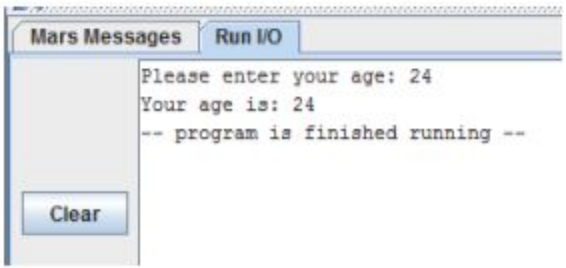
I/O syscall demo




```
# MARS syscalls

.data
age:    .word   0
msg1:   .asciiz "Please enter your age: "
msg2:   .asciiz "Your age is: "

.text
main:
        # prompt user for age
        la      $a0, msg1
        li      $v0, 4
        syscall
        # get int from user
        li      $v0, 5
        syscall
        sw      $v0, age

        # echo data to user
        la      $a0, msg2
        li      $v0, 4
        syscall
        lw      $a0, age
        li      $v0, 1
        syscall

exit:   li      $v0, 10
        syscall
```

Coding Practice

Write a program to:

- get the user's name
- get the user's age
- get a neighbor's name
- get the neighbor's age
- print a message echoing both names and the combined years of wisdom
- print a message with the difference in ages

Extended Hello World

Assembler directive ".align 2" forces the next item to begin on a word boundary

Sample run:

```
Please enter your name: Daisy
Hello Daisy
```

Memory after run:

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	a e l P	e e s	r e t n	u o y	a n r	: e m	l e H \0	\0 o l
0x10010020	s i a D	\0 \0 \n y	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
0x10010040	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
0x10010060	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
0x10010080	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
0x100100a0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0

```
# Hello World with user input

.data
prompt: .asciiz "Please enter your name: "
hello:  .asciiz "Hello "
        .align 2
name:   .space  20   # buffer for user input

.text
main:
        # prompt user for name
        li      $v0, 4
        la      $a0, prompt
        syscall

        # get name from user and save it
        li      $v0, 8
        la      $a0, name
        li      $a1, 20
        syscall

        # say hello to user
        li      $v0, 4
        la      $a0, hello
        syscall
        li      $v0, 4
        la      $a0, name
        syscall

        li      $v0, 10
        syscall                 # syscall to exit program
```

MIPS and 32

- 32 registers, 32 bits each
- 32-bit words in memory
- Instructions are 32 bits
- Addresses are 32 bits

Assembler directives

directive	action
.data	start of data segment
.text	start of code segment
.ascii str	store ascii string, not null-terminated
.asciiz str	store ascii string, null-terminated
.byte b1,...,bn	store these values in successive bytes of memory
.half h1,...,hn	store these values in successive half words of memory
.word w1,...,wn	store these values in successive words
.space n	allocate n bytes of memory
.float f1,...,fn	store single precision values in successive words
.double d1,...,dn	store double precision values in successive locations
.align n	align next item on 2^n boundary

Assembler directives

```
        .data
n:      .word   5
c:      .word   'c'
d:      .asciiz "abc"
```

Address	Value (+0)	Value (+4)	Value (+8)
0x10010000	0x00000005	0x00000063	0x00636261

Static data

Format:

```
Name:  storage type     values(s)
```

Examples:

```
.data
array1: .byte  ‘a’,’b’
        .align  2
array2: .space  24
str1:   .asciiz “hello”
```

MIPS operands

1. Registers
2. Memory locations
3. Constant (immediate)

Each opcode type works with a specific set of operand types

MIPS machine code

- Each MIPS instruction assembles in to a 32-bit word of machine code

- Opcode represented as a 6-bit binary number
- See opcodes in MIPS card in Piazza
- Registers are represented as 5 bits, $2^5 = 32$
- $t0 – $t7 are reg's 8 – 15
- $t8 – $t9 are reg's 24 – 25
- $s0 – $s7 are reg's 16 – 23

arithmetic/logic instructions

R format:

Code	Basic	Source
0x012a4020	add $8,$9,$10	9: add $t0, $t1, $t2

	opcode	rs	rt	rd	shamt	funct
R	opcode	rs	rt	rd	shamt	funct
	31 26	25 21	20 16	15 11	10 6	5 0

Caution: instruction is rd, rs, rt but machine code is rs, rt, rd

arithmetic/logic instructions

R format:

	opcode	rs	rt	rd	shamt	funct
R	31 26	25 21	20 16	15 11	10 6	5 0

Practice: Translate sub $s0, $t3, $t4 into machine code binary/hex.

load/store instructions

la - load address (pseudo)

lb - load byte

lbu - load byte unsigned

lw - load word

sb - store byte

sw - store word

Load/Store use I-format

- Rt is the destination
- Rs is the source
- Address = constant + rs
- Constant can be positive or negative

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

Load/Store byte example

```
        .data
s1:     .byte   'a', 'b'
        .align  2
s2:     .space  2

        .text
        la   $t1, s1
        la   $t2, s2
        lbu  $t0, ($t1)
        sb   $t0, ($t2)
        addi $t1, $t1, 1
        addi $t2, $t2, 1
        lbu  $t0, ($t1)
        sb   $t0, ($t2)

        li  $v0, 10
        syscall
```

Address	Value (+0)	Value (+4)
0x10010000	\0 \0 b a	\0 \0 b a

Word = 32 bits = 4 bytes

Bytes are stored big-endian

High address byte stored at low address within a word.

“Hiya”

Stored as:

a y i H

Note that this doesn't apply to storing a word like integer 5.

Practice

Hand assemble the following instruction into machine code:

```
lw $t0, 8($t1)
```

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

Practice

The “addi” instruction uses the I-format because it needs a constant.

Hand assemble the following instruction into machine code:

```
addi  $t1, $t2, 5
```

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

Writing a MIPS program

One approach:

1. Write pseudocode or code in your favorite higher-level language
2. Think about what data you need and reserve space in the .data section
3. Break the problem into pieces, like: input data, process, output data
4. Code and test each section at a time

Debugging MIPS

Makes you appreciate HLL and IDEs.

- Take advantage of breakpoints in MARS
- Stop and look at registers/memory to see if the program is doing what you thought it would do
- Stop after a few lines of coding to inspect and see if it’s working, don’t wait until you finish the program

Summary

You know how to:

- Write simple MIPS programs
- Reserve static memory in a MIPS program for integers, text
- Hand assemble MIPS instructions
- Run/debug MIPS programs

Debug Practice

What's wrong with this program?

```
# Implement the following expression in MIPS:
#     result = (a - b) + (c - d) + 2a
# Test:  30  = (12 - 5) + (10 - 4) + 12 + 12

.data
msg:    .asciiz "Your result is: "
result: .word
a:      .word   12
b:      .word   5
c:      .word   10
d:      .word   4

.text
main:
        # load data into registers
        lw      $t1, a
        lw      $t2, b
        lw      $t3, c
        lw      $t4, d

        # compute  (a - b) + (c - d) + 2a
        sub     $t1, $t1, $t2   # (a - b)
        sub     $t3, $t3, $t4   # (c - d)
        add     $t1, $t1, $t1   # 2a
        add     $t5, $t1, $t3   # combine intermediate results

        # store result
        sw      $t5, result

        # output result
        li      $v0, 4
        lw      $a0, msg
        syscall                 # output msg
        li      $v0, 1
        la      $a0, result
        syscall                 # output result

        # exit program
        li      $v0, 10
        syscall
```