Homework 1

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

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Count</td>
<td>35</td>
</tr>
<tr>
<td>Mean</td>
<td>67.92857143</td>
</tr>
<tr>
<td>Std</td>
<td>15.14447233</td>
</tr>
<tr>
<td>Min</td>
<td>27</td>
</tr>
<tr>
<td>Max</td>
<td>75</td>
</tr>
<tr>
<td>Mode</td>
<td>75</td>
</tr>
<tr>
<td>Median</td>
<td>75</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CF</td>
<td>Carry Flag</td>
</tr>
<tr>
<td>ZF</td>
<td>Zero Flag</td>
</tr>
<tr>
<td>SF</td>
<td>Sign Flag</td>
</tr>
<tr>
<td>OF</td>
<td>Overflow Flag</td>
</tr>
</tbody>
</table>

Implicit Setting By Arithmetic Operations

\[ \text{addl } Src, Dest \]

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
  \( (a > 0 \land b > 0 \land t < 0) \lor (a < 0 \land b < 0 \land t > 0) \)

*Not Set by leal instruction*
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

<table>
<thead>
<tr>
<th>SetX</th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sete</td>
<td>ZF</td>
<td>Equal / Zero</td>
</tr>
<tr>
<td>setne</td>
<td>~ZF</td>
<td>Not Equal / Not Zero</td>
</tr>
<tr>
<td>sets</td>
<td>SF</td>
<td>Negative</td>
</tr>
<tr>
<td>setns</td>
<td>~SF</td>
<td>Nonnegative</td>
</tr>
<tr>
<td>setg</td>
<td>~ (SF^OF) &amp; ~ZF</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>setge</td>
<td>~ (SF^OF)</td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td>setl</td>
<td>(SF^OF)</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td>setle</td>
<td>(SF^OF)</td>
<td>ZF</td>
</tr>
<tr>
<td>seta</td>
<td>~CF &amp; ~ZF</td>
<td>Above (unsigned)</td>
</tr>
<tr>
<td>setb</td>
<td>CF</td>
<td>Below (unsigned)</td>
</tr>
</tbody>
</table>
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

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

Body

```assembly
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

<table>
<thead>
<tr>
<th>jX</th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jmp</td>
<td>1</td>
<td>Unconditional</td>
</tr>
<tr>
<td>je</td>
<td>ZF</td>
<td>Equal / Zero</td>
</tr>
<tr>
<td>jne</td>
<td>~ZF</td>
<td>Not Equal / Not Zero</td>
</tr>
<tr>
<td>js</td>
<td>SF</td>
<td>Negative</td>
</tr>
<tr>
<td>jns</td>
<td>~SF</td>
<td>Nonnegative</td>
</tr>
<tr>
<td>jg</td>
<td>~(SF^OF) &amp; ~ZF</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>jge</td>
<td>~(SF^OF)</td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td>jl</td>
<td>(SF^OF)</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td>jle</td>
<td>(SF^OF)</td>
<td>ZF</td>
</tr>
<tr>
<td>ja</td>
<td>~CF &amp; ~ZF</td>
<td>Above (unsigned)</td>
</tr>
<tr>
<td>jb</td>
<td>CF</td>
<td>Below (unsigned)</td>
</tr>
</tbody>
</table>
Conditional Branch Example

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

```assembly
_max:
pushl %ebp
movl %esp,%ebp

movl 8(%ebp),%edx
movl 12(%ebp),%eax

CMPl %eax,%edx
jle L9

movl %edx,%eax

L9:

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

Set Up

Body

Finish
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
L9:                  # Done:

Skipped when x \leq y
“Do-While” Loop Example

C Code

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

Goto Version

```c
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
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

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

First Goto Version

```c
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

```c
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

```c
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

General Form

\[
\text{for (Init; Test; Update )}
\text{Body}
\]

Init

result = 1

Test

p != 0

Update

p = p >> 1

Body

\{
  \text{if (p & 0x1)}
  \text{result *= x;}
  \text{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:

Init
result = 1

Test
p != 0

Update
p = p >> 1

Body
{
  if (p & 0x1)
    result *= x;
  x = x*x;
  p = p >> 1;
  if (p != 0)
    goto loop;
}

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:
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 '?';
    }
}
## Jump Table Structure

### Switch Form

```java
switch(op) {
    case 0:
        Block 0
    case 1:
        Block 1
    ...
    case n-1:
        Block n-1
}
```

### Jump Table

<table>
<thead>
<tr>
<th>jtab:</th>
<th>Targ0</th>
<th>Targ1</th>
<th>Targ2</th>
<th>Targn-1</th>
</tr>
</thead>
</table>

### Jump Targets

- **Targ0**: Code Block 0
- **Targ1**: Code Block 1
- **Targ2**: Code Block 2
- **Targn-1**: Code Block n-1

### Approx. Translation

```c
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) {
        ...
    }
}

unparse_symbol:
    pushl %ebp
    movl %esp, %ebp
    movl 8(%ebp), %eax
    cmpl $5, %eax
    ja .L49
    jmp *.L57(,%eax,4)

Setup:
    # Setup
    # Setup
    # eax = op
    # Compare op : 5
    # If > goto done
    # goto Table[op]
Assembly Setup Explanation

Symbolic Labels
• Labels of form `.Lxx` 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

<table>
<thead>
<tr>
<th>Section .rodata</th>
<th>.align 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>.L57:</td>
<td></td>
</tr>
<tr>
<td>.long .L51 #Op = 0</td>
<td></td>
</tr>
<tr>
<td>.long .L52 #Op = 1</td>
<td></td>
</tr>
<tr>
<td>.long .L53 #Op = 2</td>
<td></td>
</tr>
<tr>
<td>.long .L54 #Op = 3</td>
<td></td>
</tr>
<tr>
<td>.long .L55 #Op = 4</td>
<td></td>
</tr>
<tr>
<td>.long .L56 #Op = 5</td>
<td></td>
</tr>
</tbody>
</table>

### Enumerated Values

<table>
<thead>
<tr>
<th>ADD</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>MULT</td>
<td>1</td>
</tr>
<tr>
<td>MINUS</td>
<td>2</td>
</tr>
<tr>
<td>DIV</td>
<td>3</td>
</tr>
<tr>
<td>MOD</td>
<td>4</td>
</tr>
<tr>
<td>BAD</td>
<td>5</td>
</tr>
</tbody>
</table>

### 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

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

<table>
<thead>
<tr>
<th>Contents of section .rodata:</th>
</tr>
</thead>
<tbody>
<tr>
<td>8048bc0 30870408 37870408 40870408 47870408 0...7...@...G...</td>
</tr>
<tr>
<td>8048bd0 50870408 57870408 46616374 28256429 P...W...Fact(%d)</td>
</tr>
<tr>
<td>8048be0 203d2025 6c640a00 43686172 203d2025 = %ld..Char = %</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

• E.g., 30870408 really means 0x08048730
Disassembled Targets

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
<th>Bytes</th>
<th>Instruction</th>
<th>Address</th>
<th>Label</th>
</tr>
</thead>
<tbody>
<tr>
<td>8048730</td>
<td>b8</td>
<td>0x2b</td>
<td>movl $0x2b, %eax</td>
<td>8048730</td>
<td>b8 0x2b, %eax</td>
</tr>
<tr>
<td>8048735</td>
<td>eb</td>
<td>0x25</td>
<td>jmp 804875c &lt;unparse_symbol+0x44&gt;</td>
<td>8048735</td>
<td>eb 0x25, jmp 804875c &lt;unparse_symbol+0x44&gt;</td>
</tr>
<tr>
<td>8048737</td>
<td>b8</td>
<td>0x2a</td>
<td>movl $0x2a, %eax</td>
<td>8048737</td>
<td>b8 0x2a, %eax</td>
</tr>
<tr>
<td>804873c</td>
<td>eb</td>
<td>0x1e</td>
<td>jmp 804875c &lt;unparse_symbol+0x44&gt;</td>
<td>804873c</td>
<td>eb 0x1e, jmp 804875c &lt;unparse_symbol+0x44&gt;</td>
</tr>
<tr>
<td>804873e</td>
<td>89</td>
<td>0xf6</td>
<td>movl %esi, %esi</td>
<td>804873e</td>
<td>89 f6, movl %esi, %esi</td>
</tr>
<tr>
<td>8048740</td>
<td>b8</td>
<td>0x2d</td>
<td>movl $0x2d, %eax</td>
<td>8048740</td>
<td>b8 0x2d, movl $0x2d, %eax</td>
</tr>
<tr>
<td>8048745</td>
<td>eb</td>
<td>0x15</td>
<td>jmp 804875c &lt;unparse_symbol+0x44&gt;</td>
<td>8048745</td>
<td>eb 0x15, jmp 804875c &lt;unparse_symbol+0x44&gt;</td>
</tr>
<tr>
<td>8048747</td>
<td>b8</td>
<td>0x2f</td>
<td>movl $0x2f, %eax</td>
<td>8048747</td>
<td>b8 0x2f, movl $0x2f, %eax</td>
</tr>
<tr>
<td>804874c</td>
<td>eb</td>
<td>0xe</td>
<td>jmp 804875c &lt;unparse_symbol+0x44&gt;</td>
<td>804874c</td>
<td>eb 0xe, jmp 804875c &lt;unparse_symbol+0x44&gt;</td>
</tr>
<tr>
<td>804874e</td>
<td>89</td>
<td>0xf6</td>
<td>movl %esi, %esi</td>
<td>804874e</td>
<td>89 f6, movl %esi, %esi</td>
</tr>
<tr>
<td>8048750</td>
<td>b8</td>
<td>0x25</td>
<td>movl $0x25, %eax</td>
<td>8048750</td>
<td>b8 0x25, movl $0x25, %eax</td>
</tr>
<tr>
<td>8048755</td>
<td>eb</td>
<td>0x05</td>
<td>jmp 804875c &lt;unparse_symbol+0x44&gt;</td>
<td>8048755</td>
<td>eb 0x05, jmp 804875c &lt;unparse_symbol+0x44&gt;</td>
</tr>
<tr>
<td>8048757</td>
<td>b8</td>
<td>0x3f</td>
<td>movl $0x3f, %eax</td>
<td>8048757</td>
<td>b8 0x3f, movl $0x3f, %eax</td>
</tr>
</tbody>
</table>
Matching Disassembled Targets

Entry
0x08048730
0x08048737
0x08048740
0x08048747
0x08048750
0x08048757

<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
<th>Instruction</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>8048730</td>
<td>b8 2b</td>
<td>movl</td>
<td></td>
</tr>
<tr>
<td>8048735</td>
<td>eb 25</td>
<td>jmp</td>
<td></td>
</tr>
<tr>
<td>8048737</td>
<td>b8 2a</td>
<td>movl</td>
<td></td>
</tr>
<tr>
<td>804873c</td>
<td>eb 1e</td>
<td>jmp</td>
<td></td>
</tr>
<tr>
<td>804873e</td>
<td>89 f6</td>
<td>movl</td>
<td></td>
</tr>
<tr>
<td>8048740</td>
<td>b8 2d</td>
<td>movl</td>
<td></td>
</tr>
<tr>
<td>8048745</td>
<td>eb 15</td>
<td>jmp</td>
<td></td>
</tr>
<tr>
<td>8048747</td>
<td>b8 2f</td>
<td>movl</td>
<td></td>
</tr>
<tr>
<td>804874c</td>
<td>eb 0e</td>
<td>jmp</td>
<td></td>
</tr>
<tr>
<td>804874e</td>
<td>89 f6</td>
<td>movl</td>
<td></td>
</tr>
<tr>
<td>8048750</td>
<td>b8 25</td>
<td>movl</td>
<td></td>
</tr>
<tr>
<td>8048755</td>
<td>eb 05</td>
<td>jmp</td>
<td></td>
</tr>
<tr>
<td>8048757</td>
<td>b8 3f</td>
<td>movl</td>
<td></td>
</tr>
</tbody>
</table>
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:
  \texttt{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 `%esp`
- Increment `%esp` by 4
- Write to `Dest`
Stack Operation Examples

- **pushl %eax**
  - Before: 0x110 0x10c 0x108 0x104
  - After: 0x110 0x10c 0x108 0x104
  - %eax: 213
  - %edx: 123
  - %esp: 0x108

- **popl %edx**
  - Before: 0x110 0x10c 0x108 0x104
  - After: 0x110 0x10c 0x108 0x104
  - %eax: 213
  - %edx: 555
  - %esp: 0x108
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:
\[
\text{call } \textit{label} \quad \text{Push return address on stack; Jump to } \textit{label}
\]

Return address value
\begin{itemize}
  \item Address of instruction beyond \texttt{call}
  \item Example from disassembly
\end{itemize}

\begin{verbatim}
804854e:    e8 3d 06 00 00    call 8048b90 <main>
8048553:    50 pushl %eax
\end{verbatim}
– Return address = 0x8048553

Procedure return:
\begin{itemize}
  \item \texttt{ret} \quad \text{Pop address from stack; Jump to address}
\end{itemize}
**Procedure Call / Return Example**

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

%esp
%eip

0x108  123
0x10c
0x110

%esp  0x108
%esp  0x104
%eip  0x804854e
%eip  0x8048b90
%eip  0x8048553

%esp
%eip

0x108  123
0x10c
0x110

%esp  0x108
%esp  0x104
%eip  0x804854e
%eip  0x8048b90
%eip  0x8048553

%esp
%eip

0x108  123
0x10c
0x110

%esp  0x108
%esp  0x104
%eip  0x804854e
%eip  0x8048b90
%eip  0x8048553

%esp
%eip

%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

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

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

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

amI(...) {
  
}
```

- Procedure `amI` recursive

Call Chain

```
 yoo
  ↓
 who
  ↓
 amI
  ↓
 amI
  ↓
 amI
```
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

```c
/* 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: \( x^p = z_0 \cdot z_1^2 \cdot (z_2^2)^2 \cdot \ldots \cdot ((z_{n-1}^2)^2)^2 \)
  \( z_i = 1 \) when \( p_i = 0 \)
  \( z_i = x \) when \( p_i = 1 \)
- Complexity \( O(\log p) \)

Example

\[ 3^{10} = 3^2 \cdot 3^8 = 3^2 \cdot ((3^2)^2)^2 \]
int ipwr(int x, unsigned p)
{
    int result = 1;
    while (p) {
        if (p & 0x1)
            result *= x;
        x = x*x;
        p = p>>1;
    }
    return result;
}

<table>
<thead>
<tr>
<th>result</th>
<th>x</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>9</td>
<td>5</td>
</tr>
<tr>
<td>9</td>
<td>81</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>6561</td>
<td>1</td>
</tr>
<tr>
<td>531441</td>
<td>43046721</td>
<td>0</td>
</tr>
</tbody>
</table>
“While” → “Do-While” → “Goto”

\[
\begin{align*}
\text{int result } &= \text{1; \\
while (p) { \\
  \text{if (p & 0x1) \\
      result } &= \times; \\
  x &= x\times x; \\
  p &= p \gg 1; \\
\}}
\end{align*}
\]

\[
\begin{align*}
\text{int result } &= \text{1; \\
if (!p) goto done; \\
do { \\
  \text{if (p & 0x1) \\
      result } &= \times; \\
  x &= x\times x; \\
  p &= p \gg 1; \\
} while (p); \\
done:
\end{align*}
\]

\[
\begin{align*}
\text{int result } &= \text{1; \\
if (!p) goto done; \\
loop: \\
  \text{if (!}(p \& 0x1)) \\
      goto skip; \\
  result } &= \times; \\
  \text{skip:} \\
  x &= x\times x; \\
  p &= p \gg 1; \\
  \text{if (p) \\
      goto loop; \\
done:}
\end{align*}
\]

- Also converted conditional update into test and branch around update code
Example #2 Compilation

Goto Version

```c
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:
        x = x*x;
        p = p>>1;
        if (p)
            goto loop;
L36: # Done:
    movl %ebp,%esp # Finish
    popl %ebp # Finish
    ret # Finish
```