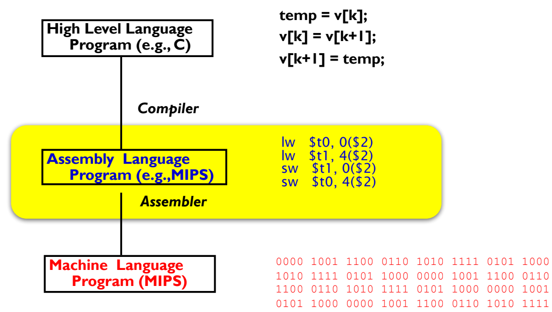
Lecture 6: Introduction to MIPS -Functions

(CPEG323: Intro. to Computer System Engineering)

Levels of Program Code



Implementing Functions in MIPS

Functions in C

```
main() {
    int i,j;
    i = factorial(10);
    ...
    j = factorial(25);
}
int factorial(int n) {
    if (n<1) return 1;
    return n*factorial(n-1);
}
```

What happens when making function calls?

Functions in C (step 1)

```
main() {
    int i,j;
    i = factorial(10);
    ...
    j = factorial(25);
}
int factorial(int n) {
    if (n<1) return 1;
    return n*factorial(n-1);
}
```

The program's flow of control must be changed.

Functions in C (step 2)

```
main() {
    int i,j;
    i = factorial(10);
    ...
    j = factorial(25);
}
int factorial(int n) {
    if (n<1) return 1;
    return n*factorial(n-1);
}
```

Arguments and return values are passed back and forth

Functions in C (step 3)

```
main() {
    int i,j;
    i = factorial(10);
    ...
    j = factorial(25);
}
int factorial(int n) {
    if (n<1) return 1;
    return n*factorial(n-1);
}
```

Local variables are allocated and then destroyed.

MIPS Function Call Example (1/4)

```
… sum(a,b);… /* a,b:$s0,$s1 */
}
int sum(int x, int y) {
    return x+y;
}
```

address (in decimal)

1000
1004
1008
1012
1016
2000
2004

All MIPS instructions are 4 bytes, and stored in memory just like data.

MIPS Function Call Example

```
… sum(a,b);… /* a,b:$s0,$s1 */
}
int sum(int x, int y) {
    return x+y;
}
```

address (in decimal)

```
1000   add  $a0,$s0,$0      # x = a
1004   add  $a1,$s1,$0      # y = b
1008   addi $ra,$0,1016     #$ra=1016
1012   j    sum             #jump to sum
1016   ra: ...
 …
2000   sum: add $v0,$a0,$a1
2004   j    ra              # return to the caller
```


MIPS Function Call Example

```
… sum(a,b);… /* a,b:$s0,$s1 */
}
int sum(int x, int y) {
    return x+y;
}
```

address (in decimal)

```
1000   add  $a0,$s0,$0      # x = a
1004   add  $a1,$s1,$0      # y = b
1008   addi $ra,$0,1016     #$ra=1016
1012   j    sum             #jump to sum
1016   …
2000   sum: add $v0,$a0,$a1
2004   jr   $ra             # new instruction
```


MIPS Function Call Example

```
… sum(a,b);… /* a,b:$s0,$s1 */
}
int sum(int x, int y) {
    return x+y;
}
```

address (in decimal)

```
1000   add  $a0,$s0,$0      # x = a
1004   add  $a1,$s1,$0      # y = b
1008   jal sum              # $ra=1012, jump to sum
1012   …
2000   sum: add $v0,$a0,$a1
2004   jr   $ra             # new instruction
```


MIPS Function Calls (1) – Registers

- Registers are used to store information related to function calls.
  - Registers are much faster than memory.
- Special registers are used.
  - $a0–$a3: four *argument* registers to pass parameters
  - $v0–$v1: two *value* registers to return values
  - $ra: one *return address* register to save where a function is called from.
  - $s0-$s7: local variables

MIPS Function Calls (2) – Instructions

- Make a function call: **jal**
  - jump and link instruction: jal FunctionName
  - Jumps to label and simultaneously saves the location of following instruction in register $ra
- Return from function: **jr**
  - jump register instruction: jr $ra
  - Unconditional jump to address specified in register $ra

MIPS Function Calls (3) – Summary

- *Caller* Function
  - Put parameters into registers $a0 to $a3
  - Invoke callee X using **jal X**.
    - PC (Program counter) is a special register used to store the address of currently executed instruction.
    - Jal puts PC+4 into $ra, then jumps to label X
- *Callee* Function
  - Read parameters from register $a0 to $a3.
  - Execute instructions inside the function
  - Store return values in $v0 and $v1.
  - Return to caller function using **jr $ra**
    - It puts address inside $ra into PC.

Wait a minute! How about nested function calls?

- What happens when A calls B and B calls C?
  - The arguments for the call to C would be placed in $a0-$a3, thus *overwriting* the original arguments for B.
  - Similarly, jal C overwrites the return address that was saved in $ra by the earlier jal B.

```
A:   ...
     # Put B's args in $a0-$a3
     jal  B                $ra=A2
A2:  ...
```

```
B:   ...                   Erasing B's arguments!
     # Put C's args in $a0-$a3,

     jal  C                $ra=B2
B2:  ...

     jr   $ra              Where will it go?
```

```
C:   ...
     jr   $ra
```

Solution to the Overwriting Problem

- Spill registers to memory
  - Save "important" registers to memory (in particular, stack) before the function call
  - Restore these registers after the function call
- Who spills? Caller or callee?

Calling Conventions

Who saves the registers?

- Option 1 - Caller!
  - The caller knows which registers are important to it and should be saved. So caller should save.
- Option 2 – Callee!
  - The callee knows exactly which registers it will use and potentially overwrite. So callee should save.

Both approaches may wastefully save registers they don't really need to.

- Final solution – divide the job! The caller and callee together save all of the important registers.
  - Caller assumes callee will destroy: $t0-$t9 $a0-$a3 $v0-$v1
  - Callee assumes caller will need: $s0-$s7 $ra

Example

- frodo (caller) only needs to save registers $a0 and $a1, while gollum (callee) only has to save registers $s0 and $s2.

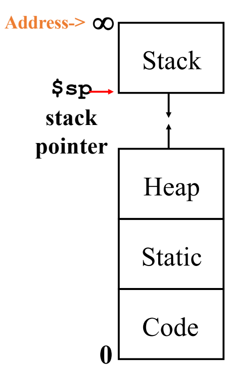
```
frodo: li   $a0, 3              gollum:
       li   $a1, 1
       li   $s0, 4
       li   $s1, 1
                                        li   $a0, 2
                                        li   $a2, 7
                                        li   $s0, 1
                                        li   $s2, 8
       jal  gollum                      ...



       add  $v0, $a0, $a1               jr   $ra
       add  $v1, $s0, $s1
       jr   $ra
```

Where are the registers saved?

- Memory!

- Each function call should have its own private memory area.
  - This would prevent other function calls from overwriting the saved registers—otherwise using memory is no better than using registers.
  - We could use this private memory for other purposes too, like storing local variables.

Recall: Memory Layout



Space for saved procedure information

Dynamic allocated space

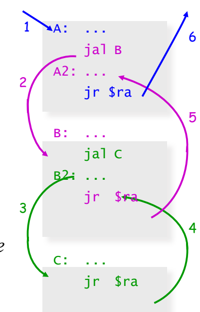
Variables declared once per program; e.g., globals

Program

Stacks and Functions Calls

- Notice function calls and returns occur in a stack-like order: the most recently called function is the first one to return.
  1. Someone calls A
  2. A calls B
  3. B calls C
  4. C returns to B
  5. B returns to A
  6. A returns

- Here, for example, C must return to B *before* B can return to A.



Stacks and function calls (Cont.)

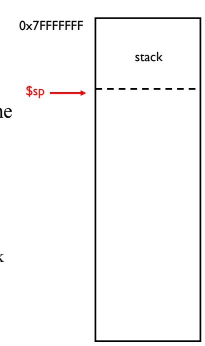
- It's natural to use a stack for function call storage.
- A block of stack space, called a stack frame, can be allocated for each function call.
  - When a function is called, it creates a new frame onto the stack, which will be used for local storage.
  - Before the function returns, it must pop its stack frame, to restore the stack to its original state.
- The stack frame can be used for several purposes.
  - Caller- and callee-save registers can be put in the stack.
  - The stack frame can also hold local variables, or extra arguments and return values.

The MIPS stack

- The stack grows downward in terms of memory addresses.

- The address of the newest element of the stack is stored in the "stack pointer" register, $sp.

- Instructions related to MIPS
  - "Push" – adding an element in the stack
  - "Pop" – removing an element from the stack
  - MIPS does not provide "push" and "pop" instructions. Instead, they must be done explicitly by the programmer.



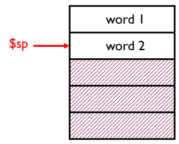
Pushing elements

- To push elements onto the stack:
  - Move the stack pointer $sp down to make room for the new data.
  - Store the elements into the stack.
- For example, to push registers $t1 and $t2 onto the stack:

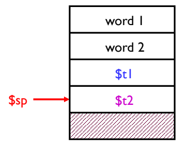
```
sub $sp, $sp, 8
sw  $t1, 4($sp)
sw  $t2, 0($sp)
```

- An equivalent sequence is:

```
sw  $t1, -4($sp)
sw  $t2, -8($sp)
sub $sp, $sp, 8
```



Before

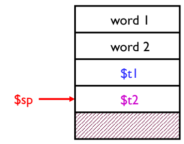


After

Accessing and popping elements

- Access an element
  - Compute its memory address based on the $sp, and read it from the memory
  - For example, to retrieve the value of $t1:

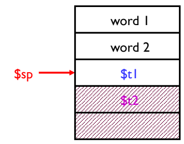
```
lw   $s0, 4($sp)
```



- Pop (or erase) an element
  - Adjust the stack pointer upwards.
  - For example, to pop the value of $t2:

```
addi   $sp, $sp, 4
```

  - Note that the popped data are still present in memory, but data past the stack pointer are considered invalid.



An Example of Function Call

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

What should the calling function store before the function call?

- $ra,   since it will be overwritten when calling *mult*
- $a1, need to reuse $a1 to pass second argument to mult, but need current value (y) later.

An Example of Function Call (cont.)

```
                              int sumSquare(int x, int y) {
sumSquare:                         return mult(x,x)+ y; }
          addi $sp,$sp,-8     # make space on stack
          sw $ra, 4($sp)      # save ret addr
"push"    sw $a1, 0($sp)      # save y
          add $a1,$a0,$zero   # set 2nd mult arg
          jal mult            # call mult
          lw $a1, 0($sp)      # restore y
          add $v0,$v0,$a1     # ret val = mult(x,x)+y
"pop"     lw $ra, 4($sp)      # get ret addr
          addi $sp,$sp,8      # restore stack
          jr $ra
mult: ...
```

Another Example

```
int plusOne(int x) {
  return x + 1;
}

void main() {
  int x = 5;
  x += plusOne(x);
}
```

```
plusOne:  # a0 = x
  addi  $v0, $a0, 1
  jr    $ra

main:
  li    $a0, 5
  addi  $sp, $sp, -8
  sw    $ra, 0($sp)
  sw    $a0, 4($sp)
  jal   plusOne
  lw    $ra, 0($sp)
  lw    $a0, 4($sp)
  addi  $sp, $sp, 8
  add   $a0, $a0, $v0
  jr    $ra
```


Summary: Function Calls in MIPS (1)

- Instructions:
  - To call a function: jal func
    - It sets $ra to address of instruction after jal
  - To return from a function: jr $ra
- Registers
  - arguments "passed" in registers: $a0, …, $a3
  - return values in registers: $v0, $v1
  - Calling convention
    - Caller needs to save: $t0-$t9 $a0-$a3 $v0-$v1
    - Callee needs to save: $s0-$s7 $ra

Summary: Function Calls in MIPS (2)

- To save *k* registers onto stack:
  - Grow stack by subtracting 4*k from $sp
  - Store the elements into the stack (array)
- Examples:
  - Push $t1 and $t2 onto stack

```
addi  $sp, $sp, -8
sw    $t1, 0($sp)
sw    $t2, 4($sp)
```

  - Restore registers by reading from stack e.g.:

```
lw    $t2, 4($sp)
```

  - Remember to restore $sp to its original value:

```
addi  $sp, $sp, 8
```


Recursive Functions

Recursive Functions

- Recall that recursive functions have one or more base-cases, and one or more recursive calls. Example:

```
int rec_max(int *array, int n) {
  if(n == 1) return array[0];
  return max(array[n-1], rec_max(array, n-1));
}
```

- Useful tip: Translate the base case first

```
rec_max:  # $a0 = array = &array[0],   $a1 = n
  bne  $a1, 1, rec_case
  lw   $v0, 0($a0)       # return-value = array[0]
  jr   $ra
rec_case: ...
```


The Recursive Case

- Let's examine the recursive step more carefully:

```
return max(array[n-1], rec_max(array, n-1));
```

- Useful tip: Figure out what we need to remember across the recursive function call: array[n-1]   (array, n-1)

```
rec_case:
  addi   $sp, $sp, -12 # save space for 3 regs
  addi   $a1, $a1, -1  # compute n-1
  sw     $a0, 0($sp)   # save &array[0]
  sw     $a1, 4($sp)   # save n-1
  sw     $ra, 8($sp)   # save $ra, since I'm doing jal!
  jal    rec_max       # recursive call with new args
  # restore $a0, $a1 and $ra
  # compare array[n-1] and $v0, and put larger into $v0
```


Summary

- **General rules for making functions calls**
  - Calls a function with a jal function, returns with a jr $ra
  - Accepts up to 4 arguments in $a0-$a3.
  - Return value is always in $v0 (and $v1).
  - Must follow register calling conventions
- **Specific steps for making a function call**
  - Save necessary registers onto the stack
  - Assign arguments, if any
  - Call the function (jal)
  - Restore register values from the stack

Reading

- 5th Edition: 2.8