CPS 104
Computer Organization
Lecture-7 : MIPS ISA and Assembler

Sep. 17, 1999

Dietolf (Dee) Ramm
http://www.cs.duke.edu/~dr/cps104.html
Overview of Today’s Lecture:

- Review: MIPS Assembly Language Programming Conventions.
- System calls
- Simple Assembly program constructs
- Data structures in assembler.
- The Stack
- Writing Functions in Assembler.
### MIPS: Software conventions for Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>zero constant 0</td>
</tr>
<tr>
<td>1</td>
<td>at reserved for assembler</td>
</tr>
<tr>
<td>2</td>
<td>v0 expression evaluation &amp;</td>
</tr>
<tr>
<td>3</td>
<td>v1 function results</td>
</tr>
<tr>
<td>4</td>
<td>a0 arguments</td>
</tr>
<tr>
<td>5</td>
<td>a1</td>
</tr>
<tr>
<td>6</td>
<td>a2</td>
</tr>
<tr>
<td>7</td>
<td>a3</td>
</tr>
<tr>
<td>8</td>
<td>t0 temporary: caller saves</td>
</tr>
<tr>
<td>9</td>
<td>...</td>
</tr>
<tr>
<td>15</td>
<td>t7</td>
</tr>
<tr>
<td>16</td>
<td>s0 callee saves</td>
</tr>
<tr>
<td>17</td>
<td>...</td>
</tr>
<tr>
<td>23</td>
<td>s7</td>
</tr>
<tr>
<td>24</td>
<td>t8 temporary (cont’d)</td>
</tr>
<tr>
<td>25</td>
<td>t9</td>
</tr>
<tr>
<td>26</td>
<td>k0 reserved for OS kernel</td>
</tr>
<tr>
<td>27</td>
<td>k1</td>
</tr>
<tr>
<td>28</td>
<td>gp Pointer to global area</td>
</tr>
<tr>
<td>29</td>
<td>sp Stack pointer</td>
</tr>
<tr>
<td>30</td>
<td>fp frame pointer</td>
</tr>
<tr>
<td>31</td>
<td>ra Return Address (HW)</td>
</tr>
</tbody>
</table>
System call

- System call is used to communicate with the system and do simple I/O.
- Load system call code into Register $v0
- Load arguments (if any) into registers $a0, $a1 or $f12 (for floating point).
- do: syscall
- Results returned in registers $v0 or $f0.

<table>
<thead>
<tr>
<th>code</th>
<th>service</th>
<th>Arguments</th>
<th>Result</th>
<th>comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>print int</td>
<td>$a0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>print float</td>
<td>$f12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>print double</td>
<td>$f12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>print string</td>
<td>$a0</td>
<td></td>
<td>(address)</td>
</tr>
<tr>
<td>5</td>
<td>read integer</td>
<td>$a0</td>
<td>integer in $v0</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>read float</td>
<td>$f12</td>
<td>float in $f0</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>read double</td>
<td>$f12</td>
<td>double in $f0</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>read string</td>
<td>$a0=buffer, $a1=length</td>
<td>integer in $v0</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>sbrk</td>
<td>$a0=amount</td>
<td>address in $v0</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>exit</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example: Arrays and pointer arithmetic using C++

```cpp
#include <iostream.h>
int a[100]; // a is a static array
main()
{
    int *p = a; // p points to a[0]
    int k;

    for (k = 0; k < 100; k++)
    {
        *p = k; // set the entry value
        p++; // go to the next array element
    }

    cout << "entry 3 = " << a[3] << endl;
    // we will use simple I/O in Assembler
}
```
Example1: Assembler

```
.text                      # Code
.align 2
.globl main

main:                      # MAIN program entrance
  subiu $sp, 40          # \ Push the stack
  sw $ra, 36($sp)       # \ Save return address
  sw $fp, 34($sp)       #
  • • •
  la $t0, a             # p = a ; a static array
  sw $t0, 32($sp)       # store the address
  move $t1, $0          # k=0
  L1: sw $t1, 0($t2)     # *p = k
  addiu $t0, $t0, 4     # p++
  addi $t1, $t1, 1      # k++
  blt $t1, 100, L1      # if (k < 100) go to L1
  la $a0, str           # load string address
  li $v0, 4             #
  syscall               # Print a string
  lw $t0, 32($sp)       # $t0 = a
  lw $ao, 12($t0)       # $a0 = a[3]
  li $v0, 1             #
  syscall               # print integer
  • • •
```
Example-2: Array access via index

#include <iostream.h>
int a[100]; // a is a static array
main()
{
  int k;

  for (k = 0; k < 100; k++)
    a[k] = k; // set the array values

  cout << "entry 3 = " << a[3] << endl;
  // we will use simple I/O in Assembler
}
Example-2: arrays index in assembler

```assembly
.text                     # Code
.align 2
.globl main
main:                      # MAIN procedure Entrance
    subiu $sp, 40          #\ Push the stack
    sw $ra, 36($sp)       # \ Save return address
    sw $fp, 34($sp)       #
    la $t0, a            # $t0 = a
    sw $t0, 32($sp)       # store a’s address
    move $t1, $0          # k=0
L1: mul $t2, $t1, 4        # $t2 = 4*k
    addu $t3, $t0, $t2    # $t3 = a + 4*k
    sw $t1, 0($t3)        # a[k] = k
    addi $t1, $t1, 1      # k++
    blt $t1, 100, L1       # if (k < 100) go to L1
    la     $a0, str
    li     $v0, 4          # \
    syscall             # >  Print a string
    lw     $t0, 32($sp)    # $t0 = a
    lw     $ao, 12($t0)    # $a0 = a[3]
    li     $v0, 1          #
    syscall             # print integer
• • •

CPS104 MIPS_Assembler.9
```
Example-3: if ( cond.) {⋯⋯}

The C++ code

```cpp
If(a < b){
    temp = a;
    a = b;
    b = temp;
}
a = a + 5;
⋯⋯
```

Assembler code

```
# assume $t1 = a; $t2=b; $t3=temp
ble $t2, $t1, L1 # if b <= a goto L1
move $t3, $t1      # temp = a
move $t1, $t2      # a = b
move $t2, $t3      # b = temp
L1: addi $t1, $t1, 5 # a += 5
```

©GK&DR fall 1999
Calls: Why Are Stacks So Great?

Stacking of Subroutine Calls & Returns and Environments:

Some machines provide a memory stack as part of the architecture (e.g., VAX)

Sometimes stacks are implemented via software convention (e.g., MIPS)
Procedure Call (Stack) Frame

- Procedures use a frame in the stack to:
  - Hold values passed to procedures as arguments.
  - Save registers that a procedure may modify, but which the procedure’s caller does not want changed. (ex: $s0 - $s7)
  - Save the procedure return address ($ra), and frame pointer ($fp)
  - Provide space for local variables (variables with local scope)
  - Used to evaluate complex expressions.

- There are two special registers $sp and $fp that are used as special data reference
  - The stack pointer $sp points to the top of the stack.
  - The $fp points to the frame beginning.
Call-Return Linkage: Stack Frames

- Reference args and local variables at fixed offset from FP
- Grows and shrinks during expression evaluation

- Many variations on stacks possible (up/down, last pushed / next)
- Block structured languages contain link to lexically enclosing frame
- Compilers normally keep scalar variables in registers, not memory!
MIPS/GCC Procedure Calling Conventions

Calling Procedure:

- **Step-1:** Pass the arguments:
  - The first four arguments are passed in registers $a0–a3$
  - Remaining arguments are pushed into the stack
    - (in reversed order arg5 is at the top of the stack).

- **Step-2:** Save caller-saved registers
  - Save registers $t0–t9$ if they contain live values at the call site.

- **Step-3:** Execute a jal instruction.
MIPS/GCC Procedure Calling Conventions (cont.)

Called Routine

- **Step-1:** Establish stack frame.
  - Subtract the frame size from the stack pointer.
    ```
    subiu $sp, $sp, <frame-size>
    ```
  - Typically, minimum frame size is 32 bytes (8 words).

- **Step-2:** Save *callee* saved registers in the frame.
  - Register $fp is always saved.
  - Register $ra is saved if routine makes a call.
  - Registers $a0–$a3 are saved if they are changed.
  - Registers $s0–$s7 are saved if they are used.

- **Step-3:** Establish Frame pointer $fp
  - Add the stack <frame size -4> to the address in $sp
    ```
    addiu $fp, $sp, <frame-size> - 4
    ```
MIPS/GCC Procedure Calling Conventions (cont.)

On return from a call

- **Step-1:** Put returned values in registers $v0, [$v1].
  (if values are returned)
- **Step-2:** Restore callee-saved registers.
  - Restore $fp and other saved registers. [$ra, $s0 – $s7]
- **Step-3:** Pop the stack
  - Add the frame size to $sp.
    \[\text{addiu } \$sp, \$sp, \text{<frame-size>}\]
- **Step-4:** Return
  - Jump to the address in $ra.
    \[\text{jr } \$ra\]
MIPS / GCC Calling Conventions

act:
    subiu $sp, $sp, 32
    sw $ra, 20($sp)
    sw $fp, 16($sp)
    f addiu $fp, $sp, 28
    ...
    sw $a0, 0($fp)
    ...
    lw $ra, 20($sp)
    lw $fp, 16($sp)
    addiu $sp, $sp, 32
    jr $ra

First four arguments are passed in registers.
Example: Factorial

```c
main()
{
    printf("The factorial of 10 is \%d\n", fact(10));
}

int fact (int n)
{
    if (n < 1) return(1);
    return (n * fact (n-1));
}
```
.text
.global main
main:
 subiu    $sp, $sp, 32    # stack frame size is 32 bytes
 sw       $ra,20($sp)     # save return address
 sw       $fp,16($sp)     # save frame pointer
 addu     $fp, $sp,32    # set frame pointer
 li        $a0,10        # load argument (10) in $a0
 jal      fact          # call fact
 la       $a0 LC        # load string address in $a0
 move     $a1,$v0       # load fact result in $a1
 jal      printf        # call printf
 lw       $ra,20($sp)   # restore $sp
 lw       $fp,16($sp)   # restore $fp
 addu     $sp, $sp,32   # pop the stack
 jr        $ra          # exit()

.rdata
LC:
 .asciiz "The factorial of 10 is %d\n"
.text

fact:

    subiu $sp,$sp,32      # stack frame is 32 bytes
    sw $ra,20($sp)        #save return address
    sw $fp,16($sp)        #save frame pointer
    addiu $fp, $sp,28     # set frame pointer

    sw $a0,0($fp)         # save argument(n)
    lw $v0,0($fp)         # load n
    bgtz $v0, L2          # if n>0 go to $L2
    li $v0, 1             #
    j L1                  # return(1)

L2:

    lw $v1, 0($fp)        # load n
    sub $v0,$v1,1         # compute n-1
    move $a0,$v0          # load argument (n-1) in $a0
    jal fact              # call fact
    lw $v1,0($fp)         # load n
    mul $v0,$v0,$v1       # fact(n-1)*n

L1:

    lw $ra,20($sp)        # restore $ra
    lw $fp, 16($sp)       # restore $fp
    addiu $sp,$sp,32      # pop the stack
    jr $ra                 #return
Example: Factorial

Stack

<table>
<thead>
<tr>
<th>Old $ra</th>
<th>Old $fp</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Old $ra</td>
<td>Old $fp</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Old $ra</td>
<td>Old $fp</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Old $ra</td>
<td>Old $fp</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Old $ra</td>
<td>Old $fp</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Old $ra</td>
<td>Old $fp</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Old $ra</td>
<td>Old $fp</td>
</tr>
</tbody>
</table>

Main

fact(10)

fact(9)

fact(8)

fact(7)

fact(6)

fact(5)

Stack grows