Introduction to MIPS Assembly Programming

January 23–25, 2013

Outline

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

Components of an assembly program

Lexical category	**Example(s)**
Comment	`# do the thing`
Assembler directive	`.data`, `.asciiz`, `.global`
Operation mnemonic	`add`, `addi`, `lw`, `bne`
Register name	`$10`, `$t2`
Address label (decl)	`hello:`, `length:`, `loop:`
Address label (use)	`hello`, `length`, `loop`
Integer constant	`16`, `-8`, `0xA4`
String constant	`"Hello, world!\n"`
Character constant	`'H'`, `'?'`, `'\n'`

Lexical categories in hello world

```
# Author:       Eric Walkingshaw
# Date:         Jan 18, 2013
# Description:  A simple hello world program!

.data                   # add this stuff to the data segment

                        # load the hello string into data memory
hello:  .asciiz "Hello, world!"

.text                   # now we're in the text segment

        li   $v0, 4     # set up print string syscall
        la   $a0, hello # argument to print string
        syscall         # tell the OS to do the syscall
        li   $v0, 10    # set up exit syscall
        syscall         # tell the OS to do the syscall
```

Pseudocode

What is pseudocode?

- **informal** language
- intended to be read by humans

Useful in two different roles in this class:

1. for understanding assembly instructions
2. for describing algorithms to translate into assembly

Example of role 1: `lw $t1, 8($t2)`

Pseudocode: `$t1 = Memory[$t2+8]`

Pseudocode is not “real” code!
Just a way to help understand what an operation does

How to write assembly code

Writing assembly can be overwhelming and confusing

Strategy

1. develop algorithm in pseudocode
2. break it into small pieces
3. implement (and test) each piece in assembly

It is **extremely helpful** to annotate your assembly code with the pseudocode it implements!

- helps to understand your code later
- much easier to check that code does what you intended

Outline

MIPS register names and conventions

Number	Name	Usage	Preserved?
`$0`	`$zero`	constant `0x00000000`	N/A
`$1`	`$at`	assembler temporary	✗
`$2–$3`	`$v0–$v1`	function return values	✗
`$4–$7`	`$a0–$a3`	function arguments	✗
`$8–$15`	`$t0–$t7`	temporaries	✗
`$16–$23`	`$s0–$s7`	saved temporaries	✓
`$24–$25`	`$t8–$t9`	more temporaries	✗
`$26–$27`	`$k0–$k1`	reserved for OS kernel	N/A
`$28`	`$gp`	global pointer	✓
`$29`	`$sp`	stack pointer	✓
`$30`	`$fp`	frame pointer	✓
`$31`	`$ra`	return address	✓

(for reference)

Integer logic and arithmetic

```
# Instruction           # Meaning in pseudocode

add  $t1, $t2, $t3      # $t1 = $t2 + $t3
sub  $t1, $t2, $t3      # $t1 = $t2 - $t3
and  $t1, $t2, $t3      # $t1 = $t2 & $t3 (bitwise and)
or   $t1, $t2, $t3      # $t1 = $t2 | $t3 (bitwise or)

# set if equal:
seq  $t1, $t2, $t3      # $t1 = $t2 == $t3 ? 1 : 0

# set if less than:
slt  $t1, $t2, $t3      # $t1 = $t2 <  $t3 ? 1 : 0

# set if less than or equal:
sle  $t1, $t2, $t3      # $t1 = $t2 <= $t3 ? 1 : 0
```

Some other instructions of the same form

- `xor`, `nor`
- `sne`, `sgt`, `sge`

Immediate instructions

Like previous instructions, but second operand is a **constant**

- constant is 16-bits, sign-extended to 32-bits
- (reason for this will be clear later)

```
# Instruction            # Meaning in pseudocode

# add/subtract/and immediate:
addi $t1, $t2, 4         # $t1 = $t2 + 4
subi $t1, $t2, 15        # $t1 = $t2 - 15
andi $t1, $t2, 0x00FF    # $t1 = $t2 & 0x00FF

# set if less than immediate:
slti $t1, $t2, 42        # $t1 = $t2 < 42 ? 1 : 0
```

Some other instructions of the same form

- `ori`, `xori`

Multiplication

Result of multiplication is a 64-bit number

- stored in two 32-bit registers, **hi** and **lo**

```
# Instruction          # Meaning in pseudocode
mult $t1, $t2          # hi,lo = $t1 * $t2
mflo $t0               # $t0 = lo
mfhi $t3               # $t3 = hi
```

Shortcut (macro instruction):

```
mul  $t0, $t1, $t2     # hi,lo = $t1 * $t2; $t0 = lo
```

Expands to:

```
mult $t1, $t2
mflo $t0
```

Integer division

Computes quotient and remainder and simultaneously

- stores quotient in `lo`, remainder in `hi`

```
# Instruction          # Meaning in pseudocode
div  $t1, $t2          # lo = $t1 / $t2; hi = $t1 % $t2
```

Manipulating register values

```
# Instruction              # Meaning in pseudocode

# copy register:
move  $t1, $t2             # $t1 = $t2

# load immediate: load constant into register (16-bit)
li    $t1, 42              # $t1 = 42
li    $t1, 'k'             # $t1 = 0x6B

# load address into register
la    $t1, label           # $t1 = label
```

Outline

- Overview of assembly programming
  - MARS tutorial
  - MIPS assembly syntax
  - Role of pseudocode
- Some simple instructions
  - Integer logic and arithmetic
  - Manipulating register values
- Interacting with data memory
  - Declaring constants and variables
  - Reading and writing
- Performing input and output
  - Memory-mapped I/O, role of the OS
  - Using the systemcall interface

Declaring constants and variables

Parts of a declaration: (in data segment)

1. label: memory address of variable
2. directive: “type” of data
   (used by assembler when initializing memory, not enforced)
3. constant: the initial value

```
.data

# string prompt constant
prompt:  .asciiz "What is your favorite number?: "

# variable to store response
favnum:  .word   0
```

No real difference between constants and variables!
All just memory we can read and write

Sequential declarations

Sequential declarations will be loaded sequentially in memory

- we can take advantage of this fact

Example 1: Splitting long strings over multiple lines

```
# help text
help:    .ascii  "The best tool ever. (v.1.0)\n"
         .ascii  "Options:\n"
         .asciiz "    --h   Print this help text.\n"
```

Example 2: Initializing an “array” of data

```
fibs:    .word   0, 1, 1, 2, 3, 5, 8, 13, 21, 35, 55, 89, 144
```

Reserving space

Reserve space in the data segment with the `.space` directive

- argument is number of **bytes** to reserve
- useful for arrays of data we don't know in advance

Example: Reserve space for a ten integer array

```
array:   .space    40
```

`array` is the address of the 0th element of the array

- address of other elements:
  `array+4`, `array+8`, `array+12`, `...`, `array+36`

(MARS demo: Decls.asm)

Reading from data memory

Basic instruction for reading data memory ("load word"):

```
lw  $t1, 4($t2)    # $t1 = Memory[$t2+4]
```

- $t2 contains the **base address**
- 4 is the **offset**

```
lw  $t1, $t2   ⇒   lw  $t1, 0($t2)
```

Macro instructions to make reading memory at labels nice:

- `lw  $t1, label      # $t1 = Memory[label]`
- `lw  $t1, label + 4  # $t1 = Memory[label+4]`

Writing to data memory

Basic instruction for writing to memory ("store word"):

```
sw  $t1, 4($t2)    # Memory[$t2+4] = $t1
```

- $t2 contains the **base address**
- 4 is the **offset**

```
sw  $t1, $t2   ⇒   sw  $t1, 0($t2)
```

Macro instructions to make writing memory at labels nice:

- `sw  $t1, label      # Memory[label] = $t1`
- `sw  $t1, label + 4  # Memory[label+4] = $t1`

(MARS demo: Add3.asm)

Sub-word addressing

Reading sub-word data

- **`lb`**: load byte (sign extend)
- **`lh`**: load halfword (sign extend)
- **`lbu`**: load byte unsigned (zero extend)
- **`lhu`**: load halfword unsigned (zero extend)

Remember, little-endian addressing:

3	2	1	0	7	6	5	4	11	10	9	8	15	14	13	12

(MARS demo: SubWord.asm)

Reading/writing data memory wrap up

Writing sub-word data

- **`sb`**: store byte (low order)
- **`sh`**: store halfword (low order)

Important: **`lw`** and **`sw`** must respect word boundaries!

- address (base+offset) must be divisible by 4

Likewise for **`lh`**, **`lhu`**, and **`sh`**

- address must be divisible by 2

Outline

Memory-mapped I/O

Problem: architecture must provide an interface to the world

- should be general (lots of potential devices)
- should be simple (RISC architecture)

Solution: Memory-mapped I/O

Memory and I/O share the same **address space**

A range of addresses are reserved for I/O:

- **input**: load from a special address
- **output**: store to a special address

So we can do I/O with just `lw` and `sw`!

(at least in embedded systems)

Role of the operating system

Usually, however:

- we don’t know (or want to know) the special addresses
- user programs don’t have permission to use them directly

Operating system (kernel)

- knows the addresses and has access to them
- provides **services** to interact with them
- services are requested through **system calls**
- (the operating system does a lot more too)

System calls

System calls are an **interface** for asking the OS to do stuff

How system calls work, from our perspective

1. `syscall` — "hey OS, I want to do something!"
2. OS checks `$v0` to see what you want to do
3. OS gets arguments from `$a0`–`$a3` (if needed)
4. OS does it
5. OS puts results in registers (if applicable)

MARS help gives a list of system call services

(MARS demo: Parrot.asm)