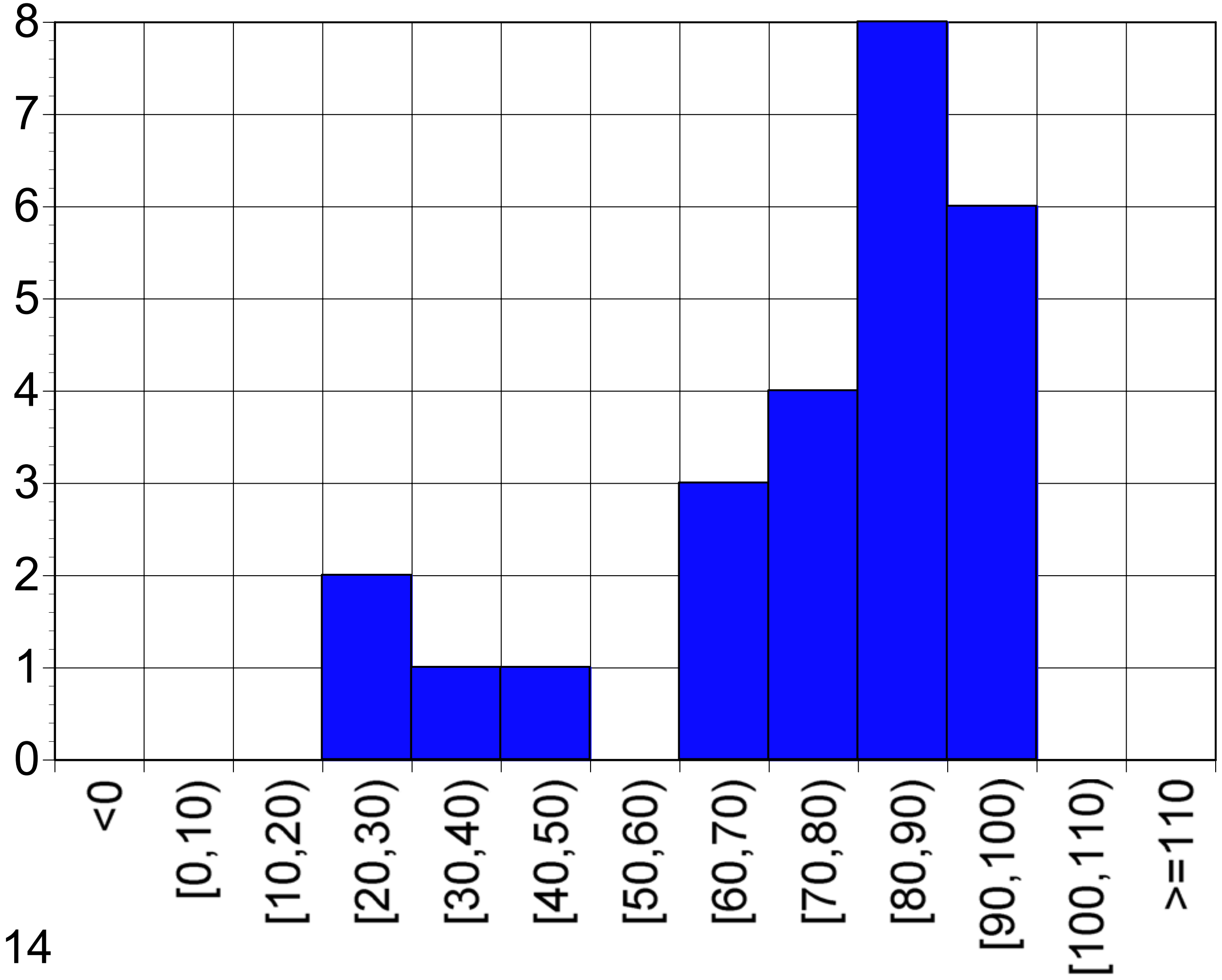
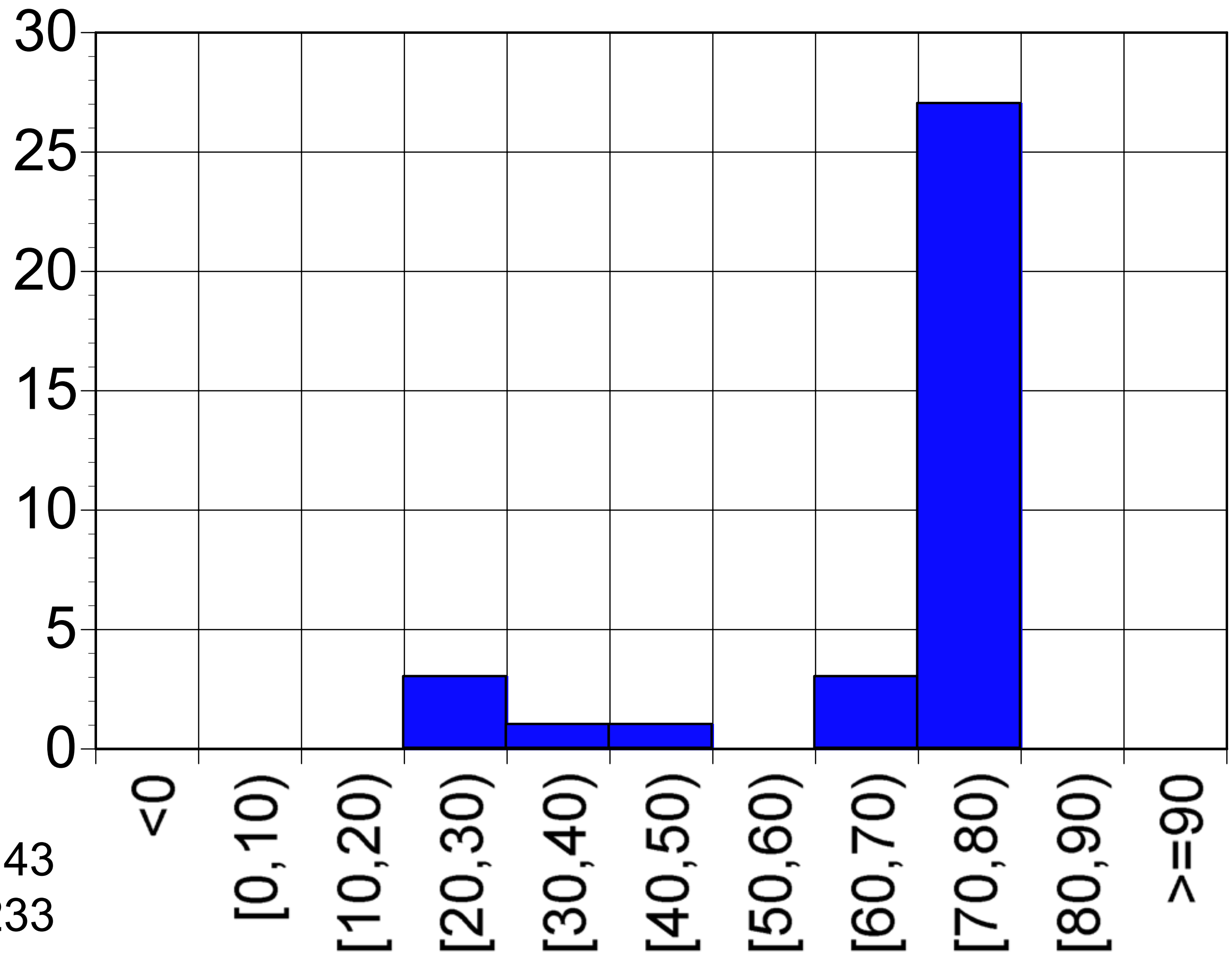



**Homework 1**

| | |
|---|---|
| Count | 25 |
| Mean | 73.8 |
| Std | 21.12414 |
| Min | 22 |
| Max | 96 |
| Mode | 66 |
| Median | 80.5 |




**Datalab**

| | |
|---|---|
| Count | 35 |
| Mean | 67.92857143 |
| Std | 15.14447233 |
| Min | 27 |
| Max | 75 |
| Mode | 75 |
| Median | 75 |

# Today

**Control flow**

- **if/while/do while/for/switch**

**Maybe start on procedures**

- **Stack discipline**
- **Stack-based languages and call chains**
- **Stack frames**

# Condition Codes

## Single Bit Registers

| | |
|---|---|
| `CF` | Carry Flag |
| `ZF` | Zero Flag |
| `SF` | Sign Flag |
| `OF` | Overflow Flag |

## Implicit Setting By Arithmetic Operations

`addl` *Src*,*Dest*

C analog: `t = a+b`

- CF set if carry out from most significant bit
  - Used to detect unsigned overflow
- ZF set if `t == 0`
- SF set if `t < 0`
- OF set if two's complement overflow

  `(a>0 && b>0 && t<0) || (a<0 && b<0 && t>0)`

## *Not* Set by `leal` instruction

# Setting Condition Codes (cont.)

## Explicit Setting by Compare Instruction

`cmpl` *Src2,Src1* *Note Inverted Order*

- `cmpl b,a` like computing `a-b` without setting destination
- CF set if carry out from most significant bit
  - Used for unsigned comparisons
- ZF set if `a == b`
- SF set if `(a-b) < 0`
- OF set if two's complement overflow

  `(a>0 && b<0 && (a-b)<0) || (a<0 && b>0 && (a-b)>0)`

## Explicit Setting by Test instruction

`testl` *Src2,Src1*

- Sets condition codes based on value of *Src1* & *Src2*
  - Useful to have one of the operands be a mask
- `testl b,a` like computing `a&b` without setting destination
- ZF set when `a&b == 0`
- SF set when `a&b < 0`

# Reading Condition Codes

## SetX *Dest* Instructions

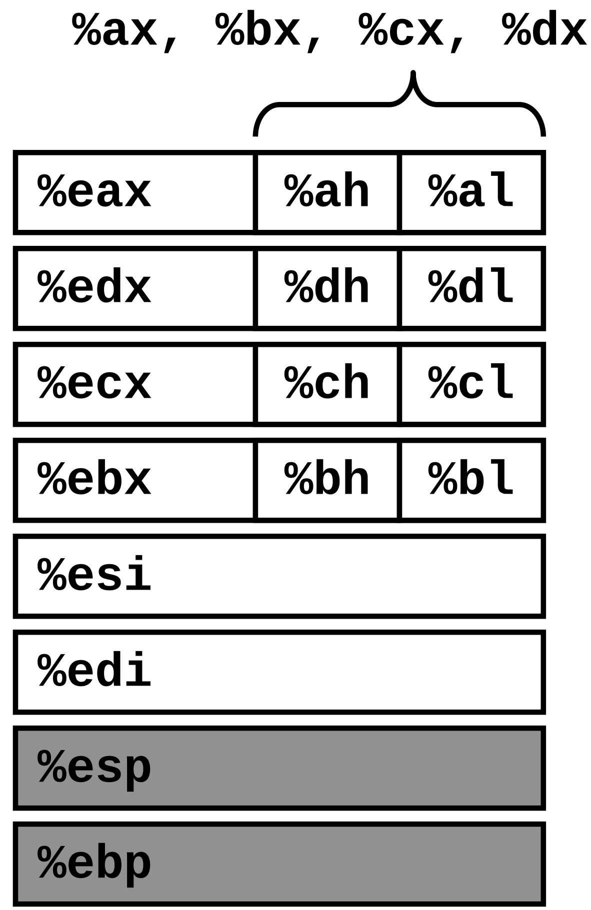
- **Set single byte based on combinations of condition codes**

| SetX | Condition | Description |
|---|---|---|
| `sete` | ZF | Equal / Zero |
| `setne` | ~ZF | Not Equal / Not Zero |
| `sets` | SF | Negative |
| `setns` | ~SF | Nonnegative |
| `setg` | `~(SF^OF)&~ZF` | Greater (Signed) |
| `setge` | `~(SF^OF)` | Greater or Equal (Signed) |
| `setl` | `(SF^OF)` | Less (Signed) |
| `setle` | `(SF^OF)\|ZF` | Less or Equal (Signed) |
| `seta` | ~CF&~ZF | Above (unsigned) |
| `setb` | CF | Below (unsigned) |

# Reading Condition Codes (Cont.)

## SetX Instructions

- **Set single byte based on combinations of condition codes**
- **One of 8 addressable byte registers**
  - Embedded within first 4 integer registers
  - Does not alter remaining 3 bytes
  - Typically use `andl 0xFF,%eax` to finish job




```
int gt (int x, int y)
{
  return x > y;
}
```

**Body**

```
movl 12(%ebp),%eax   # eax = y
cmpl %eax,8(%ebp)    # Compare x : eax
setg %al             # al = x > y
andl $255,%eax       # Zero rest of %eax
```

**Note inverted ordering!**
**Cmpl y,x => (x-y)**

# Jumping – Goto instructions

## jX Instructions

- Jump to different part of code depending on condition codes

| jX | Condition | Description |
|---|---|---|
| `jmp` | `1` | Unconditional |
| `je` | `ZF` | Equal / Zero |
| `jne` | `~ZF` | Not Equal / Not Zero |
| `js` | `SF` | Negative |
| `jns` | `~SF` | Nonnegative |
| `jg` | `~(SF^OF)&~ZF` | Greater (Signed) |
| `jge` | `~(SF^OF)` | Greater or Equal (Signed) |
| `jl` | `(SF^OF)` | Less (Signed) |
| `jle` | `(SF^OF)\|ZF` | Less or Equal (Signed) |
| `ja` | `~CF&~ZF` | Above (unsigned) |
| `jb` | `CF` | Below (unsigned) |

# Conditional Branch Example

```c
int max(int x, int y)
{
  if (x > y)
    return x;
  else
    return y;
}
```

```
_max:
      pushl %ebp            } Set
      movl %esp,%ebp        } Up

      movl 8(%ebp),%edx     }
      movl 12(%ebp),%eax    }
      cmpl %eax,%edx        } Body
      jle L9                }
      movl %edx,%eax        }
L9:

      movl %ebp,%esp        }
      popl %ebp             } Finish
      ret                   }
```

# Conditional Branch Example (Cont.)

```
int goto_max(int x, int y)
{
  int rval = y;
  int ok = (x <= y);
  if (ok)
    goto done;
  rval = x;
done:
  return rval;
}
```

- **C allows "goto" as means of transferring control**
  - Closer to machine-level programming style
- **Generally considered bad coding style**
- **Machine only does gotos**
- **Compiler tries to have one return**

```
    movl 8(%ebp),%edx    # edx = x
    movl 12(%ebp),%eax   # eax = y
    cmpl %eax,%edx       # x : y
    jle L9               # if <= goto L9
    movl %edx,%eax       # eax = x   } Skipped when x ≤ y
  L9:                  # Done:
```

# "Do-While" Loop Example

**C Code**

```
int fact_do
  (int x)
{
  int result = 1;
  do {
    result *= x;
    x = x-1;
  } while (x > 1);
  return result;
}
```

**Goto Version**

```
int fact_goto(int x)
{
  int result = 1;
loop:
  result *= x;
  x = x-1;
  if (x > 1)
    goto loop;
  return result;
}
```

- **Use backward branch to continue looping**
- **Only take branch when "while" condition holds**

# "Do-While" Loop Compilation

## Goto Version

```
int fact_goto
  (int x)
{
  int result = 1;
loop:
  result *= x;
  x = x-1;
  if (x > 1)
    goto loop;
  return result;
}
```

## Registers

```
%edx   x
%eax   result
```

## Assembly

```
_fact_goto:
  pushl %ebp            # Setup
  movl %esp,%ebp        # Setup

  movl $1,%eax          # eax = 1
  movl 8(%ebp),%edx     # edx = x

L11:
  imull %edx,%eax       # result *= x
  decl %edx             # x--
  cmpl $1,%edx          # Compare x : 1
  jg L11                # if > goto loop

  movl %ebp,%esp        # Finish
  popl %ebp             # Finish
  ret                   # Finish
```

# General "Do-While" Translation

## C Code

```
do
  Body
  while (Test);
```

## Goto Version

```
loop:
  Body
  if (Test)
    goto loop
```

- ***Body* can be any C statement**
  - Typically compound statement:

```
{
  Statement_1;
  Statement_2;
    …
  Statement_n;
}
```

- ***Test* is expression returning integer**

  = 0 interpreted as false      ≠0 interpreted as true

# "While" Loop Example #1

## C Code

```
int fact_while
  (int x)
{
  int result = 1;
  while (x > 1) {
    result *= x;
    x = x-1;
  };
  return result;
}
```

## First Goto Version

```
int fact_while_goto
  (int x)
{
  int result = 1;
loop:
  if (!(x > 1))
    goto done;
  result *= x;
  x = x-1;
  goto loop;
done:
  return result;
}
```

- **Is this code equivalent to the do-while version?**
- **Must jump out of loop if test fails**

# Actual "While" Loop Translation

## C Code

```
int fact_while(int x)
{
  int result = 1;
  while (x > 1) {
    result *= x;
    x = x-1;
  };
  return result;
}
```

- Uses same inner loop as do-while version
- Guards loop entry with extra test

## Second Goto Version

```
int fact_while_goto2
  (int x)
{
  int result = 1;
  if (!(x > 1))
    goto done;
loop:
  result *= x;
  x = x-1;
  if (x > 1)
    goto loop;
done:
  return result;
}
```

# General “While” Translation

## C Code

```
while (Test)
  Body
```

## Do-While Version

```
  if (!Test)
    goto done;
  do
    Body
    while(Test);
done:
```

## Goto Version

```
  if (!Test)
    goto done;
loop:
  Body
  if (Test)
    goto loop;
done:
```

# "For" Loop Example

```
int result;
for (result = 1;
     p != 0;
     p = p>>1) {
  if (p & 0x1)
    result *= x;
  x = x*x;
}
```

## General Form

for (*Init*; *Test*; *Update* )

*Body*

***Init***

```
result = 1
```

***Test***

```
p != 0
```

***Update***

```
p = p >> 1
```

***Body***

```
{
  if (p & 0x1)
    result *= x;
  x = x*x;
}
```

# "For"→ "While"

## For Version

```
for (Init; Test; Update )
    Body
```

## While Version

```
Init;
while (Test ) {
    Body
    Update ;
}
```

## Do-While Version

```
  Init;
  if (!Test)
    goto done;
  do {
    Body
    Update ;
  } while (Test)
done:
```

## Goto Version

```
  Init;
  if (!Test)
    goto done;
loop:
  Body
  Update ;
  if (Test)
    goto loop;
done:
```

# "For" Loop Compilation

## Goto Version

```
  Init;
  if (!Test)
    goto done;
loop:
  Body
  Update ;
  if (Test)
    goto loop;
done:
```

```
  result = 1;
  if (p == 0)
    goto done;
loop:
  if (p & 0x1)
    result *= x;
  x = x*x;
  p = p >> 1;
  if (p != 0)
    goto loop;
done:
```

***Init***

```
result = 1
```

***Test***

```
p != 0
```

***Update***

```
p = p >> 1
```

***Body***

```
{
  if (p & 0x1)
    result *= x;
  x = x*x;
}
```

# Switch Statements

```
typedef enum
 {ADD, MULT, MINUS, DIV, MOD, BAD}
    op_type;

char unparse_symbol(op_type op)
{
  switch (op) {
  case ADD :
    return '+';
  case MULT:
    return '*';
  case MINUS:
    return '-';
  case DIV:
    return '/';
  case MOD:
    return '%';
  case BAD:
    return '?';
  }
}
```

## Implementation Options

- **Series of conditionals**
  - Good if few cases
  - Slow if many
- **Jump Table**
  - Lookup branch target
  - Avoids conditionals
  - Possible when cases are small integer constants
- **GCC**
  - Picks one based on case structure
- **Bug in example code**
  - No default given

# Jump Table Structure

## Switch Form

```
switch(op) {
  case 0:
    Block 0
  case 1:
    Block 1
    • • •
  case n-1:
    Block n–1
}
```

## Jump Table

jtab:

- Targ0
- Targ1
- Targ2
- •
- •
- •
- Targn-1

## Jump Targets

Targ0: Code Block 0

Targ1: Code Block 1

Targ2: Code Block 2

•
•
•

Targn-1: Code Block *n*–1

## Approx. Translation

```
target = JTab[op];
goto *target;
```

# Switch Statement Example

## Branching Possibilities

```
typedef enum
  {ADD, MULT, MINUS, DIV, MOD,
BAD}
    op_type;

char unparse_symbol(op_type op)
{
  switch (op) {
    • • •
  }
}
```

## Enumerated Values

```
ADD   0
MULT  1
MINUS 2
DIV   3
MOD   4
BAD   5
```

**Setup:**

```
unparse_symbol:
 pushl %ebp            # Setup
 movl %esp,%ebp        # Setup
 movl 8(%ebp),%eax     # eax = op
 cmpl $5,%eax          # Compare op : 5
 ja .L49               # If > goto done
 jmp *.L57(,%eax,4)    # goto Table[op]
```

# Assembly Setup Explanation

## Symbolic Labels

- **Labels of form `.L`*XX* translated into addresses by assembler**

## Table Structure

- **Each target requires 4 bytes**
- **Base address at `.L57`**

## Jumping

`jmp .L49`

- Jump target is denoted by label `.L49`

`jmp *.L57(,%eax,4)`

- Start of jump table denoted by label `.L57`
- Register `%eax` holds `op`
- Must scale by factor of 4 to get offset into table
- Fetch target from effective Address `.L57 + op*4`

# Jump Table

## Table Contents

```
.section .rodata
   .align 4
.L57:
  .long .L51 #Op = 0
  .long .L52 #Op = 1
  .long .L53 #Op = 2
  .long .L54 #Op = 3
  .long .L55 #Op = 4
  .long .L56 #Op = 5
```

## Enumerated Values

```
ADD     0
MULT    1
MINUS   2
DIV     3
MOD     4
BAD     5
```

## Targets & Completion

```
.L51:
    movl $43,%eax # '+'
    jmp .L49
.L52:
    movl $42,%eax # '*'
    jmp .L49
.L53:
    movl $45,%eax # '-'
    jmp .L49
.L54:
    movl $47,%eax # '/'
    jmp .L49
.L55:
    movl $37,%eax # '%'
    jmp .L49
.L56:
    movl $63,%eax # '?'
    # Fall Through to .L49
```

# Switch Statement Completion

```
.L49:                       # Done:
    movl %ebp,%esp          # Finish
    popl %ebp               # Finish
    ret                     # Finish
```

## Puzzle

- What value returned when op is invalid?

## Answer

- Register `%eax` set to op at beginning of procedure
- This becomes the returned value

## Advantage of Jump Table

- Can do *k*-way branch in *O*(1) operations

# Object Code

## Setup

- Label `.L49` becomes address `0x804875c`
- Label `.L57` becomes address `0x8048bc0`

```
08048718 <unparse_symbol>:
8048718:55              pushl  %ebp
8048719:89 e5           movl   %esp,%ebp
804871b:8b 45 08        movl   0x8(%ebp),%eax
804871e:83 f8 05        cmpl   $0x5,%eax
8048721:77 39           ja     804875c <unparse_symbol+0x44>
8048723:ff 24 85 c0 8b  jmp    *0x8048bc0(,%eax,4)
```

# Object Code (cont.)

## Jump Table

- **Doesn't show up in disassembled code**
- **Can inspect using GDB**

**`gdb code-examples`**

**`(gdb) x/6xw 0x8048bc0`**

- Examine 6 hexadecimal format "words" (4-bytes each)
- Use command "**`help x`**" to get format documentation

```
0x8048bc0 <_fini+32>:
  0x08048730
  0x08048737
  0x08048740
  0x08048747
  0x08048750
  0x08048757
```

# Extracting Jump Table from Binary

## Jump Table Stored in Read Only Data Segment (.rodata)

- Various fixed values needed by your code

## Can examine with objdump

`objdump code-examples –s --section=.rodata`

- Show everything in indicated segment.

## Hard to read

- Jump table entries shown with reversed byte ordering

```
Contents of section .rodata:
 8048bc0 30870408 37870408 40870408 47870408  0...7...@...G...
 8048bd0 50870408 57870408 46616374 28256429  P...W...Fact(%d)
 8048be0 203d2025 6c640a00 43686172 203d2025   = %ld..Char = %
 …
```

- E.g., `30870408` really means `0x08048730`

# Disassembled Targets

- No-operations (`movl %esi,%esi`) inserted to align target addresses

```
8048730:b8 2b 00 00 00   movl   $0x2b,%eax
8048735:eb 25            jmp    804875c <unparse_symbol+0x44>
8048737:b8 2a 00 00 00   movl   $0x2a,%eax
804873c:eb 1e            jmp    804875c <unparse_symbol+0x44>
804873e:89 f6            movl   %esi,%esi
8048740:b8 2d 00 00 00   movl   $0x2d,%eax
8048745:eb 15            jmp    804875c <unparse_symbol+0x44>
8048747:b8 2f 00 00 00   movl   $0x2f,%eax
804874c:eb 0e            jmp    804875c <unparse_symbol+0x44>
804874e:89 f6            movl   %esi,%esi
8048750:b8 25 00 00 00   movl   $0x25,%eax
8048755:eb 05            jmp    804875c <unparse_symbol+0x44>
8048757:b8 3f 00 00 00   movl   $0x3f,%eax
```

# Matching Disassembled Targets

**Entry**

```
0x08048730
0x08048737
0x08048740
0x08048747
0x08048750
0x08048757
```

```
8048730:b8 2b 00 00 00    movl
8048735:eb 25             jmp
8048737:b8 2a 00 00 00    movl
804873c:eb 1e             jmp
804873e:89 f6             movl
8048740:b8 2d 00 00 00    movl
8048745:eb 15             jmp
8048747:b8 2f 00 00 00    movl
804874c:eb 0e             jmp
804874e:89 f6             movl
8048750:b8 25 00 00 00    movl
8048755:eb 05             jmp
8048757:b8 3f 00 00 00    movl
```

# Relationship to C++

```
Class Animal {
  public: virtual void MakeSound();


class Cow : public Animal {
  private: MooData moodata;
  public:   virtual void MakeSound();
  }
```

- **Each class has a jumptable (vtable) associated with it**
  - one entry for each virtual function
  - Animal table contains table with pointer to Animal MakeSound function
  - Cow table contains table with pointer to Cow MakeSound function
  - Both MakeSounds are at the **same offset** in the table
- **Each instance contains a special pointer (vtable pointer) to its class's vtable**
- **Animal *animal=new Cow; animal->MakeSound();**
  - Read vtable pointer to get vtable – Cow vtable
  - Read vtable entry at function's offset – Cow::MakeSound
  - "jump" to that address (really a procedure call -> next time)
  - **(*(animal->_vptr[MAKESOUND]))(animal);**

# Summarizing

## C Control

- if-then-else
- do-while
- while
- switch

## Assembler Control

- jump
- Conditional jump

## Compiler

- Must generate assembly code to implement more complex control

## C++

- Virtual function calls

## Standard Techniques

- All loops converted to do-while form
- Large switch statements use jump tables

## Conditions in CISC

- CISC machines generally have condition code registers

## Conditions in RISC

- Use general registers to store condition information
- Special comparison instructions
- E.g., on Alpha:

```
cmple $16,1,$1
```

  – Sets register $1 to 1 when Register $16 <= 1

# IA32 Stack

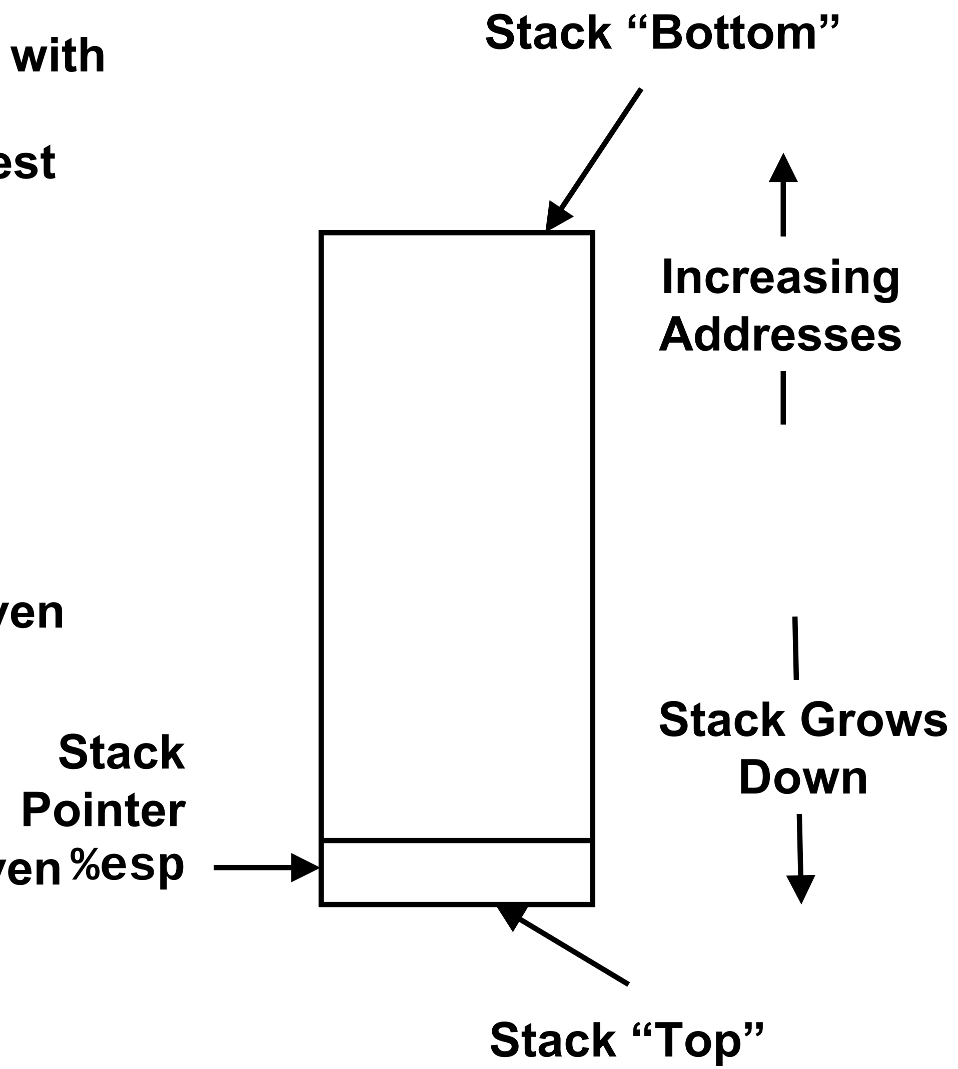
- **Region of memory managed with stack discipline**
- **Register `%esp` indicates lowest allocated position in stack**
  - i.e., address of top element

## Pushing

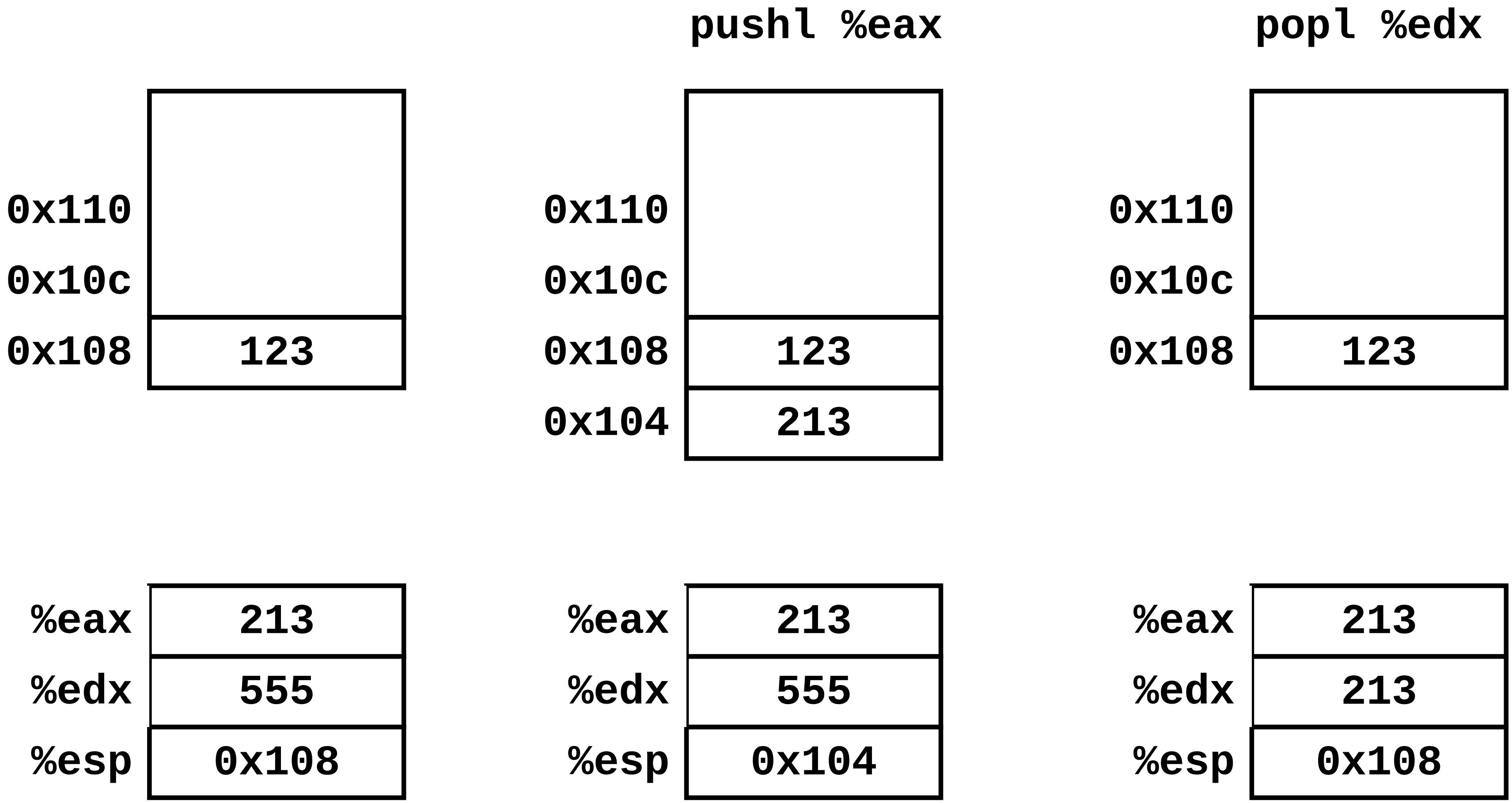
- **`pushl` *Src***
- **Fetch operand at *Src***
- **Decrement `%esp` by 4**
- **Write operand at address given by `%esp`**

## Popping

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



# Stack Operation Examples



# Procedure Control Flow

**Use stack to support procedure call and return**

**Stack stores the context of a procedure call**

**Derrida: There is no such thing as meaning without context.**

**Derrida: Each context may give a different meaning**

## 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
```

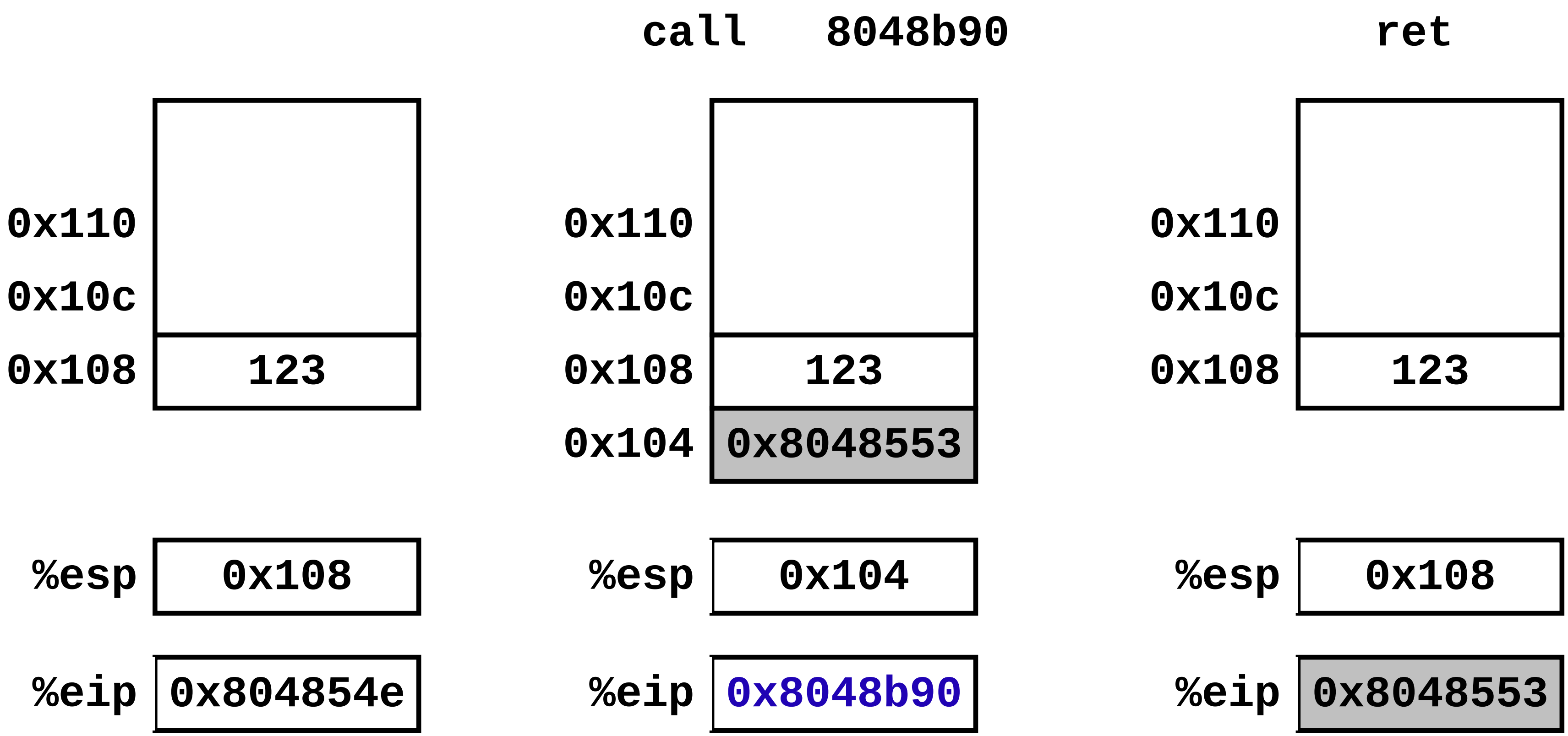
  – Return address = **`0x8048553`**

## Procedure return:

- **`ret`** Pop address from stack; Jump to address

# Procedure Call / Return Example

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




**%eip is program counter**

# Stack-Based Languages

## Languages that Support Recursion

- **e.g., C, C++, Pascal, Java, Fortran 9x, …**
- **Code must be "*Reentrant*"**
  - Multiple simultaneous instantiations of single procedure
- **Need some place to store state of each instantiation**
  - Arguments
  - Local variables
  - Return pointer

`Context`

## 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

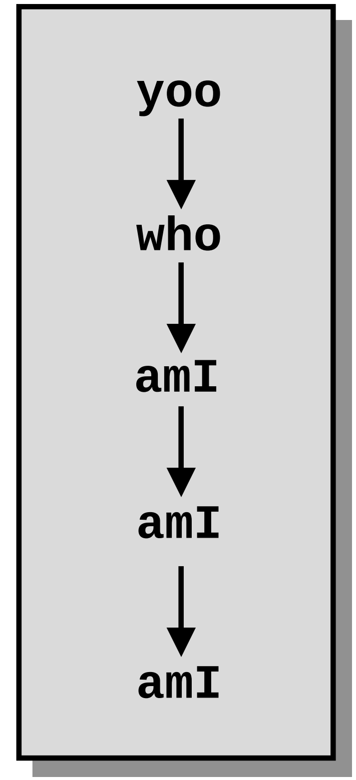
```
yoo(…)
{
   •
   •
   who();
   •
   •
}
```

```
who(…)
{
   •
   •
   amI();
   •
   •
}
```

```
amI(…)
{
   •
   •
   amI();
   •
   •
}
```

- Procedure `amI` recursive

## Call Chain



# IA32 Stack Structure

## Stack Growth

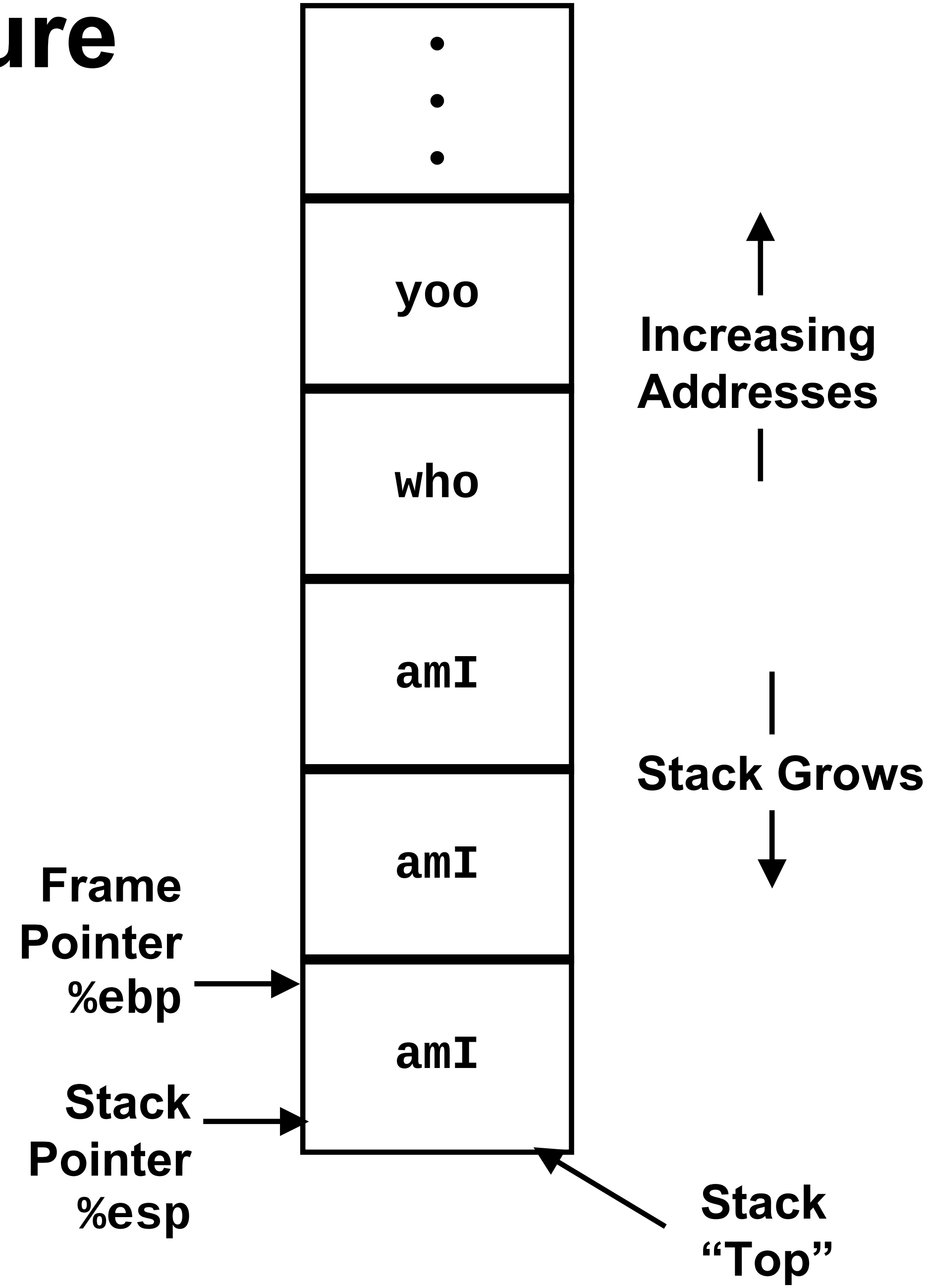
- Toward lower addresses

## Stack Pointer

- Address of next available location in stack
- Use register %esp

## Frame Pointer

- Start of current stack frame
- Use register %ebp



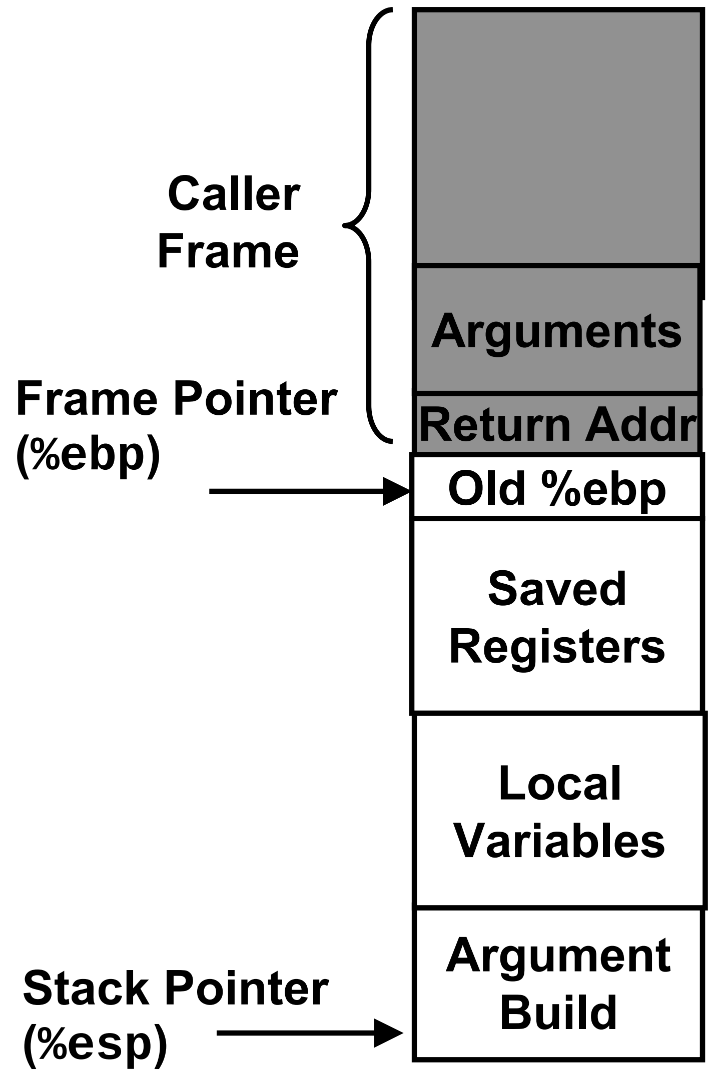
# IA32/Linux Stack Frame

## Callee Stack Frame ("Top" to Bottom)

- **Parameters for called functions**
- **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**



# Spares follow here

# "While" Loop Example #2

```
/* Compute x raised to nonnegative power p */
int ipwr_while(int x, unsigned p)
{
  int result = 1;
  while (p) {
    if (p & 0x1)
      result *= x;
    x = x*x;
    p = p>>1;
  }
  return result;
}
```

## Algorithm

- **Exploit property that** $p = p_0 + 2p_1 + 4p_2 + \ldots 2^{n-1}p_{n-1}$
- **Gives:** $x^p = z_0 \cdot z_1^{\ 2} \cdot (z_2^{\ 2})^{\ 2} \cdot \ldots \cdot (\ldots((z_{n-1}^{\ 2})^{\ 2})\ldots)^{\ 2}$

  $\underbrace{\qquad\qquad}_{n \text{ times}}$

  $z_i = 1$ **when** $p_I = 0$

  $z_i = x$ **when** $p_I = 1$
- **Complexity** $O(\log p)$

**Example**

$3^{10}$

$= 3^2 * 3^8$

$= 3^2 * ((3^2)^{\ 2})^{\ 2}$

# ipwr Computation

```
int ipwr(int x, unsigned p)
{
  int result = 1;
  while (p) {
    if (p & 0x1)
      result *= x;
    x = x*x;
    p = p>>1;
  }
  return result;
}
```

| result | x | p |
|---:|---:|---:|
| 1 | 3 | 10 |
| 1 | 9 | 5 |
| 9 | 81 | 2 |
| 9 | 6561 | 1 |
| 531441 | 43046721 | 0 |

# “While” → “Do-While ” → “Goto ”

```
  int result = 1;
  while (p) {
    if (p & 0x1)
      result *= x;
    x = x*x;
    p = p>>1;
  }
```

```
  int result = 1;
  if (!p) goto done;
  do {
    if (p & 0x1)
      result *= x;
    x = x*x;
    p = p>>1;
  } while (p);
done:
```

```
  int result = 1;
  if (!p)
    goto done;
loop:
  if (!(p & 0x1))
    goto skip;
  result *= x;
skip:
  x = x*x;
  p = p>>1;
  if (p)
    goto loop;
done:
```

- **Also converted conditional update into test and branch around update code**

# Example #2 Compilation

## Goto Version

```
  int result = 1;
  if (!p)
    goto done;
loop:
  if (!(p & 0x1))
    goto skip;
    result *= x;
skip:
    x = x*x;
    p = p>>1;
  if (p)
    goto loop;
done:
```

## Registers

```
%ecx   x
%edx   p
%eax   result
```

```
   pushl %ebp             # Setup
   movl %esp,%ebp         # Setup
   movl $1,%eax           # eax = 1
   movl 8(%ebp),%ecx      # ecx = x
   movl 12(%ebp),%edx     # edx = p
   testl %edx,%edx        # Test p
   je L36                 # If 0, goto done
L37:                    # Loop:
   testb $1,%dl           # Test p & 0x1
   je L38                 # If 0, goto skip
   imull %ecx,%eax        # result *= x
L38:                    # Skip:
   imull %ecx,%ecx        # x *= x
   shrl $1,%edx           # p >>= 1
   jne L37                # if p goto Loop
L36:                    # Done:
   movl %ebp,%esp         # Finish
   popl %ebp              # Finish
   ret                    # Finish
```