CS 61C: 
Great Ideas in Computer Architecture
Strings and Functions
Instructor: 
Krste Asanovic, Randy H. Katz
http://inst.eecs.Berkeley.edu/~cs61c/sp12

New-School Machine Structures
(It’s a bit more complicated!)

• Parallel Requests
  Assigned to computer
  e.g., Search “Katz”

• Parallel Threads
  Assigned to core
  e.g., Lookup, Ads

• Parallel Instructions
  >1 instruction @ one time
  e.g., 5 pipelined instructions

• Parallel Data
  >1 data item @ one time
  e.g., Add of 4 pairs of words

• Hardware descriptions
  All gates @ one time

Strings: C vs. Java

• Recall: a string is just a long sequence of characters (i.e., array of chars)
• C: 8-bit ASCII, define strings with end of string character NUL (0 in ASCII)
• Java: 16-bit Unicode, first entry gives length of string
Strings

• “Cal” in Unicode in Java; How many bytes?
• Using 1 integer per byte, what does it look like?
  (For Latin alphabet, 1st byte is 0, 2nd byte is ASCII)

<table>
<thead>
<tr>
<th>Character</th>
<th>Unicode</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>65</td>
<td>65</td>
</tr>
<tr>
<td>B</td>
<td>66</td>
<td>66</td>
</tr>
<tr>
<td>C</td>
<td>67</td>
<td>67</td>
</tr>
<tr>
<td>D</td>
<td>68</td>
<td>68</td>
</tr>
<tr>
<td>E</td>
<td>69</td>
<td>69</td>
</tr>
<tr>
<td>F</td>
<td>70</td>
<td>70</td>
</tr>
<tr>
<td>G</td>
<td>71</td>
<td>71</td>
</tr>
<tr>
<td>H</td>
<td>72</td>
<td>72</td>
</tr>
<tr>
<td>I</td>
<td>73</td>
<td>73</td>
</tr>
<tr>
<td>J</td>
<td>74</td>
<td>74</td>
</tr>
<tr>
<td>K</td>
<td>75</td>
<td>75</td>
</tr>
<tr>
<td>L</td>
<td>76</td>
<td>76</td>
</tr>
<tr>
<td>M</td>
<td>77</td>
<td>77</td>
</tr>
<tr>
<td>N</td>
<td>78</td>
<td>78</td>
</tr>
<tr>
<td>O</td>
<td>79</td>
<td>79</td>
</tr>
<tr>
<td>P</td>
<td>80</td>
<td>80</td>
</tr>
<tr>
<td>Q</td>
<td>81</td>
<td>81</td>
</tr>
<tr>
<td>R</td>
<td>82</td>
<td>82</td>
</tr>
<tr>
<td>S</td>
<td>83</td>
<td>83</td>
</tr>
<tr>
<td>T</td>
<td>84</td>
<td>84</td>
</tr>
<tr>
<td>U</td>
<td>85</td>
<td>85</td>
</tr>
<tr>
<td>V</td>
<td>86</td>
<td>86</td>
</tr>
<tr>
<td>W</td>
<td>87</td>
<td>87</td>
</tr>
<tr>
<td>X</td>
<td>88</td>
<td>88</td>
</tr>
<tr>
<td>Y</td>
<td>89</td>
<td>89</td>
</tr>
<tr>
<td>Z</td>
<td>90</td>
<td>90</td>
</tr>
</tbody>
</table>

Support for Characters and Strings

• Load a word, use andi to isolate byte
  `lw $s0,0($s1)`
  `andi $s0,$s0,255` # Zero everything but last 8 bits
• RISC Design Principle: “Make the Common Case Fast”—Many programs use text: MIPS has
  load byte instruction (`lb`)  
  `lb $s0,0($s1)`
• Also store byte instruction (`sb`)  

Fast String Copy Code in C

• Copy x[] to y[]
  ```
  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') {
  }
  ```

Support for Characters and Strings

• Load a word, use andi to isolate half of word
  `lw $s0,0($s1)`
  `andi $s0,$s0,65535` # Zero everything but last 16 bits
• RISC Design Principle #3: “Make the Common Case Fast”—Many programs use text, MIPS has
  load halfword instruction (`lh`)  
  `lh $s0,0($s1)`
• Also store halfword instruction (`sh`)  

Fast String Copy in MIPS Assembly

Get addresses of x and y into $s1, $s2
p and q are assigned to these registers
```mips
# $s1 = &p (BA), q @ &p + 4
# $s1 = p
# $s2 = q
# $s2 = "p
# $q = $s2
# $p = $p + 1
# if *p == 0, go to Exit
# go to Loop
```

Exit: # N characters => N*6 + 3 instructions
Fast String Copy in MIPS Assembly

Which statement is TRUE?

```c
char *p, *q;
p = x[0]; q = y[0];
while(*q++ = *p++) != '\0';
```

- □ $t1$ corresponds to $p$
- □ $s1$ corresponds to $p$
- □ $s1$ corresponds to $q$
- □ $s1$ corresponds to $*p$

Agenda

- Review
- Strings in C and MIPS
- Administrivia
- Functions
- And in Conclusion, ...

Administrivia

- Map-Reduce Project #1
  - Two Parts, first part due Sunday
  - Write-up posted last night (thanks Alan!)
- Lab #3
  - Hands on EC2, needed for Project #1, Part 2
- HW #3
  - C practice/numbers and strings
- ... Midterm is coming in < one month!
  - Let us know special accommodation now ...
  - CS 188 students, ask your instructors to return my emails!
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 faster than memory, so use them
- $a0-$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: \texttt{jump and link} instruction (\texttt{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
  \hspace{1cm} \texttt{jal ProcedureAddress}
- Return from function: \texttt{jump register} instruction (\texttt{jr})
  - Unconditional jump to address specified in register $jr$

Notes on Functions
- Calling program (\textit{caller}) puts parameters into registers $a0$–$a3$ and uses \texttt{jal X} to invoke X (\textit{callee})
- Must have register in computer with address of currently executing instruction
  - Instead of Instruction Address Register (better name), historically called \textit{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 \texttt{jal X} place into $ra$? ????

Where Are Old Register Values Saved to Restore Them After Function Call
- Need a place to save old values before call function, restore them when return, and delete
- Ideal is \textit{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 \textit{stack pointer} in MIPS
- Convention is grow from high to low addresses
  - Push decrements $sp$, Pop increments $sp$
  - (28 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`

Stack Before, During, After Function

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

MIPS Code for `leaf_example`

```
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 $s0 for caller
    # restore $t0 for caller
    # delete 2 items from stack
    # jump back to calling routine
```

What will the `printf` output?

```
static int *p;
int leaf (int g, int h, int i, int j)
{
    int f; p = &f;
    f = (g + h) - (i + j);
    return f;
}
```

- `Print -4`
- `Print 4`
- `a.out will crash`
- `None of the above`

What If a Function Calls a Function? Recursive Function Calls?

- Would clobber values in `$a0` to `$a3` 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
   - $s0, s1, sp, $gp, “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 fffe<sub>hex</sub>
- MIPS programs (text segment) in low end
  - 0040 0000<sub>hex</sub>
- static data segment (constants and other static variables) above text for static variables
  - MIPS convention global pointer ($gp) points to static
  - (30 of 32, 2 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

- Stack is the default place for local variables and arguments
- Dynamic data is for data structures that grow and shrink during program execution
Register Allocation and Numbering

<table>
<thead>
<tr>
<th>Name</th>
<th>Register number</th>
<th>Usage</th>
<th>Preserved on exit?</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0</td>
<td>0</td>
<td>The constant zero</td>
<td>n/a</td>
</tr>
<tr>
<td>$1-4</td>
<td>2-3</td>
<td>Vectors for results and expression evaluation</td>
<td>no</td>
</tr>
<tr>
<td>$5-31</td>
<td>4-7</td>
<td>Arguments</td>
<td>no</td>
</tr>
<tr>
<td>$32-71</td>
<td>8-15</td>
<td>Temporaries</td>
<td>no</td>
</tr>
<tr>
<td>$72-77</td>
<td>16-23</td>
<td>Saved</td>
<td>yes</td>
</tr>
<tr>
<td>$78-819</td>
<td>24-25</td>
<td>More temporaries</td>
<td>no</td>
</tr>
<tr>
<td>$28</td>
<td>28</td>
<td>Caller pointer</td>
<td>yes</td>
</tr>
<tr>
<td>$29</td>
<td>29</td>
<td>Stack pointer</td>
<td>yes</td>
</tr>
<tr>
<td>$30</td>
<td>30</td>
<td>Frame pointer</td>
<td>yes</td>
</tr>
<tr>
<td>$31</td>
<td>31</td>
<td>Return address</td>
<td>yes</td>
</tr>
</tbody>
</table>

Which statement is FALSE?

- MIPS uses jal to invoke a function and jr to return from a function
- jal saves PC+1 in %ra
- The callee can use temporary registers (%t0) without saving and restoring them
- The caller can rely on save registers (srl) without fear of callee changing them

And in Conclusion, ...

- C strings are char arrays, byte per character, null terminated
- Distinguish pointers and the memory they point to
  - * for dereference, & for address
- 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