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

Instructors:
Vladimir Stojanovic and Nicholas Weaver
http://inst.eecs.Berkeley.edu/~cs61c/sp16
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
```
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

```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`
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
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

- **a.** High address
- **b.** Saved argument registers (if any)
- **c.** Saved return address

- **Low address**
- **High address**
- **$fp$**
- **$sp$**

- **Local arrays and structures (if any)**
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;
}
```

```
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:  ...
```

“push”

```
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
```

“pop”

```
lw $ra, 4($sp)  # get ret addr
addi $sp,$sp,8  # restore stack
jr $ra
```
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
MIPS Memory Allocation

- 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>
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
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 $ra
C: The callee can use temporary registers ($t1) without saving and restoring them
D: The caller can rely on save registers ($s1) without fear of callee changing them
Administrivia

- Project 1 now due Tuesday 2/9 @ 23:59:59
- Project 2-1 released the next day, due 2/16 @ 23:59:59 (a Tuesday as well)
- Guerrilla Sessions
  - MIPS I, Wed 2/10 3 - 5 PM @ TBD
  - MIPS II, Sat 2/13 1 - 3 PM @ 521 Cory
ENIAC (U. Penn., 1946)
First Electronic General-Purpose Computer

- Blazingly fast (multiply in 2.8ms!)
  - 10 decimal digits x 10 decimal digits
- But needed 2-3 days to setup new program, as programmed with patch cords and switches
Big Idea: Stored-Program Computer

- Instructions are represented as bit patterns - can think of these as numbers
- Therefore, entire programs can be stored in memory to be read or written just like data
- Can reprogram quickly (seconds), don’t have to rewire computer (days)
- Known as the “von Neumann” computers after widely distributed tech report on EDVAC project
  - Wrote-up discussions of Eckert and Mauchly
  - Anticipated earlier by Turing and Zuse
EDSAC (Cambridge, 1949)
First General Stored-Program Computer

- Programs held as numbers in memory
- 35-bit binary 2’s complement words
Consequence #1: Everything Addressed

• Since all instructions and data are stored in memory, everything has a memory address: instructions, data words
  – both branches and jumps use these

• C pointers are just memory addresses: they can point to anything in memory
  – Unconstrained use of addresses can lead to nasty bugs; up to you in C; limited in Java by language design

• One register keeps address of instruction being executed: “Program Counter” (PC)
  – Basically a pointer to memory: Intel calls it Instruction Pointer (a better name)
Consequence #2: Binary Compatibility

• Programs are distributed in binary form
  – Programs bound to specific instruction set
  – Different version for Macintoshes and PCs

• New machines want to run old programs ("binaries") as well as programs compiled to new instructions

• Leads to “backward-compatible” instruction set evolving over time

• Selection of Intel 8086 in 1981 for 1st IBM PC is major reason latest PCs still use 80x86 instruction set; could still run program from 1981 PC today
Instructions as Numbers (1/2)

• Currently all data we work with is in words (32-bit chunks):
  – Each register is a word.
  – `lw` and `sw` both access memory one word at a time.

• So how do we represent instructions?
  – Remember: Computer only understands 1s and 0s, so “`add $t0, $0, $0`” is meaningless.
  – MIPS/RISC seeks simplicity: since data is in words, make instructions be fixed-size 32-bit words also
Instructions as Numbers (2/2)

• One word is 32 bits, so divide instruction word into “fields”.
• Each field tells processor something about instruction.
• We could define different fields for each instruction, but MIPS seeks simplicity, so define 3 basic types of instruction formats:
  – R-format
  – I-format
  – J-format
Instruction Formats

• **I-format**: used for instructions with immediates, `lw` and `sw` (since offset counts as an immediate), and branches (`beq` and `bne`) – (but not the shift instructions; later)
• **J-format**: used for `j` and `jal`
• **R-format**: used for all other instructions
• It will soon become clear why the instructions have been partitioned in this way
R-Format Instructions (1/5)

• Define “fields” of the following number of bits each: $6 + 5 + 5 + 5 + 5 + 6 = 32$

| 6 | 5 | 5 | 5 | 5 | 6 |

• For simplicity, each field has a name:

| opcode | rs | rt | rd | shamt | funct |

• Important: On these slides and in book, each field is viewed as a 5- or 6-bit unsigned integer, not as part of a 32-bit integer
  
  – Consequence: 5-bit fields can represent any number 0-31, while 6-bit fields can represent any number 0-63
R-Format Instructions (2/5)

• What do these field integer values tell us?
  – **opcode**: partially specifies what instruction it is
    • Note: This number is equal to 0 for all R-Format instructions
  – **funct**: combined with **opcode**, this number exactly specifies the instruction

• Question: Why aren’t **opcode** and **funct** a single 12-bit field?
  – We’ll answer this later
R-Format Instructions (3/5)

• More fields:
  – **rs** (Source Register): *usually* used to specify register containing first operand
  – **rt** (Target Register): *usually* used to specify register containing second operand (note that name is misleading)
  – **rd** (Destination Register): *usually* used to specify register which will receive result of computation
R-Format Instructions (4/5)

• Notes about register fields:
  – Each register field is exactly 5 bits, which means that it can specify any unsigned integer in the range 0-31. Each of these fields specifies one of the 32 registers by number.
  – The word “usually” was used because there are exceptions that we’ll see later
R-Format Instructions (5/5)

• Final field:
  – **shamt**: This field contains the amount a shift instruction will shift by. Shifting a 32-bit word by more than 31 is useless, so this field is only 5 bits (so it can represent the numbers 0-31)
  – This field is set to $0$ in all but the shift instructions

• For a detailed description of field usage for each instruction, see green insert in COD (You can bring with you to all exams)
R-Format Example (1/2)

• MIPS Instruction:
  \texttt{add} \quad \$8, \$9, \$10

\begin{align*}
\texttt{opcode} &= 0 \text{ (look up in table in book)} \\
\texttt{funct} &= 32 \text{ (look up in table in book)} \\
\texttt{rd} &= 8 \text{ (destination)} \\
\texttt{rs} &= 9 \text{ (first operand)} \\
\texttt{rt} &= 10 \text{ (second operand)} \\
\texttt{shamt} &= 0 \text{ (not a shift)}
\end{align*}
R-Format Example (2/2)

- **MIPS Instruction:**
  
  \[
  \text{add} \quad \$8, \$9, \$10
  \]

  Decimal number per field representation:

<table>
<thead>
<tr>
<th>0</th>
<th>9</th>
<th>10</th>
<th>8</th>
<th>0</th>
<th>32</th>
</tr>
</thead>
</table>

  Binary number per field representation:

  \[
  000000 \quad 01001 \quad 01010 \quad 01000 \quad 00000 \quad 100000
  \]

  hex representation: \[012A \quad 4020_{\text{hex}}\]

  Called a **Machine Language Instruction**