Today’s Lecture

- Homework #1 due today
- Homework #2 due Tuesday September 24

Outline
- Review
- Assembly Programming

Reading
  Chapter 3, Appendix A
Review: A Program

```c
#include <iostream.h>

main()
{
    int *a = new int[100];
    int *p = a;
    int k;
    for (k = 0; k < 100; k++)
    {
        *p = k;
        p++;
    }
    cout << "entry 3 = " << a[3] << endl;
}
```

Review: Stored Program Computer

- **Instructions**: a fixed set of built-in operations

- Instructions and data are stored in the (same) computer memory.

- **Fetch Execute Cycle**
  ```c
  while (!done)
  {
    fetch instruction
    execute instruction
  }
  ```
Review: What Must be Specified?

- Instruction Format
  - how do we tell what operation to perform?
- Location of operands and result
  - where other than memory?
  - how many explicit operands?
  - how are memory operands located?
  - which can or cannot be in memory?
- Data type and Size
- Operations
  - what are supported
- Successor instruction
  - jumps, conditions, branches
- fetch-decode-execute is implicit!

Review: MIPS ISA Categories

- Arithmetic
  - add, sub, mul, etc
- Logical
  - AND, OR, SHIFT
- Data Transfer
  - load, store
  - MIPS is LOAD/STORE architecture
- Conditional Branch
  - implement if, for, while... statements
- Unconditional Jump
  - support method invocation (procedure calls)
Review: MIPS Instruction Formats

R-type: Register-Register

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>Rs</td>
<td>Rt</td>
<td>Rd</td>
<td>shamt</td>
<td>func</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

I-type: Register-Immediate

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>Rs</td>
<td>Rt</td>
<td>immediate</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

J-type: Jump / Call

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>target</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Terminology

Op = opcode
Rs, Rt, Rd = register specifier

Assembler and Assembly Language

- **Machine language** is a sequence of binary words.
- **Assembly language** is a text representation for machine language plus extras that make assembly language programming easier (more readable too!).
MIPS Assembly Language

- One instruction per line.
- **Numbers** are base-10 integers or Hex.
- **Identifiers**: alphanumeric, _, . string starting in a letter or _
- **Labels**: identifiers starting at the beginning of a line followed by "::
- **Comments**: everything following # till end-of-line.
- Instruction format: Space and ",” separated fields.
  - `[Label:]` <op> reg1, [reg2], [reg3]  [# comment]
  - `[Label:]` <op> reg1, offset(reg2)  [# comment]
  - `.Directive` [arg1], [arg2], ...

Assembly Language (cont.)

- **Pseudo-instructions**: extend the instruction set for convenience
- **Examples**
  - `move $2, $4`  # $2 = $4, (copy $4 to $2)
    Translates to:
    ```
    add $2, $4, $0
    ```
  - `li $8, 40`  # $8 = 40, (load 40 into $8)
    ```
    addi $8, $0, 40
    ```
  - `sd $4, 0($29)`  # mem[$29] = $4; Mem[$29+4] = $5
    ```
    sw $4, 0 ($29)
    sw $5, 4 ($29)
    ```
  - `la $4, 0x1000056c`  # Load address $4 = <address>
    ```
    lui $4, 0x1000
    ori $4, $4, 0x056c
    ```
Assembly Language (cont.)

- **Directives**: tell the assembler what to do...
- **Format “.”<string> [arg1], [arg2]...**

**Examples**

- `.align n`  # align data or text on $2^n$ byte boundary.
- `.ascii <string>`  # store a string in memory.
- `.asciz <string>`  # store a null terminated string in memory
- `.data [address]`  # start a data segment.
- `.text [address]`  # start a code segment.
- `.word w1, w2, . . . , wn`  # store n words in memory.

---

A Simple Program

- **Add two numbers x & y together**

```
.text  # declare text segment
.align 2  # align it on 4-byte boundary
main:  # label for main
    la $3, x  # load address of x into R3 (pseudo-inst)
    lw $4, 0($3)  # load value of x into R4
    la $3, y  # load address of y into R3 (pseudo-inst)
    lw $5, 0($3)  # load value of y into R5
    add $6, $4, $5  # compute x+y
    jr $31  # return to calling routine

.data  # declare data segment
x:.word 10  # initialize x to 10
y:.word 3  # initialize y to 3
```
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;
}

Let's write the assembly ... :)
```

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
}
```

<table>
<thead>
<tr>
<th>PC</th>
<th>$31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x10000</td>
<td>??</td>
</tr>
<tr>
<td>0x10004</td>
<td>addi $1, $0, 43</td>
</tr>
<tr>
<td>0x10008</td>
<td>addi $2, $0, 2</td>
</tr>
<tr>
<td>0x1000c</td>
<td>jal 0x30408</td>
</tr>
<tr>
<td>0x30408</td>
<td>addi $3, $0, 0</td>
</tr>
<tr>
<td>0x3040c</td>
<td>bne $1, $2, 8</td>
</tr>
<tr>
<td>0x30410</td>
<td>addi $3, $0, 1</td>
</tr>
<tr>
<td>0x30414</td>
<td>jr $31</td>
</tr>
</tbody>
</table>
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
MIPS Register Naming Conventions

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></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.)

**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 $s0-$s7 are saved if they are used.

- **Step-3: Establish frame pointer**
  - Add the stack <frame size> - 4 to the address in $sp
    - `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.
    - `addiu $sp, $sp, <frame-size>`

- **Step-4: Return**
  - Jump to the address in $ra.
    - `jr $ra`
MIPS / GCC Calling Conventions

```
fact:
  subiu $sp, $sp, 32
  sw $ra, 20($sp)
  sw $fp, 16($sp)
  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

```
Stack
Old $ra
Old $fp
Old $ra
Old $fp
Old $a0
Old $ra
Old $fp
Old $a0
Old $ra
Old $fp
Old $a0
Old $ra
Old $fp
Old $a0
Old $ra
Old $fp
Old $a0

Main
fact(10)
fact(9)
fact(8)
fact(7)
fact(6)
```

Stack grows
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>
Echo number and string

.text
main:
  li $v0, 5  # code to read an integer
  syscall  # do the read (invokes the OS)
  move $a0, $v0  # copy result from v0 to a0

  li $v0, 1  # code to print an integer
  syscall  # print the integer

  li $v0, 4  # code to print string
  la $a0, nln  # address of string (newline)
  syscall

.data
.align 2
name: .word 0,0
nln: .asciiz \n"
Example2

# Example for CPS 104
# Program to add together list 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     #/

Example2 (cont.)

# Main code segment
again:              # Begin main loop
    lw $t6, 0($s0)      #\ "work"
    addu $s1, $s1, $t6 #/ SPIM I/O
    li $v0, 4           #\ "Print a string"
    move $a0, $s2       #> Print a number
    syscall             #/
    li $v0, 1           #\ "Print a string (eol)"
    move $a0, $s1       #> end of loop
    syscall             #/
    addu $s0, $s0, 4    #\ index update and
    bne $s0, $s3, again #/
Example2 (cont.)

```assembly
# Exit Code
move $v0, $0          #
lw $s0, 20($sp)       # \ Closing Housekeeping
lw $s1, 24($sp)       # 
lw $s2, 28($sp)       # / restore registers
lw $s3, 32($sp)       #   / load return address
lw $ra, 36($sp)       # / 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 

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 immediates are zero extended to 32 bits
  arithmetic immediates are 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 exception handler

**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

Summary

- Assembler Translates Assembly to Machine code
- Pseudo Instructions
- System Call
- Procedure Calls

Next Time

- Other Instruction Sets
- PowerPC, Intel x86

Reading

- Ch. 3, Appendix A