Introduction to Intel x86 Assembly, Architecture, Applications

Modified version of slides by

Xeno Kovah – 2009/2010

xkovah at gmail

Approved for Public Release: 10-3348. Distribution Unlimited

Simple Hello World

```
#include <stdio.h>
int main(){
    printf(“Hello World!\n”);
    return 0x1234;
}
```

Is the same as…

```
.text:00401730 main
»  .text:00401730              push    ebp
»  .text:00401731              mov     ebp, esp
»  .text:00401733              push    offset aHelloWorld ; "Hello world\n"
»  .text:00401738              call    ds:__imp__printf
»  .text:0040173E              add     esp, 4
»  .text:00401741              mov     eax, 1234h
»  .text:00401746              pop     ebp
»  .text:00401747              retn
```

Windows Visual C++ 2005, /GS (buffer overflow protection) option turned off
Disassembled with IDA Pro 4.9 Free Version

Is the same as…

```
08048374 <main>:
 8048374:       8d 4c 24 04             lea    0x4(%esp),%ecx
 8048378:       83 e4 f0                and    $0xfffffff0,%esp
 804837b:       ff 71 fc                pushl  -0x4(%ecx)
 804837e:       55                      push   %ebp
 804837f:       89 e5                   mov    %esp,%ebp
 8048381:       51                      push   %ecx
 8048382:       83 ec 04                sub    $0x4,%esp
 8048385:       c7 04 24 60 84 04 08    movl   $0x8048460,(%esp)
 804838c:       e8 43 ff ff ff          call   80482d4 <puts@plt>
 8048391:       b8 2a 00 00 00          mov    $0x1234,%eax
 8048396:       83 c4 04                add    $0x4,%esp
 8048399:       59                      pop    %ecx
 804839a:       5d                      pop    %ebp
 804839b:       8d 61 fc                lea    -0x4(%ecx),%esp
 804839e:       c3                      ret
 804839f:       90                      nop
```

Ubuntu 8.04, GCC 4.2.4
Disassembled with “objdump -d”

Is the same as…

```
_main:
00001fca        pushl   %ebp
00001fcb        movl    %esp,%ebp
00001fcd        pushl   %ebx
00001fce        subl    $0x14,%esp
00001fd1        calll   0x00001fd6
00001fd6        popl    %ebx
00001fd7        leal    0x0000001a(%ebx),%eax
00001fdd        movl    %eax,(%esp)
00001fe0        calll   0x00003005     ; symbol stub for: _puts
00001fe5        movl    $0x00001234,%eax
00001fea        addl    $0x14,%esp
00001fed        popl    %ebx
00001fee        leave
00001fef        ret
```

Mac OS 10.5.6, GCC 4.0.1
Disassembled from command line with “otool -tV”

But it all boils down to…

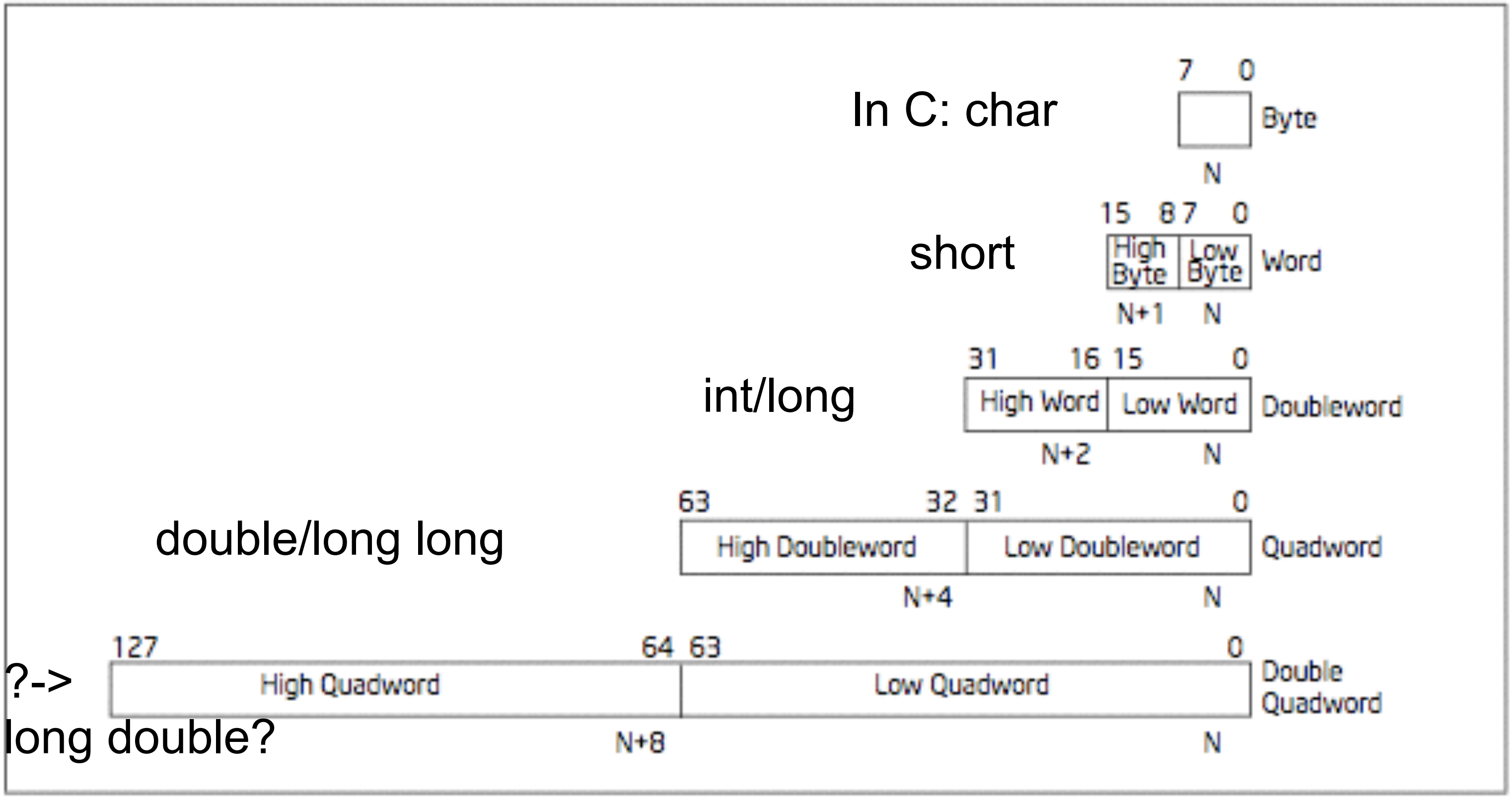
```
.text:00401000 main
.text:00401000       push    offset aHelloWorld ; "Hello world\n"
.text:00401005       call    printf
.text:0040100C       mov     eax, 1234h
.text:00401011       retn
```

Windows Visual C++ 2005, /GS (buffer overflow protection) option turned off
Optimize for minimum size (/O1) turned on
Disassembled with IDA Pro 4.9 Free Version

Take Heart!

- By one measure, only 14 assembly instructions account for 90% of code!
  - http://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Bilar.pdf
- I think that knowing about 20-30 (not counting variations) is good enough that you will have the check the manual very infrequently
- You've already seen 11 instructions, just in the hello world variations!

Refresher - Data Types



In C: char

short

int/long

double/long long

?->
long double?

Figure 4-1. Fundamental Data Types

Architecture - CISC vs. RISC

- Intel is CISC - Complex Instruction Set Computer
  - Many very special purpose instructions that you will never see, and a given compiler may never use - just need to know how to use the manual
  - Variable-length instructions
- Other major architectures are typically RISC - Reduced Instruction Set Computer
  - Typically more registers, less and fixed-size instructions
  - Examples: PowerPC, ARM, SPARC, MIPS

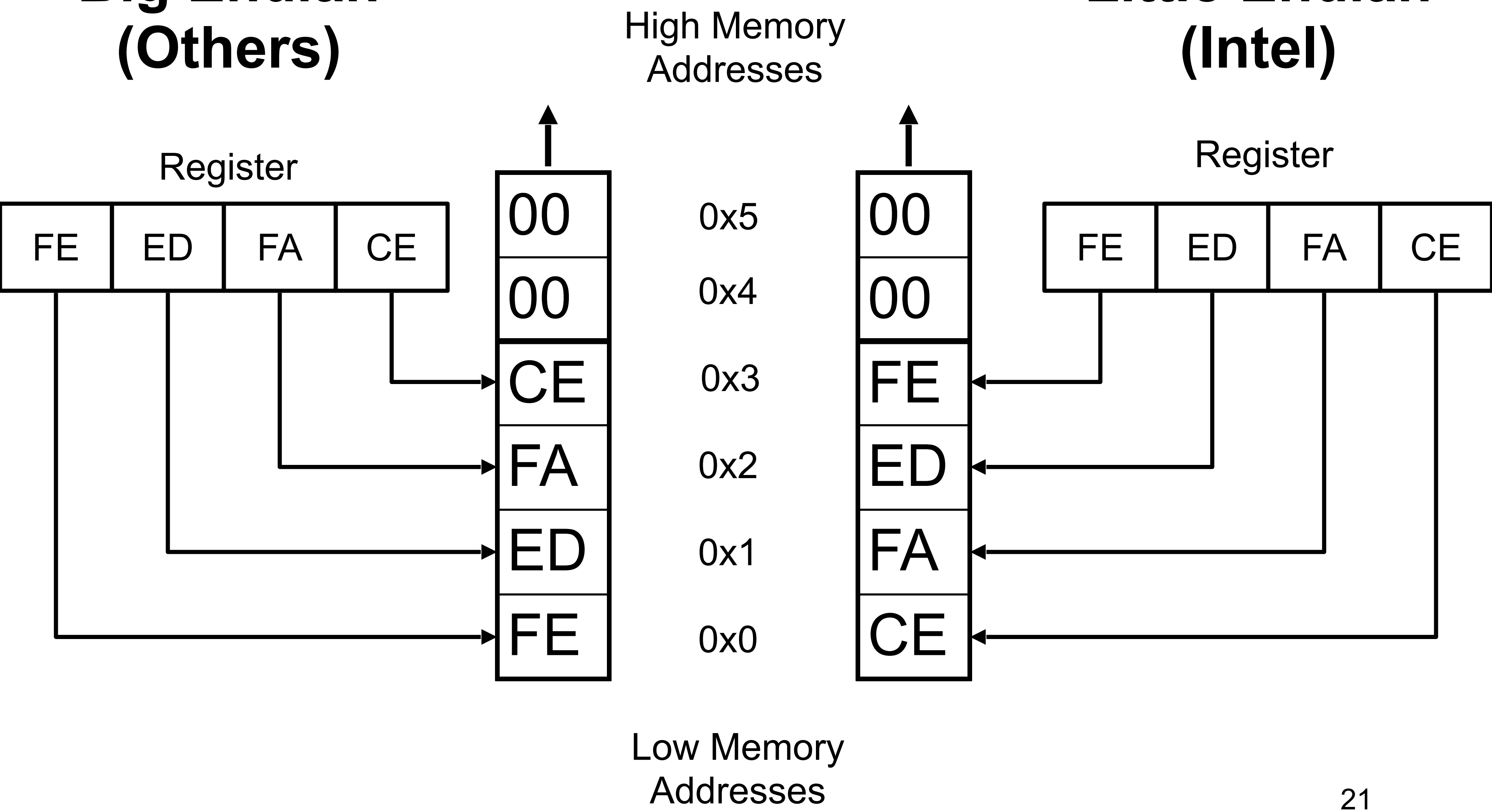
Architecture - Endian

- Little Endian - 0x12345678 stored in RAM “little end” first. The least significant byte of a word or larger is stored in the lowest address. E.g. 0x78563412
  - Intel is Little Endian
- Big Endian - 0x12345678 stored as is.
  - Network traffic is Big Endian
  - Most everyone else you've heard of (PowerPC, ARM, SPARC, MIPS) is either Big Endian by default or can be configured as either (Bi-Endian)

Endianess pictures

Big Endian (Others)

Little Endian (Intel)



High Memory Addresses

Register

FE ED FA CE

00
00
CE
FA
ED
FE

0x5
0x4
0x3
0x2
0x1
0x0

00
00
FE
ED
FA
CE

Register

FE ED FA CE

Low Memory Addresses

Architecture - Registers

- Registers are small memory storage areas built into the processor
- 8 “general purpose” registers + the instruction pointer which points at the next instruction to execute
  - But two of the 8 are not that general
- On x86-32, registers are 32 bits long
- On x86-64, they're 64 bits

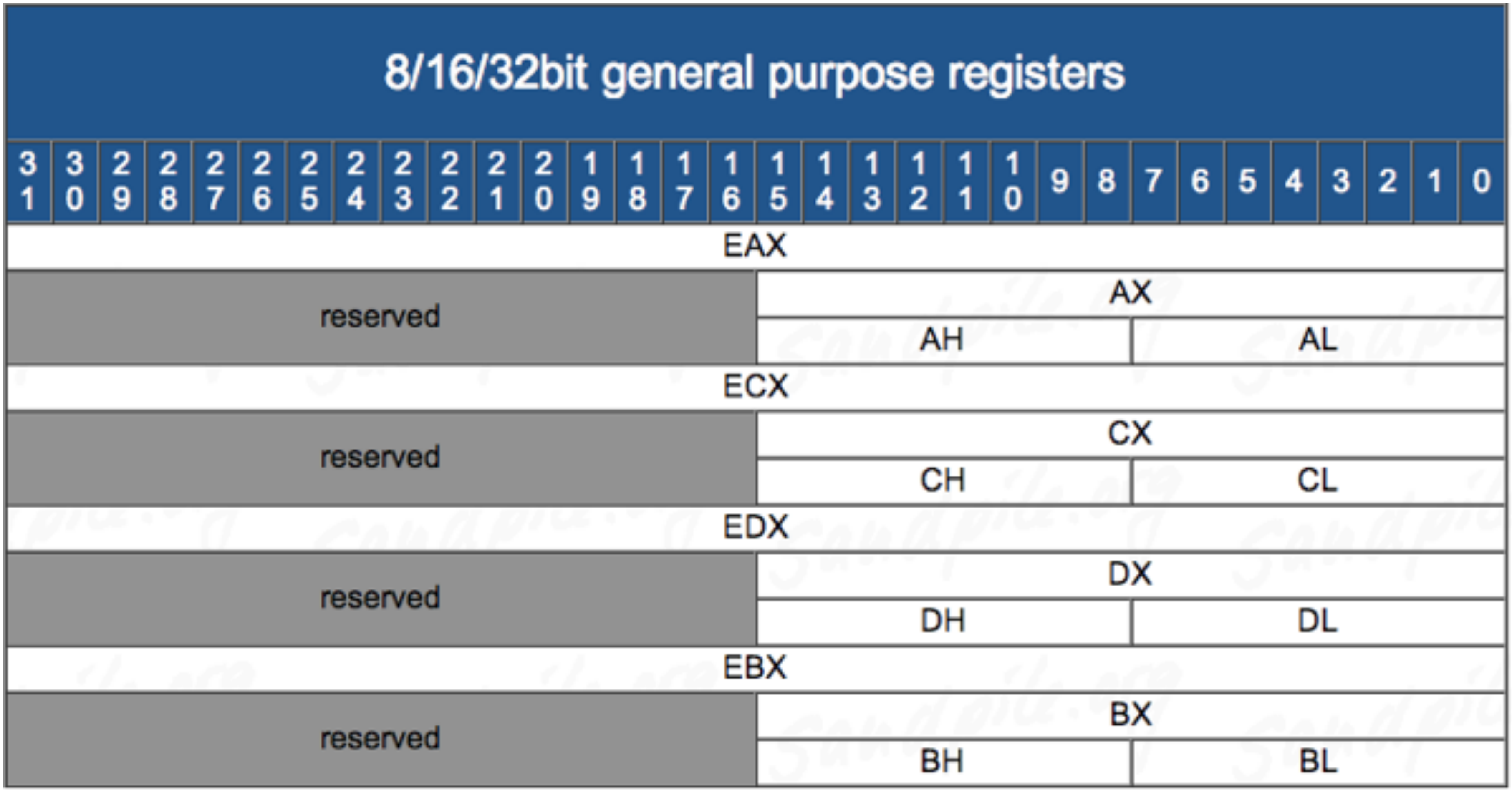
Architecture - Registers Conventions 2

- **ESI** – Source pointer for string operations
- **EDI** – Destination pointer for string operations
- **ESP** – Stack pointer
- **EBP** – Stack frame base pointer
- **EIP** - Pointer to next instruction to execute ("instruction pointer")

Architecture - Registers Conventions 3

- Caller-save registers - eax, edx, ecx
  - If the caller has anything in the registers that it cares about, the caller is in charge of saving the value before a call to a subroutine, and restoring the value after the call returns
  - Put another way - the callee can (and is highly likely to) modify values in caller-save registers
- Callee-save registers - ebp, ebx, esi, edi
  - If the callee needs to use more registers than are saved by the caller, the callee is responsible for making sure the values are stored/restored
  - Put another way - the callee must be a good citizen and not modify registers which the caller didn't save, unless the callee itself saves and restores the existing values

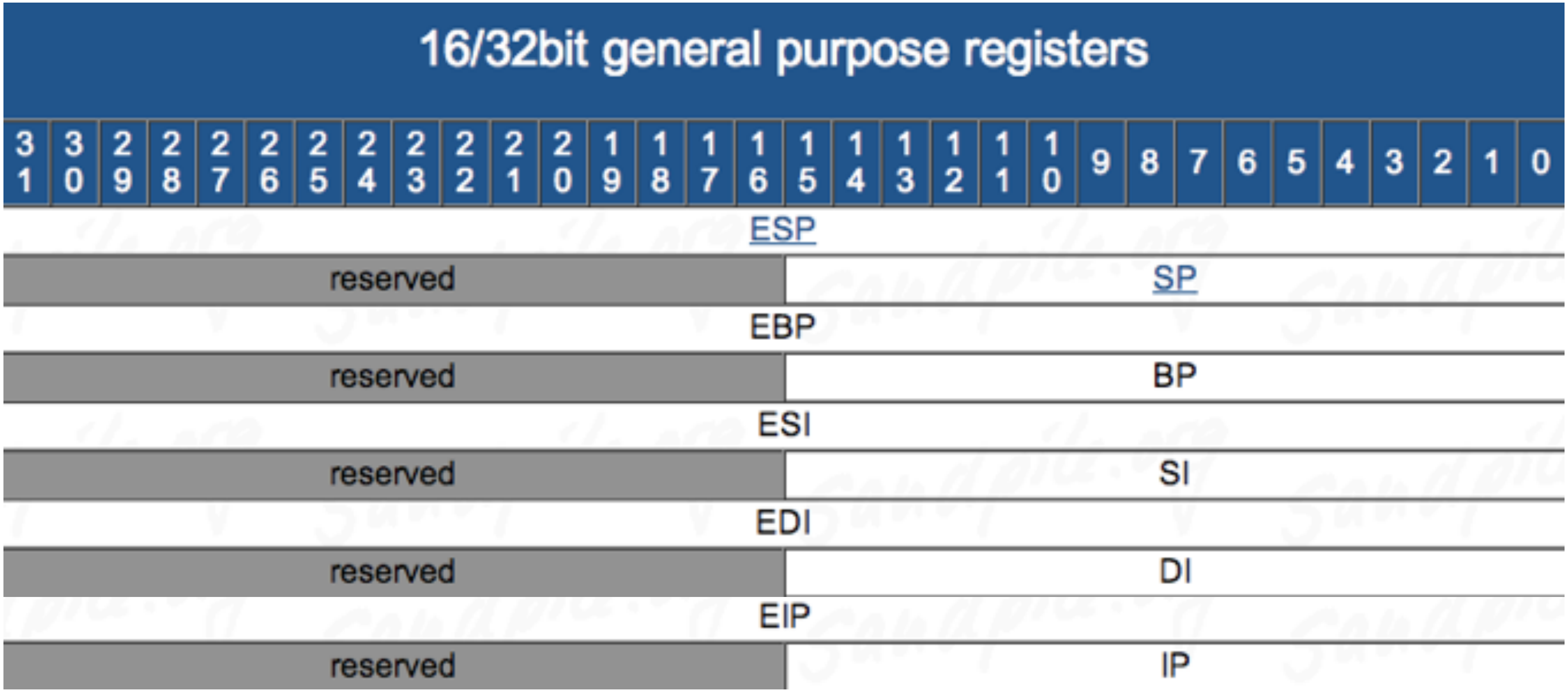
Architecture - Registers - 8/16/32 bit addressing 1



<table>
<tr><th colspan="32">8/16/32bit general purpose registers</th></tr>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="32">EAX</td></tr>
<tr><td colspan="16" rowspan="2">reserved</td><td colspan="16">AX</td></tr>
<tr><td colspan="8">AH</td><td colspan="8">AL</td></tr>
<tr><td colspan="32">ECX</td></tr>
<tr><td colspan="16" rowspan="2">reserved</td><td colspan="16">CX</td></tr>
<tr><td colspan="8">CH</td><td colspan="8">CL</td></tr>
<tr><td colspan="32">EDX</td></tr>
<tr><td colspan="16" rowspan="2">reserved</td><td colspan="16">DX</td></tr>
<tr><td colspan="8">DH</td><td colspan="8">DL</td></tr>
<tr><td colspan="32">EBX</td></tr>
<tr><td colspan="16" rowspan="2">reserved</td><td colspan="16">BX</td></tr>
<tr><td colspan="8">BH</td><td colspan="8">BL</td></tr>
</table>

http://www.sandpile.org/ia32/reg.htm

Architecture - Registers - 8/16/32 bit addressing 2



<table>
<tr><th colspan="32">16/32bit general purpose registers</th></tr>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="32">ESP</td></tr>
<tr><td colspan="16">reserved</td><td colspan="16">SP</td></tr>
<tr><td colspan="32">EBP</td></tr>
<tr><td colspan="16">reserved</td><td colspan="16">BP</td></tr>
<tr><td colspan="32">ESI</td></tr>
<tr><td colspan="16">reserved</td><td colspan="16">SI</td></tr>
<tr><td colspan="32">EDI</td></tr>
<tr><td colspan="16">reserved</td><td colspan="16">DI</td></tr>
<tr><td colspan="32">EIP</td></tr>
<tr><td colspan="16">reserved</td><td colspan="16">IP</td></tr>
</table>

http://www.sandpile.org/ia32/reg.htm

Architecture - EFLAGS

- EFLAGS register holds many single bit flags. Will only ask you to remember the following for now.
  - Zero Flag (ZF) - Set if the result of some instruction is zero; cleared otherwise.
  - Sign Flag (SF) - Set equal to the most-significant bit of the result, which is the sign bit of a signed integer. (0 indicates a positive value and 1 indicates a negative value.)

Intel Vol 1 Sec 3.4.3 - page 3-20

1

Your first x86 instruction: NOP

- NOP - No Operation! No registers, no values, no nothin'!
- Just there to pad/align bytes, or to delay time
- Bad guys use it to make simple exploits more reliable. But that's another class ;)

The Stack

- The stack is a conceptual area of main memory (RAM) which is designated by the OS when a program is started.
  - Different OS start it at different addresses by convention
- A stack is a Last-In-First-Out (LIFO/FILO) data structure where data is "pushed" on to the top of the stack and "popped" off the top.
- By convention the stack grows toward lower memory addresses. Adding something to the stack means the top of the stack is now at a lower memory address.

The Stack 2

- As already mentioned, esp points to the top of the stack, the lowest address which is being used
  - While data will exist at addresses beyond the top of the stack, it is considered undefined
- The stack keeps track of which functions were called before the current one, it holds local variables and is frequently used to pass arguments to the next function to be called.
- A firm understanding of what is happening on the stack is *__essential__* to understanding a program's operation.

2

PUSH - Push Word, Doubleword or Quadword onto the Stack

- For our purposes, it will always be a DWORD (4 bytes).
  - Can either be an immediate (a numeric constant), or the value in a register
- The push instruction automatically decrements the stack pointer, esp, by 4.

Book p. 120

Registers Before

eax	0x00000003
esp	0x0012FF8C

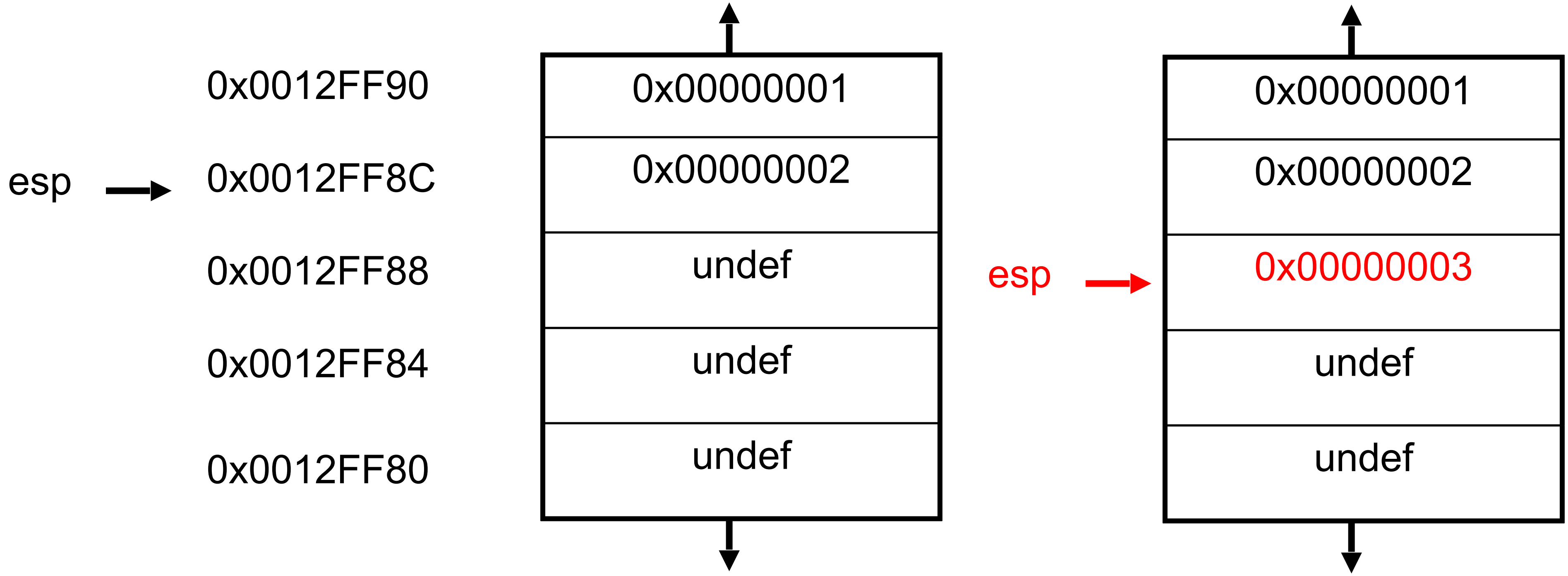
push eax

Registers After

eax	0x00000003
esp	0x0012FF88

Stack Before

Stack After



	Stack Before		Stack After
0x0012FF90	0x00000001		0x00000001
esp → 0x0012FF8C	0x00000002		0x00000002
0x0012FF88	undef	esp →	0x00000003
0x0012FF84	undef		undef
0x0012FF80	undef		undef

3

POP- Pop a Value from the Stack

- Take a DWORD off the stack, put it in a register, and increment esp by 4

Book p. 120

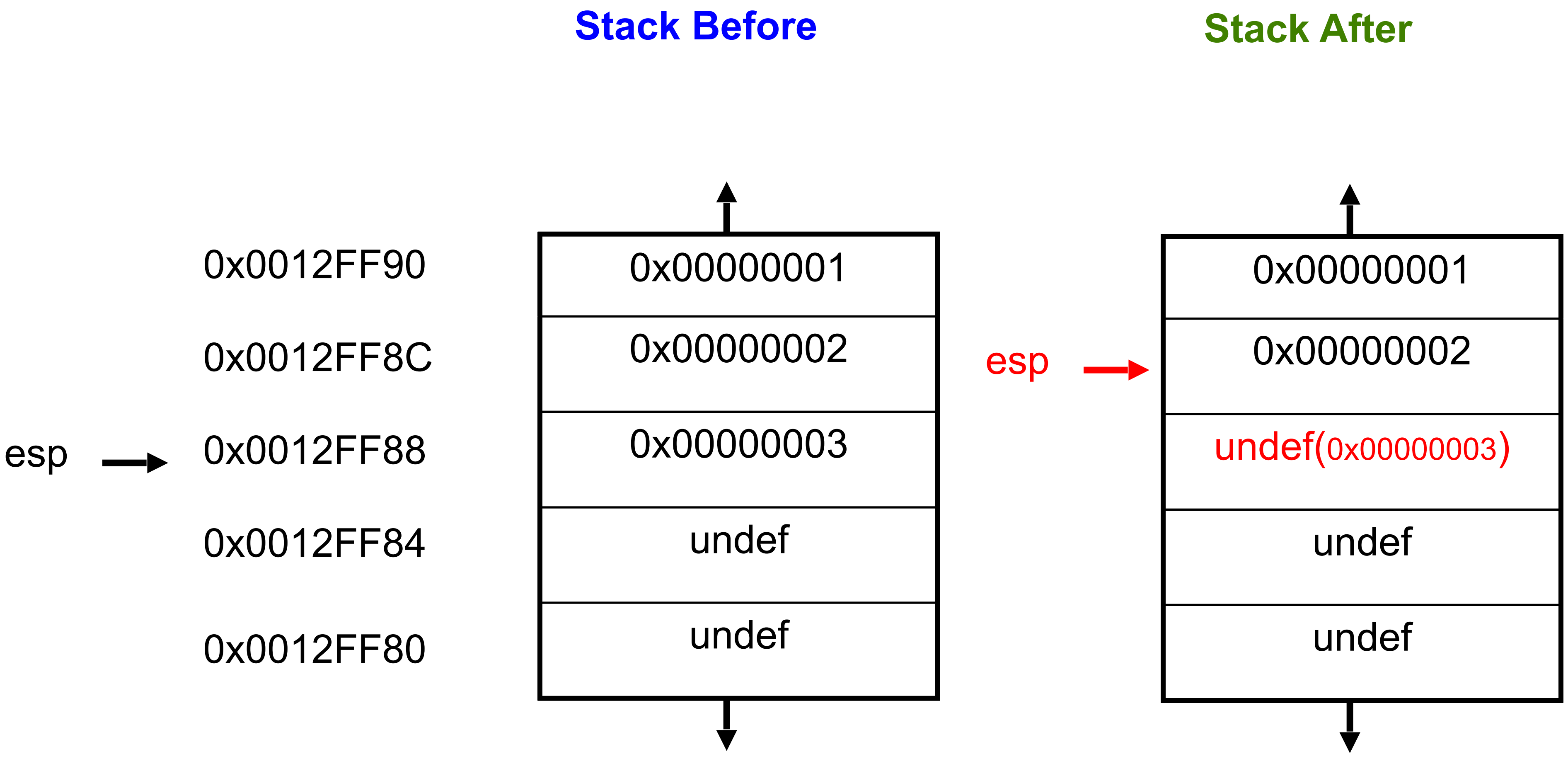
Registers Before

eax	0xFFFFFFFF
esp	0x0012FF88

pop eax

Registers After

eax	0x00000003
esp	0x0012FF8C



Stack Before

Stack After

0x0012FF90

0x0012FF8C

esp → 0x0012FF88

0x0012FF84

0x0012FF80

0x00000001

0x00000002

0x00000003

undef

undef

0x00000001

esp → 0x00000002

undef(0x00000003)

undef

undef

Calling Conventions

- How code calls a subroutine is compiler-dependent and configurable. But there are a few conventions.
- We will only deal with the “cdecl” and “stdcall” conventions.
- More info at
  - http://en.wikipedia.org/wiki/X86_calling_conventions
  - http://www.programmersheaven.com/2/Calling-conventions

Calling Conventions - cdecl

- “C declaration” - most common calling convention
- Function parameters pushed onto stack right to left
- Saves the old stack frame pointer and sets up a new stack frame
- eax or edx:eax returns the result for primitive data types
- Caller is responsible for cleaning up the stack

Calling Conventions - stdcall

- I typically only see this convention used by Microsoft C++ code - e.g. Win32 API
- Function parameters pushed onto stack right to left
- Saves the old stack frame pointer and sets up a new stack frame
- eax or edx:eax returns the result for primitive data types
- Callee responsible for cleaning up any stack parameters it takes
- Aside: typical MS, “If I call my new way of doing stuff 'standard' it must be true!”

4

CALL - Call Procedure

- CALL's job is to transfer control to a different function, in a way that control can later be resumed where it left off
- First it pushes the address of the next instruction onto the stack
  - For use by RET for when the procedure is done
- Then it changes eip to the address given in the instruction
- Destination address can be specified in multiple ways
  - Absolute address
  - Relative address (relative to the end of the instruction)

5

RET - Return from Procedure

- Two forms
  - Pop the top of the stack into eip (remember pop increments stack pointer)
    - In this form, the instruction is just written as “ret”
    - Typically used by cdecl functions
  - Pop the top of the stack into eip and add a constant number of bytes to esp
    - In this form, the instruction is written as “ret 0x8”, or “ret 0x20”, etc
    - Typically used by stdcall functions

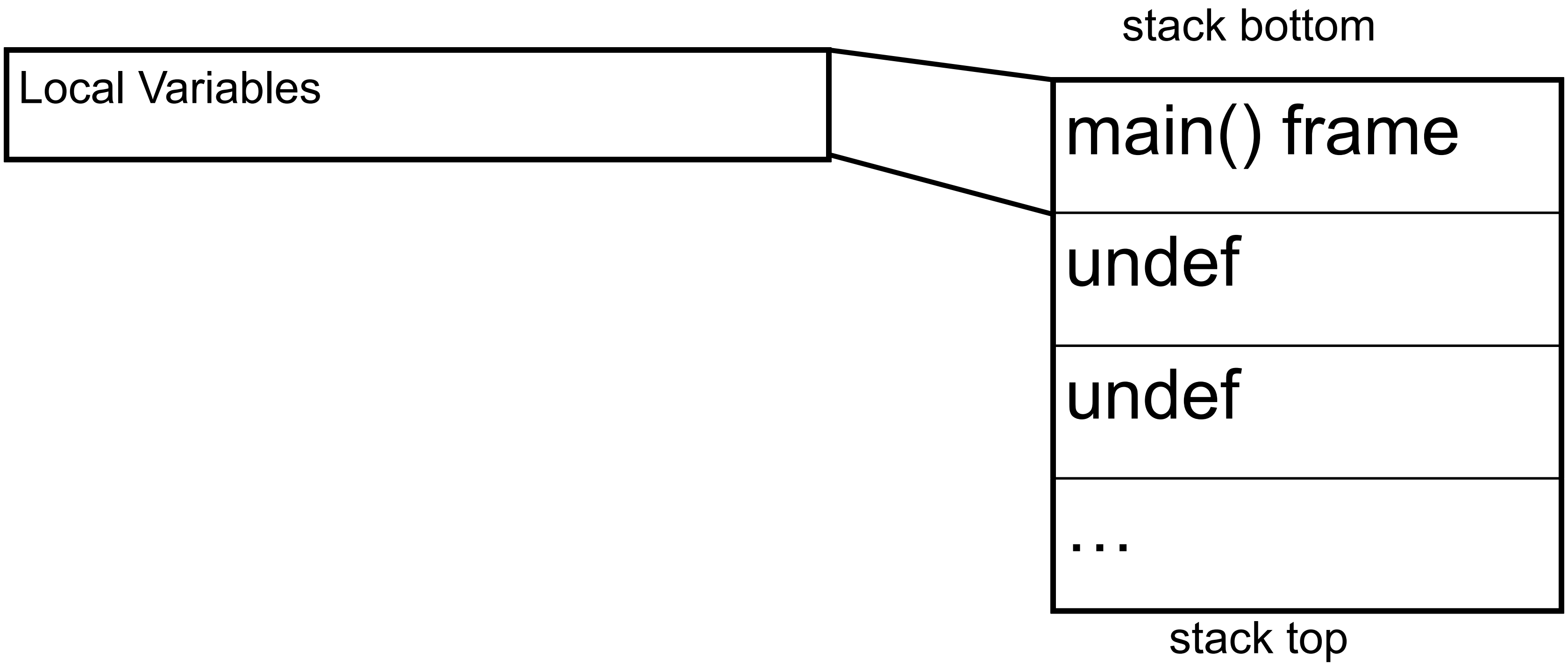
6

MOV - Move

- Can move:
  - register to register
  - memory to register, register to memory
  - immediate to register, immediate to memory
- Never memory to memory!
- Memory addresses are given in r/m32 form talked about later

General Stack Frame Operation

We are going to pretend that main() is the very first function being executed in a program. This is what its stack looks like to start with (assuming it has any local variables).



stack bottom

Local Variables

main() frame

undef

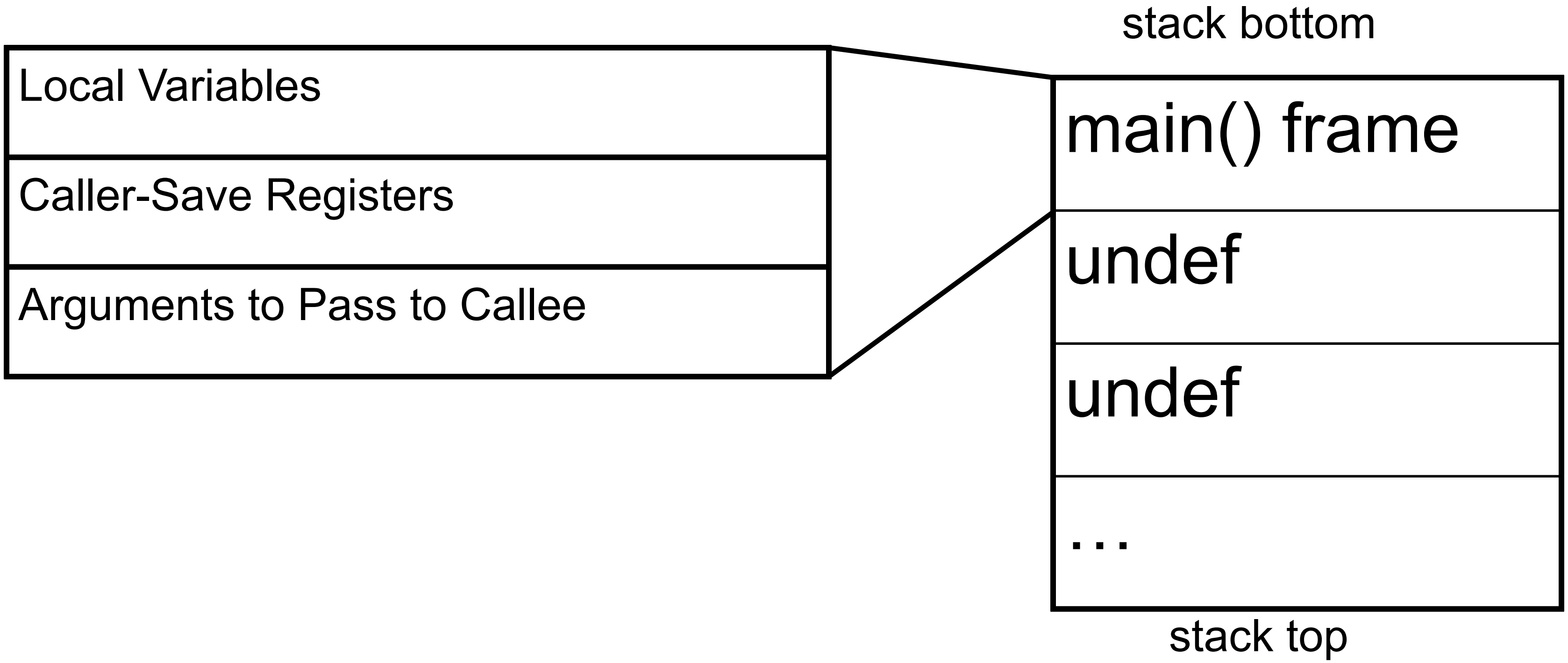
undef

…

stack top

General Stack Frame Operation 2

When main() decides to call a subroutine, main() becomes “the caller”. We will assume main() has some registers it would like to remain the same, so it will save them. We will also assume that the callee function takes some input arguments.



stack bottom

Local Variables

Caller-Save Registers

Arguments to Pass to Callee

main() frame

undef

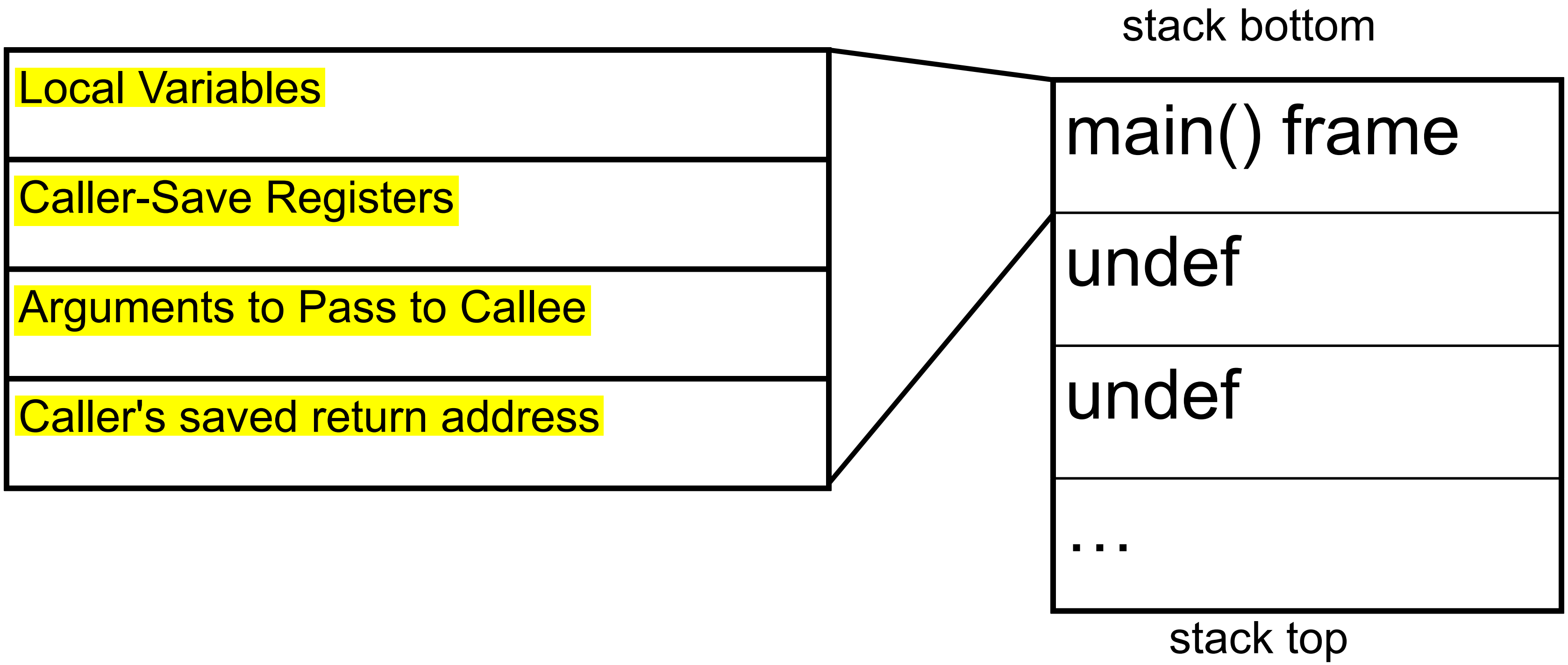
undef

…

stack top

General Stack Frame Operation 3

When main() actually issues the CALL instruction, the return address gets saved onto the stack, and because the next instruction after the call will be the beginning of the called function, we consider the frame to have changed to the callee.



Local Variables

Caller-Save Registers

Arguments to Pass to Callee

Caller's saved return address

stack bottom

main() frame

undef

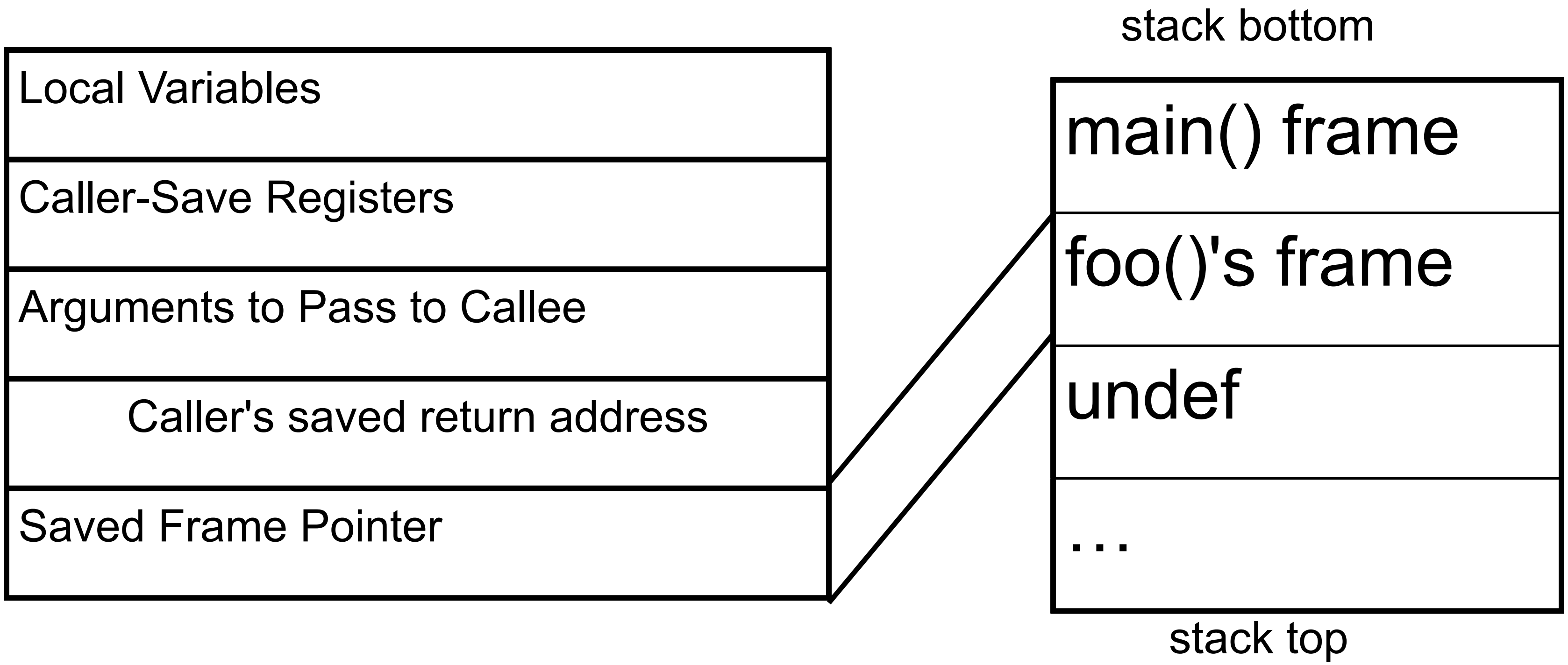
undef

…

stack top

General Stack Frame Operation 4

When foo() starts, the frame pointer (ebp) still points to main()'s frame. So the first thing it does is to save the old frame pointer on the stack and set the new value to point to its own frame.



Local Variables

Caller-Save Registers

Arguments to Pass to Callee

Caller's saved return address

Saved Frame Pointer

stack bottom

main() frame

foo()'s frame

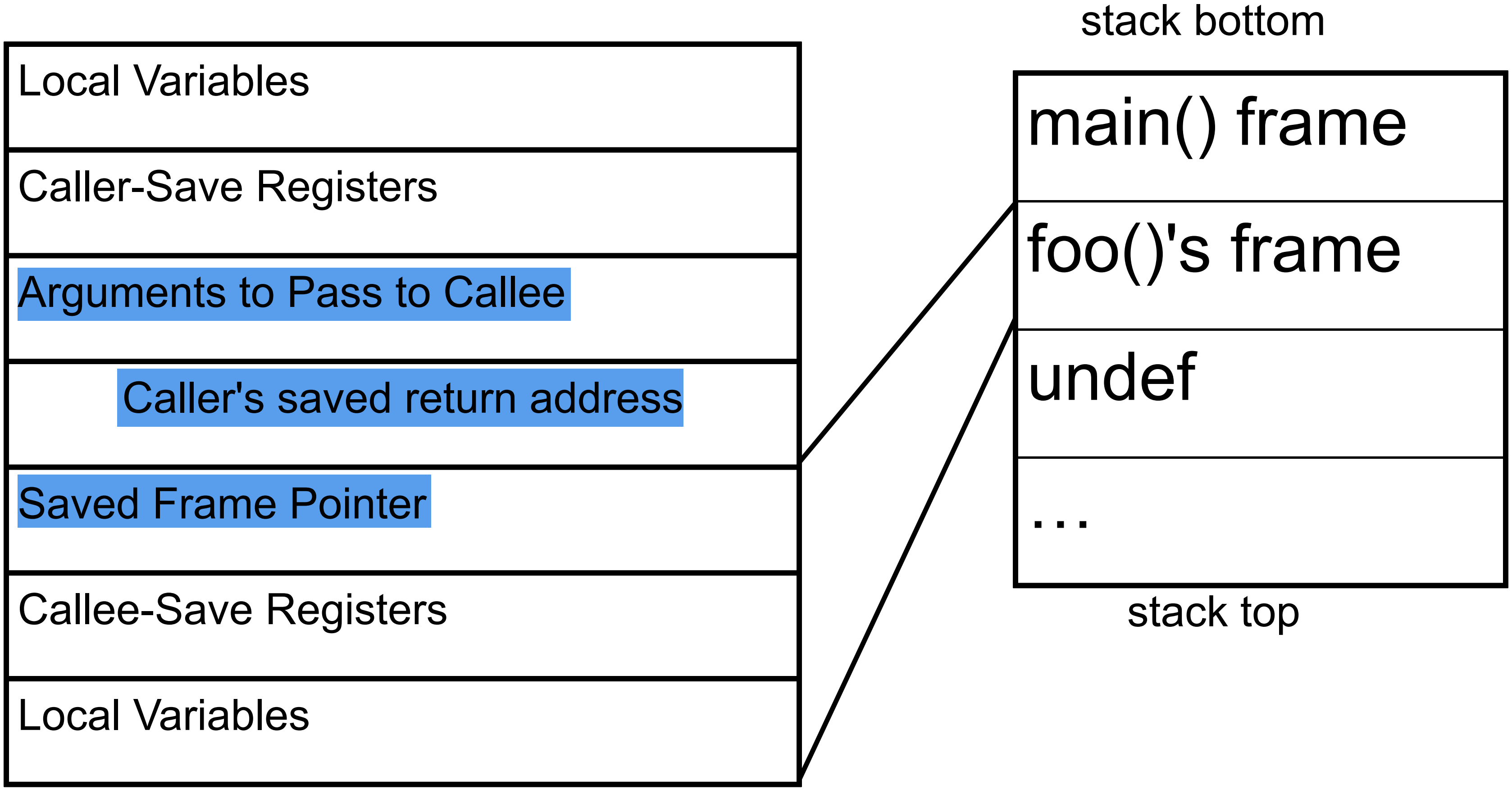
undef

…

stack top

General Stack Frame Operation 5

Next, we'll assume the the callee foo() would like to use all the registers, and must therefore save the callee-save registers. Then it will allocate space for its local variables.



Local Variables

Caller-Save Registers

Arguments to Pass to Callee

Caller's saved return address

Saved Frame Pointer

Callee-Save Registers

Local Variables

stack bottom

main() frame

foo()'s frame

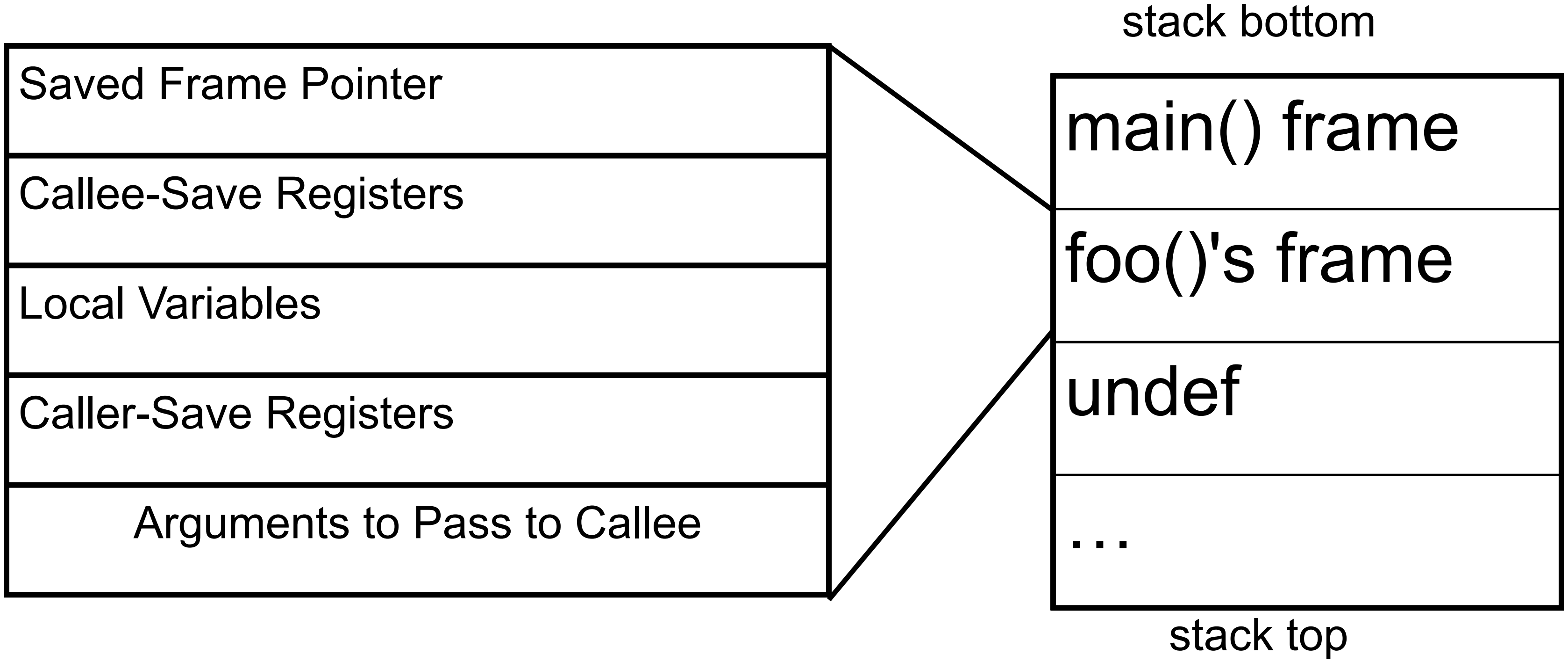
undef

...

stack top

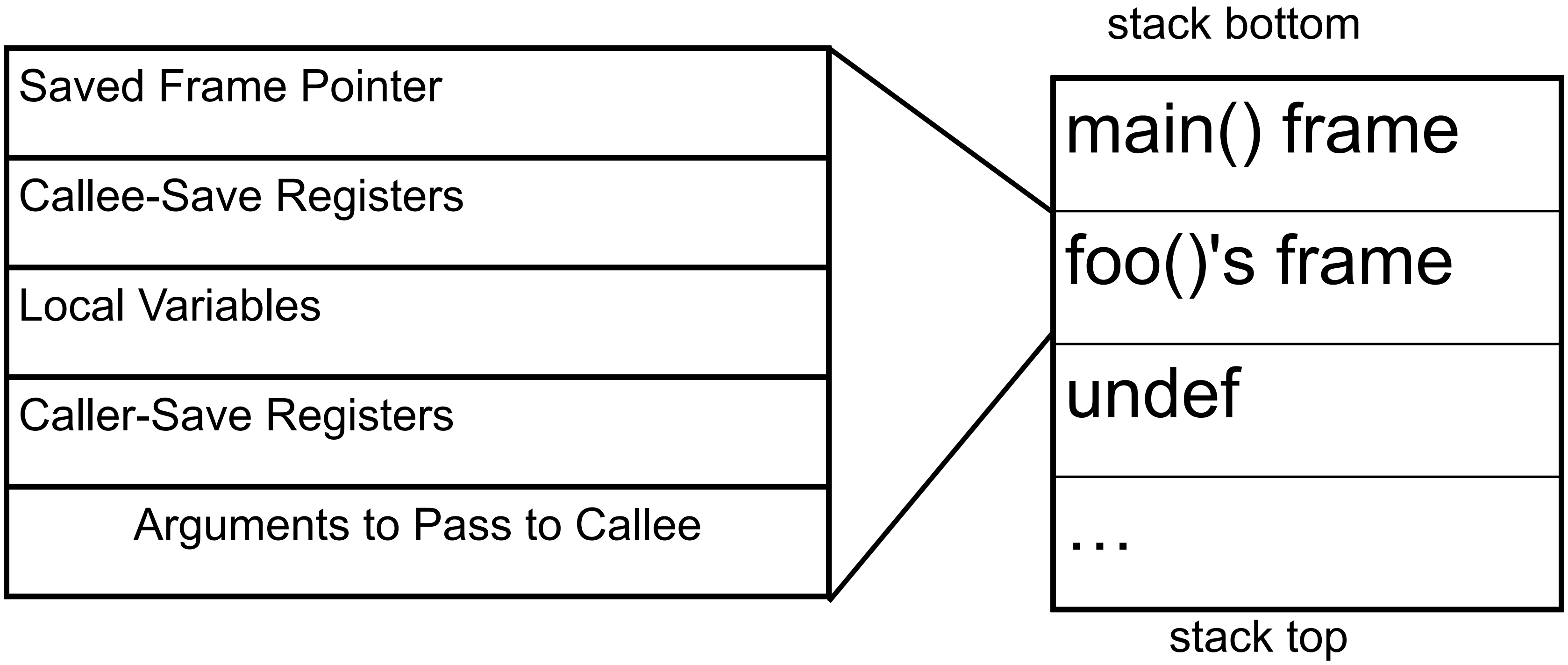
General Stack Frame Operation 6

At this point, foo() decides it wants to call bar(). It is still the callee-of-main(), but it will now be the caller-of-bar. So it saves any caller-save registers that it needs to. It then puts the function arguments on the stack as well.



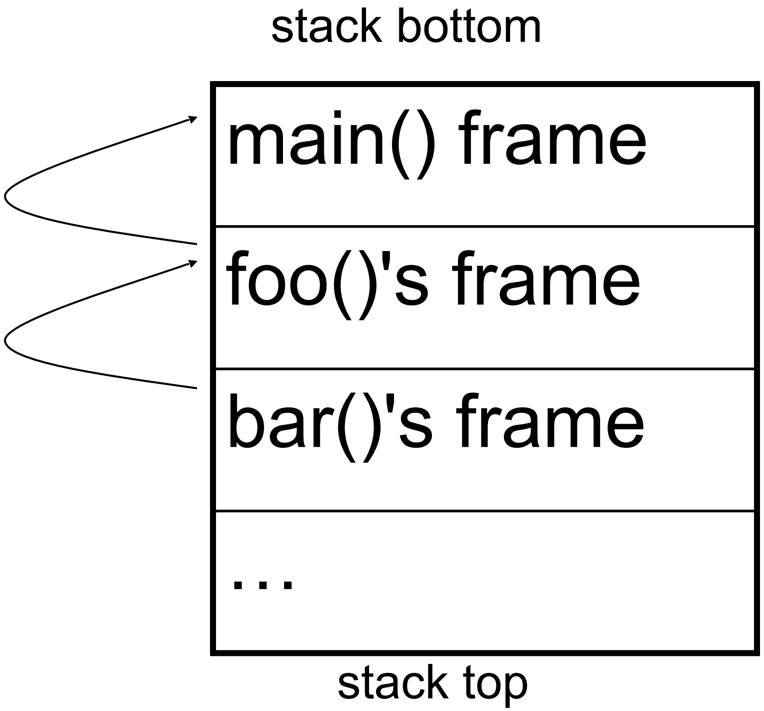
General Stack Frame Layout

Every part of the stack frame is technically optional (that is, you can hand code asm without following the conventions.) But compilers generate code which uses portions if they are needed. Which pieces are used can sometimes be manipulated with compiler options. (E.g. omit frame pointers, changing calling convention to pass arguments in registers, etc.)



Stack Frames are a Linked List!

The ebp in the current frame points at the saved ebp of the previous frame.



"r/m32" Addressing Forms

- Anywhere you see an r/m32 it means it could be taking a value either from a register, or a memory address.
- I'm just calling these “r/m32 forms” because anywhere you see “r/m32” in the manual, the instruction can be a variation of the below forms.
- In Intel syntax, most of the time square brackets [] means to treat the value within as a memory address, and fetch the value at that address (like dereferencing a pointer)
  - mov eax, ebx
  - mov eax, [ebx]
  - mov eax, [ebx+ecx*X] (X=1, 2, 4, 8)
  - mov eax, [ebx+ecx*X+Y] (Y= one byte, 0-255 or 4 bytes, 0-2^32-1)
- Most complicated form is: `[base + index*scale + disp]`

More info: Intel v2a, Section 2.1.5 page 2-4
in particular Tables 2-2 and 2-3

7

LEA - Load Effective Address

- Frequently used with pointer arithmetic, sometimes for just arithmetic in general
- Uses the r/m32 form but **is the exception to the rule** that the square brackets [ ] syntax means dereference (“value at”)
- Example: ebx = 0x2, edx = 0x1000
  - lea eax, [edx+ebx*2]
  - eax = 0x1004, not the value at 0x1004

Not covered in book

8 9

ADD and SUB

- Adds or Subtracts, just as expected
- Destination operand can be r/m32 or register
- Source operand can be r/m32 or register or immediate
- No source ***and*** destination as m32s, because that could allow for memory to memory transfer, which isn't allowed on x86
- Evaluates the operation as if it were on signed AND unsigned data, and sets flags as appropriate. Instructions modify OF, SF, ZF, AF, PF, and CF flags
- add esp, 8
- sub eax, [ebx*2]

Add p. 202, Sub p. 210

Intro x86 Part 2:
More Examples and Analysis

Modified version of slides by
Xeno Kovah – 2009/2010
xkovah at gmail

Approved for Public Release: 10-3348. Distribution Unlimited

Control Flow

- Two forms of control flow
  - Conditional - go somewhere if a condition is met. Think “if”s, switches, loops
  - Unconditional - go somewhere no matter what. Procedure calls, goto, exceptions, interrupts.
- We’ve already seen procedure calls manifest themselves as push/call/ret, let’s see how goto manifests itself in asm.

Example2.999repeating.c:

(I missed this when I reordered slides and then didn't want to change everything else again. Also, VS orders projects alphabetically, otherwise I would have just called it GotoExample.c. Say 'lah vee' :P)

```
//Goto example
#include <stdio.h>
int main(){
		goto mylabel;
		printf("skipped\n");
mylabel:
		printf("goto ftw!\n");
		return 0xf00d;
}
```

```
00401010  push      ebp
00401011  mov       ebp,esp
00401013  jmp       00401023
00401015  push      405000h
0040101A  call      dword ptr ds:[00406230h]
00401020  add       esp,4
mylabel:
00401023  push      40500Ch
00401028  call      dword ptr ds:[00406230h]
0040102E  add       esp,4
00401031  mov       eax,0F00Dh
00401036  pop       ebp
00401037  ret
```

10

JMP - Jump

- Change eip to the given address
- Main forms of the address
  - Short relative (1 byte displacement from end of the instruction)
    - "jmp 00401023" doesn't have the number 00401023 anywhere in it, it's really "jmp 0x0E bytes forward"
    - Some disassemblers will indicate this with a mnemonic by writing it as "jmp short"
  - Near relative (4 byte displacement from current eip)
  - Absolute (hardcoded address in instruction)
  - Absolute Indirect (address calculated with r/m32)
- jmp -2 == infinite loop for short relative jmp :)

Book p. 129

Example3.c

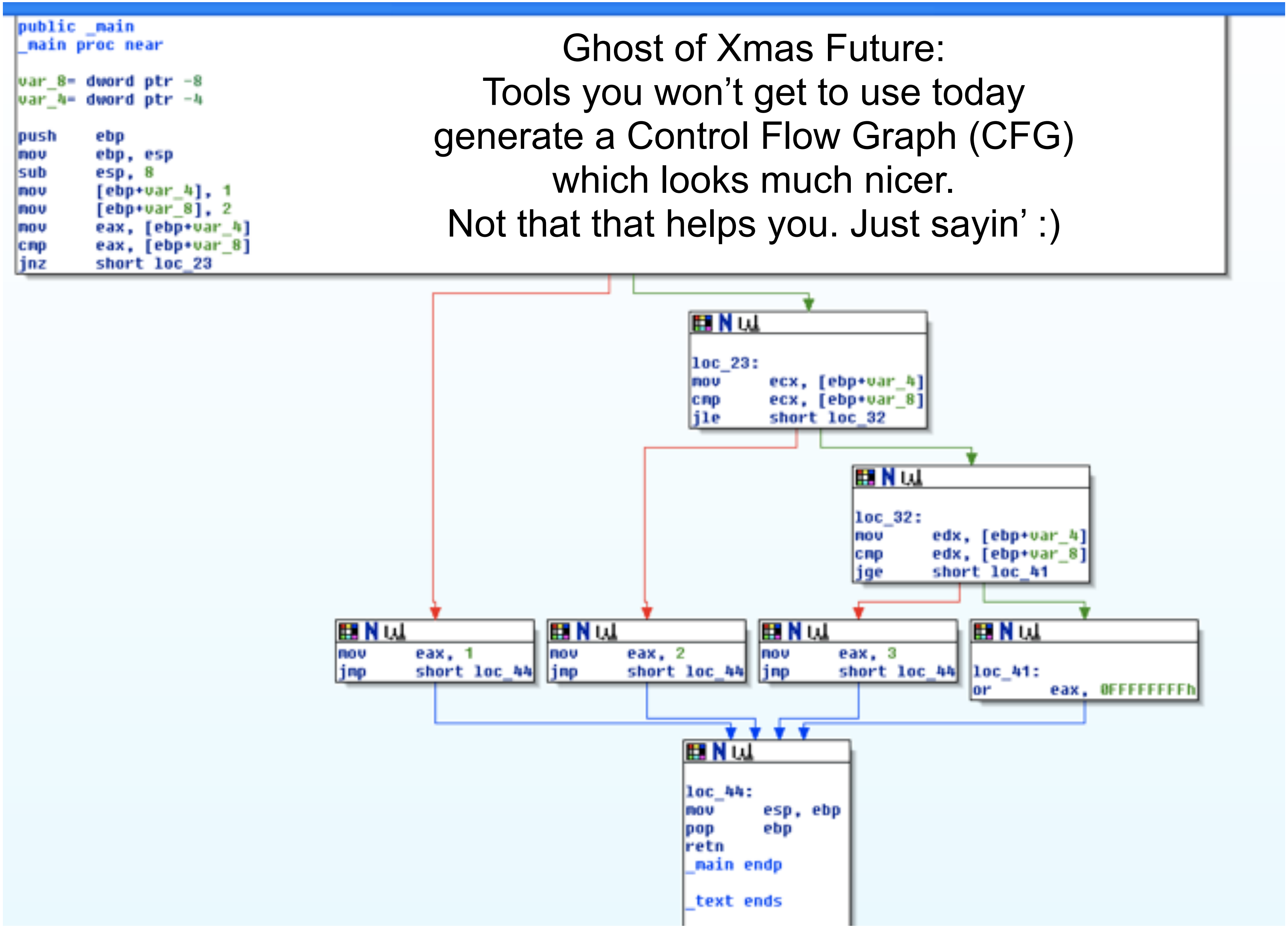
(Remain calm)

```
int main(){
        int a=1, b=2;
        if(a == b){
                return 1;
        }
        if(a > b){
                return 2;
        }
        if(a < b){
                return 3;
        }
        return 0xdefea7;
}
```

Jcc

```
main:
00401010  push      ebp
00401011  mov       ebp,esp
00401013  sub       esp,8
00401016  mov       dword ptr [ebp-4],1
0040101D  mov       dword ptr [ebp-8],2
00401024  mov       eax,dword ptr [ebp-4]
00401027  cmp       eax,dword ptr [ebp-8]
0040102A  jne       00401033
0040102C  mov       eax,1
00401031  jmp       00401056
00401033  mov       ecx,dword ptr [ebp-4]
00401036  cmp       ecx,dword ptr [ebp-8]
00401039  jle       00401042
0040103B  mov       eax,2
00401040  jmp       00401056
00401042  mov       edx,dword ptr [ebp-4]
00401045  cmp       edx,dword ptr [ebp-8]
00401048  jge       00401051
0040104A  mov       eax,3
0040104F  jmp       00401056
00401051  mov       eax,0DEFEA7h
00401056  mov       esp,ebp
00401058  pop       ebp
00401059  ret
```

Ghost of Xmas Future:
Tools you won’t get to use today
generate a Control Flow Graph (CFG)
which looks much nicer.
Not that that helps you. Just sayin’ :)



```
public _main
_main proc near

var_8= dword ptr -8
var_4= dword ptr -4

push    ebp
mov     ebp, esp
sub     esp, 8
mov     [ebp+var_4], 1
mov     [ebp+var_8], 2
mov     eax, [ebp+var_4]
cmp     eax, [ebp+var_8]
jnz     short loc_23
```

```
loc_23:
mov     ecx, [ebp+var_4]
cmp     ecx, [ebp+var_8]
jle     short loc_32
```

```
loc_32:
mov     edx, [ebp+var_4]
cmp     edx, [ebp+var_8]
jge     short loc_41
```

```
mov     eax, 1
jmp     short loc_44
```

```
mov     eax, 2
jmp     short loc_44
```

```
mov     eax, 3
jmp     short loc_44
```

```
loc_41:
or      eax, 0FFFFFFFFh
```

```
loc_44:
mov     esp, ebp
pop     ebp
retn
_main endp

_text ends
```

11

Jcc - Jump If Condition Is Met

- There are more than 4 pages of conditional jump types! Luckily a bunch of them are synonyms for each other.
- JNE == JNZ (Jump if not equal, Jump if not zero, both check if the Zero Flag (ZF) == 0)

Book p. 137

Some Notable Jcc Instructions

- JZ/JE: if ZF == 1
- JNZ/JNE: if ZF == 0
- JLE/JNG : if ZF == 1 or SF != OF
- JGE/JNL : if SF == OF
- JBE: if CF == 1 OR ZF == 1
- JB: if CF == 1
- Note: Don’t get hung up on memorizing which flags are set for what. More often than not, you will be running code in a debugger, not just reading it. In the debugger you can just look at eflags and/or watch whether it takes a jump.

Flag setting

- Before you can do a conditional jump, you need something to set the condition flags for you.
- Typically done with CMP, TEST, or whatever instructions are already inline and happen to have flag-setting side-effects

12

CMP - Compare Two Operands

- "The comparison is performed by subtracting the second operand from the first operand and then setting the status flags in the same manner as the SUB instruction."
- What's the difference from just doing SUB? Difference is that with SUB the result has to be stored somewhere. With CMP the result is computed, the flags are set, but the result is discarded. Thus this only sets flags and doesn't mess up any of your registers.
- Modifies CF, OF, SF, ZF, AF, and PF
- (implies that SUB modifies all those too)

Book p. 138

13

TEST - Logical Compare

- “Computes the bit-wise logical AND of first operand (source 1 operand) and the second operand (source 2 operand) and sets the SF, ZF, and PF status flags according to the result.”
- Like CMP - sets flags, and throws away the result

Book p. 232

14

AND - Logical AND

- Destination operand can be r/m32 or register
- Source operand can be r/m32 or register or immediate (No source *and* destination as r/m32s)

and al, bl

	00110011b (al - 0x33)
AND	01010101b (bl - 0x55)
result	00010001b (al - 0x11)

and al, 0x42

	00110011b (al - 0x33)
AND	01000010b (imm - 0x42)
result	00000010b (al - 0x02)

Book p. 231

15

OR - Logical Inclusive OR

- Destination operand can be r/m32 or register
- Source operand can be r/m32 or register or immediate (No source *and* destination as r/m32s)

or al, bl

	00110011b (al - 0x33)
OR	01010101b (bl - 0x55)
result	01110111b (al - 0x77)

or al, 0x42

	00110011b (al - 0x33)
OR	01000010b (imm - 0x42)
result	01110011b (al - 0x73)

Book p. 231

16

XOR - Logical Exclusive OR

- Destination operand can be r/m32 or register
- Source operand can be r/m32 or register or immediate (No source *and* destination as r/m32s)

xor al, al

	00110011b (al - 0x33)
XOR	00110011b (al - 0x33)
result	00000000b (al - 0x00)

xor al, 0x42

	00110011b (al - 0x33)
OR	01000010b (imm - 0x42)
result	01110001b (al - 0x71)

XOR is commonly used to zero a register, by XORing it with itself, because it's faster than a MOV

Book p. 231

17

NOT - One's Complement Negation

- Single source/destination operand can be r/m32

not al

NOT	00110011b (al - 0x33)
result	11001100b (al - 0xCC)

not [al+bl]

al	0x10000000
bl	0x00001234
al+bl	0x10001234
[al+bl]	0 (assumed memory at 0x10001234)
NOT	00000000b
result	11111111b

Book p. 231

Example5.c - simple for loop

```
#include <stdio.h>

int main(){
   int i;
   for(i = 0; i < 10; i++){
      printf("i = %d\n“, i);
   }
}
```

```
main:
00401010  push     ebp
00401011  mov      ebp,esp
00401013  push     ecx
00401014  mov      dword ptr [ebp-4],0
0040101B  jmp      00401026
0040101D  mov      eax,dword ptr [ebp-4]
00401020  add      eax,1
00401023  mov      dword ptr [ebp-4],eax
00401026  cmp      dword ptr [ebp-4],0Ah
0040102A  jge      00401040
0040102C  mov      ecx,dword ptr [ebp-4]
0040102F  push     ecx
00401030  push     405000h
00401035  call     dword ptr ds:[00406230h]
0040103B  add      esp,8
0040103E  jmp      0040101D
00401040  xor      eax,eax
00401042  mov      esp,ebp
00401044  pop      ebp
00401045  ret
```

What does this add say about the calling convention of printf()?

Interesting note: Defaults to returning 0

Instructions we now know(17)

- NOP
- PUSH/POP
- CALL/RET
- MOV/LEA
- ADD/SUB
- JMP/Jcc
- CMP/TEST
- AND/OR/XOR/NOT

Example6.c

```
//Multiply and divide transformations
//New instructions:
//shl - Shift Left, shr - Shift Right

int main(){
        unsigned int a, b, c;
        a = 0x40;
        b = a * 8;
        c = b / 16;
        return c;
}
```

```
main:
 push      ebp
 mov       ebp,esp
 sub       esp,0Ch
 mov       dword ptr [ebp-4],40h
 mov       eax,dword ptr [ebp-4]
 shl       eax,3
 mov       dword ptr [ebp-8],eax
 mov       ecx,dword ptr [ebp-8]
 shr       ecx,4
 mov       dword ptr [ebp-0Ch],ecx
 mov       eax,dword ptr [ebp-0Ch]
 mov       esp,ebp
 pop       ebp
 ret
```

18

Book p. 224

SHL - Shift Logical Left

- Can be explicitly used with the C “<<” operator
- First operand (source and destination) operand is an r/m32
- Second operand is either cl (lowest byte of ecx), or a 1 byte immediate. The 2nd operand is the number of places to shift.
- It **multiplies** the register by 2 for each place the value is shifted. More efficient than a multiply instruction.
- Bits shifted off the left hand side are “shifted into” (set) the carry flag (CF)
- For purposes of determining if the CF is set at the end, think of it as n independent 1 bit shifts.

shl cl, 2

	00110011b (cl - 0x33)
result	11001100b (cl - 0xCC) CF = 0

shl cl, 3

	00110011b (cl - 0x33)
result	10011000b (cl - 0x98) CF = 1

19

Book p. 225

SHR - Shift Logical Right

- Can be explicitly used with the C “>>” operator
- First operand (source and destination) operand is an r/m32
- Second operand is either cl (lowest byte of ecx), or a 1 byte immediate. The 2nd operand is the number of places to shift.
- It **divides** the register by 2 for each place the value is shifted. More efficient than a multiply instruction.
- Bits shifted off the right hand side are “shifted into” (set) the carry flag (CF)
- For purposes of determining if the CF is set at the end, think of it as n independent 1 bit shifts.

shr cl, 2

	00110011b (cl - 0x33)
result	00001100b (cl - 0x0C) CF = 1

shr cl, 3

	00110011b (cl - 0x33)
result	00000110b (cl - 0x06) CF = 0

Example7.c

```
//Multiply and divide operations
//when the operand is not a
//power of two
//New instructions: imul, div

int main(){
        unsigned int a = 1;
        a = a * 6;
        a = a / 3;
        return 0x2bad;
}
```

```
main:
  push     ebp
  mov      ebp,esp
  push     ecx
  mov      dword ptr [ebp-4],1
  mov      eax,dword ptr [ebp-4]
★ imul     eax,eax,6
  mov      dword ptr [ebp-4],eax
  mov      eax,dword ptr [ebp-4]
  xor      edx,edx
  mov      ecx,3
★ div      eax,ecx
  mov      dword ptr [ebp-4],eax
  mov      eax,2BADh
  mov      esp,ebp
  pop      ebp
  ret
```

20

Book p. 218

IMUL - Signed Multiply

- Wait…what? Weren’t the operands unsigned?
  - Visual Studio seems to have a predilection for imul over mul (unsigned multiply). I haven’t been able to get it to generate the latter for simple examples.
- Three forms. One, two, or three operands
  - imul r/m32 edx:eax = eax * r/m32
  - imul reg, r/m32 reg = reg * r/m32
  - imul reg, r/m32, immediate reg = r/m32 * immediate
- **Three** operands? Only one of it’s kind?(see link in notes)

initial

edx	eax	r/m32(ecx)
0x0	0x44000000	0x4

eax	r/m32(ecx)
0x20	0x4

eax	r/m32(ecx)
0x20	0x4

operation

imul ecx

imul eax, ecx

imul eax, ecx, 0x6

result

edx	eax	r/m32(ecx)
0x1	0x10000000	0x4

eax	r/m32(ecx)
0x80	0x4

eax	r/m32(ecx)
0x18	0x4

21

Book p. 221

DIV - Unsigned Divide

- Two forms
  - Unsigned divide ax by r/m8, al = quotient, ah = remainder
  - Unsigned divide edx:eax by r/m32, eax = quotient, edx = remainder
- If dividend is 32bits, edx will just be set to 0 before the instruction (as occurred in the Example7.c code)
- If the divisor is 0, a divide by zero exception is raised.

initial

ax	r/m8(cx)
0x8	0x3

edx	eax	r/m32(ecx)
0x0	0x8	0x3

operation

div ax, cx

div eax, ecx

result

ah	al
0x2	0x2

edx	eax	r/m32(ecx)
0x1	0x2	0x3

Example8.c

```
//VisualStudio runtime check
//buffer initialization
//auto-generated code
//New instruction: rep stos

int main(){
		char buf[40];
		buf[39] = 42;
		return 0xb100d;
}
```

Example8.c

```
main:
00401010  push      ebp
00401011  mov       ebp,esp
00401013  sub       esp,30h
00401016  push      edi
00401017  lea       edi,[ebp-30h]
0040101A  mov       ecx,0Ch
0040101F  mov       eax,0CCCCCCCCh
00401024  rep stos  dword ptr es:[edi]
00401026  mov       byte ptr [ebp-5],2Ah
0040102A  mov       eax,0B100Dh
0040102F  push      edx
00401030  mov       ecx,ebp
00401032  push      eax
00401033  lea       edx,[ (401048h)]
00401039  call      _RTC_CheckStackVars (4010B0h)
0040103E  pop       eax
0040103F  pop       edx
00401040  pop       edi
00401041  mov       esp,ebp
00401043  pop       ebp
00401044  ret
```

22

REP STOS - Repeat Store String

- One of a family of “rep” operations, which repeat a single instruction multiple times. (i.e. “stos” is also a standalone instruction)
  - Rep isn’t technically it’s own instruction, it’s an instruction prefix
- All rep operations use ecx register as a “counter” to determine how many times to loop through the instruction. Each time it executes, it decrements ecx. Once ecx == 0, it continues to the next instruction.
- Either moves one byte at a time or one dword at a time.
- Either fill byte at [edi] with al or fill dword at [edi] with eax.
- Moves the edi register forward one byte or one dword at a time, so that the repeated store operation is storing into consecutive locations.
- So there are 3 pieces which must happen before the actual rep stos occurs: set edi to the start destination, eax/al to the value to store, and ecx to the number of times to store

Book p. 284

rep stos setup

```
004113AC  lea         edi,[ebp-0F0h]
```

Set edi - the destination

```
004113B2  mov         ecx,3Ch
```

Set ecx - the count

```
004113B7  mov         eax,0CCCCCCCCh
```

Set eax - the value

- `004113BC  rep stos    dword ptr es:[edi]`
  - **Start the repeated store**
  - So what's this going to do? Store 0x3C copies of the dword 0xCCCCCCCC starting at ebp-0xF0
  - And that just happens to be 0xF0 bytes of 0xCC!