6.828 Lecture Notes: x86 and PC architecture

Outline

- PC architecture
- x86 instruction set
- gcc calling conventions
- PC emulation

PC architecture

- A full PC has:
    - an x86 CPU with registers, execution unit, and memory management
    - CPU chip pins include address and data signals
    - memory
    - disk
    - keyboard
    - display
    - other resources: BIOS ROM, clock, ...
- We will start with the original 16-bit 8086 CPU (1978)
- CPU runs instructions:

```
for(;;){
        run next instruction
}
```

- Needs work space: registers
    - four 16-bit data registers: AX, BX, CX, DX
    - each in two 8-bit halves, e.g. AH and AL
    - very fast, very few
- More work space: memory
    - CPU sends out address on address lines (wires, one bit per wire)
    - Data comes back on data lines
    - *or* data is written to data lines
- Add address registers: pointers into memory
    - SP - stack pointer
    - BP - frame base pointer
    - SI - source index
    - DI - destination index
- Instructions are in memory too!
    - IP - instruction pointer (PC on PDP-11, everything else)
    - increment after running each instruction
    - can be modified by CALL, RET, JMP, conditional jumps
- Want conditional jumps
    - FLAGS - various condition codes
        - whether last arithmetic operation overflowed
        - ... was positive/negative
        - ... was [not] zero
        - ... carry/borrow on add/subtract
        - ... etc.
        - whether interrupts are enabled
        - direction of data copy instructions

- JP, JN, J[N]Z, J[N]C, J[N]O ...
- Still not interesting - need I/O to interact with outside world
  - Original PC architecture: use dedicated *I/O space*
    - Works same as memory accesses but set I/O signal
    - Only 1024 I/O addresses
    - Accessed with special instructions (IN, OUT)
    - Example: write a byte to line printer:

```
#define DATA_PORT    0x378
#define STATUS_PORT  0x379
#define   BUSY 0x80
#define CONTROL_PORT 0x37A
#define   STROBE 0x01
void
lpt_putc(int c)
{
  /* wait for printer to consume previous byte */
  while((inb(STATUS_PORT) & BUSY) == 0)
    ;

  /* put the byte on the parallel lines */
  outb(DATA_PORT, c);

  /* tell the printer to look at the data */
  outb(CONTROL_PORT, STROBE);
  outb(CONTROL_PORT, 0);
}
```

  - Memory-Mapped I/O
    - Use normal physical memory addresses
      - Gets around limited size of I/O address space
      - No need for special instructions
      - System controller routes to appropriate device
    - Works like ``magic'' memory:
      - *Addressed* and *accessed* like memory, but ...
      - ... does not *behave* like memory!
      - Reads and writes can have ``side effects''
      - Read results can change due to external events
- What if we want to use more than 2^16 bytes of memory?
  - 8086 has 20-bit physical addresses, can have 1 Meg RAM
  - the extra four bits usually come from a 16-bit "segment register":
  - CS - code segment, for fetches via IP
  - SS - stack segment, for load/store via SP and BP
  - DS - data segment, for load/store via other registers
  - ES - another data segment, destination for string operations
  - virtual to physical translation: pa = va + seg*16
  - e.g. set CS = 4096 to execute starting at 65536
  - tricky: can't use the 16-bit address of a stack variable as a pointer
  - a *far pointer* includes full segment:offset (16 + 16 bits)
  - tricky: pointer arithmetic and array indexing across segment boundaries
- But 8086's 16-bit addresses and data were still painfully small
  - 80386 added support for 32-bit data and addresses (1985)
  - boots in 16-bit mode, boot.S switches to 32-bit mode
  - registers are 32 bits wide, called EAX rather than AX
  - operands and addresses that were 16-bit became 32-bit in 32-bit mode, e.g. ADD does 32-bit arithmetic

- prefixes 0x66/0x67 toggle between 16-bit and 32-bit operands and addresses: in 32-bit mode, MOVW is expressed as 0x66 MOVW
- the .code32 in boot.S tells assembler to generate 0x66 for e.g. MOVW
- 80386 also changed segments and added paged memory...

- Example instruction encoding

```
b8 cd ab             16-bit CPU,  AX <- 0xabcd
b8 34 12 cd ab       32-bit CPU, EAX <- 0xabcd1234
66 b8 cd ab          32-bit CPU,  AX <- 0xabcd
```

x86 Physical Memory Map

- The physical address space mostly looks like ordinary RAM
- Except some low-memory addresses actually refer to other things
- Writes to VGA memory appear on the screen
- Reset or power-on jumps to ROM at 0xfffffff0 (so must be ROM at top...)

```
+------------------+  <- 0xFFFFFFFF (4GB)
|      32-bit      |
|  memory mapped   |
|     devices      |
|                  |
/\/\/\/\/\/\/\/\/\/\

/\/\/\/\/\/\/\/\/\/\
|                  |
|      Unused      |
|                  |
+------------------+  <- depends on amount of RAM
|                  |
|                  |
| Extended Memory  |
|                  |
|                  |
+------------------+  <- 0x00100000 (1MB)
|     BIOS ROM     |
+------------------+  <- 0x000F0000 (960KB)
|  16-bit devices, |
|  expansion ROMs  |
+------------------+  <- 0x000C0000 (768KB)
|   VGA Display    |
+------------------+  <- 0x000A0000 (640KB)
|                  |
|    Low Memory    |
|                  |
+------------------+  <- 0x00000000
```

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*
- 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
- 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
  ```

  or

  ```
  enter $0, $0
  ```

  enter usually not used: 4 bytes vs 3 for pushl+movl, not on hardware fast-path anymore
- function epilogue can easily find return EIP on stack:

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

  or

  ```
  leave
  ```

  leave used often because it's 1 byte, vs 3 for movl+popl

- Big example:
  - C code

    ```
    int main(void) { return f(8)+1; }
    int f(int x) { return g(x); }
    int g(int x) { return x+3; }
    ```

  - assembler

    ```
    _main:
                                    prologue
            pushl %ebp
            movl %esp, %ebp
                                    body
            pushl $8
            call _f
            addl $1, %eax
                                    epilogue
            movl %ebp, %esp
            popl %ebp
            ret
    _f:
                                    prologue
            pushl %ebp
            movl %esp, %ebp
                                    body
            pushl 8(%esp)
    ```

```
        call _g
                                epilogue
        movl %ebp, %esp
        popl %ebp
        ret

_g:
                                prologue
        pushl %ebp
        movl %esp, %ebp
                                save %ebx
        pushl %ebx
                                body
        movl 8(%ebp), %ebx
        addl $3, %ebx
        movl %ebx, %eax
                                restore %ebx
        popl %ebx
                                epilogue
        movl %ebp, %esp
        popl %ebp
        ret
```

- Super-small `_g`:

```
_g:
        movl 4(%esp), %eax
        addl $3, %eax
        ret
```

- Shortest `_f`?
- 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

PC emulation

- The Bochs emulator works by
  - doing exactly what a real PC would do,
  - only implemented in software rather than hardware!
- Runs as a normal process in a "host" operating system (e.g., Linux)
- Uses normal process storage to hold emulated hardware state: e.g.,
  - Stores emulated CPU registers in global variables

```
int32_t regs[8];
#define REG_EAX 1;
#define REG_EBX 2;
#define REG_ECX 3;
...
int32_t eip;
int16_t segregs[4];
...
```

- Stores emulated physical memory in Boch's memory

```
char mem[256*1024*1024];
```

- Execute instructions by simulating them in a loop:

```
for (;;) {
        read_instruction();
        switch (decode_instruction_opcode()) {
        case OPCODE_ADD:
                int src = decode_src_reg();
                int dst = decode_dst_reg();
                regs[dst] = regs[dst] + regs[src];
                break;
        case OPCODE_SUB:
                int src = decode_src_reg();
                int dst = decode_dst_reg();
                regs[dst] = regs[dst] - regs[src];
                break;
        ...
        }
        eip += instruction_length;
}
```

- Simulate PC's physical memory map by decoding emulated "physical" addresses just like a PC would:

```
#define KB              1024
#define MB              1024*1024

#define LOW_MEMORY      640*KB
#define EXT_MEMORY      10*MB

uint8_t low_mem[LOW_MEMORY];
uint8_t ext_mem[EXT_MEMORY];
uint8_t bios_rom[64*KB];

uint8_t read_byte(uint32_t phys_addr) {
        if (phys_addr < LOW_MEMORY)
                return low_mem[phys_addr];
        else if (phys_addr >= 960*KB && phys_addr < 1*MB)
                return rom_bios[phys_addr - 960*KB];
        else if (phys_addr >= 1*MB && phys_addr < 1*MB+EXT_MEMORY) {
                return ext_mem[phys_addr-1*MB];
        else ...
}

void write_byte(uint32_t phys_addr, uint8_t val) {
        if (phys_addr < LOW_MEMORY)
                low_mem[phys_addr] = val;
        else if (phys_addr >= 960*KB && phys_addr < 1*MB)
                ; /* ignore attempted write to ROM! */
        else if (phys_addr >= 1*MB && phys_addr < 1*MB+EXT_MEMORY) {
                ext_mem[phys_addr-1*MB] = val;
        else ...
}
```

- Simulate I/O devices, etc., by detecting accesses to "special" memory and I/O space and emulating the correct behavior: e.g.,
  - Reads/writes to emulated hard disk transformed into reads/writes of a file on the host system

- Writes to emulated VGA display hardware transformed into drawing into an X window
- Reads from emulated PC keyboard transformed into reads from X input event queue