Levels of Representation/Interpretation

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

Compiler

Assembly Language Program (e.g., MIPS)

Assembler

Machine Language Program (MIPS)

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)
```

Anything can be represented as a number, i.e., data or instructions

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```
Agenda

• Support for Strings
• Administrivia
• Implementing Functions In MIPS
• Break
• Calling Conventions
• Summary
Strings: C vs. Java

- C: each char is 8-bit ASCII. Deal with 1 byte at a time.
- Java: 16-bit UNICODE (a much larger character vocabulary). Deal with 2 bytes at a time.
Support for Characters and Strings

• Load a word, use andi to isolate byte

\[
\text{lw } \$s0,0(\$s1) \\
\text{andi } \$s0,\$s0,255 \ # \text{Zero everything but last 8 bits}
\]

• RISC Design Principle: “Make the Common Case Fast”—Many programs use text: MIPS has load byte instruction (\text{lb})

\[
\text{lb } \$s0,0(\$s1)
\]

• Also store byte instruction (\text{sb})
Loading, Storing bytes

• What do with other 24 bits in the 32 bit register?
  – lb: sign extends to fill upper 24 bits

\[
\begin{array}{cccccccc}
  xxx & xxx & xxx & xxx & xxx & xxx & xxx & xxx \\
\end{array}
\]

...is copied to “sign-extend”

• Normally don’t want to sign extend chars
• MIPS instruction that doesn’t sign extend when loading bytes:
  – load byte unsigned \((lbu)\)
Support for Characters and Strings

• MIPS also provides fast support for Unicode:

• *load halfword* instruction (*lh*)

  \[
  lh \ $s0, 0($s1) \\
  \]
  – There’s also *load halfword unsigned* (*lhu*)

• *store halfword* instruction (*sh*)

  \[
  sh \ $s0, 0($s1) \\
  \]
MIPS Signed vs. Unsigned – Three Different Meanings!

- MIPS terms Signed/Unsigned “overloaded”:
  - Do/Don't sign extend
    - (lb, lbu, lh, lhu)
  - Do/Don't overflow
    - (add, addi, sub, mult, div)
    - (addu, addiu, subu, multu, divu)
  - Do signed/unsigned compare
    - (slt, slti/sltu, sltiu)
What Do We Know Now?

• Arithmetic
  – add, addi, addu, addiu, sub, subu, and, andi, or, ori, sll, slr, slt, slti, sltiu!

• Data Transfer
  – lw, sw, lb, lbu, sb, lh, lhu, sh!

• Branches and Jumps
  – bne, beq, j
Agenda

- Support for Strings
- **Administrivia**
- Implementing Functions In MIPS
- **Break**
- Calling Conventions
- Summary
Administrivia

• HW2 Posted.
  – Remember, it’s big!

• Project 1 posted by tomorrow, due 7/10.
  – No homework that week.

• Lab 4 is up.

• Please vote on times for Midterm and Final, and tell us any conflicts you have.
  – Survey linked on Piazza, emailed out.
Agenda

• Support for Strings
• Administrivia
• Implementing Functions In MIPS
• Break
• Calling Conventions
• Summary
Functions in C

```c
void foo() {
    int i, j;
    i = factorial(10);
    ...
    j = factorial(25);
}

int factorial(int n) {
    if (n<1) return 1;
    return n*factorial(n-1);
}
```

Functions can be called from multiple places.

Functions have their own local storage.

We pass arguments to functions, and functions return values.

Functions can call other functions, even themselves!
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 Registers for Function Calls

- 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 that saves where a function is called from.
- Including $s$ and $t$ registers, we now know 26 of 32 registers!
MIPS Instructions for Function Calls

• **Invoke function:** *jump and link* instruction (**jal**)  
  – “link” means storing the location of the calling site.  
  – Jumps to label and simultaneously saves the location of following instruction in register $ra
    
    \[
    \text{jal ProcedureAddress}
    \]

• **Return from function:** *jump register* instruction (**jr**)  
  – Unconditional jump to address specified in register $ra
    
    \[
    \text{jr } $ra
    \]
In MIPS, all instructions are 4 bytes, and stored in memory just like data. So here we show the addresses of where the programs are stored.
Function Call Example (2/4)

... sum(a,b);... /* a,b:$s0,$s1 */

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

address (shown in decimal)

```
1000  add  $a0,$s0,$zero    # x = a
1004  add  $a1,$s1,$zero    # y = b
1008  addi $ra,$zero,1016    # $ra=1016
1012  j sum                   # jump to sum
1016
...
2000  sum: add $v0,$a0,$a1
2004  jr  $ra                  # new instruction
```
Function Call Example (3/4)

... sum(a,b);... /* a,b:$s0,$s1 */

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

<table>
<thead>
<tr>
<th>address (shown in decimal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000 add $a0,$s0,$zero   # x = a</td>
</tr>
<tr>
<td>1004 add $a1,$s1,$zero   # y = b</td>
</tr>
<tr>
<td>1008 jal sum             # $ra = 1012 , goto sum</td>
</tr>
<tr>
<td>1012</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>2000 sum: add $v0,$a0,$a1</td>
</tr>
<tr>
<td>2004 jr $ra              # new instruction</td>
</tr>
</tbody>
</table>
int sum(int x, int y) {
    return x + y;
}

• Question: Why use \texttt{jr} here? Why not use \texttt{j}?

• Answer: \texttt{sum} might be called by many places, so we can’t return to a fixed place. The calling proc to \texttt{sum} must be able to say “return here” somehow.
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!
  – Doesn’t count as one of the 32 we’ve mentioned.

• jal puts (PC+4) into $ra$ (then jumps to label)
• jr $ra$ puts address inside $ra$ into PC
What If a Function Calls another Function?

• Would overwrite $ra$
  – Might also need to reuse other registers
• What is the solution?
• Save $ra$ (and anything else that might be needed later) on the stack!
• $sp$ register contains pointer to current bottom of stack.
Recall: Memory Layout

Address -> ∞

Stack

$sp$ stack pointer

Heap

<table>
<thead>
<tr>
<th>Static</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code</td>
</tr>
</tbody>
</table>

Space for saved procedure information

Explicitly created space, i.e., `malloc()`

Variables declared once per program; e.g., globals

Program
Example: sumSquare

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

- Call to mult will overwrite $ra$, must save it.
- Need to reuse $a1$ to pass second argument to mult, but need current value ($y$) later. Must save $a1$.
- To save something to stack, move $sp$ down required amount and fill the space it’s created.
Example: sumSquare

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

**sumSquare:**
```
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    # ret val = mult(x,x)+y
lw $ra, 4($sp)     # get ret addr
addi $sp,$sp,8     # restore stack
jr $ra
```

mult: ...
Basic Structure of a Function

**Prologue**

```
func_label:
addi $sp, $sp, -framesize
sw $ra, framesize-4($sp)
save other regs if need be
```

**Body**

(call other functions...)

... 

**Epilogue**

```
restore other regs if need be
lw $ra, framesize-4($sp)
addi $sp, $sp, framesize
jr $ra
```
Local Variables and Arrays

• Any local variables the compiler cannot assign to registers will be allocated as part of the stack frame.
• Locally declared arrays and structs are also allocated as part of the stack frame.
• Stack manipulation is same as before: move stack pointer down an extra amount, use the space it created as storage.
Stack Before, During, After Call

Diagram:

- **High address**
  - `$fp`:
  - `$sp`:

- **Low address**
  - a.
  - b. Saved argument registers (if any)
    - Saved return address
    - Saved saved registers (if any)
    - Local arrays and structures (if any)
  - c.

6/29/2011
Agenda

• Support for Strings
• Administrivia
• Implementing Functions In MIPS
• Break
• Calling Conventions
• Summary
Agenda

• Support for Strings
• Administrivia
• Implementing Functions In MIPS
• Break
• Calling Conventions
• Summary
# MIPS Registers

<table>
<thead>
<tr>
<th>Category</th>
<th>Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>The constant 0</td>
<td>$0</td>
</tr>
<tr>
<td>Reserved for Assembler</td>
<td>$1</td>
</tr>
<tr>
<td>Return Values</td>
<td>$2-$3</td>
</tr>
<tr>
<td>Arguments</td>
<td>$4-$7</td>
</tr>
<tr>
<td>Temporary</td>
<td>$8-$15</td>
</tr>
<tr>
<td>Saved</td>
<td>$16-$23</td>
</tr>
<tr>
<td>More Temporary</td>
<td>$24-$25</td>
</tr>
<tr>
<td>Used by Kernel</td>
<td>$26-27</td>
</tr>
<tr>
<td>Global Pointer</td>
<td>$28</td>
</tr>
<tr>
<td>Stack Pointer</td>
<td>$29</td>
</tr>
<tr>
<td>Frame Pointer</td>
<td>$30</td>
</tr>
<tr>
<td>Return Address</td>
<td>$31</td>
</tr>
</tbody>
</table>

- *Already discussed*
- *Not yet*

(From MIPS green sheet)
Use names for registers -- code is clearer!
Register Conventions (1/4)

- **Calle**$: the calling function
- **Calle**$: the function being called
- When callee returns from executing, the caller needs to know which registers may have changed and which are guaranteed to be unchanged.
- **Register Conventions**: A set of generally accepted rules as to which registers will be unchanged after a procedure call (**jal** and which may be changed.
Register Conventions (2/4) – saved

• $0: No Change. Always 0.

• $s0-$s7: Restore if you change. Very important, that’s why they’re called saved registers. If the callee changes these in any way, it must restore the original values before returning.

• $sp: Restore if you change. The stack pointer must point to the same place before and after the jal call, or else the caller won’t be able to restore values from the stack.

• HINT -- All saved registers start with $!
Register Conventions (3/4) – volatile

• $ra: Can Change. The jal call itself will change this register. Caller needs to save on stack if nested call.

• $v0-$v1: Can Change. These will contain the new returned values.

• $a0-$a3: Can change. These are volatile argument registers. Caller needs to save if they are needed after the call.

• $t0-$t9: Can change. That’s why they’re called temporary: any procedure may change them at any time. Caller needs to save if they’ll need them afterwards.
Register Conventions (4/4)

• What do these conventions mean?
  – If function $R$ calls function $E$, then function $R$ must save any temporary registers that it may be using onto the stack before making a `jal` call.
  – Function $E$ must save any $S$ (saved) registers it intends to use before garbling up their values.

• Remember: caller/callee need to save only temporary/saved registers they are using, not all registers.
Where is the Stack in Memory?

• MIPS convention
• Stack starts in high memory and grows down
  – Hexadecimal: $7ffe \text{ ff}f c_{\text{hex}}$
• MIPS programs (*text segment*) in low end
  – $0040 \ 0000_{\text{hex}}$
• *static data segment* (constants and other static variables) above text for static variables
  – MIPS convention *global pointer* ($g_{\text{p}}$) points to static
• Heap above static for data structures that grow and shrink; grows up to high addresses
MIPS Memory Allocation

Diagram showing memory allocation with:
- \( sp \rightarrow 7fff \ fff \text{c}_{hex} \)
- \( gp \rightarrow 1000 \ 8000 \text{hex} \)
- \( 1000 \ 0000 \text{hex} \)
- \( pc \rightarrow 0040 \ 0000 \text{hex} \)

Memory sections:
- Stack
- Dynamic data
- Static data
- Text
- Reserved
And in Conclusion, ...

- MIPS instructions and conventions for implementing functions.
  - Jump and link (jal) invokes, jump register (jr $ra) returns
  - Registers $a0–$a3 for arguments, $v0–$v1 for return values
  - Register conventions determine responsibilities of Caller and Callee for preserving values of registers.

- Stack for spilling registers, saving return address, local variables