IA32
- stack discipline
- Register saving conventions
- Creating pointers to local variables

x86-64
- Argument passing in registers
- Minimizing stack usage
- Using stack pointer as only reference
IA32 Stack

- Region of memory managed with stack discipline
- Grows toward lower addresses
- Register `%esp` indicates lowest stack address
  - Address of top element

![IA32 Stack Diagram](image)
IA32 Stack Pushing

Pushing
- `pushl Src`
- Fetch operand at `Src`
- Decrement `%esp` by 4
- Write operand at address given by `%esp`
IA32 Stack Popping

Popping

- `popl Dest`
- Read operand at address given by `%esp`
- Increment `%esp` by 4
- Write to `Dest`

Stack Pointer `%esp`

Stack "Top"

Stack Grows Down

Increasing Addresses

Stack "Bottom"
Procedure Control Flow

- Use stack to support procedure call and return

Procedure call:

```
call label
Push return address on stack; Jump to label
```

Return address value

- Address of instruction beyond call
- Example from disassembly

```
804854e: e8 3d 06 00 00 call 8048b90 <main>
8048553: 50 pushl %eax
```

- Return address = 0x8048553

Procedure return:

```
ret
Pop address from stack; Jump to address
```
Procedure Call Example

804854e:  e8 3d 06 00 00  
call  8048b90 <main>

8048553:  50

pushl %eax

call  8048b90

%esp  0x108

%esp  0x110

%esp  0x10c

%eip  0x804854e

%eip  0x8048553

%eip  0x8048b90

%eip is program counter
Procedure Return Example

8048591: c3       ret

%esp  0x104
%eip  0x8048591

0x108  123
0x10c  
0x110  

%esp  0x104
%eip  0x8048591

0x108  123
0x10c  
0x110  

%esp  0x108
%eip  0x8048591

%eip is program counter

ret
Stack-Based Languages

Languages that Support Recursion

- e.g., C, Pascal, Java
- Code must be “Reentrant”
  - Multiple simultaneous instantiations of single procedure
- Need some place to store state of each instantiation
  - Arguments
  - Local variables
  - Return pointer

Stack Discipline

- State for given procedure needed for limited time
  - From when called to when return
- Callee returns before caller does

Stack Allocated in Frames

- state for single procedure instantiation
Call Chain Example

Code Structure

yoo (...)
{
 •
 •
 who ();
 •
 •
}

who (...)
{
 • • •
 • • •
 amI ();
 • • •
 amI ();
 • • •

 Procedure amI
 recursive

Call Chain

yoo

who

amI  amI

amI

amI

amI

amI

amI

amI
Stack Frames

Contents
- Local variables
- Return information
- Temporary space

Management
- Space allocated when enter procedure
  - “Set-up” code
- Deallocated when return
  - “Finish” code

Pointers
- Stack pointer %esp indicates stack top
- Frame pointer %ebp indicates start of current frame
Stack Operation

```
yoo (...)  
{          
  •        
  •        
  who ();  
  •        
  •        
}
```

Call Chain

Frame Pointer %ebp
Stack Pointer %esp

yoo
yoo

-1-
Stack Operation

```c
who(...) {
    ... amI(); ...
    amI(); ...
}
```

Call Chain

- Frame Pointer `%ebp`
- Stack Pointer `%esp`
- `who`
Stack Operation

Call Chain

```
void amI(...) {
    ...
    amI();
    ...
}
```
Stack Operation

```
amI(...) {
  ...
  amI();
  ...
}
```

Call Chain

```
Frame Pointer %ebp
Stack Pointer %esp
```

```
Call Chain

- yoo
- who
- amI
- amI
```
Stack Operation

```c
amI(...) {
    .
    .
    amI();
    .
    .
}
```

Call Chain

```
    .
    .
    yoo
    who
    amI
    amI
    amI
    Frame Pointer %ebp
    Stack Pointer %esp
```

yoo
who
amI
amI
amI
amI
amI
Stack Operation

Call Chain

```c
amI(...) {
    •
    •
    amI();
    •
    •
}
```

Frame Pointer
%ebp

Stack Pointer
%esp

•

•

yoo

who

amI

amI

amI

amI

amI

amI

amI

amI

amI

amI

amI

amI

amI

amI

amI

amI

amI
Stack Operation

```c
amI(...) {
    •
    •
    amI();
    •
}
```

Call Chain

Frame Pointer
%ebp

Stack Pointer
%esp

yoo

who

amI

amI
Stack Operation

who(...) {
    ...
    amI();
    ...
    amI();
    ...
}

Call Chain

Frame Pointer
%ebp

Stack Pointer
%esp

who

amI

amI

amI

yoo

yoo
Stack Operation

```c
amI(...) {
    ...
    ...
    ...
    ...
}
```

Call Chain

```
  yoo
  |
  v
who
  |
  v
amI
  |
  v
amI
  |
  v
amI
  |
  v
amI
```

Frame Pointer

```
%ebp
```

Stack Pointer

```
%esp
```
Stack Operation

Stack Operation

who(...) {
    ...
    amI();
    ...
    amI();
    ...
}

Call Chain

Frame Pointer
%ebp

Stack Pointer
%esp

yoo

who

amI

amI

amI

amI

amI

yoo

who

amI

amI

amI

amI

amI
Stack Operation

```c
yoo(...) {
  
  who();
  
}
```

Call Chain

Frame Pointer
%ebp

Stack Pointer
%esp

```plaintext
yoo
who
amI amI
amI
amI
```
IA32/Linux Stack Frame

Current Stack Frame ("Top" to Bottom)

- Parameters for function about to call
  - "Argument build"
- Local variables
  - If can't keep in registers
- Saved register context
- Old frame pointer

Caller Stack Frame

- Return address
  - Pushed by call instruction
- Arguments for this call
Revisiting swap

```c
int zip1 = 15213;
int zip2 = 91125;

void call_swap()
{
    swap(&zip1, &zip2);
}

void swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

Calling swap from call_swap

call_swap:

```assembly
    pushl $zip2  # Global Var
    pushl $zip1  # Global Var
    call swap
...```

Resulting Stack

- `$esp`
- `Rtn adr`
- `&zip1`
- `&zip2`
Revisiting swap

void swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}

swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
    movl 12(%ebp),%ecx
    movl 8(%ebp),%edx
    movl (%ecx),%eax
    movl (%edx),%ebx
    movl %eax,(%edx)
    movl %ebx,(%ecx)
    movl -4(%ebp),%ebx
    movl %ebp,%esp
    popl %ebp
    ret

Set Up

Body

Finish
swap Setup #1

Entering Stack

- %ebp
- &zip2
- &zip1
- Rtn adr

Resulting Stack

- %ebp
- yp
- xp
- Rtn adr
- Old %ebp

\[
\text{swap:}
\]

\[
\begin{align*}
\text{pushl } & %ebp \\
\text{movl } & %esp, %ebp \\
\text{pushl } & %ebx
\end{align*}
\]
swap Setup #2

Enter Stack

\[ \text{\%ebp} \]

\[ \&\text{zip2} \]

\[ \&\text{zip1} \]

\[ \text{Rtn adr} \]

\%esp

\%ebp

\%esp

Resulting Stack

\[ \text{\%ebp} \]

\[ \text{\%esp} \]

\[ \text{yp} \]

\[ \text{xp} \]

\[ \text{Rtn adr} \]

\[ \text{Old \%ebp} \]

\%ebp

\%esp

\%ebp

\%esp

swap:

\[ \text{pushl \%ebp} \]

\[ \text{movl \%esp,\%ebp} \]

\[ \text{pushl \%ebx} \]
swap Setup #3

Entering Stack

\[\begin{array}{c}
\vdots \\
\vdots \\
\vdots \\
&\text{zip2} \\
&\text{zip1} \\
\text{Rtn adr} \\
\end{array}\]

\[\begin{array}{c}
%ebp \\
%esp \\
\end{array}\]

Resulting Stack

\[\begin{array}{c}
\vdots \\
\vdots \\
\vdots \\
\text{YP} \\
\text{xp} \\
\text{Rtn adr} \\
\text{Old %ebp} \\
\text{Old %ebx} \\
\end{array}\]

\[\begin{array}{c}
%ebp \\
%esp \\
\end{array}\]

\text{swap:}

pushl %ebp
movl %esp,%ebp
pushl %ebx
**Effect of swap Setup**

**Entering Stack**

- `%ebp`
- `%esp`
- `&zip2`
- `&zip1`
- `Rtn adr`

**Resulting Stack**

- `%ebp`
- `%esp`
- `yp`
- `xp`
- `Old %ebp`
- `Old %ebx`

Offset (relative to `%ebp`)

- `&zip2`: 12
- `&zip1`: 8
- `Rtn adr`: 4
- `0`

Body

```
movl 12(%ebp), %ecx  # get yp
movl 8(%ebp), %edx   # get xp
...
```
swap Finish #1

**Observation**
- Saved & restored register `%ebx`

```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```
swap Finish #2

swap’s Stack

<table>
<thead>
<tr>
<th>Offset</th>
<th>yp</th>
<th>xp</th>
<th>Rtn adr</th>
<th>Old %ebp</th>
<th>Old %ebx</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>%ebp</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-4</td>
<td>%esp</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

swap’s Stack

<table>
<thead>
<tr>
<th>Offset</th>
<th>yp</th>
<th>xp</th>
<th>Rtn adr</th>
<th>Old %ebp</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>%ebp</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

movl -4(%ebp), %ebx
movl %ebp, %esp
popl %ebp
ret
swap Finish #3

swap’s Stack

Offset
12
8
4
0

Old %ebp

Rtn adr

swap’s Stack

Offset
12
8
4

%ebp

%esp

%ebp

%esp

Rtn adr

movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
Observation
- Saved & restored register %ebx
- Didn’t do so for %eax, %ecx, or %edx

movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
Register Saving Conventions

When procedure \textit{yoo} calls \textit{who}:
\begin{itemize}
  \item \textit{yoo} is the \textit{caller}, \textit{who} is the \textit{callee}
\end{itemize}

Can Register be Used for Temporary Storage?

\begin{itemize}
  \item Contents of register \texttt{%edx} overwritten by \textit{who}
\end{itemize}
Register Saving Conventions

When procedure yoo calls who:

- yoo is the caller, who is the callee

Can Register be Used for Temporary Storage?

Conventions

- “Caller Save”
  - Caller saves temporary in its frame before calling

- “Callee Save”
  - Callee saves temporary in its frame before using
IA32/Linux Register Usage

Integer Registers

- Two have special uses
  - %ebp, %esp
- Three managed as callee-save
  - %ebx, %esi, %edi
  - Old values saved on stack prior to using
- Three managed as caller-save
  - %eax, %edx, %ecx
  - Do what you please, but expect any callee to do so, as well
- Register %eax also stores returned value

- %eax
- %edx
- %ecx
- %ebx
- %esi
- %edi
- %esp
- %ebp
**Recursive Factorial**

```c
int rfact(int x)
{
    int rval;
    if (x <= 1)
        return 1;
    rval = rfact(x-1);
    return rval * x;
}
```

**Registers**

- `%eax` used without first saving
- `%ebx` used, but save at beginning & restore at end
Rfact Stack Setup

Entering Stack

rfact:
pushl %ebp
movl %esp,%ebp
pushl %ebx
Recursion

```
int rfact(int x)
{
    int rval;
    if (x <= 1)
        return 1;
    rval = rfact(x-1) ;
    return rval * x;
}
```

Registers

- `%ebx` Stored value of `x`
- `%eax` Temporary value of `x-1`
- Returned value from `rfact(x-1)`
- Returned value from this call

```
movl 8(%ebp),%ebx  # ebx = x
cmpl $1,%ebx      # Compare x : 1
jle  .L78         # If <= goto Term
leal -1(%ebx),%eax # eax = x-1
pushl %eax        # Push x-1
call rfact        # rfact(x-1)
imull %ebx,%eax    # rval * x
jmp   .L79        # Goto done
.L78:              # Term:
    movl $1,%eax   # return val = 1
.L79:              # Done:
```
Rfact Recursion

\texttt{leal -1(\%ebx),\%eax}

\begin{align*}
\text{\%eax} &\quad x-1 \\
\text{\%ebx} &\quad x \\
\text{\%ebp} &\quad \text{Old} \\
\text{\%esp} &\quad \text{Old} \\
\end{align*}

\begin{align*}
\text{\%eax} &\quad x-1 \\
\text{\%ebx} &\quad x \\
\text{\%ebp} &\quad \text{Old} \\
\text{\%esp} &\quad \text{Old} \\
\end{align*}

\begin{align*}
\text{\%eax} &\quad x \\
\text{\%ebx} &\quad \text{Rtn adr} \\
\text{\%ebp} &\quad \text{Old} \\
\text{\%esp} &\quad \text{Old} \\
\end{align*}

\begin{align*}
\text{\%eax} &\quad x \\
\text{\%ebx} &\quad \text{Rtn adr} \\
\text{\%ebp} &\quad \text{Old} \\
\text{\%esp} &\quad \text{Old} \\
\end{align*}

\begin{align*}
\text{\%eax} &\quad x-1 \\
\text{\%ebx} &\quad x \\
\text{\%ebp} &\quad \text{Old} \\
\text{\%esp} &\quad \text{Old} \\
\end{align*}

\begin{align*}
\text{\%eax} &\quad x-1 \\
\text{\%ebx} &\quad x \\
\text{\%ebp} &\quad \text{Old} \\
\text{\%esp} &\quad \text{Old} \\
\end{align*}

\begin{align*}
\text{\%eax} &\quad x \\
\text{\%ebx} &\quad \text{Rtn adr} \\
\text{\%ebp} &\quad \text{Old} \\
\text{\%esp} &\quad \text{Old} \\
\end{align*}

\begin{align*}
\text{\%eax} &\quad x \\
\text{\%ebx} &\quad \text{Rtn adr} \\
\text{\%ebp} &\quad \text{Old} \\
\text{\%esp} &\quad \text{Old} \\
\end{align*}

\begin{align*}
\text{\%eax} &\quad x-1 \\
\text{\%ebx} &\quad x \\
\text{\%ebp} &\quad \text{Old} \\
\text{\%esp} &\quad \text{Old} \\
\end{align*}

\begin{align*}
\text{\%eax} &\quad x \\
\text{\%ebx} &\quad \text{Rtn adr} \\
\text{\%ebp} &\quad \text{Old} \\
\text{\%esp} &\quad \text{Old} \\
\end{align*}
Return from Call

Assume that rfact(x-1) returns (x-1)! in register %eax
Rfact Completion

```assembly
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```
void s_helper
  (int x, int *accum)
{
  if (x <= 1)
    return;
  else {
    int z = *accum * x;
    *accum = z;
    s_helper (x-1,accum);
  }
}

int sfact(int x)
{
  int val = 1;
  s_helper(x, &val);
  return val;
}

- Pass pointer to update location
Creating & Initializing Pointer

Initial part of _sfact

Push %ebp
# Save %ebp
movl %esp,%ebp # Set %ebp
subl $16,%esp # Add 16 bytes
movl 8(%ebp),%edx # edx = x
movl $1,-4(%ebp) # val = 1

Using Stack for Local Variable

- Variable val must be stored on stack
  - Need to create pointer to it
- Compute pointer as -
  4 (%ebp)
- Push on stack as second argument

```c
int sfact(int x)
{
    int val = 1;
    s_helper(x, &val);
    return val;
}
```
Passing Pointer

Calling `s_helper` from `sfact`

```c
int sfact(int x)
{
    int val = 1;
    s_helper(x, &val);
    return val;
}
```

```
leal -4(%ebp),%eax  # Compute &val
pushl %eax          # Push on stack
pushl %edx          # Push x
    call s_helper   # call
movl -4(%ebp),%eax  # Return val
    • • •           # Finish
```

Stack at time of call

- `%ebp`
- `x`
- `val = x!`
- `Rtn adr`
- `Old %ebp`
- `Unused`
- `&val`
- `%esp`
Using Pointer

```c
void s_helper(int x, int *accum) {
    int z = *accum * x;
    *accum = z;
}
```

- **Register %ecx holds x**
- **Register %edx holds accum**
  - Assume memory initially has value V
  - Use access (%edx) to reference memory

```assembly
    movl %ecx, %eax        # z = x
    imull (%edx), %eax     # z *= *accum
    movl %eax, (%edx)      # *accum = z
```

The Stack Makes Recursion Work

- Private storage for each instance of procedure call
  - Instantiations don’t clobber each other
  - Addressing of locals + arguments can be relative to stack positions
- Can be managed by stack discipline
  - Procedures return in inverse order of calls

IA32 Procedures Combination of Instructions + Conventions

- Call / Ret instructions
- Register usage conventions
  - Caller / Callee save
  - %ebp and %esp
- Stack frame organization conventions
### x86-64 General Purpose Registers

<table>
<thead>
<tr>
<th>%rax</th>
<th>%eax</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rbx</td>
<td>%ebx</td>
</tr>
<tr>
<td>%rcx</td>
<td>%ecx</td>
</tr>
<tr>
<td>%rdx</td>
<td>%edx</td>
</tr>
<tr>
<td>%r8</td>
<td>%r8d</td>
</tr>
<tr>
<td>%r9</td>
<td>%r9d</td>
</tr>
<tr>
<td>%r10</td>
<td>%r10d</td>
</tr>
<tr>
<td>%r11</td>
<td>%r11d</td>
</tr>
<tr>
<td>%r12</td>
<td>%r12d</td>
</tr>
<tr>
<td>%r13</td>
<td>%r13d</td>
</tr>
<tr>
<td>%r14</td>
<td>%r14d</td>
</tr>
<tr>
<td>%r15</td>
<td>%r15d</td>
</tr>
</tbody>
</table>

- Twice the number of registers
- Accessible as 8, 16, 32, or 64 bits
## x86-64 Register Conventions

<table>
<thead>
<tr>
<th>%rax</th>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rbx</td>
<td>Callee Saved</td>
</tr>
<tr>
<td>%rcx</td>
<td>Argument #4</td>
</tr>
<tr>
<td>%rdx</td>
<td>Argument #3</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument #2</td>
</tr>
<tr>
<td>%rdi</td>
<td>Argument #1</td>
</tr>
<tr>
<td>%rsp</td>
<td>Stack Pointer</td>
</tr>
<tr>
<td>%rbp</td>
<td>Callee Saved</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>%r8</th>
<th>Argument #5</th>
</tr>
</thead>
<tbody>
<tr>
<td>%r9</td>
<td>Callee Saved</td>
</tr>
<tr>
<td>%r10</td>
<td>Callee Saved</td>
</tr>
<tr>
<td>%r11</td>
<td>Used for linking</td>
</tr>
<tr>
<td>%r12</td>
<td>C: Callee Saved</td>
</tr>
<tr>
<td>%r13</td>
<td>Callee Saved</td>
</tr>
<tr>
<td>%r14</td>
<td>Callee Saved</td>
</tr>
<tr>
<td>%r15</td>
<td>Callee Saved</td>
</tr>
</tbody>
</table>
x86-64 Registers

Arguments passed to functions via registers
- If more than 6 integral parameters, then pass rest on stack
- These registers can be used as caller-saved as well

All References to Stack Frame via Stack Pointer
- Eliminates need to update %ebp

Other Registers
- 6+1 callee saved
- 2 or 3 have special uses
x86-64 Long Swap

void swap(long *xp, long *yp) {
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}

- Operands passed in registers
  - First (xp) in %rdi, second (yp) in %rsi
  - 64-bit pointers
- No stack operations required

Avoiding Stack
- Can hold all local information in registers

swap:
  movq (%rdi), %rdx
  movq (%rsi), %rax
  movq %rax, (%rdi)
  movq %rdx, (%rsi)
  ret
x86-64 Locals in the Red Zone

Avoiding Stack Pointer Change

- Can hold all information within small window beyond stack pointer

```c
/* Swap, using local array */
void swap_a(long *xp, long *yp)
{
    volatile long loc[2];
    loc[0] = *xp;
    loc[1] = *yp;
    *xp = loc[1];
    *yp = loc[0];
}
```

swap_a:
```asm
    movq (%rdi), %rax
    movq %rax, -24(%rsp)
    movq (%rsi), %rax
    movq %rax, -16(%rsp)
    movq -16(%rsp), %rax
    movq %rax, (%rdi)
    movq -24(%rsp), %rax
    movq %rax, (%rsi)
    ret
```

- Can hold all information within small window beyond stack pointer

<table>
<thead>
<tr>
<th>Address</th>
<th>Name</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>-8</td>
<td>unused</td>
<td></td>
</tr>
<tr>
<td>-16</td>
<td>loc[1]</td>
<td></td>
</tr>
<tr>
<td>-24</td>
<td>loc[0]</td>
<td></td>
</tr>
</tbody>
</table>

\[ \text{rtn Ptr} \rightarrow \%rsp \]
long scount = 0;
/* Swap a[i] & a[i+1] */
void swap_ele_se
  (long a[], int i)
{
  swap(&a[i], &a[i+1]);
  scount++;
}

swap_ele_se:
  movslq %esi,%rsi          # Sign extend i
  leaq  (%rdi,%rsi,8), %rdi # &a[i]
  leaq  8(%rdi), %rsi       # &a[i+1]
  call  swap                # swap()
  incq  scount(%rip)         # scount++;
  ret

- No values held while swap being invoked
- No callee save registers needed
long scount = 0;
/* Swap a[i] & a[i+1] */
void swap_ele
  (long a[], int i)
{
  swap(&a[i], &a[i+1]);
}

swap_ele:
  movslq %esi,%rsi      # Sign extend i
  leaq (%rdi,%rsi,8), %rdi # &a[i]
  leaq 8(%rdi), %rsi   # &a[i+1]
  jmp    swap          # swap()

- When swap executes ret, it will return from swap_ele
- Possible since swap is a “tail call”
x86-64 Stack Frame Example

- Keeps values of a and i in callee save registers
- Must set up stack frame to save these registers

```c
long sum = 0;
/* Swap a[i] & a[i+1] */
void swap_ele_su
  (long a[], int i)
{
  swap(&a[i], &a[i+1]);
  sum += a[i];
}
```

```assembly
swap_ele_su:
  movq  %rbx, -16(%rsp)
  movslq %esi,%rbx
  movq  %r12, -8(%rsp)
  movq  %rdi, %r12
  leaq  (%rdi,%rbx,8), %rdi
  subq  $16, %rsp
  leaq  8(%rdi), %rsi
  call  swap
  movq  (%r12,%rbx,8), %rax
  addq  %rax, sum(%rip)
  movq  (%rsp), %rbx
  movq  8(%rsp), %r12
  addq  $16, %rsp
  ret
```
Understanding x86-64 Stack Frame

swap_ele_su:

movq  %rbx, -16(%rsp)  # Save %rbx
movslq %esi,%rbx      # Extend & save i
movq  %r12, -8(%rsp)  # Save %r12
movq  %rdi, %r12       # Save a
lea    (%rdi,%rbx,8), %rdi  # &a[i]
subq  $16, %rsp        # Allocate stack frame
lea    8(%rdi), %rsi   # &a[i+1]
call   swap            # swap()
movq  (%r12,%rbx,8), %rax  # a[i]
addq  %rax, sum(%rip)   # sum += a[i]
movq  (%rsp), %rbx     # Restore %rbx
movq  8(%rsp), %r12    # Restore %r12
addq  $16, %rsp        # Deallocate stack frame
ret
Stack Operations

movq %rbx, -16(%rsp)  # Save %rbx
movq %r12, -8(%rsp)   # Save %r12
subq $16, %rsp       # Allocate stack frame

movq (%rsp), %rbx   # Restore %rbx
movq 8(%rsp), %r12  # Restore %r12
addq $16, %rsp      # Deallocate stack frame
Interesting Features of Stack Frame

Allocate Entire Frame at Once
- All stack accesses can be relative to %rsp
- Do by decrementing stack pointer
- Can delay allocation, since safe to temporarily use red zone

Simple Deallocation
- Increment stack pointer
x86-64 Procedure Summary

Heavy Use of Registers
- Parameter passing
- More temporaries

Minimal Use of Stack
- Sometimes none
- Allocate/deallocate entire block

Many Tricky Optimizations
- What kind of stack frame to use
- Calling with jump
- Various allocation techniques