CS 61C: Great Ideas in Computer Architecture (Machine Structures)  
More MIPS Machine Language

Instructors:  
Randy H. Katz  
David A. Patterson  
http://inst.eecs.Berkeley.edu/~cs61c/sp11

Levels of Representation/Interpretation

<table>
<thead>
<tr>
<th>High Level Language Program (e.g., C)</th>
<th>Compiler</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assembly Language Program (e.g., MIPS)</td>
<td>Assembly</td>
</tr>
<tr>
<td>Machine Language Program (MIPS)</td>
<td>Machine Interpretation</td>
</tr>
</tbody>
</table>

- **High Level Language Program**: High level program in e.g., C
- **Compiler**: Translates high level language to assembly language
- **Assembly Language Program**: Assembly language program in e.g., MIPS
- **Machine Language Program (MIPS)**: Machine language program

Temp = V[k];  
V[k] = V[k+1];  
V[k+1] = Temp.

Agenda

- **String Copy Example**
- **Administrivia**
- **Functions**
- **Technology Break**
- **Memory Heap**
- **Summary**

Fast String Copy Code in C

```c
char *p, *q;
p = &x[0]; /* p = x */
/* set p to address of 1st char of x */
q = &y[0]; /* q = y also OK */
/* set q to address of 1st char of y */
while((*q++ = *p++) != \0) ;
```
Fast String Copy in MIPS Assembly

Get addresses of x and y into $s1, $s2
p and q are assigned to these registers

\[
\begin{align*}
&\text{Loop:} \\
&\text{j Loop} \\
&\text{Exit:} \# \text{N characters} => \#N*6 + 3 \text{ instructions}
\end{align*}
\]

Spring 2011 - Lecture #6
Student Example
Fast String Copy in MIPS Assembly

Get addresses of x and y into $s1, $s2

- $p$ and $q$ are assigned to these registers
  
  `lw $t1, Base Address (e.g., BA)`
  `lw $s1,0($t1)`  # $s1 = p`
  `lw $s2,4($t1)`  # $s2 = q`

- Loop:
  `lb $t2,0($s1)`  # $t2 = *p`
  `sb $t2,0($s2)`  # $q = $t2`
  `addi $s1,$s1,1`  # $q = q + 1`
  `beq $t2,$zero, Exit`  # if *p == 0, go to Exit
  `j Loop`  # go to Loop

Exit:  # N characters => N*6 + 3 instructions

Assembler Pseudo-instructions

- Load Address (asm temp regs): $at = Label Address
  
  ```
  la $at, LabelAddr
  ```
  
  Implemented as:
  ```
  lui $at, LabelAddr[31:16];
  ori $at, $at, LabelAddr[15:0]
  ```

Agenda

- String Copy Example
- Administrivia
- Functions
- Technology Break
- Memory Heap
- Summary

Administrivia

- This week in lab and homework:
  - Lab #3 EC2 (TAs say it is more doable than last week)
    - Note: labs graded on a scale of 0 to 2
  - HW #3 Posted
    - Note: HWs graded on a scale from 0 to 3
  - Project #1 posted
    - Note: Intermediate checkpoint due Sunday
    - 12 students have already checked it in!
    - 2 students have already completed Project #1!
- CSUA announcement
Intel Identifies Chipset Design Error, Implementing Solution

Updates Outlook to Incorporate Effects of Error, Infineon Acquisition and Expected McAfee Acquisition

- Chipset circuit design issue identified, fix implemented, customers being notified
- Infineon Technologies AC Wireless Solutions business (MSL) acquisition closed Jan. 31
- McAfee, Inc. (MVF) acquisition expected to close by the end of the first quarter
- Fourth-quarter, third-quarter and 54-year outlook revised to reflect impact of chipset issue, MVL closure, expected MVE closure by the end of the first quarter

SANTA CLARA, Calif., Jan. 31, 2011 – As part of ongoing quality assurance, Intel Corporation has discovered a design issue in a recently released support chip, the Intel 6 Series, code-named Cougar Point, and has implemented a silicon fix. In some cases, the Serial ATA (SATA) ports within the chipset may degrade over time, potentially impacting the performance or functionality of SATA-linked devices such as hard disk drives and DVD-drives. The chipset is utilized in PCs with Intel’s latest Second Generation Intel Core processors, code-named Sandy Bridge. Intel has stepped shipments of the affected support chip from its factories, Intel has also reached out to affected customers to alert them to the matter and where possible to remove the issue. The Sandy Bridge microarchitecture is unaffected and no other products are affected by this issue.

CS61c In the News

Agenda

- String Copy Example
- Administrivia
- Functions
- Technology Break
- Memory Heap
- Summary

MIPS Registers

Assembly Language Conventions

- $t0-$t9: 10 x temporaries (intermediates)
- $s0-$s7: 8 x “saved” temporaries (program variables)
- $at: 1 x assembler temporary (used by pseudo instructions)
- 19 registers ... what are the other 13 registers used for, by convention?

Six Fundamental Steps in Calling a Function

1. Put parameters in a place where function can access them
2. Transfer control to function
3. Acquire (local) storage resources needed for function
4. Perform desired task of the function
5. Put result value in a place where calling program can access it and restore any registers you used
6. Return control to point of origin, since a function can be called from several points in a program
MIPS Function Call Conventions

- Registers way faster than memory, so use registers
- $a0–a3$: four argument registers to pass parameters
- $v0–v1$: two value registers to return values
- $ra$: one return address register to return to the point of origin
- (7 of 13, 6 left!)

MIPS Function Call Instructions

- Invoke function: jump and link instruction (jal)
  - "link" means form an address or link that points to calling site to allow function to return to proper address
  - Jumps to address and simultaneously saves the address of following instruction in register $ra
  - Return from function: jump register instruction (jr)
    - Unconditional jump to address specified in register
    - jr $ra

Notes on Functions

- Calling program (caller) puts parameters into registers $a0–$a3 and uses jal X to invoke X (callee)
- Must have register in computer with address of currently executing instruction
  - Instead of Instruction Address Register (better name), historically called Program Counter (PC)
  - It's a program's counter, it doesn't count programs!
- jr $ra puts address inside $ra into PC
- What value does jal X place into $ra? ????

Where Save Old Registers Values to Restore Them After Function Call

- Need a place to place old values before call function, restore them when return, and delete
- Ideal is stack: last-in-first-out queue (e.g., stack of plates)
  - Push: placing data onto stack
  - Pop: removing data from stack
- Stack in memory, so need register to point to it
- $sp$ is the stack pointer in MIPS
- Convention is grow from high to low addresses
  - Push decrements $sp$, Pop increments $sp$
- (8 of 13, 5 left!)

Example

```c
int leaf_example
(int g, int h, int i, int j)
{
    int f;
    f = (g + h) - (i + j);
    return f;
}
```

- Parameter variables $g, h, i, j$ in argument registers $a0, a1, a2, a3$, and $f$ in $s0$
- Assume need one temporary register $t0$
Stack Before, During, After Function

- Need to save old values of $s0 and $t0

MIPS Code for leaf_example

leaf_example:
```
addi $sp, $sp, -8  # adjust stack for 2 int items
sw $t0, 4($sp)  # save $t0 for use afterwards
sw $s0, 0($sp)  # save $s0 for use afterwards
f = g + h
# $t0 = i + j
# return value (g + h) – (i + j)
# restore register $s0 for caller
# restore register $t0 for caller
# adjust stack to delete 2 items
# jump back to calling routine
```

MIPS Code for leaf_example

leaf_example:
```
addi $sp, $sp, -8  # adjust stack for 2 int items
sw $t0, 4($sp)  # save $t0 for use afterwards
sw $s0, 0($sp)  # save $s0 for use afterwards
add $s0, $a0, $a1  # f = g + h
# $t0 = i + j
# return value (g + h) – (i + j)
# restore register $s0 for caller
# restore register $t0 for caller
# adjust stack to delete 2 items
# jump back to calling routine
```

MIPS Code for leaf_example

leaf_example:
```
addi $sp, $sp, -8  # adjust stack for 2 int items
sw $t0, 4($sp)  # save $t0 for use afterwards
sw $s0, 0($sp)  # save $s0 for use afterwards
f = g + h
# $t0 = i + j
# return value (g + h) – (i + j)
# restore register $s0 for caller
# restore register $t0 for caller
# adjust stack to delete 2 items
# jump back to calling routine
```

MIPS Code for leaf_example

leaf_example:
```
addi $sp, $sp, -8  # adjust stack for 2 int items
sw $t0, 4($sp)  # save $t0 for use afterwards
sw $s0, 0($sp)  # save $s0 for use afterwards
add $s0, $a0, $a1  # f = g + h
# $t0 = i + j
# return value (g + h) – (i + j)
# restore register $s0 for caller
# restore register $t0 for caller
# adjust stack to delete 2 items
# jump back to calling routine
```
MIPS Code for leaf_example

```
leaf_example:
addi $sp,$sp, -8      # adjust stack for 2 int items
sw $t0, 4($sp)        # save $t0 for use afterwards
sw $s0, 0($sp)        # save $s0 for use afterwards
add $s0,$s0,$s1       # $f = g + h
add $t0,$s2,$s3       # $t0 = i + j
sub $v0,$s0,$t0       # return value ($g + h) - (i + j)
# restore register $s0 for caller
# restore register $t0 for caller
# adjust stack to delete 2 items
# jump back to calling routine
```

MIPS Code for leaf_example

```
leaf_example:
addi $sp,$sp, -8      # adjust stack for 2 int items
sw $t0, 4($sp)        # save $t0 for use afterwards
sw $s0, 0($sp)        # save $s0 for use afterwards
add $s0,$s0,$s1       # $f = g + h
add $t0,$s2,$s3       # $t0 = i + j
sub $v0,$s0,$t0       # return value ($g + h) - (i + j)
# restore register $s0 for caller
# restore register $t0 for caller
# adjust stack to delete 2 items
# jump back to calling routine
```

MIPS Code for leaf_example

```
leaf_example:
addi $sp,$sp, -8      # adjust stack for 2 int items
sw $t0, 4($sp)        # save $t0 for use afterwards
sw $s0, 0($sp)        # save $s0 for use afterwards
add $s0,$s0,$s1       # $f = g + h
add $t0,$s2,$s3       # $t0 = i + j
sub $v0,$s0,$t0       # return value ($g + h) - (i + j)
# restore register $s0 for caller
# restore register $t0 for caller
# adjust stack to delete 2 items
# jump back to calling routine
```

MIPS Code for leaf_example

```
leaf_example:
addi $sp,$sp, -8      # adjust stack for 2 int items
sw $t0, 4($sp)        # save $t0 for use afterwards
sw $s0, 0($sp)        # save $s0 for use afterwards
add $s0,$s0,$s1       # $f = g + h
add $t0,$s2,$s3       # $t0 = i + j
sub $v0,$s0,$t0       # return value ($g + h) - (i + j)
# restore register $s0 for caller
# restore register $t0 for caller
# adjust stack to delete 2 items
# jump back to calling routine
```

MIPS Code for leaf_example

```
leaf_example:
addi $sp,$sp, -8      # adjust stack for 2 int items
sw $t0, 4($sp)        # save $t0 for use afterwards
sw $s0, 0($sp)        # save $s0 for use afterwards
add $s0,$s0,$s1       # $f = g + h
add $t0,$s2,$s3       # $t0 = i + j
sub $v0,$s0,$t0       # return value ($g + h) - (i + j)
# restore register $s0 for caller
# restore register $t0 for caller
# adjust stack to delete 2 items
# jump back to calling routine
```

MIPS Code for leaf_example

```
leaf_example:
addi $sp,$sp, -8      # adjust stack for 2 int items
sw $t0, 4($sp)        # save $t0 for use afterwards
sw $s0, 0($sp)        # save $s0 for use afterwards
add $s0,$s0,$s1       # $f = g + h
add $t0,$s2,$s3       # $t0 = i + j
sub $v0,$s0,$t0       # return value ($g + h) - (i + j)
# restore register $s0 for caller
# restore register $t0 for caller
# adjust stack to delete 2 items
# jump back to calling routine
```

MIPS Code for leaf_example

```
leaf_example:
addi $sp,$sp, -8      # adjust stack for 2 int items
sw $t0, 4($sp)        # save $t0 for use afterwards
sw $s0, 0($sp)        # save $s0 for use afterwards
add $s0,$s0,$s1       # $f = g + h
add $t0,$s2,$s3       # $t0 = i + j
sub $v0,$s0,$t0       # return value ($g + h) - (i + j)
# restore register $s0 for caller
# restore register $t0 for caller
# adjust stack to delete 2 items
# jump back to calling routine
```
What If a Function Calls a Function?
Recursive Function Calls?
• Would clobber values in $a0 to $a1 and $ra
• What is the solution?

Recursive Function Factorial
int fact (int n)
{
    if (n < 1) return (1);
    else return (n * fact(n-1));
}

Recursive Function Factorial
Fact:
  # adjust stack for 2 items
  addi $sp,$sp,-8
  # save return address
  sw $ra, 4($sp)
  # save argument n
  sw $a0, 0($sp)
  # test for n < 1
  slti $t0,$a0,1
    # if n >= 1, go to L1
    beq $t0,$zero,L1
    # then part (n=1) return 1
    addi $v0,$zero,1
    jr $ra
    # else part (n>=1)
    # arg. gets (n-1)
    addi $a0,$a0,-1
    # pop 2 items off stack
    addi $sp,$sp,8
    # return to caller
    jr $ra
    # mul is a pseudo instruction
    mul

Agenda
• String Copy Example
• Administrivia
• Functions
• Technology Break
• Memory Heap
• Summary
Allocating Space on Stack

- C has two storage classes: automatic and static
  - Automatic variables are local to function and discarded when function exits.
  - Static variables exist across exits from and entries to procedures
- Use stack for automatic (local) variables that don’t fit in registers
- Procedure frame or activation record: segment of stack with saved registers and local variables
  - Some MIPS compilers use a frame pointer ($fp$) to point to first word of frame
- (9 of 13, 4 left!)

Stack Before, During, After Call

Peer Instruction Question

```c
static int *p;
int leaf (int g, int h, int i, int j)
{ int f; p = &f;
f = (g + h) - (i + j);
return f;
}
int main(void) { int x;
... x = leaf(1,2,3,4);
... x = leaf(3,4,1,2);
... printf("%d\n",*p);
... }
```

- What will a.out do? Red. Print -4 Orange. Print 4 Green. a.out will crash Yellow. None of the above

Peer Instruction Answer

```c
static int *p;
int leaf (int g, int h, int i, int j)
{ int f; p = &f;
f = (g + h) - (i + j);
return f;
}
int main(void) { int x;
... x = leaf(1,2,3,4);
... x = leaf(3,4,1,2);
... printf("%d\n",*p);
... }
```

- What will a.out do? Red. Print -4 Orange. Print 4 Green. a.out will crash Yellow. None of the above

Optimized Function Convention

To reduce expensive loads and stores from spilling and restoring registers, MIPS divides registers into two categories:

1. Preserved across function call
   - Caller can rely on values being unchanged
   - $ra$, $sp$, $gp$, $fp$, “saved registers” $s8$–$s7$
2. Not preserved across function call
   -Caller cannot rely on values being unchanged
   - Return value registers $s0$–$s1$, Argument registers $s2$–$s3$, “temporary registers” $s10$–$s19$

Register Numbering
Where is the Stack in Memory?

- MIPS convention
- Stack starts in high memory and grows down
  - Hexadecimal (base 16): 7ffe ffex
- MIPS programs (text segment) in low end
  - 0040 0000
- Static data segment (constants and other static variables) above text for static variables
  - MIPS convention global pointer ($gp$) points to static
  - (10 of 13, 3 left!)
- Heap above static for data structures that grow and shrink; grows up to high addresses

And in Conclusion, ...

- C is function oriented; code reuse via functions
  - Jump and link (jal) invokes, jump register (jr $ra) returns
  - Registers $a0-$a3 for arguments, $v0-$v1 for return values
- Stack for spilling registers, nested function calls, C local (automatic) variables