Lecture 7: Procedures

CSE 30: Computer Organization and Systems Programming
Winter 2010

Rajesh Gupta / Ryan Kastner
Dept. of Computer Science and Engineering
University of California, San Diego
Outline

- Composition and Composability
  - Composition without interfaces
  - Composition with interfaces
- Composing Programs
- Composing Procedural Programs in C on MIPS

Reference: P+H 2.8
C functions

```c
main() {
    int a, b, c;
    ...
    c = sum(a, b); /* a, b, c: $s0, $s1, $s2*/
    ...
}

/* really dumb sum function */
int sum(int x, int y) {
    return x + y;
}
```

What information must the compiler/programmer keep track of?
What instructions can accomplish this?
Registers play a major role in keeping track of information for function calls

Register conventions:
- Return address: $ra
- Arguments: $a0, $a1, $a2, $a3
- Return value: $v0, $v1
- Local variables: $s0, $s1, ..., $s7

The stack is also used; more later
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.
Instruction Support for Functions

```c
... sum(a,b);... /* a,b:$s0,$s1 */
}

int sum(int x, int y) {
    return x+y;
}
```

```assembly
address
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 ...  
2000   sum: add $v0,$a0,$a1
2004   jr $ra             # new instruction
```
Instruction Support for Functions

... sum(a,b);... /* a,b:$s0,$s1 */
}

C int sum(int x, int y) {
    return x+y;
}

Question: Why use \texttt{jr} here? Why not simply use \texttt{j}?

Answer: \texttt{sum} might be called by many functions, so we can’t return to a fixed place. The calling proc to \texttt{sum} must be able to say “return here” somehow.

\texttt{sum}: add $v0,$a0,$a1
\texttt{jr} $ra  # new instruction
Instruction Support for Functions

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

- Before:
  ```
  1008  addi $ra,$zero,1016  #$ra=1016
  1012  j  sum  #go to sum
  ```

- After:
  ```
  1008  jal  sum  # $ra=1012,go to sum
  ```

- Why have a jal? Make the common case fast: function calls are very common. Also, you don’t have to know where the code is loaded into memory with jal.
Instruction Support for Functions

- Syntax for `jal` (jump and link) is same as for `j` (jump):
  
  ```
  jal label
  ```

- `jal` should really be called `laj` for “link and jump”:
  - Step 1 (link): Save address of next instruction into `$ra` (Why next instruction? Why not current one?)
  - Step 2 (jump): Jump to the given label
Instruction Support for Functions

- Syntax for \texttt{jr} (jump register):
  \texttt{jr} \ register

- Instead of providing a label to jump to, the \texttt{jr} instruction provides a register which contains an address to jump to

- Only useful if we know exact address to jump

- Very useful for function calls:
  - \texttt{jal} stores return address in register (\$ra)
  - \texttt{jr} \ $ra jumps back to that address
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`.
Nested Procedures

- 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
  - **Stack**: Space to be used by procedure during execution; this is where we can save register values
C Memory Allocation

Address $\infty$

$\texttt{sp}$

stack pointer

Stack

Space for saved procedure information

Heap

Explicitly created space, e.g., malloc(); C pointers

Static

Variables declared once per program

Code

Program
Using the Stack

- 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

Hand-compile

```
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
```

“push”

“pop”
Steps for Making a Procedure Call

1) Save necessary values onto stack
2) Assign argument(s), if any
3) jal call
4) Restore values from stack
Rules for Procedures

- Called with a \texttt{jal} instruction, returns with a \texttt{jr} $\texttt{ra}$
- Accepts up to 4 arguments in $\texttt{a0}$, $\texttt{a1}$, $\texttt{a2}$ and $\texttt{a3}$
- Return value is always in $\texttt{v0}$ (and if necessary in $\texttt{v1}$)
- Must follow \texttt{register conventions} (even in functions that only you will call)! So what are they?
# MIPS Registers

<table>
<thead>
<tr>
<th>Description</th>
<th>Register</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>The constant 0</td>
<td>$0</td>
<td>$zero</td>
</tr>
<tr>
<td>Reserved for Assembler</td>
<td>$1</td>
<td>$at</td>
</tr>
<tr>
<td>Return Values</td>
<td>$2-$3</td>
<td>$v0-$v1</td>
</tr>
<tr>
<td>Arguments</td>
<td>$4-$7</td>
<td>$a0-$a3</td>
</tr>
<tr>
<td>Temporary</td>
<td>$8-$15</td>
<td>$t0-$t7</td>
</tr>
<tr>
<td>Saved</td>
<td>$16-$23</td>
<td>$s0-$s7</td>
</tr>
<tr>
<td>More Temporary</td>
<td>$24-$25</td>
<td>$t8-$t9</td>
</tr>
<tr>
<td>Used by Kernel</td>
<td>$26-27</td>
<td>$k0-$k1</td>
</tr>
<tr>
<td>Global Pointer</td>
<td>$28</td>
<td>$gp</td>
</tr>
<tr>
<td>Stack Pointer</td>
<td>$29</td>
<td>$sp</td>
</tr>
<tr>
<td>Frame Pointer</td>
<td>$30</td>
<td>$fp</td>
</tr>
<tr>
<td>Return Address</td>
<td>$31</td>
<td>$ra</td>
</tr>
</tbody>
</table>

Use names for registers -- code is clearer!
**Other Registers**

- `$at`: may be used by the assembler at any time; unsafe to use
- `$k0–$k1`: may be used by the OS at any time; unsafe to use
- `$gp`, `$fp`: don’t worry about them

Note: Feel free to read up on `$gp` and `$fp` in Appendix A, but you can write perfectly good MIPS code without them.
Register Conventions

- **Calle\(\text{R}\)**: the calling function
- **Calle\(\text{E}\)**: the function being called

When callee returns from executing, the caller needs to know which registers may have changed and which are guaranteed to be unchanged.

**Register Conventions**: A set of generally accepted rules as to which registers will be unchanged after a procedure call (jal) and which may be changed.
Saved Register Conventions

- $0: No Change. Always 0.
- $s0-$s7: Restore if you change. Very important, that’s why they’re called saved registers. If the callee changes these in any way, it must restore the original values before returning.
- $sp: Restore if you change. The stack pointer must point to the same place before and after the jal call, or else the caller won’t be able to restore values from the stack.
- HINT -- All saved registers start with S!
Volatile Register Conventions

- $ra: Can Change. The \texttt{jal} call itself will change this register. \texttt{Caller} needs to save on stack if nested call.

- $v0-$v1: Can Change. These will contain the new returned values.

- $a0-$a3: Can change. These are volatile argument registers. \texttt{Caller} needs to save if they’ll need them after the call.

- $t0-$t9: Can change. That’s why they’re called temporary: any procedure may change them at any time. \texttt{Caller} needs to save if they’ll need them afterwards.
Register Conventions

- What do these conventions mean?
  - If function R calls function E, then function R must save any temporary registers that it may be using onto the stack before making a `jal` call.
  - Function E must save any S (saved) registers it intends to use before garbling up their values.
  - Remember: Caller/callee need to save only temporary/saved registers they are using, not all registers.
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
Example

```c
main() {
    int i, j, k, m; /* i-m:$s0-$s3 */
    ...
    i = mult(j, k); ...
    m = mult(i, i); ...
}

int mult (int mcand, int mlier) {
    int product;
    product = 0;
    while (mlier > 0) {
        product += mcand;
        mlier -= 1;
    }
    return product;
}
```
Example

```
main() {
    int i,j,k,m; /* i-m:$s0-$s3 */

    i = mult(j,k); ...  
    m = mult(i,i); ...  
}

__start:
    add $a0,$s1,$0    # arg0 = j
    add $a1,$s2,$0    # arg1 = k
    jal mult          # call mult
    add $s0,$v0,$0    # i = mult()

    ...

    add $a0,$s0,$0    # arg0 = i
    add $a1,$s0,$0    # arg1 = i
    jal mult          # call mult
    add $s3,$v0,$0    # m = mult()

    ...

done
```
Notes:

- `main function ends with done, not jr $ra, so there’s no need to save $ra onto stack`
- `All variables used in main function are saved registers, so there’s no need to save these onto stack`
Example

```c
int mult (int mcand, int mller) {
    int product = 0;
    while (mller > 0) {
        product += mcand;
        mller -= 1;
    }
    return product;
}

mult: add $t0,$0,$0 # prod=0

Loop: slt $t1,$0,$a1 # mller > 0?
    beq $t1,$0,Fin  # no=>Fin
    add $t0,$t0,$a0 # prod+=mc
    addi $a1,$a1,-1 # mller-=1
    j Loop # goto Loop

Fin: add $v0,$t0,$0 # $v0=prod
    jr $ra # return
```
Example

Notes:

- No jal calls are made from mult and we don’t use any saved registers, so we don’t need to save anything onto stack.
- Temp registers are used for intermediate calculations (could have used s registers, but would have to save the caller’s on the stack.)
- $a1 is modified directly (instead of copying into a temp register) since we are free to change it.
- Result is put into $v0 before returning.
Conclusion

- Functions are called with `jal`, and return with `jr $ra`.

- The stack is your friend: Use it to save anything you need. Just be sure to leave it the way you found it.

- **Register Conventions**: Each register has a purpose and limits to its usage. Learn these and follow them, even if you’re writing all the code yourself.
Conclusion

- Instructions we know so far:
  - Arithmetic: add, addi, addu, addiu, sub, subi, subu, mult, div, multu, divu
  - Bitwise: and, andi, or, ori,
  - Memory: lw, sw, lb, lbu, sb
  - Decision: beq, bne, slt, slti, sltu, sltiu
  - Unconditional Branches (Jumps): j, jal, jr

- Registers we know so far
  - All of them!