PC Architecture and Processor Setup

Outline

- x86 instruction set
- GCC calling conventions

x86 Instruction Set

- Intel syntax: `op dst, src` (Intel manuals!)
- AT&T (gcc/gas) syntax: `op src, dst` (labs, xv6)
  - uses b, w, l suffix on instructions to specify size of operands
- Operands are registers, constant, memory via register, memory via constant
- Examples:

AT&T syntax	"C"-ish equivalent	
movl %eax, %edx	edx = eax;	*register mode*
movl $0x123, %edx	edx = 0x123;	*immediate*
movl 0x123, %edx	edx = *(int32_t*)0x123;	*direct*
movl (%ebx), %edx	edx = *(int32_t*)ebx;	*indirect*
movl 4(%ebx), %edx	edx = *(int32_t*)(ebx+4);	*displaced*

- Instruction classes
  - data movement: MOV, PUSH, POP, ...
  - arithmetic: TEST, SHL, ADD, AND, ...
  - i/o: IN, OUT, ...
  - control: JMP, JZ, JNZ, CALL, RET
  - string: REP MOVSB, ...
  - system: IRET, INT
- Intel architecture manual Volume 2 is *the* reference

gcc x86 calling conventions

- x86 dictates that stack grows down:

Example instruction	What it does
pushl %eax	subl $4, %esp movl %eax, (%esp)
popl %eax	movl (%esp), %eax addl $4, %esp
call 0x12345	pushl %eip (*) movl $0x12345, %eip (*)
ret	popl %eip (*)

  (*) *Not real instructions*
- Use example of a function foo() calling a function bar() with two arguments. Assume that bar returns the sum of the two arguments; write bar's code in C and assembly, and explain the conventions.
- Formally, introduce the conventions. GCC dictates how the stack is used. Contract between caller and callee on x86:
  - at entry to a function (i.e. just after call):
    - %eip points at first instruction of function
    - %esp+4 points at first argument
    - %esp points at return address
  - after ret instruction:
    - %eip contains return address
    - %esp points at arguments pushed by caller
    - called function may have trashed arguments
    - %eax (and %edx, if return type is 64-bit) contains return value (or trash if function is `void`)
    - %eax, %edx (above), and %ecx may be trashed
    - %ebp, %ebx, %esi, %edi must contain contents from time of `call`
  - Terminology:
    - %eax, %ecx, %edx are "caller save" registers
    - %ebp, %ebx, %esi, %edi are "callee save" registers

- Discuss the frame pointer, and why it is needed --- so that the name of an argument, or a local variable does not change throughout the function body. Discuss the implications of using a frame pointer on the prologue and epilogue of the function body.
- Functions can do anything that doesn't violate contract. By convention, GCC does more:
  - each function has a stack frame marked by %ebp, %esp

```
               +------------+   |
               | arg 2      |   \
               +------------+    >- previous function's stack frame
               | arg 1      |   /
               +------------+   |
               | ret %eip   |   /
               +============+
               | saved %ebp |   \
        %ebp-> +------------+   |
               |            |   |
               |   local    |   \
               | variables, |    >- current function's stack frame
               |    etc.    |   /
               |            |   |
               |            |   |
        %esp-> +------------+   /
```

  - %esp can move to make stack frame bigger, smaller
  - %ebp points at saved %ebp from previous function, chain to walk stack
  - function prologue:

```
pushl %ebp
movl %esp, %ebp
```

  - function epilogue can easily find return EIP on stack:

```
movl %ebp, %esp
popl %ebp
```

- The frame pointer (in `ebp`) is not strictly needed because a compiler can compute the address of its return address and function arguments based on its knowledge of the current depth of the stack pointer (in `esp`).
- The frame pointer is useful for debugging purposes, especially the current backtrace (function call chain) can be computed by following the frame pointers. The current function is based on the current value of `eip`. The current value of `*(ebp+4)` provides the return address of the caller. The current value of `*((*ebp) + 4)` (where `*ebp` contains the saved `ebp` of the caller) provides the return address of the caller's caller, the current value of `*(*(*ebp) + 4)` provides the return address of the caller's caller's caller, and so on . . .
- Discuss the code for the "backtrace" function in gdb.
- Compiling, linking, loading:
  - *Preprocessor* takes C source code (ASCII text), expands #include etc, produces C source code
  - *Compiler* takes C source code (ASCII text), produces assembly language (also ASCII text)
  - *Assembler* takes assembly language (ASCII text), produces `.o` file (binary, machine-readable!)
  - *Linker* takes multiple '`.o`'s, produces a single *program image* (binary)
  - *Loader* loads the program image into memory at run-time and starts it executing