Compilers:
From Programming to Execution

David Brumley
Carnegie Mellon University
You will find

at least one error

on each set of slides. :)

2
To answer the question

“Is this program safe?”

We need to know

“What will executing this program do?”
What will **executing** this program do?

```c
#include <stdio.h>

void answer(char *name, int x){
    printf("%s, the answer is: %d\n", name, x);
}

void main(int argc, char *argv[]){
    int x;
    x = 40 + 2;
    answer(argv[1], x);
}
```

42.c
void answer(char *name, int x){
    printf("%s, the answer is: %d\n", name, x);
}

void main(int argc, char *argv[]){
    int x;
    x = 40 + 2;
    answer(argv[1], x);
}

David, the answer is 42

The compiler and machine determines the semantics.
“Compiled Code”

Source Language → Compilation → Target Language

Input

Output
“Interpreted Code”

Source Language Input → Interpretation → Output
Today: Overview of Compilation

1. How is C code translated to executable code?
2. What is the machine model for executing code?
Key Concepts

• Compilation workflow
• x86 execution model
• Endian
• Registers
• Stack
• Heap
• Stack frames
Compilation Workflow
Compilation

Source Language
42.c in C

Pre-processor (cpp)
42.c

Compiler (cc1)

Assembler (as)

Linker (ld)

Target Language
42 in x86
Pre-processor (cpp)

#include <stdio.h>
void answer(char *name, int x){
    printf("%s, the answer is: %d\n", name, x);
}
...

#include expansion
#define substitution

$ cpp
Pre-processor (cpp) → Compiler (cc1) → Assembler (as) → Linker (ld)

```
#include <stdio.h>
void answer(char *name, int x){
  printf("%s, the answer is: %d\n", name, x);
}
...
```

$ gcc -S

Creates Assembly
gcc –S 42.c outputs 42.s

_answer:
Leh_func_begin1:
    pushq  %rbp
Ltmp0:
    movq  %rsp, %rbp
Ltmp1:
    subq  $16, %rsp
Ltmp2:
    movl  %esi, %eax
    movq  %rdi, -8(%rbp)
    movl  %eax, -12(%rbp)
    movq  -8(%rbp), %rax
    ....
Pre-processor (cpp) -> Compiler (cc1) -> Assembler (as) -> Linker (ld)

$ as <options>

Creates object code

_answer:
Leh_func_begin1:
  pushq %rbp
Ltmp0:
  movq %rsp, %rbp
Ltmp1:
  subq $16, %rsp
Ltmp2:
  movl %esi, %eax
  movq %rdi, -8(%rbp)
  movl %eax, -12(%rbp)
  movq -8(%rbp), %rax
  ....
42.s
$ ld <options>

Links with other files and libraries to produce an exe
Disassembling

• Today: using objdump (part of binutils)
  – objdump -D <exe>

• If you compile with “-g”, you will see more information
  – objdump -D -S

• Later: Disassembly
The program **binary** (aka executable)

Final executable consists of several **segments**

- Text for code written
- Read-only data for constants such as “hello world” and globals
- ... 

```
$ readelf -S <file>
```
Basic Execution Model
Basic Execution

File system

Binary
  Code
  Data
  ...

Process Memory
  Stack
  Heap

Processor

Fetch, decode, execute

read and write
x86 Processor

- EIP
- EFLAGS
- EAX
- EDX
- ECX
- EBX
- ESP
- EBP
- ESI
- EDI

- Address of next instruction
- Condition codes
- General Purpose
Registers have up to 4 addressing modes

1. Lower 8 bits
2. Mid 8 bits
3. Lower 16 bits
4. Full register
EAX, EDX, ECX, and EBX

• 32 bit registers (three letters)
• Lower bits (bits 0-7) (two letters with L suffix)
• Mid-bits (bits 8-15) (two letters with H suffix)

<table>
<thead>
<tr>
<th>EAX</th>
<th>AH</th>
<th>AL</th>
</tr>
</thead>
<tbody>
<tr>
<td>EDX</td>
<td>DH</td>
<td>DL</td>
</tr>
<tr>
<td>ECX</td>
<td>CH</td>
<td>CL</td>
</tr>
<tr>
<td>EBX</td>
<td>BH</td>
<td>BL</td>
</tr>
</tbody>
</table>

- Bit 32
- 16 15
- 8 7

<table>
<thead>
<tr>
<th>EAX</th>
<th>AX</th>
</tr>
</thead>
<tbody>
<tr>
<td>EDX</td>
<td>DX</td>
</tr>
<tr>
<td>ECX</td>
<td>CX</td>
</tr>
<tr>
<td>EBX</td>
<td>BX</td>
</tr>
</tbody>
</table>

- Bit 32
- 16 15
- 0

- Lower 16 bits (bits 0-15) (2 letters with X suffix)
**ESP, EBP, ESI, and EDI**

<table>
<thead>
<tr>
<th>EAX</th>
<th>AH</th>
<th>AL</th>
</tr>
</thead>
<tbody>
<tr>
<td>EDX</td>
<td>DH</td>
<td>DL</td>
</tr>
<tr>
<td>ECX</td>
<td>CH</td>
<td>CL</td>
</tr>
<tr>
<td>EBX</td>
<td>BH</td>
<td>BL</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ESP</th>
<th>SP</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBP</td>
<td>BP</td>
</tr>
<tr>
<td>ESI</td>
<td>SI</td>
</tr>
<tr>
<td>EDI</td>
<td>DI</td>
</tr>
</tbody>
</table>

Bit 32 | 16 | 15 | 0

- Lower 16 bits (bits 0-15) (2 letters)
Basic Ops and AT&T vs Intel Syntax

<table>
<thead>
<tr>
<th>Meaning</th>
<th>AT&amp;T</th>
<th>Intel</th>
</tr>
</thead>
<tbody>
<tr>
<td>ebx = eax</td>
<td>movl %eax, %ebx</td>
<td>mov ebx, eax</td>
</tr>
<tr>
<td>eax = eax + ebx</td>
<td>addl %ebx, %eax</td>
<td>add eax, ebx</td>
</tr>
<tr>
<td>ecx = ecx &lt;&lt; 2</td>
<td>shl $2, %ecx</td>
<td>shl ecx, 2</td>
</tr>
</tbody>
</table>

- AT&T is at odds with assignment order. It is the default for objdump, and traditionally used for UNIX.

- Intel order mirrors assignment. Windows traditionally uses Intel, as is available via the objdump ‘-M intel’ command line option.
Memory Operations
x86: **Byte Addressable**

**It's convention:** lower address at the bottom

- Address 3 holds 1 byte
- Address 2 holds 1 byte
- Address 1 holds 1 byte
- Address 0 holds 1 byte

I can fetch bytes at any address

Memory is just like using an array!

Alternative: **Word addressable**

**Example:** For 32-bit word size, it’s valid to fetch 4 bytes from Mem[0], but not Mem[6] since 6 is not a multiple of 4.
x86: Addressing bytes

Addresses are indicated by operands that have a bracket “[ ]” or paren “()”, for Intel vs. AT&T, resp.

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>eax</td>
<td>0x3</td>
</tr>
<tr>
<td>edx</td>
<td>0x0</td>
</tr>
<tr>
<td>ebx</td>
<td>0x5</td>
</tr>
</tbody>
</table>

What does `mov dl, [al]` do?

Moves 0xccc into dl
x86: Addressing bytes

Addresses are indicated by operands that have a bracket “[]” or paren “()”, for Intel vs. AT&T, resp.

What does `mov edx, [eax]` do?

Which 4 bytes get moved, and which is the LSB in edx?

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>eax</td>
<td>0x3</td>
</tr>
<tr>
<td>edx</td>
<td>0xcc</td>
</tr>
<tr>
<td>ebx</td>
<td>0x5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
</tr>
<tr>
<td>0xaa</td>
</tr>
<tr>
<td>0xbb</td>
</tr>
<tr>
<td>0xcc</td>
</tr>
<tr>
<td>0xee</td>
</tr>
<tr>
<td>0xff</td>
</tr>
</tbody>
</table>
Endianness

- **Endianness**: Order of individually addressable units
- **Little Endian**: Least significant byte first

so address \( a \) goes in littlest byte (e.g., AL), \( a+1 \) in the next (e.g., AH), etc.

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>eax</td>
<td>0x3</td>
</tr>
<tr>
<td>edx</td>
<td>0xcc</td>
</tr>
<tr>
<td>ebx</td>
<td>0x5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
</tr>
<tr>
<td>0xaa</td>
</tr>
<tr>
<td>0xbb</td>
</tr>
<tr>
<td>0xdd</td>
</tr>
<tr>
<td>0xee</td>
</tr>
<tr>
<td>0xff</td>
</tr>
</tbody>
</table>
**mov edx, [eax]**

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>eax</td>
<td>0x3</td>
</tr>
<tr>
<td>edx</td>
<td>0xcc</td>
</tr>
<tr>
<td>ebx</td>
<td>0x5</td>
</tr>
</tbody>
</table>

**EDX**

**EDX = 0xffeeddcc!**

**Endianess**: Ordering of individually addressable units

**Little Endian**: Least significant byte first

... so ...

address \(a\) goes in the least significant byte (the littlest bit) \(a+1\) goes into the next byte, and so on.
mov [eax], ebx

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>eax</td>
<td>0x3</td>
</tr>
<tr>
<td>edx</td>
<td>0xcc</td>
</tr>
<tr>
<td>ebx</td>
<td>0x5</td>
</tr>
</tbody>
</table>

**Endianess**: Ordering of individually addressable units

**Little Endian**: Least significant byte first

... so ...

address \( a \) goes in the least significant byte (the **littlest** bit) \( a+1 \) goes into the next byte, and so on.
There are other ways to address memory than just [register].

These are called *Addressing Modes*.

An *Addressing Mode* specifies how to calculate the effective memory address of an operand by using information from registers and constants contained with the instruction or elsewhere.
Type buf[s];
buf[index] = *(<buf addr>+sizeof(Type)*index)
typedef uint32_t addr_t;
uint32_t w, x, y, z;
uint32_t buf[3] = {1,2,3};
addr_t ptr = (addr_t) buf;

w = buf[2];
x = *(buf + 2);

What is x? what memory cell does it ref?
Motivation: Addressing Buffers

typedef uint32_t addr_t;
uint32_t w, x, y, z;
uint32_t buf[3] = {1,2,3};
addr_t ptr = (addr_t) buf;
w = buf[2];
x = *(buf + 2);
y = *( (uint32_t *) (ptr+8));

Equivalent
(addr_t) (ptr + 8) = (uint32_t *) buf+2
Motivation: Addressing Buffers

Type buf[s];
buf[index] = *(<buf addr>+sizeof(Type)*index)

Say at imm + r₁
Constant scaling factor s, typically 1, 2, 4, or 8
Say in Register r₂

imm + r₁ + s*r₂

AT&T: imm (r₁, r₂, s)
Intel: r₁ + r₂*s + imm
AT&T Addressing Modes for Common Codes

<table>
<thead>
<tr>
<th>Form</th>
<th>Meaning on memory M</th>
</tr>
</thead>
<tbody>
<tr>
<td>imm (r)</td>
<td>M[r + imm]</td>
</tr>
<tr>
<td>imm (r₁, r₂)</td>
<td>M[r₁ + r₂ + imm]</td>
</tr>
<tr>
<td>imm (r₁, r₂, s)</td>
<td>M[r₁ + r₂*s + imm]</td>
</tr>
<tr>
<td>imm</td>
<td>M[imm]</td>
</tr>
</tbody>
</table>
Referencing Memory

Loading a value from memory: mov

\[ \langle \text{eax} \rangle = \ast \text{buf}; \]

\[
\begin{align*}
\text{mov} & \ -0x38(\%ebp),\%eax \ (I) \\
\text{mov} & \ \text{eax}, \ [\text{ebp}-0x38] \ (A)
\end{align*}
\]

Loading an address: lea

\[ \langle \text{eax} \rangle = \text{buf}; \]

\[
\begin{align*}
\text{lea} & \ -0x38(\%ebp),\%eax \ (I) \\
\text{lea} & \ \text{eax}, \ [\text{ebp}-0x38] \ (A)
\end{align*}
\]
Suppose I want to access address 0xdeadbeef directly

Loads the address

```
lea eax, 0xdeadbeef  (I)
```

Deref the address

```
mov eax, 0xdeadbeef  (I)
```

Note missing $. This distinguishes the address from the value
Control Flow
Assembly is “Spaghetti Code”

Nice C Abstractions
• if-then-else
• while
• for loops
• do-while

Assembly
• Jump
  – Direct: jmp addr
  – Indirect: jmp reg
• Branch
  – Test EFLAG
  – if(EFLAG SET) goto line
Jumps
• jmp 0x45, called a **direct jump**
• jmp *eax, called an **indirect jump**

Branches
• if (EFLAG) jmp x
  Use one of the 32 EFLAG bits to determine if jump taken

**Note:**
No direct way to get or set EIP
Implementing “if”

C
1. if(x <= y)
2.   z = x;
3. else
4.   z = y;

Psuedo-Assembly
1. Computing x – y. Set eflags:
   1. CF =1 if x < y
   2. ZF =1 if x==y
2. Test EFLAGS. If both CF and ZF not set, branch to E
3. mov x, z
   Jump to F
4. mov y, z
   <end of if-then-else>

Assembly is 2 instrs
1. Set eflag to conditional
2. Test eflag and branch
If \( x > y \)

%eax holds \( x \) and \( 0xc(\%ebp) \) holds \( y \)

cmp \( 0xc(\%ebp), \%eax \)

ja addr

Same as “sub” instruction

\( r = \%eax - M[\%ebp+0xc] \), i.e., \( x - y \)

Jump if \( CF=0 \) and \( ZF=0 \)

\( (x \geq y) \land (x \neq y) \implies x > y \)
Setting EFLAGS

• Instructions may set an eflag, e.g.,
• “cmp” and arithmetic instructions most common
  – Was there a carry (CF Flag set)
  – Was the result zero (ZF Flag set)
  – What was the parity of the result (PF flag)
  – Did overflow occur (OF Flag)
  – Is the result signed (SF Flag)
Aside: Although the x86 processor knows every time integer overflow occurs, C does not make this result visible.

From the Intel x86 manual
See the x86 manuals available on Intel’s website for more information

<table>
<thead>
<tr>
<th>Instr.</th>
<th>Description</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>JO</td>
<td>Jump if overflow</td>
<td>OF == 1</td>
</tr>
<tr>
<td>JNO</td>
<td>Jump if not overflow</td>
<td>OF == 0</td>
</tr>
<tr>
<td>JS</td>
<td>Jump if sign</td>
<td>SF == 1</td>
</tr>
<tr>
<td>JZ</td>
<td>Jump if zero</td>
<td>ZF == 1</td>
</tr>
<tr>
<td>JE</td>
<td>Jump if equal</td>
<td>ZF == 1</td>
</tr>
<tr>
<td>JL</td>
<td>Jump if less than</td>
<td>SF &lt;&gt; OF</td>
</tr>
<tr>
<td>JLE</td>
<td>Jump if less than or equal</td>
<td>ZF == 1 or SF &lt;&gt; OF</td>
</tr>
<tr>
<td>JB</td>
<td>Jump if below</td>
<td>CF == 1</td>
</tr>
<tr>
<td>JP</td>
<td>Jump if parity</td>
<td>PF == 1</td>
</tr>
</tbody>
</table>
Memory Organization
The Stack grows down towards lower addresses.
Variables

• On the stack
  – Local variables
  – Lifetime: stack frame

• On the heap
  – Dynamically allocated via new/malloc/etc.
  – Lifetime: until freed
Procedures

• Procedures are not native to assembly
• Compilers *implement* procedures
  – On the stack
  – Following the call/return stack discipline
Procedures/Functions

• We need to address several issues:
  1. How to allocate space for local variables
  2. How to pass parameters
  3. How to pass return values
  4. How to share 8 registers with an infinite number of local variables

• A stack frame provides space for these values
  – Each procedure invocation has its own stack frame
  – Stack discipline is LIFO
    • If procedure A calls B, B’s frame must exit before A’s
orange(...) {
    ...
    red()
    ...
}

red(...) {
    ...
    green()
    ...
    green()
}

green(...) {
    ...
    green()
    ...
}
Frame for
- locals
- pushing parameters
- temporary space

Call to red
"pushes"
new frame

When green returns it
"pops"
its frame

Function Call Chain

orange
↓
red
↓
green
↓
green
↓
...

On the stack

```c
int orange(int a, int b) {
    char buf[16];
    int c, d;
    if(a > b)
        c = a;
    else
        c = b;
    d = red(c, buf);
    return d;
}
```

Calling convention determines the above features

Need to access arguments

Need space to store local vars (buf, c, and d)

Need space to put arguments for callee

Need a way for callee to return values
cdecl – the default for Linux & gcc

```c
int orange(int a, int b) {
    char buf[16];
    int c, d;
    if(a > b)
        c = a;
    else
        c = b;
    d = red(c, buf);
    return d;
}
```

Don’t worry! We will walk through these one by one.

Frame structure:
- `%ebp` frame
- `%esp` stack

Parameter area (caller):
- `return addr`
- `callee’s ebp`
- `caller’s ebp`
- `locals` (buf, c, d ≥ 24 bytes if stored on stack)
- `caller-save`
- `buf`
- `c`
- `return addr`
- `orange’s ebp`
- `...`

Flow:
- Before calling `red`
- After `red` has been called
When **orange** attains control,

1. return address has already been pushed onto stack by caller
When *orange* attains control,

1. return address has already been pushed onto stack by caller

2. own the frame pointer
   - push caller’s ebp
   - copy current esp into ebp
   - first argument is at ebp+8
When orange attains control,

1. return address has already been pushed onto stack by caller

2. own the frame pointer
   - push caller’s ebp
   - copy current esp into ebp
   - first argument is at ebp+8

3. save values of other callee-save registers if used
   - edi, esi, ebx: via push or mov
   - esp: can restore by arithmetic
When orange attains control,

1. return address has already been pushed onto stack by caller

2. own the frame pointer
   - push caller’s ebp
   - copy current esp into ebp
   - first argument is at ebp+8

3. save values of other callee-save registers if used
   - edi, esi, ebx: via push or mov
   - esp: can restore by arithmetic

4. allocate space for locals
   - subtracting from esp
   - “live” variables in registers, which on contention, can be “spilled” to stack space
For *caller orange* to call *callee red*,

```
...  
b  
a  
return addr  
caller's ebp  
callee-save  
locals  
(buf, c, d ≥ 24 bytes if stored on stack)  
%ebp  
%esp  
```
For *caller orange* to call *callee red*,

1. push any caller-save registers if their values are needed after *red* returns
   - eax, edx, ecx
For *caller orange* to call *callee red*,

1. push any caller-save registers if their values are needed after *red* returns
   - `eax`, `edx`, `ecx`

2. push arguments to *red* from right to left (reversed)
   - from callee’s perspective, argument 1 is nearest in stack
For *caller orange* to call *callee red*,

1. push any caller-save registers if their values are needed after *red* returns
   - eax, edx, ecx
2. push arguments to *red* from right to left (reversed)
   - from callee’s perspective, argument 1 is nearest in stack
3. push return address, i.e., the next instruction to execute in *orange* after *red* returns
For *caller orange* to call *callee red*,

1. push any caller-save registers if
   their values are needed after
   *red* returns
   - eax, edx, ecx

2. push arguments to *red* from
   right to left (reversed)
   - from callee’s perspective,
     argument 1 is nearest in stack

3. push return address, i.e., the
   *next* instruction to execute in
   *orange* after *red* returns

4. transfer control to *red*
   - usually happens together with
     step 3 using *call*
When red attains control,

1. return address has already been pushed onto stack by orange
When red attains control,

1. return address has already been pushed onto stack by orange
2. own the frame pointer
When \textit{red} attains control,

1. return address has already been pushed onto stack by \textit{orange}
2. own the frame pointer
3. ... (red is doing its stuff) ...
When red attains control,

1. return address has already been pushed onto stack by orange
2. own the frame pointer
3. ... (red is doing its stuff) ...
4. store return value, if any, in eax
5. deallocate locals
   - adding to esp
6. restore any callee-save registers
When *red* attains control,

1. return address has already been pushed onto stack by *orange*
2. own the frame pointer
3. ... (*red* is doing its stuff) ...
4. store return value, if any, in eax
5. deallocate locals
   - adding to esp
6. restore any callee-save registers
7. restore *orange*’s frame pointer
   - pop %ebp

<table>
<thead>
<tr>
<th>...</th>
<th>%esp</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td></td>
</tr>
<tr>
<td>return addr</td>
<td></td>
</tr>
<tr>
<td>caller’s ebp</td>
<td></td>
</tr>
<tr>
<td>callee-save</td>
<td></td>
</tr>
<tr>
<td>locals</td>
<td></td>
</tr>
<tr>
<td>return addr</td>
<td></td>
</tr>
<tr>
<td>%ebp</td>
<td></td>
</tr>
<tr>
<td>%esp</td>
<td></td>
</tr>
</tbody>
</table>

(buf, c, d ≥ 24 bytes if stored on stack)

caller-save

buf
c

return addr

%ebp

%esp
When *red* attains control,

1. return address has already been pushed onto stack by *orange*
2. own the frame pointer
3. *(red is doing its stuff)* ...
4. store return value, if any, in eax
5. deallocate locals
   - adding to esp
6. restore any callee-save registers
7. restore *orange’s* frame pointer
   - pop %ebp
8. return control to *orange*
   - ret
   - pops return address from stack and jumps there

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>return addr</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>caller’s ebp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>callee-save</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>locals</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(buf, c, d ≥ 24 bytes if stored on stack)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>caller-save</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>buf</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

%ebp %esp
When *orange* regains control,
When orange regains control,

1. clean up arguments to red
   - adding to esp

2. restore any caller-save registers
   - pops

3. ...

![Diagram of stack frame with ebp, esp, return addr, caller’s ebp, callee-save, locals (buf, c, d ≥ 24 bytes if stored on stack)]
Terminology

• **Function Prologue** – instructions to set up stack space and save callee saved registers
  – Typical sequence:
    - push ebp
    - ebp = esp
    - esp = esp - <frame space>

• **Function Epilogue** - instructions to clean up stack space and restore callee saved registers
  – Typical Sequence:
    - leave     // esp = ebp, pop ebp
    - ret       // pop and jump to ret addr
### cdecl – One Convention

<table>
<thead>
<tr>
<th>Action</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>caller saves: eax, edx, ecx</td>
<td>push (old), or mov if esp already adjusted</td>
</tr>
<tr>
<td>arguments pushed right-to-left</td>
<td>call pushes return addr</td>
</tr>
<tr>
<td>linkage data starts new frame</td>
<td></td>
</tr>
<tr>
<td>callee saves: ebx, esi, edi, ebp, esp</td>
<td>ebp often used to deref args and local vars</td>
</tr>
<tr>
<td>return value</td>
<td>pass back using eax</td>
</tr>
<tr>
<td>argument cleanup</td>
<td>caller’s responsibility</td>
</tr>
</tbody>
</table>
Q&A

• Why do we need calling conventions?

• Does the callee *always* have to save callee-saved registers?

• How do you think varargs works (va_start, va_arg, etc)?

```c
void myprintf(const char *fmt, ...){}
```
Today’s Key Concepts

• Compiler workflow
• Register to register moves
  – Register mnemonics
• Register/memory
  – mov and addressing modes for common codes
• Control flow
  – EFLAGS
• Program Memory Organization
  – Stack grows down
• Functions
  – Pass arguments, callee and caller saved, stack frame
For more information

• Overall machine model: 
  Computer Systems, a Programmer’s Perspective by Bryant and O’Hallaron

• Calling Conventions:
Questions?
END
Backup slides here.

• Titled cherries because they are for the pickin. (credit due to maverick for wit)
“Compiled Code”

Source Language $\xrightarrow{\text{Compilation}}$ Target Language

VS

“Interpreted Code”

Input $\xrightarrow{\text{Interpretation}}$ Output
Stencils
Other Colors from Adobe Kuler

Don’t use these unless absolutely necessary. We are not making skittles, so there is no rainbow of colors necessary.

Mac application for Adobe Kuler:
http://www.lithoglyph.com/mondrianum/
http://kuler.adobe.com/
To answer the question

“Is this program safe?”

We need to know

“What will executing this program do?”

Understanding the compiler and machine semantics are key.