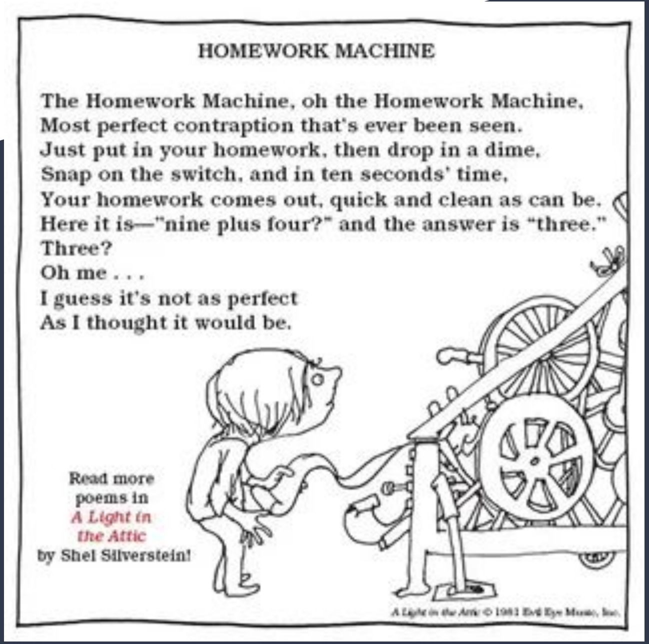
CS 2340 – Computer Architecture

02 Introduction to Assembly Language Programming
Dr. Alice Wang



Shel Silverstein predicted ChatGPT

Housekeeping

- To be counted for the Attendance policy you don’t need to get the answer correct for the Attendance Quiz.
- Don’t be concerned if you got the wrong answer
- I won’t check any of the grades, I will only check that the quiz is submitted

Review

Last time

- Computing is pervasive
- All computers are made up of CPU, memory and I/O

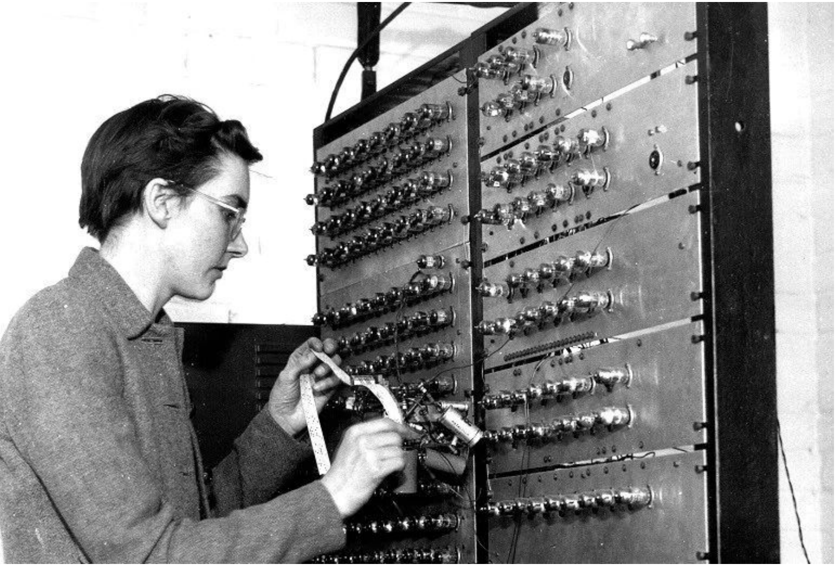
Today

- Introduction to MIPS assembly language
- Not comprehensive, but will be enough to write simple programs
- Tutorial/Demo of MARS

Download the MARS RISC ASSEMBLER/SIMULATOR on eLearning or in MS Teams

History

Kathleen Booth	
Born	Kathleen Hylda Valerie Britten 9 July 1922 Stourbridge, Worcestershire, England
Died	29 September 2022 (aged 100)
Alma mater	University of London
Known for	Invented the first assembly language for her University's computer
Spouse	Andrew Booth (m. 1950; died 2009)
Scientific career	
Fields	Computer science
Institutions	Birkbeck College



- At London’s Birkbeck College joined a team of scientists who were performing calculations using X-ray images to determine crystal structures. She was the sole programmer.
- She was credited with the first “assembly language,” and assembler to create the machine language

Review: Microprocessor Software

Software is a set of instructions, data or programs used to operate computers and execute specific tasks

	Level	Readability	Code-type	Examples
High-level code	High-level	Human-readable	Machine-Independent	C, Java, Python
Assembly code	Low-level	Human-readable	Machine-dependent	ARM, MIPS, x86
Machine code	Lowest-level	Machine-readable	Machine-dependent	0’s and 1’s

Review: Coding Example

High-level

```
int f, g, y;  // global


int main(void)
{
  f = 2;
  g = 3;

  y = sum(f, g);
  return y;
}

int sum(int a, int b) {
  return (a + b);
}
```

Assembly code

```
.data
f:
g:
y:
.text
main:
  addi $sp, $sp, -4    # stack frame
  sw   $ra, 0($sp)     # store $ra
  addi $a0, $0, 2      # $a0 = 2
  sw   $a0, f          # f = 2
  addi $a1, $0, 3      # $a1 = 3
  sw   $a1, g          # g = 3
  jal  sum             # call sum
  sw   $v0, y          # y = sum()
  lw   $ra, 0($sp)     # restore$ra
  addi $sp, $sp, 4     # restore $sp
  jr   $ra             # return to
    OS
sum:
  add  $v0, $a0, $a1   # $v0 = a + b
  jr   $ra             # return
```

Machine coding

Executable file header	Text Size	Data Size
	0x34 (52 bytes)	0xC (12 bytes)

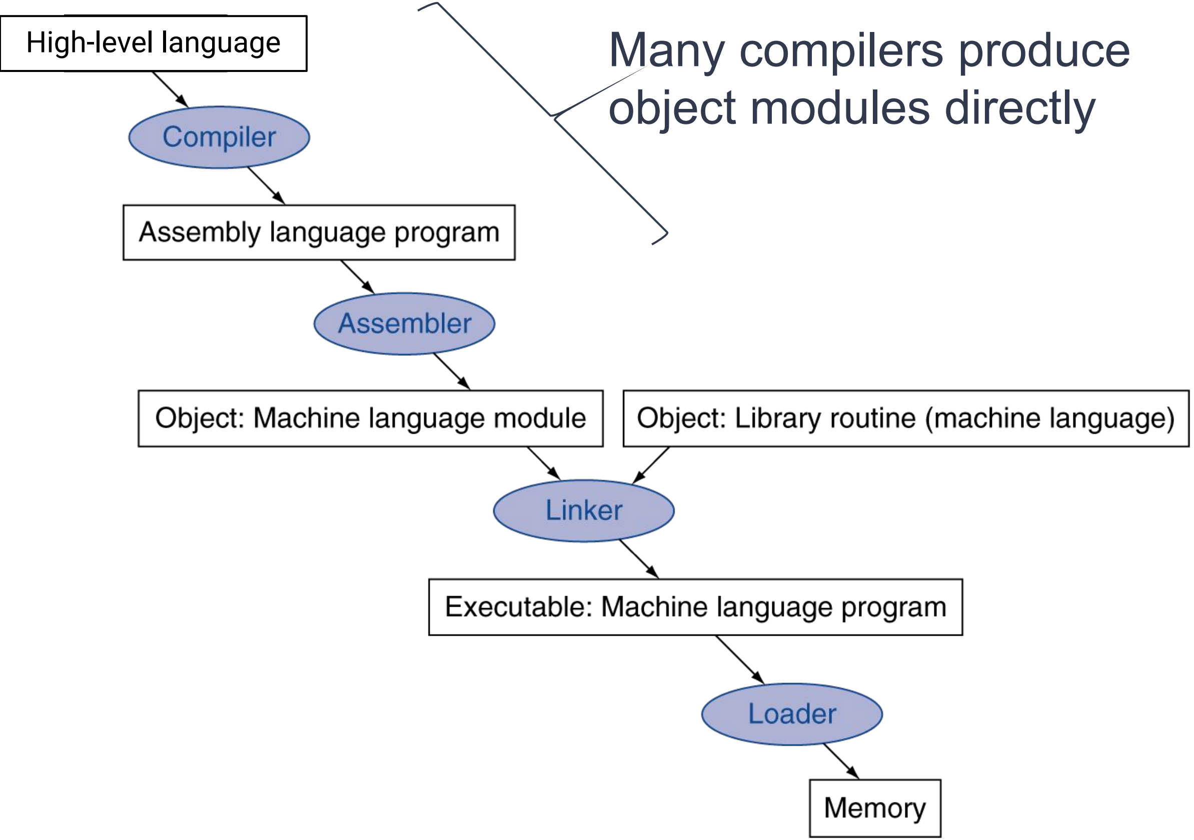
Text segment	Address	Instruction
addi $sp, $sp, -4	0x00400000	0x23BDFFFC
sw $ra, 0 ($sp)	0x00400004	0xAFBF0000
addi $a0, $0, 2	0x00400008	0x20040002
sw $a0, 0x8000 ($gp)	0x0040000C	0xAF848000
addi $a1, $0, 3	0x00400010	0x20050003
sw $a1, 0x8004 ($gp)	0x00400014	0xAF858004
jal 0x0040002C	0x00400018	0x0C10000B
sw $v0, 0x8008 ($gp)	0x0040001C	0xAF828008
lw $ra, 0 ($sp)	0x00400020	0x8FBF0000
addi $sp, $sp, -4	0x00400024	0x23BD0004
jr $ra	0x00400028	0x03E00008
add $v0, $a0, $a1	0x0040002C	0x00851020
jr $ra	0x00400030	0x03E00008

Data segment	Address	Data
	0x10000000	f
	0x10000004	g
	0x10000008	y

Do I need to know Assembly?

- Yes!
- Access the bare metal HW
- Used to address low-level performance issues
- Commonly used on device drivers and embedded systems
- Tricky part: Assembly code is specific to the machine
  - You can’t use x86 Assembly Code on ARM CPU

Translation Hierarchy



Many compilers produce object modules directly

What is a Compiler?

com·pil·er

/kəmˈpīlər/

noun

1. a person who produces a list or book by assembling information or written material collected from other sources.
   "this passage was revised in different ways by later compilers"
2. **COMPUTING**
   a program that converts instructions into a machine-code or lower-level form so that they can be read and executed by a computer.
   "conversion would require more than just running it through a different compiler"

- Lower-level aka Assembly code

What is an Assembler?

as·sem·bler

/əˈsemblər/

A software tool for converting instructions written in assembly code into machine code (zeros and ones). The output is often an output file (.o or .obj) which includes instructions, data, and information to place the program into memory.

MARS

- MARS is a MIPS emulator, assembler, debugger (using breakpoints) and runtime simulator written in Java
- It also has an editing function for writing MIPS assembly language programs
- We will be using MARS extensively in our projects and in our lecture

What is an Instruction Set Architecture (ISA)?

- ISA and Computer Architecture are interchangeable
  - Defines instructions, data types, registers, memory management and features, addressing modes, and the input/output model
- Most common ISA’s
  - Reduced Instruction Set Computer (RISC)
    - ARM, MIPS, RISC-V
  - Complex Instruction Set Computer (CISC)
    - X86
- Custom ISA’s - Application specific
  - GPU (CUDA, NVIDIA), TPU (Google)
- Not the implementation of the processor (microarchitecture)

MIPS instruction set

- Used as the example throughout this class
- Invented at Stanford University
  - Later commercialized by MIPS Technologies
- Where you can find MIPS CPU's in the wild
  - Classic game consoles: Nintendo 64 (1996), Sony Playstation (1994)
  - Cars: Tesla Model S

Assembly code – Template

Header is required for all code (Best Practice)

Comments preceded by # - won’t be compiled

```
# Assembly program template
# Author: your name
# Date: current date
# Description: high-level description of your program

.data

# Data segment:
# constant and variable definitions go here

.text

# Text segment: assembly instructions go here
# your code
```

.data is the start of the section where you store all of your data (constants & variables)

.text is the start of the section where your assembly code goes

Data storage

Use local labels to easily access the data's location

```
.data
  val1: .word 1 # the first value
  val2: .word 2 # the second value
  A:    .word 1,2,3,4,5,6,7
             # An Array of data called A

  name: .asciiz "Daniel" # my name
  prompt:  .asciiz "Enter a number"
                 # Prompt to user
  newline: .asciiz "\n"
```

Memory Labels (similar to variables)

- It is often necessary to refer to a memory location by name, whether the location contains an instruction or data
- Labels in MIPS assembler must start with a letter and can contain letters, digits, dollar signs, and underscores
- The label name is followed by a colon

Data storage

```
.data
  val1:  .word 1 # the first value
  val2:  .word 2 # the second value
  A:     .word 1,2,3,4,5,6,7
        # An Array of data called A

  name:  .asciiz "Daniel" # my name
  prompt:   .asciiz "Enter a number"
                # Prompt to user
  newline: .asciiz "\n"
```

Use local labels to easily access the data's location

.word is 32-bit value or 4 Bytes **(A Byte is 8-bits)**

.asciiz stores the string in the NULL-terminated format - puts NULL at the end of the string signifying where the string ends

Assembler Directives

- **.text**, **.data** and **.asciiz** are Assembler Directives.
- Look like statements but do not compile to machine instructions
- Directive names always begin with a period

.align	.data	.eqv	.half	.set
.ascii	.double	.extern	.include	.space
.asciiz	.end_macro	.float	.kdata	.text
.byte	.macro	.globl	.ktext	.word

More about the Assembler Directives in reference

.space

- Reserves a specific number of **bytes** for data
- Use this directive for defining fixed-size buffers, arrays
- Example: Set aside 256 bytes for an array called B

```
B:     .space  256
```

.eqv

- Substitutes second operand for first
- First operand is symbol, second operand is expression (like #define)
- Can be used for strings and entire instructions

```
.eqv LIMIT     20
.eqv CTR       $t2
.eqv CLEAR_CTR add CTR, $zero, 0
```

Learning MIPS assembly coding

Four key things to know:

- Registers
- Instructions
- Memory
- Syscall interface

MIPS Register Set

Name	Register Number	Usage
`$0,$zero`	0	the constant value 0
`$at`	1	assembler temporary
`$v0-$v1`	2-3	Function return values
`$a0-$a3`	4-7	Function arguments
`$t0-$t7`	8-15	temporaries
`$s0-$s7`	16-23	saved variables
`$t8-$t9`	24-25	more temporaries
`$k0-$k1`	26-27	OS temporaries
`$gp`	28	global pointer
`$sp`	29	stack pointer
`$fp`	30	frame pointer
`$ra`	31	Function return address

Registers - where we store data that the CPU uses in instructions

- $ before name
- Example: $t0 is the 0th temporary register
- In total there are 32 registers in MIPS, each is 32-bit

Review:

1 Word = ____ bits

1 Byte = ____ bits

MIPS Register Set

Name	Register Number	Usage
`$0,$zero`	0	the constant value 0
`$at`	1	assembler temporary
`$v0-$v1`	2-3	Function return values
`$a0-$a3`	4-7	Function arguments
`$t0-$t7`	8-15	temporaries
`$s0-$s7`	16-23	saved variables
`$t8-$t9`	24-25	more temporaries
`$k0-$k1`	26-27	OS temporaries
`$gp`	28	global pointer
`$sp`	29	stack pointer
`$fp`	30	frame pointer
`$ra`	31	Function return address

Register names indicate their specific purposes:

- $0 always holds the constant value 0. Very useful!
- $s0 - $s7, the *saved variables*, are used to hold variables
- $t0 - $t9, the *temporary registers*, are used to hold intermediate values during a larger computation
- $v0 - $v1 (return values) and $a0 - $a3 (arguments) have specific uses in procedures and system calls
- We will discuss others at a later date

Instructions: Getting started

Python Code

```
a = 5;
b = 7;
```

MIPS assembly code

```
li $a0, 5
li $v0, 7
```

```
li $dst, immediate    # Load immediate (constant) into
                      # a register ($dst)
```

Instructions: Arithmetic operations

Python Code

```
a = b + c;
a = b - c;
```

MIPS assembly code

```
add $t0, $t1, $t2
sub $s0, $s1, $s2
```

```
add $dst, $src0, $src1 # add $t1 to $t2 and store in $t0
sub $dst, $src0, $src1 # sub $s2 from $s1 and store in $s0
```

The first register is the destination register, the second two are source or inputs

Instructions: Adding constants

Python Code

```
a = b + 5;
a = b - 7;
```

MIPS assembly code

```
addi $t0, $s0, 5
```

`addi $dst, $src0, immediate`

```
# add the immediate (constant) to the value in $src0
and store in register $dst
```

Question: Do we need a subi instruction?

Example: Simple MIPS assembly

Register	Register Value
`$t0`	0
`$t1`	0
`$t2`	0
`$t3`	0

Instructions

```
.text
li $t1, 5      # load immediate 5 to reg $t1
li $t2, 7      # load immediate 7 to reg $t2
add $t3, $t1, $t2
```

Example: Simple MIPS assembly

Register	Register Value
`$t0`	0
`$t1`	5
`$t2`	0
`$t3`	0

Instructions

```
.text
li $t1, 5      # load immediate 5 to reg $t1
li $t2, 7      # load immediate 7 to reg $t2
add $t3, $t1, $t2
```

Example: Simple MIPS assembly

Register	Register Value
`$t0`	0
`$t1`	5
`$t2`	7
`$t3`	0

Instructions

```
.text
li $t1, 5      # load immediate 5 to reg $t1
li $t2, 7      # load immediate 7 to reg $t2
add $t3, $t1, $t2
```

Example: Simple MIPS assembly

Register	Register Value
`$t0`	0
`$t1`	5
`$t2`	7
`$t3`	**12**

Instructions

```
.text
li $t1, 5     # load immediate 5 to reg $t1
li $t2, 7     # load immediate 7 to reg $t2
add $t3, $t1, $t2
```

Mars demo Part 1

Mars demo

- How to startup MARS
  - Look at syllabus if you have a MAC
- How to edit your code
- How to save your code
- The Register Table
- How to Assemble your code
- How to Run your code one-line at a time

Register vs Memory

Register	Register Value
`$t0`	0
`$t1`	5
`$t2`	7
`$t3`	12

Address	Memory Array
0	783
1	90
2	34
3	453
4	44
…	

Registers store data (Variables)

Registers are few (e.g. <100)
→ Only a subset of data

Data is stored by name
(e.g. $t1 = 5)

Memories store data (Arrays and Variables)

Memories are big (1GB~10TB)
→ Stores all of your data

Data is stored by address
(Mem[1] = 90, Address = 1)

Register vs Memory

Register	Register Value
`$t0`	0
`$t1`	5
`$t2`	7
`$t3`	12

Address	Memory Array
0	783
1	90
2	34
3	453
4	44
…	

MIPS CPU **can** perform calculations directly on register data

MIPS CPU **cannot** perform calculations directly on memory data

Therefore, if we want to perform calculation on memory data we have to copy the data from the memory to registers first

Example: MIPS lw

Register	Register Value
`$t0`	23
`$t1`	53
`$t2`	564
`$t3`	100

	Address	Memory Array
x	0	783
y	1	90
A[0]	2	34
A[1]	3	453
A[2]	4	44
…	…	

Python Code

```
x = 783
y = 90
A = np.array[34,453,44]
```

MIPS assembly code

```
.data
x:  .word 783
y:  .word 90
A:  .word 34,453,44
```

How do we execute `A[0]+A[2]`?

Example: MIPS lw

Register	Register Value
`$t0`	23
`$t1`	53
`$t2`	564
`$t3`	100

	Address	Memory Array
	`0`	783
	`1`	90
A[0] ⇨	`2`	34
A[1]	`3`	453
A[2]	`4`	44
…	…	

How do we execute `A[0]+A[2]`?

First, move A[0] from memory to register. 2 step process
(1) Put Address into a register. Use Load Address (`la`)
(2) Put Mem[Address] into a register

Example: MIPS lw

Register	Register Value
`$t0`	23
`$t1`	53
`$t2`	564
`$t3`	100

	Mem Address	Memory Array
	0	783
	1	90
A[0] ⇨	2	34
A[1]	3	453
A[2]	4	44
…	…	

Instructions

```
.text
la $dst, $symbol          # Load Address (label) into a
                          # register ($dst)
```

(1) Put Address into a register. Use Load Address (`la`)

Example: MIPS lw

Register	Register Value
`$t0`	23
`$t1`	53
`$t2`	564
`$t3`	100

	Mem Address	Memory Array
	0	783
	1	90
A[0]	2	34
A[1]	3	453
A[2]	4	44
…	…	

Instructions

```
.text
la $t1, A          # load base address of Array A to $t1
```

(1) Put Address into a register. Use Load Address (`la`)

Example: MIPS lw

Register	Register Value
`$t0`	23
`$t1`	2
`$t2`	564
`$t3`	100

	Mem Address	Memory Array
	0	783
	1	90
A[0] ⇨	2	34
A[1]	3	453
A[2]	4	44
…	…	

Instructions

```
.text
lw $dst, offset($src)    # Load a Word from the memory at
                         # address = $src + offset into reg $dst
```

(2) Put Mem[Address] into a register. Use Load Word (**`lw`**)

Example: MIPS lw

Register	Register Value
`$t0`	23
`$t1`	2
`$t2`	564
`$t3`	100

	Mem Address	Memory Array
	0	783
	1	90
A[0]	2	34
A[1]	3	453
A[2]	4	44
…	…	

Instructions

```
.text
la $t1, A          # load base address of Array A to $t1
lw $t3, 0($t1)     # Calculate address = $t1 + 0 = 2
                   # $t3 = A[0] = Mem[2]
```

(2) Put Mem[Address] into a register. Use Load Word (`lw`)

Example: MIPS lw – Your Turn

Register	Register Value
`$t0`	23
`$t1`	2
`$t2`	564
`$t3`	34

	Mem Address	Memory Array
	0	783
	1	90
A[0]	2	34
A[1]	3	453
A[2]	4	44
…	…	

What is the code to load A[2] to $t2

```
.text
la $t1, A             # load base address of Array A to $t1
lw ___, ___($t1)      # load word from address $t1+2 to $t2
```

Example: MIPS add

Register	Register Value
$t0	23
$t1	2
$t2	44
$t3	34

	Address	Memory Array
	0	783
	1	90
A[0] ⇨	2	34
A[1]	3	453
A[2]	4	44
…	…	

Python Code

```
A = np.array[34,453,44]

a = A[0]        # a=34
b = A[2]        # b=44
c = a+b
```

MIPS Code

```
.text
la $t1, A
lw $t3, 0($t1)
lw $t2, 2($t1)
add $t0, $t2, $t3
```

Example: MIPS sw

Register	Register Value
`$t0`	23
`$t1`	2
`$t2`	44
`$t3`	34

	Address	Memory Array
	0	783
	1	90
A[0] ⇨	2	34
A[1]	3	453
A[2]	4	44
…	…	

Instructions

```
sw $src, offset($dst)      # Store a Word from register $src
                           # to the memory at address $dst+offset
```

(3) Put register data into a Mem[Address]. Use Store Word (**sw**)

Example: MIPS sw

Register	Register Value
`$t0`	78
`$t1`	2
`$t2`	44
`$t3`	34

	Address	Memory Array
	0	783
	1	90
A[0]	2	34
A[1]	3	453
A[2]	4	44
…	…	

Instructions

```
add $t0, $t2, $t3
sw $t0, 1($t1)    # store word from $t0 to address $t1+1 = 3
                  # A[1] = Mem[3] = 78
```

(3) Put register data into a Mem[Address]. Use Store Word (**sw**)

Example: MIPS sw – Your Turn

Register	Register Value
`$t0`	78
`$t1`	2
`$t2`	44
`$t3`	34

	Address	Memory Array
	0	783
	1	90
A[0] ⇨	2	34
A[1]	3	78
A[2]	4	44
…	…	

Instructions

```
sw $t0, -2($t1)  # store word from reg _____ to address _____
```

What does this instruction do?

Instructions (so far)

```
li $dst, immediate
la $dst, $symbol
lw $dst, offset($src)
sw $src, offset($dst)
add $dst, $src0, $src1
sub $dst, $src0, $src1
addi $dst, $src0, immediate
```

Now you have enough assembly instructions (`li, la, lw, sw, add, sub, addi`) to do your first programming assignment!

What is a Pseudo-operation (Pseudo-code)?

- A Pseudo-operations are instructions that has no direct machine operation but makes it easier to code in Assembly language.
- The Hardware doesn't need to implement pseudo-operations because the Assembler knows how to read and translate pseudo-code.

Task	**Pseudo-Instruction**	**Programmer Writes**	**Assembler Translates To**
Move the contents of one register to another.	`move <dest> <source>`	`move $t0, $s0`	`addu $t0, $zero, $s0`
Load a constant into a register. (Negative values are handled slightly differently.) "li" stands for `load immediate`.	`li <dest> <immed>`	`li $s0, 10`	`ori $s0, $zero, 10`
Load the *address* of a named memory location into a register. `Value` is a label that the programmer has attached to a memory location. 16 is the offset of that memory location from the beginning of the data segment. It is calculated by the assembler for you.	`la <dest> <label>`	`la $s0, variable`	`lui $at, 0x1001` `ori $s0, $at, 16`

SYSCALL Functions and Interface

Instructions to interact with the outside world (System Calls)

Examples of SYSCALL functions:

- Print to screen : integers, fractions, strings
- Read from Keyboard: integers, fractions, strings
- Allocate to dynamic memory (heap)
- Reading from files (File I/O)
- Generate random numbers

SYSCALL

Registers with specific SYSCALL functions

- **`$v0`** - what action do we take, service number
- **`$a0-$a2,$f12`** - arguments of that action
- **`$v0-$v1,$f1`** - results from the action

Step 1. Load the service number in register $v0.
Step 2. Load argument values, if any, in $a0, $a1, $a2, or $f12
Step 3. Issue the SYSCALL instruction.
Step 4. Retrieve return values, if any, from result registers as specified.

On eLearning there is a PDF with all SYSCALL functions

SYSCALL functions available in MARS

$v0 -
Service number

$a0~$a2, $f12
Arguments to syscall

$v0~$v1, $f0
Results from syscall

Let's use these system calls

Service	Code in $v0	Arguments	Result
print integer	1	$a0 = integer to print	
print float	2	$f12 = float to print	
print double	3	$f12 = double to print	
print string	4	$a0 = address of null-terminated string to print	
read integer	5		$v0 contains integer read
read float	6		$f0 contains float read
read double	7		$f0 contains double read
read string	8	$a0 = address of input buffer $a1 = maximum number of characters to read	*See note below table*
sbrk (allocate heap memory)	9	$a0 = number of bytes to allocate	$v0 contains address of allocated memory
exit (terminate execution)	10		
print character	11	$a0 = character to print	*See note below table*
read character	12		$v0 contains character read

1: Print integer

4: Print string

5: Read integer

[Source]

A very important file – SysCalls.asm

```
# Definitions of various useful system calls for MIPS assembly.
# This file must be included using .include "SysCalls.asm" rather
# than being assembled as a separate module.
# Written by John Cole starting January 2, 2022
      .eqv  SysPrintInt      1
      .eqv  SysPrintFloat    2
      .eqv  SysPrintDouble   3
      .eqv  SysPrintString   4
      .eqv  SysReadInt       5
      .eqv  SysReadFloat     6
      .eqv  SysReadDouble    7
      .eqv  SysReadString    8
      .eqv  SysAlloc         9
      .eqv  SysExit          10
      .eqv  SysPrintChar     11
      .eqv  SysReadChar      12
      .eqv  SysOpenFile      13
      .eqv  SysReadFile      14
      .eqv  SysWriteFile     15
      .eqv  SysCloseFile     16
      .eqv  SysExitValue     17
```

```
.include "SysCalls.asm"
.text
  li  $v0, SysPrintInt     # Code for print integer
  add $a0, $t0, $zero     # load value into $a0
  syscall
```

This file is on eLearning.
You will get points deducted on your assignment if you don't include SysCalls.asm on your projects and if you don't use these names in your syscalls.

SYSCALL – Example code & demo

```
#       CS2340 Lecture 1.  Mars demo of syscall
#
#       Author: Alice Wang
#       Date: 05-03-2024
#       Location: UTD
#

.include "SysCalls.asm"         # include this file in all programs
.data
        msg:    .ascii "Hello World!"
        ArrayA: .word 1,2,3,4,5,6,7


.text
        li $v0, SysReadInt      # service call: read integer from keyboard to $v0
        syscall                 # run system call
        move $s0, $v0           # $s0 = $v0

        li $v0, ____________    # service call: print integer (SysCalls.asm)
        move $a0, $s0           # $a0 = $s0
        syscall                 # run system call

        li $v0, ____________    # service call: print string (SysCalls.asm)
        la $a0, msg             # load address of string to $a0
        syscall                 # run system call

        li $v0, ____________    # service call: exit
        syscall
```

Header (required for full credit)

Include SysCalls.asm (required to include this file in your HW Submission and use it)

Comments on almost every line (required for full credit)

SYSCALL – Example code & demo

What do you think we will see printed on the screen?

```
#        CS2340 Lecture 1.  Mars demo of syscall
#
#        Author: Alice Wang
#        Date: 05-03-2024
#        Location: UTD
#

.include "SysCalls.asm"          # include this file in all programs
.data
        msg:    .ascii "Hello World!"
        ArrayA: .word 1,2,3,4,5,6,7


.text
        li $v0, SysReadInt       # service call: read integer from keyboard to $v0
        syscall                  # run system call
        move $s0, $v0            # $s0 = $v0

        li $v0, ____________     # service call: print integer (SysCalls.asm)
        move $a0, $s0            # $a0 = $s0
        syscall                  # run system call

        li $v0, ____________     # service call: print string (SysCalls.asm)
        la $a0, msg              # load address of string to $a0
        syscall                  # run system call

        li $v0, ____________     # service call: exit
        syscall
```

Data storage

Read an integer from the keyboard

Print the same integer on the screen

Print the string to the screen

Exit gracefully (req'd)

Mars demo
Part 2

The partial code is on eLearning

“lecture2-mars demo.asm”

so that you can try it too.

Mars demo

- How to startup MARS
  - Look at syllabus if you have a MAC
- How to edit your code
- How to save your code
- The Register Table
- The Data memory

Step-by-step demo in the reference section

- How to Assemble your code
- How to Run your code
- Run I/O vs Mars Messages windows
- How to Debug your code using breakpoints
- How to view data in decimal, ASCII & hex

HW #1 – MARS setup and Assembly code practice

.data Section Requirements

Declare the following variables

- a, b, c to hold the numbers 45, 32 and 90 respectively (use .word)
- username to hold the user's name (use .space)
- animal to hold the name of an animal (use .space)

Declare string messages using .asciiz:

- A prompt for username
- A prompt for favorite animal
- Messages to display the results

HW #1 – MARS setup and Assembly code practice

.text Section Requirements

Write MIPS instructions to:

1. Prompt the user for their name and save it to memory (username)
2. Prompt the user for a name of their favorite animal and save it to memory (animal)
3. Prompt user for an integer between 1-50 (store word in variable c)
4. Calculate sum = 2*a + b - c + 21 (load word a, b and c to registers)
5. Display should look like

Run I/O screen will look like

```
Enter your name: Alice
Enter the name of an animal: cheetah
Enter one integer from 1-50: 23
Aloha Alice
Your favorite animal is the cheetah
Our crystal ball says your favorite number is 120
-- program is finished running --
```

Next lecture

CPU Performance

Reference: Directives

.align Align next data item on specified byte boundary (0=byte, 1=half, 2=word, 3=double)

.ascii Store the string in the Data segment but do not add null terminator

.asciiz Store the string in the Data segment and add null terminator

.byte Store the listed value(s) as 8 bit bytes

.data Subsequent items stored in Data segment at next available address

.double Store the listed value(s) as double precision floating point

.end_macro End macro definition. See .macro

.macro Begin macro definition. See .end_macro

.eqv Substitute second operand for first. First operand is symbol, second operand is expression (like #define)

.extern Declare the listed label and byte length to be a global data field

.float Store the listed value(s) as single precision floating point

Reference: Directives

.global Declare the listed label(s) as global to enable referencing from other files

.half Store the listed value(s) as 16 bit halfwords on halfword boundary

.include Insert the contents of the specified file. Put filename in quotes.

.kdata Subsequent items stored in Kernel Data segment at next available address

.ktext Subsequent items (instructions) stored in Kernel Text segment at next available address

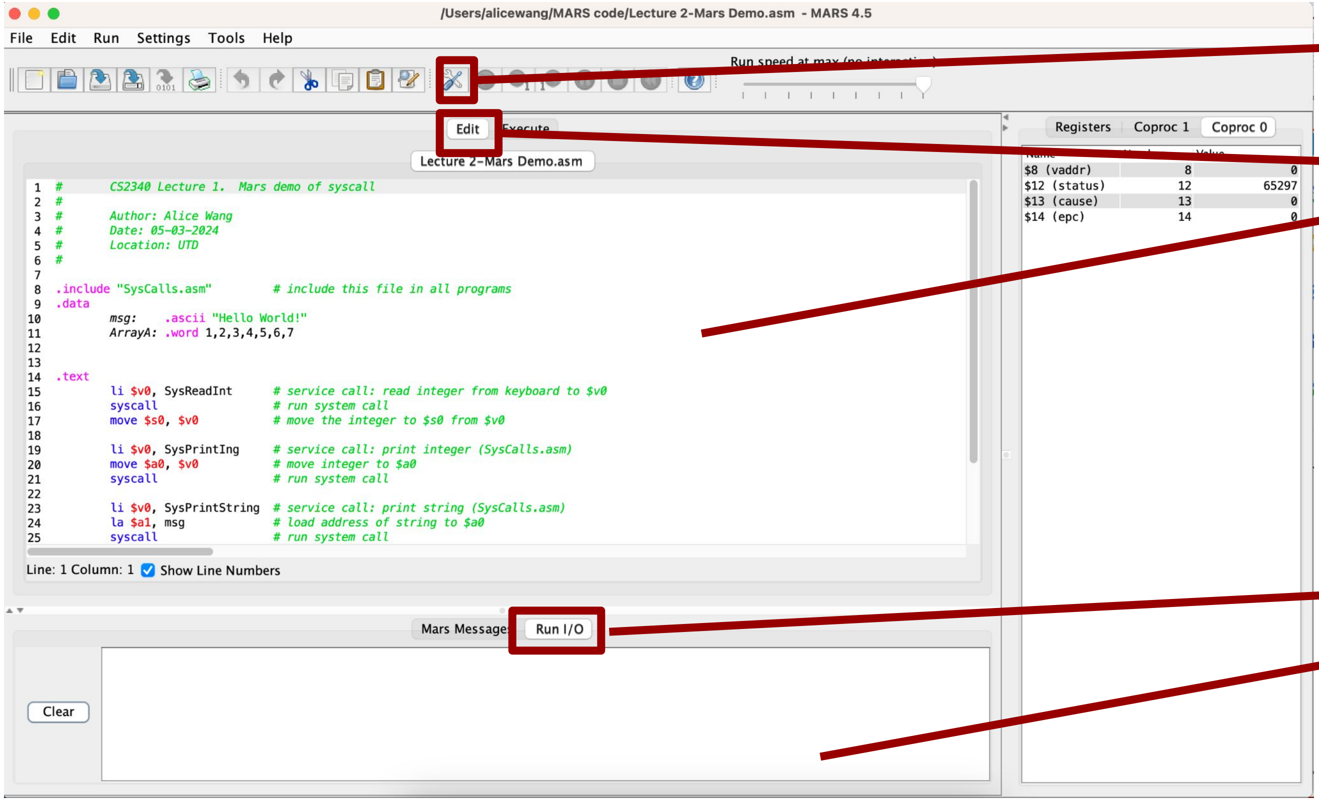
.set Set assembler variables. Currently ignored but included for SPIM compatability

.space Reserve the next specified number of bytes in Data segment

.text Subsequent items (instructions) stored in Text segment at next available address

.word Store the listed value(s) as 32 bit words on word boundary

Reference: MARS Demo

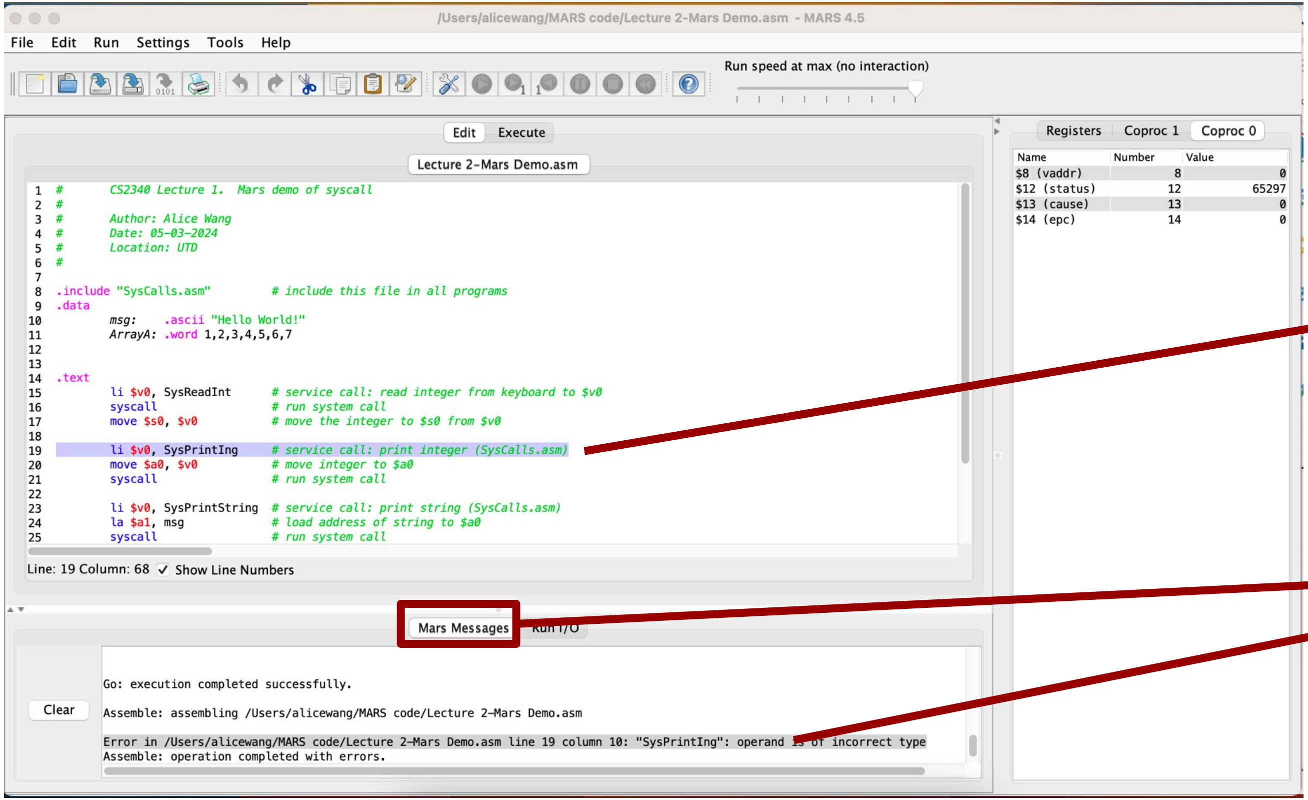


Saves and Assembles code

Click to see Assembly Code Editor: Here you can edit your code.

Click to see Run I/O Display: Print screen and Read from Keyboard

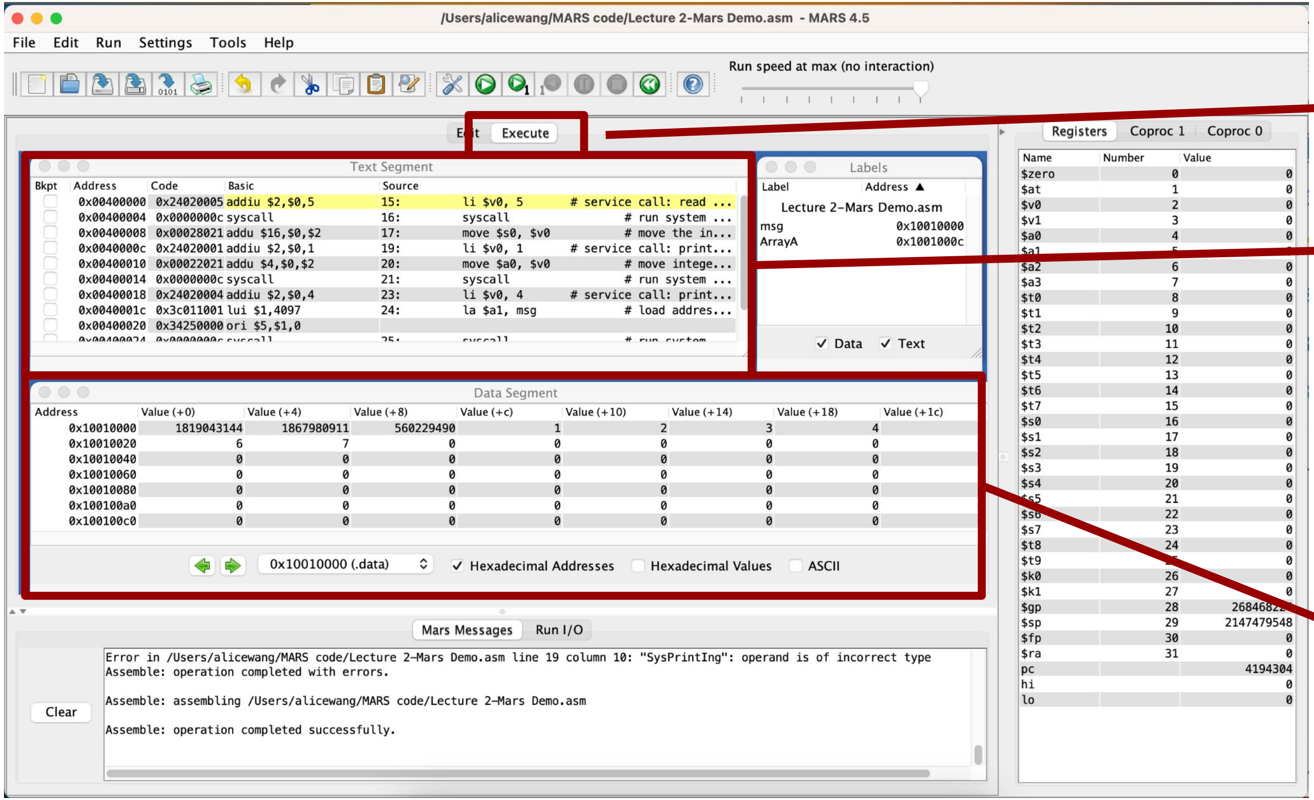
Reference: MARS Demo



Error highlighted

Click to view Mars Message window: Console

Reference: MARS Demo

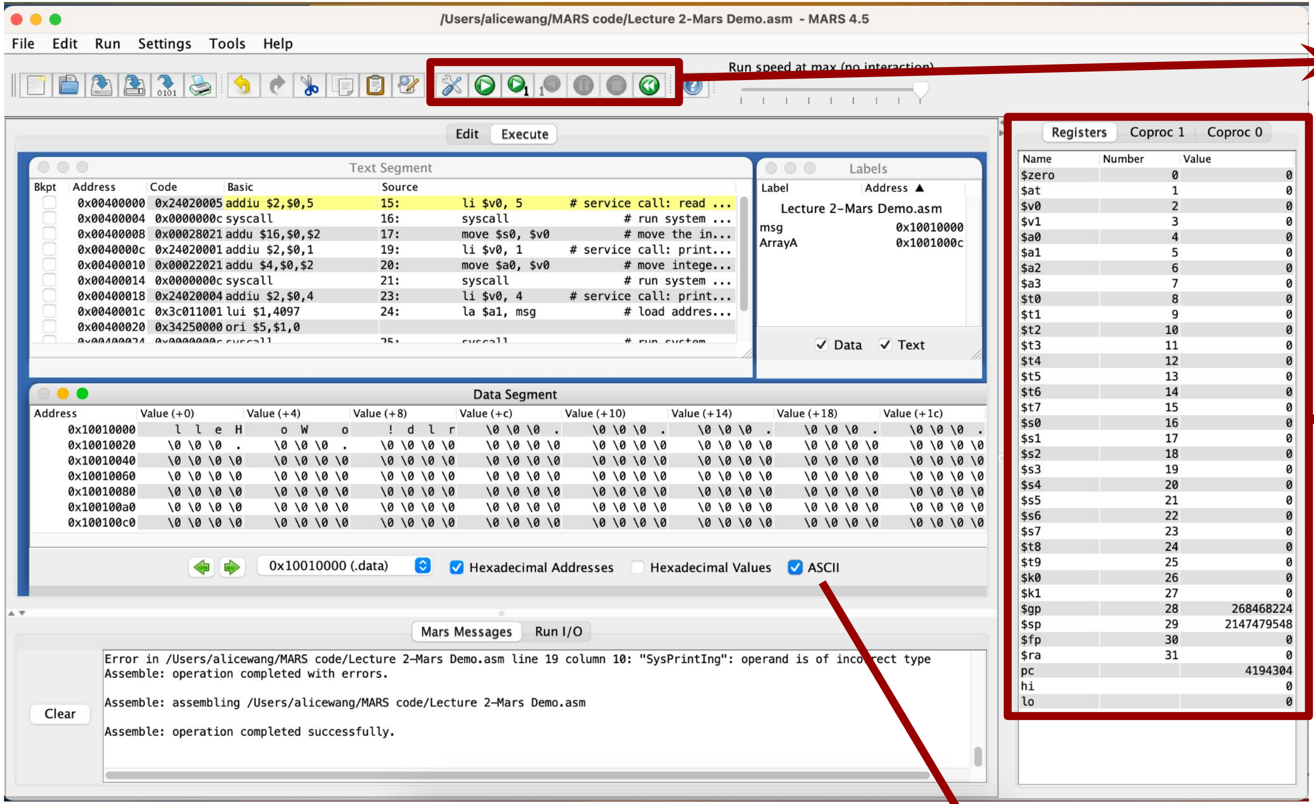


Click to view Execute windows

.text
Instruction Memory: Your code after assembled

.data
Data Memory: Where your Array, strings and variables are stored

Reference: MARS Demo



Run all

Run 1 step

Reset to start

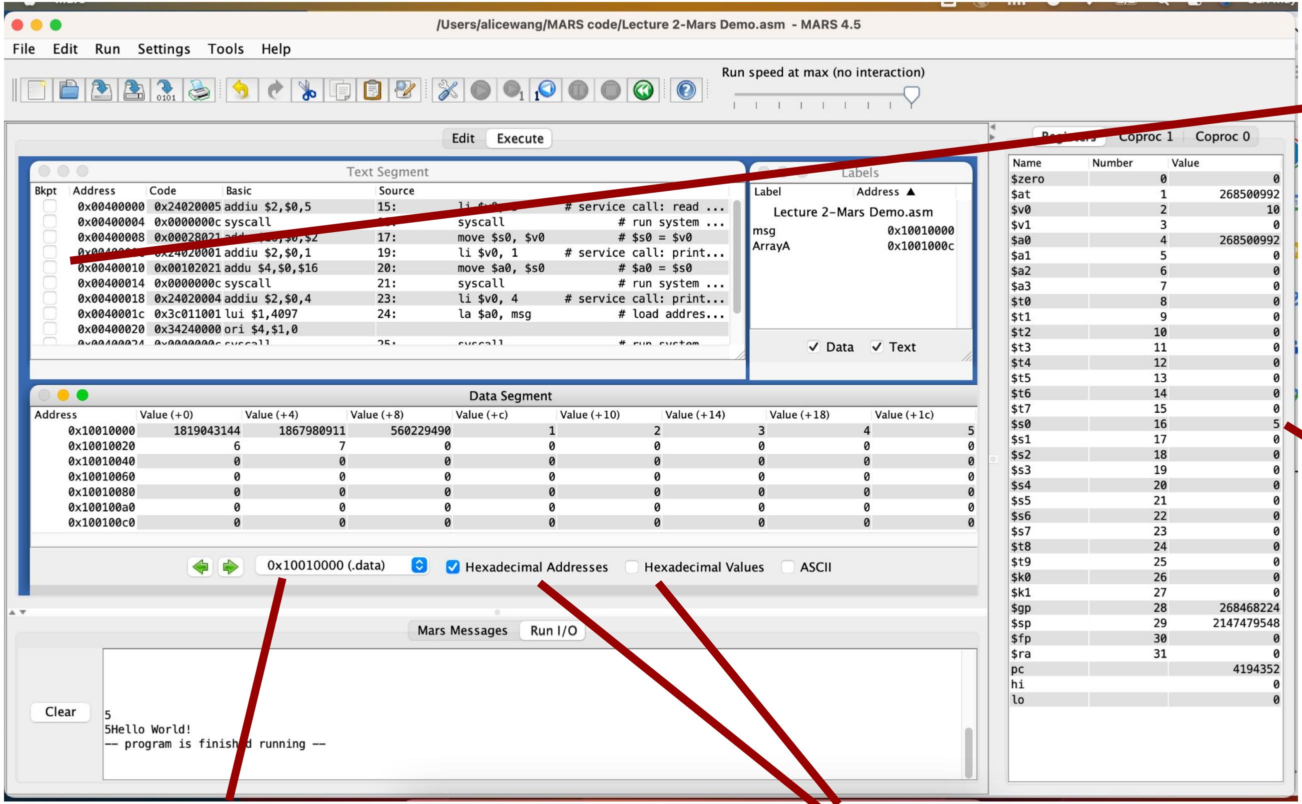
Register Table: View data stored in registers

Click here to see your strings in ASCII

http://courses.missouristate.edu/KenVollmar/MARS

A. Wang, 2340

Reference: MARS Demo



Set breakpoints here

Register $s0 is updated

Drop down to view stack, heap, etc.

Click these to view data or addresses in Hex vs Dec

References: Keyboard Shortcuts

- F3 – Assemble and run
- F5 – Run the program to the next breakpoint, if any
- F7 – Single-step one instruction
- F8 – Back up one instruction
- F12 – Reset program and start over

Extra: MARS Plug-ins

- Tools are plug-ins to extend MARS functionality
- Many useful tools are already included including one developed by a UTD student
  - VisualStack tool displays the state of the stack during a program execution (UTD)
  - Floating-point representation tool displays floating point number representations and converts between them
  - Instruction counter tool shows the number of instructions executed per type (R-type, I-type and J-type)