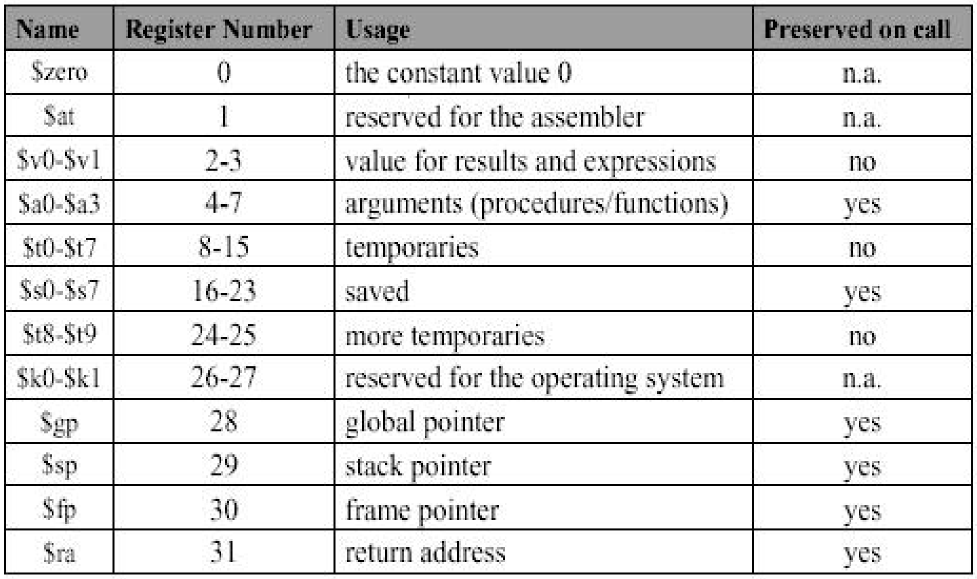
We have seen many of the 32 MIPS registers so far. Here is a complete table of them. In this lecture, we will meet the last three of them: the stack pointer, the frame pointer (which we won't use, but it is good to be aware of it), and the return address. We will also discuss the rightmost column in this table and what is meant by "preserved on call".



Name	Register Number	Usage	Preserved on call
$zero	0	the constant value 0	n.a.
$at	1	reserved for the assembler	n.a.
$v0-$v1	2-3	value for results and expressions	no
$a0-$a3	4-7	arguments (procedures/functions)	yes
$t0-$t7	8-15	temporaries	no
$s0-$s7	16-23	saved	yes
$t8-$t9	24-25	more temporaries	no
$k0-$k1	26-27	reserved for the operating system	n.a.
$gp	28	global pointer	yes
$sp	29	stack pointer	yes
$fp	30	frame pointer	yes
$ra	31	return address	yes

Functions (and procedures)

As you have learned in your first programming course, when a sequence of instructions is used many times, it is better to encapsulate these instructions: give them a name and hide the inner details. In Java, we call these encapsulated instructions *methods*. In C, we call them *functions* and *procedures*. A "procedure" is just a "function" that doesn't return a value. In this lecture, we'll look at the basics of how C-like functions are implemented in MIPS. 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. If time permits, I may say something about this at the end of the course.

When we write a MIPS function, we label the first instruction with the name that we give the function. Later we can branch to the function by using this label name. The label just indicates the address to branch to. Instructions that define a function belong to the instruction (text) segment of Memory, namely below 0x10000000 – *see slides at end of lecture 9.* We'll see an example on p. 3 of this lecture.

To pass arguments to functions and return values from functions, we use the data segment of Memory, in particular, a region called the *stack*. You learned about stacks as an abstract data type in COMP 250. In MIPS, the stack has a specific technical meaning which I will tell you about today.

Suppose you have a C function:

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

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*. (*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 just labels for an instruction, namely the first instruction in some sequence of instructions which are in the text segment of Memory.

There are a few important questions here: How does MIPS jump from the parent to child and then later return from child to parent? How do these functions share registers and Memory? First, let's consider what `main` needs to do or consider when it calls `my function`:

- it will need to save in Memory any values currently in registers that `myfunction` might destroy
- it needs to provide a "return address" so `myfunction` can later jump back to the location in `main` from where it was called;
- it needs to provide arguments, so that `myfunction` can access them;
- it will needs to access the result which will be returned from `myfunction`
- it needs to branch to the first instruction of `myfunction`;

Next consider what `myfunction` needs to do:

- it needs to access its arguments
- it needs to allocate storage for its local variables
- (optional) if it computes a result, then 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 its parent (e.g. `main`) was using at the time of the call and that its parent will need after the call
- it needs to return to the parent.

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 labelled `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 a different branching function `jal`, which stands for "jump and link."

```
jal     myfunction
```

`jal` stores a return address in register `$31`, which is also called `$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. Later when we look at the data path, we will see exactly how this works and it will make more sense!

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. (Recall each instruction is 4 bytes or 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 to address 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.

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 $\$2 - \$7$.)

[ASIDE: recall from last lecture that `syscall` uses `$v0` to specify the code for the system call e.g. print an integer, and `$a0` and `a1` for various arguments needed for the system call. If a function uses syscalls, then one must be especially careful to save these register values before the syscall and then load these register values after the syscall.]

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

`m = myfunction(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 `m` 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 m
```

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.

  In addition, the parent assumes that the `v0, v1` variables can be overwritten, so if the parent wants to keep values in those registers then the parent needs to store them in Memory prior to the call and (re)load them after the call.

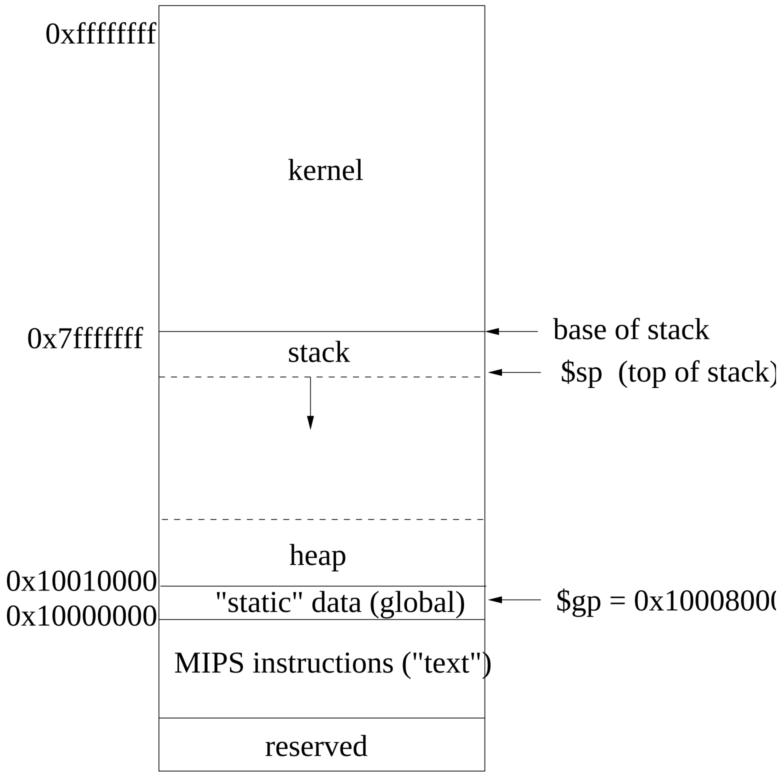
- The parent assumes that the values in the save registers `$s0,..,$s7` will still be there after the return from the call. The parent does not store the values in these registers into Memory before making the call. Thus, if a child wishes to use any of the `$s0,..,$s7` registers, it must store the previous values of those registers into Memory prior to using these values, and then load these values back into the register(s) prior to returning to the parent.

In addition, the parent assumes that the `a0, ···, a3` variables will be preserved after the function call. So if the child wants to use these registers, then it should first store them in Memory and later it should load them back 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. I emphasize that an address always means a byte address, and so when we use an address in a load/store word instruction, we mean the word that starts at a particular byte and occupies that byte and the next three bytes after it, which have larger byte addresses.

ASIDE: In the figure on the previous page, note that the stack grows downward toward another area of user Memory called the "heap" (which is not to be confused with the heap data type that you learned about in COMP 250). The heap is used to allocate memory for other variables than what we are discussing here: for example, last lecture we discussed assembler directives and how you could define strings and other data. These go on the heap. In the C programming language, when you "malloc" and "free", you use the heap. In Java, when you construct "new" objects, you use the heap. Unlike the stack, the heap is not necesarily a continuous regions of memory e.g. when you free space (C) or garbage collect an object that is no longer reference (Java) you leave a hole in the heap. The kernel will try to fill that hole later, rather than just moving the heap upwards to towards the stack, since if you just keep moving the heap boundary up then eventually the heap and stack will collide and the program will run out of space.

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 in 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:         :
          move  $a0, $s0             #  pass argument
          addi  $sp, $sp,  -8
          sw    $t2, 4($sp)          # store the temporary registers
          sw    $t5, 0($sp)

        #  sw    $t2, -4($sp)        # also correct
        #  sw    $t5, -8($sp)
        #  addi  $sp, $sp,  -8

          jal   child
          lw    $t2, 4($sp)          # (re)load tempory registers
          lw    $t5, 0($sp)
          addi  $sp, $sp,  8
                :
```

How the callee/child function uses the stack

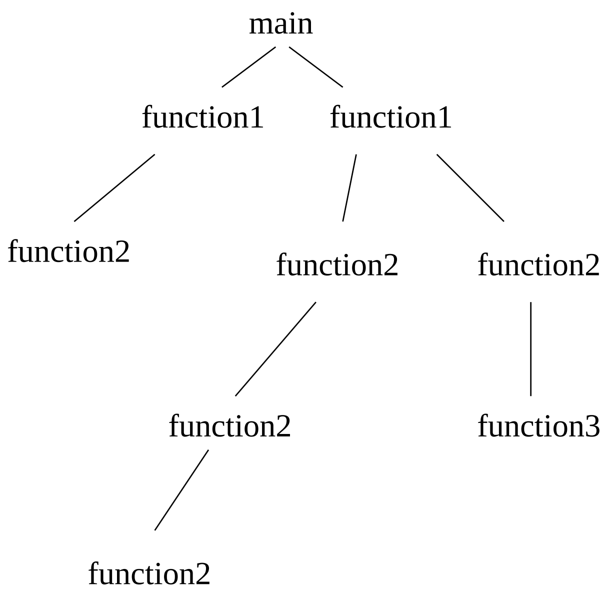
Suppose the child (e.g. `myfunction`) uses registers `$s0,$s1,$s2`. According the MIPS conventions, it must store the current contents of the registers onto the stack. Then, when it is finished it must load these previous values back in.

```
child:      sw    $s0, -4($sp)
            sw    $s1, -8($sp)
            sw    $s2, -12($sp)
            addi  $sp, $sp,  -12
                  :              #  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. (For those of you who have taken COMP 250, you know that the sequence of calls in a running program defines a tree, and the nodes are traversed in "pre-order".) In the example below, first, `main` calls `function1` which calls `function2`. Then `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`. Then `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 on the stack:

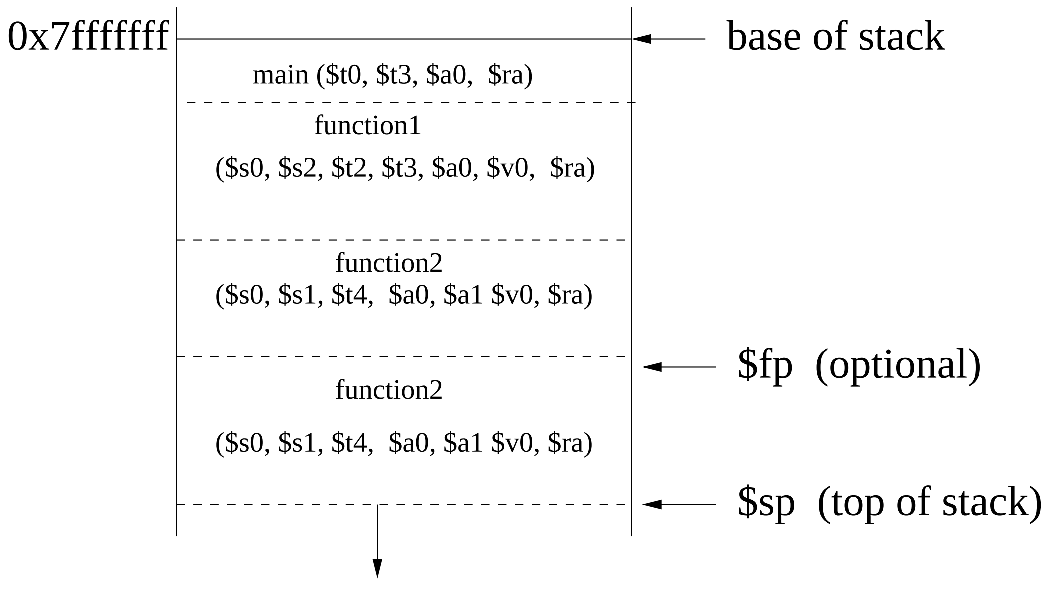
- `$ra` : *e.g.* When `function1` calls `function2`, the instruction "`jal function2`" automatically writes into `$ra`. Thus, `function1` needs to store the value of `$ra` before it makes this call, and then reloading it after the call. Otherwise it will not be able to return to `main`.)

- 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 would need to store these assigned values. (For example, if `function1` uses the `$v0` register as part of a `syscall`).

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` (unless they are used by `syscall` as part of the function).

Stack frames

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


0x7ffffff
base of stack
main ($t0, $t3, $a0, $ra)
function1
($s0, $s2, $t2, $t3, $a0, $v0, $ra)
function2
($s0, $s1, $t4, $a0, $a1 $v0, $ra)
$fp (optional)
function2
($s0, $s1, $t4, $a0, $a1 $v0, $ra)
$sp (top of stack)

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.

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. In these cases, 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. I won't be using it. I am mentioning it for completeness only.

Finally, note that the first stack frame is from `main` and that it has a return address. The reason is that when the program is over, the function will return to the kernel.

Example: sum to n

Here is a simple example of recursion. It is similar to the "factorial" function except that here we are computing the sum of the values from 1 to n rather than the product of values from 1 to n. Of course, you would not use recursion to compute the sum of the first n numbers from 1 to n since in fact this sum is n(n+1)/2. I use this example only to illustrate how recursion works.

Here is the code in a high level language, following by MIPS code.

```
#   int  sumton( int n) {
#         if (n == 0)
#                return 0
#         else
#               return n + sumton( n-1 )
#    }

sumton:
    beq    $a0, $zero, basecase    #  if n == 0 then branch to base case
    addi   $sp,$sp,-8          # else , make space for 2 items on the stack
    sw     $ra,  4($sp)        # store return address on stack
    sw     $a0, 0($sp)         # store argument n on stack
                               # (will need it to calculate returned value)
    addi   $a0, $a0, -1        # compute argument for next call:  n = n-1
    jal    sumton              # jump and link to sumton  (recursive)

    lw     $ra,  4($sp)        # load the return address
    lw     $a0,  0($sp)        # load n from the stack
    addi   $sp,  $sp,8         # change the stack pointer

    # register $v0 contains result of sumton(n-1)
    add    $v0, $a0, $v0       # add n  to $v0
    jr     $ra                 # return to parent

basecase:
    addi   $v0, $zero, 0       # assign 0 as the value in $v0
    jr     $ra                 # return to parent
```