CS 61c: Great Ideas in Computer Architecture

MIPS Functions

**Instructor:** Alan Christopher

July 1, 2014
Review I

- **RISC Design Principles**
  - Smaller is faster: 32 registers, fewer instructions
  - Keep it simple: rigid syntax, fixed instruction length
- **MIPS Registers:** $s0$–$s7$, $t0$–$t9$, $0$
  - Only operands used by instructions
  - No variable types, just bits
- **Memory is byte-addressed**
  - Need to watch endianness when mixing words and bytes
Review II

- **MIPS Instructions**
  - Arithmetic: `add`, `sub`, `addi`, `mult`, `div`, `addu`, `subu`, `addiu`
  - Data Transfer: `lw`, `sw`, `lb`, `sb`, `lbu`
  - Branching: `beq`, `bne`, `j`
  - Bitwise: `and`, `andi`, `or`, `ori`, `nor`, `xor`, `xori`
  - Shifting: `sll`, `sllv`, `srl`, `srlv`, `sra`, `srav`
Great Idea #1: Levels of Representation/Interpretation

```
High Level Language Program (e.g. C)
```

```
Compiler
```

```
Assembly Language Program (e.g. MIPS)
```

```
Assembler
```

```
Machine Language Program
```

```
Machine Interpretation
```

```
Hardware Architecture Description (e.g. block diagrams)
```

```
Architecture Implementation
```

```
Logic Circuit Description (Circuit Schematic Diagrams)
```

```
temp = v[k];
v[k] = v[k+1]
v[k+1] = temp;
```

```
lw $t0, 0($2)
lw $t1, 4($2)
sw $t1, 0($2)
sw $t0, 4($2)
```

```
1000 1100 0100 1000 0000 0000 0000 0000
1000 1100 0100 1001 0000 0000 0000 0100
1010 1100 0100 1001 0000 0000 0000 0000
1010 1100 0100 1000 0000 0000 0000 0100
```
Outline

Inequalities
   ISA Support

Pseudo-instructions
   Why and What

Administrivia

Functions in MIPS
   Implementation
   Calling Conventions

Summary

Instructor: Alan Christopher
CS 61c: Great Ideas in Computer Architecture
Inequalities in MIPS

- Inequality tests: $<$, $\leq$, $>$, $\geq$
  - RISC-y idea: Use one instruction for all of them
- **Set on Less Than** ($\text{slt}$)
  - $\text{slt} \; \text{dst}, \; \text{src1}, \; \text{src2}$
  - Stores 1 in $\text{dst}$ if $\text{src1} < \text{src2}$, else 0
- Combine with $\text{bne}$, $\text{beq}$, and $\text{0}$, to implement comparisons
Inequalities in MIPS

- C Code:
  ```c
  if (a < b) {
      ...
      /* then */
  }
  ```

- MIPS Code:
  ```mips
  #a->$s0, b->$s1
  # $t0 = (a < b)
  slt $t0, $s0, $s1
  # if (a < b) goto then
  bne $t0, $0, then
  ```
Inequalities in MIPS

- C Code:
  ```c
  if (a < b) {
    ... /* then */
  }
  ```

- MIPS Code:
  ```mips
  #a->$s0, b->$s1
  # $t0 = (a < b)
  slt $t0, $s0, $s1
  # if (a < b) goto then
  bne $t0, $0, then
  ```

- Try to work out the other two on your own:
  - try swapping src1 and src2
  - try switching beq and bne
Immediates in Inequalities

- Three other variants of `slt`
  - `sltu` dst,src1,src2: unsigned comparison
  - `slti` dst,src,imm: compare against constant
  - `sltiu` dst,src,imm: unsigned comparison against constant

- Example:
  
  ```
  addi $s0,$0,-1  # $s0=0xFFFFFFFF
  slti $t0,$s0,1  # $t0=1
  sltiu $t1,$s0,1 # $t1=0
  ```
MIPS Signed vs. Unsigned

- MIPS terms “signed” and “unsigned” appear in 3 different contexts:
  - Signed vs. unsigned bit extension
    - lb
    - lbu
  - Detect vs. don’t detect overflow
    - add, addi, sub, mult, div
    - addu, addiu, subu, multu, divu
  - Signed vs. unsigned comparison
    - slt, slti
    - sltu, sltiu
Question: What C code properly fills in the following blank?

do {i--;} while (_______________);

Loop: # i-0>$s0, j->$s1
addi $s0, $s0, -1
slti $t0, $s1, 2
beq $t0, $0, Loop
slt $t0, $s1, $s0
bne $t0, $0, Loop

(white) j >= 2 || j < i
(white) j >= 2 && j < i
(white) j < 2 || j >= i
(white) j < 2 && j >= i
Question: What C code properly fills in the following blank?

do {i--;}

while (______________);

---

Loop:

# i->$s0, j->$s1
addi $s0, $s0, -1  # i = i - 1
slti $t0, $s1, 2    # $t0 = (j < 2)
beq $t0, $0, Loop  # goto Loop if $t0 == 0
slt $t0, $s1, $s0   # $t0 = (j < i)
bne $t0, $0, Loop  # goto Loop if $t0 != 0

---

(blue) j >= 2 || j < i
(green) j >= 2 && j < i
(purple) j < 2 || j >= i
(yellow) j < 2 && j >= i
Outline

Inequalities

ISA Support

Pseudo-instructions

Why and What

Administrivia

Functions in MIPS

Implementation

Calling Conventions

Summary

Instructor: Alan Christopher
CS 61c: Great Ideas in Computer Architecture
Assembler Pseudo-Instructions

- Certain C statements are implemented unintuitively in MIPS
  - e.g. assignment \((a=b)\) via addition with 0
- MIPS has a set of “pseudo-instructions” to make programming easier
  - More intuitive to read, but get translated into actual instructions later

- Example:
  
  ```
  move dst,src is translated to
  addi dst,src,0
  ```
Assembler Pseudo-Instructions

  - List also includes the translations for each instruction
- **Load Address** (*la*)
  - *la* dst, label
  - Loads address of specified label into dst
- **Load Immediate** (*li*)
  - *li* dst, imm
  - Loads a 32-bit immediate into dst
- MARS supports more pseudo-instructions (see help)
  - Don’t go overboard, it’s easy to confuse yourself with esoteric syntax.

**Instructor:** Alan Christopher
CS 61c: Great Ideas in Computer Architecture
Assembler Register

Problem:
- When breaking up a pseudo-instruction, the assembler may need to use an extra register
- If it uses a regular register it might overwrite data that the program was using
Problem:
- When breaking up a pseudo-instruction, the assembler may need to use an extra register.
- If it uses a regular register it might overwrite data that the program was using.

Solution:
- Reserve a register (\$1 or \$at for “assembler temporary”) that assembler will use to break up pseudo-instructions.
- Since the assembler may use this at any time, it’s not safe to code with it.
MAL vs. TAL

- True Assembly Language (TAL)
  - The instructions a computer understands and executes

- MIPS Assembly Language (MAL)
  - Instructions the assembly programmer can use (including pseudo-instructions)
  - Each MAL instruction maps directly to 1 or more TAL instructions

- $\text{TAL} \subseteq \text{MAL}$
Outline

Inequalities
- ISA Support

Pseudo-instructions
- Why and What

Administrivia

Functions in MIPS
- Implementation
- Calling Conventions

Summary

Instructor: Alan Christopher
CS 61c: Great Ideas in Computer Architecture
Adminstrivia

- HW2 due Friday
- HW3 due Sunday
- No class (lab or lecture) on Thursday
Outline

Inequalities
  ISA Support

Pseudo-instructions
  Why and What

Administrivia

Functions in MIPS
  Implementation
  Calling Conventions

Summary

Instructor: Alan Christopher
CS 61c: Great Ideas in Computer Architecture
Six Steps of Calling a Function

1. Put *arguments* in place where the function can access them
2. Transfer control to the function
3. The function will acquire any (local) storage resources it needs
4. The function performs its desired task
5. The function puts *return value* in an accessible place and “cleans up”
6. Control is returned to the caller
MIPS Registers for Function Calls

- Registers are *much* faster than memory, so use them whenever possible
- \( \$a0-\$a3 \): four *argument* registers to pass parameters
- \( \$v0-\$v1 \): two *value* registers for return values
- \( \$ra \): *return address* register that saves where a function is called from
MIPS Instructions for Function Calls

- **Jump and Link** (`jal`)
  - Saves the location of the *following* instruction in register `$ra` and then jumps to label (function address)
  - Used to invoke a function

- **Jump Register** (`jr`)
  - `jr src`
  - Unconditionally jump to the address specified in `src` (almost always used with `$ra`)
  - Most commonly used to return from a function
Instruction Addresses

- `jal` puts the `address` of an instruction in $ra
- *Instructions are stored as data in memory!*
  - **Recall**: Code Section
- In MIPS, all instructions are 4 bytes long, so each instruction address differs by 4
  - **Remember**: Memory is byte-addressed
- Labels get converted to instruction address eventually
Program Counter

- The *program counter* (PC) is a special register that holds the address of the current instruction being executed
  - This register is not (directly) accessible to the programmer, (is accessible to `jal`)
- `jal` stores `PC + 4` into `$ra`
  - Why not `PC + 1`?
  - What would happen if we stored `PC` instead?
- All branches and jumps (`beq`, `bne`, `j`, `jal`, `jr`) work by storing an address into `PC`
Function Call Example

```c
... sum(a,b); ... /* a->$s0, b->$s1 */
int sum(int x, int y) {
    return x + y;
}
```

<table>
<thead>
<tr>
<th>Address</th>
<th>MIPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>addi $a0, $s0, 0 # x = a</td>
</tr>
<tr>
<td>1004</td>
<td>addi $a1, $s1, 0 # y = b</td>
</tr>
<tr>
<td>1008</td>
<td>jal sum</td>
</tr>
<tr>
<td>1012</td>
<td># $ra = 1012, goto sum</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>2000</td>
<td>sum: add $v0,$a0,$a1</td>
</tr>
<tr>
<td>2004</td>
<td>jr $ra</td>
</tr>
<tr>
<td></td>
<td># return</td>
</tr>
</tbody>
</table>
Six Steps of Calling a Function

1. Put *arguments* in place where the function can access them ($a0$–$a3$)
2. Transfer control to the function (*jal*)
3. The function will acquire any (local) storage resources it needs
4. The function performs its desired task
5. The function puts *return value* in an accessible place ($v0$–$v1$) and “cleans up”
6. Control is returned to the caller (*jr*)
Saving and Restoring Registers

- Why might we need to save registers?

Instructor: Alan Christopher
CS 61c: Great Ideas in Computer Architecture
Saving and Restoring Registers

- Why might we need to save registers?
  - Limited number of registers to use
  - What happens if a function calls another function? ($ra$ would get overwritten!)

- Where should we save registers?
Saving and Restoring Registers

- Why might we need to save registers?
  - Limited number of registers to use
  - What happens if a function calls another function? ($ra$ would get overwritten!)

- Where should we save registers? **The stack**
- $sp$ (stack pointer) register contains pointer to the current bottom (last used space) of the stack
Review: Memory Layout

- **Stack**: Space for saved procedure information
- **Heap**: Dynamically allocated space
- **Static Data**: Global variables, string literals
- **Code**: Program instructions

Instructor: Alan Christopher

CS 61c: Great Ideas in Computer Architecture
Example: sum_square

```c
int sum_square(int x, int y) {
    return mult(x, x) + y;
}
```

- What do we need to save?
  - Call to `mult` will overwrite $ra$, so save it
  - Reusing $a1$ to pass 2nd argument to `mult`, but need current value ($y$) later, so save $a1$
- To save something on the stack, `move $sp down` the required amount and fill the “created” space.
Example: sum_square

```c
int sum_square(int x, int y) {
    return mult(x, x) + y;
}
```

```
sum_square:
    ### Push the stack ###
    addi $sp, $sp, -8    # make space on stack
    sw $ra, 4($sp)       # save ret addr
    sw $a1, 0($sp)       # save y
    add $a1, $a0, $zero  # set 2nd mult arg
    jal mult            # call mult
    lw $a1, 0($sp)       # restore y
    add $v0, $v0, $a1    # retval = mult(x, x) + y
    ### Pop the stack ###
    lw $ra, 4($sp)       # get ret addr
    addi $sp, $sp, 8     # restore stack
    jr $ra
mult: ...
```
Canonical Function Structure

- **Prologue:**
  
  ```
  func_label:
  addiu $sp, $sp, -$framesize
  sw $ra, <framesize - 4>($sp)
  ...
  # save other registers as needed
  ```

- **Body**
  
  ```
  ...
  # whatever the function actually does
  ```

- **Epilogue**
  
  ```
  ...
  # restore other registers as needed
  lw $ra, <framesize - 4>($sp)
  addiu $sp, $sp, $framesize
  ```
Local Variables and Arrays

- Any local variables the compiler cannot assign to registers will be allocated as part of the stack frame (Recall: spilling to memory)
- Locally declared arrays and structs are also allocated on the stack frame
- Stack manipulation is the same as before
  - Move $sp down an extra amount and use the space created as storage
Stack Before, During, After Call

Instructor: Alan Christopher
CS 61c: Great Ideas in Computer Architecture
Technology Break
Register Conventions

- **CalleR**: The calling function
- **CalleE**: The function being called
- **Register Conventions**: A set of generally accepted rules governing which registers will be unchanged after a procedure call (jal) and which may have changed (“been clobbered”)

Instructor: Alan Christopher
CS 61c: Great Ideas in Computer Architecture
Saved Registers

- These registers are expected to be the same before and after a function
  - If the callee uses them, it must restore the values before returning
  - Usually means saving the old values, using the register, and then reloading the old values back into the registers
Saved Registers

- These registers are expected to be the same before and after a function
  - If the callee uses them, it must restore the values before returning
  - Usually means saving the old values, using the register, and then reloading the old values back into the registers
- $s0$–$s7$ (saved registers)
Saved Registers

- These registers are expected to be the same before and after a function
  - If the callee uses them, it must restore the values before returning
  - Usually means saving the old values, using the register, and then reloading the old values back into the registers
- $s0$–$s7$ (saved registers)
- $sp$ (stack pointer)
Saved Registers

- These registers are expected to be the same before and after a function
  - If the callee uses them, it must restore the values before returning
  - Usually means saving the old values, using the register, and then reloading the old values back into the registers

- $s0$–$s7$ (saved registers)
- $sp$ (stack pointer)
- $ra$ (return address)
Volatile Registers

- These registers can be freely changed by the callee
  - If callee needs them, it must save those values before making a procedure call
Volatile Registers

- These registers can be freely changed by the callee
  - If callee needs them, it must save those values before making a procedure call
- $t0$–$t9$ (temporary registers)
Volatile Registers

- These registers can be freely changed by the callee
  - If callee needs them, it must save those values before making a procedure call
- $t0$–$t9$ (temporary registers)
- $v0$–$v1$ (return values)
  - These will contain the functions return values
Volatile Registers

- These registers *can be freely changed by the callee*
  - If callee needs them, it must save those values before making a procedure call

- $t0$–$t9$ (temporary registers)
- $v0$–$v1$ (return values)
  - These will contain the functions return values
- $a0$–$a3$ (return address and arguments)
  - These will change if the callee invokes another function
  - Nested functions mean that callee is also a caller

**Instructor:** Alan Christopher

CS 61c: Great Ideas in Computer Architecture
Register Conventions Summary

- One more time:
  - CallER must save any *volatile* registers it is using onto the stack before making a procedure call
  - CallEE must save any *saved* registers before clobbering their contents

- Notes:
  - CallER and CallEE only need to save the registers they *actually use* (not all!)
  - Don’t forget to restore values after finished clobbering registers

- Analogy: Throwing a party while your parents are away
Example: Using Saved Registers

```mips
myFunc:
    addiu $sp, $sp, -12
    sw $ra, 8($sp)
    sw $s0, 4($sp)
    sw $s1, 0($sp)

    jal func1
    jal func2

lw $s1, 0($sp)
lw $s0, 4($sp)
lw $ra, 8($sp)
addiu $sp, $sp, 12
jr $ra
```

- `# Uses $s0 and $s1`
- `# This is the Prologue`
- `# Save saved registers`
- `# Do stuff with $s0, $s1`
- `# $s0, $s1 unchanged by func calls, so can keep using them normally.`
- `# Do stuff with $s0, $s1`
- `# This is the Epilogue`
- `# Restore saved registers`
- `# return`
Example: Using Volatile Registers

```mips
myFunc:
    addiu $sp, $sp, -4   # Uses $s0 and $s1
    sw $ra, 0($sp)       # This is the Prologue
    ...                 # Save saved registers
    addiu $sp, $sp, -4   # Do stuff with $t0
    sw $t0, 0($sp)       # Save volatile registers
    jal func1            # before func call
    lw $t0, 0($sp)       # function may clobber $t0
                            # Restore volatile registers
    addiu $sp, $sp, 4    # Do stuff with $t0
    ...                 # Restore saved registers
lw $ra, 0($sp)         # This is the Epilogue
addiu $sp, $sp, 4     # return
jr $ra
```

Instructor: Alan Christopher
CS 61c: Great Ideas in Computer Architecture
Choosing your Registers

- Minimize register footprint
  - Optimize to reduce number of registers you need to save by choosing which registers to use in a function
  - Only save to memory when absolutely necessary
- Leaf functions
  - Use only $t0$–$t9$ and there is nothing to save
- Functions that call other functions
  - Values that you need throughout go in $s0$–$s7$
  - Others go in $t0$–$t9$
Question: Which statement below is FALSE?

(Blue) MIPS uses jal to invoke functions and jr to return from functions

(Green) jal saves PC+1 in $ra

(Purple) The callee can use temporary registers ($t#) without saving and restoring them

(Yellow) The caller can rely on save registers ($s#) without fear of the callee changing them
Question: Which statement below is FALSE?

(blue) MIPS uses jal to invoke functions and jr to return from functions

(green) jal saves PC+1 in $ra

(purple) The callee can use temporary registers ($t#) without saving and restoring them

(yellow) The caller can rely on save registers ($s#) without fear of the callee changing them
Outline

Inequalities
  ISA Support

Pseudo-instructions
  Why and What

Administrivia

Functions in MIPS
  Implementation
  Calling Conventions

Summary
And in Conclusion

- Inequalities done using `slt` and allow us to implement the rest of control flow
- Pseudo-instructions make code more readable
  - Part of MAL, translated into TAL
- MIPS function implementation
  - Jump and link (jal) invokes, jump register (jr $ra) returns
  - Registers $a0-$a3 for arguments, $v0,$v1 for return values

Instructor: Alan Christopher
CS 61c: Great Ideas in Computer Architecture
And in Conclusion II

- Register conventions preserve values of registers between function calls
  - Different responsibilities for the caller and callee
  - Registers split between *saved* and *volatile*
- Use the stack for spilling registers, saving return addresses, and local variables