Compiling Techniques

Lecture 10: An Introduction to MIPS assembly

Christophe Dubach / Jose Cano

20 October 2017

Table of contents

Assembly program template

`.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

Category	Example
Comment	`# I am a comment`
Assembler directives	`.data`, `.asciiz`
Operation mnemonic	`add`, `addi`, `lw`, `bne`
Register name	`$zero`, `$t3`
Address label (declaration)	`loop1:`
Address label (use)	`loop1`
Integer constant	`8`, `-4`, `0xA9`
Character constant	`'h'`, `'\t'`
String constant	`"Hello, world\n"`

Hello world example

```
# 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

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

Registers

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

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

Arithmetic Instructions

- Most use three operands
- All operands are registered (no memory access)
- All operands are 4 bytes (a word)

Arithmetic Instructions

```
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 quantity in special register Hi to $t0:   $t0 = Hi
mflo    $t1
# move quantity in special register Lo to $t1:   $t1 = Lo

move    $t2,$t3     # $t2 = $t3
```

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

Load / Store Instructions

- 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 contents of mem location into register $t0:  $t0 = 23
li  $t1, 5    #  $t1 = 5   ("load immediate")
sw  $t1, var1 # store contents of $t1 into mem:  var1 = 5
```

Indirect and Based Addressing

- load address:
  - `la $t0, var1`
    copy memory address of var1 into register $t0
- indirect addressing:
  - `lw $t1, ($t0)`
    load word at memory address contained in $t0 into $t2
  - `sw $t2, ($t0)`
    store word in register $t2 into memory at address contained in $t0
- based/indexed addressing (useful for field access in struct):
  - `lw $t2, 4($t0)`
    load word at memory address ($t0+4) into register $t2
  - `sw $t2, -12($t0)`
    store content of register $t2 into memory at address ($t0-12)

Indirect and Based Addressing

Example

```
.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 the assembly program corresponding to the following C code:

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

void main() {
  struct point_t p;
  int arr[12];
  p.x = 2;
  p.y = 4;
  arr[3] = 6;
}
```

Control structures

- Branches:

```
b      target           # unconditional branch to target
beq    $t0,$t1,target   # branch to target if  $t0 = $t1
blt    $t0,$t1,target   # branch to target if  $t0 < $t1
ble    $t0,$t1,target   # branch to target if  $t0 <= $t1
bgt    $t0,$t1,target   # branch to target if  $t0 > $t1
bge    $t0,$t1,target   # branch to target if  $t0 >= $t1
bne    $t0,$t1,target   # branch to target if  $t0 <> $t1
```

All branch instructions use a target label: example

```
  addi $t0, $zero, 0  # t0 = 0
  addi $t1, $zero, 10 # t1 = 10
loop:
  addi $t0, $t0, 1    # t0 = t0+1
  blt  $t0, $t1, loop # branch to loop if t0<t1 (t0<10)
```

Control structures

- Jumps:

```
j        target
#  unconditional jump to program label target

jr       $t3
#  jump to address contained in $t3 ("jump register")
```

- Subroutine (function) call:

```
jal  label #  "jump and link"
```

  - copy program counter (return address) to register $ra (return address register)
  - jump to program instruction at label

```
jr $ra #  "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.

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 tables:

Service	$v0	Arguments	Result
print integer	1	$a0 = integer to print	
print string	4	$a0 = address of null-terminated string to print	
print character	11	$a0 = character to print	
read integer	5		$v0 = integer read
read character	12		$v0 = character read
allocate heap memory	9	$a0 = number of bytes to allocate	$v0 = address of allocated memory

Next lecture:

- Introduction to Code Generation