Objectives

• History of the Intel x86 processor family
• Architecture of the x86 CPU
## History

<table>
<thead>
<tr>
<th>Model</th>
<th>Year</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>8086</td>
<td>1978</td>
<td>- 16 bit registers</td>
</tr>
<tr>
<td>80286</td>
<td>1982</td>
<td>- 20 bit addressing (1MB address space)</td>
</tr>
<tr>
<td>80386</td>
<td>1985</td>
<td>- Protected mode</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- 24 bit addressing (16 MB address space)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- various memory protection mechanisms</td>
</tr>
<tr>
<td>80486</td>
<td>1989</td>
<td>- 32 bit registers</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- introduced paging</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- 32 bit addressing bus (4 GB address space)</td>
</tr>
<tr>
<td>Pentium</td>
<td>1993</td>
<td>- Parallel execution capability</td>
</tr>
<tr>
<td>AMD K8</td>
<td>2000</td>
<td>- Increased performance</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- 64 bit architecture (x86_64)</td>
</tr>
</tbody>
</table>
History

Code created for CPUs released in 1978 still executes on latest CPUs!

<table>
<thead>
<tr>
<th>CPU</th>
<th>Clock Frequency</th>
<th>Transistors per die</th>
<th>Address space</th>
</tr>
</thead>
<tbody>
<tr>
<td>8086</td>
<td>8 MHz</td>
<td>29 K</td>
<td>1 MB</td>
</tr>
<tr>
<td>80486</td>
<td>25 MHz</td>
<td>1.2 M</td>
<td>4 GB</td>
</tr>
<tr>
<td>Pentium 4</td>
<td>1.5 GHz</td>
<td>42 M</td>
<td>64 GB</td>
</tr>
</tbody>
</table>

Moore’s Law (Named after Intel cofounder Gordon Moore):
“The number of transistors that would be incorporated on a silicon die would double every 18 months for the next several years.”
Intel CPU Architecture (32 bit)

- Eight 32-bit Registers
- Six 16-bit Registers
- 32 bits
- 32 bits
- General Purpose Registers
- Segment Registers
- EFLAGS Registers
- EIP (Instruction Pointer Registers)
- Floating-Point Data Registers
- FPU Registers
- Eight 80-bit Registers

Address Space

$2^{32} - 1$
Data Types

- **Byte**: 7 0
- **Word**: 15 7 0
- **DoubleWord**: 31 16 15 0
- **QuadWord**: 63 32 31 0
- **DoubleQuadWord**: 127 64 63 0
Little/Big Endian (1)

Different computer architectures order information in different ways.

\[ X = b_3 \times 2^{24} + b_2 \times 2^{16} + b_1 \times 2^8 + b_0 \times 2^0 \]

Little Endian  
(e.g. Intel x86)

Big Endian  
(e.g. Sun SPARC)
The following C program tests if the machine it is running on has a little or big endian architecture.

```c
#include <stdio.h>

union {
    int x;
    char c[sizeof(int)];
} u;

void main() {
    u.x = 1;
    if (u.c[0] == 1)
        printf(“Little Endian\n”);
    else
        printf(“Big Endian\n”);
}
Data Types

Word at address BH contains FE06H

Byte at address 9H contains 1FH

Word at address 6H contains 230BH

Word at address 2H contains 74CBH

Word at address 1H contains CB31H

12H 1FH 23H 2H
7AH 06H 36H 3H
FEH FEH 23H 4H
EH DH BH 4H

Quadword at address AH contains 7AFE0636BH

Double quadword at address 6H contains 7AFE0636FA4230BH

Double quadword at address 0H contains 127AFE0636FA4230B456774CB3112H
Registers

- Registers are like variables of C, but there exist only a finite amount.
- We will look at Segment Registers later.
## General Purpose Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Lower 8 Bits</th>
<th>Lower 16 Bits</th>
<th>Upper 32 Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX</td>
<td>AL</td>
<td>AX</td>
<td>AH</td>
</tr>
<tr>
<td>EBX</td>
<td>BL</td>
<td>BX</td>
<td>BH</td>
</tr>
<tr>
<td>ECX</td>
<td>CL</td>
<td>CX</td>
<td>CH</td>
</tr>
<tr>
<td>EDX</td>
<td>DL</td>
<td>DX</td>
<td>DH</td>
</tr>
<tr>
<td>EBP</td>
<td>BP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ESI</td>
<td>SI</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EDI</td>
<td>GI</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ESP</td>
<td>SP</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Some registers are only available for certain machine instructions.
EFLAGS Register

- EFLAGS = Extended Flags
- 32 bit register, where each bit indicates a certain status
- Machine instructions such as ADD, SUB, MUL, DIV modify the EFLAGS Register.

<table>
<thead>
<tr>
<th>Flag</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CF</td>
<td>0</td>
<td>Carry Flag: Indicates an overflow condition for unsigned integer arithmetic.</td>
</tr>
<tr>
<td>ZF</td>
<td>6</td>
<td>Zero Flag: Set if the result is zero; cleared otherwise.</td>
</tr>
<tr>
<td>SF</td>
<td>7</td>
<td>Sign Flag: Set equal to the most significant bit of the result.</td>
</tr>
<tr>
<td>OF</td>
<td>11</td>
<td>Overflow Flag: Set if the integer result is too large to fit in the destination operand.</td>
</tr>
<tr>
<td>IF</td>
<td>9</td>
<td>Interrupt Flag: If set, enables the recognition of external interrupts.</td>
</tr>
</tbody>
</table>
x86 Instruction Overview
Objectives

• Quick introduction to x86 assembly
• Provide examples for common use cases
• Show how C is mapped to assembly
• Show how to embed assembly into C-code
Assembly Syntax

• IMPORTANT: all examples will use the 32-bit version of the x86 (TOS runs in 32-bit mode)

• Two major syntaxes for writing x86 assembly code:

• Intel format:
  – destination, source
  – Exists in boot loader (*tools/boot/*.s)

• AT&T syntax:
  – source, destination
  – Produced by gcc, used in all class slides
x86 Instruction Overview

**Memory Operations**
- MOV: move data
- Push: push data onto stack
- Pop: Pop data off the stack

**Logical and arithmetic operations**
- AND: Bitwise and
- OR: Bitwise or
- XOR: Bitwise exclusive or
- ADD: Addition
- SUB: Subtraction

**Control flow operations**
- JMP: Jump
- JZ: Jump if Zero
- JNZ: Jump if NOT Zero
- CALL: Call Subroutine
- RET: Return from subroutine
Anatomy of a Move Instruction

- This instruction will move the value 0x1234 into register %AX
- General format of move instruction: mov src, dest
- NOTE: We assume AT&T assembly syntax!
Move Operations

<table>
<thead>
<tr>
<th>Addr</th>
<th>Machine code</th>
<th>Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:</td>
<td>b0 12</td>
<td>mov $0x12,%al</td>
</tr>
<tr>
<td>2:</td>
<td>b4 34</td>
<td>mov $0x34,%ah</td>
</tr>
<tr>
<td>4:</td>
<td>66 b8 78 56</td>
<td>mov $0x5678,%ax</td>
</tr>
<tr>
<td>8:</td>
<td>b4 ab</td>
<td>mov $0xab,%ah</td>
</tr>
<tr>
<td>a:</td>
<td>bb ef be ad de</td>
<td>mov $0xdeadbeef,%ebx</td>
</tr>
<tr>
<td>f:</td>
<td>66 89 d8</td>
<td>mov %bx,%ax</td>
</tr>
<tr>
<td>12:</td>
<td>88 e3</td>
<td>mov %ah,%bl</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>EIP</th>
<th>EAX</th>
<th>EBX</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00000012</td>
<td>00000000</td>
</tr>
<tr>
<td>2</td>
<td>00003412</td>
<td>00000000</td>
</tr>
<tr>
<td>4</td>
<td>00005678</td>
<td>00000000</td>
</tr>
<tr>
<td>8</td>
<td>0000AB78</td>
<td>00000000</td>
</tr>
<tr>
<td>A</td>
<td>0000AB78</td>
<td>DEADBEEF</td>
</tr>
<tr>
<td>F</td>
<td>0000BEEF</td>
<td>DEADBEEF</td>
</tr>
<tr>
<td>12</td>
<td>0000BEEF</td>
<td>DEADBEBE</td>
</tr>
</tbody>
</table>
Logic / Arithmetic Instructions (1)

- Adds 4 to the value in register %ECX
- Second operand (%ECX in this example) is also the destination
- ADD, SUB for arithmetic
- AND, OR, XOR for Boolean logic
## Logical / Arithmetic Instructions (2)

<table>
<thead>
<tr>
<th>Addr</th>
<th>Machine code</th>
<th>Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:</td>
<td>66 b8 20 00</td>
<td>mov $0x20,%ax</td>
</tr>
<tr>
<td>4:</td>
<td>66 bb 0a 00</td>
<td>mov $0xa,%bx</td>
</tr>
<tr>
<td>8:</td>
<td>66 01 d8</td>
<td>add %bx,%ax</td>
</tr>
<tr>
<td>b:</td>
<td>66 0d 00 34</td>
<td>or $0x3400,%ax</td>
</tr>
<tr>
<td>f:</td>
<td>66 25 00 ff</td>
<td>and $0xff00,%ax</td>
</tr>
<tr>
<td>13:</td>
<td>66 31 db</td>
<td>xor %bx,%bx</td>
</tr>
<tr>
<td>16:</td>
<td>66 43</td>
<td>inc %bx</td>
</tr>
</tbody>
</table>

### EIP, AX, BX

<table>
<thead>
<tr>
<th></th>
<th>EIP</th>
<th>AX</th>
<th>BX</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0020</td>
<td>0000</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>0020</td>
<td>000A</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>002A</td>
<td>000A</td>
</tr>
<tr>
<td>B</td>
<td>B</td>
<td>342A</td>
<td>000A</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>3400</td>
<td>000A</td>
</tr>
<tr>
<td>13</td>
<td>13</td>
<td>3400</td>
<td>0000</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
<td>3400</td>
<td>0001</td>
</tr>
</tbody>
</table>
Jump Instructions

- Jump instruction changes %EIP to modify flow of control
  - Used to implement if statements and loops
- Target of a jump is the address of an instruction (like a C pointer-to-function)
- Assembler labels reference an address
- Instructions:
  - JMP (unconditional jump)
  - JZ (Jump if Zero)
  - JNZ (Jump if Not Zero)
### EFLAGS

<table>
<thead>
<tr>
<th>Addr</th>
<th>Machine code</th>
<th>Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:</td>
<td>66 31 c9</td>
<td>xor %cx,%cx</td>
</tr>
<tr>
<td>3:</td>
<td>66 b8 03 00</td>
<td>mov $0x3,%ax</td>
</tr>
<tr>
<td>7:</td>
<td>66 01 c1</td>
<td>L1: add %ax,%cx</td>
</tr>
<tr>
<td>a:</td>
<td>66 48</td>
<td>dec %ax</td>
</tr>
<tr>
<td>c:</td>
<td>75 f9</td>
<td>jnz L1</td>
</tr>
<tr>
<td>e:</td>
<td>66 89 c8</td>
<td>mov %cx,%ax</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>EIP</th>
<th>AX</th>
<th>CX</th>
<th>Z-Flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>0000</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0003</td>
<td>0000</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>0003</td>
<td>0003</td>
<td>0</td>
</tr>
<tr>
<td>A</td>
<td>0002</td>
<td>0003</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>0002</td>
<td>0003</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0002</td>
<td>0005</td>
<td>0</td>
</tr>
<tr>
<td>A</td>
<td>0001</td>
<td>0005</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>0001</td>
<td>0005</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0001</td>
<td>0006</td>
<td>0</td>
</tr>
<tr>
<td>A</td>
<td>0000</td>
<td>0006</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>0000</td>
<td>0006</td>
<td>1</td>
</tr>
<tr>
<td>E</td>
<td>0006</td>
<td>0006</td>
<td>1</td>
</tr>
</tbody>
</table>
Indirect Addressing

• Assembly equivalent of dereferencing a pointer
• General format:
  offset(register)
• Examples:
  (%ecx)
  4 (%esp)
## Indirect Addressing

<table>
<thead>
<tr>
<th>Addr</th>
<th>Machine code</th>
<th>Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:</td>
<td>b8 00 80 0b 00</td>
<td>mov $0xb8000,%eax</td>
</tr>
<tr>
<td>5:</td>
<td>66 bb 34 12</td>
<td>mov $0x41,%bl</td>
</tr>
<tr>
<td>9:</td>
<td>66 89 18</td>
<td>mov %bl,(%eax)</td>
</tr>
</tbody>
</table>

Before last mov-instr.  

<table>
<thead>
<tr>
<th>Addr</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0xb8003</td>
<td>00</td>
</tr>
<tr>
<td>0xb8002</td>
<td>00</td>
</tr>
<tr>
<td>0xb8001</td>
<td>00</td>
</tr>
<tr>
<td>0xb8000</td>
<td>00</td>
</tr>
</tbody>
</table>

After last mov-instr.  

<table>
<thead>
<tr>
<th>Addr</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0xb8003</td>
<td></td>
</tr>
<tr>
<td>0xb8002</td>
<td></td>
</tr>
<tr>
<td>0xb8001</td>
<td></td>
</tr>
<tr>
<td>0xb8000</td>
<td>41</td>
</tr>
</tbody>
</table>

Equivalent to the following C-code:

```c
char* screen_base = (char *) 0xb8000;
*screen_base = ‘A’;
```
### Pushing and Popping

<table>
<thead>
<tr>
<th>Addr</th>
<th>Machine code</th>
<th>Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:</td>
<td>bc 00 00 05 00</td>
<td>mov $0x50000,%esp</td>
</tr>
<tr>
<td>5:</td>
<td>66 b8 34 12</td>
<td>mov $0x1234,%ax</td>
</tr>
<tr>
<td>9:</td>
<td>66 bb 78 56</td>
<td>mov $0x5678,%bx</td>
</tr>
<tr>
<td>d:</td>
<td>66 50</td>
<td>push %ax</td>
</tr>
<tr>
<td>f:</td>
<td>66 53</td>
<td>push %bx</td>
</tr>
<tr>
<td>11:</td>
<td>66 58</td>
<td>pop %ax</td>
</tr>
<tr>
<td>13:</td>
<td>66 5b</td>
<td>pop %bx</td>
</tr>
</tbody>
</table>

---

#### Diagram:

- **ESP**
  - Initial state: 0x50000
  - After push: 0x4FFFE
  - After pop: 0x50000

- **AX**
  - Initial state: 0x1234
  - After push: 0x5678
  - After pop: 0x1234

- **BX**
  - Initial state: 0x5678
  - After push: 0x5678
  - After pop: 0x5678

- **EIP**
  - Initial state: 0
  - After push: 0xD
  - After pop: 0x11

---

25
Subroutines

<table>
<thead>
<tr>
<th>Addr</th>
<th>Machine code</th>
<th>Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>bc 00 00 05 00</td>
<td>mov $0x50000,%esp</td>
</tr>
<tr>
<td>5</td>
<td>66 b8 34 12</td>
<td>mov $0x1234,%ax</td>
</tr>
<tr>
<td>9</td>
<td>e8 02 00 00 00</td>
<td>call L2</td>
</tr>
<tr>
<td>e</td>
<td>eb fe</td>
<td>L1: jmp L1</td>
</tr>
<tr>
<td>10</td>
<td>66 40</td>
<td>L2: inc %ax</td>
</tr>
<tr>
<td>12</td>
<td>c3</td>
<td>ret</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>EIP</th>
<th>ESP</th>
<th>AX</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x50000</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>0x50000</td>
<td>0x1234</td>
</tr>
<tr>
<td>9</td>
<td>0x4FFFFC</td>
<td>0x1234</td>
</tr>
<tr>
<td>0x10</td>
<td>0x4FFFFC</td>
<td>0x1235</td>
</tr>
<tr>
<td>0x12</td>
<td>0x50000</td>
<td>0x1235</td>
</tr>
<tr>
<td>0xE</td>
<td>0x50000</td>
<td>0x1235</td>
</tr>
</tbody>
</table>
# Subroutines

<table>
<thead>
<tr>
<th>Addr</th>
<th>Machine code</th>
<th>Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:</td>
<td><code>bc 00 00 05 00</code></td>
<td><code>mov $0x50000,%esp</code></td>
</tr>
<tr>
<td>5:</td>
<td><code>e8 06 00 00 00</code></td>
<td><code>call L2</code></td>
</tr>
<tr>
<td>a:</td>
<td><code>66 b8 34 12</code></td>
<td><code>mov $0x1234,%ax</code></td>
</tr>
<tr>
<td>e:</td>
<td><code>eb fe</code></td>
<td><code>L1: jmp L1</code></td>
</tr>
<tr>
<td>10:</td>
<td>58</td>
<td><code>L2: pop %eax</code></td>
</tr>
<tr>
<td>11:</td>
<td><code>83 c0 04</code></td>
<td><code>add $0x4,%eax</code></td>
</tr>
<tr>
<td>14:</td>
<td>50</td>
<td><code>push %eax</code></td>
</tr>
<tr>
<td>15:</td>
<td>c3</td>
<td><code>ret</code></td>
</tr>
</tbody>
</table>

Diagram:

- `0x50000`
  - ESP
  - A
  - E
- `0x4FFFC`
  - EIP
  - ESP
  - EAX
- `0x0A`
  - 0x10
  - 0x50000
- `0x0E`
  - 0x14
  - 0x50000
  - 0x50000
int add (int x, int y)  
{  
    return x + y;  
}  
void main()  
{  
    int sum = add (3, 4);  
    printf ("3 + 4 = %d\n", sum);  
}  

Compile with: gcc –fomit-frame-pointer -01 –S add.c
C and Assembly

Observations:

• C-functions are called via the x86 `call` instruction.
• The caller pushes the actual parameters onto the stack.
• The actual parameters are pushed from right to left.
• The callee accesses the parameters as offset to the current value of the %ESP register.
• After the function call returns, the caller has to clean up the stack.
• `printf()` and `main()` are treated just like any other function.
• Return values are placed in %EAX
Memory Layout

• When running a program under a modern OS, memory is divided into code, heap, stack
• This environment is set up by the OS, when writing the OS we don’t have such an environment.
• Memory layout managed manually by the OS
Stack Layout (1)

```c
void f()
{
    int x;
}

void g()
{
    int y;
    f();  /* Address 2 */
}

void main()
{
    int z;
    g();  /* Address 1 */
}
```
void f()
{
    int x;
}

void g()
{
    int y;
    f();   /* Address 2 */
}

void main()
{
    int z;
    g();    /* Address 1 */
}
void f()
{
    int x;
}

void g()
{
    int y;
    f();   /* Address 2 */
}

void main()
{
    int z;
    g();   /* Address 1 */
}
 Embedding Assembly into C

**test.c**

```c
void enable_interrupts()
{
    asm( "sti" );
}
```

**test.s**

```assembly
enable_interrupts:
    #APP
    sti
    #NO_APP
    ret
```

- Assembly instructions can be embedded anywhere C-statements are allowed.
- This is done with the `asm`-instruction (gcc-specific!)
- Application specific assembly is surrounded by `#APP` and `#NO_APP`.
- Note: `sti` – instruction for enabling the interrupts.
# Embedding Assembly in C

```c
int add (int x, int y)
{
    int sum = x;
    asm ("add %1, %0" : "=r" (sum) : "m" (y));
    return sum;
}
```

```
add:    subl  $4, %esp
        movl  8(%esp), %eax
        movl  %eax, (%esp)
#APP
        add   12(%esp), %eax
#NO_APP
        addl  $4, %esp
        ret
```
This example is mostly identical from an earlier slide entitled “EFLAGS”. Differences are that it uses 32-bit registers and is assembled for a 64-bit architecture (note that TOS is compiled for 32-bit architectures). This subroutine will compute the sum of $1 + 2 + 3 = 6$ which is stored in %eax.
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/mman.h>

void main() {
    unsigned char code[] = {
        0x31, 0xC9, 0xB8, 0x03, 0x00, 0x00, 0x00, 0x01,
        0xC1, 0xFF, 0xC8, 0x75, 0xFA, 0x89, 0xC8, 0xC3
    };
    void* mem = mmap(NULL, sizeof(code), PROT_WRITE | PROT_EXEC,
                     MAP_ANON | MAP_PRIVATE, -1, 0);
    memcpy(mem, code, sizeof(code));
    int (*func)() = mem;
    int result = func();
    printf("Result: %d\n", result);
}

• This code needs to be compiled on a Linux 64-bit x86 machine. It will print 6.

Booting TOS

• When the computer is turned on, several things must happen:
  – TOS kernel gets loaded into memory
  – Execution stack (%esp) is established
  – Kernel starts by jumping to `kernel_main()`

• For a regular program, this process is done by the OS `loader`.

• In the case of the kernel, we have no OS so this work is done by the `boot loader`. 
Booting TOS

- Due to constraints of the x86 architecture, booting TOS happens in two stages.
- Code for the two boot loader stages is in:
  - tos/tools/boot/boot.s
  - tos/tools/boot/second-stage.s
- This code is a mess, you don’t need to understand it!
- But, it is important to understand the environment that the boot loader sets up for the kernel!
Memory Layout

• The memory visible to a “regular” program is divided into code, heap, and stack

• For a “regular” program, the OS (kernel) can do things such as:
  – Prevent the program from modifying its code
  – Allow the heap to grow (i.e., as a result of a call to malloc())

• But, we are writing the kernel so we don’t have these conveniences!
TOS Memory Layout

- No heap in TOS, just code and stack
- The only usable addresses are 0-640KB
Hardware Protection

• OS needs to protect applications from each other
  – Want protection from malicious programs as well as from programs that are just buggy

• We will discuss the details of implementing protection for specific hardware resources (memory, CPU, etc.) throughout the semester.
Hardware Protection

• Regardless of the hardware, the kernel needs to run with more “privileges” than other programs.

• Modern hardware can switch between two modes with different privileges:
  – Kernel mode, in which the processor may do anything
  – User mode, in which some operations are restricted
(Lack of) Protection in TOS

• Protection is complex in the x86 architecture.
• We will not implement protection in TOS (a malicious or badly-written program can crash the OS, interfere with other programs, etc.)
• Nevertheless, we will study how protection is implemented in “real” operating systems.