Agenda
- Learn x86 architecture
  - Core 32-bit part only
    - Ignore crufty, backward-compatible things
  - Target language for D
- After we've reviewed the x86 we'll look at how to map language constructs to code

Why do we need to know assembly language?
- What compilers generate
- What your compiler will generate
- Helps understand how different language features are implemented and why they are efficient/inefficient
- ex. Why do people sometimes say that calling a function is “expensive” or that recursion is “inefficient”?

gcc and assembly
- gcc –S hello.c
  - Generates hello.s, a text file containing assembly instructions
- gcc –o hello.exe hello.s
  - Creates an executable from hello.s
- Similarly, we will be compiling the output of your compiler with gcc (combined with a bootstrap C file I will give you) to create an executable.

x86 Selected History

<table>
<thead>
<tr>
<th>Processor</th>
<th>Intro Year</th>
<th>Intro Clock</th>
<th>Transistors</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>8086</td>
<td>1978</td>
<td>8 MHz</td>
<td>29 K</td>
<td>16-bit regs., segments</td>
</tr>
<tr>
<td>286</td>
<td>1982</td>
<td>12.5 MHz</td>
<td>134 K</td>
<td>Protected mode</td>
</tr>
<tr>
<td>386</td>
<td>1985</td>
<td>20 MHz</td>
<td>275 K</td>
<td>32-bit regs., paging</td>
</tr>
<tr>
<td>486</td>
<td>1989</td>
<td>25 MHz</td>
<td>1.2 M</td>
<td>On-board FPU</td>
</tr>
<tr>
<td>Pentium</td>
<td>1993</td>
<td>60 MHz</td>
<td>3.1 M</td>
<td>MMX on late models</td>
</tr>
<tr>
<td>Pentium Pro</td>
<td>1995</td>
<td>200 MHz</td>
<td>5.5 M</td>
<td>P6 core, bigger caches</td>
</tr>
<tr>
<td>Pentium II</td>
<td>1997</td>
<td>266 MHz</td>
<td>7 M</td>
<td>P6 w/MMX</td>
</tr>
<tr>
<td>Pentium III</td>
<td>1999</td>
<td>700 MHz</td>
<td>28 M</td>
<td>SSE (Streaming SIMD)</td>
</tr>
<tr>
<td>Pentium 4</td>
<td>2000</td>
<td>1.5 GHz</td>
<td>42 M</td>
<td>NetBurst core, SSE2</td>
</tr>
<tr>
<td>Xeon</td>
<td>2001</td>
<td>3.2 GHz</td>
<td>55 M</td>
<td>Hyper-Threading</td>
</tr>
<tr>
<td>Pentium M</td>
<td>2003</td>
<td>1.6 GHz</td>
<td>77 M</td>
<td>Shorter pipelines vs P4</td>
</tr>
</tbody>
</table>

And It’s Backward-Compatible!
- Pentium/Xeon processors will run code written for the 8086(!)
- ∴ Much of the Intel descriptions of the architecture are loaded down with modes and flags that obscure the modern, fairly simple 32-bit processor model
- Links to the Intel manuals on the course web
- These slides try to cover the core x86 32-bit instructions
Assembler source formats

- D compiler project output will be an assembly language source program
- We will let the assembler handle the translation to binary encodings, address resolutions, etc.
- Examples here use Intel/Microsoft MASM format
- MASM is an assembler included in Visual Studio.NET
- For our project we will use AT &T/GNU assembler format
- Slightly different syntax, but instructions are the same. Differences are noted in the "x86 Overview" and "Code Generation for D" web pages.

Statements

- Format is:
  
  optLabel: opcode operands ; comment

  - optLabel is an optional label
  - opcode and operands make up the assembly language instruction
  - Anything following a ';' is a comment
- Language is very free-form
- Comments and labels may appear on separate lines by themselves (we'll take advantage of this)

x86 Memory Model

- 8-bit bytes, byte addressable
- 16-, 32-, 64-bit words
  
  (32 bit = doublewords), (64 bit = quadwords)
- Usually data should be aligned on "natural" boundaries; huge performance penalty on modern processors if it isn't
- Little-endian – address of a 4-byte integer is address of low-order byte

Processor Registers

- Eight 32-bit, mostly general purpose registers
  
  eax, ebx, ecx, edx, esi, edi
  
  ebp (base pointer), esp (stack pointer)
- Other registers:
  
  - Not directly addressable, you will not use these.
  - 32-bit eflags register
    
    - Holds condition codes, processor state, etc.
  - 32-bit "instruction pointer" eip
    
    - Holds address of first byte of next instruction to execute

Registers in x86

Processor Fetch-Execute Cycle

- Basic cycle
  
  while (running) {
    fetch instruction beginning at eip address
    eip <- eip + instruction length
    execute instruction
  }
- Execution continues sequentially unless a jump is executed, which stores a new "next instruction" address in eip
Instruction Format

- Typical data manipulation instruction
  - opcode dst, src
- Meaning is
  - dst <- dst op src

(*Note in GNU as the order is reversed: opcode src, dst)

Instruction Operands

- Normally:
  - one operand is a register
  - the other is a register, memory location, or integer constant
- In particular, we can’t have both operands be memory locations – not enough bits to encode this

x86 Memory Stack

- Register esp points to the "top" of stack
  - Dedicated for this use; don’t use otherwise
  - Points to the last 32-bit doubleword pushed onto the stack (not to the "next available location")
- Should always be doubleword aligned
  - Access in 4-byte increments, or
  - Use pop and push
  - Stack grows towards lower memory addresses (i.e. towards zero)

Stack Instructions

- push src
  - esp <- esp – 4;
  - memory[esp] <- src
  - (e.g., push src onto the stack)
- pop dst
  - dst <- memory[esp];
  - esp <- esp + 4
  - (e.g., pop top of stack into dst and logically remove it from the stack)

Stack Frames

- When a method is called, a stack frame is traditionally allocated on the top of the stack to hold its local variables
- Frame is popped on method return
- By convention, ebp (base pointer) points to a known offset into the stack frame
  - Local variables and parameters referenced relative to ebp

Operand Address Modes

- These should cover most of what we’ll need:
  - mov eax, 17    ; store 17 in eax
  - mov eax, ecx  ; copy ecx to eax
  - mov eax, [ebp-12] ; copy memory to eax
  - mov [ebp+8], eax ; copy eax to memory
Addressing Memory
(ignorable details)

Memory address can be specified by adding together up to:
- 2 registers, and
- 1 32-bit signed constant
- One register can be optionally pre-multiplied by 2, 4, 8.

- mov eax, ebx
- mov eax, [ebx]
- mov [var], ebx
- mov eax, [esi - 4]
- mov [esi+eax], cl
- mov edx, [esi+4*ebx]

Incorrect: (why?)
- mov eax, [ebx – ecx]
- mov [eax+esi+edi], ebx
- mov [4*eax+2*ebx], ecx

Operand Address Modes
(ignorable details)

- In full generality, a memory address can combine the contents of two registers (with one being scaled) plus a constant displacement:
  \[ \text{base reg} + \text{index} \times \text{scale} + \text{constant} \]
- Scale can be 2, 4, 8
- Main use is for array subcripting
- Example: suppose
  - Array of 4-byte ints
  - Address of the array A is in ecx
  - Subscript i is in eax
  - Code to store the value 17 in A[i] would be:

Load Effective Address
(ignorable details)

- The unary & operator in C
  - lea dst, src ; dst <- address of src
- dst must be a register
- Address of src includes any address arithmetic or indexing
- Useful to capture addresses for pointers, reference parameters, etc.

Example

- mov ecx, eax
- mov edx, [ebx]
- mov esi, [edx+eax+4]
- mov [esi], 45
- mov [a], 15
- lea edi, [a]

Basic Data Movement and Arithmetic Instructions

- mov dst, src
- dst <- src
- add dst, src
- dst <- dst + src
- sub dst, src
- dst <- dst – src
- inc dst
- dst <- dst + 1
- dec dst
- dst <- dst – 1
- neg dst
- dst <- - dst

(2’s complement arithmetic negation)

Integer Multiply

- imul dst, src
- dst <- dst * src
- 32-bit product
- dst must be a register
Bitwise Operations

- **and** `dst, src`
  - `dst <- dst & src`
- **or** `dst, src`
  - `dst <- dst | src`
- **xor** `dst, src`
  - `dst <- dst ^ src`
- **not** `dst`
  - `dst <- ~ dst`

Shifts and Rotates

- **shl** `dst, count`
  - `dst shifted left count bits`
- **shr** `dst, count`
  - `dst shifted right count bits (0 fill)`
- **sar** `dst, count`
  - `dst shifted right count bits (sign bit fill)`
- **rol** `dst, count`
  - `dst shifted left count bits`
- **ror** `dst, count`
  - `dst shifted right count bits`

Uses for Shifts and Rotates

- Can often be used to optimize multiplication and division by small constants.
- There are additional instructions that shift and rotate double words, use a calculated shift amount instead of a constant, etc.

Control Flow - GOTO

- At this level, all we have is goto and conditional goto.
- Loops and conditional statements are synthesized from these.
- A jump (goto) stores the destination address in eip, the register that points to the next instruction to be fetched.
- **Optimization note**: Jumps play havoc with pipeline efficiency; much work is done in modern compilers and processors to minimize this impact.

Unconditional Jumps

- **jmp** `dst`
  - `eip <- address of dst`
  - Assembly language note: `dst` will be a label. Execution continues at first machine instruction in the code following that label.
  - Can have multiple labels on separate lines in front of an instruction.

Conditional Jumps

- Most arithmetic instructions set bits in eflags to record information about the result (zero, non-zero, positive, etc.).
  - True of add, sub, and, or; but not `imul` or `idiv`
  - Other instructions that set eflags:
    - `cmp dst, src` ; compare dst to src
Conditional Jumps Following Arithmetic Operations

```plaintext
jz label ; jump if result == 0
jnz label ; jump if result != 0
jg label ; jump if result > 0
jng label ; jump if result <= 0
jge label ; jump if result >= 0
jnge label ; jump if result < 0
jl label ; jump if result < 0
jnl label ; jump if result >= 0
jle label ; jump if result <= 0
jnle label ; jump if result > 0
```

Compare and Jump Conditionally

- Very common pattern: compare two operands and jump if a relationship holds between them
- Would like to do this
  ```plaintext
  jmp cond op1,op2,label
  ```
  but can’t, because 3-address instructions aren’t included in the architecture

**cmp and jcc**

- Instead, use a 2-instruction sequence
  ```plaintext
cmp op1,op2
jcc label
  ```
  where jcc is a conditional jump that is taken if the result of the comparison matches the condition cc

Conditional Jumps Following a cmp instruction

```plaintext
cmp op1,op2
```

The possibilities include:

- `je` label ; jump if op1 == op2
- `jne` label ; jump if op1 != op2
- `jg` label ; jump if op1 > op2
- `jng` label ; jump if op1 <= op2
- `jge` label ; jump if op1 >= op2
- `jnge` label ; jump if op1 < op2
- `jl` label ; jump if op1 < op2
- `jnl` label ; jump if op1 >= op2
- `jle` label ; jump if op1 <= op2
- `jnle` label ; jump if op1 > op2

Subroutine Calling Issues?

Function Call and Return

- The x86 instruction set itself only provides for transfer of control (jump) and return
  - Stack is used to capture return address and recover it
- Everything else – parameter passing, stack frame organization, register usage – is a matter of convention and not defined by the hardware
call and ret Instructions

- **call** label
  - Push address of next instruction and jump
    - \( \text{esp} \leftarrow \text{esp} - 4; \)
    - \( \text{memory}[	ext{esp}] \leftarrow \text{eip} \)
    - \( \text{eip} \leftarrow \text{address of label} \)
- **ret**
  - Pop address from top of stack and jump
    - \( \text{eip} \leftarrow \text{memory}[	ext{esp}]; \)
    - \( \text{esp} \leftarrow \text{esp} + 4 \)
  - **WARNING!** The word on the top of the stack had better be an address, not some leftover data

Win 32 C Function Call Conventions

- Wintel compilers obey the following conventions for C programs
- We’ll use these conventions in our code

Win 32 C Register Conventions

- These registers must be restored to their original values before a function returns, if they are altered during execution
  - \( \text{esp}, \text{ebp}, \text{ebx}, \text{esi}, \text{edi} \)
- Traditional: push/pop from stack to save/restore
- A function may use the other registers (eax, ecx, edx) however it wants, without having to save/restore them
- A 32-bit function result is expected to be in eax when the function returns

Call Site

- **Caller** is responsible for:
  - Pushing arguments on the stack from right to left
  - Execute call instruction
  - "Pop" arguments from stack after return
    - For us, this means \( \text{add} / 4 \times (\# \text{ arguments}) \) to esp after the return, since everything is a 32-bit variable (int)

Example Function

- Source code
  
  ```c
  int sumOf(int x, int y) {
    int a, int b;
    a = x;
    b = a + y;
    return b;
  }
  ```

Stack Frame for sumOf

```c
int sumOf(int x, int y) {
  int a, int b;
  a = x;
  b = a + y;
  return b;
}
```
Call Example

\[ n = \text{sumOf}(17, 42) \]
\[
\begin{align*}
  &\text{push 42} & \text{; push args} \\
  &\text{push 17} & \text{; jump & push addr} \\
  &\text{call sumOf} & \text{; jump & push addr} \\
  &\text{add esp, 8} & \text{pop args} \\
  &\text{mov [ebp + offset], eax} & \text{store result}
\end{align*}
\]

Callee

- Called function must do the following
  - Save registers if necessary
  - Allocate stack frame for local variables
  - Execute function body
  - Ensure result of non-void function is in eax
  - Restore any required registers if necessary
  - Pop the stack frame
  - Return to caller

Win32 Function Prologue

- The code that needs to be executed before the statements in the body of the function are executed is referred to as the **prologue**
- For a Win32 function \( f \), it looks like this:
  \[
  \begin{align*}
  f: &\text{push ebp} & \text{save old frame pointer} \\
  &\text{mov ebp, esp} & \text{new frame ptr is top of} \\
  &\text{sub esp, "# bytes needed"} & \text{stack after arguments and} \\
  &\text{allocate stack frame} & \text{return address are pushed}
  \end{align*}
  \]

Win32 Function Epilogue

- The **epilogue** is the code that is executed to obey a return statement (or if execution "falls off" the bottom of a void function)
- For a Win32 function, it looks like this:
  \[
  \begin{align*}
  &\text{mov eax, "function result"} & \text{put result in eax if not already} \\
  &\text{mov esp, ebp} & \text{restore esp to old value} \\
  &\text{pop ebp} & \text{before stack frame allocated} \\
  &\text{ret} & \text{return to caller}
  \end{align*}
  \]

Assembly Language Version

\[
\begin{align*}
  ;; &\text{int sumOf(int x, int y)} \{ \\
  ;; &\text{int a, int b;} \\
  &\text{sumOf:} \\
  &\text{push ebp ; prologue} \\
  &\text{mov ebp, esp} \\
  &\text{sub esp, 8} \\
  &\text{a = x;} \\
  &\text{mov eax, [ebp+8]} \\
  &\text{mov [ebp-4], eax} \\
  &\text{\}; b = a + y;} \\
  &\text{mov eax, [ebp-4]} \\
  &\text{add eax, [ebp+12]} \\
  &\text{mov [ebp-8], eax} \\
  &\text{\}; return b;} \\
  &\text{mov eax, [ebp-8]} \\
  &\text{mov esp, ebp} \\
  &\text{pop ebp} \\
  &\text{ret} \\
  &\text{\);}
  \end{align*}
\]

C/C++ Calling Convention: Caller

**Caller** (before you call the callee)
- Save caller-saved registers (EAX, ECX, EDX)
- Push parameters on stack (in inverted order)
- Call !!

**Caller** (when you return from the callee)
- Pop parameters off stack
- Return value will be in EAX
- Restore caller-saved registers
**C/C++ Calling Convention: Callee**

- **Callee (prologue)**
  - Push caller’s EBP onto stack, copy ESP into EBP
  - Allocate local variables on stack
  - Save callee-saved registers (EBX, EDI, ESI)
  - [then actually do the stuff in the callee function]

- **Callee (epilogue)**
  - Put return value in EAX
  - Restore callee-saved registers
  - De-allocate local variables
  - Restore caller’s EBP
  - ret

**Caller-Saved Registers**

- (EAX, ECX, EDX)
  - Caller saves these registers if it cares about the values currently in those registers
  - (The compiler will tend to put temporary values in these registers)

**Callee-Saved Registers**

- (EBX, ESI, EDI)
  - Callee saves these registers if it needs more registers than just EAX, ECX, EDX
  - (The compiler will tend to put long-lived values in these registers)