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

Instructors:
Randy H. Katz
David A. Patterson
http://inst.eecs.Berkeley.edu/~cs61c/fa10

Agenda

- Review
- Pointers in C vs. Arrays indices
- C functions vs. Java methods
- Administrivia
- Technology Break
- Functions in MIPS
- Summary

Review from Last Lecture

- People need text as well as numbers
- C: 8-bit ASCII, Java: 16-bit Unicode
- load byte (lb), store byte (sb), load halfword (lh), store halfword (sh) support characters
- Decisions in C and Java (if, while, repeat, for, …) via conditional branch:
  - Branch equal (beq) register1, register2, address
  - Branch not equal (bne) register1, register2, addr.
- Unconditional Branch (jump) too
  - Jump (j)

Review from Last Lecture: String Copy in C

```
i = 0;
while( (x[i] = y[i]) != '\0') /* copy & test byte */
    i += 1;
i => $s0, source address in $s1, destination in $s2
        add $s0,$zero,$zero   # i = 0+0
Loop:   add $t1,$s0,$a1       # address of y[i] in $t1
        lb $t2, 0($t1)        # $t2 = y[i]
        add $t3,$s0,$a0       # address of x[i] in $t3
        sb $t2, 0($t3)        # x[i] = y[i]
        beq $t2,$zero,Exit    # if y[i] == 0, go to Exit
        addi $s0, $s0,1       # i = i + 1
        j Loop                # go to Loop
Exit: # N characters => N*7 + 6 instructions
```


Faster way to write string copy?

Why not just increment addresses in $s1, $s2?

```
i => $s0, source address $s1, destination in $s2
Loop:  lb $t2, 0($s1)       # $t2 = y[i]
       sb $t2, 0($s2)       # x[i] = y[i]
       beq $t2,$zero,Exit   # if y[i] == 0, go to Exit
       addi $s1, $s1,1      # next address to load
       addi $s2, $s2,1      # next address to store
       j Loop               # go to Loop
Exit: # Now N characters => N*6 + 3 instructions
 (if N=4, 34 instructions before vs 27 now)
```


How Get Fast Code in C?

- C added concept of *pointer*
- C pointer is exactly a MIPS address
- Incrementing/decrementing pointer is simply incrementing/decrementing MIPS address
- **`char *p`** means **`p`** is a pointer to a string
- **`&x[0]`** means address of first element of x
- **`*p++`** means read value pointed to by **`p`** and then increment **`p`** by the size of object in bytes

How Get Fast Code in C?

- New string copy

```
char *p, *q;
  p = &x[0];
     /* set p to address of 1st char of x */
  q = &y[0];
     /* set q to address of 1st char of y */
  while(*q++ = *p++) != '\0') ;
```


Why Pointers in C?

- At time C was invented (1972), compilers often didn't produce efficient code
  - Computers 25,000 times faster today, compilers better
- C designed to let programmer say what want code to do without compiler getting in way
  - Even give compilers hints which registers to use!
- Today's compilers produce much better code, so may not need to use pointers
  - Compilers even ignore hints since they do it better!

C vs. Java

	C	Java
Type of Language	Function Oriented	Object Oriented
Program-ming Unit	Function	Class = Abstract Data Type
Compilation	gcc hello.c creates machine language code	javac Hello.java creates Java virtual machine language bytecode
Execution	a.out loads and executes program	java Hello interprets bytecode
hello, world	`#include<stdio.h> int main(void) { printf("Hello\n"); return 0; }`	`public class HelloWorld { public static void main (String[] args) { System.out.printl("Hello"); } }`
Storage	Manual (**malloc**, **free**)	Automatic (garbage collection)

From http://www.cs.princeton.edu/introcs/faq/c2java.html

Functions in C

- Functions: How to structure C programs for understandability and to get reuse
- Calling function: **`s =max(x, y, 100);`**
- If no value to return, declare it type **`void`**
  - Function with no return value called *procedure* in other programming languages

C Functions

- Give name of function and type of value it returns

```
int max(a, b, c) /* declaration */
int a, b, c; /* type of params */
{
  int m;
  m = (a>b)? a:b;
  return(m>c? m:c);
}
```


6 Steps in Calling a Function

1. Put parameters in a place where function can access them
2. Transfer control to function
3. Acquire storage resources needed for function
4. Perform desired task
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 registers
- **`$a0–$a3`**: 4 *argument* registers to pass parameters
- **`$v0–$v1`**: 2 *value* registers to return values
- **`$ra`**: one *return address* register to return to the point of origin

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?

When is Midterm, Final?

- To reduce time pressure, 3 hours for 1.5 hour midterm
- **Midterm Exam Wednesday October 6, 6 – 9PM, Pimental 1**
- **Final Exam Monday December 13, 8 – 11AM, TBD**

The Rules
(and we really mean it!)

Agenda

- Review
- Pointers in C vs. Arrays indices
- C functions vs. Java methods
- Administrivia
- Technology Break
- Functions in C
- Summary

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

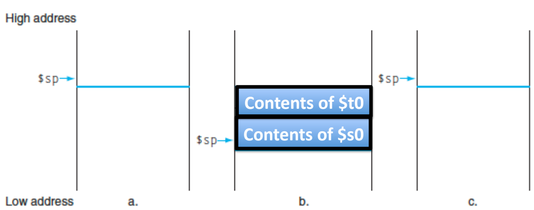
Example

```
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, and j in argument registers $a0, $a1, $a2, and $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:

```
        # adjust stack for 2 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
  jr $ra # 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

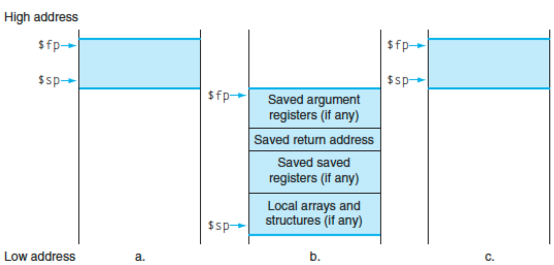
```
fact:addi $sp,$sp,-8
    # adjust stack for 2 items
    sw $ra, 4($sp)
    # save return address
    sw $a0, 0($sp)
    # save argument n
    slti $t0,$a0,1 # test for n < 1
    beq $t0,$zero,L1
    # if n >= 1, go to L1
    addi $v0,$zero,1
    # Then part (n==1) return 1
    addi $sp,$sp,8
    # pop 2 items off stack
    jr $ra # return to caller

L1: addi $a0,$a0,-1
    # Else part (n >= 1)
    # arg. gets (n – 1)
    jal fact
    # call fact with (n – 1)
    lw $a0, 0($sp)
    # return from jal: restore n
    lw $ra, 4($sp)
    # restore return address
    addi $sp, $sp,8
    # adjust sp to pop 2 items
    mul $v0,$a0,$v0
    # return n * fact (n – 1)
    jr $ra # return to the caller
```


Allocating space on stack

- C has two storage class: automatic and static
  - *Automatic* variables are local to function and discarded when function exits.
  - *Static* variables exist across exits from and entries to procedures
- Can 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

Stack before, during, after call



Question?

```
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?

A. Print -4
B. Print 4
C. a.out will crash
D. 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" $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

Register Numbering

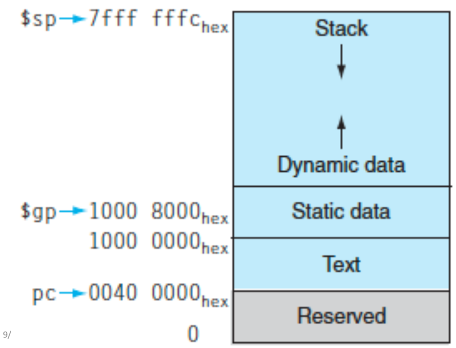
Name	Register number	Usage	Preserved on call?
$zero	0	The constant value 0	n.a.
$v0–$v1	2–3	Values for results and expression evaluation	no
$a0–$a3	4–7	Arguments	no
$t0–$t7	8–15	Temporaries	no
$s0–$s7	16–23	Saved	yes
$t8–$t9	24–25	More temporaries	no
$gp	28	Global pointer	yes
$sp	29	Stack pointer	yes
$fp	30	Frame pointer	yes
$ra	31	Return address	yes

Where is stack in memory?

- MIPS convention
- Stack starts in high memory and grows down
  - Hexadecimal (base 16) : 7fff $fffc_{hex}$
- MIPS programs (*text segment*) in low end
  - 0040 0000_{hex}
- *static data segment* (constants and other static variables) above text for static variables
  - MIPS convention *global pointer* ($gp) points to static
- Heap above static for data structures that grow and shrink ; grows up to high addresses

9/2/10 Fall 2010 -- Lecture #1 32

MIPS Memory Allocation



9/ 33

Summary

- 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
- Pointers/pointer arithmetic to reduce array overhead
  - No pointers to automatic data!
- Registers selectively saved/restored on call
  - Saved registers $s0-$s7; temporary regs $t0-$t9 not saved
- C splits memory into text, static, heap, stack, with registers dedicated to support: *$gp, $sp, $fp*

9/2/10 Fall 2010 -- Lecture #1 34