Functions in Assembly

What is a function?

```
int foo(a, b) {
    int c = a + b;
    return c; }

int d = foo(2, 4);
// keep going from here
```

What is a function?

```
int foo(a, b) {
        int c = a + b;
        return c;
}

int d = foo(2, 4);
// keep going from here
```

Must be able to

- Pass control to elsewhere in code
- Pass arguments
- Return back to original place in code
- Return result of function

Going to functions

```
fun:
     addi $t0, $0, 1

main:
     addi $s0, $0, 5
     sll $s1, $s0, 2
     j fun
     add $s2, $s0, $s1
```

Moving from one place to another in code is easy – use a jump

Going to functions

Moving from one place to another in code is easy – use a jump

Take another look at previous code and try stepping through it. What happens?

Going to functions

```
fun:
     addi $t0, $0, 1

main:
     addi $s0, $0, 5
     sll $s1, $s0, 2
     j fun
     add $s2, $s0, $s1
```

Infinite loop!

Going to functions

Infinite loop!

“Function” does not return where we want it to

Assembly never does anything we do not explicitly tell it to do. Unless told to jump/branch, it continues on to next instruction

Returning from functions

Solution is to jump back from function – but how?

Function should be callable from many different places. Do not want to jump to one specific spot in code when function finishes

Use special instruction designed for this purpose: **jump and link** (`jal`)

Returning from functions

`jal` jumps like any other jump, but also saves location it jumped *from* into a register

That register tells us where to go back to when function finishes

Special register for this purpose is `$ra` (return address)

`$ra` automatically set when `jal` called

Returning from functions

Recall we had another jump command that jumps to location specified in register rather than jumping to label

`jr` (jump register)

`jr` and `jal` used mainly for implementing functions

`jal` sets `$ra` and goes to function; function ends with `jr $ra` to go back

Returning from functions

```
fun:
     addi $t0, $0, 1
     jr $ra

main:
     addi $s0, $0, 5
     sll $s1, $s0, 2
     jal fun
     add $s2, $s0, $s1  # jr returns here
```

Function Calls

C Code

```
int main() {
  simple();
  a = b + c;
}

void simple() {
  return;
}
```

MIPS assembly code

```
0x00400200 main: jal  simple
0x00400204       add  $s0, $s1, $s2
...

0x00401020 simple: jr $ra
```

jal: jumps to `simple`
`$ra` = PC + 4 = 0x00400204

jr $ra: jumps to address in `$ra` (0x00400204)

Taken from Harris and Harris

Arguments and return values

Must be able to

- ~~Pass control to elsewhere in code~~
- Pass arguments
- ~~Return back to original place in code~~
- Return result of function

How do we get values to different parts of code? Consider where we store values in the first place

Arguments and return values

Almost nothing happens behind the scenes in assembly

All values stored in registers or memory (or as constant immediates)

Function arguments are no different – they must be stored in registers or memory

Arguments and return values

```
fun:
        addi $t0, $0, 1
        jr $ra

main:
        addi $s0, $0, 5
        sll $s1, $s0, 2
        jal fun
        add $s2, $s0, $s1  # jr returns here
```

Other than return-address register and PC, nothing changes during jump-and-link

In this example, whatever was in `$t0` before ”function call” (i.e., jump and link) will be there at start of function

Arguments and return values

MIPS has several registers `$a0` - `$a3` reserved for passing arguments

For example, if code is about to call function with one argument, it puts that argument into `$a0` before calling `jal`

When control reaches function (after `jal`), function can use value in `$a0`

This is *the same* `$a0` seen by caller – any changes to register within function affect `$a0` everywhere

Input Arguments & Return Value

C Code

```
int main()
{
  int y;
  ...
  y = diffofsums(2, 3, 4, 5);  // 4 arguments
  ...
}

int diffofsums(int f, int g, int h, int i)
{
  int result;
  result = (f + g) - (h + i);
  return result;                // return value
}
```

Input Arguments & Return Value

MIPS assembly code

```
# $s0 = y

main:
   ...
   addi $a0, $0, 2     # argument 0 = 2
   addi $a1, $0, 3     # argument 1 = 3
   addi $a2, $0, 4     # argument 2 = 4
   addi $a3, $0, 5     # argument 3 = 5
   jal  diffofsums     # call Function
   add  $s0, $v0, $0   # y = returned value
   ...

# $s0 = result
diffofsums:
   add $t0, $a0, $a1   # $t0 = f + g
   add $t1, $a2, $a3   # $t1 = h + i
   sub $s0, $t0, $t1   # result = (f + g) - (h + i)
   add $v0, $s0, $0    # put return value in $v0
   jr  $ra             # return to caller
```

Arguments and return values

Unlike `$ra`, which is linked by architecture to `jal` call, there is nothing special about `$a*` registers

It is a *convention* all MIPS programmers use that arguments will be placed in those registers so functions know where to look for them

Could write function that looked for arguments in `$t0` - `$t3`, but everyone who tried calling your functions would hate you (and you would not pass unit tests)

Arguments and return values

Return values work exactly the same way

By convention, everyone agrees to put return values from functions in `$v0` and `$v1`

After call to `jr $ra`, caller can look in `$v*` registers if it expects output from function

Another requirement

```
int foo(a, b) {
     int c = a + b;
     return c;
}

int c = 10;
int d = foo(2, 4);
// c is still 10
```

Function should not change values outside its scope

- In assembly terms, this means function must not overwrite memory or registers used by caller

Input Arguments & Return Value

MIPS assembly code

```
# $s0 = result
diffofsums:
  add $t0, $a0, $a1   # $t0 = f + g
  add $t1, $a2, $a3   # $t1 = h + i
  sub $s0, $t0, $t1   # result = (f + g) - (h + i)
  add $v0, $s0, $0    # put return value in $v0
  jr  $ra             # return to caller
```

- `diffofsums` overwrote 3 registers: `$t0`, `$t1`, `$s0`
- `diffofsums` can use *stack* to temporarily store registers