Procedure Calls, SPIM, Other ISAs

CPS 104
Lecture 7

Administrivia

- Homework #2, Due Thursday Feb 6
- Software Project: Simulate MIPS
- Friday office hours canceled

Outline
- Review
- Procedure calls
- Recursion
- powerPC, Intel 80x86 ISA

Reading
- Appendix A

Next week
- Logic Design
- Reading Appendix B.1-B.3
Review: The C / C++ code

#include <iostream.h>

int main ( )
{
    int i;
    int sum = 0;
    for(i=0; i <= 100; i++)
        sum = sum + i*i ;
    cout << "The answer is " << sum << endl;
}

Let's write the assembly ... :)

Review: Assembly Language Example 1

.text
.align 2
main:
    move $14, $0 # i = 0
    move $15, $0 # tmp = 0
    move $16, $0 # sum = 0
loop:
    mul $15, $14, $14 # i*i
    add $16, $16, $15 # sum+i*i
    add $14, $14, 1 # i++
    ble $14, 100, loop # i < 100

    go print answer
    exit
Review: System Call Instruction

- System call is used to communicate with the operating system, and request services (memory allocation, I/O)
- **SPIM supports “system-call-like”**
- 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.

### SPIM System Call Support

<table>
<thead>
<tr>
<th>Code</th>
<th>Service</th>
<th>Arguments</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>print</td>
<td>int</td>
<td>$a0</td>
</tr>
<tr>
<td>2</td>
<td>print</td>
<td>float</td>
<td>$f12</td>
</tr>
<tr>
<td>3</td>
<td>print</td>
<td>double</td>
<td>$f12</td>
</tr>
<tr>
<td>4</td>
<td>print</td>
<td>string</td>
<td>$a0 (string address)</td>
</tr>
<tr>
<td>5</td>
<td>read</td>
<td>integer</td>
<td>integer in $v0</td>
</tr>
<tr>
<td>6</td>
<td>read</td>
<td>float</td>
<td>float in $f0</td>
</tr>
<tr>
<td>7</td>
<td>read</td>
<td>double</td>
<td>double in $f0</td>
</tr>
<tr>
<td>8</td>
<td>read</td>
<td>string</td>
<td>$a0=buffer, $a1=length</td>
</tr>
<tr>
<td>9</td>
<td>sbrk</td>
<td>$a0=amount</td>
<td>address in $v0</td>
</tr>
<tr>
<td>10</td>
<td>exit</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Review: Example 2

Task: sum together the integers stored in memory

```assembly
.text # Code
  .align 2 # align on word boundary
  .globl main # declare main
main: # MAIN procedure Entrance

# fill in what goes here

data # Start of data segment
  list: .word 35, 16, 42, 19, 55, 91, 24, 61, 53
  msg: .asciiz "The sum is 
  nln: .asciiz "\n"
```

Review: Procedure Call and Return

```c
int equal(int a1, int a2) {
  int tsame;
  tsame = 0;
  if (a1 == a2)
    tsame = 1;
  return(tsame);
}
main()
{
  int x,y,same;
  x = 43;
y = 2;
same = equal(x,y);
  // other computation
```

```assembly

text
  .align 2 # align on word boundary
  .globl main # declare main
main: # MAIN procedure Entrance

# fill in what goes here

data # Start of data segment
  list: .word 35, 16, 42, 19, 55, 91, 24, 61, 53
  msg: .asciiz "The sum is 
  nln: .asciiz "\n"
```
Procedure Call GAP

**ISA Level**
- call and return instructions

**C++ Level**
- Local Name Scope
  - change `tsame` to `same`
- Recursion
- Arguments and Return Value (functions)

**Assembly Level**
- Must bridge gap between HLL and ISA
- Supporting Local Names
- Passing Arguments (arbitrary number?)

Supporting Procedures

- What data structure?
**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.
  - To provide space for local variables.
    (variables with local scope)
  - To evaluate complex expressions.

**Call-Return Linkage: Stack Frames**

- High Mem
  - Arguments and local variables at fixed offset from FP
- Low Mem
  - Grows and shrinks during expression evaluation
Review: The C / C++ code

```c
#include <stdio>

int main ( )
{
    int i;
    int sum = 0;
    for(i=0; i <= 100; i++)
        sum = sum + i*i ;
    cout << "The answer is " << sum << endl;
}
```

Already did iterative version, recursive version...

Procedure Call GAP

**ISA Level**
- call and return instructions

**C++ Level**
- Local Name Scope
  - change tsame to same
- Recursion
- Arguments and Return Value (functions)

**Assembly Level**
- Must bridge gap between HLL and ISA
- Supporting Local Names
- Passing Arguments (arbitrary number?)
MIPS Register Naming Conventions

<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>10</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
</tr>
<tr>
<td>14</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>18</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td></td>
</tr>
<tr>
<td>22</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>

MIPS/GCC Procedure Calling Conventions

**Calling Procedure**

- **Step-1:** Pass the arguments:
  - The first four arguments (arg0-arg3) are passed in registers $a0$-$a3$
  - Remaining arguments are pushed onto the stack
    (in reverse 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.
    ```assembly
    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 $s0-$s7 are saved if they are used.
- **Step-3:** Establish Frame pointer
  - Add the stack <frame size> - 4 to the address in $sp
    ```assembly
    addiu $fp, $sp, <frame-size> - 4
    ```

**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.
    ```assembly
    addiu $sp, $sp, <frame-size>
    ```
- **Step-4:** Return
  - Jump to the address in $ra.
    ```assembly
    jr $ra
    ```
Review: Example2

Task: sum together the integers stored in memory

.text  # Code
.align 2  # align on word boundary
.globl main  # declare main
main:
# fill in what goes here

.data  # Start of data segment
list: .word 35, 16, 42, 19, 55, 91, 24, 61, 53
msg: .asciiz "The sum is 
nl: .asciiz "\n"

Recursive Example

.align 2
.text
foo:
  subu $sp, $sp, 32  # stack frame 32 bytes
  sw $ra, 20($sp)  # save return address
  sw $fp, 16($sp)  # save old frame pointer
  addu $fp, $fp, 16  # set up new frame pointer
  move $v0, $zero  # clear return value
  slti $t0, $a0, 101  # if > 100 do not recurse
  beq $t0, $zero, done  # if > 100 do not recurse
  mul $t1, $a0, $a0  # i*i
  sw $t1, 12($sp)  # save multiply
  addi $a0, $a0, 1  # i++
  jal foo  # recurse
  lw $t1, 12($sp)  # restore multiply
  add $v0, $v0, $t1  # sum += i*i

done:
  lw $fp, 16($sp)  # restore old frame ptr
  lw $ra, 20($sp)  # restore return address
  addu $sp, $sp, 32  # remove frame
  j $ra
Recursive Example (Continued)

.text
.globl main
main:
    subu $sp, $sp, 32    # stack frame 32 bytes
    sw $ra, 20($sp)    # save return address
    sw $fp, 16($sp)    # save old frame pointer
    addu $fp, $fp, 16    # set up new frame pointer
    li $v0, 4          # 4 = print string
    la $a0, mystr      # load address of string
    syscall                 # print string
    li $a0, 0          # initialize $i
    jal foo             # call foo
    move $a0, $v0        # put result in $a0
    li $v0, 1          # 1 = print int
    syscall                 # print int
    li $v0, 4          # 4 = print string
    la $a0, endl       # load address of endl
    syscall                 # print string
    lw $fp, 16($sp)    # restore old frame ptr
    lw $ra, 20($sp)    # restore return address
    addu $sp, $sp, 32    # remove frame
    j $ra

.data

mystr: .asciiz "The answer is 
endl: .asciiz "\n"

• This is on course web page (code...)
Example 2

# Example for CPS 104
# Program to add together array of 9 numbers.
.text                   # Code
.align 2
.globl main
main:                           # MAIN procedure Entrance
    subu $sp, 40         #\ Push the stack
    sw      $ra, 36($sp) # \ Save return address
    sw      $s3, 32($sp) # \ > Entry Housekeeping
    sw      $s2, 28($sp) # / save registers on stack
    sw      $s1, 24($sp) # / move $v0, $0 #/ initialize exit code to 0
    move    $s1, $0       #\
    la      $s0, list    # \ Initialization
    la      $s2, msg     # /
    la      $s3, list+36 #/

# Main code segment
again:                          # Begin main loop
    lw      $t6, 0($s0)    #\ Actual "work"
    addu    $s1, $s1, $t6 # SPIM I/O
    li      $v0, 4        #\
    move    $a0, $s2      # > Print a string
    syscall             #/
    li      $v0, 1        #\
    move    $a0, $s1      # > Print a number
    syscall             #/
    li      $v0, 4        #\
    la      $a0, nln      # > Print a string (eol)
    syscall             #/
    addu    $s0, $s0, 4   #\ index update and
    bne     $s0, $s3, again #/ end of loop
Example 2 (Continued)

```assembly
  # Exit Code
  move $v0, $0  #
  lw  $s0, 20($sp)  # \
  lw  $s1, 24($sp)  # \  
  lw  $s2, 28($sp)  # \ Closing Housekeeping 
  lw  $s3, 32($sp)  # / restore registers 
  lw  $ra, 36($sp)  # / load return address 
  addu $sp, 40  # / Pop the stack 
  jr  $ra  #/ exit(0) ;
  .end main  # end of program 

  # Data Segment
  .data       # Start of data segment
  list:       .word   35, 16, 42, 19, 55, 91, 24, 61, 53
  msg:        .asciiz "The sum is 
  nln:        .asciiz "\n"
```

Example 2

- Let’s write it recursively
Details of the MIPS instruction set

- Register zero always has the value zero
  - even if you try to write it
- Branch and jump instructions put the return address PC+4 into the link register
- All instructions change all 32 bits of the destination register (lui, lb, lh) and read all 32 bits of sources (add, sub, and, or, …)
- Immediate arithmetic and logical instructions are extended as follows:
  - logical immediate: zero extended to 32 bits
  - arithmetic immediate: sign extended to 32 bits
- lb and lh extend data as follows:
  - lbu, lhu are zero extended
  - lb, lh are sign extended

Miscellaneous MIPS Instructions

break  A breakpoint trap occurs, transfers control to exception handler
syscall A system trap occurs, transfers control to operating system
coprocessor instrs Support for floating point.
TLB instructions Support for virtual memory: discussed later
restore from exception Restores previous interrupt mask & kernel/user mode bits into status register
load word left/right Supports unaligned word loads
store word left/right Supports unaligned word stores
PowerPC ISA

• Very similar to MIPS
• Indexed Addressing (register+register)
  \[ \text{lw} \; \$t1, \; \$a0, \; \$s3 \; \Rightarrow \; \$t1 = \text{mem}[\$a0+\$s3] \]
• Update Addressing
  \[ \text{lw} \; \$t1, \; 4(\$a0) \; \Rightarrow \; \$t1 = \text{mem}[\$a0+4]; \; \$a0 += 4; \]
• Load/Store Multiple
• Counter Register
  \[ \text{bc loop, ctr} \neq 0 \]
  \[ \text{decrement ctr, if ctr} \neq 0 \text{ go to loop} \]

Intel 80x86 ISA

• Long history
• Binary compatibility
• 1978: 8086, 16-bit, registers have dedicated uses
• 1980: 8087, added floating point (stack)
• 1982: 80286, 24-bit
• 1985: 80386, 32-bit, new insts -> GPR almost
• 1989-95: 80486, Pentium, Pentium II
• 1997: Added MMX
• 1999: Pentium III
• 2002: Pentium 4
80x86 Registers and Addressing Modes

- eight 32-bit GPRs
  - EAX, ECX, EDX, EBX, ESP, EBP, ESI,EDI
- six 16-bit Registers for code, stack, & data
- 2 address ISA
  - one operand is both source and destination
- Not Load/Store
  - One operand can be in memory

80x86 Addressing Modes

- Register Indirect
  - mem[reg]
  - not ESP or EBP
- Base + displacement (8 or 32 bit)
  - mem[reg + const]
  - not ESP or EBP
- Base + scaled Index
  - mem[reg + (2^{scale} x index)]
  - scale = 0,1,2,3
  - base any GPR, Index not ESP
- Base + scaled Index + displacement
  - mem[reg + (2^{scale} x index) + displacement]
  - scale = 0,1,2,3
  - base any GPR, Index not ESP
Condition Codes

- Both PowerPC and x86 ISA have condition codes
- Special HW register, that has values set as side effect of instruction execution
- Example conditions
  - Zero
  - Negative
- Example use
  - subi $t0, $t0, 1
  - bz loop

80x86 Instruction Encoding

- Variable Size 1-byte to 17-bytes
- Jump (JE) 2-bytes
- Push 1-byte
- Add Immediate 5-bytes
- W bit says 32-bits or 8-bits
- D bit indicates direction
  - memory -> reg or reg -> memory
  - movw EBX, [EDI + 45]
  - movw [EDI + 45], EBX
Summary

- Procedure calls
- SPIM
- powerPC, Intel 80x86 ISA

Next Time
- Boolean Algebra, Logic Gates

Reading
- Appendix B