Arguments: a at $ebp+8, b at $ebp+12

Registers: a in %ecx, b in %ebx, result in %eax, %edx set to apb (a+b)

1    movl    8(%ebp), %ecx        Get a
2    movl    12(%ebp), %ebx       Get b
3    movl    $1, %eax            Set result = 1
4    cmpl    %ebx, %ecx          Compare a:b
5    jge     .L11                If >=, goto done
6    leal    (%ebx,%ecx), %edx    Compute apb = a+b
7    movl    $1, %eax            Set result = 1
8    .L12:
9    imull    %edx, %eax          Compute result *= apb
10   addl     $1, %ecx           Compute a++
11   addl     $1, %edx           Compute apb++
12   cmpl    %ecx, %ebx          Compare b:a
13   jg      .L12                If >, goto loop
14   .L11:
   
   Return result

---

int loop_while_goto(int a, int b)
{
    int result = 1;
    if (a >= b)
        goto done;
    /* apb has same value as a+b in original code */
    int apb = a+b;
    loop:
    result *= apb;
    a++;        
    apb++;      
    if (b > a)
        goto loop;
    done:
    return result;
}
.type _Z15loop_while_gotoi, @function _Z15loop_while_gotoi:

.LFB31:
  .cfi_startproc
  pushl %ebp
  .cfi_def_cfa_offset 8
  movl %esp, %ebp
  .cfi_offset 5, -8
  .cfi_def_cfa_register 5
  pushl %ebx
  movl 8(%ebp), %edx
  movl 12(%ebp), %ebx
  .cfi_offset 3, -12
  .cfi_def_cfa_register 5
  pushl %ebx
  movl 8(%ebp), %edx
  movl 12(%ebp), %ebx
  .cfi_offset 3, -12
  movl $1, %eax
  cmpl %edx, %ebx
  jge .L2
  .L3:
  imull %edx, %eax
  addl $1, %eax
  addl $1, %ecx
  cmpl %edx, %ebx
  jg .L2
  leal (%ebx,%edx), %ecx
  .L2:
  popl %ebx
  .cfi_restore 3
  popl %ebp
  .cfi_restore 5
  .cfi_def_cfa 4, 4
  ret
  .cfi_endproc

Arguments: a at %ebp+8, b at %ebp+12
Registers: a in %ecx, b in %ebx, result in %eax, %edx set to apb (a+b)
1  movl  8(%ebp), %ecx   Get a
2  movl  12(%ebp), %ebx  Get b
3  movl $1, %eax         Set result = 1
4  cmpl %ebx, %ecx       Compare a:b
5  jge  .L11            If >=, goto done
6  leal (%ebx,%ecx), %edx Compute apb = a+b
7  movl $1, %eax        Set result = 1
8  .L12:                loop:
9  imull %edx, %eax      Compute result *= apb
10 addl $1, %ecx        Compute a++
11 addl $1, %edx        Compute apb++
12 cmpl %ecx, %ebx      Compare b:a
13 jg   .L12            If >, goto loop
14 .L11:                done:
   Return result
int E[10]; // an array of integers
int i; // index into array

E is in %edx, i is in %ecx
Move something into %eax

E int * movl %edx, %eax
E[0] int movl (%edx), %eax
E[i] int movl (%edx, %ecx, 4), %eax
&E[2] int* leal 8(%edx), %eax
&\text{E[i]}-\text{E} int movl %ecx, %eax
#define vs. const for array allocation
- const actually not in K&R
- const needs a lookup
- #define is a pre-processor step
- #define doesn’t need a type!
IA 32 Procedure Summary

• 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
  – 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
int proc(void)
{
    int x,y;
    scanf("%x %x", &y, &x);
    return x-y;
}

call scanf

Diagam stack frame at this point

movl -4(%ebp), %eax
subl -8(%ebp), %eax
leave
ret

Register    Value
%esp 0x800040
%ebp 0x800060

Suppose proc calls scanf (line 10), and that scanf reads values 0x46 and 0x53 from the standard input. Assume that the string “%x %x” is stored at memory location 0x300070.

A. What value does %ebp get set to on line 3?
B. What value does %esp get set to on line 4?
C. At what addresses are local variables x and y stored?
D. Draw a diagram of the stack frame for proc right after scanf returns. Include as much information as you can about the addresses and the contents of the stack frame elements.
E. Indicate the regions of the stack frame that are not used by proc.
A. We started with `%esp` having value 0x800040. The `pushl` instruction on line 2 decrements the stack pointer by 4, giving 0x80003C, and this becomes the new value of `%ebp`.

B. Line 4 decrements the stack pointer by 40 (hex 0x28), yielding 0x800014.

C. We can see how the two `leal` instructions (lines 5 and 7) compute the arguments to pass to `scanf`, while the two `movl` instructions (lines 6 and 8) store them on the stack. Since the function arguments appear on the stack at increasingly positive offsets from `%esp`, we can conclude that line 5 computes &x, while line 7 computes line &y. These have values 0x800038 and 0x800034, respectively.

D. The stack frame has the following structure and contents:

```
1  proc:
2    pushl  %ebp
3    movl  %esp, %ebp
4    subl  $40, %esp
5    leal  -4(%ebp), %eax
6    movl  %eax, 8(%esp)
7    leal  -8(%ebp), %eax
8    movl  %eax, 4(%esp)
9    movl  $.LC0, (%esp)  \(\text{Pointer to string "%x \%x"}\)
10   call  scanf
```

\(\text{Diagram stack frame at this point}\)

```
11   movl  -4(%ebp), %eax
12   subl  -8(%ebp), %eax
13   leave
14   ret
```
### Referencing Examples

<table>
<thead>
<tr>
<th>zip_dig cmu;</th>
<th>1</th>
<th>5</th>
<th>2</th>
<th>1</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>16</td>
<td>20</td>
<td>24</td>
<td>28</td>
<td>32</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>zip_dig mit;</th>
<th>0</th>
<th>2</th>
<th>1</th>
<th>3</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>36</td>
<td>40</td>
<td>44</td>
<td>48</td>
<td>52</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>zip_dig ucb;</th>
<th>9</th>
<th>4</th>
<th>7</th>
<th>2</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>56</td>
<td>60</td>
<td>64</td>
<td>68</td>
<td>72</td>
</tr>
</tbody>
</table>

- **Reference**
  - **Address**
    - `mit[3]`: $36 + 4 \times 3 = 48$
    - `mit[5]`: $36 + 4 \times 5 = 56$
    - `mit[-1]`: $36 + 4 \times (-1) = 32$
    - `cmu[15]`: $16 + 4 \times 15 = 76$
  - **Value**
    - `mit[-1]`: 3
  - **Guaranteed?**
    - `mit[3]`: Yes
    - `mit[5]`: No
    - `mit[-1]`: No
    - `cmu[15]`: No

- No bound checking
- Out of range behavior implementation-dependent
- No guaranteed relative allocation of different arrays
Optimizing Dynamic Array Multiplication

- **Optimizations**
  - Performed when set optimization level to `-O2`
- **Code Motion**
  - Expression `i*n` can be computed outside loop
- **Strength Reduction**
  - Incrementing `j` has effect of incrementing `j*n+k` by `n`
- **Operations count**
  - 4 adds, 1 mult
- **Compiler can optimize regular access patterns**

```
{ int j;
  int result = 0;
  for (j = 0; j < n; j++)
    result +=
      a[i*n+j] * b[j*n+k];
  return result;
}

{ int j;
  int result = 0;
  int iTn = i*n;
  int jTnPk = k;
  for (j = 0; j < n; j++) {
    result +=
      a[iTn+j] * b[jTnPk];
    jTnPk += n;
  }
  return result;
}
```
Structures

```c
struct rec {
    int i;
    int a[3];
    int *p;
};
```

- **Concept**
  - Contiguously-allocated region of memory
  - Refer to members within structure by names
  - Members may be of different types

- **Accessing Structure Member**

```c
void set_i(struct rec *r, int val) {
    r->i = val;
}
```

**IA32 Assembly**

```
# %eax = val
# %edx = r
movl %eax, (%edx)  # Mem[r] = val
```
Generating Pointer to Structure Member

```c
struct rec {
    int i;
    int a[3];
    int *p;
};
```

- Generating Pointer to Array Element
  - Offset of each structure member determined at compile time

```c
int *find_a
    (struct rec *r, int idx)
{
    return &r->a[idx];
}
```

```
# %ecx = idx
# %edx = r
leal 0(%ecx,4),%eax  # 4*idx
leal 4(%eax,%edx),%eax  # r+4*idx+4
```
Structure Referencing (Cont.)

- C Code

```c
struct rec {
    int i;
    int a[3];
    int *p;
};

void set_p(struct rec *r) {
    r->p = &r->a[r->i];
}
```

```assembly
# %edx = r
movl (%edx),%ecx     # r->i
leal 0(%ecx,4),%eax  # 4*(r->i)
leal 4(%edx,%eax),%eax # r+4+4*(r->i)
movl %eax,16(%edx)   # Update r->p
```
Union Allocation

- Allocate according to largest element
- Can only use ones field at a time

```c
union U1 {
    char c;
    int i[2];
    double v;
} *up;

struct S1 {
    char c;
    int i[2];
    double v;
} *sp;
```
Using Union to Access Bit Patterns

typedef union {
    float f;
    unsigned u;
} bit_float_t;

float bit2float(unsigned u)
{
    bit_float_t arg;
    arg.u = u;
    return arg.f;
}

unsigned float2bit(float f)
{
    bit_float_t arg;
    arg.f = f;
    return arg.u;
}

Same as (float) u ?

Same as (unsigned) f ?
Alignment

• Aligned Data
  – Primitive data type requires K bytes
  – Address must be multiple of K
  – Required on some machines; advised on IA32
    • treated differently by IA32 Linux, x86-64 Linux, and Windows!

• Motivation for Aligning Data
  – Memory accessed by (aligned) chunks of 4 or 8 bytes
    (system dependent)
    • Inefficient to load or store datum that spans quad word boundaries
    • Virtual memory very tricky when datum spans 2 pages

• Compiler
  – Inserts gaps in structure to ensure correct alignment of fields
Specific Cases of Alignment (IA32)

• **1 byte**: `char, ...`
  – no restrictions on address

• **2 bytes**: `short, ...`
  – lowest 1 bit of address must be $0_2$

• **4 bytes**: `int, float, char *, ...`
  – lowest 2 bits of address must be $00_2$

• **8 bytes**: `double, ...`
  – Windows (and most other OS’s & instruction sets):
    • lowest 3 bits of address must be $000_2$
  – Linux:
    • lowest 2 bits of address must be $00_2$
    • i.e., treated the same as a 4-byte primitive data type

• **12 bytes**: `long double`
  – Windows, Linux:
    • lowest 2 bits of address must be $00_2$
    • i.e., treated the same as a 4-byte primitive data type
Different Alignment Conventions

- IA32 Linux
  - $K = 4$; `double` treated like a 4-byte data type

```c
struct S1 {
    char c;
    int i[2];
    double v;
} *p;
```
Arrays of Structures

• Satisfy alignment requirement for every element

```c
struct S2 {
    double v;
    int i[2];
    char c;
} a[10];
```
Accessing Array Elements

• Compute array offset 12i
• Compute offset 8 with structure
• Assembler gives offset a+8
  – Resolved during linking

```
short get_j(int idx)
{
    return a[idx].j;
}
```

```
struct S3 {
    short i;
    float v;
    short j;
} a[10];
```
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

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

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

**Initial part of sfact**

```assembly
_sfact:
    pushl %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
    pushl %ebp
    movl %esp,%ebp
    subl $16,%esp
```
Passing Pointer

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

Calling s_helper from sfact

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
And who thought this was a good idea?

- Pointers!
- Pointers to functions!?
  - void (*foo)(int);
  - foo is a pointer to a function taking one argument, an integer, and that returns void
  - void *( *foo)(int *);
  - foo is a pointer to a function that returns a void * (e.g., a pointer to an unknown data type) and takes an int *

- Pointers to arrays of functions!?!?
- Pointers to pointers to functions with pointers!?!?
Nested Array Example

```c
#define PCOUNT 4
zip_dig pgh[PCOUNT] =
    {{1, 5, 2, 0, 6},
     {1, 5, 2, 1, 3 },
     {1, 5, 2, 1, 7 },
     {1, 5, 2, 2, 1 }};
```

- "`zip_dig pgh[4]`" equivalent to "`int pgh[4][5]`"
  - Variable `pgh`: array of 4 elements, allocated contiguously
  - Each element is an array of 5 `int`'s, allocated contiguously

- "Row-Major" ordering of all elements guaranteed
Multidimensional (Nested) Arrays

• Declaration
  \[ T \ A[R][C]; \]
  – 2D array of data type \( T \)
  – \( R \) rows, \( C \) columns
  – Type \( T \) element requires \( K \) bytes

• Array Size
  – \( R \times C \times K \) bytes

• Arrangement
  – Row-Major Ordering

\[
\begin{array}{cccc}
A[0][0] & \cdots & A[0][C-1] \\
\vdots & \ddots & \vdots \\
A[R-1][0] & \cdots & A[R-1][C-1]
\end{array}
\]
Nested Array Row Access

- **Row Vectors**
  - $A[i]$ is array of $C$ elements
  - Each element of type $T$ requires $K$ bytes
  - Starting address $A + i \times (C \times K)$

```c
int A[R][C];
```

---

The diagram illustrates the memory layout and address calculation for row-major access in a nested array. Each row is represented as a vector of $C$ elements, with each element requiring $K$ bytes. The starting address for each row is calculated as $A + i \times (C \times K)$.
Nested Array Row Access Code

```c
int *get_pgh_zip(int index) {
    return pgh[index];
}
```

```c
#define PCOUNT 4
zip_dig pgh[PCOUNT] = {
    {1, 5, 2, 0, 6},
    {1, 5, 2, 1, 3 },
    {1, 5, 2, 1, 7 },
    {1, 5, 2, 2, 1 };
```

```assembly
# %eax = index
leal (%eax,%eax,4),%eax
leal pgh(,%eax,4),%eax
```
Nested Array Row Access Code

```c
int *get_pgh_zip(int index)
{
    return pgh[index];
}
```

```c
#define PCOUNT 4
zip_digits pgh[PCOUNT] =
{{1, 5, 2, 0, 6},
 {1, 5, 2, 1, 3 },
 {1, 5, 2, 1, 7 },
 {1, 5, 2, 2, 1 }};
```

```c
#define PCOUNT 4
zip_digits pgh[PCOUNT] =
{{1, 5, 2, 0, 6},
 {1, 5, 2, 1, 3 },
 {1, 5, 2, 1, 7 },
 {1, 5, 2, 2, 1 }};
```

• **Row Vector**
  – `pgh[index]` is array of 5 `int`'s
  – Starting address `pgh + 20 * index`

• **IA32 Code**
  – Computes and returns address
  – Compute as `pgh + 4 * (index + 4 * index)`
Nested Array Row Access

• Array Elements
  – $\text{A}[i][j]$ is element of type $T$, which requires $K$ bytes
  – Address $A + i \times (C \times K) + j \times K = A + (i \times C + j) \times K$

```c
int A[R][C];
```

\[
\begin{array}{c}
\text{A[0]}
\end{array}
\]

\[
\begin{array}{c}
\text{A[0][0]}
\end{array}
\]

\[
\begin{array}{c}
\text{A[0][C-1]}
\end{array}
\]

\[
\begin{array}{c}
\text{A[i]}
\end{array}
\]

\[
\begin{array}{c}
\text{A[i][j]}
\end{array}
\]

\[
\begin{array}{c}
\text{A[R-1][0]}
\end{array}
\]

\[
\begin{array}{c}
\text{A[R-1][C-1]}
\end{array}
\]

\[
\begin{array}{c}
\text{A+i*C*4}
\end{array}
\]

\[
\begin{array}{c}
\text{A+(R-1)*C*4}
\end{array}
\]

\[
\begin{array}{c}
\text{A+i*C*4+j*4}
\end{array}
\]
Nested Array Element Access Code

```c
int get_pgh_digit
    (int index, int dig)
{
    return pgh[index][dig];
}
```

- **Array Elements**
  - `pgh[index][dig]` is int
  - Address: `pgh + 20*index + 4*dig`

- **IA32 Code**
  - Computes address `pgh + 4*dig + 4*(index+4*index)`
  - `movl` performs memory reference
Strange Referencing Examples

```c
zip_dig
pgh[4];
```

- **Reference Address**
  - `pgh[3][3]`
  - `pgh[2][5]`
  - `pgh[2][-1]`
  - `pgh[4][-1]`
  - `pgh[0][19]`
  - `pgh[0][-1]`

- **Value Guaranteed?**
  - `76 + 20*3 + 4*3 = 148`
  - `76 + 20*2 + 4*5 = 136`
  - `76 + 20*2 + 4*-1 = 112`
  - `76 + 20*4 + 4*-1 = 152`
  - `76 + 20*0 + 4*19 = 152`
  - `76 + 20*0 + 4*-1 = 72`
Strange Referencing Examples

• Reference Address

```
pgh[4];  
zip_dig
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Guaranteed?</th>
</tr>
</thead>
<tbody>
<tr>
<td>pgh[3][3]</td>
<td>76+20<em>3+4</em>3 = 148</td>
<td>Yes</td>
</tr>
<tr>
<td>pgh[2][5]</td>
<td>2</td>
<td>Yes</td>
</tr>
<tr>
<td>pgh[2][-1]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pgh[4][-1]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pgh[0][19]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pgh[0][-1]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Strange Referencing Examples

<table>
<thead>
<tr>
<th>Reference</th>
<th>Address</th>
<th>Value</th>
<th>Guaranteed?</th>
</tr>
</thead>
<tbody>
<tr>
<td>pgh[3][3]</td>
<td>76+20<em>3+4</em>3 = 148</td>
<td>2</td>
<td>Yes</td>
</tr>
<tr>
<td>pgh[2][5]</td>
<td>76+20<em>2+4</em>5 = 136</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td>pgh[2][-1]</td>
<td>76+20<em>2+4</em>-1 = 112</td>
<td>3</td>
<td>Yes</td>
</tr>
<tr>
<td>pgh[4][-1]</td>
<td>76+20<em>4+4</em>-1 = 152</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td>pgh[0][19]</td>
<td>76+20<em>0+4</em>19 = 152</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td>pgh[0][-1]</td>
<td>76+20<em>0+4</em>-1 = 72</td>
<td>??</td>
<td>No</td>
</tr>
</tbody>
</table>

– Code does not do any bounds checking
– Ordering of elements within array guaranteed