Table of contents

Overview

Registers

Instructions
  Arithmetic
  Memory
  Control Structures
  System Calls
Overview
.data

Data segment: constant and variable definitions go here (including statically allocated arrays)

- format for declarations: name: storage_type value
- create storage for variable of specified type with given name and value
  - var1: .word 3 # one word of storage with initial value 3
  - array1: .space 40 # 40 bytes of storage for array1

.text

Text segment: assembly instructions go here
## Components of an assembly program

<table>
<thead>
<tr>
<th>Category</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comment</td>
<td># I am a comment</td>
</tr>
<tr>
<td>Assembler directives</td>
<td>.data, .asciiz</td>
</tr>
<tr>
<td>Operation mnemonic</td>
<td>add, addi, lw, bne</td>
</tr>
<tr>
<td>Register name</td>
<td>$zero, $t3</td>
</tr>
<tr>
<td>Address label (declaration)</td>
<td>loop1:</td>
</tr>
<tr>
<td>Address label (use)</td>
<td>loop1</td>
</tr>
<tr>
<td>Integer constant</td>
<td>8, -4, 0xA9</td>
</tr>
<tr>
<td>Character constant</td>
<td>'h', '\t'</td>
</tr>
<tr>
<td>String constant</td>
<td>&quot;Hello, world\n&quot;</td>
</tr>
</tbody>
</table>
# Description: a simple hello world program

.data

hellostr: .asciiz "Hello, world\n"

.text

li $v0, 4  # setup print syscall
la $a0, hellostr  # argument to print string
syscall  # tell the OS to do the system call
li $v0, 10  # setup exit syscall
syscall  # tell the OS to perform the syscall
Registers
Register

- 32 general-purpose registers
- register preceded by $ in assembly language
- two formats: name ($zero) or number ($0)
- holds 32 bits value (= 4 bytes = 1 word)
- stack grows from high memory to low memory
### Registers

<table>
<thead>
<tr>
<th>Register number</th>
<th>Alternative name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$zero</td>
<td>the value 0</td>
</tr>
<tr>
<td>1</td>
<td>$at</td>
<td>assembler temporary: reserved by the assembler</td>
</tr>
<tr>
<td>2-3</td>
<td>$v0-$v1</td>
<td>function return values</td>
</tr>
<tr>
<td>4-7</td>
<td>$a0-$a3</td>
<td>function arguments: first four parameters for function (no preserved across function call)</td>
</tr>
<tr>
<td>8-15</td>
<td>$t0-$t7</td>
<td>temporaries (not preserved across function calls)</td>
</tr>
<tr>
<td>16-23</td>
<td>$s0-$s7</td>
<td>saved temporaries (preserved across function calls)</td>
</tr>
<tr>
<td>24-25</td>
<td>$t8-$t9</td>
<td>temporaries: (not preserved across function calls)</td>
</tr>
<tr>
<td>26-27</td>
<td>$k0-$k1</td>
<td>reserved for use by the interrupt/trap handler</td>
</tr>
<tr>
<td>28</td>
<td>$gp</td>
<td>global pointer: base of global data segment</td>
</tr>
<tr>
<td>29</td>
<td>$sp</td>
<td>stack pointer: points to last location on stack</td>
</tr>
<tr>
<td>30</td>
<td>$s8/$fp</td>
<td>saved value / frame pointer (preserved across function call)</td>
</tr>
<tr>
<td>31</td>
<td>$ra</td>
<td>return address</td>
</tr>
</tbody>
</table>

- Special Hi and Lo registers (not shown above) holds result of multiplication and division (see example later)
Instructions
Instructions

Arithmetic
Arithmetic Instructions

- Most use three operands
- All operands are registers or immediate values (no memory access)
- All operands are 4 bytes (a word)
### Arithmetic & Move Instruction Examples

```
add    $t0,$t1,$t2
# $t0 = $t1 + $t2;
# add as signed (2’s complement) integers

sub    $t2,$t3,$t4  # $t2 = $t3 - $t4
addi   $t2,$t3, 5  # $t2 = $t3 + 5; "add immediate"
addu   $t1,$t6,$t7  # $t1 = $t6 + $t7; add as unsigned integers
subu   $t1,$t6,$t7  # $t1 = $t6 + $t7; subtract as unsigned integers

mult   $t3,$t4
# multiply 32-bit quantities in $t3 and $t4, and store 64-bit
# result in special registers Lo and Hi: (Hi,Lo) = $t3 * $t4

div    $t5,$t6
# Lo = $t5 / $t6  (integer quotient)
# Hi = $t5 mod $t6 (remainder)

mfhi   $t0
# move value from special register Hi to $t0:  $t0 = Hi
mflo   $t1
# move value from special register Lo to $t1:  $t1 = Lo
move   $t2,$t3  # $t2 = $t3
```
Instructions

Memory
Load / Store Instructions

- Memory access only allowed with explicit load and store instructions (load/store architecture)
- All other instructions use register operands

Load

- `lw` register_destination, mem_source
  copy a word (4 bytes) at source memory location to destination register
- `lb` register_destination, mem_source
  copy a byte to low-order byte of destination register (sign extend higher-order bytes)
- `li` register_destination, value
  load immediate value into destination register (pseudo-instruction)
- **Store**
  - `sw register_source, mem_destination`
    - store a word (4 bytes) from source register to memory location
  - `sb register_source, mem_destination`
    - store a byte (low-order) from source register to memory location

---

**Example**

```
.data
var1: .word 23  # declare storage for var1; initial value is 23

.text
lw  $t0, var1  # load content of mem location into register $t0:  $t0 = 23
li  $t1, 5    # $t1 = 5    ("load immediate")
sw  $t1, var1  # store content of $t1 into mem:  *var1 = 5
```

⚠️ var1 represents a pointer to a word since it is an address.
load address:
- \texttt{la }$t0$, \texttt{var1}
  
copy memory address of \texttt{var1} into register $t0$

indirect addressing:
- \texttt{lw }$t1$, ($t0$)
  
load word at memory address contained in $t0$ into $t2$
- \texttt{sw }$t2$, ($t0$)
  
store word in register $t2$ into memory at address contained in $t0$

based/indexed addressing (useful for field access in struct):
- \texttt{lw }$t2$, 4($t0$)
  
load word at memory address ($t0+4$) into register $t2$
- \texttt{sw }$t2$, -12($t0$)
  
store content of register $t2$ into memory at address ($t0-12$)
Examples

.data
array1: .space 12 # declare 12 bytes of storage

.text
la $t0, array1 # load base address of array into $t0
li $t1, 5 # $t1 = 5 ("load immediate")
sw $t1, ($t0) # first array element set to 5
li $t1, 13 # $t1 = 13
sw $t1, 4($t0) # second array element set to 13
li $t1, -7 # $t1 = -7
sw $t1, 8($t0) # third array element set to -7
Exercise

Write a MIPS assembly program corresponding to the following C code:

```c
struct point_t {
    int x;
    int y;
};

struct point_t p;
int arr[12];

void foo() {
    p.x = 2;
    p.y = 4;
    arr[3] = 6;
}
```
Instructions

Control Structures
Control structures

- **Branches:**
  
  \[
  \begin{align*}
  b & \quad \text{target} & \quad \# \quad \text{unconditional branch to target} \\
  \text{beq} & \quad \$t0, \$t1, \text{target} & \quad \# \quad \text{branch to target if } \$t0 = \$t1 \\
  \text{blt} & \quad \$t0, \$t1, \text{target} & \quad \# \quad \text{branch to target if } \$t0 < \$t1 \\
  \text{ble} & \quad \$t0, \$t1, \text{target} & \quad \# \quad \text{branch to target if } \$t0 \leq \$t1 \\
  \text{bgt} & \quad \$t0, \$t1, \text{target} & \quad \# \quad \text{branch to target if } \$t0 > \$t1 \\
  \text{bge} & \quad \$t0, \$t1, \text{target} & \quad \# \quad \text{branch to target if } \$t0 \geq \$t1 \\
  \text{bne} & \quad \$t0, \$t1, \text{target} & \quad \# \quad \text{branch to target if } \$t0 \neq \$t1
  \end{align*}
  \]

**Example**

\[
\begin{align*}
\text{addi} & \quad \$t0, \$\text{zero}, 0 & \quad \# \quad t0 = 0 \\
\text{addi} & \quad \$t1, \$\text{zero}, 10 & \quad \# \quad t1 = 10 \\
\text{addi} & \quad \$t0, \$t0, 1 & \quad \# \quad t0 = t0+1 \\
\text{blt} & \quad \$t0, \$t1, \text{loop} & \quad \# \quad \text{branch to loop if } t0 < t1 \quad (t0<10)
\end{align*}
\]
Control structures

- Jumps:

  \[ j \quad \text{target} \]
  \# unconditional jump to program label target

  \[ \text{jr} \quad \$t3 \]
  \# jump to address contained in $t3 ("jump register")

- Subroutine (function) call:

  \[ \text{jal} \quad \text{label} \quad \# \quad "jump and link" \]
  - copy program counter (return address) to register $ra (return address register)
  - jump to program instruction at label

  \[ \text{jr} \quad \$ra \quad \# \quad "jump register" \]
  - jump to return address in $ra (stored by jal instruction)

  In case of nested function calls, the return address should be saved to the stack and restored accordingly.
Instructions

System Calls
System calls are used to interface with the operating systems. For instance input/output or dynamic memory allocation.

Using system calls:

1. load the service number in register $v0
2. load argument values in $a0, $a1, …
3. issue the syscall instruction
4. retrieve return value if any

**Example: printing integer on the console**

```
li $v0, 1
# service 1 is print integer

add $a0, $t0, $zero
# load desired value into argument register $a0

syscall
```
System calls table

<table>
<thead>
<tr>
<th>Service</th>
<th>$v0</th>
<th>Arguments</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>print integer</td>
<td>1</td>
<td>$a0 = integer to print</td>
<td></td>
</tr>
<tr>
<td>print string</td>
<td>4</td>
<td>$a0 = address of null-terminated string to print</td>
<td></td>
</tr>
<tr>
<td>print character</td>
<td>11</td>
<td>$a0 = character to print</td>
<td></td>
</tr>
<tr>
<td>read integer</td>
<td>5</td>
<td></td>
<td>$v0 = integer read</td>
</tr>
<tr>
<td>read character</td>
<td>12</td>
<td></td>
<td>$v0 = character read</td>
</tr>
<tr>
<td>allocate heap</td>
<td>9</td>
<td>$a0 = number of bytes to allocate</td>
<td>$v0 = address of allocated memory</td>
</tr>
</tbody>
</table>
Next lecture:

- Introduction to Code Generation