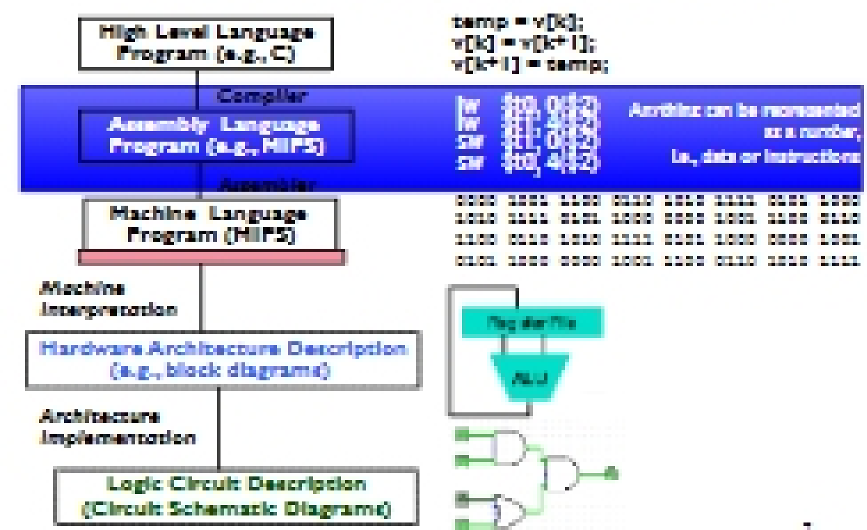
Lecture 6: Introduction to MIPS -Functions

(CPEG323: Intro. to Computer System Engineering)

1

Levels of Representation



2

Implementing Functions in MIPS

3

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?

4

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.

5

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

6

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);
}
```

Local variables can be allocated and destroyed.

Function Call Example (1/4)

C

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

MIPS

address (shown in decimal)

```
1000
1004
1008
1012
1016

2000
2004
```

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.

Function Call Example (2/4)

C

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

MIPS

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 ...

2000 sum: add $v0,$a0,$a1
2004 jr   $ra                # new instruction
```

Function Call Example (3/4)

C

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

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

MIPS

address (shown in decimal)

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

Function Call Example (4/4)

C

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

MIPS

- Question: Why use jr here? Why not simply 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
```

MIPS Registers for Function Calls

Registers way faster than memory, so use registers

- $a0-$a3: four *argument* registers to pass parameters
- $v0-$v1: two *value* registers to return values
- $ra: one *return address* register that saves *where a function is called from.*
- $s0-$s7: local variables

MIPS Instructions for Function Calls

- Invoke function: *jump and link* instruction (jal)
  - "link" means storing the location of the calling site.
  - Jumps to label and simultaneously saves the location of following instruction in register $ra

    jal ProcedureAddress
- Return from function: *jump register* instruction (jr)
  - Unconditional jump to address specified in register

    jr $ra

Warnings and Problems

- A problem: What if a function uses a register that the main program needs *after* the function call?
  - Things get really nasty with nested/recursive functions

Nested functions

- Let's say A calls B, which 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:   ...
     # Put C's args in $a0-$a3,
     # erasing B's args!
     jal  C        # $ra = B2
B2:  ...
     jr   $ra      # Where does
                   # this go???
```

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

Solution

- Solution: *spill* registers to memory (stack)
  - save "important" registers to memory before function call
  - restore these registers after the function call
- Who spills? Caller or callee?

Calling Conventions

Who saves the registers?

- Argument 1: The caller knows which registers are important to it and should be saved. So caller should save.
- Argument 2: 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.
- But the caller and callee must not assume anything about each other
  - may be written by different people or companies
  - should be able to interface with any caller/callee
- Solution:

  Caller assumes callee will destroy: $t0-$t9 $a0-$a3 $v0-$v1

  Callee assumes caller will need: $s0-$s7 $ra