ENCM 369: Computer Organization
Lab9 - for the week of March 29, 2004

Updates / Corrections

If updates or corrections are needed they will appear in this space in the online version of this handout.

You may work in pairs on this assignment

You may complete this assignment individually or with one partner, *who must be in the same lab section as you*. Two students working together should hand in a *single* assignment with both names on the cover page.

Students working in pairs must make sure both partners understand all of the exercises being handed in.

Due Dates

The Due Date for this assignment is 1:30pm, Monday, April 5. The Late Due Date is 1:30pm, Tuesday, April 6.

The penalty for handing in an assignment after the Due Date but before the Late Due Date is 3 marks. In other words, X/Y becomes (X–3)/Y if the assignment is late. There will be no credit for assignments turned in after the Late Due Date; they will be returned unmarked.

Marking Scheme

A: 4 marks / B: 3 marks / C: 2 marks / D: 4 marks / total: 13 marks

Exercise A: Simple questions about pipelining

- Exercise 6.1 on page 529 of Patterson and Hennessy. Give a clear argument to support your answer.
- Using a drawing similar to Figure 6.8 on page 447, show the forwarding paths needed to execute the following five instructions:

```
subu    $s2, $t3, $t4
addu    $t7, $s2, $zero
addu    $t8, $s2, $t3
addu    $t9, $t8, $t8
sw      $t9, ($t8)
```

(This is Exercise 6.2 from the text, but with a different sequence of instructions.)

Exercise B: Branch (and jump) delays and load delays

Read This First

As explained in a lecture, real MIPS processors use features called *branch delay* and *load delay* to avoid certain pipeline hazards. For convenience, here is a quick summary of what was presented in the lecture:

- The instruction that follows a branch instruction is executed regardless of whether the branch is taken.
- The destination of a load instruction must not be used as a source operand in the instruction that follows the load instruction.

Here is a fact that might not have been made clear in the lecture: A jump instruction such as `j`, `jal` and `jr` is handled much like a branch: the instruction following the jump instruction is executed, *then* the jump target instruction is executed. For example, suppose `foo` is a C function that takes one `int` argument. A compiler might translate this C statement:

```
foo(7);
```

into this pair of instructions:

```
jal    foo
addiu  $a0, $zero, 7 # Executed *before* 1st insn of foo!
```

The `jal` instruction writes to `$ra` the address of the instruction after the `addiu` instruction.

What to do, Part I

Unlike with a branch instruction, there is no runtime decision to be made about whether or not to take a jump. Nevertheless it is useful for designers of pipelined hardware to have a policy of executing the instruction following a jump before executing the jump target. Explain why this policy helps keep a pipeline running at full speed.

What to do, Part II

Copy the files from `/local/courses/encm369/lab9/exB`

`ptr2ptr.spim` is a SPIM translation of `ptr2ptr.c`. Read both files, and make sure you understand why the SPIM code matches the C code. Then go through the following steps.

- Run the SPIM program with `xspim` in the usual way. Set a breakpoint on the instruction `addu $v0, $zero, $zero` near the end of `main` and check that `count_first_char` has returned the expected value to `main`.
- Quit `xspim` and start it up again, but this time with the command

```
xspim -delayed_branches -delayed_loads &
```

  This will give you a version of SPIM that simulates a MIPS processor with branch-delay and load-delay rules in effect. Run the program. Notice that there is an error message due to a load at an invalid address, and that `count_first_char` returns an incorrect value to `main`.
- Rewrite the assembly-language procedure `count_first_char` so that it would be correct on a machine with branch, jump and load delays. (The `main` procedure is already properly coded for such a machine.) As much as possible, avoid inserting `nop` instructions in delay slots.

Hand in a printout of your modified program.

Exercise C: Introduction to endianness

Read This First

Endianness is an important issue related to memory systems, but hasn't been discussed in lectures. Endianness refers to the organization of bytes within a memory word. Before explaining exactly what endianness is, I will review some points about bit indexing within bytes and words, and about addresses of bytes and words in memory. (It's clear from some of the questions I've received in labs and tutorials that not all students understand these points.)

A *byte* is an 8-bit chunk of data. (To be precise, eight bits is the near-universal size of a byte these days; some older computers had bytes of sizes of different sizes). Indexing of bits within a byte goes from 0 for the least significant bit to 7 for the most significant bit.

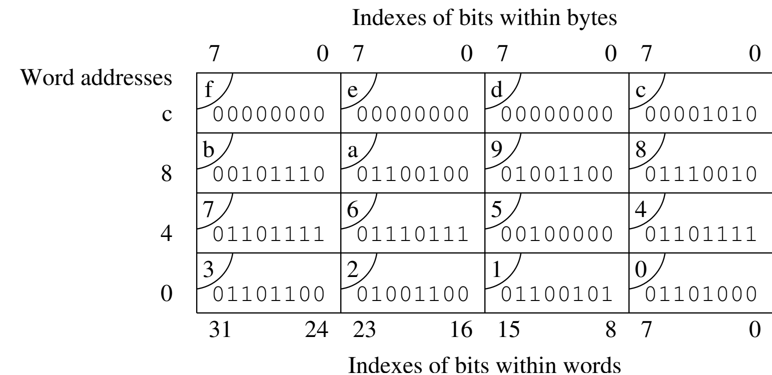

Figure 1: Little-endian memory organization. Addresses are in base sixteen; byte addresses are given in the upper left corner of each byte.

A *word* is a collection of bits that is typically larger than a byte. On MIPS systems, a word has 32 bits, but on other systems a word might have some other number of bits. Indexing of bits with in an n-bit word goes from 0 for the least significant bit to n–1 for the most significant bit. (These bit-indexing conventions are near-universal, and are used throughout your textbook and lecture notes.)

On MIPS and most other machines with 8-bit bytes and 32-bit words, memory can be accessed as a sequence of words or as a sequence of bytes (or as sequence of 16-bit *halfwords*, but this course pretty much ignores the existence of halfwords). Each memory word can also be accessed as a collection of four bytes. For example, the word at address `0x7fff_ff24` can also be viewed as four bytes at addresses `0x7fff_ff24`, `0x7fff_ff25`, `0x7fff_ff26`, and `0x7fff_ff27`. The issue of endianness is this: If a processor accesses a byte within a memory word, which bits within that word are being accessed? For example, if a MIPS processor stores a *byte* at address `0x7fff_ff24`, how does that affect the *word* at `0x7fff_ff24`? Obviously eight bits of the word are accessed, but which eight bits? Bits 31–24 of the word or bits 7–0 of the word? Or some other set of eight bits within the word?

It turns out there are two main schemes in use for ordering of bytes within words: little-endian and big-endian. In a *little-endian* system with 32-bit words, bits 31–0 of the word are stored in bytes as shown in Figure 1 on page 2. In a *big-endian* system with 32-bit words, bits 31–0 of the word are stored in bytes as shown in Figure 2 on page 3.

Consider the following sequence of MIPS instructions, and assume that `$a0`

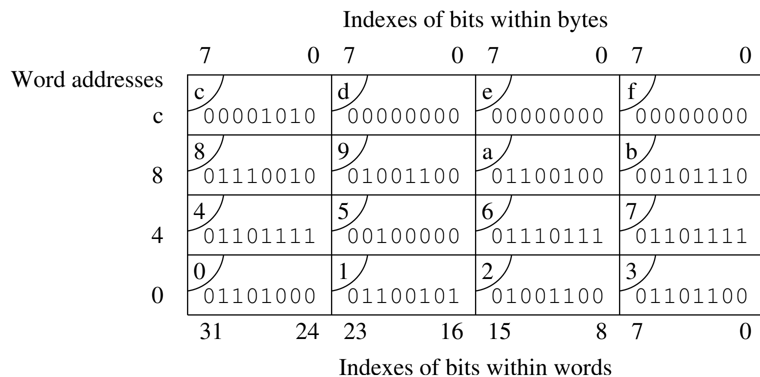

Figure 2: Big-endian memory organization. Addresses are in base sixteen; byte addresses are given in the upper left corner of each byte.

points to a word in the data segment of a program:

```
ori     $t0, $zero, 0x9
sb      $t0, 0($a0)
ori     $t0, $zero, 0xa
sb      $t0, 1($a0)
ori     $t0, $zero, 0xb
sb      $t0, 2($a0)
ori     $t0, $zero, 0xc
sb      $t0, 3($a0)
lw      $s0, 0($a0)
```

On a little-endian machine, the above sequence would put `0x0c0b0a09` in `$s0`, but on a big-endian machine, the sequence would put `0x090a0b0c` in `$s0`.

Here are some notes about endianness on various computer systems:

- x86 machines are little-endian.
- Machines that use the Motorola 68000 series of processors are big-endian.
- Apple Macintosh computers are big-endian.
- Machines based on the Sun SPARC processor are big-endian.
- The original MIPS processors were big-endian.
- SPIM's endianness is the same as the endianness of the machine on which it runs. So SPIM on an x86-based computer is little-endian, but SPIM on a Macintosh would be big-endian.
- Modern MIPS processors can work in either big-endian or little-endian mode.

What to do

Assume that `$a0` points to a word in the data segment of a big-endian MIPS-like computer. What will be in `$s0`–`$s3` just after the following instructions have been executed?

```
lui     $t0, 0x9f7d
ori     $t0, $t0, 0x65ea
sw      $t0, 0($a0)
lb      $s0, 1($a0)
lb      $s1, 0($a0)
lb      $s2, 3($a0)
lb      $s3, 2($a0)
```

Repeat the exercise, assuming this time that the machine is little-endian.

Exercise D: Endianness and binary files

Read This First

One area where endianness is of major concern to applications programmers is a situation where a *binary file* is written by one computer system and read by another.

Before introducing binary file operations, let's very briefly review text file operations. Consider a C program using `fprintf` to write an `int` to a *text file*. Suppose that the character set in use is ASCII, that `fp` is of type `FILE*` and has been opened for output, and that `i` is an `int`

```
i = 12345;
fprintf(fp, "%d\n", 12345);
```

The `fprintf` function converts the computer's internal representation of 12345 to a sequence of bytes: the ASCII code for `'1'`, the ASCII code for `'2'`, and so on. These five bytes followed by the code for `'\n'` are written to the file. So the effect of the function call is to put this sequence of bytes in the file:

```
00110001
00110010
```

```
00110011
00110100
00110101
00001010
```

Unlike a text file, a binary file is not necessarily a sequence of character codes. The C view of a binary file is a sequence of bytes that might be character codes, pieces of instructions, pieces of integers or pieces of floating point numbers, or that might have some other meaning. The key C library functions for binary file input and output are `fread` and `fwrite`. The following code demonstrates writing the bit pattern for an `int` to a binary file. Again suppose that `fp` is of type `FILE*` and has been opened for input:

```
i = 12345;
fwrite((void *) &i, sizeof(int), 1, fp);
```

The arguments are: the address of the first byte to be copied to the file; the size, in bytes, of the data items to be copied to the file; the number of data items to be written to the file; `fp`, which specifies the file. The 32-bit representation of 12345 is

```
0000_0000_0000_0000_0011_0000_0011_1001
```

`fwrite` simply copies a sequence of bytes from memory to a file. So on a 32-bit big-endian machine, the call to `fwrite` will put the following sequence of bytes in the file:

```
00000000
00000000
00110000
00111001
```

But on a little-endian machine the bytes would be written in this order:

```
00111001
00110000
00000000
00000000
```

Note that in both cases the bytes are totally different from what is produced by the call to `fprintf`.

The function to read from a binary file is `fread`.

It should be clear that endianness can cause problems if a binary file is written using `fwrite` on a machine with one endianness and read using `fread` on a machine with the opposite endianness.

What to do

Make a copy of the directory `/local/courses/encm369/lab9/exD`

There are two C programs in the directory. The program `binwrite.c` generates a small binary file with the following format: the first four bytes are an `unsigned int` giving the number of array elements stored in the file, and the remaining bytes are elements from an array of `unsigned int`s, four bytes per element. The program `binread.c` reads a file in the same format and displays the number of elements and array contents.

Read the two programs carefully to get an idea of how they work.

There are two data files in the directory: `x86_binwrite_output` is an output file produced by `binwrite.c` on a (little-endian) x86-based Linux system; `mac_binwrite_output` is an output file produced by `binwrite.c` on a (big-endian) Apple iBook running Mac OS X.

Build an executable from `binread.c`. Run it using `x86_binwrite_output` as the input file; then run it with `mac_binwrite_output` as the input file. The difference in behaviour will be obvious.

Make a copy of `binread.c` and modify it so that it can correctly read the data in `mac_binwrite_output`. Do *not* modify the calls to `fread`; instead use operations such as shifts and bitwise ands and ors to rearrange the bytes within variables.

Hand in a printout of your modified `binread.c` *file.*

Closing Remark

Storing numbers in binary files is somewhat more efficient than using text files, because binary files are smaller and because with binary files no time is spent converting base two numbers to sequences of base ten digits and vice versa.

However, endianness is only one of the many issues that can arise when reading and writing binary files on different platforms. Two of the many other issues are variations in the size of integer types (16-bit, 32-bit, or 64-bit?) and old, non-standard formats for floating-point numbers.