I/O syscall demo

```assembly
# 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:
MIPS and 32

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

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Value (+0)</th>
<th>Value (+4)</th>
<th>Value (+8)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x10010000</td>
<td>0x00000005</td>
<td>0x00000063</td>
<td>0x00636261</td>
</tr>
</tbody>
</table>
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 into 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:

Caution: instruction is rd, rs, rt but machine code is rs, rt, rd
arithmetic/logic instructions

R format:

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

la - load address (pseudo)   sb - store byte
lb - load byte               sw - store word
lbu - load byte unsigned    lw - load word
Load/Store use I-format

- Rt is the destination
- Rs is the source
- Address = constant + rs
- Constant can be positive or negative
Load/Store byte example

```
.data
s1:    .byte 'a', 'b'
.align 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
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Value (+0)</th>
<th>Value (+4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x10010000</td>
<td>\0 \0 b a</td>
<td>\0 \0 b a</td>
</tr>
</tbody>
</table>
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)
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
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?

```mips
.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
```