Review of assembly language

• Program “text” contains binary instructions
  - CPU executes one instruction at a time
  - Usually executes next sequential instruction in memory
  - Branch/jump/call inst. specifies different next instruction

• Instructions typically manipulate
  - Registers – small number of values kept by processor
  - Memory
  - “Special” registers whose bits have particular significance
  - The instruction pointer (IP) – which inst. to execute next
  - I/O devices
Review of x86 assembly

- Mostly two operand instructions
- Unfortunately *two* prevalent syntaxes
  - “Intel syntax”: op dst, src
  - “AT&T (gcc/gas) syntax”: op src, dst
  - We will always use AT&T syntax
  - But a lot of documentation uses Intel syntax

- Examples:

<table>
<thead>
<tr>
<th>Assembly</th>
<th>C pseudo-code</th>
</tr>
</thead>
<tbody>
<tr>
<td>movl %eax,%edx</td>
<td>edx = eax;</td>
</tr>
<tr>
<td>movl $0x123, %edx</td>
<td>edx = 0x123;</td>
</tr>
<tr>
<td>movl 0x124, %edx</td>
<td>edx = <em>((int32_t</em>) 0x124);</td>
</tr>
<tr>
<td>movl (%ebx), %edx</td>
<td>edx = <em>((int32_t</em>) ebx);</td>
</tr>
<tr>
<td>movl 4(%ebx), %edx</td>
<td>edx = <em>((int32_t</em>) (ebx+4));</td>
</tr>
</tbody>
</table>
Real vs. protected mode

- **Real mode** – 16-bit registers, 1 MB virtual mem
  - Segment registers provide top 4 bits of physical address:
    \[
    \text{movw} (%ax),%dx \text{ means } dx = \ast (\text{int}_32\_t\ast)(16 \times ds + ax)
    \]
  - This is probably what you’ve used in earlier classes

- **Protected mode** – segment registers virtualized
  - Load segment registers from table of *segment descriptors*
  - Depending on `%cs` descriptor, default ops can be 32 bits
  - 32-bit virtual address space, can optionally be paged
  - 32- or 36-bit physical address space, depending on mode

- **We will mostly use 32-bit protected mode**
  - All remaining examples will be 32-bit code
  - 32-bit AT&T Instructions have `l` suffix, for long
More 32-bit instructions

- **ALU ops:** addl, subl, andl, orl, xorl, notl, ...
  - incl, decl – add or subtract 1
  - cmpl – like subl, but discards subtraction result

- **Stack instructions:**

<table>
<thead>
<tr>
<th>Stack op</th>
<th>equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>pushl %eax</td>
<td>subl $4,%esp</td>
</tr>
<tr>
<td></td>
<td>movl %eax,(%esp)</td>
</tr>
<tr>
<td>popl %eax</td>
<td>movl (%esp),%eax</td>
</tr>
<tr>
<td></td>
<td>addl $4,%esp</td>
</tr>
</tbody>
</table>

- **Other stack instructions:** pushfl, pushal
  - leave means: movl %ebp,%esp; popl %ebp
Conditional branches

- Conditional branches based EFLAGS reg. bits
  - CF (carry flag) set if op carried/borrowed → jc, jnc
  - ZF (zero flag) set if result zero → jz/je, jnz/jne
  - SF (sign flag) set to high bit of result → jn, jp
  - OF (overflow flag) set if result to large → jo, jno
  - jge → “Jump if greater or equal”, i.e., SF=OF
  - jg → “Jump if greater”, i.e., SF=OF and ZF=0

- jmp unconditional jump, call/ret uses stack:

<table>
<thead>
<tr>
<th>Stack op</th>
<th>pseudo-asm equiv</th>
</tr>
</thead>
<tbody>
<tr>
<td>call $0x12345</td>
<td>pushl %eip</td>
</tr>
<tr>
<td></td>
<td>movl $0x12345,%eip</td>
</tr>
<tr>
<td>ret</td>
<td>popl %eip</td>
</tr>
</tbody>
</table>
Example

```c
for (i = 0; i < a; i++)
    sum += i;
```

```assembly
xorl %edx,%edx  # i = 0 (more compact than movl)
cmpl %ecx,%edx  # test (i - a)
jge .L4          # >= 0 ? jump to end
movl sum,%eax   # cache value of sum in register

.L6:
addl %edx,%eax  # sum += i
incl %edx       # i++
cmpl %ecx,%edx  # test (i - a)
jl .L6           # < 0 ? go to top of loop
movl %eax,sum   # store value of sum back in memory
.L4:
```
Assembler local labels

- Often want to define macros in assembly language
  - Typically .S files are C-preprocessor source

- Problem: how to choose unique labels
  - If there’s a loop in macro, and used multiple times
  - You would have a duplicate label

- Solution: Numeric labels are local
  - f suffix means forwards
  - b suffix means backwards
Example w. local labels

for (i = 0; i < a; i++)
    sum += i;

 常用的本地标记

xorl %edx,%edx       # i = 0 (more compact than movl)
cmpl %ecx,%edx       # test (i - a)
jge 2f               # >= 0 ? jump to end
movl sum,%eax        # cache value of sum in register

1:

addl %edx,%eax       # sum += i
incl %edx            # i++
cmpl %ecx,%edx       # test (i - a)
jl 1b                # < 0 ? go to top of loop
movl %eax,sum        # store value of sum back in memory

2:
32-bit protected-mode registers

**Caller-saved:**
- %eax
- %edx
- %ecx

**Callee-saved:**
- %ebx
- %esi
- %edi

- %ebp ← frame pointer
- %esp ← stack pointer

**Special-purpose:**
- eflags, %cr3, GDTR, IDTR, LDTR, TSS

**Segment Registers:** %cs %ss %ds %es [%fs %gs]

<table>
<thead>
<tr>
<th>INDEX</th>
<th>TI</th>
<th>RPL</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>...</td>
<td>3</td>
</tr>
</tbody>
</table>

$TI : 0 = \text{global}/1 = \text{local table}$

$RPL : \text{Requestor privilege level} \,(0–3)$
Calling conventions

- GCC dictates how stack is used

- **After call instruction:**
  - `%esp` points at return address
  - `%esp+4` points at first argument

- **After ret:**
  - `%esp` points at arguments pushed by caller
  - called function may have trashed arguments
  - `%eax` contains return value (or trash if function is void)
  - `%ecx`, `%edx` may be trashed
  - `%ebp`, `%ebx`, `%esi`, `%edi` must have previous contents
- Code may push temp vars on stack at any time
  - So refer to args and locals using %ebp
Typical function code

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

------------------------------------------------------------main:

```asm
pushl %ebp
movl %esp,%ebp
...
pushl $8
call f
incl %eax
leave
ret
```
code for f

```c
int f(int x) { return g(x); }
```

f:
```
pushl  %ebp
movl   %esp, %ebp
subl   $20, %esp
pushl  8(%ebp)
call   g
leave
ret
```
code for g

```c
int g(int x) { return x+3; }
```

g:
```
pushl  %ebp
movl   %esp, %ebp
movl   8(%ebp), %eax
addl   $3, %eax
leave
ret
```
Inline assembly language

• **Large assembly language files are a pain**
  - Often want to write C, but need a particular asm instruction
  - Thus, gcc provides asm extension

• **Straw man, just inject assembly language:**
  - E.g., `asm ("movl %esp,%eax");`
  - But what if compiler needed value in %eax?
  - And what if you need some value the compiler has?
    (remember how gcc cached value of sum in %eax)
GCC inline assembly language

- Specify values needed, output, and clobbered

  ```c
  asm ("statements" : output_values
       : input_values : clobbered);
  ```

- Example:

  ```c
  u_int32_t stkp;
  asm ("movl %esp,%0" : "=r" (stkp) ::);
  printf ("The stack pointer is 0x%x\n", stkp);
  ```

- Notes:
  - “r” means any register, or can specify w. a/b/c/d/S/D
  - “m” means memory, “g” general, I small constant
  - If in/out value same, specify, e.g., “0” for in value
  - clobbered may need “memory” and/or “flags”
I/O instructions

• How to interact with devices?

• PC design – use special I/O space
  - special instructions inb/inw, outb/outw (for 8/16 bits)
  - Load and store bytes & words, like normal memory
  - But special processor I/O pin says “this is for I/O space”

• To access from C code:

  static inline u_char inb (int port) {
    u_int8_t data;
    asm volatile("inb %w1,%0" : "=a" (data) : "d" (port));
    return data;
  }

  static inline void outb(int port, u_int8_t data) {
    asm volatile("outb %0,%w1" :: "a" (data), "d" (port));
  }
x86 hardware tables

LDT/GDT. Descriptor tables, indexed by segment registers.

IDT. Vectors for 256 exceptions, interrupts, and user traps.

TSS. Task state segment.

- Stack pointers for privilege increases.
- I/O-space permissions with byte granularity (allows cli).

Page Directory/Tables. Two-level page tables in hardware.

<table>
<thead>
<tr>
<th>VA:</th>
<th>DIR</th>
<th>TABLE</th>
<th>OFFSET</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 ... 22</td>
<td>21 ... 12</td>
<td>11 ... 0</td>
<td></td>
</tr>
</tbody>
</table>

Special register %cr3 points to page directory.
x86 segments

32 types of segments: 16/32-bit, expand-up/down, read/write, code/data, conforming/non-conforming, call/trap/interrupt/task gate, available/busy TSS, LDT.

- **user segments.** 32-bit base, 16-bit limit (granularity byte/4K). RPL bits of %cs and %ss determine current privilege level.
- **trap gates.** 16-bit segment selector, 32-bit offset.
- **interrupt gates.** Same as trap gates, but disables interrupts.

Loading segments:

- direct load, far jump, int: $\text{MIN}(\text{CPL}, RPL) \leq DPL$
- exception, interrupt: $DPL$ not checked
- all gates: adjust $CPL$ to $DPL$ of designated segment.
Segments are mostly a pain

- Segment base + offset known as *linear address*
- Usually don’t want to worry about segments
  - But can’t disable segmentation hardware
- **Solution:** Flat model – offset = linear address
  - Give all segments a base address of 0
  - Now mostly don’t have to worry about segments
- **However, still need segments for interrupts/traps**
x86 paging

- Translation occurs on linear address output of segmentation.

- 4K pages.

- PTEs have the following options:
  - **writeable.** Disables user and kernel (486+) mode writes.
  - **user.** Access with $CPL = 3$ when set, otherwise just 0–2.
  - cache disable bit, cache write-through bit
  - dirty bit, accessed bit, present bit.

- `%cr3` designates address space by selecting page directory. Loading `%cr3` flushes the TLB.
32 bits aligned onto a 4-KByte boundary

Directory  Table  Offset

Page Directory

Directory Entry

Page Table

Page-Table Entry

Physical Address

4-KByte Page

Linear Address

31  22  21  12  11  0

1024 PDE × 1024 PTE = 2^{20} Pages

*32 bits aligned onto a 4-KByte boundary
<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page Base Address</td>
<td>31</td>
<td></td>
</tr>
<tr>
<td>Avail</td>
<td>12</td>
<td>Available for system programmer’s use</td>
</tr>
<tr>
<td>G</td>
<td>11</td>
<td>Global Page</td>
</tr>
<tr>
<td>PAT</td>
<td>9</td>
<td>Page Table Attribute Index</td>
</tr>
<tr>
<td>D</td>
<td>8</td>
<td>Dirty</td>
</tr>
<tr>
<td>A</td>
<td>7</td>
<td>Accessed</td>
</tr>
<tr>
<td>PCD</td>
<td>6</td>
<td>Cache Disabled</td>
</tr>
<tr>
<td>PWT</td>
<td>5</td>
<td>Write-Through</td>
</tr>
<tr>
<td>URS</td>
<td>4</td>
<td>User/Supervisor</td>
</tr>
<tr>
<td>R/W</td>
<td>3</td>
<td>Read/Write</td>
</tr>
<tr>
<td>Present</td>
<td>2</td>
<td>Present</td>
</tr>
<tr>
<td>P</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

The diagram shows a page-table entry for a 4-KByte page, with various fields indicating different attributes such as availability, global page, and page table attribute index. Each field is represented by a bit position in the entry, with specific functionalities and states associated with each.
Interrupts and traps

- **CPU supports 256 interrupts**
  - IDT contains segment descriptors for each int
  - Trap gate says what code segment / offset to use
  - Interrupt gate like trap gate, but disables interrupts

- **How does CPU vector to IDT entry?**
  - int, int3, into instructions
  - Built-in trap (e.g., page fault, trap numbers hard-coded 0–19)
  - Interrupt from external device (8-bit interrupt number supplied on CPU pins)
Trap frame

- Only some traps have error codes
- Interrupts do not cause error code to be pushed
Example: page fault – 14

- Has error code, bits mean:
  - bit 0 – 0 = page not present, 1 = protection violation
  - bit 1 – 0 = access was read, 1 = access was write
  - bit 2 – 0 = fault in user mode, 1 = supervisor mode

- In addition, special register %cr2 holds faulting virtual address
Discussion

- Why might page fault occur in supervisor mode?
- Where does stack pointer come from after trap?
  - Why is this important?
- What happens if user code calls int 14?
- W\(\wedge\)X
- 8259A