Last Time

- **For loops**
  - for loop → while loop → do-while loop → goto version
  - for loop → while loop → goto “jump to middle” version

- **Switch statements**
  - Jump tables: `jmp *.L62 (%edx,4)`
  - Decision trees (not shown) when we have *sparse* cases

```
jmp * .L62 (%edx,4)
```

Jump table

```
.section .rodata
.align 4
.L62:
.long .L61  # x = 0
.long .L56  # x = 1
.long .L57  # x = 2
.long .L58  # x = 3
.long .L61  # x = 4
.long .L60  # x = 5
.long .L60  # x = 6
```
Procedures/Stacks

- Stacks
- Procedures
- Parameter passing
Memory Layout

- **Instructions**: Read-only; executable. Execution lifetime; immutable.
- **Literals**: Read-only; not executable. Execution lifetime; immutable.
- **Static Data**: Writable; not executable. Execution lifetime; mutable.
- **Dynamic Data (Heap)**: Writable; not executable. Programmer controlled lifetime; mutable.
- **Stack**: Writable; not executable. “Automatic” lifetime; mutable.

*Note: executability of data areas is system dependent...*
Memory Layout

- **Instructions**
- **Literals**
- **Static Data** (including global variables (C))
- **Dynamic Data (Heap)**
- **Stack**

- Literals (e.g., “example”)
- *static* variables
- *new*ed variables
- Local variables

Diagram:

```
 0

Instructions

Literals

Static Data

Dynamic Data (Heap)

Stack

2^{N-1}
```
IA32 Stack

- Region of memory managed with a stack discipline
- Grows toward lower addresses
- Customarily shown “upside-down”

- Register `%esp` contains lowest stack address = address of “top” element

Stack Pointer: `%esp`
IA32 Stack: Push

- pushl Src

Stack Pointer: %esp

Stack "Bottom"

Increasing Addresses

Stack Grows Down

Stack "Top"
IA32 Stack: Push

- `pushl Src`
  - Fetch operand at `Src`
  - Decrement `%esp` by 4
  - Write operand at address given by `%esp`

Stack Pointer: `%esp`
IA32 Stack: Pop

- `popl Dest`

Stack Pointer: `%esp`

Stack "Bottom"

- Increasing Addresses
- Stack Grows Down

Stack "Top"

+4
IA32 Stack: Pop

- `popl Dest`
  - Read operand at address `%esp`
  - Increment `%esp` by 4
  - Write operand to `Dest`
Procedure Control Flow

- Use stack to support procedure call and return
- **Procedure call:** `call label`
  - Push return address on stack
  - Jump to `label`
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:**
  - Address of instruction beyond `call`
  - Example from disassembly
    
    | Address  | Instruction          |     |
    |----------|----------------------|-----|
    | 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

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>804854e:</td>
<td>e8 3d 06 00 00</td>
<td>call 8048b90 &lt;main&gt;</td>
</tr>
<tr>
<td>8048553:</td>
<td>50</td>
<td>pushl %eax</td>
</tr>
</tbody>
</table>

**Diagram:**

- **%esp:** 0x108
- **%eip:** 0x804854e
- **%esp:** 0x104
- **%eip:** 0x8048553

**Program Counter:**

- 0x108
- 0x10c
- 0x110

- **pushl %eax**:
  - Store the value of %eax to the stack.

- **call 8048b90**:
  - Call the function at 8048b90.

**%esp**:

- Initial value: 0x108
- After pushl: 0x104

**%eip**:

- Initial value: 0x804854e
- After call: 0x8048553

---

*%eip: program counter*
Procedure Call Example

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

8048591: c3

\[
\begin{array}{l}
0x110 \\
0x10c \\
0x108 \\
0x104 \\
%esp \\
%eip
\end{array}
\begin{array}{l}
%esp \\
%eip
\end{array}
\]

ret

\[
\begin{array}{l}
0x110 \\
0x10c \\
0x108 \\
0x104 \\
%esp \\
%eip
\end{array}
\begin{array}{l}
%esp \\
%eip
\end{array}
\]

ret

\[
\begin{array}{l}
0x8048553 \\
123
\end{array}
\]

\[
\begin{array}{l}
0x8048553 \\
123
\end{array}
\]

%eip: program counter
Stack-Based Languages

- **Languages that support recursion**
  - e.g., C, Pascal, Java
  - Code must be *re-entrant*
    - Multiple simultaneous instantiations of single procedure
  - Need some place to store state of each instantiation
    - Arguments
    - Local variables
    - Return pointer

- **Stack discipline**
  - State for a given procedure needed for a limited time
    - Starting from when it is called to when it returns
    - Callee always returns before caller does

- **Stack allocated in frames**
  - State for a single procedure instantiation
Call Chain Example

Procedure `amI` is recursive (calls itself)
Stack Frames

Contents
- Local variables
- Return information
- Temporary space

Management
- Space allocated when procedure is entered
  - “Set-up” code
- Space deallocated upon return
  - “Finish” code
Example

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

Stack

```
%ebp

%esp

yoo

amI

amI

amI

amI
```
Example

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

```
Stack
```

```
who
```

```
ymoo
```

```
%ebp
```

```
%esp
```

```
amI
```

```
amI
```

```
amI
```

```
amI
```
Example

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

Stack

```
%ebp
| %esp
| amI
| who
| yoo
```

Diagram:
- `amI(...)`
- `amI();
  - amI
  - amI
  - amI
- Stack:
  - `%ebp`
  - `%esp`
  - `amI`
  - `who`
  - `yoo`
Example

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

Stack

```
%ebp
%esp
amI
amI
who
yoo
```
Example

```
ami (...) {
    •
    •
    ami();
    •
    •
}
```

Stack

- yoo
- who
- ami
- ami
- ami
- ami
- %ebp
- %esp
Example

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

Stack

```
%ebp

%esp
```

```
yoo
who
amI
amI
```

```
Example

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

Stack

```
%ebp
%esp
```

- `yoo`
- `who`
- `amI`
- `amI`
- `amI`
- `amI`
Example

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

Stack

```
 Stack:
    yoo
    who
    %ebp
    %esp
```
Example

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

Stack

```
%ebp

%esp

yoo

who

amI

amI

amI

amI
```
Example

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

Stack

```
Stack

yoo

%ebp

%esp

who
```
Example

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

Stack

```
yoo
%ebp
%esp
```
IA32/Linux Stack Frame

**Current Stack Frame (“Top” to Bottom)**
- Old frame pointer
- Local variables
  - If can’t be just kept in registers
- Saved register context
  - When reusing registers
- “Argument build area”
  - Parameters for function about to be called

**Caller Stack Frame**
- Return address
  - Pushed by `call` instruction
- Arguments for this call
**Revisiting swap**

```c
int zip1 = 15213;
int zip2 = 98195;

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

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

**Calling swap from call_swap**

```
call_swap:
    . . .
pushl $zip2     # Global Var
pushl $zip1     # Global Var
call swap
    . . .
```

**Resulting Stack**

```
      .
      .
      .
  &zip2
  &zip1
  Rtn adr
      %esp
```
Revisiting swap

```c
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

**swap:**

- pushl %ebp
- movl %esp,%ebp
- pushl %ebx
swap Setup #1

Entering Stack

\[
\begin{align*}
\text{swap:} & \quad \text{pushl}\ %\text{ebp} \\
& \quad \text{movl}\ %\text{esp},%\text{ebp} \\
& \quad \text{pushl}\ %\text{ebx}
\end{align*}
\]
swap Setup #1

Entering Stack

Resulting Stack

swap:

```
pushl %ebp
movl %esp,%ebp
pushl %ebx
```
swap Setup #1

Entering Stack

\[
\begin{align*}
\text{\&zip2} \\
\text{\&zip1} \\
\text{Rtn adr}
\end{align*}
\]

\[
\begin{align*}
\text{\%ebp} \\
\text{\%esp}
\end{align*}
\]

\[
\begin{align*}
\text{\%ebp} \\
\text{\%esp}
\end{align*}
\]

swap:
\[
\begin{align*}
pushl \ %ebp \\
movl \ %esp,\%ebp \\
pushl \ %ebx
\end{align*}
\]
**swap Setup #1**

**Entering Stack**

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

**Resulting Stack**

Offset relative to new `%ebp`

- `yp` (12)
- `xp` (8)
- `Rtn adr`
- `Old %ebp`
- `Old %ebx`

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

swap Finish #1

swap’s Stack

Resulting Stack

Observation: Saved and restored register %ebx
swap Finish #2

swap’s Stack

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

**swap's Stack**

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

**Resulting Stack**

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

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

swap’s Stack

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

swap’s Stack

Resulting Stack

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

**swap’s Stack**

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

swap’s Stack

Resulting Stack

Observation

- Saved & restored register %ebx
- Didn’t do so for %eax, %ecx, or %edx
Disassembled swap

080483a4 <swap>:

080483a4:   55          push   %ebp
080483a5:   89 e5       mov    %esp,%ebp
080483a7:   53          push   %ebx
080483a8:   8b 55 08    mov    0x8(%ebp),%edx
080483ab:   8b 4d 0c    mov    0xc(%ebp),%ecx
080483ae:   8b 1a       mov    (%edx),%ebx
080483b0:   8b 01       mov    (%ecx),%eax
080483b2:   89 02       mov    %eax,(%edx)
080483b4:   89 19       mov    %ebx,(%ecx)
080483b6:   5b          pop    %ebx
080483b7:   c9          leave
080483b8:   c3          ret

Calling Code

08048409:   e8 96 ff ff ff ff       call 80483a4 <swap>
0804840e:   8b 45 f8       mov    0xfffffffff8(%ebp),%eax

0x0804840e + 0xffffffff96 = 0x080483a4
Register Saving Conventions

- When procedure `yoo` calls `who`:
  - `yoo` is the *caller*
  - `who` is the *callee*

- Can Register be used for temporary storage?

  ```
  yoo:
  • • •
  movl $15213, %edx
  call who
  addl %edx, %eax
  • • •
  ret

  who:
  • • •
  movl 8(%ebp), %edx
  addl $98195, %edx
  • • •
  ret
  ```

- Contents of register `%edx` overwritten by `who`
Register Saving Conventions

- When procedure \textit{yoo} calls \textit{who}:
  - \textit{yoo} is the \textit{caller}
  - \textit{who} is the \textit{callee}

- Can register be used for temporary storage?
- Conventions
  - “\textit{Caller Save}”
    - Caller saves temporary in its frame before calling
  - “\textit{Callee Save}”
    - Callee saves temporary in its frame before using
IA32/Linux Register Usage

- **%eax, %edx, %ecx**
  - Caller saves prior to call if values are used later

- **%eax**
  - also used to return integer value

- **%ebx, %esi, %edi**
  - Callee saves if wants to use them

- **%esp, %ebp**
  - special
Recursive Factorial

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

**Registers**

- `%ebx` used, but saved at beginning & restored at end
- `%eax` used without first saving
  - expect caller to save
  - pushed onto stack as parameter for next call
  - used for return value

```assembly
rfact:
pushl %ebp
movl %esp,%ebp
pushl %ebx
movl 8(%ebp),%ebx
cmpl $1,%ebx
jle .L78
leal -1(%ebx),%eax
pushl %eax
call rfact
imull %ebx,%eax
jmp .L79
.align 4
.L78:
movl $1,%eax
.L79:
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

```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 $-4 \times \%ebp$
- Push on stack as second argument

---

Initial part of `sfact`

```
_sfact:
pushl %ebp
    movl %esp,%ebp
    subl $16,%esp
    movl 8(%ebp),%edx
    movl $1,-4(%ebp)
```

---

```
x
    8  Rtn adr
    4  Old %ebp
    0  val = 1
    -4
    -8
    -12
    -16
           Unused
```

- $\%ebp$
- $\%esp$
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**

```
_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
```

<table>
<thead>
<tr>
<th>x</th>
<th>Rtn adr</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Old %ebp</td>
</tr>
<tr>
<td>4</td>
<td>val = 1</td>
</tr>
<tr>
<td>8</td>
<td>Unused</td>
</tr>
</tbody>
</table>

%ebp and %esp
Passing Pointer

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

Calling `s_helper` from `sfact`

```
leal -4(%ebp),%eax
pushl %eax
pushl %edx
    call s_helper
movl -4(%ebp),%eax
    . . .
```

Stack at time of call

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>x</td>
</tr>
<tr>
<td>4</td>
<td>Rtn adr</td>
</tr>
<tr>
<td>0</td>
<td>Old %ebp</td>
</tr>
<tr>
<td>-4</td>
<td>val=x!</td>
</tr>
<tr>
<td>-8</td>
<td>Unused</td>
</tr>
<tr>
<td>-12</td>
<td></td>
</tr>
<tr>
<td>-16</td>
<td>&amp;val</td>
</tr>
</tbody>
</table>

Variables:
- `%ebp`:
- `%esp`:
Passing Pointer

```c
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:

- 8: `x`
- 4: Rtn adr
- 0: Old %ebp
- 4: Unused
- 8: val=x!
- 12: &val
- 16: x
IA 32 Procedure Summary

- 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