Functions (and procedures)

When we have a sequence of instructions that is used repeatedly, it is better to encapsulate these instructions with a name. In C, we call them *functions* and *procedures*. (A "procedure" is just a "function" that doesn't return a value.) In Java, we call them methods. In this lecture and the next, we'll look at the basics of how C-like functions are implemented in MIPS.[1]

The first idea for MIPS functions is that we label the first instruction with the name of the function. Here we are thinking about the instruction (text) segment of Memory. The more interesting of functions is how information is passed to them and returned from them, and how the data used by the function is represented. These issues involve the data segment of Memory.

Suppose you have a C function:

```
int myfunction(int j){
   int m;
      :          //  other declarations here, and many instructions
   return m;
}
```

Such a function would be called by another function, for example, `main`. We will refer to the function that makes the call as the *parent* and the function that gets called as the *child*. (Alternatively, the terms *caller* and *callee* are used, respectively.) In this example, `main` is the parent, and `myfunction` is the child.

```
main(){
   int i,j,m;
      :
   m = myfunction(i);
      :
   j = myfunction(5*i);
      :
}
```

How might this work in MIPS? `main` and `myfunction` are each a sequence of MIPS instructions which are in the text segment of Memory. Each of these MIPS instructions has an address. (Recall that the two least significant bits of an instruction address are always 00, i.e. instructions are word aligned.)

How does these functions share registers and Memory? And how does MIPS jump from the parent to child and then from child to parent? First, let's consider what `main` needs to do:

- it needs to branch to the first instruction of `myfunction`;
- it needs to store a "return address" so `myfunction` can later jump back to the location in `main` from where it was called;

[1] We won't look at Java methods because it would take us off track – I would first need to tell you how classes and their instances (objects) are represented. Interesting stuff, but its not within the scope of this course.

- it needs to store the arguments that it passes to `myfunction`, so that `myfunction` can access them;
- (optional) it needs to prepare some location for the result returned from `myfunction`

Next consider what `myfunction` needs to do:

- it needs to access its arguments
- it needs to allocate storage for its local variables
- if it computes a result, it needs to put this result in the place that `main` expects it to be
- it must not destroy any data (in Memory or in registers) that `main` was using at the time of the call, and that `main` will need afterwards.

The same rules should apply to any parent and child function. Let's deal with these issues one by one.

Jump to and return from a function

Suppose the first line of the MIPS code for `myfunction` is given a label `myfunction`. To branch to `myfunction`, you might expect `main` to use the MIPS instruction:

```
j     myfunction
```

This is not sufficient, however, since `myfunction` also needs to know where it is supposed to eventually return to. Here's how it is done. When `main` branches to the instruction labelled `myfunction`, it writes down the address where it is supposed to return to. This is done automatically by the branching function `jal`, which stands for "jump and link."

```
jal     myfunction
```

`jal` stores a return address in register `$31`, which is called `$ra`. *The return address is written automatically as part of* `jal` *instruction.* That is, `$ra` is not part of the syntax of the `jal` instruction. Rather, `jal` has the same syntax as the jump instruction `j` we saw earlier. It is a J format instruction. So you can jump to 2^{26} possible word addresses, namely the words in the current instruction segment of Memory. (Recall that there are two instruction segments, namely the user's and the kernel's – both are 2^{26} words.)

The return address is the address of the current instruction + 4, where the current instruction is "`jal myfunction`". That is, it is the address of the instruction following the call. (4 bytes = 1 word)

When `myfunction` is finished executing, it must branch back to the caller `main`. The address to which it jumps is in `$ra`. It uses the instruction

```
jr      $ra
```

where `jr` stands for `jump register`. This is different from the jump instruction `j` that we saw earlier. `jr` has R-format.

So the basic idea for jumping to and returning from a function is this:

```
myfunction:     ⋮
                ⋮
                jr      $ra
                ⋮


main:           ⋮
                jal     myfunction
                ⋮
```

Passing arguments to and returning values from functions

The next issue is how arguments are passed to a function, and how a value computed by the function is returned. When writing MIPS code, one uses specific registers to pass arguments to a function. Of the 32 registers we discussed previously, four are dedicated to hold arguments to be passed to a functions. These are `$4,$5,$6,$7,` and they are given the names `$a0, $a1, $a2, $a3` respectively. There are also two registers that are used to return the values computed by a function. These are `$2,$3` and they have names `$v0,$v1`, respectively.
[ASIDE: recall from last lecture that `syscall` used `$v0` to specify the code for the system call e.g. print an integer. I find this strange, since the code for a system call seems like an argument, not a returned value. If anyone can suggest why `$v0` is used for this code instead of say `$a0`, let me know.]

For example, if a function has three arguments, then the arguments should be put in `$a0,$a1,$a2`. If it returns just one value, this value should be put in `$v0`. (It is possible to pass more than four arguments and to return more than two values, but this requires using registers other than $\$2 - \$7$.)

What are the MIPS instructions in `main` that would correspond to the C statement

$$\texttt{k} = \texttt{myfunction}(\texttt{i});$$

The argument of `myfunction` is the variable `i` and let's suppose `i` is in `$s0`. Suppose the returned value will be put into `$s1`, that is, the C variable `k` will be represented in the MIPS code by `$s1`. Here are the instructions in `main`:

```
move $a0, $s0      #  copy i into argument register
jal  myfunction
move  $s1, $v0     #  copy result to variable k
```

MIPS register conventions

One situation hat we need to avoid is that the child function writes over data (in registers or Memory) of the parent function. The parent will need this data after the program returns to it. Just as people have conventions that allow us to avoid bumping into each other like walking on the

right side of a hallway or stairwell, MIPS designers invented conventions that MIPS programmers should follow to avoid erasing data.

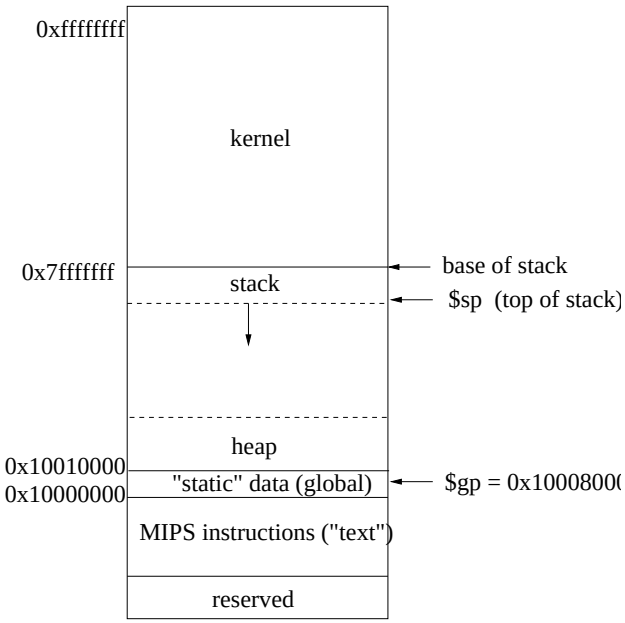
Here are the rules for the `$s0,...,$s7` registers and `$t0,...,$t7` registers:

- The parent assumes that temporary registers `$t0,...,$t7` may be written over by the child. If the parent needs to keep any values that are in `$t0,...,$t7` prior to the call, then it should store these values into Memory prior to the call.
- The parent assumes that the values in the save registers `$s0,..,$s7` will still be there after the return from the call. The parent does not need to store the values in these registers into Memory. Thus, if a child wishes to use any of the `$s0,..,$s7` registers, it must store the previous values into Memory prior to using them, and then load these values back into the register(s) prior to returning to the parent.

Let's look at how this is done.

The Stack

Register values that are stored away by either the parent or child function are put on the *stack*, which is a special region of MIPS Memory. The stack grows downwards rather than upwards (which is strange since we sometimes we will use the phrase "top of the stack"). We refer to the *base* of the stack as the address of the top of the stack when the stack is empty. The base of the stack is at a (fixed) MIPS address `0x7fffffff` which is the halfway point in MIPS Memory, and the last location in the user part of Memory. (Recall that the kernel begins at `0x80000000`.)



The address of the "top" of the stack is stored in register `$29` which is named `$sp`, for *stack pointer*. This address is a byte address, namely the smallest address of a byte of data on the stack. To push a word onto the stack, we decrease `$sp` by 4 and use `sw`. To pop a word from the stack, we use `lw` and increase the `$sp` by 4.

How the caller/parent uses the stack

Suppose that, when the caller (e.g. `main`) calls a function (e.g. `myfunction`), there may be values some of the temporary registers (say `$t3,$t5`) that the caller will need after `myfunction` returns. According to MIPS conventions, the caller should not expect the values in these `$t` registers to still be present after the function call. Thus, caller should store away the values in these temporary registers before the call, and should load them again after the call. The stack is used to hold these temporary values:

```
parent:          :
             addi  $sp, $sp,  -8
             sw    $t2, 4($sp)
             sw    $t5, 0($sp)
             move  $a0, $s0          #  pass argument too
             jal   child
             lw    $t2, 4($sp
             lw    $t5, 0($sp)
             addi  $sp, $sp,  8
                   :
```

An equivalent way to do it is as follows:

```
    parent:        :
             sw    $t2, -4($sp)
             sw    $t5, -8($sp)
             addi  $sp, $sp,  -8
             move  $a0, $s0         #  pass argument too
             jal   child
             addi  $sp, $sp,  8
             lw    $t2, -4($sp)
             lw    $t5, -8($sp)
                  :
```

How the callee/child function uses the stack

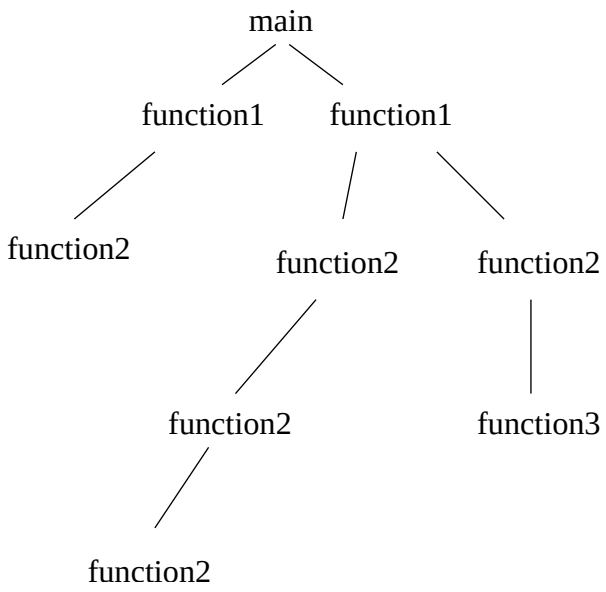
Suppose the child (e.g. `myfunction`) uses registers `$s0,$s1,$s2`. According the MIPS conventions, it must first store the contents of the registers onto the stack and then later replace them.

```
    child:       addi  $sp, $sp,  -12
                 sw    $s0, 8($sp)
                 sw    $s1, 4($sp)
                 sw    $s2, 0($sp)
                       :              #  DO WORK OF FUNCTION HERE
                       :
                 lw    $s0, 8($sp)    # restore the registers from the stack
                 lw    $s2, 4($sp)
```

```
lw    $s3, 0($sp)
addi  $sp, $sp, 12
move  $v0, $s2       # write the value to return
jr    $ra            # return to caller
```

A child can be a parent

Up to now we have only considered the case that the parent function calls a child, which returns directly to the parent. It can also happen that the child function is itself a parent. It might call another function, or it might call itself. For example, consider the call tree shown below. We interpret the call tree in the usual way, namely pre-order traversal (as learned in COMP 250). First, `main` calls `function1` which calls `function2`. `function2` returns to `function1` which then returns to `main`. Then, `main` calls `function1` again, which calls `function2`, but this time `function2` calls itself, recursively, twice. `function2` then returns to itself, twice, and then returns to `function1`. `function1` then calls `function2` which calls `function 3`. `function3` returns to `function2` which returns to `function1` which returns to `main`.

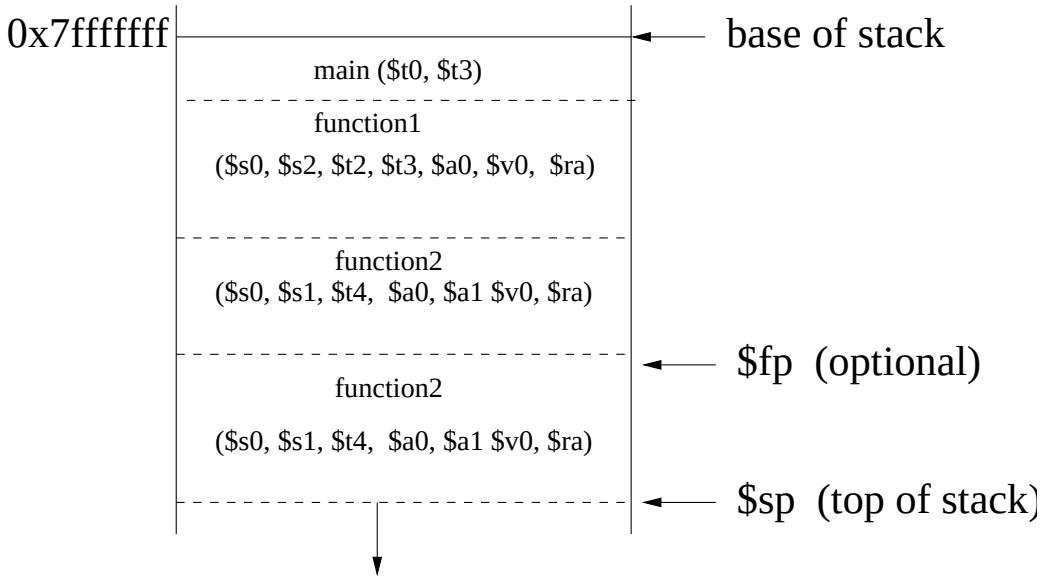


In this example, `main`, `function1`, and `function2` are all functions that call other functions. `function3` however, does not call any other other functions. When a function calls itself, or when it calls another function, it needs to store any temporary registers (`$t`), as discussed above. But it also needs to store:

- `$ra` : *e.g.* When `function1` calls `function2`, the instruction "`jal function2`" automatically writes into `$ra`. `function1` needs to store the value of `$ra` before it makes this call, otherwise it will will not be able to return to `main`.)
- any of `$a0,$a1,$a2,$a3` that are needed by the caller: e.g. when `function1` was called by `main`, it was passed certain arguments. These arguments will typically be different from the arguments that `function1` passes to `function2`

- `$v0,$v1`: if `function1` has assigned values to these registers prior to calling `function2`, then it needs to store these assigned values.

If a function (say `function3`) does not call any other functions (including itself, recursively) then we say that such a function is a *leaf* function, in that it is a leaf in the call tree. Leaf functions are relatively simple because there is no need to store `$ra` or argument registers `$a0`, .., `$a3` or value registers `$v0`, `$v1`.



Stack frames

Each function uses a contiguous set of words on the stack to store register values that are needed by its call tree parent or that it will need once its call tree child has returned. This set of contiguous words in memory is called a *stack frame*. Each function that gets called has its own stack frame.

Stack frames can hold other values as well. If a function declares local variables that are too big to be kept in registers (for example, many local, or an array), then these variables will also be located in the stack frame.

[ASIDE: Sometimes you don't know in advance how big the stack frame will be, because it may depend on data that is defined only at runtime. For example, the function might make a system call and might (based on some condition that may or not be met) read data from the console, and that data could be put on the stack. In these cases, you need to move the stack pointer as the stack frame changes size. It may be useful to mark the beginning of the stack frame. Such a marker is called the *frame pointer*. There is a special register `$fp = $30` for pointing to the beginning of the stack frame on the top of the stack.]

Examples

In the slides, I went over some examples with MIPS code. Here is a link to some MIPS programs:

- `http://www.cim.mcgill.ca/~langer/273/sumton.asm`

- `http://www.cim.mcgill.ca/~langer/273/sumtonNonRecursive.asm`
- `http://www.cim.mcgill.ca/~langer/273/factorial.asm`

Integer multiplication in MIPS

In lecture 7, I sketched out a circuit for multiplying two unsigned integers. In MIPS, there is a multiplication instruction for signed integers, `mult`, and for unsigned integers `multu`. Since multiplication takes two 32 bit numbers and returns a 64 bit number, special treatment must be given to the result. The 64 bit product is located in a "product" register. You have to access this register using two separate instructions.

```
mult      $s0, $s1    # Multiply the numbers stored in these registers.  This
                      # yields a 64 bit number, which is stored in two
                      # 32 bits parts:  "hi" and "lo"
mfhi      $t0         # loads the upper 32 bits from the product register
mflo      $t1         # loads the lower 32 bits from the product register
```

You can only read from the product register. You cannot manipulate it directly. In SPIM, the product register is shown as two 32 bit registers, HI and LO.

For example, see the program `factorial.asm` (link above). It computes the factorial of numbers up to n=12. After that, you need more than 32 bits to represent the result. I wrote the code so that it returns 0 in the case that you need more than 32 bits, i.e. if the HI register is non-zero.

Integer division in MIPS

What about division? To understand division, we need to recall some terminology. If we divide one positive integer by another, say 78/21, or more generally "dividend/divisor" then we get a quotient and a remainder, i.e.

$$\text{dividend} \;=\; \text{quotient} \;*\; \text{divisor} \;+\; \text{remainder}$$

$$e.g. \quad 78 = 3 * 21 + 15$$

In MIPS, the divide instruction also uses the HI and LO registers, as follows:

```
div       $s2, $s3      # Hi contains remainder, Lo contains the quotient
mfhi      $t0           # $t0 gets the $remainder
mflo      $t1           # $t1 gets the quotient
```

Note that `mult, div, mfhi, mflo` are all R format instructions.