A very brief introduction to x86 Architecture
The author's views expressed in this presentation do not necessarily reflect the views of IBM.
Agenda

Introduction
  x86 history
  x86 success

x86 in detail - x86 vs LC-3
  Data types
  Registers
  Instructions
  Memory addressing

A simple example
  A program doing nothing
Microprocessor

- As opposed to mainframes
- All CPU functionality on a single chip
- Started with popular home computers
  - 8-bit 6502/6510, Z80
  - 32-bit Motorola 68000 (my time)
Intel Microprocessors

- **4004**
  - 1971: First single-chip microprocessor; 740 kHz

- **8008**
  - 1972: First 8-bit microprocessor; 800 kHz

- **8080**
  - 1974: larger instruction set; 2 MHz

- **8086/88**
  - 1979: ~5MHz 29,000 transistors (today >1B)
  - Used by NASA at least until 2002 for space shuttle operations
  - IBM PC revolution

  - Manufactured by:
    - Intel
    - AMD
    - NEC
    - Fujitsu,
    - OKI,
    - Siemens
    - ...
IBM PC specs (1981):
- 8088 4.77 MHz
  - External only 8 bit
  - No FPU
- 64KB memory (256KB max)
- 1 x 5.25” floppy 160KB
- Monochrome monitor
- PC Dos 1.0

$3000
IBM PC mainboard
IBM AT specs:
- 80286 6 MHz
  - No FPU
  - true 16-bit
- 256KB memory (16MB max)
- 1 x 5.25" floppy 1.2MB
- 20MB HDD
- Monochrome monitor
Introduction

Intel 80x86 Series history

- 4004
- 8008
- 8080
- 8086
- 80286
- 80386
- 80486
- Pentium
- Pentium II
- Pentium III
- Pentium 4/D
- Core 2
- Core i7
NYT, 5/6/91:
“…Cray Research Inc., the king of supercomputing, says it is more worried by "killer micros" -- compact, extremely fast work stations that sell for less than $100,000… “
NYT, 5/6/91:
“…Cray Research Inc., the king of supercomputing, says it is more worried by "killer micros" -- compact, extremely fast work stations that sell for less than $100,000… “
Top 5 of the TOP500 supercomputers

<table>
<thead>
<tr>
<th>NAME</th>
<th>SPECS</th>
<th>SITE</th>
<th>COUNTRY</th>
</tr>
</thead>
<tbody>
<tr>
<td>TITAN</td>
<td>Cray XK7, Operon 6274 16C 2.2 GHz + Nvidia Kepler GPU, Custom interconnect</td>
<td>DOE/OS/ORNL</td>
<td>USA</td>
</tr>
<tr>
<td>SEQUOIA</td>
<td>IBM BlueGene/Q, Power BQC 16C 1.60 GHz, Custom interconnect</td>
<td>DOE/NNSA/LLNL</td>
<td>USA</td>
</tr>
<tr>
<td>K COMPUTER</td>
<td>Fujitsu SPARC64 VIIIfx 2.0GHz, Custom interconnect</td>
<td>RIKEN AICS</td>
<td>Japan</td>
</tr>
<tr>
<td>MIRA</td>
<td>IBM BlueGene/Q, Power BQC 16C 1.60 GHz, Custom interconnect</td>
<td>DOE/OS/ANL</td>
<td>USA</td>
</tr>
<tr>
<td>JuQUEEN</td>
<td>IBM BlueGene/Q, Power BQC 16C 1.60 GHz, Custom interconnect</td>
<td>Forschungszentrum Jülich</td>
<td>Germany</td>
</tr>
</tbody>
</table>

Mostly non-x86 architectures:
- Nvidia GPU’s, with AMD x86 though
- IBM Blue Gene/Q
- Fujitsu SPARC
x86 is the dominant architecture today

Market Share of Top 500 systems
x86 is the dominant architecture today

Market Share of Top 500 systems

- IDC report 5/2012 states: “The market for non-x86 servers declined 16.1% … Non-x86 based systems now comprise 28.5% of the server market, the lowest level ever reported in IDC's quarterly server tracker
CISC vs. RISC

- **Complex Instruction Set Computer vs. Reduced Instruction Set Computer**

- **CISC**:
  - Eased programming effort in the early days
  - Many, possibly complex (larger) instructions
  - Larger instructions take more time to decode and execute
  - Today, complex instructions often implemented as microcode
CISC vs. RISC

- **Complex Instruction Set Computer** vs. **Reduced Instruction Set Computer**

- **CISC**:
  - Eased programming effort in the early days
  - Many, possibly complex (larger) instructions
  - Larger instructions take more time to decode and execute
  - Today, complex instructions often implemented as microcode

- **RISC**:
  - Small, simple instructions
  - Popular in 90’s workstations
  - Require more instructions
  - Can run at higher clock speed due to simplicity
  - Reflected in microcode approach of new x86 CISC chips
CISC vs. RISC

- **Complex Instruction Set Computer vs. Reduced Instruction Set Computer**

- **CISC:**
  - Eased programming effort in the early days
  - Many, possibly complex (larger) instructions
  - Larger instructions take more time to decode and execute
  - Today, complex instructions often implemented as microcode

- **RISC:**
  - Small, simple instructions
  - Popular in 90’s workstations
  - Require more instructions
  - Can run at higher clock speed due to simplicity
  - Reflected in microcode approach of new x86 CISC chips

- Key distinction: load-and-operate = combined memory access and computation, i.e., instructions can operate directly on memory

- Examples:
  - CISC: Intel, AMD
  - RISC: LC-3
Agenda

Introduction
  x86 history
  x86 success

x86 in detail - x86 vs LC-3
  Data types
  Registers
  Instructions
  Memory addressing

A simple example
  A program doing nothing
## X86 ISA - a closer look

<table>
<thead>
<tr>
<th>Feature</th>
<th>X86</th>
<th>vs.</th>
<th>LC-3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address space:</td>
<td>$2^{32}$</td>
<td>vs.</td>
<td>$2^{16}$</td>
</tr>
<tr>
<td>Addressability:</td>
<td>8</td>
<td>vs.</td>
<td>16 bits</td>
</tr>
<tr>
<td>Registers:</td>
<td>8x 32</td>
<td>vs.</td>
<td>8x 16 bits</td>
</tr>
<tr>
<td>Opcodes:</td>
<td>hundreds</td>
<td>vs.</td>
<td>15</td>
</tr>
<tr>
<td>Datatypes:</td>
<td>many</td>
<td>vs.</td>
<td>2’s complement 16-bit int</td>
</tr>
<tr>
<td><strong>X86</strong></td>
<td><strong>vs.</strong></td>
<td><strong>LC-3</strong></td>
<td></td>
</tr>
<tr>
<td>----------</td>
<td>---------</td>
<td>----------</td>
<td></td>
</tr>
<tr>
<td><strong>Address space:</strong></td>
<td>$2^{32}$</td>
<td>vs.</td>
<td>$2^{16}$</td>
</tr>
<tr>
<td>– 2$^{64}$ in 64-bit mode, x86-64</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>– only 48 bit are currently implemented</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Addressability:</strong></td>
<td>8</td>
<td>vs.</td>
<td>16 bits</td>
</tr>
<tr>
<td>– 8,16,32,64,128,256 bits (aligned!)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Registers:</strong></td>
<td>8x 32</td>
<td>vs.</td>
<td>8x 16 bits</td>
</tr>
<tr>
<td>– additional 8x 64-bit registers in 64-bit mode</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>– 8087: 8x 80-bit FP registers,</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>– 8 64-bit MMX (re-purposed FP),</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>– 8 128-bit SSE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>– 16 256-bit SSE registers</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Opcodes:</strong></td>
<td>hundreds vs.</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>– More than 1k instructions</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Datatypes:</strong></td>
<td>many</td>
<td>vs.</td>
<td>2’s complement 16-bit int</td>
</tr>
<tr>
<td>– 8,16,32,64 bit int, signed &amp; unsigned</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>– 32 &amp; 64-bit floating point</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>– Binary coded decimal (historic)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>– 64,128,256bit vectors of integers/floats</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Data types

- LC-3 only supports 16-bit two's complement integers (signed)
- x86 has 8-, 16-, 32-, and 64-bit integers:
  - char, short, int, long long int
  - Implicitly signed
  - explicitly unsigned to gain an extra bit ...
Data types

- LC-3 only supports 16-bit two's complement integers (signed)
- x86 has 8-, 16-, 32-, and 64-bit integers:
  - char, short, int, long long int
  - Implicitly signed
  - explicitly unsigned to gain an extra bit …
- 32- and 64-bit floating point numbers
  - float and double
- For historical reasons: Binary Coded Decimal (BCD):
  - decimal digits represented in chunks of 4 bits (packed) or 8 bits (unpacked)
  - small range of numbers, 0-9999 (packed) or 0-99 (unpacked)
  - Limited arithmetic:
    • Add, sub, mul, div
    • Operands < 99 for multiply and <9999 other operations
- Vector data types
  - 128 bit floating point and integer vectors
  - 256 bit floating point vectors
  - This is where the x86 ISA really evolves (more later)
x86 Registers

- Overview – 16-bit integer registers
  - “General” purpose (with exceptions): AX, BX, CX, DX
  - Pointer registers: SP (Stack pointer), BP (Base Pointer)
  - For array indexing: DI, SI
  - Segment registers: CS, DS, SS, ES (legacy)
  - FLAGS register to store flags, e.g. CF, OF, ZF
  - Instruction Pointer: IP
x86 Registers

- Overview – 16-bit integer registers
  - “General” purpose (with exceptions): AX, BX, CX, DX
  - Pointer registers: SP (Stack pointer), BP (Base Pointer)
  - For array indexing: DI, SI
  - Segment registers: CS, DS, SS, ES (legacy)
  - FLAGS register to store flags, e.g. CF, OF, ZF
  - Instruction Pointer: IP

- Larger Registers (64 & 32bit) comprise the smaller ones lower half
  - 32 bit prefixed with “e”, 64 bit with “r”

<table>
<thead>
<tr>
<th>RAX</th>
<th>EAX</th>
<th>AX</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>63</td>
<td>31</td>
<td>15</td>
</tr>
<tr>
<td>31</td>
<td>15</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>15</td>
<td>0 bit</td>
</tr>
</tbody>
</table>
x86 Registers

- Overview – 16-bit integer registers
  - “General” purpose (with exceptions): AX, BX, CX, DX
  - Pointer registers: SP (Stack pointer), BP (Base Pointer)
  - For array indexing: DI, SI
  - Segment registers: CS, DS, SS, ES (legacy)
  - FLAGS register to store flags, e.g. CF, OF, ZF
  - Instruction Pointer: IP

- Larger Registers (64 & 32bit) comprise the smaller ones lower half
  - 32 bit prefixed with “e”, 64 bit with “r”

<table>
<thead>
<tr>
<th>RAX</th>
<th>EAX</th>
<th>AX</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>63</td>
<td>31</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

- Careful when you work with multiple register sizes:
  - Don’t overwrite results!
  - Postfix assembly instruction with word size

```assembly
movw (%ecx),%ax
movzwl %ax,%ebx
```
x86 Registers (contd.)

- Floating point registers
  - Registers ST(0) – ST(7) organized as a stack
  - By default operates on top of the stack

```
faddp ; ST(0)+ST(1) -> ST(1), POP
fadd %st, %st(5) ; ST(0)+ST(5) -> ST(5)
fadd (%ecx) ; ST(0)+MEM -> ST(0)
```

- Extended 80-bit precision by default
- Can switch FPU between single(32bit), double(64) and extended(80) precision (FLDCW)

- MMX Registers (legacy)
  - 64-bit shared Shared with FPU
x86 Registers (contd.)

- Floating point registers
  - Registers ST(0) – ST(7) organized as a stack
  - By default operates on top of the stack

```
faddp ; ST(0)+ST(1) -> ST(1), POP
fadd %st, %st(5) ; ST(0)+ST(5) -> ST(5)
fadd (%ecx) ; ST(0)+MEM -> ST(0)
```

- Extended 80-bit precision by default
- Can switch FPU between single(32bit), double(64) and extended(80) precision (FLDCW)

- MMX Registers (legacy)
  - 64-bit shared Shared with FPU

- SSE registers
  - Registers XMM0-XMM7 operate on 128-bit vectors
  - Does not need to be loaded from aligned address (slower)

```c
movdqu (%eax),%xmm0
movdqa (%ebx),%xmm1
```
x86 Registers (contd.)

- **Floating point registers**
  - Registers ST(0) – ST(7) organized as a stack
  - By default operates on top of the stack
    
    \[
    \text{faddp} \quad ; \text{ST(0)+ST(1) -> ST(1), POP} \\
    \text{fadd \%st, \%st(5)} \quad ; \text{ST(0)+ST(5) -> ST(5)} \\
    \text{fadd \(%ecx\)} \quad ; \text{ST(0)+MEM -> ST(0)}
    \]
  - Extended 80-bit precision by default
  - Can switch FPU between single(32bit), double(64) and extended(80) precision (FLDCW)

- **MMX Registers (legacy)**
  - 64-bit shared Shared with FPU

- **SSE registers**
  - Registers XMM0-XMM7 operate on 128-bit vectors
  - Does not need to be loaded from aligned address (slower)
    
    \[
    \text{movdqu \(%eax\), \%xmm0} \\
    \text{movdqa \(%ebx\), \%xmm1}
    \]
  - Can address content as any (vector) data type:
    - 8,16,32,64 bit integer or 32,64 bit floating point numbers
      
      \[
      \text{paddb \%xmm0, \%xmm1} \\
      \text{paddw \%xmm0, \%xmm1} \\
      \text{paddd \%xmm0, \%xmm1} \\
      \text{paddq \%xmm0, \%xmm1}
      \]
x86 Registers (contd.)

- Floating point registers
  - Registers ST(0) – ST(7) organized as a stack
  - By default operates on top of the stack

\[
\begin{align*}
faddp & \quad ; \ ST(0) + ST(1) \rightarrow ST(1), \ POP \\
fadd \ %st, \ %st(5) & \quad ; \ ST(0) + ST(5) \rightarrow ST(5) \\
fadd \ (%ecx) & \quad ; \ ST(0) + \text{MEM} \rightarrow ST(0)
\end{align*}
\]

- Extended 80-bit precision by default
- Can switch FPU between single(32bit), double(64) and extended(80) precision (FLDCW)

- MMX Registers (legacy)
  - 64-bit shared Shared with FPU

- SSE registers
  - Registers XMM0-XMM7 operate on 128-bit vectors
  - Does not need to be loaded from aligned address (slower)

\[
\begin{align*}
\text{movdqu} \ (%eax),\ %xmm0 \\
\text{movdqa} \ (%ebx),\ %xmm1
\end{align*}
\]

- Can address content as any (vector) data type:
  - 8,16,32,64 bit integer or 32,64 bit floating point numbers

\[
\begin{align*}
paddb \ %xmm0,\ %xmm1 \\
paddw \ %xmm0,\ %xmm1 \\
paddd \ %xmm0,\ %xmm1 \\
paddq \ %xmm0,\ %xmm1
\end{align*}
\]

Where does the result go?
x86’s 2 operand instructions: \( a + b = b \)

- LC-3 instructions require
  - 2 input operands
  - 1 output operand

- X86 instructions have only 2 operands
  - 1 input
  - 1 input & output
  - 2nd input gets overwritten
  - You need to copy it elsewhere, if you want to keep it

```assembly
add %eax, %ebx ; b += a
```
x86’s 2 operand instructions: \( a \times b = a \)

- LC-3 instructions require
  - 2 input operands
  - 1 output operand

- X86 instructions have only 2 operands
  - 1 input
  - 1 input & output
  - 2nd input gets overwritten
  - You need to copy it elsewhere, if you want to keep it

```plaintext
add %eax, %ebx ; b += a
```

- Some x86 instructions only operate on specific registers, e.g. multiply
  - First operand has to be register AL, AX, EAX, RAX
  - 2nd operand any register or memory
  - Result stored in AX, AX:DX, EAX:EDX, RAX:RDX

```plaintext
mul %bl ; ax = al \times bl
mul %bx ; dx:ax = ax \times bx
```
x86’s 2 operand instructions: \( a \times b = a \)

- LC-3 instructions require
  - 2 input operands
  - 1 output operand

- X86 instructions have only 2 operands
  - 1 input
  - 1 input & output
  - 2nd input gets overwritten
  - You need to copy it elsewhere, if you want to keep it

```
add %eax, %ebx ; b += a
```

- Some x86 instructions only operate on specific registers, e.g. multiply
  - First operand has to be register AL, AX, EAX, RAX
  - 2nd operand any register or memory
  - Result stored in AX, AX:DX, EAX:EDX, RAX:RDX

```
mul %bl ; ax = al * bl
mul %bx ; dx:ax = ax * bx
```

- Why 2 registers for some results?
  - What happens if you multiply two 32-bit numbers?
# MUL—Unsigned Multiply

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F6 /4</td>
<td>MUL r/m8</td>
<td>Valid</td>
<td>Valid</td>
<td>Unsigned multiply (AX ← AL * r/m8).</td>
</tr>
<tr>
<td>REX + F6 /4</td>
<td>MUL r/m8*</td>
<td>Valid</td>
<td>N.E.</td>
<td>Unsigned multiply (AX ← AL * r/m8).</td>
</tr>
<tr>
<td>F7 /4</td>
<td>MUL r/m16</td>
<td>Valid</td>
<td>Valid</td>
<td>Unsigned multiply (DX:AX ← AX * r/m16).</td>
</tr>
<tr>
<td>F7 /4</td>
<td>MUL r/m32</td>
<td>Valid</td>
<td>Valid</td>
<td>Unsigned multiply (EDX:EAX ← EAX * r/m32).</td>
</tr>
<tr>
<td>REX.W + F7 /4</td>
<td>MUL r/m64</td>
<td>Valid</td>
<td>N.E.</td>
<td>Unsigned multiply (RDX:RAX ← RAX * r/m64).</td>
</tr>
</tbody>
</table>

**NOTES:**

* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.
Instructions

- LC-3 instructions are fixed length - 16 bits
- X86 instructions range from 1 to 15 bytes

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Opcode</th>
<th>Mod</th>
<th>Reg</th>
<th>R/M</th>
<th>Scale</th>
<th>Index</th>
<th>Base</th>
<th>Disp.</th>
<th>Imm.</th>
</tr>
</thead>
</table>

Prefix (0-4 Byte)
- Modify the interpretation of the following instruction
- Lock/Repeat
- Segment override, e.g. 0x26: ES segment override – protected (32/64 bit) mode
- Operand size override
- Address size override

\[\text{e.g. REX prefix to operate in 64 bit mode}\]
Instructions

- LC-3 instructions are fixed length - 16 bits
- X86 instructions range from 1 to 15 bytes

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Opcode</th>
<th>Mod</th>
<th>Reg</th>
<th>R/M</th>
<th>Scale</th>
<th>Index</th>
<th>Base</th>
<th>Disp.</th>
<th>Imm.</th>
</tr>
</thead>
</table>

- Prefix (0-4 Byte)
- Opcode (1-2 Bytes)
  - Intel has hundreds of opcodes, LC-3 has 15
  - Some opcodes borrow 3 bits from the ModR/M byte, i.e. first operand
  - Did I mention that x86 is messy?
Instructions

- LC-3 instructions are fixed length - 16 bits
- X86 instructions range from 1 to 15 bytes

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Opcode</th>
<th>Mod</th>
<th>Reg</th>
<th>R/M</th>
<th>Scale</th>
<th>Index</th>
<th>Base</th>
<th>Disp.</th>
<th>Imm.</th>
</tr>
</thead>
</table>

- Prefix (0-4 Byte)
- Opcode (1-2 Bytes)
- Mod R/M (1 Byte)
  - Bits [7:6] define the addressing mode
    • 00 – direct: 2nd operand (register) contains a memory address
    • 01 – displacement: adding displacement to 2nd operand yields memory address
    • 10 – use SIB byte
    • 11 – 2nd operand is a register
  - Bits [5:3] first operand, 3-bit register number, e.g. 000 = EAX;
  - Bits [2:0] 2nd operand
Instructions

- LC-3 instructions are fixed length - 16 bits
- X86 instructions range from 1 to 15 bytes

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Opcode</th>
<th>Mod</th>
<th>Reg</th>
<th>R/M</th>
<th>Scale</th>
<th>Index</th>
<th>Base</th>
<th>Disp.</th>
<th>Imm.</th>
</tr>
</thead>
</table>

- Prefix (0-4 Byte)
- Opcode (1-2 Bytes)
- Mod R/M (1 Byte)
- Scale Index Base (1 Byte)
  - Address = Mod R/M + scale * index + base
  - Scale: Bits [7:6] are a multiplier 00,01,10 (1x,2x,4x)
  - Index: Bits [5:3] refer to a register number, e.g. 000 = EAX
  - Base: Bits [2:0] refer to a register number
Instructions

- LC-3 instructions are fixed length - 16 bits
- X86 instructions range from 1 to 15 bytes

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Opcode</th>
<th>Mod</th>
<th>Reg</th>
<th>R/M</th>
<th>Scale</th>
<th>Index</th>
<th>Base</th>
<th>Disp.</th>
<th>Imm.</th>
</tr>
</thead>
</table>

- Prefix (0-4 Byte)
- Opcode (1-2 Bytes)
- Mod R/M (1 Byte)
- Scale Index Base (1 Byte)
- Displacement (0-4 Bytes)
  - Depends on addressing mode (Mod)
Instructions

- LC-3 instructions are fixed length - 16 bits
- X86 instructions range from 1 to 15 bytes

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Opcode</th>
<th>Mod</th>
<th>Reg</th>
<th>R/M</th>
<th>Scale</th>
<th>Index</th>
<th>Base</th>
<th>Disp.</th>
<th>Imm.</th>
</tr>
</thead>
</table>

- Prefix (0-4 Byte)
- Opcode (1-2 Bytes)
- Mod R/M (1 Byte)
- Scale Index Base (1 Byte)
- Displacement (0-4 Bytes)
- Immediate (0-4 Bytes)
  - Length specified in opcode
  - Sign extended to size of other operand
Memory LC-3 has 16-bit address space and 16-bit addressability
  – \(2^{16} = 65536\) 16-bit addresses
  – can address 128KB of memory

X86 has 8-bit addressability
  – For larger word size (most), address only specifies the low-order byte
  – The word size depends on the context (mode/instruction)
  – Addresses must align with word size
  – Addressable memory?
Memory mess – addressing and addressability

- Memory LC-3 has 16-bit address space and 16-bit addressability
  - $2^{16} = 65536$ 16-bit addresses
  - can address 128KB of memory

- X86 has 8-bit addressability
  - For larger word size (most), address only specifies the low-order byte
  - The word size depends on the context (mode/instruction)
  - Addresses must align with word size
  - Addressable memory:
    - 8086 20-bit address space (1MB)
    - 80286 24-bit address space (16MB)
    - 80386 32-bit address space (4GB)
    - AMD Athlon/ Intel Core 2 64-bit address space, 48 bit implemented (256TB)
Memory LC-3 has 16-bit address space and 16-bit addressability
- $2^{16} = 65536$ 16-bit addresses
- can address 128KB of memory

X86 has 8-bit addressability
- For larger word size (most), address only specifies the low-order byte
- The word size depends on the context (mode/instruction)
- Addresses must align with word size
- Addressable memory:
  - 8086 20-bit address space (1MB)
  - 80286 24-bit address space (16MB)
  - 80386 32-bit address space (4GB)
  - AMD Athlon/ Intel Core 2 64-bit address space, 48 bit implemented (256TB)

The 8086 was a 16-bit CPU … How could you have $2^{20}$ addresses?
Memory LC-3 has 16-bit address space and 16-bit addressability
- \(2^{16} = 65536\) 16-bit addresses
- can address 128KB of memory

X86 has 8-bit addressability
- For larger word size (most), address only specifies the low-order byte
- The word size depends on the context (mode/instruction)
- Addresses must align with word size
- Addressable memory:
  - 8086 20-bit address space (1MB)
  - 80286 24-bit address space (16MB)
  - 80386 32-bit address space (4GB)
  - AMD Athlon/ Intel Core 2 64-bit address space, 48 bit implemented (256TB)

The 8086 was a 16-bit CPU … How could you have \(2^{20}\) addresses?
- Use 2 registers to describe a memory address:
  - segment register \(<< 4\) + address register = memory address
  - Goof: 1 memory address could be reached by \(2^{12}\) (4K) segment/offset combinations
Memory mess – addressing and addressability

- Memory LC-3 has 16-bit address space and 16-bit addressability
  - $2^{16} = 65536$ 16-bit addresses
  - can address 128KB of memory

- X86 has 8-bit addressability
  - For larger word size (most), address only specifies the low-order byte
  - The word size depends on the context (mode/instruction)
  - Addresses must align with word size
  - Addressable memory:
    - 8086 20-bit address space (1MB)
    - 80286 24-bit address space (16MB)
    - 80386 32-bit address space (4GB)
    - AMD Athlon/ Intel Core 2 64-bit address space, 48 bit implemented (256TB)

- The 8086 was a 16-bit CPU … How could you have $2^{20}$ addresses?
  - Use 2 registers to describe a memory address:
    - segment register $\ll 4 +$ address register = memory address
    - Goof: 1 memory address could be reached by $2^{12}$ (4K) segment/offset combinations

- With the 32-bit 80386 came a flat 32-bit memory structure
  - 32-bit register $\rightarrow$ 32-bit address?
Memory mess – application perspective

- Important for addressing are ESP and EBP
  - Memory addressing in a program/function is relative to the base pointer EBP
  - Free memory space is indicated by the stack pointer ESP
  - Stack grows negative relatively to base pointer
Memory mess – application perspective

- Important for addressing are ESP and EBP
  - Memory addressing in a program/function is relative to the base pointer EBP
  - Free memory space is indicated by the stack pointer ESP
  - Stack grows negative relatively to base pointer

- X86 program/function calling conventions
  - Before calling a function
    - Save function parameters to the stack

```
| -28 | -24 | -20 | -16 | -12 | -8  | -4  | 0   |

ESP  param 2
EBP  param 1
```
Memory mess – application perspective

- Important for addressing are ESP and EBP
  - Memory addressing in a program/function is relative to the base pointer EBP
  - Free memory space is indicated by the stack pointer ESP
  - Stack grows negative relatively to base pointer

- X86 program/function calling conventions
  - Before calling a function
    - Save function parameters to the stack
    - CALL saves return address to the stack and sets IP to a specified address
Memory mess – application perspective

- Important for addressing are ESP and EBP
  - Memory addressing in a program/function is relative to the base pointer EBP
  - Free memory space is indicated by the stack pointer ESP
  - Stack grows negative relatively to base pointer

- X86 program/function calling conventions
  - Before calling a function
    - Save function parameters to the stack
    - CALL saves return address to the stack and sets IP to a specified address
  - At the beginning of a program/function:
    - EBP is saved and set to ESP
Memory mess – application perspective

- Important for addressing are ESP and EBP
  - Memory addressing in a program/function is relative to the base pointer EBP
  - Free memory space is indicated by the stack pointer ESP
  - Stack grows negative relatively to base pointer

- X86 program/function calling conventions
  - Before calling a function
    • Save function parameters to the stack
    • CALL saves return address to the stack and sets IP to a specified address
  - At the beginning of a program/function:
    • EBP is saved and set to ESP
Memory mess – application perspective

- Important for addressing are ESP and EBP
  - Memory addressing in a program/function is relative to the base pointer EBP
  - Free memory space is indicated by the stack pointer ESP
  - Stack grows negative relatively to base pointer

- X86 program/function calling conventions
  - Before calling a function
    - Save function parameters to the stack
    - CALL saves return address to the stack and sets IP to a specified address
  - At the beginning of a program/function:
    - EBP is saved and set to ESP
    - Registers EDI, ESI, EBX are “callee saved”, i.e. need to be saved (to the stack) if our program/function uses them
Memory mess – application perspective

- Important for addressing are ESP and EBP
  - Memory addressing in a program/function is relative to the base pointer EBP
  - Free memory space is indicated by the stack pointer ESP
  - Stack grows negative relatively to base pointer

- X86 program/function calling conventions
  - Before calling a function
    - Save function parameters to the stack
    - CALL saves return address to the stack and sets IP to a specified address
  - At the beginning of a program/function:
    - EBP is saved and set to ESP
    - Registers EDI, ESI, EBX are "callee saved", i.e. need to be saved (to the stack) if our program/function uses them
    - ESP is set to top of stack (incl. local variables)
Memory mess – application perspective

- Important for addressing are ESP and EBP
  - Memory addressing in a program/function is relative to the base pointer EBP
  - Free memory space is indicated by the stack pointer ESP
  - Stack grows negative relatively to base pointer

- X86 program/function calling conventions
  - Before calling a function
    - Save function parameters to the stack
    - CALL saves return address to the stack and sets IP to a specified address
  - At the beginning of a program/function:
    - EBP is saved and set to ESP
    - Registers EDI, ESI, EBX are “callee saved”, i.e. need to be saved (to the stack) if our program/function uses them
    - ESP is set to top of stack (incl. local variables)
  - At the end of a program/function
Memory mess – application perspective

- Important for addressing are ESP and EBP
  - Memory addressing in a program/function is relative to the base pointer EBP
  - Free memory space is indicated by the stack pointer ESP
  - Stack grows negative relatively to base pointer

- X86 program/function calling conventions
  - Before calling a function
    - Save function parameters to the stack
    - CALL saves return address to the stack and sets IP to a specified address
  - At the beginning of a program/function:
    - EBP is saved and set to ESP
    - Registers EDI, ESI, EBX are “callee saved”, i.e. need to be saved (to the stack) if our program/function uses them
    - ESP is set to top of stack (incl. local variables)
  - At the end of a program/function
    - All registers are restored
Memory mess – application perspective

- Important for addressing are ESP and EBP
  - Memory addressing in a program/function is relative to the base pointer EBP
  - Free memory space is indicated by the stack pointer ESP
  - Stack grows negative relatively to base pointer

- X86 program/function calling conventions
  - Before calling a function
    - Save function parameters to the stack
    - CALL saves return address to the stack and sets IP to a specified address
  - At the beginning of a program/function:
    - EBP is saved and set to ESP
    - Registers EDI, ESI, EBX are “callee saved”, i.e. need to be saved (to the stack) if our program/function uses them
    - ESP is set to top of stack (incl. local variables)
  - At the end of a program/function
    - All registers are restored
    - LEAVE instruction clears the stack, i.e. sets ESP to EBP and pops EBP
Memory mess – application perspective

- Important for addressing are ESP and EBP
  - Memory addressing in a program/function is relative to the base pointer EBP
  - Free memory space is indicated by the stack pointer ESP
  - Stack grows negative relatively to base pointer

- X86 program/function calling conventions
  - Before calling a function
    - Save function parameters to the stack
    - CALL saves return address to the stack and sets IP to a specified address
  - At the beginning of a program/function:
    - EBP is saved and set to ESP
    - Registers EDI, ESI, EBX are “callee saved”, i.e. need to be saved (to the stack) if our program/function uses them
    - ESP is set to top of stack (incl. local variables)
  - At the end of a program/function
    - All registers are restored
    - LEAVE instruction clears the stack, i.e. sets ESP to EBP and pops EBP
Memory mess – application perspective

- Important for addressing are ESP and EBP
  - Memory addressing in a program/function is relative to the base pointer EBP
  - Free memory space is indicated by the stack pointer ESP
  - Stack grows negative relatively to base pointer

- X86 program/function calling conventions
  - Before calling a function
    - Save function parameters to the stack
    - CALL saves return address to the stack and sets IP to a specified address
  - At the beginning of a program/function:
    - EBP is saved and set to ESP
    - Registers EDI, ESI, EBX are "callee saved", i.e. need to be saved (to the stack) if our program/function uses them
    - ESP is set to top of stack (incl. local variables)
  - At the end of a program/function
    - All registers are restored
    - LEAVE instruction clears the stack, i.e. sets ESP to EBP and pops EBP
    - RET instruction restores state in calling function …
Agenda

**Introduction**
- x86 history
- x86 success

**x86 in detail - x86 vs LC-3**
- Data types
- Registers
- Instructions
- Memory addressing

**A simple example**
- A program doing nothing
A simple program doing nothing

```c
int main()
{
    int nothing = 0;
    return nothing;
}
```
A simple program doing nothing

```c
int main(){
    int nothing = 0;
    return nothing;
}
```

- The assembly should look something like that:

```assembly
pushl %ebp            ; save BP
movl %esp, %ebp      ; set BP to SP
subl $4, %esp        ; make space for local var "nothing"
movl $0, -4(%ebp)    ; store local var "nothing" on stack
movl -4(%ebp), %eax  ; return "nothing" in EAX
addl $4, %esp        ; clear stack
popl %ebp            ; restore BP
ret
```
The assembly should look something like that:

```assembly
pushl %ebp          ; save BP
movl %esp, %ebp     ; set BP to SP
subl $4, %esp       ; make space for local var "nothing"
movl $0, -4(%ebp)   ; store local var "nothing" on stack
movl -4(%ebp), %eax ; return "nothing" in EAX
addl $4, %esp       ; clear stack
popl %ebp           ; restore BP
ret
```

In reality it's a (tiny) bit more than that
– Let's take a look at the assembly gcc really generates
– As with all live demos ... no guarantees that it will work ;-)
Confused?
“The x86 isn't all that complex... it just doesn't make a lot of sense”

Mike Johnson, AMD, 1994