Foothill College
Computer Science Department
CS10 Computer Architecture and Organization
Lab # 3

MIPS Instruction Set Architecture and Data Representation

1 MIPS Data Types

The MIPS CPU provides us with a lot of flexibility in terms of integer data types. Not only are there four sizes:

- 8-bit bytes,
- 16-bit halfwords,
- 32-bit words, and
- 64-bit doublewords

but MIPS can treat any value as either *signed* (i.e. twos-complement) or *unsigned*. Unsigned means that there are no negative numbers, and the most-significant bit is just another power of two to add to make a positive value.

For example:

Signed byte: $–53_{10}$							
-ve							
1	0	1	1	0	1	0	1

Unsigned byte: 128+64+16+4= **212_{10}**							
128	**64**	**32**	**16**	**8**	**4**	**2**	**1**
1	1	0	1	0	1	0	0

It's worth remembering the ranges of the 8 possible MIPS integer types:

	Signed	**Unsigned**
Byte	-128 to +127	0 to 255
Halfword	-32768 to +32767	0 to 65535
Word	-2147483648 to +2147483647	0 to 4294967296
Doubleword	-verybig to +verybig	0 to verybig

1.1 Endianness

For our purposes, the MIPS architecture is *little-endian*. This means that, for multi-byte data types such as halfwords, words and doublewords, the bytes which hold the most significant bits come **last**. Let's take the decimal number 257229004_{10}, which is the 32-bit binary pattern 00001111 01010101 00000000 11001100. For demonstration, a pattern has been chosen where each group of 8 bits is different. If this word is stored by the MIPS CPU in main memory at locations 0 to 3 (memory locations are byte-sized), then the number will be stored as:

Location 0	Location 1	Location 2	Location 3
11001100	00000000	01010101	00001111

Or, if we convert into hexadecimal, the 32-bit pattern 0xF5500CC will be stored as 0xCC, 0x00, 0x50, 0xF5, i.e. the *little end* of the value comes *first* in memory.

1.2 Data Alignment

The MIPS architecture requires that:

- halfword (2-byte) values are stored starting at addresses which are multiples of 2, e.g. 0, 2, 4, 6, 8.
- word (4-byte) values are stored starting at addresses which are multiples of 4, e.g. 0, 4, 8, 12, 16.
- doubleword (8-byte) values are stored starting at addresses which are multiples of 8, e.g. 0, 8, 16, 24.

This is known as *data alignment*. You have to be mindful of this, as you can end up with some wasted memory. Consider the following assembly language program which stores several numbers in memory.

```
        .data
num1:   .byte    5       # Store 5 as a byte
num2:   .half    6       # Store 6 as a halfword, i.e. 16 bits
num3:   .byte    7       # Store 7 as a byte
num4:   .word    8       # Store 7 as a word
```

Let's assume that the assembler stores these numbers consecutively, starting at address 0. Here's how the memory would be allocated:

Location	0	1	2	3	4	5	6	7	8	9	10	11
Value	num1	unused	num2	num2	num3	unused	unused	unused	num4	num4	num4	num4

See that *num3* which is a byte can be stored at location 4, but *num4* cannot be stored at location 5 onwards, as 5 is not a multiple of 4. We have to store *num4* starting at location 8 onwards. Likewise, *num2* has to start at location 2.

2 Arithmetic Operations on MIPS Registers

Internally, the MIPS registers are all 32 bits in size, so all the arithmetic and logical operations performed by the MIPS CPU are done on 32 bit values. However, we can choose to perform *signed* or *unsigned* arithmetic operations. Here are some of the arithmetic operations:

Instruction	Result	Comment
`li Rd, Imm`	Rd = Imm	Load register with constant value
`add Rd, Rs, Rt`	Rd= Rs + Rt	Signed addition of 2 registers
`addi Rd, Rs, Imm`	Rd= Rs + Imm	Signed addition of Rs with 16-bit constant
`addu Rd, Rs, Rt`	Rd= Rs + Rt	Unsigned addition of 2 registers
`addiu Rd, Rs, Imm`	Rd= Rs + Imm	Unsigned addition of Rs with 16-bit constant
`sub Rd, Rs, Rt`	Rd= Rs - Rt	Signed subtraction of 2 registers
`subu Rd, Rs, Rt`	Rd= Rs - Rt	Unsigned subtraction of 2 registers
`sll Rd, Rs, shamt`	Rd = Rs << shamt	Shift Rs left by shamt bits, store in Rd
`srl Rd, Rs, shamt`	Rd = Rs >> shamt	Shift Rs right by shamt bits, store in Rd
`sra Rd, Rs, shamt`	Rd = Rs >> shamt	Shift signed Rs right by shamt bits, store in Rd

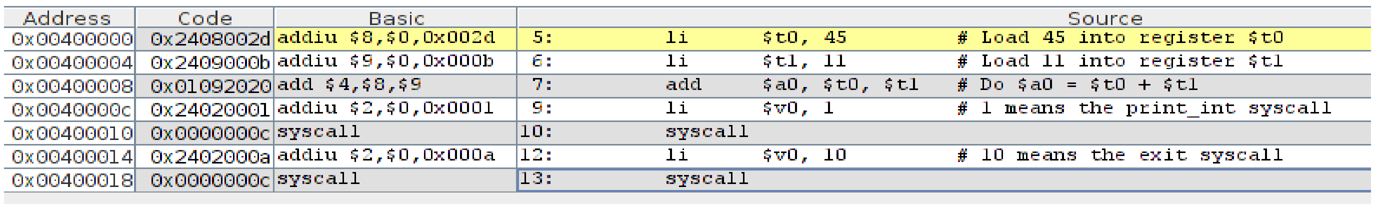
2.1 Task 1

Download this assemby program: **w3add.asm**. Open up the MARS simulator and load the program. The code loads 45 into register $t0 and 11 into $t1, add them and stores the result into register $a0. The program then calls the *print_int* system call to print the result out.

Assemble the program and switch over to Execute view. Single-step the instructions using the ▶1 icon, and watch as the two constants are loaded into $t0 and $t1, then as the addition is stored into $a0.

2.2 Pseudo-instructions

In the Execute view, look at the Text Segment sub-window: you should see this:



Address	Code	Basic	Source			
0x00400000	0x2408002d	addiu $8,$0,0x002d	5:	li	$t0, 45	# Load 45 into register $t0
0x00400004	0x2409000b	addiu $9,$0,0x000b	6:	li	$t1, 11	# Load 11 into register $t1
0x00400008	0x01092020	add $4,$8,$9	7:	add	$a0, $t0, $t1	# Do $a0 = $t0 + $t1
0x0040000c	0x24020001	addiu $2,$0,0x0001	9:	li	$v0, 1	# 1 means the print_int syscall
0x00400010	0x0000000c	syscall	10:	syscall		
0x00400014	0x2402000a	addiu $2,$0,0x000a	12:	li	$v0, 10	# 10 means the exit syscall
0x00400018	0x0000000c	syscall	13:	syscall		

The assembler has stored the program's code in memory starting at address 0x00400000. The Code column shows the actual machine code for each instruction in hexadecimal. Note also that the *li* (load immediate) instruction is not a real instruction: it's a *pseudo-instruction*. In this case, `li $t0, 45` has been converted by the assembler to `addiu $t0, $0, 45`. Why are these equivalent? Because $t0 = 45 is also the same as $t0 = 0 + 45. Remember, the $0 register always contains the value zero!

2.3 Task 2

Write your own **.asm** program to perform the following addition. Use the $t0 to $t7 registers, calculate the following sum into the $a0 register, and print out the answer:

400 + 17

3 Data Overflows

We saw above that twos-complement 32-bit words have a range from around -2 billion up to +2 billion. What happens if we try to add two numbers together to produce a result which is bigger than 2 billion?

3.1 Task 3

Reload the original ***w3add.asm*** program. Modify the code to load 2 billion (2 followed by 9 zeroes) into both $t0 and $t1. Now run or single-step the program. You should see very large hex values being stored in $t0 and $t1. When the simulator hits the add instruction, however, it crashes with this message:

```
Error in w3add.asm: Runtime exception at 0x00400010: arithmetic overflow
Go: execution terminated with errors.
```

With signed instructions like *add*, if the result exceeds either the maximum or minimum of the signed range, the CPU throws an exception and the program crashes.

3.2 Task 4

Keep the loading of 2 billion into the two registers, but this time change the *add* instruction to an *addu* (unsigned add) instruction. This tells the CPU to treat both registers as holding unsigned values. The result will be 4 billion, which is well inside the range for an unsigned 32-bit number.

The program should run and print out a result. Question: why was the result not 4000000000 but some negative number? Hint: the *print_int* system call expects that the value to print is a signed value.

4 Loading from Memory, Storing to Memory

So far, all of our data has been stored and manipulated in the MIPS registers. But with only 32 registers, that's not enough to hold large amounts of data: consider writing a program which needs an array of 5,000 integers.

To rectify this, we now look at the instructions available to load data from memory into registers, and to store data into memory from registers.

Instruction	Result	Comment

`lb Rd, label`	Rd = byte value at label	Sign extended
`lbu Rd, label`	Rd = byte value at label	Unsigned
`lh Rd, label`	Rd = halfword value at label	Sign extended
`lhu Rd, label`	Rd = halfword value at label	Unsigned
`lw Rd, label`	Rd = word value at label	
`ld Rd, label`	(Rd, Rd+1) = doubleword at label	
`sb Rd, label`	byte value at label = Rd	
`sh Rd, label`	halfword value at label = Rd	
`sw Rd, label`	word value at label = Rd	
`sd Rd, label`	doubleword at label = (Rd, Rd+1)	
`move Rd, Rs`	Rd = Rs	

Some notes to go with this table:

- The label is a labeled location in the `.data` section of your program
- For data sizes smaller than a word (byte, halfword), you have to option to load the value into a register, and either fill the missing bits with all zeroes (unsigned), or to replicate the sign bit from the value to preserve it (signed). The latter is known as *sign extension*.
- For doubleword values, the value is loaded into/stored from a pair of adjacent registers.

4.1 Task 5

Load the following program and for now just look at the source: **w3loadstore.asm**. The program asks the assembler to fill memory locations with these data sizes and values:

```
    num1:   .word   7              # Store 7 as a word into num1 location
    num2:   .word   6              # num2 = 6
    num3:   .half   -10            # Store -10 into 16-bit location num3
    num4:   .half   3              # num4 = 3
    num5:   .byte   64             # num5 is only 1 byte long
    num6:   .byte   1
```

Note that things have been put in decreasing size order: this ensures that we don't waste memory due to alignment issues.

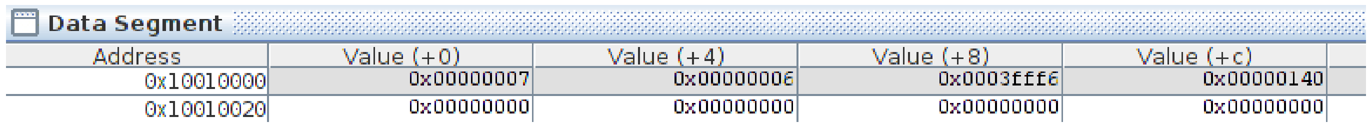
Each value is only small and would fit into a single byte: for multibyte data sizes, the remaining bytes should be filled with zeros (or ones for the -10 value). Assuming that the assembler stores these adjacent values in memory starting at location 0, predict which bytes will be all zeroes, all ones, or have the bit pattern of the values shown. Remember that multibyte data is stored in **little-endian** format! Fill in the following table (on paper) with your guesses.

Location	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**

4.2 Task 6

Now assemble the program, switch to Execute view, and look at the Labels and Data Segment sub-windows. The assembler has actually stored these values starting at location 0x10010000. You should see this layout of data:

Label	Address
main	0x00400000
num1	0x10010000
num2	0x10010004
num3	0x10010008
num 4	0x1001000a
num 5	0x1001000c
num 6	0x1001000d


Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)
0x10010000	0x00000007	0x00000006	0x0003fff6	0x00000140
0x10010020	0x00000000	0x00000000	0x00000000	0x00000000

You can see that *num1* and *num2* take up 4 bytes each,*num3* and *num4* 2 bytes each, and *num5* is only 1 byte. We can assume that *num6* is 1 byte, but we need to know what comes next to be completely sure.

Your answer to the data layout question should be:

Location	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
	07	00	00	00	06	00	00	00	F6	FF	03	00	40	01	00	00
Value	num1				num2				num3		num4		num5	num6		

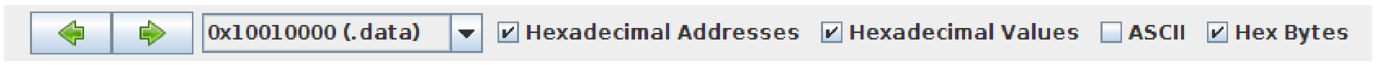
Unfortunately, the Data Segment view shows memory as groups of 4 bytes, treated as little-endian words. This means:

- the first 4 bytes 07 00 00 00 are flipped around and we see them as 0x00000007: no problem, that's *num1*.
- the next 4 bytes 06 00 00 00 are flipped around and we see them as 0x00000006: no problem, that's *num2*.
- the next 4 bytes F6 FF 03 00 are flipped around and we see them as 0x0003fff6: it looks like *num4* comes before *num3*.
- the last 4 bytes 40 01 00 00 are flipped around and we see them as 0x00000140: seems like zeroes comes before *num6* then *num5*.

This is all an artifact of the way the MARS simulator displays groups of 4 bytes as little-endian words. If MARS had a byte-oriented display of memory, you would see the little-endian layout of the words and halfwords properly.

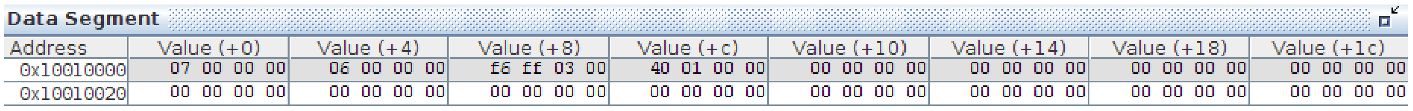
Update

You can modify Mars to show the bytes in memory in the actual order that they appear. Under the memory display you should see these controls:



0x10010000 (.data) [✓ Hexadecimal Addresses] [✓ Hexadecimal Values] ASCII [✓ Hex Bytes]

If you enable the "Hex Bytes" option, then the display of the memory will look like this:



Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	07 00 00 00	06 00 00 00	f6 ff 03 00	40 01 00 00	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
0x10010020	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00

Now you can see that the value 7_{10}at location 0x10010000 is really stored as the 4 bytes 07 00 00 00, and the same for the following words, halfwords and byte values.

4.3 Task 7

Now run the program, see that it loads the data items from memory into registers, peforms the calculation, saves the result back to memory, and the copies the result into the $a0 register for printing. Play around with the load and store operations, and see what they do.

4.4 Task 8

Write your own **.asm** program to perform the command:

num1 = num1 + num2

In the data segment store 400 as a word into the *num1* location; store 17 as a word into *num2* location. Print out the answer.

5 Outlook for the Next Lab

In the next lab, we will look at MIPS addressing modes, and the instructions which change the flow of control and allow you to make decisions and do loops.