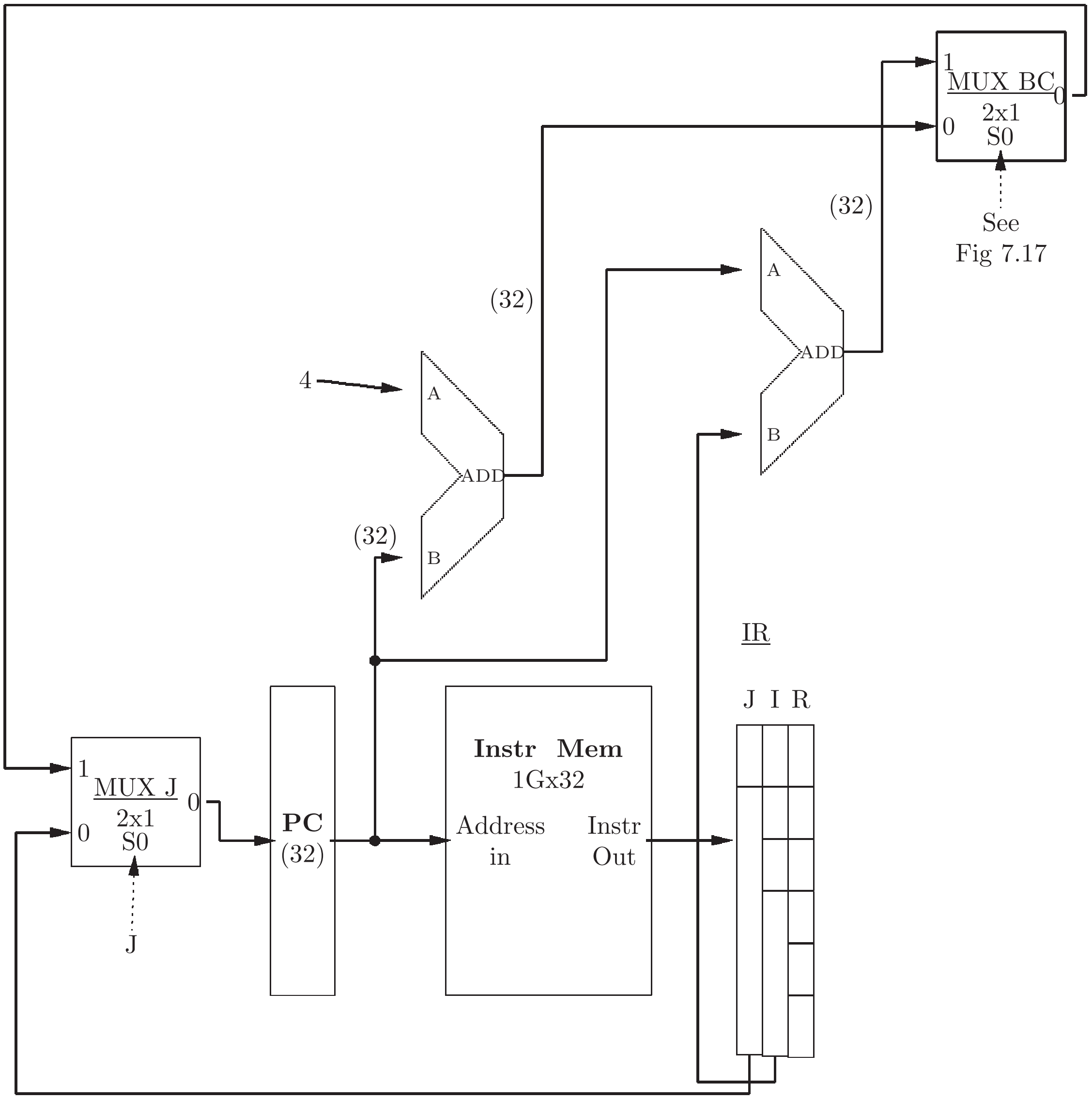


Figure 7.18: Select signal to the MUX J multiplexer is the signal labeled J, from the control unit.

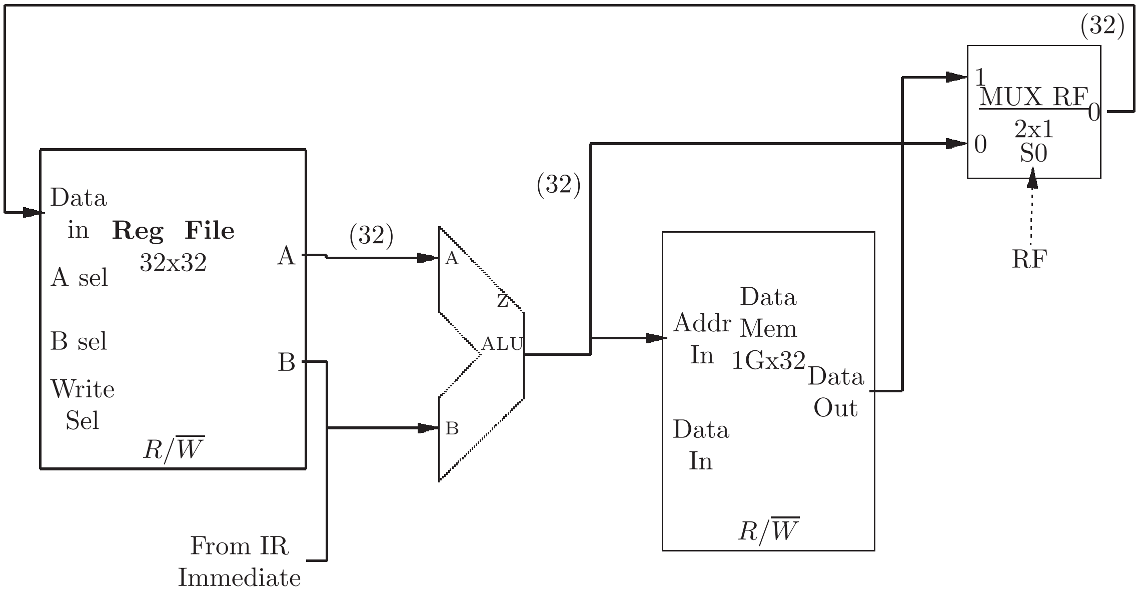


Figure 7.19: Using a multiplexer to resolve the contradiction at the Data In to the Register File. The RF signal is produced by the control unit.

Register File. This contradiction must be resolved with another multiplexer. We call it `MUX RF`, and it is shown in Fig 7.19.

IR to Write Select

In comparing Fig 7.9 with Fig 7.10 we see anothe contradiction which needs to be resolved. This is at the Write Select input to the Register File, which determines which register is to receive the result of the operation. For an R format instruction, such as `add`, the Write Select should come from the rd field of the IR. For the I format instruction `lw` (load word) the Write Select comes from the rs field of the IR. Thus a multiplexer is needed to resolve this contradiction as shown in Fig 7.20. The control unit will select the `rs` field only if the instruction is `lw`. Otherwise it will select the `rd` field.

B Input to the ALU

Before showing the detailed logic of the control unit, there is one more contradiction which needs to be resolved. In the case of an R format instruction, the B input to the ALU is taken from the rs register (through the B output of the Register File), as shown in Figs 7.8 and 7.9. However, for load and store instructions (`lw` and `sw`), the B input to the ALU is taken from the immediate field of the instruction so the effective address can be computed as the sum of the rs register and the immediate field, as shown in Fig 7.10. This contradiction

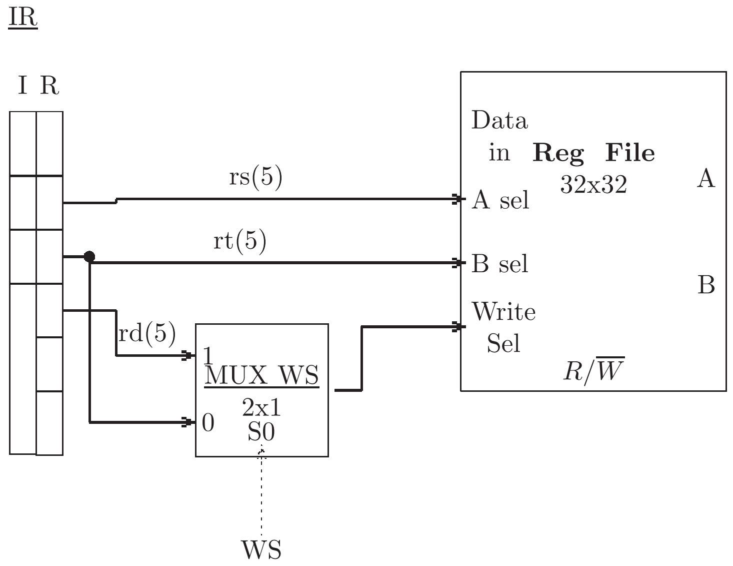


Figure 7.20: Using a muiltplexer to resolve the contradiction at the Write Select input to the Register File. The WS signal is produced by the control unit.

needs to be resolved with yet another multplexer. We call this multiplexer `MUX ALU`, and its inputs are the B output of the Register File and the immediate field of the instruction.[7] The output of the multiplexer forms the B input to the ALU. The select signal for this multiplexer is called `ALU B`, and it comes from the control unit as shown in Fig 7.21.

Logic for the Control Unit

In the preceding sections we have seen the need for several control signals from the control unit. The inputs to the control unit are the operation code (`op`) and function code (`func`) fields from the Instruction Register (IR). These determine the instruction being executed. All that remains is to show how the control unit outputs are computed from its inputs. We summarize the contol unit outputs below:

- RegW: $R/\overline{W}$ signal to the Register File. 1=Read, 0=Write.
- DW: $R/\overline{W}$ signal to the Data Memory. 1=Read, 0=Write.
- ALUB: Select signal to the ALU B multiplexer. 1=select B register, 0=select immediate field.
- RF: Select signal to the MUX RF multiplexer. 1=select Data Memory out, 0=select ALU out.

[7] The immediate field needs to extended to a full 32 bits, with a *sign extend* to preserve the sign of the value.

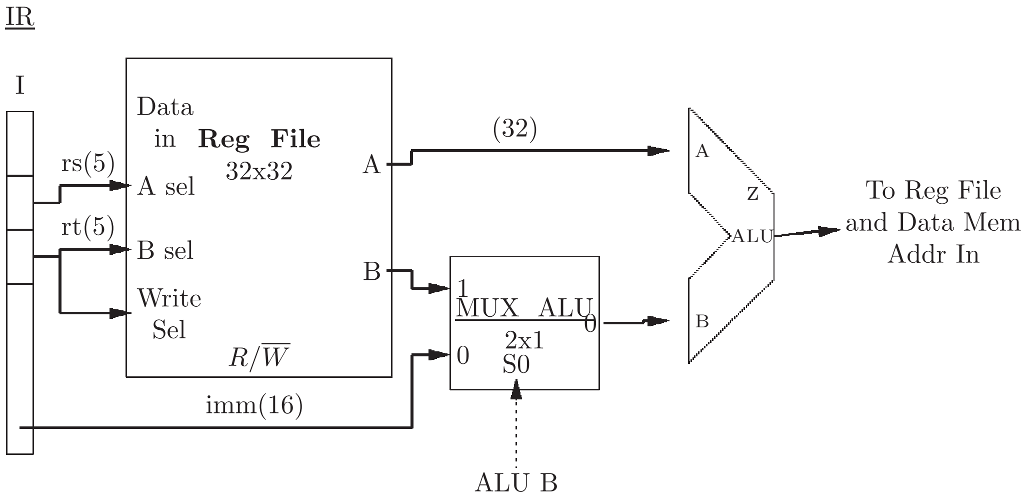


Figure 7.21: Using a multiplexer to resolve the contradiction on the B input to the ALU. The ALU B signal is produced by the control unit.

- WS: Select signal to the MUX WS multiplexer. 1=select rd field, 0=select rt field.
- J: Select signal to the MUX J multiplexer. 1=select MUX BC out, 0=select jump address from IR.
- BEQ: Used by the MUX BC multiplexer. 1=current instruction is a `be` instruction, 0=current instruction is not a `be` instruction.
- BNE: Used by the MUX BC multiplexer. 1=current instruction is a `bne` instruction, 0=current instruction is not a `bne` instruction.
- ALUOP: ALU Operation Select (4 bits). See Fig 6.63.

We finally have all the information we need to build the control unit. We do this by examining each instruction in our subset, and deciding what each output of the control unit should be for that instruction. This is shown as a table in Fig 7.22. The value for `RF` for the `slt` instruction is left as an exercise.

Note that some of the entries in the table are question marks. These represent *don't care* values. These values could be either 0 or 1; it doesn't matter. This gives us more flexibility and can simplify the logic for the control unit. As an example, for a store word (`sw`) instruction, the RF signal is a don't care. This signal determines whether the Register File is loaded from the ALU output or from the Data Memory. But for a store word instruction, the Register File is not written, hence it does not matter what comes out of the MUX RF multiplexer. In general, whenever the RegW signal is 1 (the Register File is not written), the RF signal to MUX RF will be a don't care.

Instruction	RegW	DW	ALUB	RF	WS	J	BEQ	BNE	ALU OP
and	0	1	1	0	0	1	0	0	0000
or	0	1	1	0	0	1	0	0	0001
add	0	1	1	0	0	1	0	0	0010
sub	0	1	1	0	0	1	0	0	0110
nor	0	1	1	0	0	1	0	0	1100
slt	0	1	1		0	1	0	0	0110
lw	0	1	0	1	1	1	0	0	0010
sw	1	0	0	?	?	1	0	0	0010
beq	1	1	1	?	?	1	1	0	0110
bne	1	1	1	?	?	1	0	1	0110
j	1	1	?	?	?	0	?	?	????

Figure 7.22: Table showing the outputs of the control unit for each instruction. Don't cares are indicated by question marks.

We now take one instruction, the `and` instruction and explain each control unit output for that instruction.

- RegW must be 0 because the result of the `and` instruction needs to be written to the Register File.
- DW must be 1 because the `and` instruction should not write to the Data Memory.
- ALUB must be 1 so that the B input to the ALU is taken from the B output of the Register File, rather than the immediate field of the instruction.
- RF must be 0 because the output of the ALU must be written back to the Register File.
- WS must be 0 because the rd field of the instruction specifies the register to be written.
- J must be 1 because `and` is not a jump instruction.
- BEQ must be 0 because `and` is not a `beq` instruction.
- BNE must be 0 because `and` is not a `bne` instruction.
- ALUOP must be 0000 because that is the ALU operation code for logical AND.

To build the control unit we should first write a boolean expression for each output, using the instruction op code and function code (for R format instructions). For example, the table in Fig 7.22 shows that the RegW signal should be 1 when the instruction is any one of the following:

- `sw` (op code = $2b_x = 10\ 1011_2$),

- `beq` (op code = $04_x = 00\ 0100_2$),
- `bne` (op code = $05_x = 00\ 0101_2$),
- `j` (op code = $02_x = 00\ 0010_2$).

Thus the boolean expression for the RegW output signal is

$$\begin{aligned} RegW \ = \ & op_5\ op_4'\ op_3\ op_2'\ op_1\ op_0\ + \\ & op_5'\ op_4'\ op_3'\ op_2\ op_1'\ op_0'\ + \\ & op_5'\ op_4'\ op_3'\ op_2\ op_1'\ op_0\ + \\ & op_5'\ op_4'\ op_3'\ op_2'\ op_1\ op_0' \end{aligned}$$

The student should be able to write the boolean expressions for the other output signals. The logic diagrams are left as exercises.

This completes our design of the MIPS datapath. The complete diagram is shown in Fig 7.23 and Fig 7.24.

7.3.1 Exercises

1. The Control Unit output to the Register File $R/\overline{W}$ input is called `RegW`, and is shown below:
   RegW = $op_5\ op_4'\ op_3'\ op_2'\ op_1\ op_0\ +$
   $$\begin{aligned} op_5'\ op_4'\ op_3'\ op_2'\ op_1'\ op_0'\ (& f_5\ f_4'\ f_3'\ f_2'\ f_1'\ f_0'\ + \\ & f_5\ f_4'\ f_3'\ f_2'\ f_1\ f_0'\ + \\ & f_5\ f_4'\ f_3'\ f_2\ f_1'\ f_0'\ + \\ & f_5\ f_4'\ f_3'\ f_2\ f_1'\ f_0\ + \\ & f_5\ f_4\ f_3'\ f_2'\ f_1\ f_0'\ ) \end{aligned}$$
   in which *op* represents the 6-bit opcode field of the instruction and f represents the 6-bit function code. Rewrite this expression to accommodate the jump register (`jr`) instruction and the jump and link (`jal`) instruction.

2. The table used to determine the Control Unit output to the ALU operation select is shown below.

Instruction	Op code	Function code	ALU Operation Code
add	0	$20_x = 10\ 0000_2$	0010_2
sub	0	$22_x = 10\ 0010_2$	0110_2
and	0	$24_x = 10\ 0100_2$	0000_2
or	0	$25_x = 10\ 0101_2$	0001_2
lw	$23_x = 10\ 0011_2$		0010_2
sw	$2b_x = 10\ 1011_2$		0010_2

   (a) Include another row in this table for the set if less than (`slt`) instruction.

   (b) Rewrite the boolean expressions, as needed, for the Control Unit's 4-bit output to the ALU.

3. Describe in words how the two 2x1 multplexers in Fig 7.15 can be combined into a single 4x1 multiplexer (with one of its 4 inputs unused).

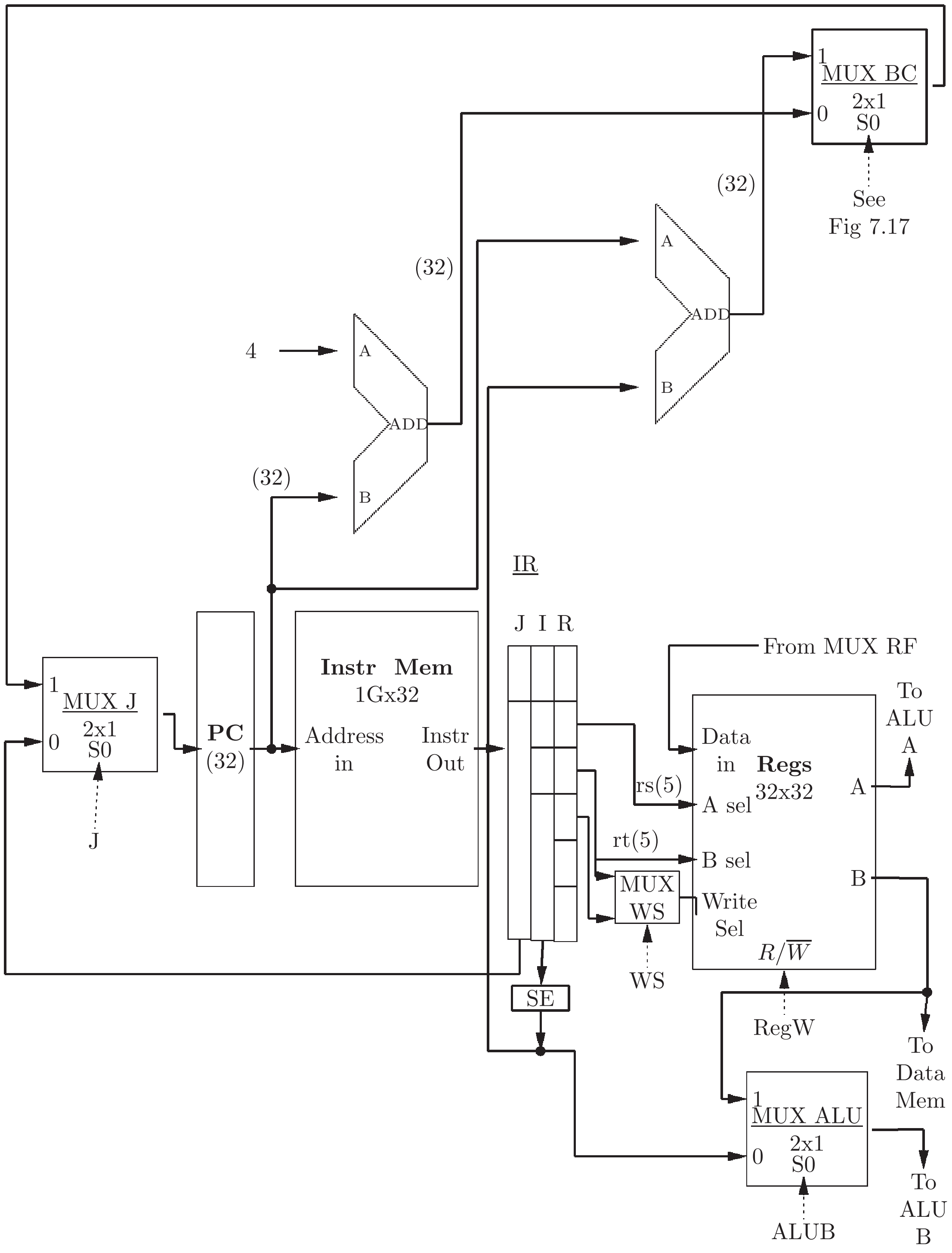


Figure 7.23: Datapath for the MIPS architecture. Signals from the control unit are shown with dashed arrows (see also Fig 7.24).

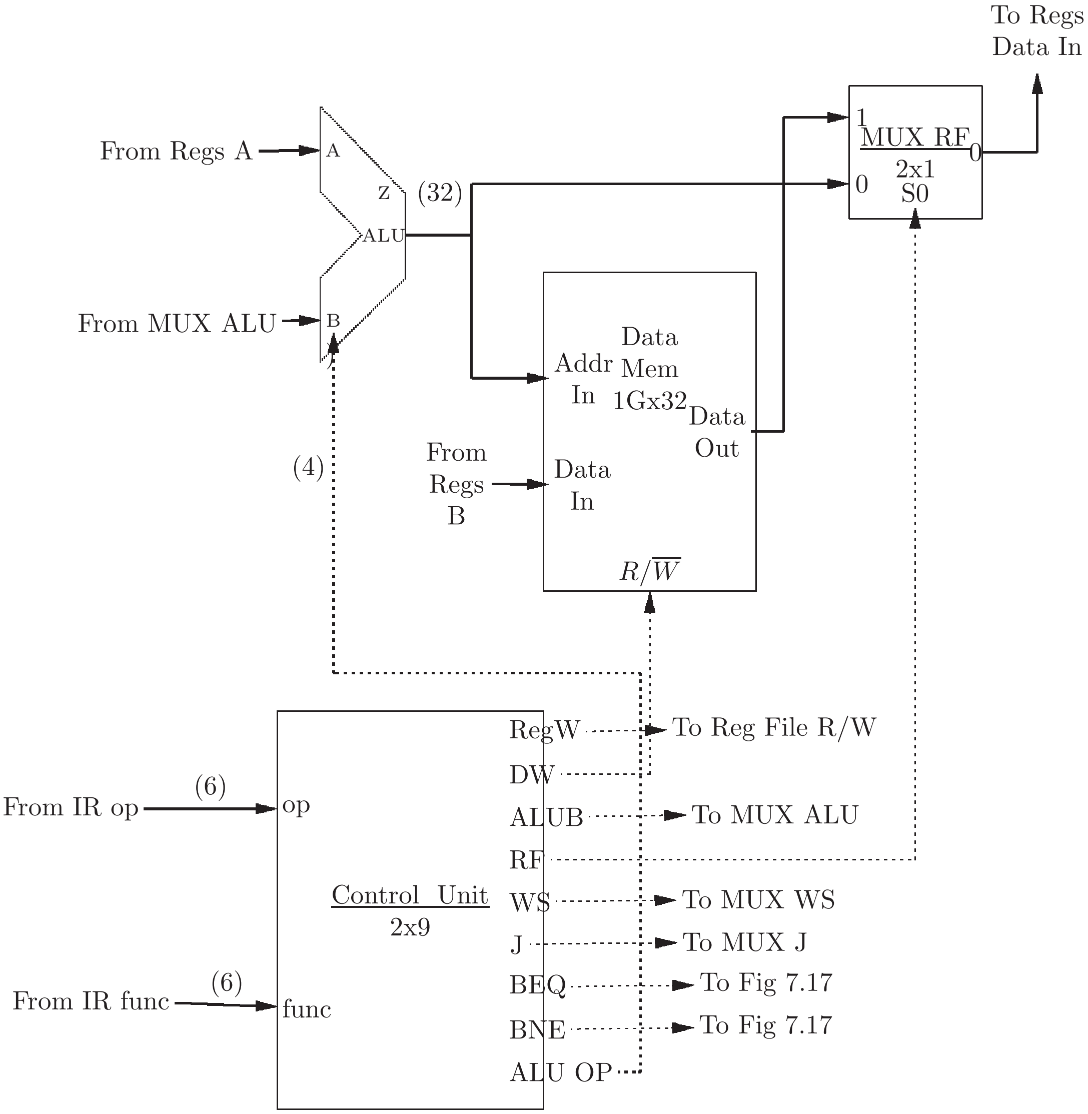


Figure 7.24: The full datapath (with Fig 7.23) showing the control unit. Control Unit signals are shown with dashed arrows.

4. What would be the output of the Control Unit if we were to include the set if less than (`slt`) instruction in our subset of MIPS instructions?
   (a) Show the value for RF in the row for `slt` in Fig 7.22.
   (b) What other changes would have to be made to the datapath?

5. (a) Show the boolean expression for the `DW` output of the Control Unit.
   (b) Show the boolean expression for the `ALUB` output of the Control Unit.
   (c) Show the boolean expression for the `RF` output of the Control Unit.
   (d) Show the boolean expression for the `WS` output of the Control Unit.
   (e) Show the boolean expression for the `J` output of the Control Unit.
   (f) Show the boolean expression for the `BEQ` output of the Control Unit.
   (g) Show the boolean expression for the `BNE` output of the Control Unit.
   (h) Show the boolean expression for the low order bit of the `ALU OP` output of the Control Unit.

Chapter 8

The Memory Hierarchy

8.1 Introduction to the Memory Hierarchy

As we have seen, the instructions and data comprising a MIPS program are stored in memory. In the MIPS architecture the instruction memory and data memory are separate components. The runtime performance of a MIPS program will depend, to a large extent, on the *speed* of the memory components (i.e. the time it takes to access the desired word).

The time it takes the CPU to execute one instruction (one clock cycle in a one-cycle implementation) will typically be much smaller than the time it takes to access memory. Memory access time is typically on the order of one thousand times the clock speed! With this simplistic model, one can imagine the CPU idle for 1000 cycles while a `lw` or `sw` instruction is waiting for a response from the memory.

There are several strategies that are employed to overcome memory latency (the delay introduced by a memory access). In the case of the instruction memory, most CPUs are designed to *prefetch* instructions. Instructions are loaded into processor registers long before they are needed, instead of loading them as needed. Thus rather than just one Instruction Register, there is an array of several Instruction Registers. Several strategies have been developed for the prefetching of instructions which follow a conditional branch instruction.

This chapter will focus on strategies used to improve latency for the data memory. The fast (but expensive) memory in the CPU is known as *cache memory*. We will investigate strategies that are used to ensure that frequently accessed words are kept in the cache as much as possible.

The principles of fast access to data memory will also apply to *virtual memory*. Virtual memory is the term normally used to describe an expansion of the RAM using a secondary storage device such as disk or flash memory. Thus the *memory hierarchy* consists of (in order from fastest and most expensive to cheapest and slowest) cache, RAM, and virtual memory.

	Memory Technology	Typical Usage	Volatile?
Fastest, Most expensive	FPGA	CPU Registers	Y
	SRAM	Cache	Y
	DRAM	RAM	Y
	Flash (removable)	Secondary Storage	N
	Solid State Disk (Flash)	Secondary Storage or Virtual Memory	N
	Fixed Magnetic Disk	Secondary Storage or Virtual Memory	N
	Optical Disk	Secondary Storage	N
Slowest, Cheapest	Removable Magnetic Disk	Secondary Storage	N

Figure 8.1: Table showing the performance parameters of various (primary) memory and (secondary) storage technologies

8.1.1 Memory Technologies

As mentioned above there are a number of different technologies that are used to implement data storage. In general, there is a trade-off between access time and cost for these technologies. Fig 8.1 shows some of the technologies available today.

A storage device is said to be *volatile*[1] if the stored data is lost when power is shut off. Permanent storage devices such as disks and flash memory are not volatile. RAM and cache memories are volatile.

8.1.2 Exercises

1. Given the following types of storage and memory:
   - Flash memory
   - Fixed Magnetic Disk
   - (Removable) Optical Disk
   - Static Random Access Memory (SRAM)
   - Magnetic Tape
   - Dynamic Random Access Memory (DRAM)
   - Removable Magnetic Disk

   (a) Order the above from fastest to slowest (access time).

   (b) Order the above from cheapest to most expensive (per byte).

   (c) Which of the above are volatile (require power to maintain data that has been stored)?

[1] We borrow this term from chemistry, in which a volatile fluid is one which evaporates quickly.

2. Define the term *memory latency*.

3. Rewrite the following MIPS program segment with an equivalent program segment that runs about 1000 times faster.

```
sub:
lp:
      ble   $t0,$0, done           # finished with loop?
      lw    $t1, incr              # increment
      add   $t2, $t2, $t1          # $t2 <- $t2 + incr
      addi  $t0, $t0, -1           # decrement loop counter
      j     lp                     # repeat
done:
```

8.2 Cache Memory

Cache memory is located in the memory hierarchy between the CPU registers and the main memory (RAM). Cache memory is faster (i.e. access time is lower) than RAM, but more expensive. Cache memory is slower than CPU registers but less expensive. The cache memory will maintain copies of data from RAM in units of fixed size known as *blocks*. A typical cache memory could consist of 512 ($= 2^9$) blocks, each of which stores 128 ($= 2^7$) bytes.
$2^9 \times 2^7 = 2^{16} = 64K$. Thus, in this example, the cache stores a total of 64K bytes.

When an instruction such as an `lw` instruction needs to access a word from the data memory (RAM), it will first attempt to find a copy of that word in the cache memory. If it succeeds, this is called a *cache hit*. If it fails, the cache memory will obtain a copy of the desired word from RAM; this is called a *cache miss*. Since cache memory is faster than RAM, a large proportion of cache hits will lead to improved performance.

When a `sw` instruction is executed, to store register contents into a word of the data memory, if a copy of that word is in cache, it will simply change that copy (cache hit). If it is not in cache, a copy is made from the appropriate block in RAM. At some point, the block containing the stored word must be copied back to RAM; this is handled in various ways, as we shall see below.

We now look at a few examples of cache implementation schemes: Direct-mapped cache and Associative cache.

8.2.1 Direct-mapped Cache

In a *Direct-mapped* cache each block of RAM is mapped to a predetermined block in cache memory. This mapping is done quickly and automatically simply by choosing the appropriate bits of a memory address. Since RAM is much larger than cache, this is a many-to-one mapping of blocks in main memory to cache blocks. Several blocks of RAM will map to the same cache block, which stores a copy of the RAM block. Thus, the cache memory will need to store information indicating the source of the block in RAM.

Each time a word (or byte) of RAM is accessed for a read operation, the CPU will first check to see if its block is already in the cache memory. If so it merely obtains its value from the cache, and there is no need to access the RAM. This is a cache hit. If the accessed word is not in cache, then its block is copied from RAM into the cache. This is a cache miss.

Each time a word (or bye) of RAM is accessed for a write operation, the CPU will check to see if its block is already in the cache memory. If so it writes the new value to the cache. But now the cache block and its corresponing RAM block are different; the difference can be reconciled in one of two ways:

- Every time a write operation occurs the cache block is copied back to the corresponding block in RAM, ensuring that the cache block agrees with its corresponding RAM block.
- The cache stores a *dirty bit* for each cache block. Initially the dirty bits are all 0. When a write operation occurs, the dirty bit for the selected cache block is set to 1. When a different RAM block is copied to cache (see above) the cache checks the dirty bit. If it 1, the cache block differs from its corresponding RAM block, so the cache block is copied to the corresponding RAM block. Then the new RAM block can be copied to the cache, and its dirty bit is cleared.

An example of a simple direct-mapped cache

The following simplified example is used to demonstrate a direct-mapped cache memory. We use a byte-addressable RAM consisting of 256 bytes; hence a RAM address will require 8 bits ($256 = 2^8$). We use a block size of 8 bytes per block. Thus there will be a total of 32 ($256/8 = 2^8/2^3 = 2^{8-3} = 2^5$) blocks in RAM. Our cache memory will store 8 blocks. Our RAM and cache are depicted in Fig 8.2 in which the cache is shown above the main memory (RAM). Each small box represents 1 byte, and a block of 8 bytes is shown as a vertical column.

Both diagrams in Fig 8.2 show the 3-bit addresses of a byte within a block as row labels. The cache diagram shows the 3-bit address of a block in the cache as column labels. The RAM diagram shows the 5-bit block addresses of selected blocks (1,3,9,11,17, and 19) as labels below the columns.[2]

If you are viewing this page in color, you will see that column 1 in the cache is yellow, as are columns 1, 9, and 17 in RAM. This shows that these blocks map directly to block 1 in the cache. In general, since there are 8 blocks in the cache, block number b in the RAM will map directly to block `b mod 8` in the cache. Also notice that block 3 in the cache is shown in blue, as are blocks 3, 11, and 19 in the RAM, showing that these blocks map directly to block 3 in the cache.

[2] Due to a lack of space, only the first 23 of 32 columns is shown for the RAM.

As mentioned above, the cache needs to write modified blocks back to RAM; at this point it will need to know which of the 4 possible RAM blocks it is storing. This information is shown as a 2-bit quantity labeled `main block` in the cache. It is also known as a *tag*.

In this example there was a reference to the byte at address $4b_x = 0100\,1011_2$. This byte is shown with a (red) circle in Fig 8.2. We can dissect that address as follows: $0100\ 1011_2 = 01\ 001\ 011_2$

```
01   block 1 of 4 in the RAM
001  block 1 of cache
011  byte 3 within the block.
```

In general, RAM address can be viewed as shown in Fig 8.3. The number of bytes in a block determines the size of the byte field. The number of blocks in the cache determines the size of the cache block field. The remaining bits in a RAM address determine the size of the tag field.

In our example, we are referencing the byte at address $4b_x$. The corresponding block has been copied from block 9 in RAM into block 1 of the cache. Also the cache stores the 2-bit main block number (or tag) 01 so that it knows which of the 4 possible RAM blocks it is currently storing.

We now provide an example to illustrate the behavior of the cache memory. Recall that when the RAM block being accessed is already in cache, it is called a cache hit, and when that block is not in the cache it is called a cache miss. In the table below we show a sequence of RAM addresses being accessed, and the effect that they have on the cache.

RAM Address	tag	cache block	Effect on cache
$9b_x = 1001\ 1011_2 = 10\ 011\ 011$	10	011	Cache Miss
$2d_x = 0010\ 1101_2 = 00\ 101\ 101$	00	101	Cache Miss
$ef_x = 1110\ 1111_2 = 11\ 101\ 111$	11	101	Cache Miss
$99_x = 1001\ 1001_2 = 10\ 011\ 001$	10	011	Cache Hit
$4a_x = 0100\ 1010_2 = 01\ 001\ 010$	01	001	Cache Miss
$ea_x = 1110\ 1010_2 = 11\ 101\ 010$	11	101	Cache Hit
$2d_x = 0010\ 1101_2 = 00\ 101\ 101$	00	101	Cache Miss

Note in the reference to $2d_x$ in the last line of the table that it is a repeat of the second line. However, the effect is a cache miss because cache block 101 has been clobbered by the reference to ef_x which is mapped to the same cache block.

8.2.2 Associative Cache

A more expensive, but potentially faster, implementation of cache memory is called *associative cache*. In an associative cache the blocks are grouped into *sets*, all of which have the same size. Each block of RAM maps to one of these sets. When a block needs to be copied from RAM into the cache, any one of the

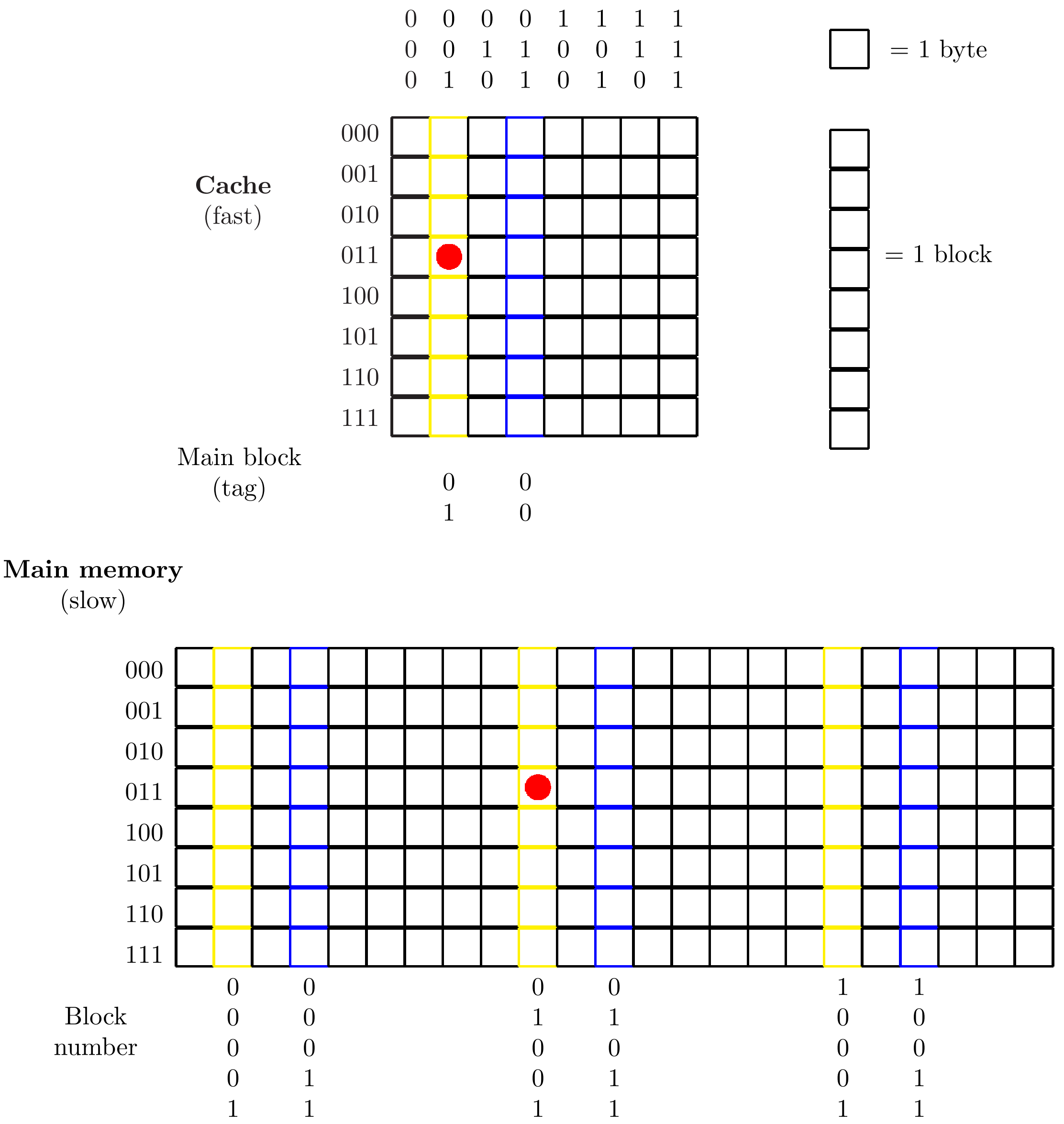


Figure 8.2: Diagram of direct-mapped cache memory and main memory (RAM). Byte at memory address $4b_x = 0100\ 1011_2 = 01\ 001\ 011_2$ is accessed.

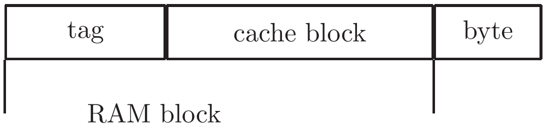


Figure 8.3: Fields of a RAM address for a direct-mapped cache memory

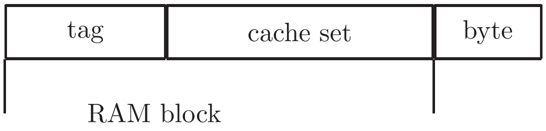


Figure 8.4: Fields of a RAM address for an associative cache memory

blocks in the cache set can be replaced by the new block. An associative cache memory in which there are n blocks in each set is called an *n-way* associative cache. A 1-way associative cache (n = 1) is the same as a direct-mapped cache.

The fields of a RAM address, with an associative cache, are similar to those shown in Fig 8.3. However, instead of a `cache block` field, we have a `set` field, as shown in Fig 8.4. Each RAM block maps to a set of blocks in the cache, rather than to an individual block.

To see why this scheme is potentially faster than a direct-mapped cache, consider the case where there are successive references to different RAM blocks, all of which map to the same RAM block. For example, referring to Fig 8.2, suppose there are references to the following memory locations in the sequence shown:
$88_x(= 1000\ 1000_2 = 10\ 001\ 000_2)$
$4b_x(= 0100\ 1011_2 = 01\ 001\ 011_2)$
$09_x(= 0000\ 1001_2 = 00\ 001\ 001_2)$
$ca_x(= 1100\ 1010_2 = 11\ 001\ 010_2)$
The bytes referenced by these addresses are all in blocks which map to the same cache block (001_2). Thus there will be a cache *miss* on each reference, which essentially nullifies the speed advantage provided by a cache memory.

If this had been a 4-way set associative cache, it could potentially store all 4 blocks in cache at the same time, thus converting 3 of the cache misses into cache hits.

An example of a simple 2-way associative cache

Our example of a 2-way associative cache memory is very similar to our example of a direct-mapped cache memory. The RAM is the same - 256 bytes, with 8 bytes in a block. The size of the cache memory is also the same, 8 8-byte blocks; however, in the associative cache we group every two blocks together, as shown in Fig 8.5. Thus there are now 4 sets in the cache. Note that the set number is only two bits, as shown in the column headers of the cache. This means that each block in RAM maps directly to one of the four sets in cache. It could be stored in either the left or right column of that set.

In Fig 8.5 there is a reference to the byte at RAM address $4b_x = 0100\ 1011_2 = 01\ 001\ 011_2$. The byte at this address is shown with a filled circle, and its block (number 01001) has been copied to one of the two blocks in cache set 01. Let's assume it is in the second column of this set, though it could as well be in the first column (see discussion of block replacement strategies, below). The tag for this block will be the three high order bits of the block number: 010.

If you are viewing Fig 8.5 in color, you will see that RAM blocks 1, 5, 9, 13, 17, 21, shown in yellow, all map to the same set, set 1, because these block numbers are all congruent to 1 (mod 4); if these numbers are expressed as 5-bit binary numbers, take the last two bits to get the set number. Similarly, RAM blocks 3, 7, 11, 15, 19, shown in blue, all map to the same set in the cache, set 3.

When a byte is referenced from RAM, if its block is already in cache, then the associative cache works exactly the same as the direct-mapped cache. However, if its block is not already in cache, the cache will need to copy the block from RAM. At this point the cache must decide which of the two blocks in the set is to be overwritten. This decision is made by a *block replacement algorithm.* For optimal performance, we would like it to overwrite the block which will not be referenced sooner as the program executes. While it is not possible to implement this optimal strategy, there are at least three different strategies which can be used to decide which block in a set is to be overwritten. We discuss one of these strategies here, without going into the details of the implementation, and mention the other two.

The first such block replacement strategy is called *Least Recently Used*, or LRU. In this strategy the cache will overwrite the block in the selected set which has been referenced least recently in the program's execution. This strategy will often increase the probability of cache hits in subsequent memory accesses.

We now take an example to illustrate the LRU block replacement strategy. We assume the state of the cache is as shown in Fig 8.6. in which a tag is shown for each cache block. The following table shows how the cache works as a sequence of bytes are read from the RAM.

RAM Address	tag	set	Effect on cache
$4a_x = 0100\ 1010_2 = 010\ 01\ 010$	010	01	Cache Hit
$18_x = 0001\ 1000_2 = 000\ 11\ 000$	000	11	Cache Hit
$ab_x = 1010\ 1101_2 = 101\ 01\ 101$	101	01	Cache Hit
$cb_x = 1100\ 1011_2 = 110\ 01\ 011$	110	01	Cache miss
			Block 11001 replaces block 01001

The final state of the cache is shown in Fig 8.7.

There are at least two other commonly used block replacement strategies for set-associative cache memories. The first is called *First-In First-Out*, or FIFO. In this strategy the block which has been residing in the cache set for the longest period of time is the one to be replaced.

Both LRU and FIFO can, in certain situations, degrade to the point that almost every memory reference causes a cache miss. This situation is known as *thrashing.*[3] The third block replacement strategy is called *Random.* A sequence of random numbers is used to determine the cache block that is to be replaced.

[3] The term 'thrashing' is more commonly used at the virtual memory level, but can also be used at the cache level, with the same meaning.

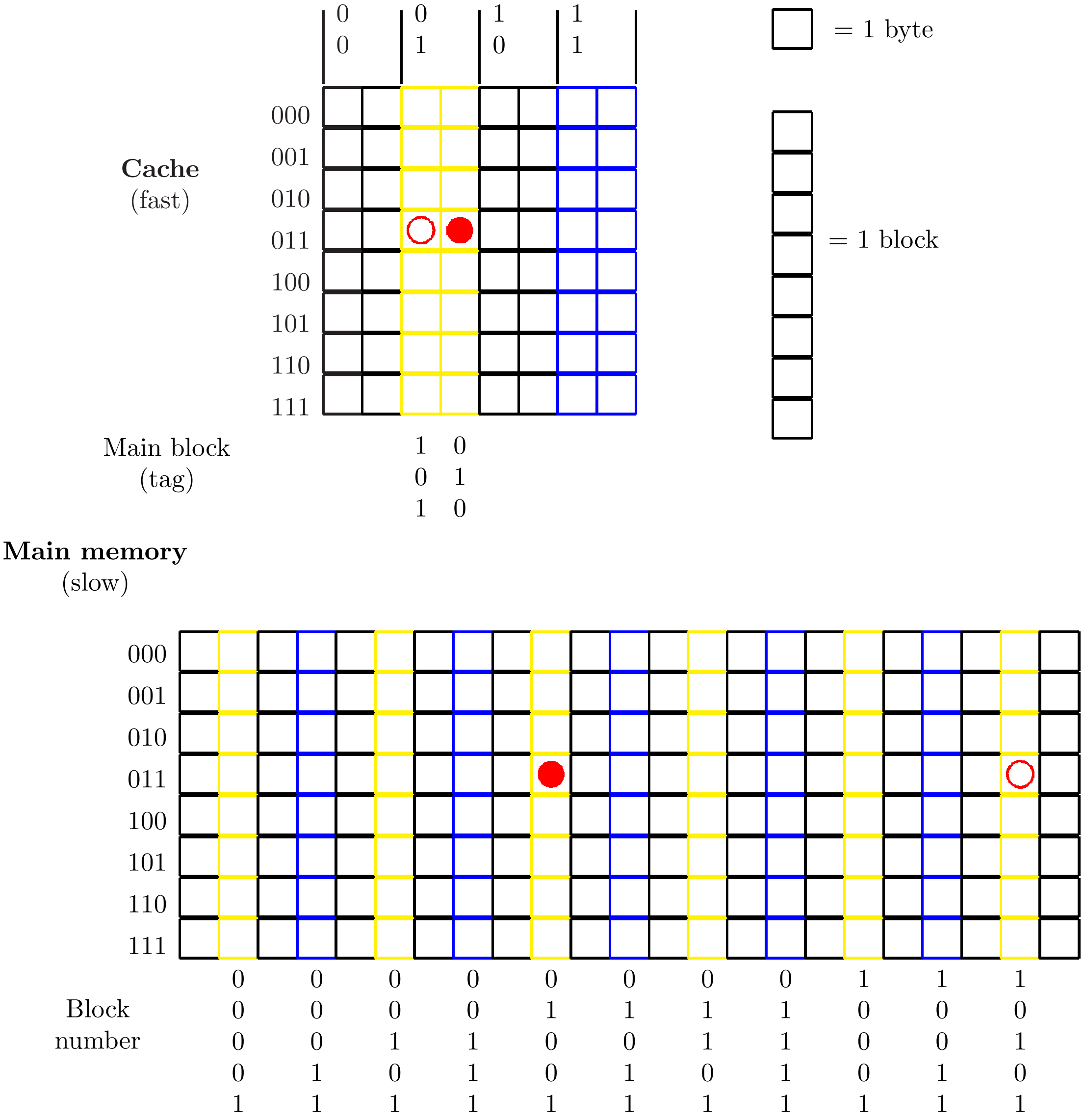


Figure 8.5: Diagram of a 2-way associative cache memory and main memory (RAM). Bytes at RAM addresses $4b_x = 0100\ 1011_2 = 01\ 001\ 011_2$ and $ab_x = 1010\ 1011_2 = 10\ 101\ 011_2$ are accessed, which map to blocks in the same cache set.

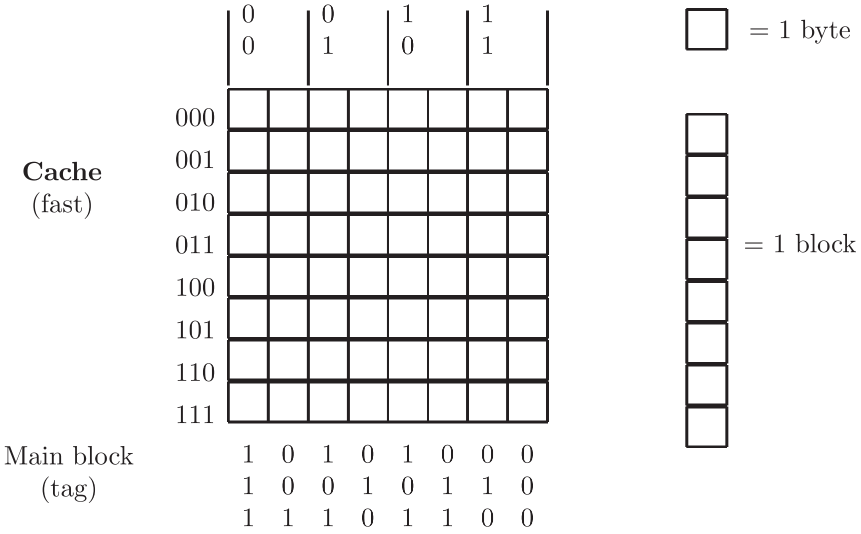


Figure 8.6: Initial state of the cache for an example of the LRU strategy

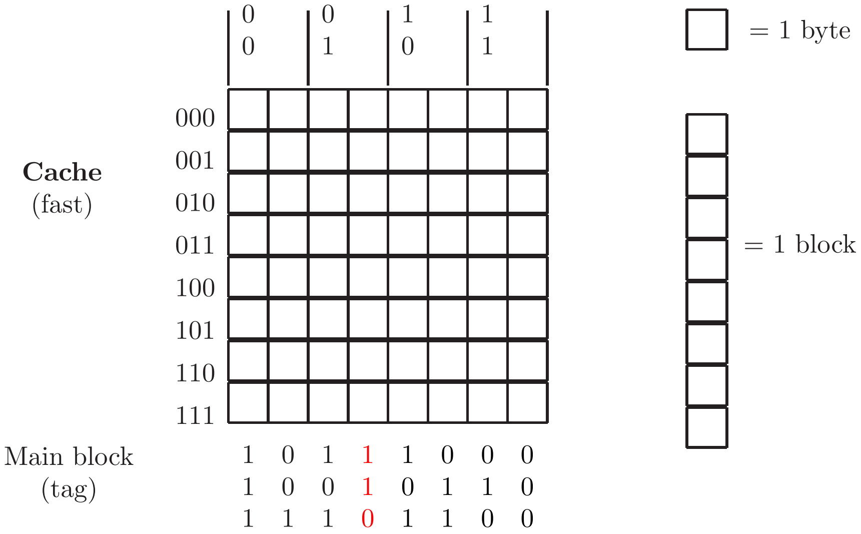


Figure 8.7: Final state of the cache for an example of the LRU strategy

If the sequence is long enough, and sufficiently random, this will effectively prevent thrashing. Of course none of these strategies will be optimal.

8.2.3 Exercises

1. You are given a byte-addressable RAM with 4K bytes and a direct-mapped cache memory with 512 bytes. The block size is 32 bytes.
   (Hint: Work with exponents of 2)

   (a) How many blocks are in the RAM?

   (b) How many blocks are in the cache memory?

   (c) If a program access the byte at location $9c7_x$, which block of RAM is copied to the cache? Give the block number as shown in Fig 8.2.

   (d) To which block of the cache is it copied? Show the column heading as shown in Fig 8.2.

   (e) What will be the tag value on that cache block?

2. (a) Complete the following table for the cache and RAM shown in Fig 8.2. Assume the memories are initially clear, so the first reference to any block is a cache miss.

RAM Address	tag	cache block	Effect on cache
$9b_x$			
dd_x			
$9d_x$			
$8e_x$			
$9f_x$			
08_x			
$f9_x$			

   (b) Show the tag value for each cache block which has been affected by the memory refences in the above table.

   (c) Show 3 more memory references, all to different blocks in the RAM, and all of which result in cache hits.

3. (a) Show a diagram similar to Fig 8.2 for a 16K RAM, and a 32 byte direct-mapped cache with 4-byte blocks. Your diagram should show at least the first 22 blocks of the RAM, and the entire cache.

   (b) Put a circle on the byte at RAM address 0057_x.

   (c) Put another circle on the cache byte to which the RAM address 0057_x is mapped.

   (d) Show the tag value (in hex) in the cache for the block containing the circled byte.

4. Given a RAM storing 256M bytes and a 4-way associative cache memory storing 64K bytes, with a block size of 128 bytes:
   Hint: Use exponents of 2.

(a) How many blocks are in the RAM?

(b) How many blocks are in the cache?

(c) How many sets of blocks are in the cache?

(d) Show a diagram of a RAM address, similar to Fig 8.4. In the diagram show the width, in bits, of each field.

5. Refer to the 2-way associative cache of Fig 8.5 and the table of RAM memory references shown below.

RAM Address	tag	set	Effect on cache
$f2_x$			
18_x			
$a5_x$			
db_x			
eb_x			
$d2_x$			
$c0_x$			
$d7_x$			
$1d_x$			

(a) Complete the table, showing which references cause cache hits/misses in the column for `Effect on cache`. Assume that an initial reference to a RAM block always causes a cache miss. Assume an LRU block replacement strategy is used.

(b) Show the final tag field(s) for each cache set which has been altered.

6. Given a 32K RAM with a 5-way cache storing 40 bytes, in which the block size is 4 bytes.

(a) How many blocks are in the RAM?

(b) How many sets are in the cache memory?

(c) Show a diagram of the RAM and cache memory similar to Fig 8.5. Show at least 20 columns in the RAM.

(d) Draw a circle in the RAM for the byte at address 0026_x

(e) Draw a circle in the cache memory for the byte to which RAM address 0026_x is mapped.

(f) Show the fields in a RAM address as in Fig 8.4, and show the number of bits in each field.

8.3 Virtual Memory

The next level of the memory hierarchy is called *virtual memory*. Whereas the purpose of cache memory is to provide faster access to the RAM, the purpose of virtual memory is to expand the capacity of RAM. A secondary storage device, such as disk or flash memory, is used to provide an extended address space for the RAM. For example, if we have a 64K RAM, it can be addressed with a 16-bit address. However, it can be expanded, using 1M byte of disk space, to achieve a virtual RAM of 1M. This means that 1M byte of disk space would be reserved for virtual memory, and cannot be used for other purposes. An address for the virtual RAM would be 20 bits.

Just as the bytes in a cache memory are grouped into blocks, the bytes in a virtual memory are grouped into *pages*.[4] A page is typically about 8K bytes.

Unlike the cache memory level, virtual memory pages are generally not directly mapped to pages in the RAM. Virtual memory is more like an associative cache memory in which there is just one set.

When the processor needs data from the RAM, if the addressed byte is in a page which is currently residing in RAM, the secondary storage is not accessed at all. However, if the addressed bye is not in the RAM, the page which contains it must be copied into the RAM; this is called a *page fault*. A RAM page is selected to be copied back to virtual memory so that it can be overwritten with the desired page from disk. A page-replacement policy (similar to the block-placement strategies used by associative cache memories) is used to determine the page. Examples of page-replacement policies are Least Recently Used (LRU) and First-In First-Out (FIFO).

When the reference to a RAM byte is a write operation, the containing page would be copied back to the disk swap space. Alternatively, a 'dirty bit' could be used to determine whether a RAM page has been written. If the page has not been altered, there is no need to write it back to the disk.

Fig 8.8 is a diagram of a virtual memory in which there are only 32 pages of RAM, but 256 pages of swap space on the disk. The total address space of virtual memory would be 256 pages. If your are viewing Fig 8.8 in color, you will notice some random collection of pages in virtual memory are shown in yellow. These are the pages which have a copy in RAM.

The implementation details of virtual memory are described in textbooks on Operating Systems. Here we provide a brief description of the implementation of virtual memory. In Fig 8.9 we show a diagram for a small virtual memory system in which the RAM stores 4 pages and the Disk Swap Space stores 16 pages. Note that 4 different pages from the Disk Swap Space have been copied into RAM, as shown by the arrows.

The system must maintain a table storing the high order bits of a virtual memory address for each page in RAM, indicating where it is to be copied back when it is replaced in RAM by another page. This table is called a *page table*. This table is shown in Fig 8.10. It shows that page 2 in RAM, for example, is a copy of page 12 in virtual memory. When it is to be replaced, the page table is consulted so the page can be copied back to the correct page in the Disk Swap Space.

[4] Since cache memories were generally developed by hardware designers, and virtual memory was developed by operating systems people, they often use different terminology for the same concepts.

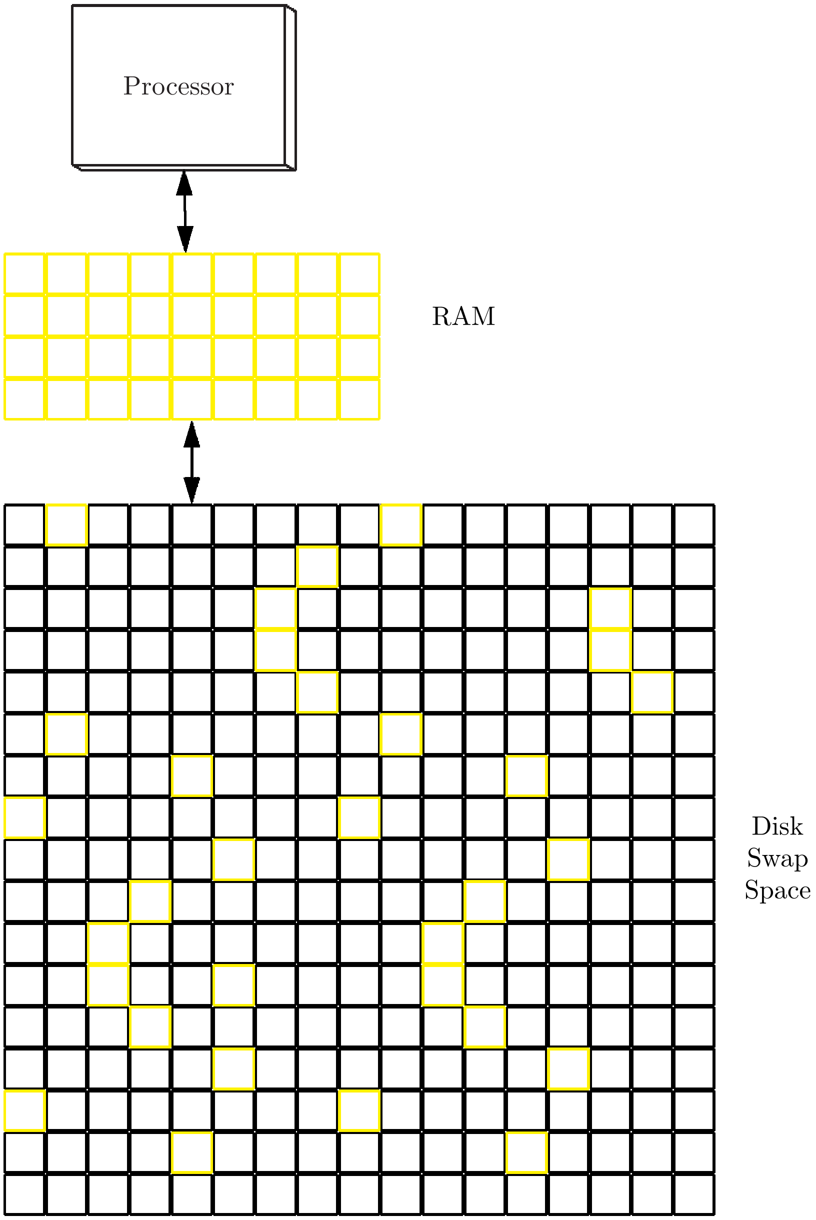


Figure 8.8: Diagram of a virtual memory system. Each small block represents one page.

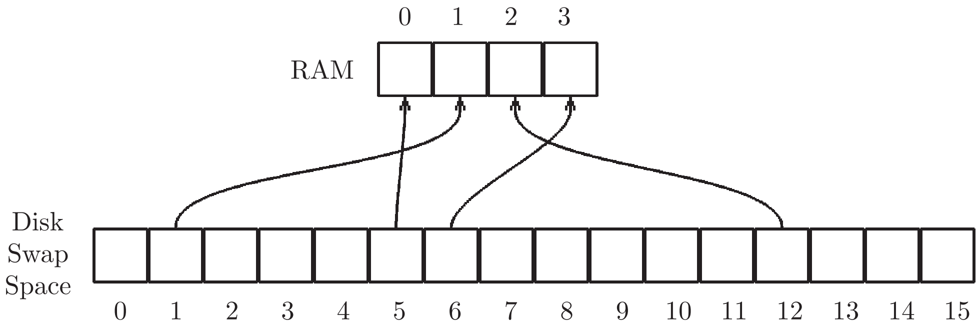


Figure 8.9: Diagram of a small virtual memory system; the page table is shown in Fig 8.10.

In our example we have a virtual memory system storing 16 pages, and only 4 pages in RAM. If we assume that each page is 8K bytes ($8K = 2^{13}$), a virtual memory address would be 17 bits: 4 bits for the page number and 13 bits for the offset within a page. For a virtual memory address of $18158_x = 1\ 1000\ 0001\ 0101\ 1000_2 = 1100\ 0\ 0001\ 0101\ 1000_2$ the page number is $12 = 1100_2$ and the byte offset within the page is $0158_x = 0\ 0001\ 0101\ 1000_2$.

As with associative cache memories, a virtual memory system will need a page replacement policy. This algorithm decides which page is replaced when a page fault occurs. Virtual memory systems typically use a Least Recently Used (LRU) algorithm, though First-In First-Out (FIFO) and random algorithms are also used (see the section on cache memories).

Using the page table of Fig 8.10 and assuming the RAM pages were loaded in order: 0, 1, 2, 3, we show what results from a reference to virtual memory address $074fc_x = 0\ 0111\ 0100\ 1111\ 1100_2 = 0011\ 10100111111100_2$. We see that it is a reference to virtual page $0011_2 = 3$, which is not currently in RAM. This is a page fault. If we are using an LRU page replacement algorithm, we will replace the page at RAM page 0. It is copied back to the disk, and replaced with page 3 from the disk swap space.

We emphasize that the cost of virtual memory is an increased latency in memory access. A *page fault* is the term used when the system must copy a page from the disk swap space into RAM. Every time there is a page fault there is a significant delay, because access time to the disk is on the order of one thousand times the access time to RAM. If an executing program causes many page faults, the system will spend more time swapping pages of memory than it will executing the user's program. This situation is called *thrashing*.

8.3.1 Exercises

1. Given a virtual memory system storing 4G bytes, a page size of 64K bytes, and a RAM storing 16M bytes.

Page Table

Disk	RAM
$5 = 0101$	0
$1 = 0001$	1
$12 = 1100$	2
$6 = 0110$	3

Figure 8.10: Page table for the small virtual memory of Fig 8.9

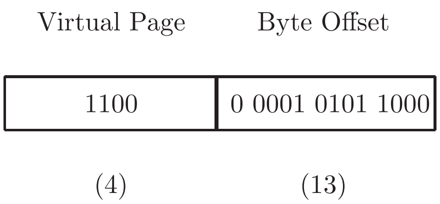


Figure 8.11: Diagram of a virtual address for a system with 16 pages in virtual memory, and 8K bytes in each page. Address is 18158_x.

Hint: Work with powers of 2.

(a) How many pages are in the RAM?

(b) How many pages are in the virtual memory?

(c) What is the page number for the virtual memory address $402a0100_x$?

(d) Show a diagram of the virtual memory address $402a0100_x$ showing the width, in bits, of each field. (see Fig 8.11).

2. Refer to Fig 8.9. The table below shows a sequence of virtual memory references. Complete the table showing the virtual memory page number, whether or not page fault has occurred, and the RAM page which is referenced.

VM Address	VM page number	fault?	RAM page
$042b8_x$		Y	0
$180cc_x$		Y	1
$042b8_x$		Y	2
$042b8_x$		Y	3
18240_x			
$074c4_x$			
$0afd0_x$			
05000_x			
$060b3_x$			

(a) Assume an LRU page replacement algorithm is used. Show the page table when completed.

(b) Assume a FIFO page replacement algorithm is used. Show the page table when completed.

8.4 Locality

As we have seen in the preceding sections, cache memory is capable of improving run-time efficiency, and virtual memory is capable of expanding the memory's capacity without degrading performance. However, both of these improvements are subject to a condition known as *locality*. If there are many references to locations in a small number of different cache blocks (or virtual memory pages), then there will be few cache misses (or page faults). In this case we say that the executing program exhibits good locality. If there are references to memory locations in many widely scattered cache blocks (or in many different virtual memory pages), performance is degraded, and thrashing may occur. In this case we say that the executing program exhibits poor locality.

This locality principle applies to both the cache memory level and the virtual memory level in the memory hierarchy. Hence we will use the phrase *memory unit* to mean either cache block or virtual memory page. In place of the phrases 'block miss' or 'page fault', we use the phrase *memory fault*.

In what follows we distinguish between the various kinds of locality on different dimensions: Data vs. Instruction, and Temporal vs. Spatial.

8.4.1 Data Locality Versus Instruction Locality

Data locality has to do with memory references for data. In the MIPS architecture, the `load word` (`lw`) and `store word` (`sw`) instructions refer to the Data Memory. When successive `lw` and `sw` refer to locations in different memory units (i.e. not near each other in memory), we can expect memory faults, and the program exhibits poor data locality.

Instruction locality is affected by transfer of control (i.e. branch or jump instructions). When a program executes sequentially, it will exhibit good instruction locality because successive references to the instruction memory will generally be to the same memory units. When a program executes branch or jump instructions to various, widely separated, instructions, the program may encounter many memory faults when fetching an instruction to the Instruction Register. This situation is not common, as most loops (even nested loops) tend to remain fairly local, remaining on just a few memory units. It is conceivable, however that a *switch* or *branch table* in a loop could cause many memory faults, resulting in poor instruction locality.

In the MIPS architecture the Data Memory and Instruction Memory are separate. However even on architectures in which instructions and data are co-located on the same chip, we still make the distinction between data locality and instruction locality.

As an example of a program segment which contrasts good versus poor data locality, we run the Java program shown in Fig 8.12. This program runs a simple loop, accessing successive memory locations in an array of ints. It then enters another loop in which it accesses random locations in the same array. When running this program on an iMac (OS X 10.11.3) the first loop executes in about 1 second, whereas the second loop, which exhibits poor locality, executes in about 6 seconds for the same number of iterations![5] Without going into the details of the Java run time environment, the IDE used to compile and run the program, or the operating system (MacOS X), it is hard to say whether the performance difference between these two loops occurs at the cache level or at the virtual memory level. It is clear, however, that the performance is affected by data locality.

This is one of the most important lessons for software developers; locality can have a huge impact on running time irrespective of a theoretic run time analysis or how fast the CPU may be.

8.4.2 Temporal Locality Versus Spatial Locality

Consider a memory with several memory units (i.e. a cache with several blocks, or a virtual memory with several pages). It is possible for a program to execute efficiently even if repeated accesses to memory are to different memory units. For example, the sequence of accesses to the blocks in a large RAM with cache memory storing 16 blocks might be:
block 12
block 3
block 106
block 33
block 7

If this sequence of memory references occurs in a loop, there will be no cache misses because only 5 different blocks are being referenced, and the cache stores 16 blocks. This would be an example of good *temporal* locality, but poor *spatial* locality. Temporal locality is good when memory units accessed are accessed subsequently soon thereafter. In this example the program would exhibit good locality despite the poor (spatial) data locality.

8.4.3 Exercises

1. Which of the following loops exhibits good spatial locality, and which exhibits good temporal locality?

   (a)
   ```
   int MAX = 10000;
   for (int i=0; i<MAX; i++)
       sum = sum + nums[i];
   ```

[5] In this program we ensure that the calls to the `nextInt` method in the `Random` class do not skew the timing results in favor of the first loop.

```
import java.util.Random;

public class Locality
{
    public static void main()
    {   final int MAX = 100000000;
        int sum = 0;
        int [] nums = new int[MAX];
        Random rand = new Random();

        System.out.println ("start sequential");

        // Good data locality
        for (int i=0; i<MAX; i++)
            sum = sum + nums[i] + rand.nextInt(MAX);

       System.out.println ("start random");

       // Poor data locality
       for (int i=0; i<MAX; i++)
            sum = sum + nums[rand.nextInt(MAX)];

       System.out.println ("done");
    }
}
```

Figure 8.12: Java program to contrast good and poor data locality

```
public class MatrixLocality
{
    public static void main()
    {   final int MAX = 10000;
        int sum = 0;
        int [][] nums = new int[MAX][MAX];

        System.out.println ("start row-major order");

        for (int row=0; row<MAX; row++)
            for (int col=0; col<MAX; col++)
                nums[row][col] = row + col;

       System.out.println ("start col-major order");

       for (int col=0; col<MAX; col++)
           for (int row=0; row<MAX; row++)
                nums[row][col] = row + col;

       System.out.println ("done");
    }
}
```

Figure 8.13: Java program to explore running time for a matrix of ints

(b)

```
int MAX = 10000;
int ctr = 0;
for (int i=0; i<MAX; i++)
  { ctr = (ctr+100) % MAX;
    sum = sum + nums[ctr];
  }
```

2. Show an example (or template) of a Java program with poor instruction locality.

3. In Java, as with most programming languages, the elements of an array are stored in row-major order. That means tbe elements are mapped to the one-dimensional memory by rows, first all the elements in row 0, then all the elements in row 1, then all the elements in row 2, etc. Consider the program shown in Fig 8.13. The main method simply uses a nested loop to store a value in each position of a two dimensional array. It does this twice, once in row-major order, and once in column-major order.

   (a) Which loop will execute faster, or do they run in the same time?

   (b) Run this program on a computer to verify your response to part (a).

   (c) Explain why one of the loop executes much faster than the other, or explain why they execute in the same amount of time.

4. Which of the following sorting algorithms would you assume exhibit good data locality, and which would exhibit poor data locality?

   - Selection Sort
   - Bubble Sort
   - Quick Sort
   - Merge Sort

Chapter 9

Alternative Architectures

Since the early years of computing, many different designs have been promoted for the central processing unit of a computer. However, there are some things which are relatively stable and commonplace. In most computer architectures today there is a *stored program* design in which a sequence of instructions is stored in memory. This design was first proposed by the Princeton mathematician John Von Neumann in 1945. It has come to be known as the Von Neuman Architecture and is defined by:

- A processing unit which contains an ALU and registers
- A control unit which contains an instruction register and program counter
- Memory that stores data and instructions[1]
- External storage (peripheral devices)
- Input and output mechanisms

Virtually every modern processor is based on the Von Neumann architecture. In this chapter we briefly examine some architectural classes, followed by two specific examples of the Von Neumann architecture: The ARM and Intel Pentium processors.

9.1 Instruction Set Architectures

When designing a CPU, one must choose from a variety of design decisions on the format of an instruction. This decision is known as an *Instruction Set Architecture* (ISA).

[1] The data and instructions can be stored in separate memories, as with the MIPS architecture; or they may be in the same memory, in which case a program is capable of modifying itself.

9.1.1 Zero-address Architecture

A zero address architecture makes use of a hardware stack, typically in (data) memory. The operands of an operation are always the top two values on the stack. The result of the operation is pushed onto the stack. The Burroughs B-Machine processors (circa 1970) utilized this kind of architecture.

The instruction set would need push and pop instructions, to reference memory.

```
   Push   x       // load a full word from memory location
 //    x, and push it onto the stack
   Pop    x       // Remove the full word from the top of
 //   the stack and store it in memory
 //   location x.
```

All arithmetic instructions can produe a result without any operands; they always operate on the top two values on the stack (popping them from the stack) and push the result of the operation onto the stack. For example, to compute `a-(b+c)`:

```
Push   a
Push   b
Push   c
Add              // Pop  c,  Pop b, Push b+c
Sub              // Pop b+c, Pop a, Push a-(b+c)
```

Alternatively, to compute `(a-b)+c`:

```
Push   a
Push   b
Sub              // Pop b, Pop a, Push a-b
Push   c
Add              // Pop c,  Pop a-b, Push (a-b)+c
```

Note that the value on top of the stack is the *right* operand of the operation.

9.1.2 One-address Architecture

In a one-address architecture each instruction has one memory address. There is also a CPU register, known as an *accumulator*, which serves as the left operand *and* the result of an operation. The DEC PDP-8 minicomputer (circa 1970) was an example of a machine with a one-address architecture.

For example an Add instruction would add the referenced word of memory to the current value in the accumulator and store the result in the accumulator:

```
Add   x       // Acc = Acc + x
```

To load a memory value in the accumulator, the accumulator could be cleared before doing an Add:

```
Clr           // Acc = 0
Add   x       // Acc = x
```

Some one-address machines would also have a *negate* operation, Neg, to form the two's complement of the value in the accumulator. This can be used for subtraction:
`a-b = a + (-b)`

To compute the value of the expression `a-(b+c)`:

```
Clr           // Acc = 0
Add   b       // Load b into Acc
Add   c       // Acc = b+c
Neg           // Acc = -(b+c)
Add   a       // Acc = a-(b+c)
```

To compute the value of the expression `(a-b)+c`:

```
Clr           // Acc = 0
Add   b       // Load b into Acc
Neg           // Acc = -b
Add   a       // Acc = a-b
Add   c       // Acc = (a-b)+c
```

9.1.3 Two-address Architecture

In a two-address architecture each instruction has two operands. These machines typically had a group of CPU registers which could store operands and results of operations. The first operand of the instruction would be a register which stores both the left operand of the operation *and* the the result of the operation. The IBM 360/370 series of mainframes (circa 1960-1980) and the Intel microprocessors are examples of two-address architectures.

For example, an instruction on a two-address architecture could be:

```
Add   r1, r2      // r1 = r1+r2
```

This machine would also have Load and Store instructions which reference memory locations:

```
Load   r1, x     // r1 = memory[x]
Store  r3, y     // memory[y] = r3
```

To compute the value of the expression `a-(b+c)`: [2]

```
Load   r1, b     // r1 = b
Add    r1, c     // r1 = b+c
Load   r2, a     // r2 = a
Sub    r2, r1    // r2 = a-(b+c)
```

[2] Note that here the second operand could be either a memory location or a register.

To compute the value of the expression `(a-b)+c`:

```
Load   r1, a     // r1 = a
Sub    r1, b     // r1 = a-b
Add    r1, c     // r2 = (a-b)+c
```

9.1.4 Three-address Architecture

In a three-address architecture the result, left operand, and right operand of the operation are all separate operands of the instruction. The MIPS machine described in chapters 3 and 4 is an example of a three-address architecture. The ARM machine described later in this chapter is also an example of a three-address architecture.

As with the two-address architecture there are several CPU registers which store operand values and results of operations. For example, to compute the value of the expression `a-(b+c)`:

```
Load   r1, a
Load   r2, b
Load   r3, c
Add    r3, r2, r3        // r3 = b+c
Sub    r3, r1, r3        // r3 = a-(b+c)
```

For example, to compute the value of the expression `(a-b)+c`:

```
Load   r1, a
Load   r2, b
Load   r3, c
Sub    r2, r1, r2        // r2 = a-b
Add    r3, r2, r3        // r3 = (a-b)+c
```

9.1.5 Exercises

1. Show how to evaluate the expression x = (a+b+c)-(d-f), where each variable represents a memory location, using
   (a) A zero-address architecture
   (b) A one-address architecture (assume a Store instruction can store the accumulater in a given memory location)
   (c) A two-address architecture (assume registers r1 and r2 are available)
   (d) A three-address architecture (assume registers r1, r2, and r3 are available)

9.2 Addressing Modes

For those instructions which access memory (such as Load and Store instructions), there are several ways in which a memory address can be specified.

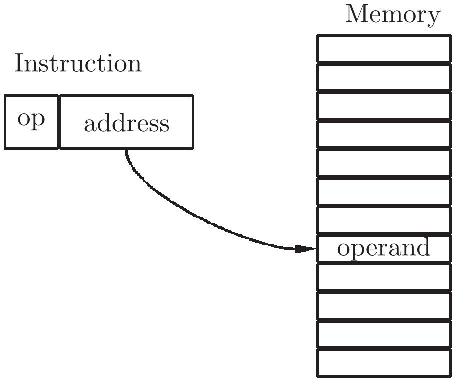


Figure 9.1: Diagram of the direct addressing mode. The instruction stores an absolute memory address

9.2.1 Direct Addressing

When the absolute address of a memory operand is specified in an instruction, this is called a *direct address mode*. In cases where the address field is not sufficiently long to address all of memory, the address is padded to form a complete memory address.

An example of direct address mode would be the unconditional jump instruction (j) in the MIPS architecture. A statement such as

```
j   exit
```

is translated by the assembler to a J format instruction in which the memory address of the jump target is a 26-bit word address. This 26-bit address is padded on the right (low-order) end with two 0 bits, to form a byte address; it is also padded on the left (high-order) end with 4 more 0 bits to form a full 32-bit address. A diagram of the direct addressing mode is shown in Fig 9.1 in which the absolute memory address of the operand is contained in the `address` field of the instruction.

9.2.2 Indirect Addressing

When the memory operand of an instruction contains the address of a memory word and that memory word contains the address of the operand, we have what is known as *indirect* addressing. Indirect addressing was often used in second generation computers (mid 1950 - mid 1960's). A diagram of the direct addressing mode is shown in Fig 9.2. In this diagram the address of the fourth word of memory is in the `address` field of the instruction. The fourth word of memory contains the address of the ninth word of memory which is used as the operand for the instruction.

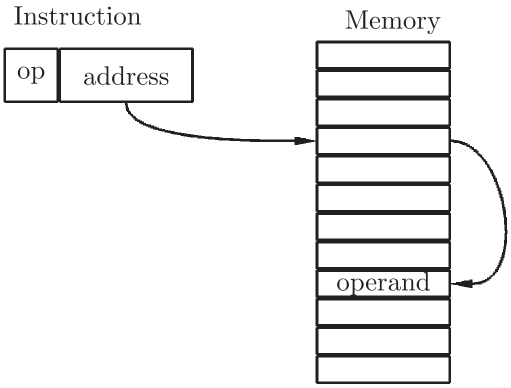


Figure 9.2: Diagram of the indirect addressing mode. The instruction stores the address of the memory word which contains the address of the operand

9.2.3 Base Register and Displacement Addressing

In the early 1960's the use of CPU general registers for memory addressing was introduced. In this addressing mode, the contents of a general register is added to an immediate displacement to form the effective address. For example, if the base register field in the instruction is r4, register r4 contains 0x40203004, and the displacement field in the instruction is 008, then the effective memory address would be `0x40203004 + 008 = 0x4020300c`. This form of memory address, known as *base-displacement* addressing, allowed for *relocatable code*. A segment of code could address instructions and data *relative* to the address in the base register.[3] A diagram of the base-displacement mode is shown in Fig 9.3. In this diagram the base register is register 3 (the fourth register) and the displacement, in the `disp` field of the instruction, is 5 memory words. The effective address of the operand is thus 5 words after the address in register 3.

9.2.4 Base Register, Index Register, and Displacement Addressing

In the mid 1960's, with the advent of the IBM 360 mainframe, an *index register* was also included in a memory address. In this addressing mode the contents of both the base and index registers are added to the displacement to form the effective memory address:

`effective address = (base reg) + (index reg) + displacement`

For example, if register r3 contains 0x40203004, register r7 contains 0x00000032, and the displacement field is 008, then the effective address would be:

`0x40203004 + 0x00000032 + 008 = 0x4020303e.`

As in base-displacement addressing, the base register allows code to be relocat-

[3] Relocatable code was essential in the implementation of multiprogrammed operating systems and virtual memory systems.

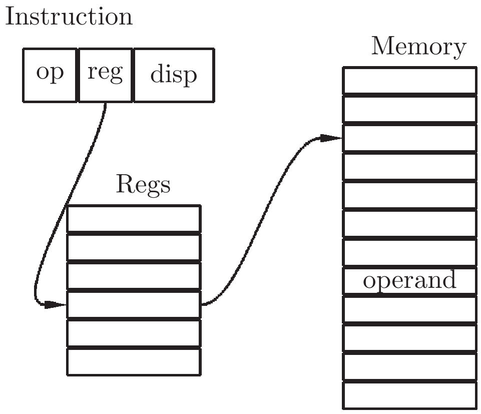


Figure 9.3: Diagram of the Base-Displacement addressing mode. The effective address is the sum of the base register plus a displacement. In this example, the base register is register 3, and the diplacement is 5 memory words.

able. Index registers are usually used to step through the elements of an array, by starting with 0 in the index register, and incrementing the index register (by the size of an array element) each time the next array element is needed. A diagram of the base-displacement mode is shown in Fig 9.4. In this diagram the instruction contains fields for a base register, an index register, and a displacement. The base register is register 4. The index register is register 1, which contains 3, and the displacement is 4 memory words. The effective address of the operand is thus 3 + 4 = 7 words beyond the address in register 4.

9.2.5 Exercises

1. Show how the six elements of a list of 32-bit contiguous numbers, named A, can be added, using instructions with each of the following addressing modes. Assume that we are using a two-address architecture with the following instructions:

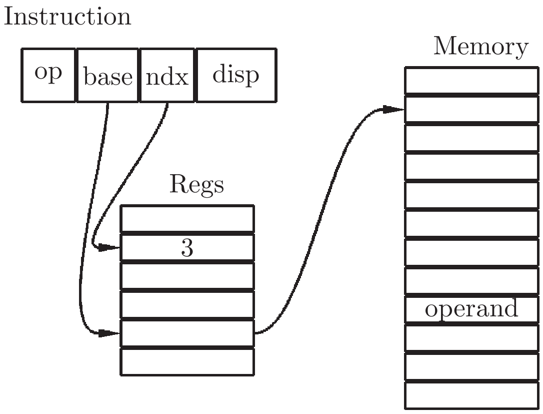


Figure 9.4: Diagram of the Base-Index-Displacement addressing mode. The effective address is the sum of the base register, the index register, and a displacement. In this example, the base register is register 4; the index register is register 1, which contains 3, and the diplacement is 4 memory words.

Instruction	Meaning
add rs,rt	reg[rs] = reg[rs] + reg[rt]
sub rs,rt	reg[rs] = reg[rs] - reg[rt]
lod rs,addr	reg[rs] = memory[addr]
sto rs,addr	memory[addr] = reg[rs]
add rs,addr	reg[rs] = reg[rs] + memory[addr]
sub rs,addr	reg[rs] = reg[rs] - memory[addr]
beq rs,rt,label	branch to label if reg[rs]==reg[rt]
blt rs,rt,label	branch to label if reg[rs]<reg[rt]
bgt rs,rt,label	branch to label if reg[rs]>reg[rt]
ble rs,rt,label	branch to label if reg[rs]≤reg[rt]
bge rs,rt,label	branch to label if reg[rs]≥reg[rt]
bne rs,rt,label	branch to label if reg[rs]≠reg[rt]

(a) Direct addressing. Assume the values in the array have labels A0, A1, A2, A3, A4, A5. Assume there is an add instruction with two operands; the first operand is a register and the second operand is an absolute memory address.

```
add reg, address
```

will add the contents of the register to the memory word at the specified address, and store the sum back into the register.

(b) Indirect addressing. Assume the addresses of the six numbers are in contiguous memory locations named A0,A1,A2,A3,A4,A5. Add the following instructions to the instruction set described above:

00000100_{16}	00 00 01 03	00 00 01 08	00 00 01 00	00 00 01 05
00000110_{16}	00 00 01 07	00 00 01 09	00 00 01 01	00 00 01 0c

Figure 9.5: Initial memory contents for exercise on addressing modes

Instruction	Meaning
lodI rs,addr	reg[rs] = memory[memory[addr]]
stoI rs,addr	memory[memory[addr]] = reg[rs]
addI rs,addr	reg[rs] = reg[rs] + memory[memory[addr]]
subI rs,addr	reg[rs] = reg[rs] - memory[memory[addr]]

(c) Base-Displacement addressing. Add the following instructions to the instruction set described above:

Instruction	Meaning
lod rs,(rt)disp	reg[rs] = memory[reg[rt]+disp]
sto rs,(rt)disp	memory[reg[rt]+disp] = reg[rs]
add rs,(rt)disp	reg[rs] = reg[rs] + memory[reg[rt]+disp]
sub rs,(rt)disp	reg[rs] = reh[rs] - memory[reg[rt]+disp]

(d) Base-Index-Displacement addressing. Add the following instructions to the instruction set described above:

Instruction	Meaning
lod rs,(rt,rx)disp	reg[rs] = memory[reg[rt] +reg[rx]+disp]
sto rs,(rt,rx)disp	memory[reg[rt] +reg[rx]+disp] = reg[rs]
add rs,(rt,rx)disp	reg[rs] = reg[rs] + memory[reg[rt]+reg[rx] +disp]
sub rs,(rt,rx)disp	reg[rs] = reh[rs] - memory[reg[rt]+reg[rx] +disp]

2. Assume you are given the instruction set from the problem above, and assume that register r0 always contains 0. Also assume that memory has been initialized as shown in the memory dump in Fig 9.5. Show the value stored in register r1 when the label `done` is reached for each of the following code segments:

(a)

```
        lod   r1, 0x100
        add   r1, 0x104
        bne   r1, r0, done
        sub   r1, 0x104
done:
```

(b)

```
        lod   r1, 0x100
        addI  r1, 0x104
        bne   r1, r0, done
        subI  r1, 0x104
done:
```

(c)

```
        lod   r1, 0x104
        add   r1, (r1)8
        bne   r1, r0, done
        sub   r1, 0x100
    done:
```

(d)
```
        lod   r1, 0x108
        lod   r2, (r0,r0)0x110
        sub   r2, (r0,r0)0x100
        lod   r1, (r1,r2)8
        bne   r1, r0, done
        sub   r1, (r0,r0)0x100
    done:
```

3. A multiprogramming system is one in which several programs, located in separate areas of memory, are executing at the same time. Explain why addressing modes with a base register, such as base-displacement addressing or base-index-displacement addressing, are important in a multiprogramming systemt.

9.3 ARM

ARM (Advanced RISC Machine) was first produced in the early 1980's by the British corporation Acorn Computers. RISC is a Reduced Instruction Set Computer. These computers typically have many registers, but just a few instructions in the instruction set, and have often outperformed computers with many more instructions.

9.3.1 Registers and instruction formats

The ARM processor has 32 registers; in some versions the registers are 64 bits and in other versions the registers are 32 bits. In assembly language, the register names are all `X` followed by a register number. Register 31 may also be referred to as `XZR` and always stores the value 0. The register names and conventions are shown in Fig 9.6

Most of the instructions in the ARM are identical to MIPS instructions; however the instruction formats are different, and are described in Fig 9.7.

R Format Instructions

An example of an R format instruction is the `ADD` instruction. In ARM assembly language it is:

```
ADD   Xd, Xn, Xm
```

The intent is that registers Xn and Xm are added, with the result placed in register Xd:

$Xd \leftarrow Xn + Xm$

For example, the instruction below will add the contents of registers X3 and

Register Name	Register Number	Use	Preserved Across a call?
X0..X7	0..7	Args/results	No
X8	8	Indirect result	No
X9..X15	9..15	Temporary	No
X16..X17	16..17	Reserved for Linker	No
X18	18	Reserved for platform	No
X19..X27	19..27	Saved regs	Yes
X28(SP)	28	Stack pointer	Yes
X29(FP)	29	Frame pointer	Yes
X30(LR)	30	Return address	Yes
XZR	31	Constant 0	

Figure 9.6: Register names and conventions for the ARM processor

R Format

Bits	Value
0..4	Rd register
5..9	Rn register
10..15	Shift amount
16..20	Rm
21..31	Opcode

I Format

Bits	Value
0..4	Rd register
5..9	Rn register
10..21	Immediate
22..31	Opcode

D Format

Bits	Value
0..4	Rd register
5..9	Rn register
10..11	Op
12..20	Dt Address
21..31	Opcode

B Format

Bits	Value
0..25	Br Address
26..31	Opcode

CB Format

Bits	Value
0..4	Rt register
5..23	Br Address
24..31	Opcode

IW Format

Bits	Value
0..4	Rd register
5..20	Immediate
21..31	Opcode

Figure 9.7: Instruction formats for the ARM processor

X9, placing the result in register X12:

```
ADD   X12, X3, X9
```

The R format instruction for subtraction corresponds exactly to the instruction for addition. To subtract the X3 register from the X9 register, putting the result into the X1 register:

```
SUB   X1, X9, X3
```

There are instructions for multiplication and division. Recall that when multiplying two n-bit values, the result could require 2n bits. In the ARM architecture fixed point multplication is handled with a few instructions. Assuming the registers are 64-bit registers, the `MUL` instrction can be used to multiply two registers, storing a 64-bit result in a third register. To multiply the X7 register by the X8 register, leaving the 64-bit product in the X2 register:

```
MUL   X2, X7, X8
```

If the result exceeds 64 bits, the above instruction will produce the low order 64 bits. To obtain the high order 64 bits we must use either the SMULH instruction (for a signed multiply) or the UMULH instruction (for an unsigned multiply). If the previous example produces a result which exceeds 64 bits, we can put the high order 64 bits of the result into register X3 as shown below:

```
SMULH   X3, X7, X8
```

For division we can obtain the quotient for a fixed point divide using the `SDIV` instruction for a signed divide, or the `UDIV` for an unsigned divide. For example to divide the X3 register by the X1 register, putting the signed quotient in the X10 register:

```
SDIV    X10, X3, X1
```

The ARM processor has no instruction to produce the remainder when x is divided by y. To obtain the remiander, one would need to use the identity shown below, where the division is an integer division:

```
x % y = x - x/y *y
```

I Format Instructions

An example of an I format instruction is the `ADDI` (Add Immediate) instruction. In ARM assembly language it is:

```
ADDI   Xd, Xn, Imm
```

The intent is that the result of adding the Imm field to the Xn register is placed in register Xd:

$Xd \leftarrow Xn + Imm$

For example, the instruction below will add the contents of register X3 and 260, placing the result in register X12:

```
ADDI   X12, X3, 260
```

D Format Instructions

An example of a D format instruction is the LDUR (LoaD Unscaled Register) instruction. In ARM assembly language it is:

```
LDUR   Xt, [Xn, #DtAddress]
```

This is a memory reference instruction; the effective memory address is in register Xn, with DtAddress as the offset. The referenced word is loaded into register Xn.

$Xn \leftarrow Mem[Xn + DtAddress]$

For example, the instruction below will load the memory word whose address is the sum of the X3 register plus 48 into register X8.

```
LDUR   X8, [X3, #48]
```
[4]

Note that if you are loading a full word from an array, the array index should be multiplied by 4 to get the DtAddress, since there are 4 bytes in a word, and the memory is byte addressable. This instruction is said to be *unscaled* because the offset is a byte address. There is also a load instruction which scales the offset by the size of the word being loaded. LDR scales the offset by multiplying it by 4, to get the effective address of a given position in an array of full words. There are also STUR and STR instructions to store a register into memory, with unscaled and scaled offsets, respectively.

B Format Instructions

A B format instruction is used for unconditional branch instructions (these were called *jump* instructions in MIPS). In assembly language we would typically have a label as the target:

```
  B   Label     // jump to Label
```

The assembler finds the memory address associated with the Label, and fills in a 26 bit address for the branch. There is also a BL instruction, Branch and Link, for function calls. It stores the return address in the X30 register (LR) and branches to the function. The BR Branch Register instruction is R formant, and branches to the address in the Rt register. It is used to return to the calling function.

9.3.2 Conditional branch instructions - CB format

Testing for equality

The CB format is used for *conditional branch* instructions. The CB instructions have a 26-bit branch address. The CBZ (conditional branch if zero) instruction branches to the branch address if the Rt register contains 0, and the CBNZ (conditional branch if not zero) instruction branches to the branch address if the Rt register does not contain 0. In assembly language the format would be:

```
CBZ   Xt   Label      // branch to Label if Xt is 0
```

An example would be:

[4] The hash mark, #, is used for an address displacement, and not to begin a comment. To begin a comment, use //, as in C++ or Java

Mnemonic	Instruction	Mnemonic	Instruction
ADDS	Add and set flags	ADDIS	Add immediate and set flags
ANDS	AND and set flags	ANDIS	AND immediate and set flags
SUBS	Subtract and set flags	SUBIS	Subtract immediate and set flags

Figure 9.8: Instructions which set (or clear) the condition code flags N,Z,V,C

Flag	Name	Detected in result (Flag=1)
N	Negative	High order bit = 1
Z	Zero	All bits are zeros
V	Overflow	Carry into high order bit $\neq$ carry out of high order bit
C	Carry	Carry out of or borrow into high order bit

Figure 9.9: Description of the condition code flags N,Z,V,C

```
CBZ   X3, done        // branch to done if X3 is 0
```

The following instruction will branch to `lp` if register X5 contains a non-zero value: `CBNZ   X5, lp        // Branch to lp if X5 != 0`

These instructions can be used to test for equality (or inequality) of two registers. Simply precede the conditional branch with a subtract instruction that sets/clears the flags (explained below), i.e. SUBS or SUBIS. For example, to branch to `done` if register X2 is equal to register X7, use the following pair of instructions (assuming register X9 is available for a temporary result):

```
SUBS   X9, X2, X7    // X9 = X2 - X7
CBZ    X9, done      // Branch to done if X9 == 0
```

Another example, branch to `lp` if register X3 does not contain 35:

```
SUBIS   X9, X2, 35    // X9 = X2 - 35
CBNZ    X9, lp        // Branch to lp if X9 == 0
```

Testing for order

There are also conditional branch instructions which test for the relative order of two registers, i.e. is number in one register smaller or larger than the number in another register? This kind of condition requires some additional explanation. The processor has a 4-bit *condition code*.[5] The four bits of the condition code are labeled N,Z,V,C, and we call them *flags*. They are set (or cleared) after certain operations are executed. These operations are shown in Fig 9.8, and they change the flags as shown in Fig 9.9

The result of the instruction determines which of the four flags are to be set, and which are to be cleared. For example, if the result is negative, the N flag is set to 1, but if the result is not negative (zero or positive), the N flag is cleared

[5] The use of condition codes is typical in processor design; MIPS is unusual in that it does not use condition codes for conditional branches.

```
  0101 = +5           1100 = -4
+ 0100 = +4         + 1001 = -7
----------          ----------
  1001 = -7           0101 = +5
```

Figure 9.10: Two examples of overflow when adding 4-bit words, two's complement representation

to 0. The Z flag is set to 1, only if all bits of the result are zeros. The V flag (oVerflow) is set to 1 only when *overflow* occurs. When the instruction is an ADD, overflow can occur when the addition of two positive numbers produces a negative result, or when the additon of two negative produces a positive result.

To explain the overflow condition more clearly, Fig 9.10 shows examples of overflow when adding 4-bit words (assuming two's complement represenation). These principles apply to a 32-bit word, or a word of any size.

When adding positive numbers, or when adding negative numbers, we can get an incorrect result because the result does not fit into a 4-bit word. This is called *overflow*.

Another way to detect the overflow condition is to note that a carry into the high order bit is different from the carry out of the high order bit. Of course, overflow can result from a subtract operation as well. In this case, overflow occurs when the borrow from the sign bit differs from the borrow into the sign bit.

The C flag is used primarily when the operands are unsigned, and we do not consider it here.

There are two kinds of conditional branch instructions, both of which are type CB:

- CBZ and CBNZ (discussed above) are used to branch if a given register stores 0.
- BC.cond is used to branch if the given condition is true (and should be preceded by a subtract instruction which sets the flags, such as SUBS or SUBIS).

Thus, conditional branches on inequalities ($<, \geq, ...$) are also possible. These conditions are shown in Fig 9.11. Again, these conditional branch instructions are designed with the intent that they be used after a subtract instruction is used to set/clear the condition code flags.[6]

These instructions will need to examine the N and V flags, in addition to the Z flag. To understand how the flags are used we need to recall that in two's complement representation, there are more negative numbers than there are positive numbers (as described in chapter 2). When testing for $>$ we clearly

[6] Note that there are now two ways to compare registers for equality: CBZ (after a subtract) and B.EQ, and two ways to compare registers for inequality: CBNZ (after a subtract) and B.NE

Comparison	Instruction	Flags tested
$=$	B.EQ	$Z = 0$
$\neq$	B.NE	$Z \neq 0$
$<$	B.LT	$N \neq V$
$\leq$	B.LE	(Z=0 & N=V)'
$>$	B.GT	(Z=0 & N=V)
$\geq$	B.GE	N=V

Figure 9.11: Usage of condition codes for conditional branch on equalities and inequalities (assuming the conditional branch is preceded by a subtract which sets/clears the flags)

```
(A)                   (B)
     0001 = +1            1010 = -6
   - 1000 = -8          - 0011 = +3
   ----------           ----------
     1001 = -7            0111 = +7
```

Figure 9.12: Two examples, A and B, showing why the overflow flag needs to be used for B.GT conditional branch

need to determine whether the result is positive. I.e. $x > y \;\; iff \;\; x - y > 0$. Thus after doing the subtraction, $x > y$ will be true if the result is not negative and also not zero (i.e. N=0 and Z=0). However, consider the cases shown in Fig 9.12, in which we are working with 4-bit words to save space.

In the first case, we wish to test whether +1 is greater than -8. The subtraction results in a negative number, which would normally indicate a false result; however we desire a true result because +1 is greater than -8. The disparity results from overflow on the subtraction.

In the second case, we wish to test whether -6 is greater than +3. The subtraction results in a positive number, which would normally indicate a true result; however we desire a false result because -6 is not greater than +3. Again, the disparity results from overflow on the subtraction.

To determine exactly how the flags should be tested, we use a truth table, shown in Fig 9.13. Assume we are subtracting x-y to determine whether $x > y$. For each possible setting of the three condition code flags, Z,N,V, we show whether x is greater than y, with a note in the last column referring to our two cases in Fig 9.12. [7]

To derive the boolean expression shown in Fig 9.11, for the B.GT instruction, we will use the Karnaugh map shown in Fig 9.14.

. In this K-map the Z flag is represnted by x, the N flag is represented by y, and the V flag is represented by z.

Using what we learned in chapter 6, we can derive a boolean expression from

[7] The last three cases in Fig 9.13 could be shown as don't cares because they cannot occur, but this will not lead to a simpler result.

Z N V	$x > y$	Overflow example (Fig 9.12)
0 0 0	1	
0 0 1	0	B ($-6 > 3$)
0 1 0	0	
0 1 1	1	A ($+1 > -8$)
1 0 0	0	
1 0 1	0	
1 1 0	0	
1 1 1	0	

Figure 9.13: Flags tested for a conditional branch if strictly greater (B.GT) instruction

	yz 00	yz 01	yz 11	yz 10
x=0	1	0	1	0
x=1	0	0	0	0

Figure 9.14: A Karnaugh map derived from Fig 9.13 showing how the condition code flags are used in the B.GT instruction

this K-map (there are no groups of adjacent 1's):
$x'y'z' + x'yz$
which represnets:
$Z'N'V' + Z'NV$
$= Z'(N'V' + NV)$
$= Z'(N = V)$
which agrees with Fig 9.11 for B.GT.

Testing condition code flags

ARM also provides instructions which can test individual condition code flags. These instructions are summarized in Fig 9.15. These instructions should be preceded by an instruction which sets the condition code flags, such as ADDS (Add and Set flags). The B.MI instruction will branch if the previous instruction had a negative result, in which case the N flag would be 1. The B.PL instruction is a slight misnomer. It will branch if the previous instruction had a *non-negative* result. Thus it will branch if the result was positive or zero[8], in which case the N flag would be 0. The B.VS instruction will branch if the previous instruction resulted in overflow, in which case the V flag would be 1. The B.VC instruction will branch if the previous instruction did not result in overflow, in which case the V flag would be 0.

As an example, suppose we wish to branch to the label `notNeg` if the result of subtracting registers X2 from register X5 results in a 0 or positive value (putting the result into register X1). We could use the following pair of instructions:

```
SUBS   X1, X5, X1
B.PL   error
```

Another example, suppose we wish to branch to the label `error` if the addition of registers X3 and X5 results in overflow (and we don't mind clobbering register X3). We could use the following pair of instructions:

```
ADDS   X3, X3, X5
B.VS   error
```

9.3.3 Exercises

1. Show ARM code to perform each of the following calculations (assume all operations are done on signed quantities):

   (a) X4 = X3 + (X2 - X1)

   (b) X4 = X2 + Memory[X19 + 32]

   (c) (X2,X3) = X5 * X6

2. (a) Show ARM code to jump unconditionally to the label `done`.

[8] Zero is neither negative nor positive.

Mnemonic	Name	Meaning	Flag tested
B.MI	Branch if minus	Branch if the previous instruction had a negative result	N=1
B.PL	Branch if plus	Branch if the previous instruction had a non-negative result	N=0
B.VS	Branch if overflow set	Branch if the previous instruction resulted in overflow	V=1
B.VC	Branch if overflow clear	Branch if the previous instruction did not result in overflow	V=0

Figure 9.15: ARM instructions which test an individual condition code flag (assuming a prior instruction set the flags)

(b) Show ARM code to branch to the label `lp` only if register X3 is equal to register X5. Assume register X1 is available for temporary use.

(c) Show ARM code to branch to the label `lp` only if register X3 is less than or equal to register X5. Assume register X1 is available for temporary use.

(d) Show ARM code to branch to the label `lp` only if register X3 is greater than register X5. Assume register X1 is available for temporary use.

3. (a) Show ARM code to call a function named FN1.

   (b) Show ARM code to return to the calling function.

4. Explain why each of the last three entries in Fig 9.13 can be don't-cares.

5. Figures 9.11 and 9.14 describe how the condition code flags are used in the B.GT instruction to branch if the first operand is strictly greater than the second operand. Show a similar map and the resulting boolean expression for each of the following instructions (your solution should agree with Fig 9.11):

   (a) B.LE (Branch if less than or equal to)

   (b) B.GE (Branch if greaterthan or equal to)

   (c) B.LT (Branch if strictly less than)

9.4 Intel Pentium

Since the advent of microcomputers in the late 1970's, the Intel series of microprocessor chips has been the industry leader. This series of CPUs includes, in chronological order, the 8086, 8088, 286, 386, 486, Pentium, Itanium, Core, and Celeron. Intel is the largest CPU chip manufacturer (by revenue).

The Intel series of CPUs are *backward compatible*, i.e. software which will execute on one of the CPUs will also execute on any of its successors. Thus,

Register	Usage	Low-order 16 bits
EAX	Accumulator	AX
EDX	Data	DX
ECX	Counter	CX
EBX	Base Register	BX
EBP	Base Pointer	BP
ESI	Source Index	SI
EDI	Destination Index	DI
ESP	Stack Pointer	SP
CL	Shift amount	

Figure 9.16: Pentium CPU Registers

for example, a program that runs on a 386 will also run on a Pentium (but a program on that runs on a Penitum is not assured to run on a 386). The architectures are all two-address architectures, with the capability of including memory operands for most instructions.

An instruction set is said to be *orthogonal* if all addressing modes are available across all instruction (or data) types. The Intel processors are *not* orthogonal, which makes them difficult to program.[9]

In this section we describe some aspects of the Intel Pentium architecture. We will describe the CPU registers, condition code flags, instruction set, and addressing modes.

9.4.1 CPU Registers

The CPU registers for the Pentium are 32-bit registers (though instructions can utilize the low-order 16 bits or low-order 8 bits for backward compatibility). Subsequent CPUs such as the Celeron use 64-bit registers. The Pentium has 8 such registers, which are named EAX, EBX, ECX, EDX, EBP, ESI,and ESP in assembly language programs. Fig 9.16 describes the purpose and actual register number of each.

Here the E stands for 'Extended' because earlier processors which used 16-bit registers had the same register names without the E (AX, BX, CX, ...).

9.4.2 The Mov Instruction and Addressing Modes

In the Pentium architecture there are no Load nor Store instructions; instead there is a *mov* (short for move) instruction. It is used to transfer data between a register and memory, or from an immediate field to a register or memory. It can also be used to transfer data from a register to another register. The uses

[9] In the mid 1980's Motorola Corp. introduced a competing series of microprocessors, the M68000 series, which *were* orthogonal. These chips were used in Apple Macintosh computers, whereas the Intel chips were used in IBM compatible computers. Ultimately the Intel chips were favored, and Apple dropped the Motorola chips from future desktop computers.

Instruction	Semantics	Description
mov r1, r2	$Regs[r1] \leftarrow Regs[r2]$	Copy from register to register
mov r1, memory	$Regs[r1] \leftarrow memory$	Load register from memory
mov memory, r1	$memory \leftarrow Regs[r1]$	Store register to memory
mov r1, imm	$Regs[r1] \leftarrow imm$	Load register from immediate field of instruction
mov memory, imm	$memory \leftarrow imm$	Store immediate field of instruction to memory

Figure 9.17: Uses of the Pentium mov instruction

Base Register	Index Register	Scaling Factor	Displacement	Effective Address
1001f04c	00000000	00000008	00000014	1001f060
1001f04c	0000000d	00000004	00000000	1001f080
1001f04c	0000000d	00000008	00000014	1001f0c8

Figure 9.18: Examples of Pentium effective addresses

of the `mov` instruction are summarized in Fig 9.17. Note that the immediate operand can be a full word.

Fig 9.17 does not show how a memory operand is specified. Here we describe the addressing modes used to form an effetive memory address. The Pentium architecture uses a variation on the base-index-displacement addressing mode, described earlier in this chapter. The only difference is that it includes a *scaling factor* for the index register. The effective memory address is thus

`base + sc * idx + disp`

where `base` is the base register, `sc` is the scale factor (which must be 1, 2, 4, or 8), and `disp` is the displacement. Thus, a `mov` instruction which references memory would have two registers and two constants for the memory address operand. Examples memory addresses are shown in Fig 9.18.

The scaling factoris designed to make array processing efficient. For example, to step through an array of full words, one would use a scaling factor of 4, since there are 4 bytes in a word. The index register would contain the index of the array position being accessed; thus, an value of 3 in the index register would address position 3 of the array. This design also optimizes matrix multiplication, which is an important operation in many scientific and simulation applications.

These addressing modes can be expressed in assembly language in many ways (not all of which are available with all assemblers). A few examples are shown in Fig 9.19

In assembly language memory is accessed through symbolic addresses as well, which we will describe later.

In assembly language, the *lea* instruction, load effective address, can be used to load the memory address of a symbolic memory location into a register. For example,

Assembly Language	Effective Address	Description
mov EAX, [EBX]	EBX	EBX is base register
mov EAX, [EBX+16]	EBX + 16	EBX is base register displacement is 16
mov EAX, [EBX+ESI]	EBX + ESI	EBX is base register ESI is index register
mov EAX, [EBX+4*ESI+80]	EBX + ESI*4 + 80	EBX is base register ESI is index register scale factor is 4 displacement is 80

Figure 9.19: Examples of explicit Pentium memory addresses in assembly language

```
   lea    EBX, myArray
```

will load the effective address of `myArray` into the EBX register.

9.4.3 Arithmetic Instructions

In this section we describe the arithmetic instructions of the Pentium instruction set. These include instructions to add, increment, subtract, decrement, compare, multiply, and divide integers. We begin with the instructions to add, subtract, and compare integers. They are summarized in Fig 9.20. For addition and subtraction the operands may both be registers, or one of the operands may be in memory. The first operand always stores the result of the operation, even if it is a memory operand. In assembly language, to calculate `result = (a+b)-(c+d)`, we could use the following sequence of instructions:

```
mov   EAX,a            ; EAX <- a
add   EAX,b            ; EAX <- a + b
mov   result,EAX       ; result <- a + b
mov   EAX,c            ; EAX <- c
add   EAX,d            ; EAX <- c + d
sub   result,EAX       ; result <- (a+b) - (c+d)
```

The increment (`inc`) and decrement (`dec`) operations are supplied merely for convenience; they provide a convenient way to increment or decrement a register of memory location. For example, a counter could be incremented by one with the instruction `inc counter`

Fig 9.20 also shows whether the instructions set the CPU flags (more on this later).

Mnemonic	Operand Formats	Description	Sets Flags
add	reg1, reg2	$reg1 \leftarrow reg1 + reg2$	Y
	reg, memory	$reg \leftarrow reg + memory$	
	memory, reg	$memory \leftarrow memory + reg$	
inc	reg	$reg + +$	Y
	memory	$memory + +$	
sub	reg1, reg2	$reg1 \leftarrow reg1 - reg2$	Y
	reg, memory	$reg \leftarrow reg - memory$	
	memory, reg	$memory \leftarrow memory + reg$	
dec	reg	$reg - -$	Y
	memory	$memory - -$	
cmp		Result is not used	
	reg1,reg2	reg1 - reg2	Y
	reg,memory	reg - memory	Y
	reg, imm	reg - imm	Y
	memory, imm	memory - imm	Y

Figure 9.20: Pentium add, subtract, and compare instructions

The compare (`cmp`) instruction is used in connection with the CPU flags and conditional branching, as described later in this section. It does not change any of the CPU registers nor memory locations.

There are also multiply and divide instructions for fixed-point integer data. These require more explanation, but are summarized in Fig 9.21. As noted in chapter 3, when multiplying two n-bit values, the result will not exceed 2n bits. Also, when dividing an m-bit value by an n-bit value, the remainder cannot exceed n bits. Thus, register *pairs* are often used with multiply and divide instructions. In the Pentium architecture, the pair of registers (EAX,EDX) is used for this purpose.

Note that when dividing, it is the programmer's responsibilty to ensure that precision is not lost in the quotient.

For example, to calculate `a*b/c`, where `a,b,c` are all positive integers, we could use the following sequence of instructions.

```
    mov   EAX,a      ; EAX <- a
    mul   EAX,b      ; EAX <- a * b
    mov   EDX,0      ; EDX <- 0
    div   EAX,c      ; EAX <- a * b / c
     ; EDX <- a * b % c
```

This also provides `a*b%c` in the EDX register as a side benefit; it is not necessary to do the division twice to get both the quotient and remainder.

The Pentium instruction set also includes multiply and divide instructions which presume the data are signed (two's complement) integers. These are the

Mnemonic	Operand	Description	Sets
	Formats		Flags
mul		Unsigned multiplication	
	EAX,reg	$(EDX, EAX) \leftarrow EAX * reg$	N
	EAX,memory	$(EDX, EAX) \leftarrow EAX * memory$	N
imul		Signed multiplication	
	reg	$(EDX, EAX) \leftarrow EAX * reg$	Y
	memory	$(EDX, EAX) \leftarrow EAX * memory$	Y
imul		Signed multiplication	
		Single precision	
	reg1,reg2	$reg1 \leftarrow reg1 * reg2$	Y
	reg,memory	$reg1 \leftarrow reg1 * memory$	Y
imul		Signed multiplication	
		Single precision	
	reg1,reg2,imm	$reg1 \leftarrow reg2 * imm$	Y
	reg,memory,imm	$reg \leftarrow memory * imm$	Y
div		Unsigned division	
	EAX,reg	$EAX \leftarrow (EDX, EAX)/reg$	N
		$EDX \leftarrow (EDX, EAX)\%reg$	
	EAX,memory	$EAX \leftarrow (EDX, EAX)/memory$	N
		$EDX \leftarrow (EDX, EAX)\%memory$	
idiv		Signed division	
	EAX,reg	$EAX \leftarrow (EDX, EAX)/reg$	N
		$EDX \leftarrow (EDX, EAX)\%reg$	
	EAX,memory	$EAX \leftarrow (EDX, EAX)/memory$	N
		$EDX \leftarrow (EDX, EAX)\%memory$	

Figure 9.21: Pentium multiply and divide instructions

imul and idiv instructions. They are similar to the unsigned instructions, with the inclusion of an immediate operand for multiply. If two (or three) operands are provided with an imul instruction, it is assumed that the result will fit in a single (32-bit) register. These instructions are also summarized in Fig 9.21

For example, to calculate result = a/b*17 for signed quantities, we could use the following instructions:

```
mov   EDX, 0          ; EDX <- 0
mov   EAX, a          ; EAX <- a
idiv  EAX, b          ; EAX <- a/b
imul  EAX, EAX, 17    ; EAX <- a/b*17
mov   result, EAX     ; result <- a/b*17
```

The multiply instructions which do not use the (EDX,EAX) register pair presume that the results will fit in a single 32-bit register; it is the programmer's responsibility to ensure that this is the case.

9.4.4 Logical Instructions

The Pentium instruction set includes all the essential logical instructions, with a wide variety of addressing modes, including not, and, or, xor. There is also a test instruction which produces no explicit result, but sets the condition code flags with a logical and operation. These instructions are summarized in Fig 9.22

It is clear in Fig 9.22 that the Pentium instruction set is not orthogonal.

These logical instructions are capable of operating on full 32-bit words, or on 8-bit bytes. By inserting the word byte in an assembly language statement, the programmer can specify the 8-bit operation as opposed to the 32-bit operation. The logical instructions all set the condition code flags, and all but the test instruction produce explicit results. Some examples of logical instructions are shown below:

```
not    EAX          ; EAX <- EAx', one's complement
not    result       ; result <- result', one's complement
and    EAX,EDX      ; EAX <- EAX and EDX
or     byte EDX,x   ; EDX[0..7] <- EDX[0..7] or x
test   byte x,17    ; x and 0001 0001
```

The last instruction above uses the value 17 as an immediate operand, forms the logical and with the one-byte memory operand x, discards the result, and sets the condition code flags accordingly. Note that to specify an 8-bit operation the programmer inserts the word byte into the statement.

As an example we show the instructions which will evaluate the logical expression, EAX = x and (EAX xor EDX'), where x is a 32-bit word in memory.

```
not    EDX     ; EDX <- EDX'
xor    EAX,EDX ; EAX <- EAX xor EDX'
and    EAX,x   ; EAX <- x and (EAX xor EDX')
```

Mnemonic	Operand	Description	Sets
	Formats		Flags
not	reg	$reg \leftarrow reg'$	Y
	byte reg	Low order byte of reg	
	memory	$memory \leftarrow memory'$	
	byte memory	Low order byte of memory	
and	reg1,reg2	$reg1 \leftarrow reg1\ op\ reg2$	Y
or	reg,memory	$reg \leftarrow reg\ op\ memory$	
xor	byte reg1,reg2	Low order byte of regs	
	byte reg,memory	Low order byte	
	memory,reg	$memory \leftarrow memory\ op\ reg$	
	byte memory,reg	Low order byte	
	reg,imm	$reg \leftarrow reg\ op\ imm$	
	byte reg,imm	Low order byte	
	memory,imm	$memory \leftarrow memory\ op\ imm$	
	byte memory,imm	Low order byte	
test	reg1,reg2	$reg1\ and\ reg2$	Y
	byte reg1,reg2	Low order byte of regs	
	memory,reg	$memory\ and\ reg$	
	byte memory,reg	Low order byte	
	reg,imm	$reg\ and\ imm$	
	byte reg,imm	Low order byte	
	memory,imm	$memory\ and\ imm$	
	byte memory,imm	Low order byte	

Figure 9.22: Pentium logical instructions

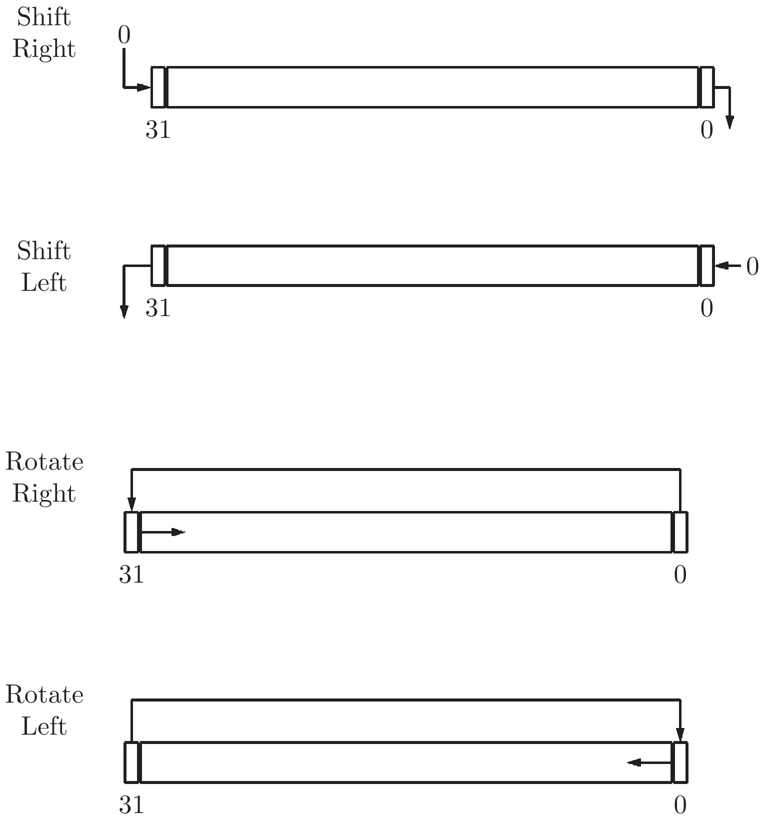


Figure 9.23: One-bit shift vs. one-bit rotate

9.4.5 Shift/Rotate Instructions

In addition to shift instructions, the Pentium also has *rotate* instructions. Rotate instructions differ from shift instructions in that the bit which is shifted from one end is shifted back in at the other end, forming a circular shift. Fig 9.23 shows the difference between a shift and a rotate.

The shift operations shown in Fig 9.23 are known as *logical* shift instructions because they operate on unsigned data. There are also *arithmetic* shift instructions which preserve the sign of the value being shifted. The rotate instructions are both *logical* rotates; they operate on unsigned values. The instruction mnemonics and operands for shift and rotate are shown in Fig 9.24

The operands for shift and rotate apply to all the instructions shown in Fig 9.24. They are described in Fig 9.25.

Some examples are shown below:

```
shl   EDX,1      ; shift EDX reg 1 bit left,
                         ; do not preserve the sign
```

Mnemonic	Description	Sets Flags
sal	shift arithmetic left	ZF, SF
shl	shift (logical) left	ZF, SF
sar	shift arithmetic right	ZF, SF
shr	shift (logical) right	ZF, SF
rol	rotate (logical) left	OF, CF
ror	rotate (logical) right	OF, CF

Figure 9.24: Pentium shift and rotate instructions

First Operand	Second Operand	Description
reg	1	reg is shifted/rotated 1 bit
memory	1	memory word is shifted/rotated 1 bit
reg	CL	reg is shifted/rotated; shift amt in CL reg
memory	CL	memory word is shifted/rotated; shift amt in CL reg
reg	imm	reg is shifted/rotated; shift amt in imm field
memory	imm	memory word is shifted/rotated; shift amt in imm field

Figure 9.25: Operands for the Pentium shift and rotate instructions

Flag	Name	Purpose
OF	Overflow	1: Result exceeds range of integers
SF	Sign	1: Result is negative
ZF	Zero	1: Result is zero
PF	Parity	1: Least significant byte of result has an even number of 1's
CF	Carry	1: Carry out of most significant bit of result is a 1

Figure 9.26: Pentium condition code flags

```
    sar   ECX,CL     ; shift ECX reg # of bits in CL reg, preserve sign
    rol   EAX,3      ; rotate EAX reg left 3 bits,
                             ; do not preserve the sign
```

9.4.6 Transfer of Control Instructions and Condition Code Flags

Condition code flags

For conditional flow of control, the Pentium uses several CPU flags, which are in a 32-bit register called EFLAGS. Not all of these bits are used in the Pentium; some are reserved for use in future versions. The five bits which are used are shown in Fig 9.26.

The *overflow* flag, OF, is set if the result of the preceding instruction exceeds the capacity of the result. More technically, the OF flag is set if the carry into the sign bit differs from the carry out of the sign bit.

The *sign* flag, SF, is set if the result of the preceding instruction is negative. The SF flag is cleared if the result is not negative, i.e. zero or positive.

The *zero* flag, ZF is set if the result of the preceding instruction is zero. The ZF flag is cleared if the result is not zero, i.e. negative or positive.

The *parity* flag, PF is set if the least significant byte of the result of the preceding instruction has an even number of 1's. The PF flag is cleared if the least significant byte of the result has an odd number of 1's.

The *carry* flag, CF is assigned the value carried out of the most significant bit of the previous instruction.

Transfer of control instructions

In the Pentium architecture all transfer of control instructions are referred to as 'jump' instructions, whether they are conditional or unconditional. The conditional jump instructions make use of one or more condition code flags (which are set by a preceding instruction). Thus two instructions are typically needed to make a conditional jump, one instruction to set the flags, and one instruction to jump. The jump instructions are shown in Fig 9.27.

Mnemonic	Description	Example(s)	Flags
jmp	Unconditional jump to dest	jmp done	
jg jnle	Jump to dest if positive	sub EAX,0 jg lp sub EAX,0 jnle lp	$(SF \oplus ZF \oplus SF)'$
jge jnl	Jump to dest if not negative	sub EAX,EDX jge lp sub EAX,EDX jnl lp	$(SF \oplus OF)'$
jl jnge	Jump to dest if negative	sub EAX,EDX jl lp sub EAX,EDX jnge lp	$SF \oplus OF$
jle jng	Jump to dest if neg or 0	sub EAX,EDX jle lp sub EAX,EDX jng lp	$(SF \oplus OF) + ZF$
jns	Jump to dest if not neg	sub EAX,EDX jns lp	SF'
jo	Jump to dest if overflow	sub EAX,EDX jo lp	OF
je jz	Jump to dest if 0	sub EAX,EDX je lp sub EAX,EDX jz lp	ZF
jne jnz	Jump to dest if not 0	sub EAX,EDX jne lp sub EAX,EDX jnz lp	ZF'

Figure 9.27: Pentium transfer of control instructions

Each jump instruction jumps to an instruction at a specified memory address. This address may be formed in the same way as it is for the `mov` instruction (see Fig **??**). In assembly language this is generally specified symbolically, giving the label of the instruction to which the jump occurs.

In Fig 9.27 the terminology used assumes that the jump instruction is preceded by a subtract instruction. Thus, if the subtraction `a-b` is performed, a negative result implies that `a < b`, and consequently a `jl` instruction will take the branch in this case. Note that it is possible for the subtraction to result in overflow, which is considered in the fourth column of Fig 9.27. Derivation of the Flag settings is left as an exercise.

9.5 Example program

Here we present an example of a Pentium assembly language program. In this example we count the number of 1's in the EDX register. The result will be left in the EAX register.

```
;  Count number of 1's in EDX reg.
   mov    ECX,33     ; loop counter
   mov    EAX,0      ; counter for result

lp:
   dec    ECX        ; ECX--
   jz     done
   mov    tmp,EDX    ; save EDX
   and    EDX,1      ; test low order bit
   jz     noIncr     ; if zero, no increment
   add    EAX,1      ; incremnt counter
noIncr:
   mov   EDX,tmp     ; load saved word
   shr   EDX,1       ; shift right logical
   jmp   lp
done:
     ; finished, result in EAX reg.
```

9.6 Exercises

1. For each of the following instructions, describe the effect it would have on a register or memory location. Also, indicate which instructions are not valid. In each case assume the EAX register contains 17, the EDX register contains 19, and the memory location `loc1` contains -2, and the memory location `loc2` contains -3.

   (a) `mov EAX,13`

   (b) `mov 13,EAX`

(c) `mov EAX,EDX`

(d) `mov EAX,loc1`

(e) `mov loc1,EAX`

(f) `mov loc2,loc1`

2. The Pentium architecture uses base-index-displacement addressing. To step through an array of double-words (64-bits each), starting at memory address 10014cd0H:

   (a) What value would be stored in the base register?

   (b) What value would initially be stored in the index register?

   (c) What would the scaling factor be?

   (d) By how much would the index register be incremented on each step?

3. Given the array of the preceding problem, show the instruction which will increment the value at position 23 in that array. Assume the EBP register contains 10014cd0H and the ESI register contains 00000017H = 23.

4. Show the pair of instructions which will compute (x-y)+1, where the value of x is in the EAX register, and the value of y is in the EDX register. Leave the result in the EAX register.

5. Show the assembly language code which will compute (x-y)*(x/y), where the value of x is in the EAX register, and the value of y is in the ECX register. Leave the result in the EAX register. Use a memory location, `tmp`, for temporary storage.

6. Show the pair of instructions which will do each of the following (do not use shift instructions, but assume the EDX register is available for use):

   (a) Clear the low order 8 bits of the EAX register

   (b) Set the high order 16 bits of the EAX register

   (c) Complement bits 8..23 of the EAX register

7. Show a single instruction which will do each of the following:

   (a) Shift the contents of the EAX register to the left by 1 bit, preserving the sign of the number.

   (b) Shift the contents of the EDX register to the right by 3 bits. Do not preserve the sign; assume it is an unsigned value.

   (c) Multiply the EBX register by 32, leaving the result in the EBX register, assuming it is a signed value. Use a shift instruction.

   (d) Divide the EDX register by 8, leaving the result in the EDX register, assuming it is unsigned. Use a shift instruction.

(e) Multiply the EBX register by 42, leaving the result in the EBX register, assuming it is a signed value. Use three shift instructions, and two add instructions. Assume the EDX register is available for use. (In this case use as many as 6 instructions)

(f) Rotate the EAX register to the left by 3 bits, leaving the result in the EAX register.

8. Show the optimal (in some sense, best) code needed to set the EDX register to 1 if it is positive, to -1 if it is negative, and leave it at 0 if it is 0. This is the so-called *signum* function:

   (a) Without using any shift or rotate instructions.

   (b) Shift and rotate instructions are permitted.

9. Show how to derive the Flag settings in the first four rows of Fig 9.27. Assume the jump instruction is preceded by a subtract (or compare) instruction.
   Hint: Use a 3-variable truth table, and a Karnaugh map, with the variables SF, OF, and ZF.

Glossary

Accumulator - The CPU register storing operation results in a one address architecture

Advanced RISC Machine - An architecture designed by the British coroporation Acorn Computers in the early 1980's (ARM)

ALU - See Arithmetic and logic unit

AND gate - A logic gate which puts out the logical AND of its inputs

API - See Application Program Interface

Application Program Interface - A description or specification for a program or function within a program

Arithmetic and logic unit - A digital component which executes several arithmetic and logical operations

Arithmetic instruction - A machine language instruction which performs an add, subtract, multiply, divide, or compare operation

Arithmetic shift - A shift operation which preserves the sign of the value

ARM - Advanced RISC Machine

Array - A group of contiguous memory locations with direct access to all members

Assembler - A program which translates a program written in symbolic assembly language to an equivalent program in binary machine language

Assembly language - A (low-level) programming language corresponding to machine language, with symbolic addresses

Associative cache memory - A cache memory in which each block of RAM maps to a set of blocks in the cache

Base displacement addressing - An addressing mode in which a register contents plus a fixed displacement form an operand address

Base index displacement addressing - An addressing mode in which the contents of two registers plus a fixed displacement form an operand address

Binary - Base two number system

Binary adder - A logic component which adds numbers in twos complement representation

Bit - A binary digit; 0 or 1

Block - A contiguous group of bytes in a cache memory

Boolean - Having the property of a bit in which 0 represents False, and 1 represents True; named after the British logician George Boole

Bootstrapping - The process of developing a translator, such as an assembler or a compiler, by developing it in its own source language

Branch instruction - See Conditional transfer of control

Byte - 8 bits

Cache memory - Fast memory used to speed up access to RAM

Cache miss - A reference to a block of RAM which is not in the cache memory

Call stack - A last-in first-out (LIFO) data structure used to save information needed to implement function calls

Canonical form - See Normal form

Central Processing Unit - That part of a computer which performs arithmetic operations and makes decisions

CISC - Complex Instruction Set Computer

Clock - A digital component which puts out repeated signals at a discrete interval

Combinational circuit - A circuit using logic gates in which there are no feedback loops

Comment - A statement in a program, ignored by the processor, to provide clarity for a human reader

Complex Instruction Set Computer - An architectue with many instructions (CISC)

Component - A logic circuit composed of logic gates and other components which can be used as a stand-alone function

Conditional code - A CPU register which stores the state of an operation, which can then be used by a conditional branch instruction

Conditional transfer of control - The process in which a program skips to another instruction, depending on the state of the machine

Contradiction - The condition of two or more output lines connecting to the same junction

Control structure - A programming entity used to affect the sequence of instructions executed

Control unit - That portion of the datapath for which the input is the instruction register and which puts out control signals to the other components

CPU - Central Processing Unit

D flip-flop - A flip-flop with one input, capable of setting or resetting the state

Datapath - That portion of the CPU which is responsible for the execution of programs

Data Locality - The degree to which successive memory references to instruction operands are located in proximity

Data Memory - A RAM module used to store program data

Decoder - A logic component with n inputs and 2^n outputs that puts out a 1 on the output line corresponding to the binary value of the input lines

Direct addressing - An addressing mode in which the instruction stores the operand's absolute memory address

Direct-mapped cache - A cache memory in which each block of RAM is is mapped directly to a block of the cache

Directive - An assembly language construct which provides information to the assembler, but which does not result in any machine language code

Dirty bit - A bit in a cache memory which indicates whether a cache block has been modified and needs to be written back to RAM; A bit in virtual memory which indicates whether a RAM page has been modified and needs to be written back to virtual memory

Disjunctive normal form - DNF - A normal form for boolean expressions

Disk - A seconday storage device

DNF - See Disjunctive normal form

Don't care - A value which is inconsequential

DRAM - Dynamic Random Access Memory (needs to be refreshed periodically)

Encoder - A logic component with 2^n inputs and n outputs that puts out the binary value corresponding to the input line which is a 1

Exclusive OR gate - A logic gate which puts out the logical Exclusive OR of its inputs

Exponent - That portion of a floating point number which is used to scale the number by a power of the base (usually either 2 or 16)

Field - A contiguous portion of an instruction or word

Field Programmable Gate Array - A digital component which can be programmed and reprogrammed to perform any digital function (FPGA)

FIFO - First-in First-out

First-in First-out - An algorithm in which the item to be removed from a data structure is the first one to be added (FIFO) Used in cache and virtual memories

Fixed disk - Secondary storage device (non-volatile) which is not removable

Flash memory - Solid state memory (non-volatile)

Flip-flop - A one-bit storage element

Floating point - A numeric data representation allowing for non-integer values very large values, and values which are very close to 0

Floating point instruction - An instruction which performs an arithmetic operation on floating point data

Floating point register - A register which stores data in floating point representation

FPGA - See Field Programmable Gate Array

Fraction - That portion of a floating point number representing the mantissa of the number, separately from the exponent

Full adder - A logic component with three inputs that puts out the logical sum and carry

Function - A group of program statements which may be invoked from elsewher in a program

Function table - A description of the operating characteristecs of a component, such as an ALU or a flip-flop

G - An abbreviation for 2^{30}

Half adder - A logic component with two inputs that puts out the logical sum and carry

Herz - A unit of frequency; one cycle per second

Hexadecimal - Base 16 number system

I format - An instruction format for immediate instructions

Immediate instruction - An instruction in which one of the operands is contained in the instruction itself

Indirect addressing - An address mode in which the instruction stores the memory address of a memory word storing the instruction's absolute address

Instruction - The basic unit of a machine language program

Instruction format - An assignment of bit fields to an instruction

Instruction Locality - The degree to which successive instructions are located in proximity

Instruction Memory - A RAM module used to store program instructions

Instruction Register - The CPU register which stores the instruction being executed (IR)

Instruction Set Architecture - The instruction formats, operations, and addressing modes included in a computer design.

Intel - Corporation with the market share of microprocessor chips

Inverter - A logic gate which puts out the logical complement of its input

IR - See Instruction register

Iteration structure - A control structure used to repeat a statement or block of statements

J format - An instruction format for jump instructions

JK flip-flop - A flip-flop which is capable of setting the state, resetting the state, complementing the state, or leaving the state unchanged

Jump and link instruction - An instruction used to invoke a function

Jump instruction - See Unconditional transfer of control

Jump register instruction - A jump instruction in which the destination address is in a register

K - An abbreviation for 2^{10}

Karnaugh map - A graphical technique used to minimize boolean expressions

Latch - A storage element used to implement a flip-flop

Latency - Time delay required for a component to respond to a signal

Least Recently Used - A cach block (or virtual memory page) algorithm which is used to determine which block (or page) is to be replaced in the cache (or RAM) (LRU)

li instruction - A pseudo-op which is used to load a constant into a register.

Load instruction - An instruction which copies a value from memory into a CPU register

Locality - The degree to which successive memory references are located in proximity

Logic gate - A fundamental (atomic in this text) component of digital circuits

Logical instruction - A machine language instruction which performs an AND, OR, Exclusive OR, NOT, NOR, NAND operation

Logical shift - A shift operation which does not preserve the sign of the value

Loop - See Iteration structure

LRU - Least Recently Used

M - An abbreviation for 2^{20}

Machine language - A (low-level) programminng language consisting of binary coded instructions to the CPU

Magnetic disk - Secondary storage device (non-volatile)

MARS - MIPS Assembler and Runtime System; developed at the University of Missouri

Mask - A bit pattern used to select certain bits from a word

Memory - Storage for instructions and data

Memory address - A binary specification of a memory location

Memory hierarchy - An ordered group of memory structures with a cost/performance trade-off

Memory reference instruction - An instruction used to load or store a register

MIPS - A RISC architecture; Microporesessor without Interlocked Pipeline States

Move operation - A pseudo-op which copies the value from one register to another register

Multiplexer - A logic component whose output is selected from several inputs

Normal form - A standard representation for values which may have many equivalent representations

Nybble - 4 bits; half byte

Octal - Base 8 number system

One address architecture - An architecture with an accumulator register which stores the results of operations

Operand - The data on which an operation operates

Optical disk - A secondary storage device which uses laser technology (non-volatile)

OR gate - A logic gate which puts out the logical OR of its inputs

Page - A contiguous group of bytes in a virtual memory

Page fault - A reference to virtual memory which is not in RAM

Parameter - Information passed to (or from) a function from the invoking function

Parity - The odd vs. even count of the 0 or 1 digits in a string of bits

Parse - To break down, or analyze, a text or number to determine its meaning or component parts

PC - See Program counter

Pentium - A popular microprocessor of the Intel Corporation, circa 1993

Peripheral devices - Devices used for permanent (non-volatile) storage or input/output (I/O)

Post-test loop - A loop in which the termination condition is tested after executing the body of the loop

Pre-test loop - A loop in which the termination condition is tested before entering the body of the loop

Program Counter - The CPU register which stores the memory address of the next instruction to be executed (PC)

Programming language - A (high-level) language with control structures and data structures used to generate a machine language program

PseudoOp - An assembly language operation which has is translated into two or more machine language instructions

R format - An instruction format for register instructions

RAM - Random Access Memory

Recursive function - A function which invokes itself

Read - A signal specifying that data is to be taken from a component's output bus

Reduced Instruction Set Computer - An architecture with a small instruction set and, typically, a large number of CPU registers (RISC)

Register - Fast, volatile storage in a CPU

Register convention - An agreement on the usage, or purpose, of various registers

Register File - A group of registers in the CPU

RISC - Reduced Instruction Set Computer

Rotate instruction - A circular shift instruction

Selection structure - A control structure used to select from one or two statements to be executed

Sequential circuit - A logic circuit with storage elements, such as flip-flops.

Shift instruction - An instruction which moves the bits in a register to the left or to the right

Sign - An indicator as to whether a number is negative, zero or positive

Sign Extend - A digital component which extends the width of a bus, while propagating the sign bit

Solid state disk - Secondary storage (non-volatile) composed of flash memory

Spatial locality - The degree to which memory references are located in proximity with respect to the number of cache blocks (or virtual memory pages)

SRAM - Static Random Access Memory (does not need to be refreshed periodically)

SR flip-flop (or latch) - A flip-flop (or latch) with two inputs, Set and Reset

State graph - A graph depicting the states and transitions of a state machine

State machine - A machine which transition from one internal state to another, depending on inputs

Statement - The basic unit of an assembly language program, corresponding to one or more machine language instructions

Store instruction - An instruction which copies a register value into memory

String - A sequence of contiguous memory locations, each of which stores one character from a given character set

Symbolic memory address - A memory address specified by a name rather than by a binary number

Syscall - A MARS statement used to call a system function, such as I/O or program termination

T - An abbreviation for 2^{40}

Tag - The portion of a memory address which stores the block number

Temporal Locality - The degree to which successive memory references are located in proximity

Thrashing - Excessively frequent cache misses in a cache memory, or excessively frequent page faults in a virtual memory

Three address architecture - An architecture in which the three memory addresses are the result, the left operand, and the right operand

Transfer of control - The process in which a program skips to another instruction non-sequentially

Two address architecture - An architecture in which the result of an operation is the same register as the left operand

Twos complement representation - A representation for signed binary integers

Unconditional transfer of control - The process in which a program skips to another instruction (not dependant on the state of the machine)

Virtual memory - An extension to the RAM using secondary storage

Volatile - Requiring continuous power to retain information

Von Neumann architecture - A classical computer design consisting of ALU, control unit, memory storing instructions and data, and external storage.

Word - A unit of memory, 32 bits in the MIPS architecture

Write - A signal to a storage component specifying that data on its input bus is to be stored

Zero address architecture - A stack machine

Appendix: MARS

The MARS (MIPS Assembler and Runtime System) can be used to assemble and execute MIPS programs; it is written in Java and should run on any computer which supports Java.

This appendix contains some basic information needed to download and use the MARS software.

.1 Downloading MARS to Your Computer

Follow the steps below to install MARS on your computer:

1. Go to the web site: http//:courses/missouristate/edu/KenVollmar/MARS
2. Click on the link Download MARS 4.5 software
3. Click on the link Download
4. Click on the link Download MARS
5. Depending on your operating system, and your security settings, you may need to allow your computer to run this software (go to Settings from the desktop to allow MARS to execute).

.2 Edit Source Files

When MARS is launched it will open a blank window. To get started, choose the **New** item from the **File** menu. You can now start typing a MIPS assembly language program, one statement on each line. To start out you could type in a few simple statements, such as:

```
li     $t0, 23            # Put 23 into register $40
add    $t1, $t0, $t0      # Register $t1 contains 46
```

Before executing your program, it must be saved to permanent storage; choose **Save** or **Save As** from the **File** menu. This is called your *source file*. It is a plain text file, and could be saved with any file extension. We usually use .asm or .s as file extensions. It could be saved as `temp.asm`. You will need to save it to a folder on some disk such as the main fixed disk for your computer. Each time you need to make changes to this source file, you can save it with the same name.

If you terminate MARS and wish to continue later, load this source file from the place where you had saved it.

.3 Assemble Source Files

To assemble your source file, do one of the following:

- Choose **Assemble** from the **Run** menu
- Use Function Key **F3**
- Click the icon which looks like a screw driver with a wrench.

If you have an incorrect statement in your source file, MARS will show error message(s) in the bottom window pane (`Messages`). Correct the errors and try to assemble again.

Once you have eliminated all syntax errors, when you assemble your source file, MARS will automatically show the `Execute` window, which normally consists of four window panes:

- The `Text Segment` pane shows your source code (or most of it) on the right. On the left it shows the machine language code (with memory addresses) which the assembler produced. The `Basic` column is an *intermediate form* which the assembler uses to translate your source file to machine language.
- The `Registers` window pane shows the value of each of 32 general registers, most of which are initially 0. (This pane should not be showing coprocessors at this point)
- The `Data Segment` pane shows the values currently stored in the Data Memory, all of which should be 0 at this point. The Data Memory addresses begin at address 0x10010000.
- The `Run I/O` pane corresponds to the former `Messages` pane. It shows error messages, along with any text output produced by your program when it executes.

.4 Execute Programs

It is now possible to execute your program and view the effects it may have on registers, memory, and output. To execute your program do one of the following:

- Choose **Go** from the **Run** menu

- Use Function Key **F5**
- Click the icon which looks like a white triangle inside a green circle.

This will execute your program at full speed, and terminate when it encounters a terminating system call (or non-valid code at the end). If you used the sample program (two statements) shown above, you can see that register $t0 has been loaded with 23 (17_{16}) and that register $t1 has been loaded with 46 ($2e_{16}$).

To execute your program in such a way that you can see intermediate results as the program executes, you have a few options:

- Slow down the run speed, using the slider at the top of the window, so that you can view changes to registers and memory as the program executes.
- Choose `Step`, click Function Key F7, or click the green circle with a 1 subscript. This will allow you to execute one statement at a time.
- Set *breakpoints* in your program by selecting one or more `Bkpt` check boxes on the left of the `Text Segment` pane. When running at full speed, execution will pause at each breakpoint.

By judiciously choosing among these options, you can diagnose difficult problems. To start over and run again, choose Function Key F12 to Reset, or click the green circle containing a double white triangle.

If there are semantic (i.e. logical) errors in your program, you will need to click on the `Edit` tab of the `Text Segment` pane (not the `Edit` menu). This will take you back to your source file. Make the necessary changes, assemble, and execute the program again to verify that it is correct.

In general you will be using an Edit/Assemble/Test cycle as you develop software.

Appendix: MIPS Instruction Set

This appendix shows selected MIPS instructions. For each instruction we show:

- A brief name of the instruction
- The assembly language mnemonic for the instruction
- The instruction format (i.e. R, I, J, FR, FI)
- The assembly language format
- The machine language format
- The semantics of the instruction (it's effect on the processor)

.5 Core Instructions

Name	Mne-monic	For-mat	Assembly Language Machine Language Semantics
Add	add	R	add $rd,$rs,$rt

31 26	25 21	20 16	15 11	10 6	5 0
00	$rs	$rt	$rd		20
opcode	rs	rt	rd	shamt	funct

$\$rd \leftarrow \$rs + \$rt$

Name	Mne-monic	For-mat	Assembly Language
Add Imm	addi	I	addi $rt,$rs,imm

31 26	25 21	20 16	15 0
08	$rs	$rt	imm
opcode	rs	rt	immediate

$\$rt \leftarrow \$rs + imm$

Name	Mne-monic	For-mat	Assembly Language
And	and	R	and$rd,$rs,$rt

31 26	25 21	20 16	15 11	10 6	5 0
00	$rs	$rt	$rd		24
opcode	rs	rt	rd	shamt	funct

$\$rd \leftarrow \$rs \wedge \$rt$

Name	Mne-monic	For-mat	Assembly Language
And Imm	andi	I	andi $rt,$rs,imm

31 26	25 21	20 16	15 0
0c	$rs	$rt	imm
opcode	rs	rt	immediate

$\$rt \leftarrow \$rs \wedge imm$

Branch Equal beq I beq $rt,$rs,addr

04	$rs	$rt	addr
31 26	25 21	20 16	15 0
opcode	rs	rt	immediate

$\rightarrow (relative)addr\ if\ \$rt = \$rs$

Branch NotEqual bne I bne $rt,$rs,addr

05	$rs	$rt	addr
31 26	25 21	20 16	15 0
opcode	rs	rt	immediate

$\rightarrow (relative)addr\ if\ \$rt \neq \$rs$

Jump j J j address

02	(absolute)address
31 26	25 0
opcode	address

$\rightarrow address$

Jump Reg jr R jr $rs

00	$rs				08
31 26	25 21	20 16	15 11	10 6	5 0
opcode	rs	rt	rd	shamt	funct

$\rightarrow \$rs$

Load Byte lbu I lbu $rt,displ($rs)

24	$rs	$rt	displ
31 26	25 21	20 16	15 0
opcode	rs	rt	immediate

$\$rt_{8..31} \leftarrow 0, \$rt_{0..7} \leftarrow Mem[\$rs + imm]_{0..7}$

Load Upper lui I lui $rt, imm

0f		$rt	imm
31 26	25 21	20 16	15 0
opcode	rs	rt	immediate

$\$rt_{0..15} \leftarrow 0, \$rt_{16..31} \leftarrow imm$

Load Word lw I lw $rt,displ($rs)

23	$rs	$rt	displ
31 26	25 21	20 16	15 0
opcode	rs	rt	immediate

$\$rt \leftarrow Mem[\$rs + imm]$

Nor nor R nor $rd,$rs,$rt

00	$rs	$rt	$rd		27
31 26	25 21	20 16	15 11	10 6	5 0
opcode	rs	rt	rd	shamt	funct

<table>
<tr><th>Name</th><th>Mnemonic</th><th>Format</th><th>Assembly Language
Machine Language
Semantics</th></tr>
<tr><td>Or</td><td>or</td><td>R</td><td>or $rd,$rs,$rt
<table>
<tr><td>00</td><td>$rs</td><td>$rt</td><td>$rd</td><td></td><td>25</td></tr>
<tr><td>31 26</td><td>25 21</td><td>20 16</td><td>15 11</td><td>10 6</td><td>5 0</td></tr>
<tr><td>opcode</td><td>rs</td><td>rt</td><td>rd</td><td>shamt</td><td>funct</td></tr>
</table>
$\$rd \leftarrow \$rs \vee \$rt$</td></tr>
<tr><td>Or Immediate</td><td>ori</td><td>I</td><td>ori $rt,$rs,imm
<table>
<tr><td>23</td><td>$rs</td><td>$rt</td><td>imm</td></tr>
<tr><td>31 26</td><td>25 21</td><td>20 16</td><td>15 0</td></tr>
<tr><td>opcode</td><td>rs</td><td>rt</td><td>immediate</td></tr>
</table>
$\$rt \leftarrow \$rs \vee imm$</td></tr>
<tr><td>Set If LessThan</td><td>slt</td><td>R</td><td>slt $rd,$rs,$rt
<table>
<tr><td>00</td><td>$rs</td><td>$rt</td><td>$rd</td><td></td><td>2a</td></tr>
<tr><td>31 26</td><td>25 21</td><td>20 16</td><td>15 11</td><td>10 6</td><td>5 0</td></tr>
<tr><td>opcode</td><td>rs</td><td>rt</td><td>rd</td><td>shamt</td><td>funct</td></tr>
</table>
$\$rd \leftarrow \$rs < \$rt ? 1 : 0$</td></tr>
<tr><td>Shift Left</td><td>sll</td><td>R</td><td>sll $rd,$rt,shamt
<table>
<tr><td>00</td><td></td><td>$rt</td><td>$rd</td><td>shamt</td><td>00</td></tr>
<tr><td>31 26</td><td>25 21</td><td>20 16</td><td>15 11</td><td>10 6</td><td>5 0</td></tr>
<tr><td>opcode</td><td>rs</td><td>rt</td><td>rd</td><td>shamt</td><td>funct</td></tr>
</table>
$\$rd \leftarrow \$rt << shamt$</td></tr>
<tr><td>Shift Right</td><td>srl</td><td>R</td><td>srl $rd,$rt,shamt
<table>
<tr><td>00</td><td></td><td>$rt</td><td>$rd</td><td>shamt</td><td>02</td></tr>
<tr><td>31 26</td><td>25 21</td><td>20 16</td><td>15 11</td><td>10 6</td><td>5 0</td></tr>
<tr><td>opcode</td><td>rs</td><td>rt</td><td>rd</td><td>shamt</td><td>funct</td></tr>
</table>
$\$rd \leftarrow \$rt >>> shamt$</td></tr>
<tr><td>Shift Right Arith</td><td>sra</td><td>R</td><td>sra $rd,$rt,shamt
<table>
<tr><td>00</td><td></td><td>$rt</td><td>$rd</td><td>shamt</td><td>03</td></tr>
<tr><td>31 26</td><td>25 21</td><td>20 16</td><td>15 11</td><td>10 6</td><td>5 0</td></tr>
<tr><td>opcode</td><td>rs</td><td>rt</td><td>rd</td><td>shamt</td><td>funct</td></tr>
</table>
$\$rd \leftarrow \$rt >> shamt$</td></tr>
</table>

Shift Left Variable   sllv   R   sllv $rd,$rs,$rt

00	$rs	$rt	$rd		04
31 26	25 21	20 16	15 11	10 6	5 0
opcode	rs	rt	rd	shamt	funct

$\$rd \leftarrow \$rt << \$rs$

Shift Right Arith Variable   srav   R   srav $rd,$rs,$rt

00	$rs	$rt	$rd		07
31 26	25 21	20 16	15 11	10 6	5 0
opcode	rs	rt	rd	shamt	funct

$\$rd \leftarrow \$rt >> \$rs$

Shift Right Logical Variable   srlv   R   srlv $rd,$rs,$rt

00	$rs	$rt	$rd		06
31 26	25 21	20 16	15 11	10 6	5 0
opcode	rs	rt	rd	shamt	funct

$\$rd \leftarrow \$rt >>> \$rs$

Store Byte   sb   I   sb $rt,displ($rs)

28	$rs	$rt	displ
31 26	25 21	20 16	15 0
opcode	rs	rt	immediate

$Mem[\$rs + imm] \leftarrow \$rt_{0..7}$

Store Word   sw   I   sw $rt,displ($rs)

2b	$rs	$rt	displ
31 26	25 21	20 16	15 0
opcode	rs	rt	immediate

$Mem[\$rs + imm] \leftarrow \$rt$

Subtract   sub   R   sub $rd,$rs,$rt

00	$rs	$rt	$rd		22
31 26	25 21	20 16	15 11	10 6	5 0
opcode	rs	rt	rd	shamt	funct

$\$rd \leftarrow \$rs - \$rt$

Excl Or   xor   R   xor $rd,$rs,$rt

00	$rs	$rt	$rd		26
31 26	25 21	20 16	15 11	10 6	5 0
opcode	rs	rt	rd	shamt	funct

$\$rd \leftarrow \$rs \oplus \$rt$

Name	Mne- monic	For- mat	Assembly Language Machine Language Semantics
Excl Or Immediate	xori	I	xori $rt,$rs,imm

31 26	25 21	20 16	15 0
0c	$rs	$rt	imm
opcode	rs	rt	immediate

$\$rt \leftarrow \$rs \oplus imm$

Name	Mne- monic	For- mat	Assembly Language
Multiply	mult	R	mult $rs,$rt

31 26	25 21	20 16	15 11	10 6	5 0
00	$rs	$rt			18
opcode	rs	rt	rd	shamt	funct

$(hi, lo) \leftarrow \$rs \cdot \$rt$

Name	Mne- monic	For- mat	Assembly Language
Divide	div	R	div $rs,$rt

31 26	25 21	20 16	15 11	10 6	5 0
00	$rs	$rt			1a
opcode	rs	rt	rd	shamt	funct

$hi \leftarrow \$rs\%\$rt, lo \leftarrow \$rs/\$rt$

Name	Mne- monic	For- mat	Assembly Language
Move From Hi	mfhi	R	mfhi $rd

31 26	25 21	20 16	15 11	10 6	5 0
00			$rd		10
opcode	rs	rt	rd	shamt	funct

$\$rd \leftarrow hi$

Name	Mne- monic	For- mat	Assembly Language
Move From Lo	mflo	R	mflo $rd

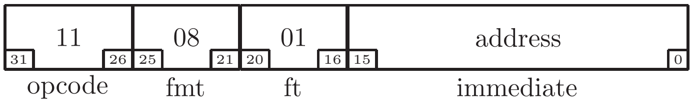
31 26	25 21	20 16	15 11	10 6	5 0
00			$rd		12
opcode	rs	rt	rd	shamt	funct

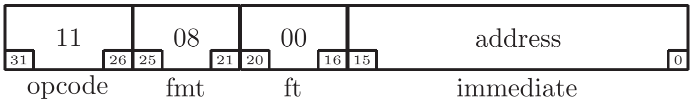
$\$rd \leftarrow lo$

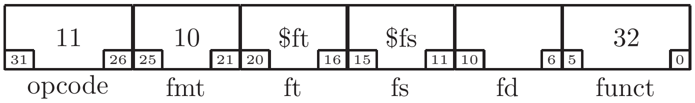
.6 Floating Point Instructions

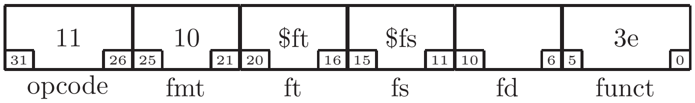
Name	Mne-monic	For-mat	Assembly Language Machine Language Semantics
Add Float	add.s	FR	add.s $fd,$fs,$ft 11 | 10 | $ft | $fs | $fd | 00 31 26 | 25 21 | 20 16 | 15 11 | 10 6 | 5 0 opcode | fmt | ft | fs | fd | funct $\$fd \leftarrow \$fs + \$ft$
Divide Float	div.s	FR	div.s $fd,$fs,$ft 11 | 10 | $ft | $fs | $fd | 03 31 26 | 25 21 | 20 16 | 15 11 | 10 6 | 5 0 opcode | fmt | ft | fs | fd | funct $\$fd \leftarrow \$fs/\$ft$
Multiply Float	mul.s	FR	mul.s $fd,$fs,$ft 11 | 10 | $ft | $fs | $fd | 02 31 26 | 25 21 | 20 16 | 15 11 | 10 6 | 5 0 opcode | fmt | ft | fs | fd | funct $\$fd \leftarrow \$fs \cdot \$ft$
Subtract Float	sub.s	FR	sub.s $fd,$fs,$ft 11 | 10 | $ft | $fs | $fd | 08 31 26 | 25 21 | 20 16 | 15 11 | 10 6 | 5 0 opcode | fmt | ft | fs | fd | funct $\$fd \leftarrow \$fs - \$ft$
Store Float	swc1	I	swc1 $ft,displ($rs) 39 | $rs | $ft | displ 31 26 | 25 21 | 20 16 | 15 0 opcode | rs | rt | immediate $Mem[\$rs + displ] \leftarrow \$ft$
Load Float	lwc1	I	lwc1 $ft,displ($rs) 31 | $rs | $ft | displ 31 26 | 25 21 | 20 16 | 15 0 opcode | rs | rt | immediate $\$ft \leftarrow Mem[\$rs + displ]$

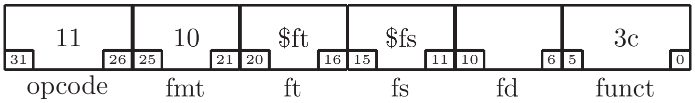
.6.1 Floating Point Conditional Branch

<table>
<tr><th>Name</th><th>Mne-
monic</th><th>For-
mat</th><th>Assembly Language
Machine Language
Semantics</th></tr>
<tr><td>Branch If FPTrue</td><td>bc1t</td><td>FI</td><td>bc1t (rel)address


→ (rel)address if FP=1</td></tr>
<tr><td>Branch If FPFalse</td><td>bc1f</td><td>FI</td><td>bc1f (rel)address


→ (rel)address if FP=0</td></tr>
<tr><td>Compare Float EQ</td><td>c.eq.s</td><td>FR</td><td>c.eq.s $fs,$ft


$FP \leftarrow \$fs = \$ft ? 1 : 0$</td></tr>
<tr><td>Compare Float LE</td><td>c.le.s</td><td>FR</td><td>c.le.s $fs,$ft


$FP \leftarrow \$fs \leq \$ft ? 1 : 0$</td></tr>
<tr><td>Compare Float LT</td><td>c.lt.s</td><td>FR</td><td>c.lt.s $fs,$ft


$FP \leftarrow \$fs < \$ft ? 1 : 0$</td></tr>
</table>

Appendix: Pseudo Operations Supported by MARS

Name	Mnemonic	Assembly Language	Semantics
Absolute Value	abs	abs rd,rs	$\$rd \leftarrow \|\$rs\|$
Branch Unconditional	b	b label	$\rightarrow label$
Branch If Equal Zero	beqz	beqz $rs, label	$\rightarrow label\ if\ \$rs = 0$
Branch If Greater Than	bgt	bgt $rs, $rt, address	$\rightarrow address(relative)\ if\ \$rs > \$rt$
Branch If Greater Or Equal	bge	bge $rs, $rt, address	$\rightarrow address(relative)\ if\ \$rs >= \$rt$
Branch If Less Than	blt	blt $rs, $rt, address	$\rightarrow address(relative)\ if\ \$rs < \$rt$
Branch If Less Or Equal	ble	ble $rs, $rt, address	$\rightarrow address(relative)\ if\ \$rs <= \$rt$
Load Immediate	li	li $rd, imm	$\$rd \leftarrow imm$
Divide	div	div $rd, $rs, $rt	$\$rd \leftarrow \$rs/\$rt$
Divide Immediate	div	div $rd, $rs, imm	$\$rd \leftarrow \$rs/imm$
Load Address	la	la $rd, label	$\$rd \leftarrow$ label's address
Move	move	move $rd, $rs	$\$rd \leftarrow \$rs$
Multiply Short	mulo	mulo $rd, $rs	$\$rd \leftarrow \$rs \cdot \$rt$
Negate	neg	neg $rd, $rs	$\$rd \leftarrow -\$rs$
Not	not	not $rd, $rs	$\$rd \leftarrow\sim \$rs$
Remainder	rem	rem $rd, $rs, $rt	$\$rd \leftarrow \$rs \bmod \$rt$
Remainder Immediate	rem	rem $rd, $rs, imm	$\$rd \leftarrow \$rs \bmod imm$
Rotate Left	rol	rol $rd, $rs, $rt	$\$rd \leftarrow \$rs \hookrightarrow \$rt$
Rotate Left Immediate	rol	rol $rd, $rs, imm	$\$rd \leftarrow \$rs \hookrightarrow imm$
Rotate Right	ror	ror $rd, $rs, $rt	$\$rd \leftarrow \$rs \hookleftarrow \$rt$
Rotate Right Immediate	ror	ror $rd, $rs, imm	$\$rd \leftarrow \$rs \hookrightarrow imm$

Name	Mnemonic	Assembly Language	Semantics
Set If Equal	seq	seq $rd, $rs, $rt	$\$rd \leftarrow \$rs == \$rt?1:0$
Set If Equal Immediate	seq	seq $rd, $rs, imm	$\$rd \leftarrow \$rs == imm?1:0$
Set If Greater or Equal	sge	sge $rd, $rs, $rt	$\$rd \leftarrow \$rs \geq \$rt?1:0$
Set If Greater or Equal Immediate	sge	sge $rd, $rs, imm	$\$rd \leftarrow \$rs \geq imm?1:0$
Set If Greater	sgt	sgt $rd, $rs, $rt	$\$rd \leftarrow \$rs > \$rt?1:0$
Set If Greater Immediate	sgt	sgt $rd, $rs, imm	$\$rd \leftarrow \$rs > imm?1:0$
Set If Less or Equal	sle	sle $rd, $rs, $rt	$\$rd \leftarrow \$rs \leq \$rt?1:0$
Set If Less or Equal Immediate	sle	sle $rd, $rs, imm	$\$rd \leftarrow \$rs \leq imm?1:0$
Set If Less	slt	slt $rd, $rs, $rt	$\$rd \leftarrow \$rs < \$rt?1:0$
Set If Less Immediate	slt	slt $rd, $rs, imm	$\$rd \leftarrow \$rs < imm?1:0$
Set If Not Equal	sne	sne $rd, $rs, $rt	$\$rd \leftarrow \$rs \neq \$rt?1:0$
Set If Not Equal Immediate	sne	sne $rd, $rs, imm	$\$rd \leftarrow \$rs \neq imm?1:0$

Appendix: ASCII Character Set

Dec	Hex	Chr	Dec	Hex	Chr	Dec	Hex	Chr	Dec	Hex	Chr
0	0	null	53	35	5	78	4e	N	103	67	g
8	8	BS	54	36	6	79	4f	O	104	68	h
9	9	HT	55	37	7	80	50	P	105	69	i
10	a	LF	56	38	8	81	51	Q	106	6a	j
13	d	CR	57	39	9	82	52	R	107	6b	k
32	20	space	58	3a	:	83	53	S	108	6c	l
33	21	!	59	3b	;	84	54	T	109	6d	m
34	22	”	60	3c	<	85	55	U	110	6e	n
35	23	#	61	3d	=	86	56	V	111	6f	o
36	24	$	62	3e	>	87	57	W	112	70	p
37	25	%	63	3f	?	88	58	X	113	71	q
38	26	&	64	40	@	89	59	Y	114	72	r
39	27	′	65	41	A	90	5a	Z	115	73	s
40	28	(	66	42	B	91	5b	[	116	74	t
41	29	)	67	43	C	92	5c	\	117	75	u
42	2a	*	68	44	D	93	5d	]	118	76	v
43	2b	+	69	45	E	94	5e	ˆ	119	77	w
44	2c	,	70	46	F	95	5f	_	120	78	x
45	2d	-	71	47	G	96	60	‘	121	79	y
46	2e	.	72	48	H	97	61	a	122	7a	z
47	2f	/	73	49	I	98	62	b	123	7b	{
48	30	0	74	4a	J	99	63	c	124	7c	|
49	31	1	75	4b	K	100	64	d	125	7d	}
50	32	2	76	4c	L	101	65	e	126	7e	∼
51	33	3	77	4d	M	102	66	f	127	7f	DEL
52	34	4									

Bibliography

[1] Barry B. Brey. *The Intel Microprocessors*. Prentice Hall, 2000.

[2] Danny Cohen. On holy wars and a plea for peace. *IEEE Computer*, 14(10):48–54, October 1981.

[3] Douglas E. Comer. *Essentials of Computer Architecture*. Pearson Prentice Hall, 2005.

[4] M. Morris Mano, Charles R. Kime, and Tom Martin. *Logic and Computer Design Fundamentals*. Pearson, 2016.

[5] Karen Miller. *An Assembly Language Introduction to Computer Architecture*. Oxford Universtiy Press, 1999.

[6] Linda Null and Julia Lobur. *The Essentials of Computer Organization and Architecture*. Jones and Bartlett, 2015.

[7] David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann, 2014.

[8] David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface (ARM Edition)*. Morgan Kaufmann, 2017.

[9] John F. Wakerlyr. *Digital Design: Principles and Practices*. Pearson Prentice Hall, 2006.

Index