Great Ideas in Computer Architecture

More MIPS Machine Language

Instructor:
David A. Patterson
http://inst.eecs.Berkeley.edu/~cs61c/sp12

Big Idea #1: Levels of Representation/Interpretation

- High Level Language
  - Programs (e.g., C)

- Compiler
  - Machine Language
    - Program (e.g., MIPS)

- Machine Interpretation
  - Hardware Architecture Description
    - (e.g., block diagrams)

- Logic Circuit Description
  - (Circuit Schematic Diagrams)

Endianness

- MIPS is Big Endian
  - Most-significant byte at least address of a word
  - c.f. Little Endian: least-significant byte at least address

Little endian byte 0

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>msb</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

Big endian byte 0

- Big Endian: address of most significant byte = word address
- Little Endian: address of least significant byte = word address

- Can only tell if access same data with load/store byte and load/store word

Review

- Computer words and vocabulary are called instructions and instruction set respectively
- MIPS is example RISC instruction set in this class
- Rigid format: 1 operation, 2 source operands, 1 destination
  - add, sub, mul, div, and, or, sll, srl
  - lw, sw to move data to/from registers from/to memory
- Simple mappings from arithmetic expressions, array access, if-then-else in C to MIPS instructions

Agenda

- String Copy Example
- Administrivia
- Functions
- Technology Break
- MIPS register allocation convention
- Memory Heap
- Summary
Fast String Copy Code in C

• Copy x[] to y[]
  char *p, *q;
  p = 4x[0]; /* p = x */
  q = 4y[0]; /* q = y */
  /* set p to address of 1st char of x */
  /* 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
lw $t1, Base Address (e.g., BA)
# $s1 = p
# $s2 = q
Loop:
# $t2 = *p
# *q = $t2
# p = p + 1
# q = q + 1
j Loop
# if *p == 0, go to Exit
# go to Loop
Exit: # N characters => N*6 + 3 instructions

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)
# $s1 = p
# $s2 = q
Loop:
# $t2 = *p
# *q = $t2
# p = p + 1
# q = q + 1
j Loop
# if *p == 0, go to Exit
# go to Loop
Exit: # N characters => N*6 + 3 instructions
Fast String Copy in MIPS Assembly

Get addresses of x and y into \( \$s1, \$s2 \)

\( p \) and \( q \) are assigned to these registers

\[
\begin{align*}
\text{lw } & \$t1, \text{ Base Address (e.g., BA)} \\
\text{lw } & \$s1,0(\$t1) \\
\text{lw } & \$s2,4(\$t1) \\
\text{Loop: } & \text{lb } \$t2,0(\$s1) \\
& \text{sb } \$t2,0(\$s2) \\
& \text{addi } \$s1, \$s1,1 \\
& \text{addi } \$s2, \$s2,1 \\
& \text{j } \text{Loop} \\
\text{Exit: } & \# \text{ N characters } => \# \text{N}+3 \text{ instructions}
\end{align*}
\]

**Which statement is TRUE?**

- \( \text{char } *p, *q; \quad p = \text{aw[0]}; \quad q = \text{aw[1]}; \) \quad \text{while}(\text{\texttt{ controversia } = } *p++ != \text{\texttt{ \\
\text{□ } \$s1 \text{ corresponds to } p \\
\text{□ } \$s1 \text{ corresponds to } p \\
\text{□ } \$s1 \text{ corresponds to } q \\
\text{□ } \$s1 \text{ corresponds to } p \\
\text{□ } \$s1 \text{ corresponds to } p \\
\text{□ } \$s1 \text{ corresponds to } q \\
\text{□ } \$s1 \text{ corresponds to } p \\
\text{□ } \$s1 \text{ corresponds to } p \\
\text{□ } \$s1 \text{ corresponds to } q \\
\text{□ } \$s1 \text{ corresponds to } p \\
\text{□ } \$s1 \text{ corresponds to } p \\
\text{□ } \$s1 \text{ corresponds to } q \\
\text{□ } \$s1 \text{ corresponds to } p \\
\text{□ } \$s1 \text{ corresponds to } p \\
\text{□ } \$s1 \text{ corresponds to } q \\
\text{□ } \$s1 \text{ corresponds to } p \\
\text{□ } \$s1 \text{ corresponds to } p \\
\text{□ } \$s1 \text{ corresponds to } q \\
\text{□ } \$s1 \text{ corresponds to } p \\
\text{□ } \$s1 \text{ corresponds to } p \\
\text{□ } \$s1 \text{ corresponds to } q \\
\text{□ } \$s1 \text{ corresponds to } p \\
\text{□ } \$s1 \text{ corresponds to } p \\
\text{□ } \$s1 \text{ corresponds to } q \\
\text{□ } \$s1 \text{ corresponds to } p \\
\text{□ } \$s1 \text{ corresponds to } p \\
\text{□ } \$s1 \text{ corresponds to } q \\
\text{□ } \$s1 \text{ corresponds to } p \\
\text{□ } \$s1 \text{ corresponds to } p \\
\text{□ } \$s1 \text{ corresponds to } q \\
\text{□ } \$s1 \text{ corresponds to } p \\
\text{□ } \$s1 \text{ corresponds to } p \\
\text{□ } \$s1 \text{ corresponds to } q \\
\text{□ } \$s1 \text{ corresponds to } p \\
\text{□ } \$s1 \text{ corresponds to } p \\
\text{□ } \$s1 \text{ corresponds to } q \\
\text{□ } \$s1 \text{ corresponds to } p \\
\text{□ } \$s1 \text{ corresponds to } p \\
\text{□ } \$s1 \text{ corresponds to } q \\
\text{□ } \$s1 \text{ corresponds to } p \\
\text{□ } \$s1 \text{ corresponds to } p \\
\text{□ } \$s1 \text{ corresponds to } q \\
\text{□ } \$s1 \text{ corresponds to } p \\
\text{□ } \$s1 \text{ corresponds to } p \\
\text{□ } \$s1 \text{ corresponds to } q \\
\text{□ } \$s1 \text{ corresponds to } p \\
\text{□ } \$s1 \text{ corresponds to } p \\
\text{□ } \$s1 \text{ corresponds to } q \\
\text{□ } \$s1 \text{ corresponds to } p \\
\text{□ } \$s1 \text{ corresponds to } p \\
\text{□ } \$s1 \text{ corresponds to } q \\
\text{□ } \$s1 \text{ corresponds to } p \\
\text{□ } \$s1 \text{ corresponds to } p \\
\text{□ } \$s1 \text{ corresponds to } q 
\end{align*}
\]

**Fast String Copy in MIPS Assembly**

Get addresses of x and y into \( \$s1, \$s2 \)

\( p \) and \( q \) are assigned to these registers

\[
\begin{align*}
\text{lw } & \$t1, \text{ Base Address (e.g., BA)} \\
\text{lw } & \$s1,0(\$t1) \\
\text{lw } & \$s2,4(\$t1) \\
\text{Loop: } & \text{lb } \$t2,0(\$s1) \\
& \text{sb } \$t2,0(\$s2) \\
& \text{addi } \$s1, \$s1,1 \\
& \text{addi } \$s2, \$s2,1 \\
& \text{j } \text{Loop} \\
\text{Exit: } & \# \text{ N characters } => \# \text{N}+3 \text{ instructions}
\end{align*}
\]

**Fast String Copy in MIPS Assembly**

Get addresses of x and y into \( \$s1, \$s2 \)

\( p \) and \( q \) are assigned to these registers

\[
\begin{align*}
\text{lw } & \$t1, \text{ Base Address (e.g., BA)} \\
\text{lw } & \$s1,0(\$t1) \\
\text{lw } & \$s2,4(\$t1) \\
\text{Loop: } & \text{lb } \$t2,0(\$s1) \\
& \text{sb } \$t2,0(\$s2) \\
& \text{addi } \$s1, \$s1,1 \\
& \text{addi } \$s2, \$s2,1 \\
& \text{beq } \$t2, \$zero, \text{Exit} \\
\text{Exit: } & \# \text{ N characters } => \# \text{N}+3 \text{ instructions}
\end{align*}
\]

**Administrivia**

- This week in lab and homework:
  - Lab #3 EC2
    - 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
- Difficulty of Project 1 part 1, time waiting for Cloud Computing in Project 1 Part 2 decided to delay HW#3 deadline a week to Sunday Feb 11

\( \text{MH} \)
Apple passes HP as top PC maker when counting iPad sales

By Mikey Campbell

Monday, January 30, 2012

New data released on Monday by market analyst Canalys shows that Apple’s iPad was instrumental to an overall growth in the client PC market for Q4 2011, as excluding the tablet’s sales numbers would see the sector posting a net loss.

Canalys notes that desktop, netbook, notebook and tablet sales grew 16 percent from the year ago quarter, however if iPads are not included as PCs, that number would drop to negative 0.4 percent.

Apple’s top performance was followed by HP, Lenovo, Dell and Acer, respectively.

Last year was said to be “The Year of the Tablet,” and Apple CEO Tim Cook quipped during the company’s “most people would agree, it was the year of the iPad for the second year in a row.”

Know Your Prof: On Wrestling Team in High School & College

Why Would Wrestling Help a CS Prof?
(from toast at 50 year reunion)

• Increased self-confidence
  -- Physical courage => later intellectual courage

• A lesson in ethics
  -- Coaching honor system that their wrestlers “made weight” (e.g., weighed ≤ 136 pounds)
  -- Wrestlers imprint coach’s ethics

• Learned how to form and lead teams
  -- Despite largely individual sport

Assembler Pseudo-instructions

• Register $zero always contains 0

• Can use “pseudo-instruction” in assembly language to make it programming easier

• Example
  
  clear $rt

• Implemented as:

  add $rt, $zero, $zero

More Pseudo-Instructions

<table>
<thead>
<tr>
<th>Name</th>
<th>Instruction syntax</th>
<th>Real instruction translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Move</td>
<td>move $r1, $r2</td>
<td>add $r1, $r2, 0</td>
</tr>
<tr>
<td>Clear</td>
<td>clear $r1</td>
<td>add $r1, $zero, $zero</td>
</tr>
<tr>
<td>Load Address</td>
<td>la $r1, LabelAddr</td>
<td>li $r1, LabelAddr[31:16]</td>
</tr>
<tr>
<td>Branch unconditionally</td>
<td>beq $r1, $r2, Label</td>
<td>beq $r1, $r2, Label</td>
</tr>
<tr>
<td>Branch and link</td>
<td>bgt $r1, $r2, Label</td>
<td>bgt $r1, $r2, Label</td>
</tr>
<tr>
<td>Branch if greater than or equal</td>
<td>bgt $r1, $r2, Label</td>
<td>bgt $r1, $r2, Label</td>
</tr>
<tr>
<td>Branch if less than</td>
<td>bgt $r1, $r2, Label</td>
<td>bgt $r1, $r2, Label</td>
</tr>
<tr>
<td>Branch if greater than or equal</td>
<td>bgt $r1, $r2, Label</td>
<td>bgt $r1, $r2, Label</td>
</tr>
<tr>
<td>Branch if less than or equal</td>
<td>bgt $r1, $r2, Label</td>
<td>bgt $r1, $r2, Label</td>
</tr>
<tr>
<td>Branch if greater than</td>
<td>bgt $r1, $r2, Label</td>
<td>bgt $r1, $r2, Label</td>
</tr>
<tr>
<td>Multiply and return only first 32 bits</td>
<td>mul $r1, $r2, $r3</td>
<td>mul $r1, $r2, $r3</td>
</tr>
</tbody>
</table>

Assembler Pseudo-instructions

• See http://en.wikipedia.org/wiki/MIPS_architecture

• 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
- MIPS register allocation convention
- Technology Break
- Memory Heap
- Summary

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 + $zero + $at of 32, 23 left!)

MIPS Registers
- Assembly Language Conventions
- $t0-$t9: 10 x temporaries (intermediates)
- $s0-$s7: 8 x “saved” temporaries (program variables)
- 18 registers
- 32 – (18 + 9) = 5 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
    jal ProcedureAddress
- 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? ????

Student Rule?
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 $sp$
- (2B out of 32, 4 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 $t0$

MIPS Code for leaf_example

```mips
leaf_example:
    # adjust stack for 2 int items
    # save $t0$ for use afterwards
    # 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
```

Stack Before, During, After Function

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

MIPS Code for leaf_example

```mips
leaf_example:
    addi $sp,$sp,-8 # adjust stack for 2 int items
    # save $t0$ for use afterwards
    # 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

```mips
leaf_example:
    # adjust stack for 2 int items
    addi $sp,$sp,-8
    sw $t0, 4($sp)
    # save $t0$ for use afterwards
    # 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

add $s0,$a0,$a1  # adjust stack for 2 int items
sw $s0, 0($sp)  # save $s0 for use afterwards
sw $t0, 4($sp)  # save $t0 for use afterwards
add $s0,$a0,$a1  # f = g + h
$t0 = i + j  # return value (g + h) – (i + j)
add $t0,$a2,$a3  # restore register $t0 for caller
lw $s0, 0($sp)  # adjust stack to delete 2 items
lw $t0, 4($sp)  # jump back to calling routine

MIPS Code for leaf_example

add $s0,$a0,$a1  # adjust stack for 2 int items
sw $s0, 0($sp)  # save $s0 for use afterwards
sw $t0, 4($sp)  # save $t0 for use afterwards
add $s0,$a0,$a1  # f = g + h
$t0 = i + j  # return value (g + h) – (i + j)
add $t0,$a2,$a3  # restore register $t0 for caller
lw $s0, 0($sp)  # adjust stack to delete 2 items
lw $t0, 4($sp)  # jump back to calling routine

MIPS Code for leaf_example

add $s0,$a0,$a1  # adjust stack for 2 int items
sw $s0, 0($sp)  # save $s0 for use afterwards
sw $t0, 4($sp)  # save $t0 for use afterwards
add $s0,$a0,$a1  # f = g + h
$t0 = i + j  # return value (g + h) – (i + j)
add $t0,$a2,$a3  # restore register $t0 for caller
lw $s0, 0($sp)  # adjust stack to delete 2 items
lw $t0, 4($sp)  # jump back to calling routine

MIPS Code for leaf_example

add $s0,$a0,$a1  # adjust stack for 2 int items
sw $s0, 0($sp)  # save $s0 for use afterwards
sw $t0, 4($sp)  # save $t0 for use afterwards
add $s0,$a0,$a1  # f = g + h
$t0 = i + j  # return value (g + h) – (i + j)
add $t0,$a2,$a3  # restore register $t0 for caller
lw $s0, 0($sp)  # adjust stack to delete 2 items
lw $t0, 4($sp)  # jump back to calling routine
MIPS Code for leaf_example

```mips
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
    add $t0, $a2, $a3 # $t0 = $i + $j
    sub $v0, $s0, $t0 # return value (g + h) – ($i + $j)
    lw $s0, 0($sp) # restore register $s0 for caller
    lw $t0, 4($sp) # restore register $t0 for caller
    addi $sp, $sp, 8 # adjust stack to delete 2 items
    jr $ra # jump back to calling routine
```

What will the printf output?

```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;
}
```

```c
int main(void) {
    int x;
    x = leaf(1, 2, 3, 4);
    x = leaf(3, 4, 1, 2);
    ...
    printf("%d\n", *p);
    }
```

What If a Function Calls a Function? Recursive Function Calls?
- Would clobber values in $a0 to $a1 and $ra
- What is the solution?

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
- (29 of 32, 3 left!)
Recursive Function Factorial

```c
int fact(int n)
{
    if (n < 1) return (1);
    else return (n * fact(n-1));
}
```

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" $s0-$s7
2. Not preserved across function call
   - Caller cannot rely on values being unchanged
   - Return value registers $v0,$v1, Argument registers $a0-$a3, "temporary registers" $t0-$t9

Where is the Stack in Memory?

- MIPS convention
- Stack starts in high memory and grows down
  - Hexadecimal (base 16): 7fff ffec
- 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
    - (30 of 32, left! – will see when talk about OS)
- Heap above static for data structures that grow and shrink; grows up to high addresses

MIPS Memory Allocation

Register Allocation and Numbering
Which statement is FALSE?

- MIPS uses `jal` to invoke a function and `jr` to return from a function
- `jal` saves `PC+1` in `%ra` without saving and restoring them
- The callee can use temporary registers (`%t`) without saving and restoring them
- The caller can rely on save registers (`%a`) without fear of callee changing them

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