

Lab: 0 MIPS Assembly File Overview

- 1) Copy the assembly language source file *intro.asm* from Moodle – this is just a text file. Using any text editor (Notepad, etc.) open this file and review it as directed below:

Note the use of horizontal ‘columns’ and vertical spacing for readability.

Use this format when doing your own future work

Column 1: labels, keywords, and full line comments

Column 2: assembly code aligns here

Column 3: short, end of line comments if needed

1a) Heading comment (#) lines provide some commentary as to author, date, and the purpose of this code.

1b) Assembler Keywords and meanings:

#	Start of a comment – comment runs until end of line.
.text	keyword that specifies that subsequent items are instructions. Text segments are created and arranged by the assembler, although an optional <address> can set the starting address of a segment.
.globl sym	declares the label <i>sym</i> as a global that can be referenced from other files. The __start label is used by the simulator to indicate the start of instructions (see below)
__start:	“double-underscore start” Special system label that marks the start of the program. <i>User-defined labels should not use __ prefix</i>

- 1c) The first line of assembly code is listed in the middle column

ori \$t1, \$zero, 15

...

This one line does a “bitwise” OR operation, *putting the result into register \$t1*

What is the expected result of 0 OR 15 ? Write your answer here _____

- 2) Near the bottom of the file you will find these keywords:

.data	specifies that subsequent items belong in the optional data segment. This is where your program data would go – if you have constants or strings, etc
.word w ₁ , w ₂ , ... w _n	Keyword to declare data storage values as words (each w _n is 32-bit value) Stores them in successive memory (word) locations
vals:	“vals” is a user-defined label declared at the current memory location (<i>you could think of it as a variable name</i>)
t:	another variable (also see x:)
.asciiz "string"	Keyword to declare storage area for ASCII strings. Stores the quoted string in memory <i>and null terminates it.</i> (i.e. it adds a zero valued byte)
.byte	Keyword to declare storage values as bytes. Stores a series of byte values in successive memory (byte) locations
.ascii "String"	(<i>not used in this lab</i>) Same as above without the null termination
.space n	(<i>not used in this lab</i>) Sets aside <i>n</i> bytes of space in memory

- 3) MARS –FU is the Furman specific version of a MIPS processor simulation tool called MARS. **Our version should be the only one you use!** It contains a built-in editor, assembler, and run-time simulator for the MIPS microprocessor.


The MARS-FU Simulator

- 4) Close your text editor. We will use the MARS-FU simulator to edit and run our programs.

MARS-FU is a simulator and a text editor.

Open MARS-FU and use *File > Open* to load the *intro.asm* file.

Click the Edit tab (just below tool bar). You will see the editor with color-coded elements for an assembly program.

- 5) Click the toolbar wrench & screwdriver  button – this assembles the program into machine code. The view now shifts to the Execute environment.

Familiarize yourself with the parts of the **Execution environment**:

Notice the main work areas: Text Segment, Registers, and Data Segment windows should be visible. You will also have a toolbar and a “Mars Messages” panel. *(other panels may be present but are not relevant now)*


- The **Text Segment** contains the assembled program: Note the left column marked *Address* is the memory address of each instruction; the next column is the raw machine *Code* for the instruction (this is also in hex); the middle area shows the *Basic* mnemonics; and the rightmost area shows the original *Source code as written*: the programmer’s assembly code with mnemonics and comments; the leftmost number in the Source column indicates the line number from your source file.

Answer the following ‘Q’ questions and enter the answers **into Moodle**

Q1) From the assembled code, find the **machine code value** for the

`ori $t1,$zero, 15` (in hex notation) 0x3409000f

- The **Registers** window (right side) shows all processor register values at the current moment. ALL VALUES ARE HEX. The 32 general purpose registers should be visible in addition to a few special purpose registers.

Q2) Use single-step execution once ( or F7), to execute ONLY the first ori (OR immediate) instruction. What is the value of the \$t1 register now? In hex 0x0000000f

- The **Data Segment** window shows data elements as they actually exist in memory. The leftmost column indicates the base memory Address for that row of data. The columns in the middle are the *words* of data with that base address plus an offset from the base address (remember, word offset is +0, +4, +8, etc.)
- At the bottom the console window **Mars Messages**, will show assembly errors as well as user I/O as needed


Q3) Use single-step execution once more, to execute the second ori instruction. Which register value has changed now? \$t2

Predict what will happen in the **and** instruction (predict ALL changes)

- a. After making the prediction, single-step over that instruction and confirm your prediction – or figure out why it was wrong

Q4) Predict what will happen in the xor instruction (predict ALL changes)

After making the prediction, single-step over that instruction and confirm your prediction – or figure out why it was wrong. What is the value in \$t0 now? In hex 0x00000006

Q5) Click the run button  and describe the output displayed in the console window (what value is output?) 6

Q6) Answer question 6 too

**Make sure that EACH PARTNER enters answers 1 – 6 into their Moodle:
see “LAB QUESTIONS”**

Data segment

In the data segment panel, memory contents are stored as words in a grid.

The left-most column is the starting address (note the 32-bit hex value like: 0x1001000)

Each of the “Value” columns is the value at the next word in memory

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0xfafafafa	0x00000000	0xfbfbfbfb	0x0656e6f	0xfcfcfcfc	0x6e656874	0x61676120	0x00006e69
0x10010020	0xfdfdfdfd	0x65646e69	0x09202078	0x61684300	0x74612072	0x656c0009	0x6874676e	0x0a000920
0x10010040	0x43415453	0x726f204b	0x56415320	0x65722045	0x48202e67	0x42205341	0x204e4545	0x4e414843
0x10010060	0x21444547	0x52524520	0x3d3d524f	0x3d3d3d3d	0x3d3d3d3d	0x3d3d3d3d	0x000a3d3d	0x0a0a0a0a
0x10010080	0x3d3d3d3d	0x3d3d3d3d	0x203d3d3d	0x5054554f	0x3d205455	0x3d3d3d3d	0x000a3d3d	0x00000000
0x100100a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

Check back to the original source file, by clicking the Edit tab – note the data section starts with several words: 9,10,11,12...

Return to the Execute environment and find each of those words in memory.

What is the address of the value 9? _____

What is the address of the value 12? (remember, memory will show the HEX value of 12) _____

From **the image of data** above – what value is stored in address 0x10010084? _____ 0x3d3d3d3d

Checking back with the MARS source code, notice we have defined an ASCII string “hello – world\n”. This string is also “null terminated”, meaning it has a ZERO valued byte at the end.

Returning to the Execution environment, can you find the address of the “h” (refer to the ASCII chart) – *remember, this character is a byte, not a word.*

Now find the “e” character...what address is it? (Hint: one more than the ‘h’) _____

Now find the two “l” characters of “hello”? What about the “o”?

What’s going on?

The displayed words contain 4 bytes – four bytes with addresses 0x1000002c, 0x1000002d 0x1000002e 0x1000002f. But which byte goes in address xxx02c and which goes in xxx02f???

The answer depends on the computer that you are running on. Some computers put the “Big end” first; while others put the “little end” first. Our computers put the little end in the lowest address – called “little endian” machines. To confirm this, look at the word value for 9 from the previous questions – there are 4 bytes at addresses ending in 00,01,02,03. The “9” value fits in the *byte* at address 0x10000000. *Viewing* this word *as a word* looks fine. However, viewing strings (bytes) as a word looks “mixed up”.

Endian-ness affects us in two ways

- 1) if you are writing low-level networking code to send data one byte at a time — everyone needs to agree on which "end" comes first
- 2) if you are *viewing* memory bytes - but they are *displayed* as words

Note: if you read and write complete words consistently, there is no need to care!

However, ASCII string values are bytes but we view them in MIPS as a series of words - hence *viewing* ASCII strings can cause confusion