CS 61C:
Great Ideas in Computer Architecture
More MIPS, MIPS Functions

Instructors:
John Wawrzynek & Vladimir Stojanovic
http://inst.eecs.Berkeley.edu/~cs61c/fa15
Levels of Representation/Interpretation

- High Level Language Program (e.g., C)
- Assembly Language Program (e.g., MIPS)
- Machine Language Program (MIPS)

**Compiler**

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

**Assembler**

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

**Machine Interpretation**

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

- Binary Examples:
  - `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`

**Architecture Implementation**

- Logic Circuit Description (Circuit Schematic Diagrams)

- Hardware Architecture Description (e.g., block diagrams)
From last lecture ...

- Computer “words” and “vocabulary” are called *instructions* and *instruction set* respectively
- MIPS is example RISC instruction set used in CS61C
- Rigid format: 1 operation, 2 source operands, 1 destination
  - `add`, `sub`, `mul`, `div`, `and`, `or`, `sll`, `srl`, `sra`
  - `lw`, `sw`, `lb`, `sb` to move data to/from registers from/to memory
  - `beq`, `bne`, `j`, `slt`, `slti` for decision/flow control
- Simple mappings from arithmetic expressions, array access, in C to MIPS instructions
Review: Components of a Computer

Processor-Memory Interface

I/O-Memory Interfaces

Input

Output

Memory

Program

Bytes

Data

Enable?
Read/Write

Address

Write
Data

Read
Data

Program Counter

Registers

Arithmetic & Logic Unit (ALU)

Datapath

Control
How Program is Stored

One MIPS Instruction = 32 bits
Assembler to Machine Code (more later in course)

- foo.S
  - Assembler
  - foo.o
  - Linker
  - a.out

- bar.S
  - Assembler
  - bar.o
  - Linker
  - lib.o
  - Pre-built object file libraries
  - Machine code executable file

Assembler source files (text)
Assembler converts human-readable assembly code to instruction bit patterns
Machine code object files
The PC (program counter) is an internal register inside the processor holding the byte address of the next instruction to be executed.

Instruction is fetched from memory, then the control unit executes the instruction using the datapath and memory system, and updates the program counter (default is to add +4 bytes to the PC, to move to the next sequential instruction).
Review *if-else* Statement

- Assuming translations below, compile

\[
\begin{align*}
    f & \rightarrow \$s0 \\
    g & \rightarrow \$s1 \\
    h & \rightarrow \$s2 \\
    i & \rightarrow \$s3 \\
    j & \rightarrow \$s4
\end{align*}
\]

\[
\text{if (} i == j \text{)} \quad \text{bne} \quad \$s3,\$s4,\text{Else}
\]

\[
    f = g + h; \quad \text{add} \quad \$s0,\$s1,\$s2
\]

\text{else}

\[
    f = g - h; \quad \text{Else: sub} \quad \$s0,\$s1,\$s2
\]

\text{Exit:}
Control-flow Graphs: A visualization

\[ \text{bne } $s3, $s4, \text{Else} \]
\[ \text{add } $s0, $s1, $s2 \]
\[ \text{j Exit} \]
\[ \text{Else: sub } $s0, $s1, $s2 \]
\[ \text{Exit: } \ldots \]
Clickers/Peer Instruction

```assembly
addi $s0,$zero,0
Start:  slt $t0,$s0,$s1
        beq $t0,$zero,Exit
        sll $t1,$s0,2
        addu $t1,$t1,$s5
        lw $t1,0($t1)
        add  $s4,$s4,$t1
        addi $s0,$s0,1
        j Start
Exit:
```

What is the code above?
A: while loop
B: do ... while loop
C: for loop
D: A or C
E: Not a loop
Administrivia

• Fill-out the form to link bitbucket and edX accounts
  – Look-out for post on Piazza

• Advertising Guerrilla sections again
  – Tuesdays and Saturdays every two weeks

• CE applications approved for all students
CS61C In the News

• MIPS Creator CI20 dev board now available
  – A lot like Raspberry Pi but with MIPS CPU
  – Supports Linux and Android

• 1.2GHz 32-bit MIPS with integrated graphics

CS61C In the News pt. 2

RISC-V ANGEL:

• Try RISC-V in a browser
• http://riscv.org/angel/
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 code 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 ($4 - $7)
- $v0, v1$: two value registers to return values ($2,$3)
- $ra$: one return address register to return to the point of origin ($31)
In MIPS, all instructions are 4 bytes, and stored in memory just like data. So here we show the addresses of where the programs are stored.
... sum(a,b);... /* a,b:$s0,$s1 */
}  
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  ...  # next instruction
...
2000  sum:  add  $v0,$a0,$a1
2004  jr    $ra  # new instr. “jump register”
... sum(a,b);... /* a,b:$s0,$s1 */
}
int sum(int x, int y) {
    return x+y;
}

• Question: Why use jr here? Why not use j?

• Answer: sum might be called by many places, so we can’t return to a fixed place. The calling proc to sum must be able to say “return here” somehow.
Instruction Support for Functions (4/4)

• Single instruction to jump and save return address: jump and link (**jal**)

• Before:
  
  1008 addi $ra,$zero,1016  #$ra=1016
  1012 j sum  #goto sum

• After:
  
  1008 jal sum  # $ra=1012, goto sum

• Why have a **jal**?
  
  – Make the common case fast: function calls very common.
  – Don’t have to know where code is in memory with **jal**!
MIPS Function Call Instructions

• Invoke function: *jump and link* instruction (jal) (really should be *laj* “link and jump”)
  – “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 the following instruction in register $ra
    ```
    jal FunctionLabel
    ```

• 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 (*callee*) at address labeled X

• 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!

• What value does *jal X* place into $ra$? ????

• *jr $ra* puts address inside $ra$ back into PC
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 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 ($29$)

• Convention is grow from high to low addresses
  – Push decrements $sp$, Pop increments $sp$
Example

```c
int Leaf
 (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()

Leaf:  

```mips
addi $sp, $sp, -8  # adjust stack for 2 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 If a Function Calls a Function? Recursive Function Calls?

- Would clobber values in $a0 to $a3 and $ra
- What is the solution?
int sumSquare(int x, int y) {
    return mult(x,x)+ y;
}

• Something called `sumSquare`, now `sumSquare` is calling `mult`

• So there’s a value in `$ra` that `sumSquare` wants to jump back to, but this will be overwritten by the call to `mult`

Need to save `sumSquare` return address before call to `mult`
Nested Procedures (2/2)

• In general, may need to save some other info in addition to $ra$.

• When a C program is run, there are 3 important memory areas allocated:
  – **Static**: Variables declared once per program, cease to exist only after execution completes - e.g., C globals
  – **Heap**: Variables declared dynamically via `malloc`
  – **Stack**: Space to be used by procedure during execution; this is where we can save register values
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
   - `$sp`, `$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” `$t0$-$t9$, `$ra$
Clickers/Peer Instruction

• Which statement is FALSE?

A: MIPS uses jal to invoke a function and jr to return from a function
B: jal saves PC+1 in $ra
C: The callee can use temporary registers ($ti) without saving and restoring them
D: The caller can rely on save registers ($si) without fear of callee changing them
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
Stack Before, During, After Call

![Diagram showing stack changes before, during, and after a call]

- **a.** Stack before call: $fp$ is at a high address, $sp$ at a low address.
- **b.** Stack during call: $fp$ is at a high address, $sp$ at a low address. The stack contains saved argument registers, saved return address, saved saved registers, and local arrays and structures.
- **c.** Stack after call: $fp$ is at a high address, $sp$ at a low address.
Using the Stack (1/2)

• So we have a register $sp$ which always points to the last used space in the stack.
• To use stack, we decrement this pointer by the amount of space we need and then fill it with info.
• So, how do we compile this?

```c
int sumSquare(int x, int y) {
    return mult(x,x)+ y;
}
```
Using the Stack (2/2)

• Hand-compile

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

```
addi $sp,$sp,-8  # space on stack
sw $ra, 4($sp)  # save ret addr
sw $a1, 0($sp)  # save y
add $a1,$a0,$zero  # mult(x,x)
jal mult  # call mult
lw $a1, 0($sp)  # restore y
add $v0,$v0,$a1  # mult()+y
lw $ra, 4($sp)  # get ret addr
addi $sp,$sp,8  # restore stack
jr $ra
```

“push”

“pop”
Basic Structure of a Function

**Prologue**

entry_label:
```assembly
calli $sp,$sp, -framesize
sw $ra, framesize-4($sp)  # save $ra
```
save other regs if need be

**Body** \(\ldots\) (call other functions\(\ldots\))

**Epilogue**

```assembly
restore other regs if need be
lw $ra, framesize-4($sp)  # restore $ra
addi $sp,$sp, framesize
jr $ra
```
Where is the Stack in Memory?

- MIPS convention
- Stack starts in high memory and grows down
  - Hexadecimal (base 16) : $7ffe\, fffe_{\text{hex}}$
- 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

$sp \to 7fff \ ffff_{hex}$

$gp \to 1000 \ 8000_{hex}$

$1000 \ 0000_{hex}$

$pc \to 0040 \ 0000_{hex}$

Stack

Dynamic data

Static data

Text

Reserved
## Register Allocation and Numbering

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

- Functions called with \texttt{jal}, return with \texttt{jr}$ra$.
- The stack is your friend: Use it to save anything you need. Just leave it the way you found it!
- Instructions we know so far...
  - Arithmetic: \texttt{add, addi, sub, addu, addiu, subu}
  - Memory: \texttt{lw, sw, lb, sb}
  - Decision: \texttt{beq, bne, slt, slti, sltu, sltiu}
  - Unconditional Branches (Jumps): \texttt{j, jal, jr}
- Registers we know so far
  - All of them!
  - $a0$-$a3$ for function arguments, $v0$-$v1$ for return values
  - $sp$, stack pointer, $fp$ frame pointer, $ra$ return address
Bonus Slides
Recursive Function Factorial

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

Fact:

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

L1:

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

mul is a pseudo instruction