Systems and machine code

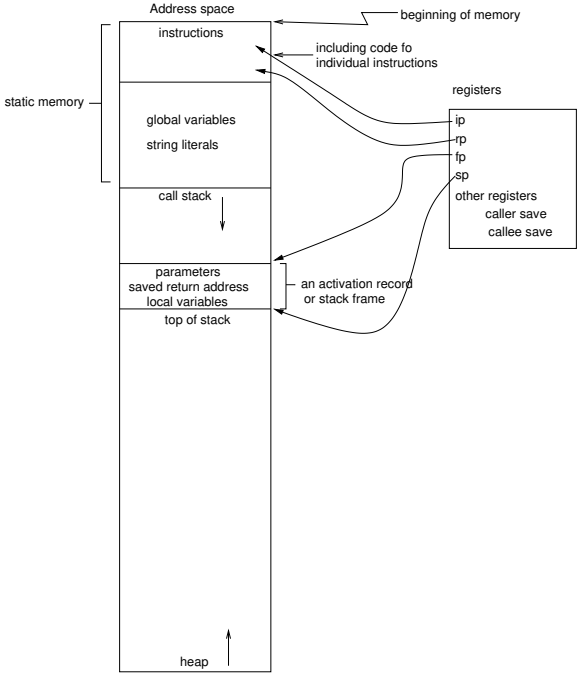
- Processors and architecture (last week Monday)
- Assembly (last week Wednesday)
- Function call and return (last week Friday and **today**)
- Function pointers (Wednesday)
- Review for final exam (Friday)

Today:

- Recap of assembly instructions so far
- Building functions and calls
- How function call and return actually work
- Assembly example with function call and return

Final exam: Thurs, May 7, 10:30 AM

	Mnemonic	Operands	Description
1	`MOVI`	`imm rd`	Copies the immediate value `imm` into register #`rd`.
2	`MOV`	`rs rd`	Copies the value in register #`rs` into register #`rd`.
3	`ADD`	`rs rd`	Adds the value in register #`rs` into register #`rd`.
4	`SUB`	`rs rd`	Subtracts the value in register #`rs` from register #`rd`.
5	`MUL`	`rs rd`	Multiplies register #`rd` by the value in register #`rs`.
6	`IDIV`	`rs rd`	Divides register #`rd` by the value in register #`rs` (as `int`s).
7	`JMP`	`ra`	"Jump." The next instruction to be executed is found at the address in register #`ra`.
8	`JNZ`	`rs ra`	If the value of register #`rs` is nonzero, jumps to the address in register #`ra`. Otherwise, does nothing and continues normally to the next instruction.
9	`OUT`	`rs`	Outputs the value in register #`rs` by printing it to the screen.
10	`HALT`		Ends the program.
11	`LD`	`ra rd`	Loads a value from memory, at the address in register #`ra`, into register #`rd`.
12	`ST`	`ra rs`	Stores the value of register #`rs` in memory, at the address in register #`ra`.


Address space
beginning of memory
instructions
including code fo
individual instructions
static memory
registers
global variables
string literals
ip
rp
fp
sp
other registers
caller save
callee save
call stack
parameters
saved return address
local variables
an activation record
or stack frame
top of stack
heap

Special registers:

0	`ip`	Instruction pointer	30	`fp`	Frame pointer
29	`rp`	Return pointer	31	`sp`	Stack pointer

When a program begins:

- The OS allocates memory for the process, called its *address space*.
- The program's instructions, as well as space for global variables and string literals, are put in the *static memory* portion of the address space.
- The `ip` is set to the beginning of the main function.

When memory is dynamically allocated:

- The code for the malloc function finds an unused portion of memory from the *heap*, the end of the address space, and marks it as in use.
- `malloc` returns a pointer to the beginning of the allocation portion of memory.

While a function is being executed

- `ip` refers to the address of the current instruction in the function.
- Some registers are used to store frequently used local variables, others for intermediate/temporary results.
- A *stack frame* or *activation record* in the address space stores information about that activation (call) of the function—for example, parameters and other local variables.
- Local variables are offsets from the beginning of the stack frame.
- `fp` points to the beginning of the stack frame.
- `sp` points to the end the stack frame or top of the stack.

When a function is (about to be) called

- "Caller-save" registers and parameters are stored in (pushed to) the stack; this entails writing to the place `sp` points to and incrementing `sp`.
- The location of the next instruction in the current function is stored in `rp`.
- `ip` is set to the first instruction in the function being called.

When a function begins

- `rp` and `fp` are stored in the stack, and `sp` is set to old `fp`.
- "Callee-save" registers are stored in the stack.
- Parameters are read from the stack.

When a function returns (or, is about to return)

- "Callee-save" registers are restored (popped from the stack)
- The return value is stored in the stack (unless returned through a register)
- The old `rp` and `fp` are restored.
- `ip` is set to the instruction after the call site (`ip = rp`).

After a function returns

- The return value is read from the stack or a register.
- Other items from the stack are popped.
- "Caller-save" registers are restored.

	Mnemonic	Operands	Description
13	`JAL`	`ra`	"Jump and link." Sets the return pointer to the address of the next instruction, then jumps to the address in register #`ra`.
14	`RET`		"Return." Jumps to the address in the return pointer.
15	`PUSH`	`rs`	Pushes the value of register #`rs` onto the stack. The value of the register is stored in memory at the top of the stack, and the stack pointer is then incremented.
16	`POP`	`rd`	Pops a value from the stack into register #`rd`. The stack pointer is decremented, and the value in memory at the top of the stack is loaded into the given register.
17	`LDLO`	`imm rd`	Loads the local value found at offset `imm` from the frame pointer into register #`rd`. The offset may be positive (for local variables) or negative (for arguments).
18	`STLO`	`imm rs`	Stores the value of register #`rs` to the local value found at offset `imm` from the frame pointer.

Coming up:

- **Due Thurs, Apr 30.** *Read prelab reading and take quiz*
- **Due Fri, May 1.** *Do Project 8, GCD in pseudo-assembly*
  (Late days may not be used on Project 8)