x86_16 real mode

(or at least enough for cos318 project 1)

Overview

- Preliminary information - How to find help
- The toolchain
- The machine

If you only remember one thing: gcc -S

- the -S (capital S) flag causes gcc to ouput assembly.

Preliminary Information

- Assembly can be hard
- Development strategies conquer risk:
  - Write small test cases.
  - Write functions, test each separately.
  - Print diagnostics frequently.
- Think defensively!
  - and the interweb is helpful too.

The Interwebs as a resource.

- The internet offers much information that seems confusing or contradictory.
- How do you sort out information "in the wild?"

Syntax

- There are (at least) two different syntaxes for x86 assembly language: AT&T and Intel.
    - AT&T: opcodes have a suffix to denote data type, use sigils, and place the destination operand on the right.
    - Intel: operands use a keyword to denote data type, no sigils, destination operand is leftmost.

Example: AT&T vs Intel

```
push   %bp
mov    %sp,%bp
sub    $0x10,%sp
movw
   0x200b(%bx),%si
mov    $0x4006,%di
mov    $0x0,%ax
call  printf
leaveq
retq
```

```
push   bp
mov    bp,sp
sub    sp,0x10
mov    si,WORD
    PTR [bx+0x200b]
mov    di,0x4006
mov    ax,0x0
call  printf
leave
ret
```

In this class, use AT&T!

Versions of the architecture

- x86 won't die. All backwards compatible.
    - 8086 -> 16bit, Real
    - 80386 / ia32 -> 32bit, Protected
    - x86_64 -> 64bit, Protected
- If you find an example:
    - For which architecture was it written?

The Register Test

- If you see "%rax", then 64-bit code; else
- If you see "%eax", then 32-bit code; else
- You are looking at 16-bit code.

Overview

- Preliminary information - How to find help
- The toolchain
- The machine

The toolchain

- The lab has all the software you need. You can connect remotely via *ssh -X labpc-yy*

- All software is available for free on *nix, Mac OS X, and probably windows.

- If you use a 64-bit machine, you may have problems.
    - Ask me offline.

Text editors

- You should know how to use an editor
- vi and emacs are popular choices...
    - ...and you should learn them, if for no other reason than to understand geek jokes.
        - s/bug/feature/
        - M-x psychoanalyze-pinhead

The Assembler: *as* or *gas*

- The cycle:
  - You write an assembly language text file (.s)
  - run: *as --32 -g source.s -o obj.o*
- A disassembler is also useful:
  - objdump -D -M i8086 obj.o > obj.s

- We have provided a makefile to make this painless

bochs

- bochs ("box") is a free, open-source emulator of a complete PC
- How do we use it?
  - Bochs treats a file as a disk in the emulated computer.
  - The computer will boot off of it.
- bochs will be discussed more in later precepts.

Overview

- Preliminary information - How to find help
- The toolchain
- The machine

Scope

- This is not an exhaustive list of x86 features.
    - It's just enough to get you rolling.

- In fact, I want to discourage some of the more advanced uses. If you keep it simple, it will be easier to develop, debug, and grade.

Again: gcc -S

- *gcc -S -m32 - fomit-frame-pointer test.c -o test.s*

About optimizing your code.

- DON"T OPTIMIZE YOUR CODE!!!111!!!!
    - I will have to read your code.

- Please keep-it-simple.
    - Memory access in separate instructions.
    - Use .EQU to give names to constants.
    - Comments that say what you're trying to do.

Caution: x86 is wonky.

- a lot of instructions, many redundant.
- very few registers, and funny rules about what each may do.
- Real vs Protected modes; Segmented Memory!
- Here, we focus on a sane subset of x86.

The syntax of a .s file

- # comment
- Register names have the %-sigil, eg %ax
- Literals have the $-sigil, eg $0x1234
    - Literals without the $-sigil mean memory!
- label:
- Instructions may have suffixes -b (byte, 8-bit) or -w (word, 16-bit).

x86_16 Registers

- General purpose registers:
    - %ax, %bx, %cx, %dx
    - %*a* h is the most-significant byte
    - %*a* l is the least.
- Pointer registers:
    - %si, %di, %sp, %bp, %ip
- Segment registers:
    - %ds, %es, %cs, %ss
- Control register:
    - %flags

Segmented Memory on x86

- Good news: you can mostly ignore it at the local instruction level.

- Bad news: you need to understand it to complete this project.

- Why is it here? In the good-ole' days...
    - pointers were small, and
    - we didn't have memory management units.

Segmented Memory: Why?

- Some machine instructions must contain memory locations.
- But, your compiler cannot know what other programs are running...
  - ...or what addresses they use.
- A layer of abstraction between instructions and physical memory solves this problem.
  - Put the code *anywhere* in *physical* memory, but give it the *logical* address it desires.

Segmented Memory on x86

- Segmented memory is a hack.
- Makes pointers slightly larger.
- Provides rudimentary support for relocation.

- Intel's solution:
    - Memory is many **overlapping** segments.
    - A pointer is an address within a segment.
    - A segment register adds 4-bits to the address space.

Segmented Memory on x86

- Suppose segment register %ds holds a segment number
- Suppose register %bx holds an address.
- Then %ds:%bx is a *logical* memory address.

- The *physical* address in memory is:
  - %ds:%bx == 16 * %ds + %bx

- The pointer is 4 bits wider.

Segments as Relocation

- Observe that:
  - x:y == (x+1):(y-16)
  - x:y == (x-1):(y+16)

- Say you have code that assumes it is at memory address zero...
- ...but, we're using address zero for something else...
- Adjust segment registers, and give the illusion that the code is at the desired address.

How segments help us in P1

- The bootloader must move itself to another physical memory location, as to make room for the kernel.
- Segmentation allows us to move, but keep logical memory addresses the same.

How segments hurt us in P1

- If the kernel is bigger than a segment (64KiB), then you will need to perform several disk reads to different segments :(
    - This is why support for >128 sectors is extra credit.

Practical Ex. of Segments

- For project 1, we write bootblock.s
- The assembler assumes logical address 0, but on x86 that address is reserved.
- Instead, BIOS loads the bootloader to 0x0:0x7c00
- Although the physical memory address has changed, 0x0: 0x7c00==0x07c0:0x0.

- If you read/write memory through segment 0x07c0, everything works as usual...

Practical Ex. of Segments

- We want to the kernel at physical address 0x0:0x1000.
- If the kernel is >27KiB,then boot loader and kernel overlap!
- Need to relocate the boot loader.

x86 Instructions

- Next, I'm going to show a bunch of instructions and their semantics.
- I'll write a general form, then the RTL semantics.
    - Memory
    - Stacks
    - Arithmetic
    - Control

x86: Memory

- mov*w* ptr, r
  - r ← Mem[ptr] (16-bit)
- mov*w* r,ptr
  - Mem[ptr] ← r (16-bit)
- where, ptr is an address expression:
  - 0x1234 - absolute address (no $-sigil)
  - (r) - address specified in register.
  - 0x1234(r) - r+0x1234
  - etc
- In segment %ds by default!

x86: More Memory

- lods*w*
    - %ax ← Mem[%ds:%si]
    - %si++
- movs*w*
    - %Mem[%es:%di] ← %Mem[%ds:%si]
    - %si++
    - %di++
- may prefix with rep:
    - rep *foo* : while(%cx != 0) { *foo* ; %cx--; }

x86: Stacks

- push x
  - --%sp
  - Mem[%ss:%sp] ← x
- pop x
  - x ← Mem[%ss:%sp]
  - %sp++

x86: Arithmetic

- addw / subw x,y
  - y ← y +/- x
- mulw r
  - %dx:%ax ← %ax * r
- divw r
  - %ax ← %dx:%ax div r
  - %dx ← %dx:%ax mod r

- inc / dec r
  - r ← r +/- 1

x86: Control

- cmpw x,y
    - if y-x == 0, set %flags<z> ←1
    - if y-x < 0, set %flags<c> ← 1
- jmp <label>
    - %ip ← label
- jz <label>
    - if %flags<z>==1, then %ip←label
- jc <label>
    - if %flags<c>==1, then %ip←label

x86: Calls

- call <label>
  - push %ip
  - jmp label
- ret
  - pop %ip

x86: More Control

- Segments aren't just for data!
  - %cs:%ip points to next instruction.

- ljmp <imm1>, <imm2>
  - %cs ← imm1
  - %ip ← imm2
- lret
  - pop %ip
  - pop %cs

x86: Software interrupts!

- int <immediate> : invoke a software interrupt.
    - int 0x10 - console output
    - int 0x13 - disk I/O
    - int 0x16 - keyboard input
- Each interrupt offers several functions.
- Specific function chosen by %ah
    - e.g. int 0x10, function %ah=02 means read disk sector.
- int 0x21 CANNOT BE USED.

Passing parameters to fcns

- No standard.
- High-level languages use stack frames.
- For P1, I recommend:
    - pass the first parameter in %ax, the second in %bx, and so on.
    - place the return value in %ax.
    - (and write comments)

x86: Common Control Patterns

- How do we combine these instructions into programs?

- if-then-else
- for-loop

x86: if-then-else

if(x < 10) { *foo* } else { *bar* }

```
    movw ($x), %ax
    cmpw $0xa, %ax
    jnc elseClause
thenClause:
    foo
    jmp endIf
elseClause:
    bar
endIf:
```

x86: for-loops

```
for(x=0; x<10; x++) { foo }

    movw $0, %cx                # use reg %cx to hold x
continueLoop:
    foo
    incw %cx
    cmpw $0xa, %cx
    jc continueLoop
breakLoop:
```

x86: Troubleshooting.

- What is the difference:
  - movw $label, %ax
  - movw label, %ax
- Why can't I write:
  - movw $label, %es
- How do I compute the size of something:
  - before:
  -     ...
  - after:
  -     mov $(after - before), %ax

Assembler Directives

- Begin with a period (.) Not instructions!
- .equ name,value
    - "equate", just like #define name value
- .code16
    - assemble code as 16-bit instructions
- .byte <imm>
    - emit the byte imm into the object file
- .word <imm>
    - emit the 16-bit word imm.
- .string "Hello World\n\r\0"
    - emit the string.

Segments in a .s file

- Organized into segments which can be relocated independently
- .text begins the "text" (or code) segment
- .data begins the "data" segment

Memory on a PC

- 0:0--0:3ff: Reserved. IVT
- 0:400--0:4ff: Reserved. Various.

- *0:500--9000:ffff: Available*
- b000:0--c000:0: Video Memory
- Everything else is reserved by various ROMs.

Disks on an PC

- Disks:
  - are divided into cylinders
    - are divided into heads
      - are divided into sectors
        - are 512 bytes.
- Disk parameters can be queried from BIOS.
- We would like to *linearize* disk addressing
  - "Logical Block Addressing" one way...

Conclusion

- gcc -S
- Keep it simple!
- segments OVERLAP and can be used for relocation
- And... we're here to help.