5

Memory Issues

Objectives

After completing this lab you will:

- have a better understanding of why memory alignment is required
- know what is the difference between Big Endian and Little Endian
- know how to use in MIPS assembly programs data of sizes other than word

Introduction

As long as the compiler generates code, the programmer may be completely unaware of the layout of data and instructions in memory. The two questions we address in this lab are:

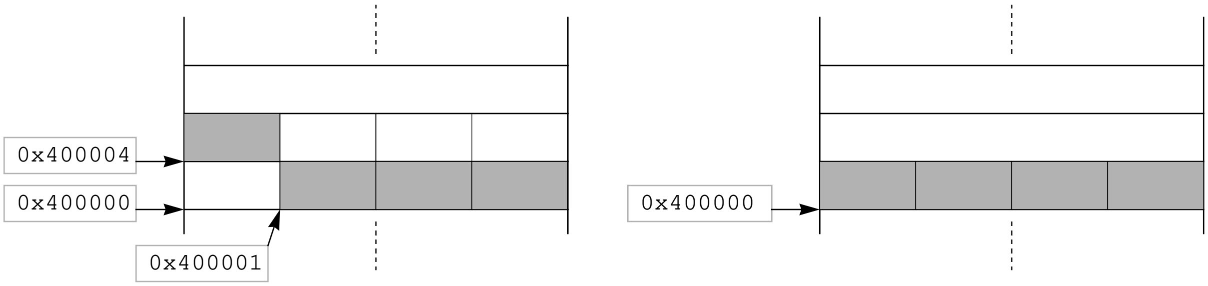
- Can data and instructions be stored in memory at any address? This is the memory alignment problem.
- When a data object larger than a byte is stored in memory, at what addresses are stored in the individual bytes of that object? This is the Big Endian versus Little Endian memory model problem.

Memory alignment

An object (data or instruction) with the size 2^n bytes is said to be aligned in memory if it is stored in at an address which is a multiple of 2^n. Since a multiple of 2^n number has the least significant n bits zero, we can also say that an object of size 2^n is memory aligned if it is stored at an address whose least significant n bits are zero.

Object	Size (bytes)	Store at address
byte	$1 = 2^0$	any address
half-word	$2 = 2^1$	multiple of 2
word	$4 = 2^2$	multiple of 4
double	$8 = 2^3$	multiple of 8

The alignment requirement for data and instructions in MIPS (as well as in other RISC architectures) is directly related to performance. An unaligned object in memory may require multiple memory accesses and/or special processing. As the next figure shows, an unaligned word in memory requires two memory accesses

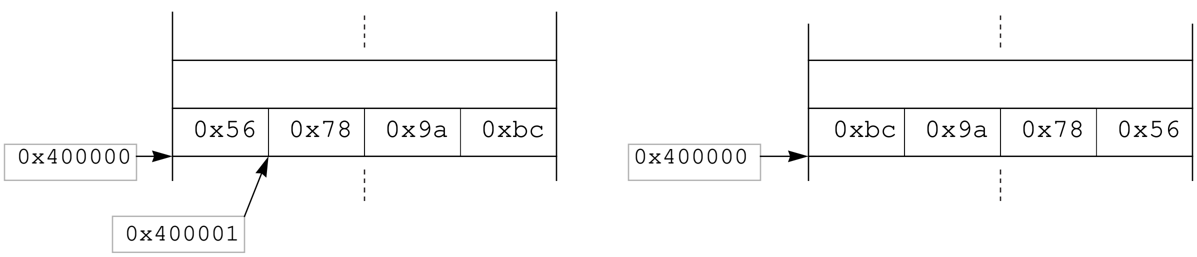


Unaligned word stored at address `0x400001` Aligned word stored at address `0x400000`

to be fetched and extra operations to be aligned in a register: the three bytes of the word read from address `0x400001` must be left shifted one byte, then the byte read from address `0x400004` merged on the least significant position. All this extra work would make the instruction execute slower.

Big Endian v. Little Endian

When data larger than a byte is stored in memory there are two possible layouts for the individual bytes as shown in the following figure where an integer (size = word) with the value `0x56789abc` is stored at address `0x400000`. If the most significant byte of the data object is stored at a smaller address than the least



Big Endian layout for the integer `0x56789abc` **Little Endian** layout for the integer `0x56789abc`

significant byte, then the layout is called Big Endian. If the most significant byte is stored at a higher address than the least significant byte, then the layout is called Little Endian.

MIPS CPUs can use either memory model. The specific model is selected at the reset time. The layout model can not change dynamically (while programs are running).

The SPIM simulator uses the memory model of the machine it is running on. If the simulator runs on a Little Endian machine then the memory model the user sees is Little Endian. If the simulator runs on a Big Endian machine then the memory model the user sees is Big Endian.

Laboratory 5: Prelab

Date ______________________ Section ______________________

Name ______________________

Memory Alignment

The MIPS assembler allows users to control data alignment using the `.align` directive. If the programmer does not use the directive, then the data will be automatically aligned in memory at the proper boundaries.

Since all instructions are the same size (word), they must be aligned in memory. Without alignment each instruction fetch would require two memory accesses, thus compromising the system's performance.

Q 1:

In MIPS each address is a *byte address*. Do you think the alignment requirement would exist if addresses were word addresses instead (each address would be the address of a word)? Give a justification for your answer.

Step 1

Start with the program P.1 below which you type and save as *lab5.1.asm*

P.1:

```
# this is a program used to test memory alignment for data
            .data 0x10000000
char1:      .byte 'a'          # reserve space for a byte
double1:    .double 1.1        # reserve space for a double
char2:      .byte 'b'          # b is 0x62 in ASCII
half1:      .half 0x8001       # reserve space for a half-word (2 bytes)
char3:      .byte 'c'          # c is 0x63 in ASCII
word1:      .word 0x56789abc   # reserve space for a word
char4:      .byte 'd'          # d is 0x64 in ASCII

            .text
            .globl main
main:       jr $ra             # return from main
```

Step 2

Load *lab5.1.asm* in the simulator and look in the data segment to see where data has been stored. Fill out the following table. In the 'Multiple of' column of the table indicate what power of two the address is a multiple of. In the 'Displacement' column' indicate what is the displacement of the data object from the beginning of the data segment.

Name	Data size (bytes)	Address	Multiple of	Displacement (bytes)
char1				
double1				
char2				
half1				
char3				
word1				
char4				

Q 2:

Based on the way data has been stored in the data segment, do you think data is aligned or not in memory? Explain why.

Q 3:

How many bytes have been wasted due to alignment?

wasted_bytes =

Step 3

Modify *lab5.1.asm* as to minimize the number of bytes wasted due to alignment. Save your work as *lab5.2.asm*.

Hint: change the order of data declarations.

Q 4:

How many bytes are wasted due to alignment in program *lab5.2.asm*?

wasted_bytes =

Step 4

Modify *lab5.1.asm* as follows:

- include the directive `.align 0` right before 'char1'
- in main load *word1* from memory in register **`$t0`**

Save this program as *lab5.3.asm*.

Step 5

Load *lab5.3.asm* in the simulator and look in the data segment to see where data has been stored. Fill out the following table. In the 'Multiple of' column of the table indicate what power of two the address is a multiple

Name	Data size (bytes)	Address	Multiple of	Displacement (bytes)
char1				
double1				
char2				
half1				
char3				
word1				
char4				

of. In the 'Displacement' column' indicate what is the displacement of the data object from the beginning of the data segment.

Q 5:

Based on the way data has been stored in the data segment, do you think data is aligned or not in memory?

Explain why.

Q 6:

How many bytes have been wasted due to alignment?

wasted_bytes =

Step 6

Run *lab5.3.asm*. You will get an error message. Write it down.

Q 7:

Why does the simulator report an error?

Step 7

Modify *lab5.1.asm* as indicated below.

- start the text segment at address `0x400001` instead of the default value.

Save the new program as *lab5.4.asm*.

Step 8

Load *lab5.4.asm*. Write down the first error message you obtain

Q 8:

Why does the simulator report an error?

Laboratory 5: Inlab

Date ______________ Section ______________

Name ______________

Big Endian v. Little Endian

Most of the time the programmer does not see the details of memory layout for data. Most of the time it is irrelevant whether the data (or instructions for that matter) is stored in Big or Little Endian format. But you do not need to do assembly programming to become aware of these details. Playing with pointers makes the issue visible very quickly.

Ex 1:

```
{
int w=0x4255664c;              // an integer is of size word
char *p;                       // pointer to byte

      p = (char*)&w;           // p now points to where w starts in memory
      cout << *p << endl;      // prints the byte pointed to by p
}
```

The last statement in this piece of code will print out the byte stored in memory at the address where the storage for `w` begins. If the memory model is Big Endian, then this byte is the most significant byte of `w` (`0x42`), and the program prints the letter `B` (`0x42` is the ASCII representation for `B`). Otherwise, it is the least significant byte of `w` (`0x4c`) and the program prints the letter `L`. ■

The MIPS architecture provides instructions that can be used to load and store data of other sizes than the native size (word). The next table presents the loads. Corresponding store instructions exist for the signed ver-

Instruction	Effect	Comment
`lb Rdest, disp(Rbase)`	Rdest $\leftarrow$ `M[Rbase + disp]`$_8$	load byte and sign extend it
`lbu Rdest, disp(Rbase)`	Rdest $\leftarrow$ `M[Rbase + disp]`$_8$	load byte
`lh Rdest, disp(Rbase)`	Rdest $\leftarrow$ `M[Rbase + disp]`$_{16}$	load half-word and sign extend it
`lhu Rdest, disp(Rbase)`	Rdest $\leftarrow$ `M[Rbase + disp]`$_{16}$	load half-word

sions of these loads.

Remember that sign-extending a datum means replicating the most significant bit of the datum until it reaches the required size. Sign-extending a byte to fit a register (32 bits) means replicating 24 times the most significant bit of the byte.

Step 1

Sign extend to 32 bits the the following two bytes

																								0	1	1	0	1	1	1	1

																								1	1	1	0	1	1	1	1

Step 2

Start with *lab5.1.asm* and create a new program named *lab5.5.asm* based on the following description.

- declare a new variable called *word2* of size word, with the initial value 0
- load (use `lb`) the four bytes of *word1* in successive registers starting with **`$t0`**
- load (use `lbu`) the four bytes of *word1* in successive registers starting with **`$t4`**
- load (use `lh`) *half1* in **`$t8`**
- load (use `lhu`) *half1* in **`$t9`**
- store the four bytes into *word2* in reverse order: the byte that was the most significant in *word1* will be the least significant byte in *word2*, and so on.

Step 3

Load the program and run it. Fill out the following table

Register	Content
`$t0`	
`$t1`	
`$t2`	
`$t3`	
`$t4`	
`$t5`	
`$t6`	
`$t7`	
`$t8`	
`$t9`	

Variable	Expected value	Value in memory
word2		

Q 1:

Why are **`$t2`** and **`$t6`** different even if they have been loaded with data from the same address?

Q 2:

Based on the content of registers can you decide whether the SPIM simulator uses a Big Endian or Little Endian memory model? Which one?

Step 4

Use *lab4.2.asm* as a template to create *lab5.6.asm* using the following description.

- prompts the user to enter an integer; store the integer in memory in a variable called *user1*
- calls a procedure named ‘Reverse_bytes’. The argument passed to the procedure is the address of the word whose bytes must be reversed. The most significant byte will become the least significant and so on.
- prints a message that reads “If bytes were layed in reverse order the number would be: ”
- prints the number whose bytes have been reversed

Run the program using the next test plan. Enter the last four digits of your SSN in the bolded row of the table.

Test plan for *lab5.6.asm*

Integer	Output
0	0
-1	-1
1	16777216
16777216	

Laboratory 5: Postlab

Date ______________________ Section ______________________

Name ______________________

Handling Unaligned Data in MIPS

The MIPS architecture provides instructions that can be used to handle unaligned data. Loading an unaligned word from memory will require at least two instructions, one to get the upper part of the data and another one for the lower part of the same data.

Using the special instructions in the instruction set will result in programs that work without generating alignment related errors. These programs will be slower than their counterparts in which data is aligned in memory at the proper boundaries.

Instruction alignment is a must. The programmer may choose whether data is aligned or not but there is no choice for instructions.

Q 1:

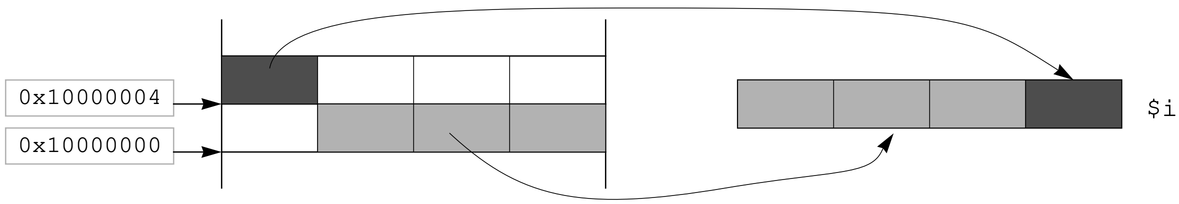
Can you see any advantage in working with unaligned data?

Aside from the fact that these instructions must generate no error, they must also place the data of interest in the proper place in the destination register.

In the figure below the most significant three bytes of a word are stored beginning with address `0x10000001`, while the least significant byte of the same word is stored at `0x10000004`. When loaded in a register, the most significant three bytes of the word must end up on the most significant three bytes in the register, while the least significant byte of the word (now stored on the most significant position of the word beginning at `0x10000004`) should be merged with the other bytes on the least significant byte of the register.

There are two instructions used to do the job, `lwl` and `lwr`. Similar instructions exist for dealing with half-words.

`lwl` will read from a memory address (possibly unaligned) and will place the bytes starting with the current address up to the next aligned word address in the upper bytes of the destination register. The next example



An unaligned word stored in memory at address `0x10000001`

The same word loaded in register `$i`

will clarify how the instruction works.

Ex 1:

```
lui $t0, 0x1000    # $t0 <- 0x10000000
lwl $t1, 1($t0)    # read from address 0x10000001
```

The three bytes stored at addresses `0x10000001`, `0x10000002`, and `0x10000003` will be loaded in register **`$t1`** on positions 3, 2, and 2 respectively (3 is the most significant byte, 2 is the byte next to it and so on). ■

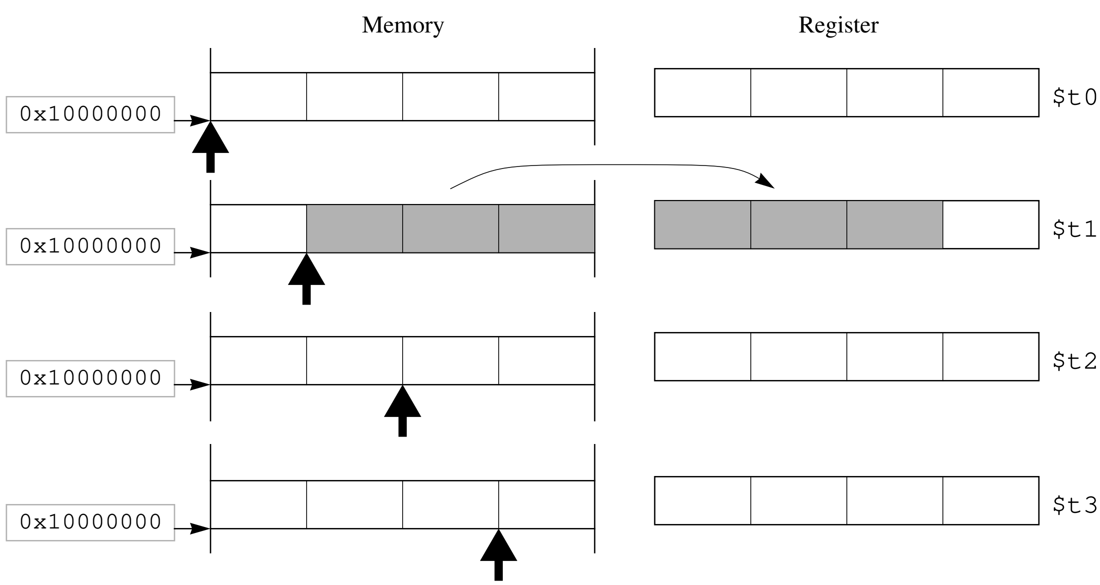
Your job is to understand how the other instruction (`lwr`) works and then use the two instructions together to load unaligned words from memory.

Step 1

Create the program *lab5.7.asm* which does

- declare a word variable named *word1* with the initial value `0x89abcdef`
- loads (using `lwl`) the registers **`$t0`** to **`$t3`** with data from memory, using as base address the address of *word1* and displacements from 0 to 3 respectively.

Run the program. Based on the values stored in registers fill the missing spaces in the following figure. The bold vertical arrow indicates the address `lwl` reads from. Use slanted lines to indicates which bytes are transferred in the register and on what position(s) in the register.

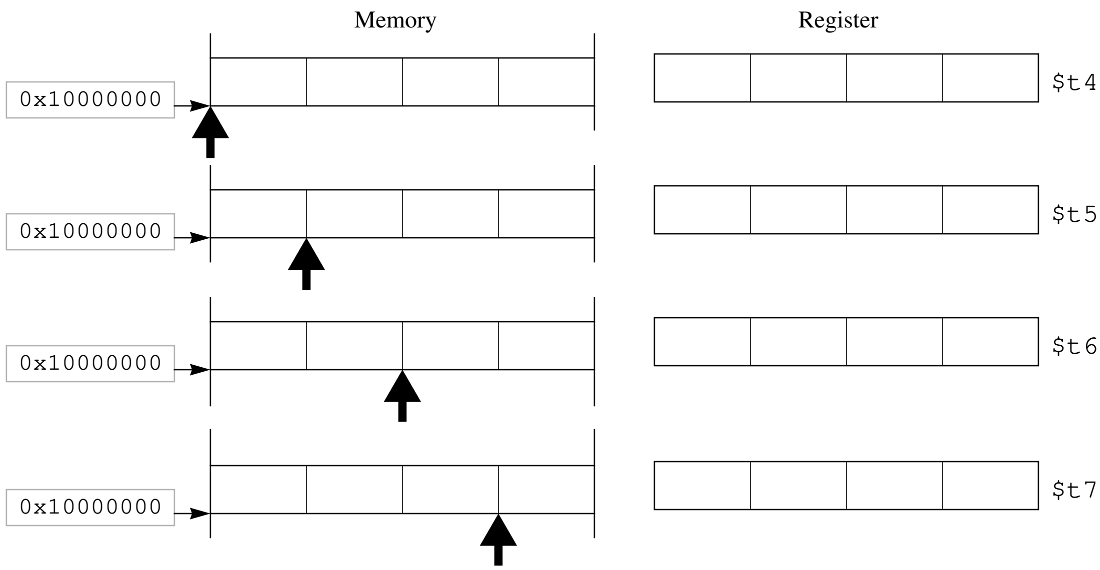


Step 2

Create the program *lab5.8.asm* which does

- declare a word variable named *word1* with the initial value `0x89abcdef`
- loads (using `lwr`) the registers **`$t4`** to **`$t7`** with data from memory, using as base address the address of *word1* and displacements from 0 to 3 respectively.

Run the program. Based on the values stored in registers fill the missing spaces in the following figure. The bold vertical arrow indicates the address `lwr` reads from. Use slanted lines to indicates which bytes are transferred in the register and on what position(s) in the register.



Step 3

When you run *lab5.3.asm* you get an error message, due to the fact that you attempt to load an unaligned word using the `lw` instruction. Modify that program and create a new one, named *lab5.9.asm*, which fixes the problem. Use only native instructions.

Hint: use a sequence of `lwl` and `lwr` instructions with the appropriate displacements.

Step 4

It is now time to use some of the store instructions for unaligned data.

Create the program *lab5.10.asm* with the following description.

- declares the unaligned variables (in this order) *ch1* of size byte, *word1* of size word, *ch2* of size byte and *word2* of size word
- the initial values of variables are ‘a’, 0x89abcdef, ‘b’ and 0 respectively
- copies the value of *word1* into *word2*
- uses only native instructions

Start the SPIM simulator using the `-bare` command line option. Load the program and run it. Look into the memory and make sure the program has changed the value of *word2* from 0 to 0x89abcdef.

Step 5

You want to evaluate the impact of unaligned data on the overall performance of some application.

If the data were aligned in the memory, then each memory access would be just one instruction. If data is not aligned, then several instructions are needed for each load or store.

Extra instructions needed to load an unaligned word =
Extra instructions needed to store an unaligned word =

Let's also assume that the overall CPI for the application does not change and that the clock cycle is the same in both cases. The next table gives the frequency of loads and stores for various data sizes in the case data is aligned. For the sake of this problem we ignore any data of size other than byte and word.

Instruction	Frequency
`lw`	9%
`sw`	4%
`lb`	2%
`sb`	0.9%

Q 2:

By how much faster is the application when data is aligned than the same application with unaligned data? Show your work.

Step 6

Return to your lab instructor copies of *lab5.7.asm* to *lab5.10.asm* together with this postlab description. Ask your lab instructor whether copies of programs must be on paper (hardcopy), e-mail or both.