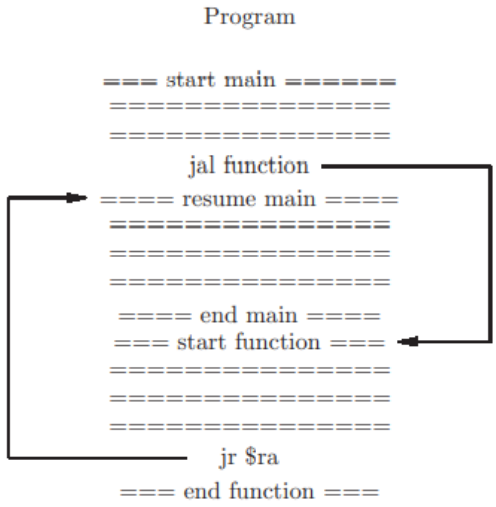
6 | Functions, System Call, Strings

A **function** is a subprogram with a designated purpose that may be invoked, or called, from a main program or from another subprogram. Examples include Java methods and Python functions.



For assembly language function calls,

- There is an unconditional jump to the function
- When the called function terminates, needs to return to correct place in calling function, aka the **return address** (use $ra register)
- A diagram of a function call and return is shown right

Jump and link is used to call or invoke a function.

- Format: **jal** **label**
- The label should label the first instruction in the function that you're calling

- Meaning:
  - $ra ← address of the next instruction after the jal
  - [unconditional] jump to label

- Example: jal mySub
  there: ## return here when mySub is finished
  ## $ra contains address of there

Jump register is used to return to the calling function when a function is ready to terminate.

- Format: **jr** **$ra** # Can give any register you want, but want to use $ra
- Meaning: [unconditional] jump to the instruction whose address is in $ra

Example: Arrange two contiguous words of memory in ascending order (if the first word is smaller than the second, leave them as is). Register $a0 contains the address of the first word. The name of the function is order2.

```
        .text
################## Begin function order2
```

```
### order2: Arrange two words of memory ascending
### Pre: $a0 contains the address of the first word (main job)
### Author: sdb
order2:
        lw      $t0, 0($a0)             # get first word w/ explicit address
        lw      $t1, 4($a0)             # get second word similarly
        ble     $t0, $t1, done          # already in order? $t0 <= $t1? branch to done
        sw      $t0, 4($a0)             # (if above false) store first word 4($a0) = $t0
        sw      $t1, 0($a0)             # replacing the address of the first word
done:
        jr      $ra                     # return to calling function
###################### End function
```

Note that the comments (hashtags) are part of an API to tell you what the function does.

How can this function be called (in the example)?

```
                .text
la      $a0, twoWords               # load addresses of twoWords into $a0
jal     order2                      # calls the function order2

                .data
twoWords:       .word       3, -5   # where the two words are
```

Example: Function, named order3, to arrange three contiguous words of memory in ascending order. (This should be easy by using order2.)

Here's the algorithm (*try it out yourself with words 12, 5, 2*):

1. Arrange the first and second words, using order2
2. Arrange the second and third words, using order2
3. Arrange the first and second words again, using order 2 (since new neighbors)

```
################## Begin function order3
### order3: Arrange three words of memory ascending
### Pre: $a0 contains the address of the first word (main job)
### Author: sdb

                .text
order3:
```

```
        jal     order2          # arrange first and second
        addi    $a0, $a0, 4     # increment $a0 by 4 to get the second word's address
        jal     order2          # arrange second and third
        addi    $a0, $a0, -4    # decrement $a0 by 4 to get first word's address
        jal     order2          # arrange first and second again
        jr      $ra             # return to the calling function
###################### End function
```

However, there is a large problem with this. order2 overwrites the return address ($ra). In order2, the jal resets the return address, and it's overwritten there (notice order3 calls order2 a bunch of times).

- order3 should save the return address somewhere so that it can be reloaded after the last call to order2
    - Before the first call to order2: `sw $ra, return`
    - Return is a label (put `return: .word 0` under a .data within the order3 method - a function can have local .data)
    - Add instruction: `lw $ra, return` to load return address before jr

Function **parameters** are known as arguments. In assembly language, the $a# registers will be used, so we'll have up to 4.

API (Application Program Interface) is a comment describing everything you need to know in order to use the program.

- For a function, include:
    - Function name
    - Author, date
    - Overall purpose
    - Preconditions (what's expected to be true before)
    - Registers containing parameter values
    - Side effects (things that have an effect on the state of the program)
    - Post conditions (what's expected to be true after)
    - Registers containing explicit return values

Register conventions are made when people working in a team agree on register usage. Standard conventions (here) should be observed even if you're not on a team. See below.

Registers	Purpose	Responsible for saving
$a0…$a3	function parameters	[see API]
$v0…$v3	function result	[see API]
$s0…$s7	temporary results	called function
$t0…$t9	temporary results	calling function
$ra	return address	called function
$sp	stack pointer	
$at	assembler temporary	[do not use]

> "Responsible for saving" means if a function overwrites a register, and a calling function needs the value that was in that register, who's responsible for saving it? Saving the value should also mean reloading it later.

Therefore, every function which calls other functions should save its return address.

How can recursive functions maintain return addresses? The solution is to use a **memory stack (call stack)** for return addresses using registers $s0…$s7 as well as the $t registers.

- A memory array with certain properties
- Last in, first out data structure (LIFO)
- Push operation: *Place value on top of the stack, decrement $sp*
- Pop operation: *Remove top value from the stack, increment $sp*

How is the call stack implemented in the MIPS memory? The separate section of memory.

- $sp initially contains the address at the high end (stack point)
- The stack grows toward a lower address
- Stack is used to store other registers, in addition to $ra
    - If $s0…$s7 is overwritten in a function, it should be saved on the stack and reloaded when the function terminates
    - $t0…$t9 if need to be preserved across function calls, should be saved on the stack before calling a function

Example of function entry/exit:

```
############## Begin function
## API: author, name, purpose, pre and post conditions

name:
        addi    $sp, $sp, -12           # pushing 3 registers onto the stack
        sw      $ra, 0($sp)             # return address
        sw      $s2, 4($sp)             # push $s registers that will be overwritten
        sw      $s7, 8($sp)

** Some code that overwrites registers $s2 and $s7

        lw      $s7, 8($sp)             # pop saved registers
        lw      $s2, 4($sp)
        lw      $ra, 0($sp)             # reload return address
        addi    $sp, $sp, 12            # original stack pointer (increment it)
        jr      $ra                     # return to calling function (termination)
```

How about label conflicts? The assembler won't permit duplicate labels in a source file. So, we should append the function name to each label (like lp_function1, or done_function1).

Let's redo function order2:

```
################## Begin function order2
        ### order2: Arrange two words of memory ascending
        ### Pre: $a0 contains the address of the first word (main job)
        ### Author: sdb
        order2:
                addi    $sp, $sp, =12           # pushing 3 words on the stack
                sw      $ra, 0($sp)             # save return address
                sw      $s0, 4($sp)             # save $s0 on stack
                sw      $s1, 8($sp)             # save $s1 on stack

                lw      $s0, 0($a0)             # get first word w/ explicit address
                lw      $s1, 4($a0)             # get second word similarly
                ble     $s0, $s1, done          # already in order? $s0 <= $s1? branch to done
                sw      $s0, 4($a0)             # (if above false) store first word 4($a0) = $s0
                sw      $s1, 0($a0)             # replacing the address of the first word (store 2nd)
        done:
                lw      $ra, 0($sp)             # load saved registers
```

```
        lw      $s0, 4($sp)             # these are the pops off the stack
        lw      $s1, 8($sp)
        addi    $sp, $sp, 12            # adjust stack pointer back to where it was
        jr      $ra                     # return to calling function
###################### End function
```

System Call

Normal program termination to tell it to stop.

```
        li      $v0, 10
        syscall
```

It will look at the value at register $v0 to determine what is to be done. If we put the value 10 there, it means to terminate the program normally.

Strings

How are characters represented in memory? *There's an integer code for each character, named* **ASCII** *(8-bit code, one character per byte), from the 1960s. Today, there is* **Unicode** *(16-bit code) for more complex foreign alphabets and other special symbols.*

- ASCII is a subcode of Unicode
- 'A' = 65 = 41_x = 0100 0001
- 'B' = 66 = 42_x = 0100 0010
- '$' = 36 = 24_x = 0010 0100
- space = 32 = 20_x = 0010 0000
- 'Z' = 90 = $5a_x$ = 0101 1010
- 'a' = 97 = 61_x = 0110 0001
- 'z' = 122 = $7a_x$ = 0111 1010
- line feed = 10 = $0a_x$ = 0000 1010
- carriage return = 13 = $0d_x$ = 0000 1101

What is the difference between uppercase/lowercase? *The only one is that at* bit position 5 *(count from right to left starting from 0). To change to an uppercase letter, clear bit position 5; to do so for a lowercase letter, set bit position 5.*

There are other characters that some want.

line feed = moving one line forward ("\n")
carriage return = moving cursor to the beginning of the line ("\r")

A **string** is a sequence of ASCII character codes in contiguous memory locations. There will be one character in each byte (4 characters in a word). How do we know when we get to the end of the string? There will be a null byte: 0.

In assembly language, there are two ways to define one:

- Use .asciiz to terminate with a null byte
- Use .ascii to use no termination byte

```
        .data
name1:  .ascii   "harry"
name2:  .asciiz  "harry"
name3:  .ascii   "billy"
```

You can view, in the "debug", the "data segment" box on how the strings are stored and check the "ASCII" box to view it without the hexadecimal values. The strings are stored backwards, strangely enough. The null byte will show as \0 in this

To locate things in memory (and see where the strings stop at), you can put words -1 in between since they're easy to locate (shown as all 'f's).

When you declare something to be a full word, it has to begin at the next full word in memory. It cannot begin right after the 'y' in "harry", for example.

MARS shows -1's as dots in the ASCII. (Play with these strings!)

How to get characters from a string into a register: lbu (meaning load byte unsigned).

- Format:
```
lbu   $rt, label        # symbolic memory address
lbu   $rt, imm($rs)     # explicit memory address
```

  A register containing an address, and then an immediate field containing a displacement, and the sum of the displacement and register contents would form the explicit memory address.

- Meaning:
  rt_{0\text{-}7}$ ← single byte from memory (at that address) and put that into that register in the lower order bits

  rt_{8\text{-}31}$ ← set the rest of them to 0's (clear them)

How to get characters from a register into memory: sb (meaning store byte).

- Format:
```
sb   $rt, label        # symbolic memory address
sb   $rt, imm($rs)     # explicit memory address
```

Doesn't have to be a full word boundary, can be any byte in memory.

- Meaning: memory byte ← $rt # doesn't matter what's in high order 24 bits

String functions operate on strings.

- strlen Finds length of a null-terminated string.
- strcmp Compares 2 strings for which is smaller (who comes first alphabetically)
- strcpy Copies a string from one memory location to another
- toupper capitalizes all lowercase letters
- tolower make all letters lowercase

strlen:

Precondition: Address of string is in $a0
Postcondition: Register $v0 contains the string's length

This can be found in strlen (lecture 3.11).asm! Step through it on your own and put breakpoints when necessary (expand the text segment window to see).