Notes for 8/28/09

A system call transfers control to the kernel. We make a system call to print "Hello World" in Lab 1. This is because a userspace process cannot print on its own, since userspace processes can only compute and make system calls. In C, we make function calls to functions like puts() and printf() that make these system calls for us.

Registers in our MIPS architecture are all 32 bits, our architecture's word size. They are not typed, so you can put any 32 bit value that you want in them, e.g. an integer, a memory pointer, etc. This data only has meaning insofar as it is subject to later interpretation. If you want, you can interpret a pointer as an integer, or vice versa, but this is generally a bad idea...

Register	Common Name	Description
$0	zero	Always zero
$1	at	Reserved for use by assembler, such as when expanding pseudo-instructions
$2 - $3	v0 - v1	Return values of function calls and system calls (v1 is used when return value is 64 bit)
$4 - $7	a0 - a3	Arguments for functions and system calls
$8 - $15	t0 - t7	Temporary. These are not preserved across function calls, so you must store these to memory before calling a function and load them back from memory before you want to use them again
$16 - $23	s0 - s7	Saved temporary. These are preserved across function calls, so, if you call a function, values in these registers will be preserved. The catch is that if you are writing a function, then you must preserve these registers for whoever called you. This means storing them to memory before you use them and loading them from memory before your function returns.
$24 - $25	t8 - t9	Same as t0 - t7. Note that, on Linux, t9 is used to store the address of the function that presently has control. This is used by a function to calculate gp (below).
$26 - $27	k0 - k1	Reserved for kernel. Userspace processes should not touch these.
$28	gp	Global pointer. On Linux, points to global offset table.
$29	sp	Points to the end of the stack.
$30	fp	Points to the beginning of the current stack frame.
$31	ra	Return address. This is set automatically by the *jal* instruction, which can be used to call functions. Thus, to return from a function, jump to the address in this register, i.e. *jr ra*.

li vs. *la* vs. *lw*...

li is "load immediate": the "immediate" just means that the value is encoded in the instruction itself as opposed to using a value in a register. *li* is actually a pseudo-instruction. This is because all of our instructions are 32 bits, which, after the space of other things encoded in the instruction are taken out,

leaves only 16 bits for our immediate value to be encoded in an instruction. If our immediate value is greater than 16 bits, this will expand into two instructions to load all 32 bits into the register. In C terms, a load immediate is like storing an integer literal into a variable.

lw is "load word": this loads the 32-bit word at the memory address stored in the second given register into the first. In C terms, this is like deferencing a pointer and storing the result into a variable.

la is "load address": this loads the address of a label into the given register. This is actually a pseudo-instruction so that the address can be offseted by the $gp (global pointer) register. Otherwise, if we just hardcoded addresses into our program, our program could not be placed anywhere in memory. In C terms, this is like copying a pointer and storing that value into a variable.

Adding and subtracting...

add $1, $2, $3 => $1 = $2 + $3
sub $1, $2, $3 => $1 = $2 - $3
addi $1, $2, 789 => $1 = $2 + 789