CS 61C:
Great Ideas in Computer Architecture
Intro to Assembly Language, MIPS Intro

Instructors:
Krste Asanovic & Vladimir Stojanovic
http://inst.eecs.Berkeley.edu/~cs61c/sp15
Levels of Representation/Interpretation

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

Assembly Language Program (e.g., MIPS)

Machine Language Program (MIPS)

Compiler

Assembler

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
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
- Control
- Datapath
  - Program Counter
  - Registers
  - Arithmetic & Logic Unit (ALU)

Memory
- Enable?
- Read/Write
- Address
- Write
- Data
- Read Data
- Bytes
- Program
- Data

Input

Output

Processor-Memory Interface

I/O-Memory Interfaces
How Program is Stored

Memory

Program

Bytes

Data

One MIPS Instruction = 32 bits
How Program is Executed: Instruction Fetch

The program counter (internal register inside processor) holds address of next instruction to be executed
Computer Decision Making

• Based on computation, do something different
• In programming languages: *if*-statement

• MIPS: *if*-statement instruction is

  \[ \text{beq register1, register2, L1} \]

means: \text{go to statement labeled L1}
if (value in register1) == (value in register2)

....otherwise, go to next statement

• \text{beq} stands for \textit{branch if equal}
• Other instruction: \texttt{bne} for \textit{branch if not equal}
Types of Branches

• **Branch** – change of control flow

• **Conditional Branch** – change control flow depending on outcome of comparison
  – branch *if* equal (\texttt{beq}) or branch *if not* equal (\texttt{bne})

• **Unconditional Branch** – always branch
  – a MIPS instruction for this: \texttt{jump (j)}
Example *if* Statement

- Assuming translations below, compile *if* block
  
  \[ f \rightarrow \$s0 \quad g \rightarrow \$s1 \quad h \rightarrow \$s2 \]
  
  \[ i \rightarrow \$s3 \quad j \rightarrow \$s4 \]

  ```
  if (i == j) bne \$s3,$s4,Exit
  f = g + h;
  add \$s0,$s1,$s2

  Exit:
  ```

- May need to negate branch condition
Example *if-else* Statement

- Assuming translations below, compile
  - \( f \rightarrow s_0 \quad g \rightarrow s_1 \quad h \rightarrow s_2 \)
  - \( i \rightarrow s_3 \quad j \rightarrow s_4 \)
  - if (\( i == j \))
    - \( f = g + h; \)
    - bne \( s_3, s_4, \text{Else} \)
    - add \( s_0, s_1, s_2 \)
  - else
    - \( f = g - h; \)
    - j Exit
    - Else: sub \( s_0, s_1, s_2 \)
    - Exit:
Inequalities in MIPS

• Until now, we’ve only tested equalities (== and != in C). General programs need to test < and > as well.

• Introduce MIPS Inequality Instruction:
  “Set on Less Than”

Syntax: \texttt{slt reg1,reg2,reg3}

Meaning: if (reg2 < reg3)
  \texttt{reg1 = 1;}
  else \texttt{reg1 = 0;}

“set” means “change to 1”,
“reset” means “change to 0”.
Inequalities in MIPS Cont.

• How do we use this? Compile by hand:
  
  ```
  if (g < h) goto Less; #g:$s0, h:$s1
  ```

• Answer: compiled MIPS code...
  
  ```
  slt $t0,$s0,$s1 # $t0 = 1 if g<h
  bne $t0,$zero,Less # if $t0!=0 goto Less
  ```

• Register $zero always contains the value 0, so `bne` and `beq` often use it for comparison after an `slt` instruction

• `sltu` treats registers as unsigned
Immediates in Inequalities

- `slti` an immediate version of `slt` to test against constants

```
Loop:  

slti $t0,$s0,1  # $t0 = 1 if $s0<1
beq $t0,$zero,Loop  # goto Loop
    # if $t0==0
    # (if ($s0>=1))
```
Clickers/Peer Instruction

Label: sll $t1,$s3,2
      addu $t1,$t1,$s5
lw  $t1,0($t1)
      add  $s1,$s1,$t1
addu $s3,$s3,$s4
bne  $s3,$s2,Label

What is the code above?
A: while loop
B: do ... while loop
C: for loop
D: Not a loop
E: Dunno
Clickers/Peer Instruction

• Simple loop in C; $A[]$ is an array of ints
  
  ```
  do {
    g = g + A[i];
    i = i + j;
  } while (i != h);
  ```

• Use this mapping: $g, h, i, j, &A[0], \$s1, \$s2, \$s3, \$s4, \$s5$

**Loop:**

```
sll $t1, \$s3, 2
addu $t1, $t1, \$s5
lw $t1, 0($t1)
add \$s1, \$s1, \$t1
addu \$s3, \$s3, \$s4
bne \$s3, \$s2, Loop
```

# $t1 = 4*i
# $t1 = addr A+4i
# $t1 = A[i]
# g = g + A[i]
# i = i + j
# goto Loop
# if i != h
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
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    $ra  # new instruction
... sum(a,b);... /* a,b:$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.

2000 sum: add $v0,$a0,$a1
2004 jr $ra    # new instruction
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 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
• Convention is grow from high to low addresses
  – Push decrements $sp, Pop increments $sp
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`, 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

  ```
  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?
Nested Procedures (1/2)

```c
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 $\texttt{ra}$.

• 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 \texttt{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
   - \$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
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 `$ra`
C: The callee can use temporary registers (`$t0`) without saving and restoring them
D: The caller can rely on the saved registers (`$si`) without fear of callee changing them
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 $ra
C: The callee can use temporary registers ($t$i) without saving and restoring them
D: The caller can rely on the saved registers ($si) without fear of callee changing them
Administrivia

- Hopefully everyone turned-in HW0
- HW1 due 11:59:59pm Sunday 2/8
In the News
MIPS for hobbyists

- 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

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
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;
}
```
• Hand-compile

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

```
sumSquare:    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
```

```
mult:        ...
```

Basic Structure of a Function

**Prologue**

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

**Body**  ...  (call other functions...)

**Epilogue**

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) : \(7fff \text{ fffc}_{\text{hex}}\)
  - **MIPS programs** (*text segment*) in low end
    - \(0040 \text{ 0000}_{\text{hex}}\)
  - **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
MIPS Memory Allocation

$sp \rightarrow 7fff \ fff c_{hex}$

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

$1000 \ 0000_{hex}$

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

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} \texttt{\$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}, \texttt{addi}, \texttt{sub}, \texttt{addu}, \texttt{addiu}, \texttt{subu}
  Memory: \texttt{lw}, \texttt{sw}, \texttt{lb}, \texttt{sb}
  Decision: \texttt{beq}, \texttt{bne}, \texttt{slt}, \texttt{slti}, \texttt{sltu}, \texttt{sltiu}
  Unconditional Branches (Jumps): \texttt{j}, \texttt{jal}, \texttt{jr}
• Registers we know so far
  – All of them!
  – \texttt{\$a0-\$a3} for function arguments, \texttt{\$v0-\$v1} for return values
  – \texttt{\$sp}, stack pointer, \texttt{\$fp} frame pointer, \texttt{\$ra} return address
Bonus Slides
Recursive Function Factorial

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