Lec 3: Assembly

IS561: Binary Code Analysis and Secure Software Systems

Sang Kil Cha

Intel (x86) Architecture

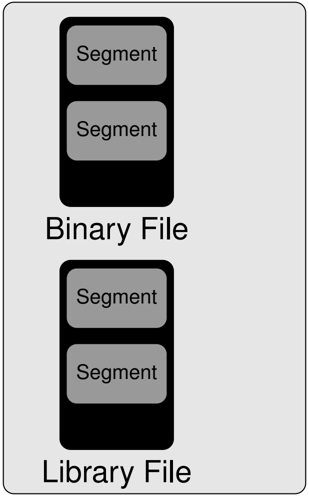
x86 Instruction Set Architecture

1. Introduced by ***Intel*** in 1978.
2. CISC (Complex Instruction Set Computer) architecture.
3. The most popular ISA.

History of x86 ISA

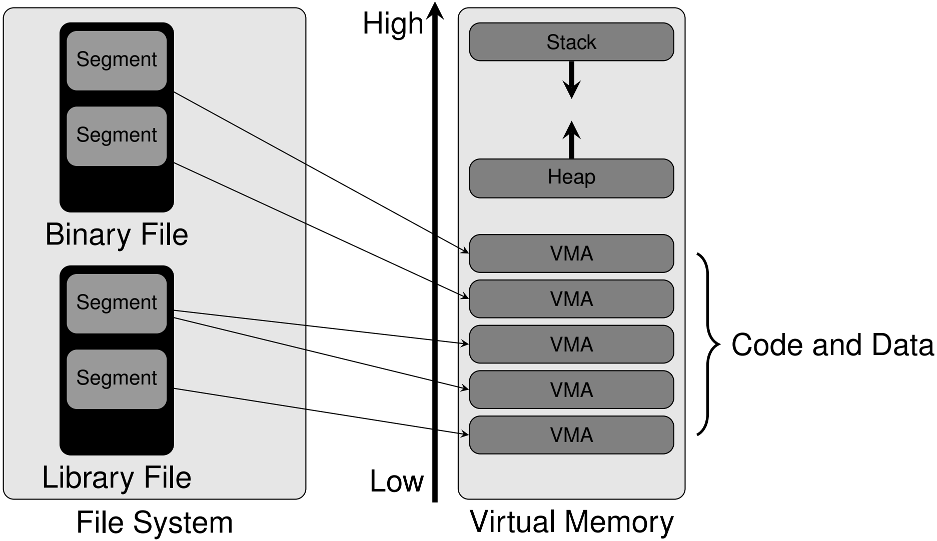
- (***8086***) 16-bit address space (in 1978).
- (***x86*** or ***IA-32***) 32-bit address space (in 1985).
- (***x86-64*** or ***x64*** or ***AMD64***) 64-bit address space (in 2003).

Memory Layout and CPU Registers

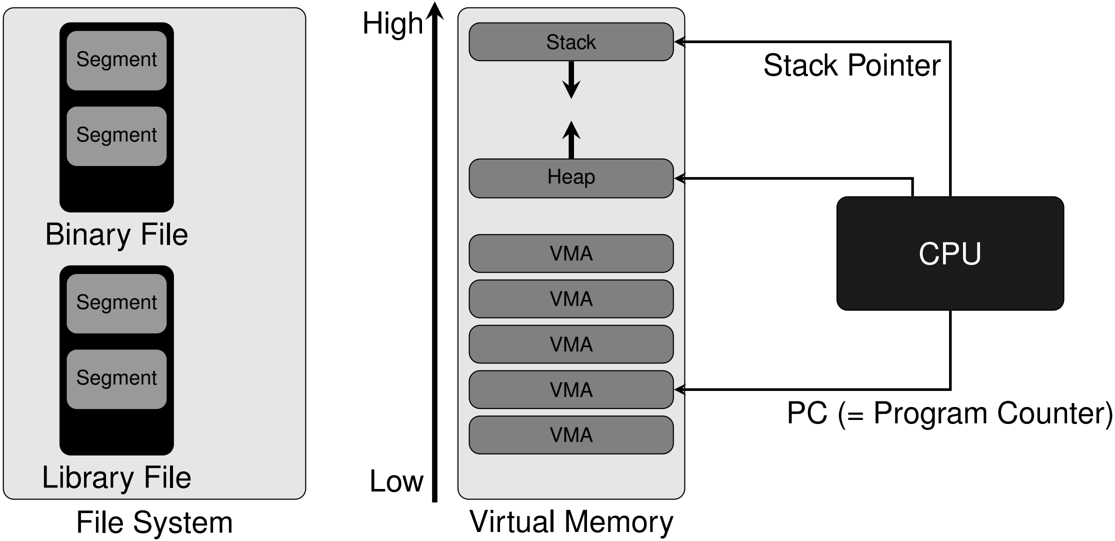


File System

Memory Layout and CPU Registers



Memory Layout and CPU Registers



Registers in x86 (and x86-64)

- Stack pointers
  - ESP (RSP) points to the top of the stack.
  - EBP (RBP) points to the base of the current stack frame.
- Program counter (instruction pointer)
  - EIP (RIP) points to the instruction to execute.
- Status register (FLAGS register)
  - EFLAGS (RFLAGS) contains the current condition flags.
- Other registers
  - EAX (RAX), EBX (RBX), ECX (RCX), EDX (RDX), ESI (RSI), EDI (RDI)
  - x86-64 only registers: R8, R9, R10, R11, R12, R13, R14, R15

Size of Registers

- x86 registers are 32-bit. Intel says they a size of a ***double word***.
- x86-64 registers are 64-bit. Intel says they have a size of a ***quad word***.

Double/Quad Word?

- A word is the natural ***unit*** of data used by a processor.
- Typically, a word is 32 bits on a 32-bit machine, and 64 bits on a 64-bit machine.
- However, Intel says a word is 16 bits on both x86 and x86-64!

What's wrong?

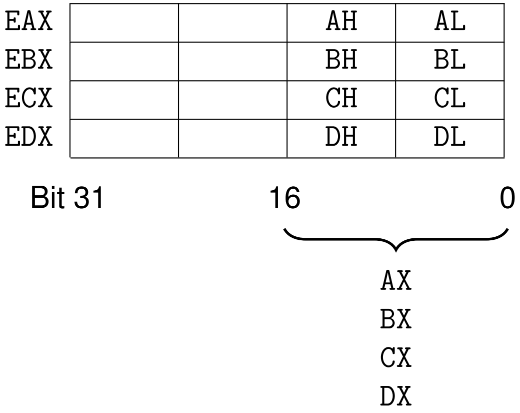
History of x86 Processors

1978: 8086 1982: 80286	16-bit processor, registers (`SP`, `BP`, `IP`, ...)
1985: 80386 1989: 80486 ...	32-bit processor, registers (`ESP`, `EBP`, `EIP`, ...)
2003: Opteron 2005: Prescott 2006: Core 2 ...	64-bit processor, registers (`RSP`, `RBP`, `RIP`, ...)

Intel x86 Convention

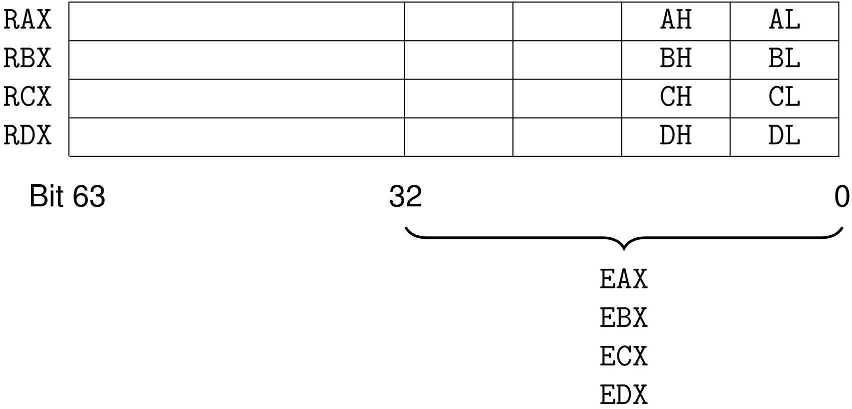
- Word = 16 bits.
- Double Word (DWORD) = 32 bits.
- Quad Word (QWORD) = 64 bits.

x86 (32-bit) Registers



EAX			AH	AL
EBX			BH	BL
ECX			CH	CL
EDX			DH	DL

x86-64 (64-bit) Registers



RAX				AH	AL
RBX				BH	BL
RCX				CH	CL
RDX				DH	DL

Bit 63 32 0

EAX
EBX
ECX
EDX

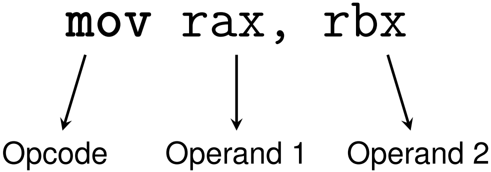
x86 Memory Access = Byte Addressing

We can access data at a byte granualrity.

How do we load/store a single bit then?

Assembly Basic

Basic Format 1: Instructions with 2 Operands



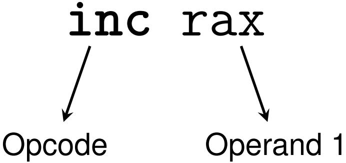
`mov rax, rbx`

Opcode

Operand 1

Operand 2

Basic Format 2: Instructions with 1 Operand

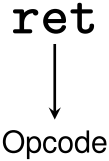


```
inc rax
```

Opcode

Operand 1

Basic Format 3: Instructions with 0 Operand



ret

Opcode

Two Kinds of x86 Assembly Syntax

- ***Intel*** Syntax: the original assembly syntax introduced by Intel.
- ***AT&T*** Syntax: used by UNIX and Linux.

We will use the Intel syntax.

Opcode Decides Semantics

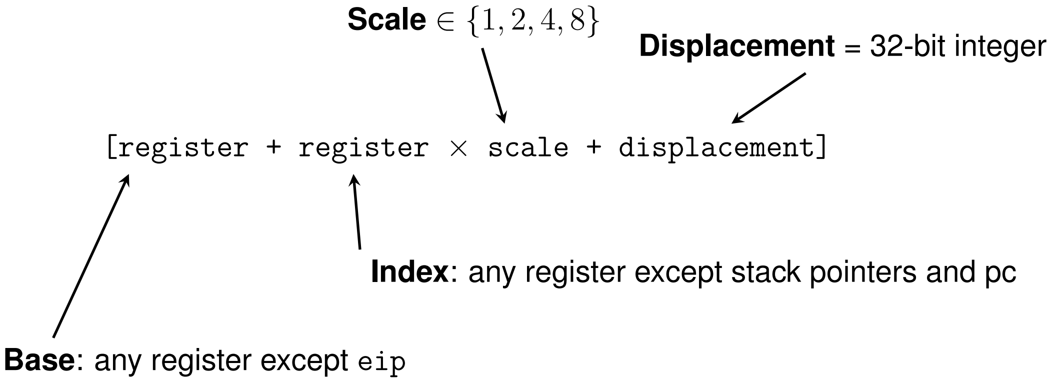
- `mov rax, rbx`
- `sub rsp, 0x8`
- `inc eax`

Addressing Modes

An addressing mode defines how a memory operand is interpreted to derive an effective address.

- register
  - mov rax, [rax]
- register + register
  - mov rax, [rax + rbx] (= mov rax, [rax + rbx * 1])
- displacement
  - mov rax, [0x1000]
- register + register × scale + displacement
  - mov rax, [rax + rbx * 4 + 0x1000]

Addressing Modes (cont'd)



Scale $\in \{1, 2, 4, 8\}$

Displacement = 32-bit integer

`[register + register` $\times$ `scale + displacement]`

Index: any register except stack pointers and pc

Base: any register except `eip`

Intel vs. AT&T Syntax

What's the AT&T representation of
`mov rax, [rax + rbx * 4 + 0x1000]`?

Intel vs. AT&T Syntax

What's the AT&T representation of `mov rax, [rax + rbx * 4 + 0x1000]`?

Answer: `mov 0x1000(%rax, %rbx, 4), %rax`

So which syntax would you like to use?

Writing Assembly with GNU AS

GNU AS (GNU Assembler) uses the AT&T syntax by default. To use the Intel syntax, you should use a special directive `.intel_syntax noprefix`.

Example

```
.intel_syntax noprefix
mov rax, [rbx]
```

Pointer Directives

```
mov [rsi], al         ; ok (compiles)
mov [rsi], 1          ; error
```

Error: ambiguous operand size for ‘mov’

Pointer Directives

```
mov [rsi], al           ; ok (compiles)
mov [rsi], 1            ; error
```

Error: ambiguous operand size for ‘mov’

Because it could be any of the followings

- `mov BYTE PTR [rsi], 1`
- `mov WORD PTR [rsi], 1`
- `mov DWORD PTR [rsi], 1`
- `mov QWORD PTR [rsi], 1`

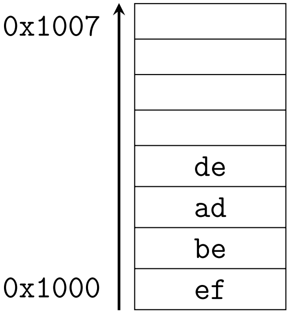
Examples: Moving Data Around

- `mov eax, ebx`
- `mov al, bl`
- `mov [rax], rbx`
- `mov rcx, [rbx]`
- `mov rcx, [rax + rbx * 4]`
- `mov al, BYTE PTR [rsi]`
- `mov rax, 42`
- `mov DWORD PTR [rax], 42`

Storing a DWORD in Memory

`mov DWORD PTR [rax], 0xdeadbeef`

(assume that `rax = 0x1000`)



0x1007

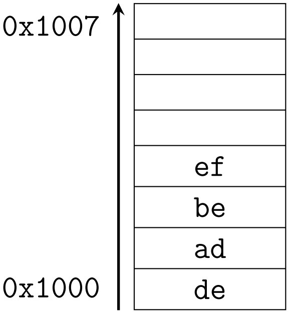
de

ad

be

0x1000 ef

vs.



0x1007

ef

be

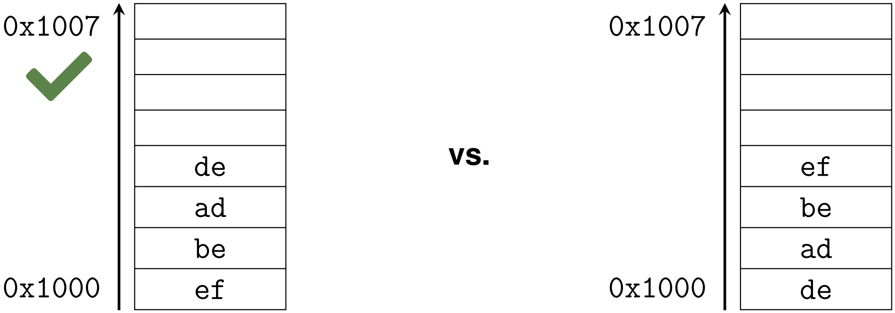
ad

0x1000 de

Storing a DWORD in Memory

`mov DWORD PTR [rax], 0xdeadbeef`

(assume that `rax = 0x1000`)



0x1007

✓

de

ad

be

0x1000 ef

vs.

0x1007

ef

be

ad

0x1000 de

Endianness

The order in which a sequence of bytes are stored in memory.

- ***Big endian***: The MSB goes to the lowest address.
- ***Little endian***: The LSB goes to the lowest address.

x86 uses Little Endian

Load Effective Address

- `lea rax, [rbx]`
- `lea rax, [rbp - 0x8]`

Same as ‘&’ in C

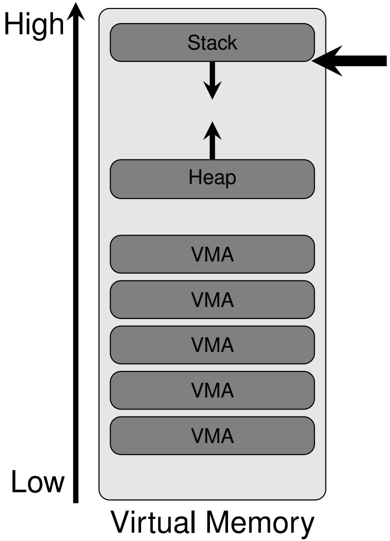
What’s the Difference?

`mov rax, [rbp + 0x10]` vs. `lea rax, [rbp + 0x10]`

Stack Memory

Stack stores data in a LIFO (Last-In-First-Out) fashion. When a function is invoked, a new ***stack frame*** is allocated at the top of the stack memory.

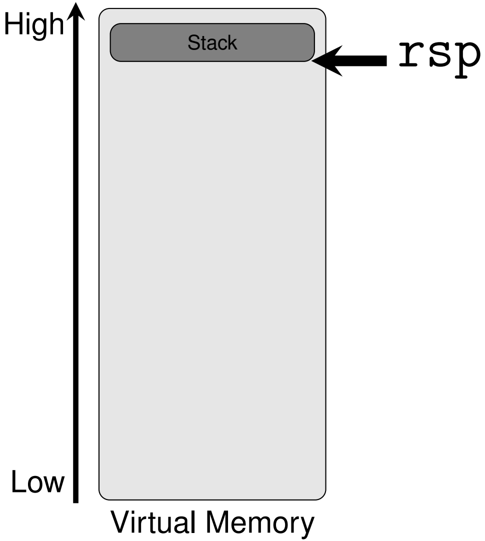
Stack Operations



`rsp` points to the top of the stack

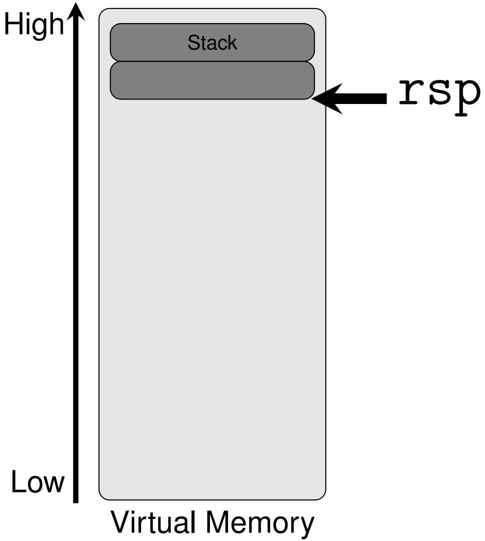
Stack grows ***backward*** (from high to low)

Stack Operations



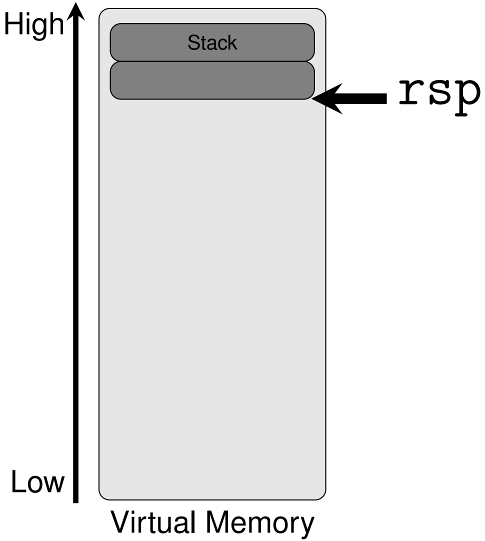
push

Stack Operations



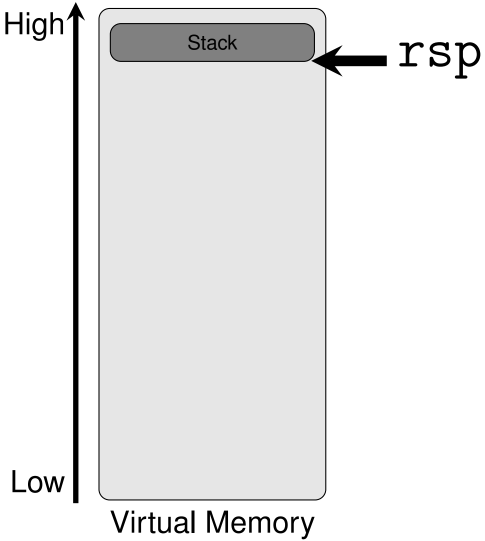
`push`

Stack Operations



`pop`

Stack Operations



`pop`

Stack Push/Pop

- (On x86): `push x` = `sub esp, 4; mov [esp], x`
- (On x64): `push x` = `sub rsp, 8; mov [rsp], x`

- (On x86): `pop x` = `mov x, [esp]; add esp, 4`
- (On x64): `pop x` = `mov x, [rsp]; add rsp, 8`

Stack Enter/Leave (x64)

```
                push rbp
enter x, 0  =   mov rbp, rsp
                sub rsp, x

                mov rsp, rbp
leave       =   pop ebp
```

Function Call

```
.intel_syntax noprefix
call foo
nextret:
nop
nop
nop
nop
foo:
nop
nop
nop
nop
```

=

```
.intel_syntax noprefix
push nextret
jmp foo
nextret:
nop
nop
nop
nop
foo:
nop
nop
nop
nop
```

Function Return

`ret` = `pop rip`

Arithmetic and Logical Operations

- `add rax, [rbx]`
- `sub rsp, 0x40`
- `inc rcx`
- `dec rcx`
- `and [rax + rcx], rbx`
- `or rdx, rbx`
- `xor rdx, rbx`
- `shl rax, 1`
- ...

Control Flows

C has a high-level control structures, such as:

```
if ( x ) { /* A */ }
else { /* B */ }

while ( x ) { }

for ( i = 0; i < n; i++ ) { }
```

Can we represent these in assembly?

Control Flows in Assembly (1)

There are only "`if`" and "`goto`" (no "`else`").

```
if ( x ) { /* A */ }
else { /* B */ }
```

→

```
if (!x) goto F;
/* A */
goto E;
F:
/* B */
E:
```

Control Flows in Assembly (2)

There are only “`if`” and “`goto`” (no “`else`”).

```
while ( x ) { /* body */ }
```

→

```
WHILE:
if ( !x ) goto DONE;
/* body */
goto WHILE;
DONE:
```

Control Flows in Assembly (3)

There are only “`if`” and “`goto`” (no “`else`”).

```
for ( i = 0; i < n; i++ ) {
  /* body */
}
```

→

```
i = 0;
LOOP:
if ( i >= n ) goto DONE;
/* body */
i++;
goto LOOP;
DONE:
```

Control Flows in Assembly (Example)

```
if (!x) goto F;
/* A */
goto E;
F:
/* B */
E:
```

→

```
cmp x, 0
jne F
; A
jmp E
F:
; B
E:
```

Control Flows in Assembly (Example)

```
if (!x) goto F;
/* A */
goto E;
F:
/* B */
E:
```

→

```
cmp x, 0
jne F
; A
jmp E
F:
; B
E:
```

Where do we store the result of comparison (`cmp`)?

EFLAGS: Storing the Processor State

- EFLAGS is a status register used in x86, which is essentially a collection of status flag bits.
- There are approximately 20 different flag bits used in x86, but we are mainly interested in 6 condition flags:
  - `OF`: Overflow flag
  - `SF`: Sign flag
  - `ZF`: Zero flag
  - `AF`: Auxiliary carry flag
  - `PF`: Parity flag
  - `CF`: Carry flag

Almost Every x86 Instruction Affects EFLAGS

```
add rax, rbx
```

rax	1
rbx	-2 (0xfffffffffffffffe)
SF	0

→

rax	-1 (0xffffffffffffffff)
rbx	-2 (0xfffffffffffffffe)
SF	1

```
and rbx, 0
```

rax	-1 (0xffffffffffffffff)
rbx	-2 (0xfffffffffffffffe)
SF	1
ZF	0

→

rax	-1 (0xffffffffffffffff)
rbx	0
SF	0
ZF	1

`cmp` Only Affects EFLAGS

`cmp` is the same as `sub`, except that it only affects EFLAGS, but not the destination operand. For example, `cmp rax, rbx` will not change the `rax` register.

Conditional Branch Instructions

Instruction[1]	Condition	Description
`ja`	`CF = 0` and `ZF = 0`	Jump if above
`jb`	`CF = 1`	Jump if below
`je`	`ZF = 1`	Jump if equal
`jl`	`SF` ≠ `OF`	Jump if less
`jle`	`ZF = 1` or `SF` ≠ `OF`	Jump if less or equal
`jz`	`ZF = 1`	Jump if zero (❗ same as `je`)
... (many more)		

[1] Assume that a comparison instruction precedes the branch instruction.

Examining the ja Case

Example code.

```
cmp rax, rbx
ja label        ; jump to label if rax > rbx
```

- cmp is the same as sub except that it only updates EFLAGS.
- CF = 0 implies that rax - rbx did not produce any carry.
- ZF = 0 implies that the result of subtraction is not zero. Hence, rax ≠ rbx.
- From both the conditions, we can say that rax > rbx.

Summary So Far

- We learned how to move around data.
  - `mov`, `lea`, `push`, `pop`, etc.
- We learned how to perform arithmetic and logical operations.
  - `add`, `sub`, `and`, `or`, etc.
- We also learned how to control program flows.
  - `cmp`, `jmp`, `ja`, `jz`, etc.

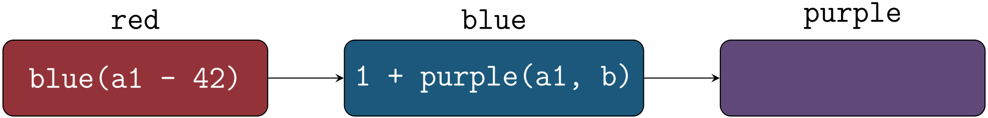
Already Turing complete!

x86 Execution Model

Our Example

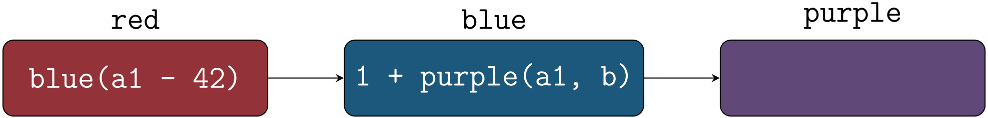
```c
int purple(int a1, int a2)
{
  return 2 + a1 - a2;
}
int blue(int a1)
{
  return 1 + purple(a1, b);
}
int red(int a1)
{
  return blue(a1 - 42);
}
```

Questions



- How do we pass function parameters?
- When a function returns, how do we restore the register values of the caller.
- Where do we store local variables?

Questions



- How do we pass function parameters?
- When a function returns, how do we restore the register values of the caller.
- Where do we store local variables?

We can easily get the answer by compiling the example program and disassembling the resulting binary.

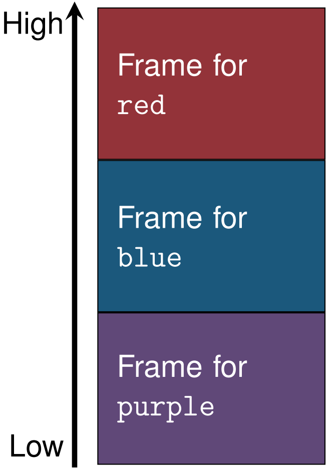
Disassembled Code (x86)

```
<red>:
   0:   push   ebp
   1:   mov    ebp,esp
   3:   sub    esp,0x28
   6:   mov    DWORD PTR [ebp-0xc],0x0
   d:   mov    eax,DWORD PTR [ebp+0x8]
  10:   sub    eax,0x2a
  13:   mov    DWORD PTR [esp],eax
  16:   call   blue
  1b:   mov    edx,DWORD PTR [ebp-0xc]
  1e:   add    eax,edx
  20:   leave
  21:   ret

<blue>:
  22:   push   ebp
  23:   mov    ebp,esp
  25:   sub    esp,0x28
  28:   mov    DWORD PTR [ebp-0xc],0x1
  2f:   mov    eax,DWORD PTR [ebp-0xc]
  32:   mov    DWORD PTR [esp+0x4],eax
  36:   mov    eax,DWORD PTR [ebp+0x8]
  39:   mov    DWORD PTR [esp],eax
  3c:   call   purple
  41:   mov    edx,DWORD PTR [ebp-0xc]
  44:   add    eax,edx
  46:   leave
  47:   ret

<purple>:
  48:   push   ebp
  49:   mov    ebp,esp
  4b:   sub    esp,0x10
  4e:   mov    DWORD PTR [ebp-0x4],0x2
  55:   mov    eax,DWORD PTR [ebp+0x8]
  58:   mov    edx,DWORD PTR [ebp-0x4]
  5b:   add    eax,edx
  5d:   sub    eax,DWORD PTR [ebp+0xc]
  60:   leave
  61:   ret
```

Stack Frames



High

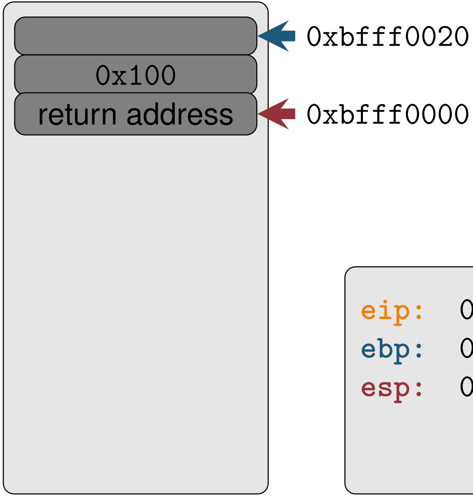
Low

Frame for `red`

Frame for `blue`

Frame for `purple`

Execution Example



Virtual Memory

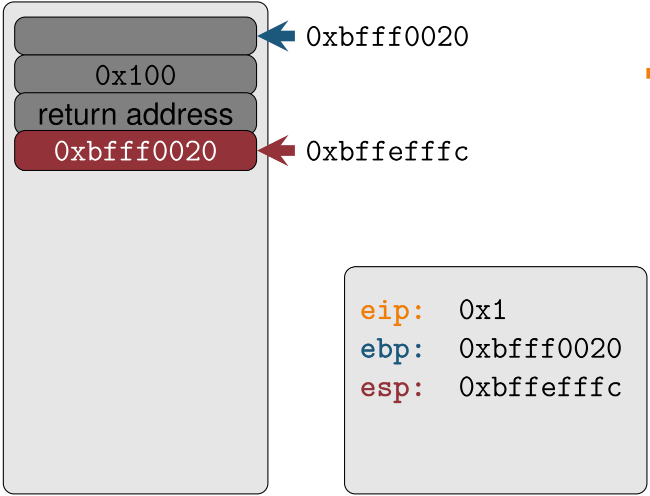
```
eip:   0x0
ebp:   0xbfff0020
esp:   0xbfff0000
```

Execution Context

```
<red>:
 0:   push  ebp
 1:   mov   ebp,esp
 3:   sub   esp,0x28
 6:   mov   DWORD PTR [ebp-0xc],0x0
 d:   mov   eax,DWORD PTR [ebp+0x8]
10:   sub   eax,0x2a
13:   mov   DWORD PTR [esp],eax
16:   call  blue
1b:   mov   edx,DWORD PTR [ebp-0xc]
1e:   add   eax,edx
20:   leave
21:   ret

<blue>:
22:   push  ebp
23:   mov   ebp,esp
...
46:   leave
47:   ret
```

Execution Example



eip: 0x1
ebp: 0xbfff0020
esp: 0xbffefffc

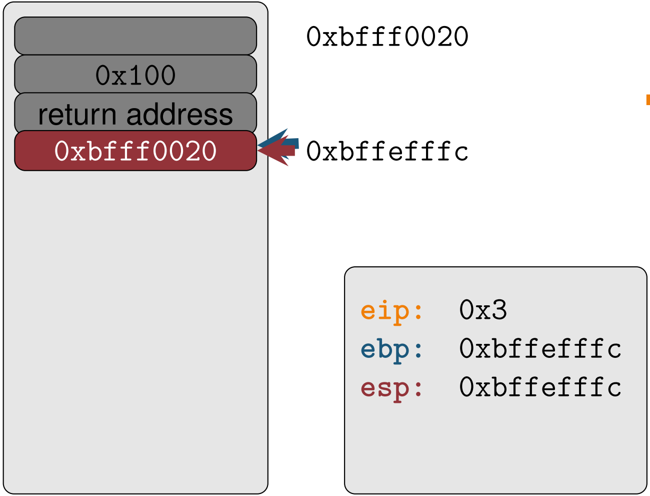
Virtual Memory

Execution Context

```
<red>:
 0:   push  ebp
 1:   mov   ebp,esp
 3:   sub   esp,0x28
 6:   mov   DWORD PTR [ebp-0xc],0x0
 d:   mov   eax,DWORD PTR [ebp+0x8]
10:   sub   eax,0x2a
13:   mov   DWORD PTR [esp],eax
16:   call  blue
1b:   mov   edx,DWORD PTR [ebp-0xc]
1e:   add   eax,edx
20:   leave
21:   ret

<blue>:
22:   push  ebp
23:   mov   ebp,esp
...
46:   leave
47:   ret
```

Execution Example



Virtual Memory

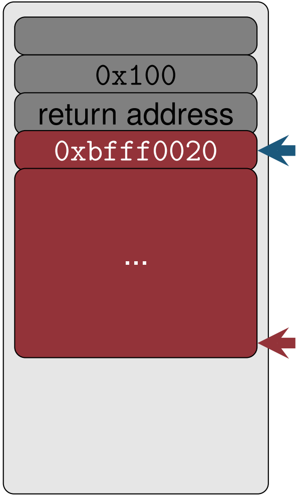
eip: 0x3
ebp: 0xbffefffc
esp: 0xbffefffc

Execution Context

```
<red>:
 0:   push  ebp
 1:   mov   ebp,esp
 3:   sub   esp,0x28
 6:   mov   DWORD PTR [ebp-0xc],0x0
 d:   mov   eax,DWORD PTR [ebp+0x8]
10:   sub   eax,0x2a
13:   mov   DWORD PTR [esp],eax
16:   call  blue
1b:   mov   edx,DWORD PTR [ebp-0xc]
1e:   add   eax,edx
20:   leave
21:   ret

<blue>:
22:   push  ebp
23:   mov   ebp,esp
...
46:   leave
47:   ret
```

Execution Example



0x100

return address

0xbfff0020

...

Virtual Memory

0xbfff0020

eip: 0x6

ebp: 0xbffefffc

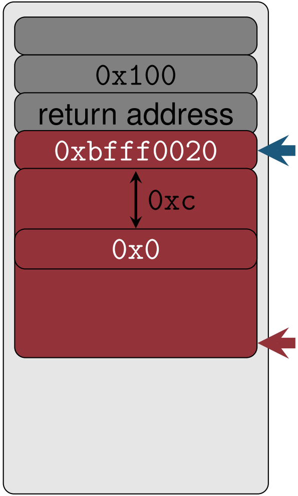
esp: 0xbffeffd4

Execution Context

```
<red>:
 0:   push  ebp
 1:   mov   ebp,esp
 3:   sub   esp,0x28
 6:   mov   DWORD PTR [ebp-0xc],0x0
 d:   mov   eax,DWORD PTR [ebp+0x8]
10:   sub   eax,0x2a
13:   mov   DWORD PTR [esp],eax
16:   call  blue
1b:   mov   edx,DWORD PTR [ebp-0xc]
1e:   add   eax,edx
20:   leave
21:   ret

<blue>:
22:   push  ebp
23:   mov   ebp,esp
...
46:   leave
47:   ret
```

Execution Example



Virtual Memory

0xbfff0020

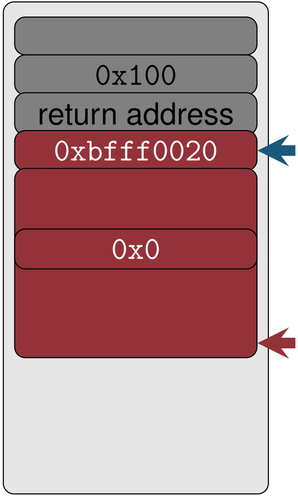
```
eip:   0xd
ebp:   0xbffefffc
esp:   0xbffeffd4
```

Execution Context

```
<red>:
 0:   push  ebp
 1:   mov   ebp,esp
 3:   sub   esp,0x28
 6:   mov   DWORD PTR [ebp-0xc],0x0
 d:   mov   eax,DWORD PTR [ebp+0x8]
10:   sub   eax,0x2a
13:   mov   DWORD PTR [esp],eax
16:   call  blue
1b:   mov   edx,DWORD PTR [ebp-0xc]
1e:   add   eax,edx
20:   leave
21:   ret

<blue>:
22:   push  ebp
23:   mov   ebp,esp
...
46:   leave
47:   ret
```

Execution Example



Virtual Memory

0xbfff0020

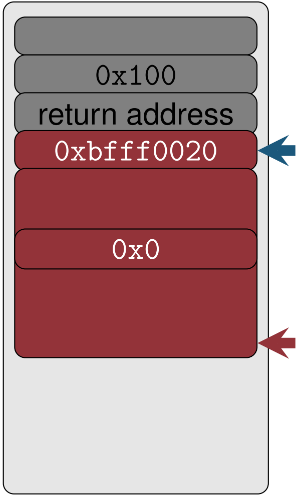
```
eip:   0x10
ebp:   0xbffeffffc
esp:   0xbffeffd4
eax:   0x100
```

Execution Context

```
<red>:
 0:   push  ebp
 1:   mov   ebp,esp
 3:   sub   esp,0x28
 6:   mov   DWORD PTR [ebp-0xc],0x0
 d:   mov   eax,DWORD PTR [ebp+0x8]
10:   sub   eax,0x2a
13:   mov   DWORD PTR [esp],eax
16:   call  blue
1b:   mov   edx,DWORD PTR [ebp-0xc]
1e:   add   eax,edx
20:   leave
21:   ret

<blue>:
22:   push  ebp
23:   mov   ebp,esp
...
46:   leave
47:   ret
```

Execution Example



0x100

return address

0xbfff0020

0x0

Virtual Memory

0xbfff0020

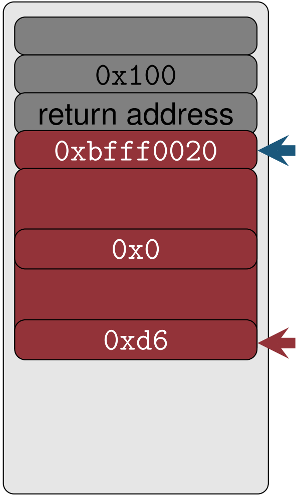
```
eip:  0x13
ebp:  0xbffefffc
esp:  0xbffeffd4
eax:  0xd6
```

Execution Context

```
<red>:
 0:   push  ebp
 1:   mov   ebp,esp
 3:   sub   esp,0x28
 6:   mov   DWORD PTR [ebp-0xc],0x0
 d:   mov   eax,DWORD PTR [ebp+0x8]
10:   sub   eax,0x2a
13:   mov   DWORD PTR [esp],eax
16:   call  blue
1b:   mov   edx,DWORD PTR [ebp-0xc]
1e:   add   eax,edx
20:   leave
21:   ret

<blue>:
22:   push  ebp
23:   mov   ebp,esp
...
46:   leave
47:   ret
```

Execution Example



Virtual Memory

0xbfff0020

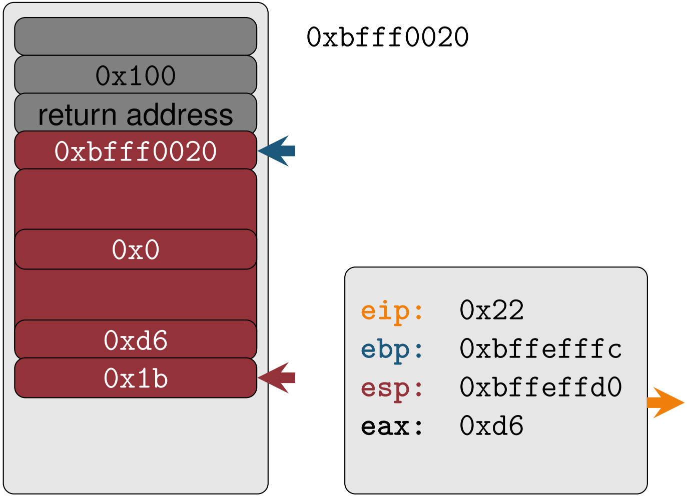
```
eip:   0x16
ebp:   0xbffefffc
esp:   0xbffeffd4
eax:   0xd6
```

Execution Context

```
<red>:
 0:   push  ebp
 1:   mov   ebp,esp
 3:   sub   esp,0x28
 6:   mov   DWORD PTR [ebp-0xc],0x0
 d:   mov   eax,DWORD PTR [ebp+0x8]
10:   sub   eax,0x2a
13:   mov   DWORD PTR [esp],eax
16:   call  blue
1b:   mov   edx,DWORD PTR [ebp-0xc]
1e:   add   eax,edx
20:   leave
21:   ret

<blue>:
22:   push  ebp
23:   mov   ebp,esp
...
46:   leave
47:   ret
```

Execution Example



0xbfff0020

0x100

return address

0xbfff0020

0x0

0xd6

0x1b

Virtual Memory

eip: 0x22

ebp: 0xbffefffc

esp: 0xbffeffd0

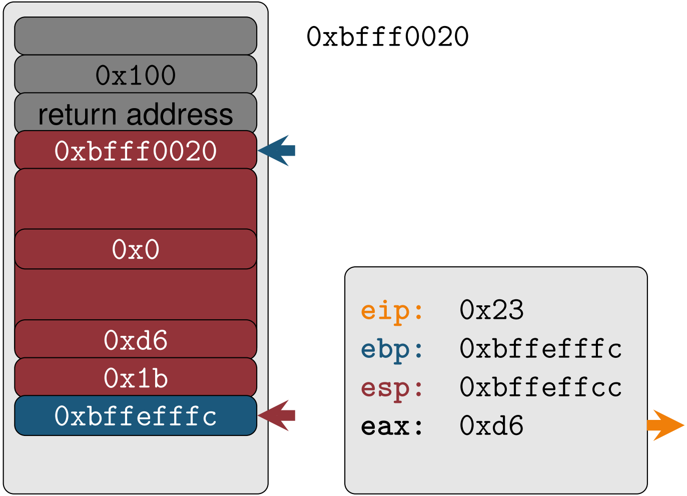
eax: 0xd6

Execution Context

```
<red>:
 0:   push  ebp
 1:   mov   ebp,esp
 3:   sub   esp,0x28
 6:   mov   DWORD PTR [ebp-0xc],0x0
 d:   mov   eax,DWORD PTR [ebp+0x8]
10:   sub   eax,0x2a
13:   mov   DWORD PTR [esp],eax
16:   call  blue
1b:   mov   edx,DWORD PTR [ebp-0xc]
1e:   add   eax,edx
20:   leave
21:   ret

<blue>:
22:   push  ebp
23:   mov   ebp,esp
...
46:   leave
47:   ret
```

Execution Example



0xbfff0020

0x100

return address

0xbfff0020

0x0

0xd6

0x1b

0xbffefffc

Virtual Memory

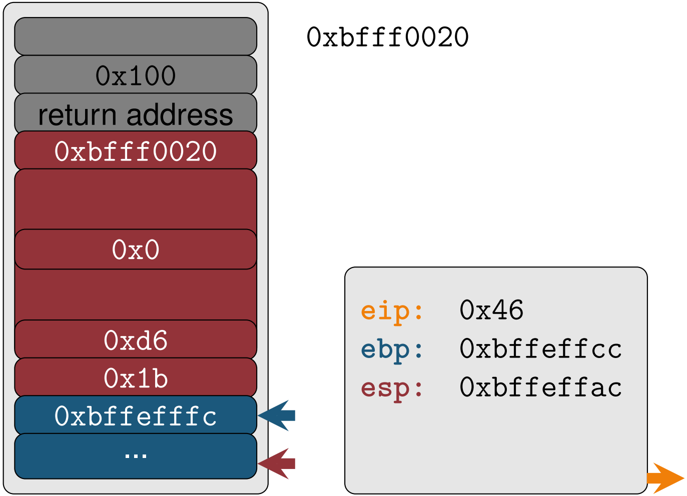
```
eip:   0x23
ebp:   0xbffefffc
esp:   0xbffeffcc
eax:   0xd6
```

Execution Context

```
<red>:
 0:   push  ebp
 1:   mov   ebp,esp
 3:   sub   esp,0x28
 6:   mov   DWORD PTR [ebp-0xc],0x0
 d:   mov   eax,DWORD PTR [ebp+0x8]
10:   sub   eax,0x2a
13:   mov   DWORD PTR [esp],eax
16:   call  blue
1b:   mov   edx,DWORD PTR [ebp-0xc]
1e:   add   eax,edx
20:   leave
21:   ret

<blue>:
22:   push  ebp
23:   mov   ebp,esp
...
46:   leave
47:   ret
```

Execution Example



0xbfff0020

0x100

return address

0xbfff0020

0x0

0xd6

0x1b

0xbffefffc

...

Virtual Memory

eip: 0x46

ebp: 0xbffeffcc

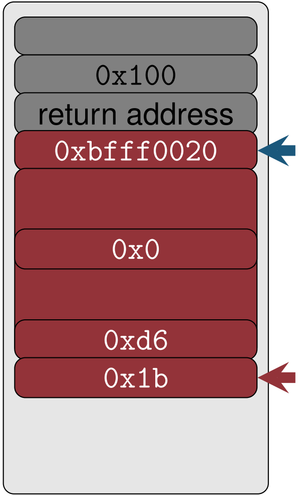
esp: 0xbffeffac

Execution Context

```
<red>:
 0:   push  ebp
 1:   mov   ebp,esp
 3:   sub   esp,0x28
 6:   mov   DWORD PTR [ebp-0xc],0x0
 d:   mov   eax,DWORD PTR [ebp+0x8]
10:   sub   eax,0x2a
13:   mov   DWORD PTR [esp],eax
16:   call  blue
1b:   mov   edx,DWORD PTR [ebp-0xc]
1e:   add   eax,edx
20:   leave
21:   ret

<blue>:
22:   push  ebp
23:   mov   ebp,esp
...
46:   leave    =  mov esp, ebp
47:   ret         pop ebp
```

Execution Example



Virtual Memory

0xbfff0020

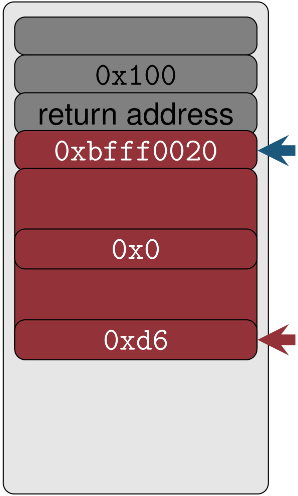
```
eip:   0x47
ebp:   0xbffefffc
esp:   0xbffeffcc
```

Execution Context

```
<red>:
 0:   push  ebp
 1:   mov   ebp,esp
 3:   sub   esp,0x28
 6:   mov   DWORD PTR [ebp-0xc],0x0
 d:   mov   eax,DWORD PTR [ebp+0x8]
10:   sub   eax,0x2a
13:   mov   DWORD PTR [esp],eax
16:   call  blue
1b:   mov   edx,DWORD PTR [ebp-0xc]
1e:   add   eax,edx
20:   leave
21:   ret

<blue>:
22:   push  ebp
23:   mov   ebp,esp
...
46:   leave
47:   ret
```

Execution Example



Virtual Memory

0xbfff0020

```
eip:   0x1b
ebp:   0xbffefffc
esp:   0xbffeffd0
```

Execution Context

```
<red>:
 0:   push  ebp
 1:   mov   ebp,esp
 3:   sub   esp,0x28
 6:   mov   DWORD PTR [ebp-0xc],0x0
 d:   mov   eax,DWORD PTR [ebp+0x8]
10:   sub   eax,0x2a
13:   mov   DWORD PTR [esp],eax
16:   call  blue
1b:   mov   edx,DWORD PTR [ebp-0xc]
1e:   add   eax,edx
20:   leave
21:   ret

<blue>:
22:   push  ebp
23:   mov   ebp,esp
...
46:   leave
47:   ret
```

Question?

Exercise

1. Write any sort function in x86 (or x86-64) assembly, and create an object file.
2. Write a C function that tests the sort function. Link with the object file and run your test.