Computer Systems  CEN591(502)
Fall 2011

Sandeep K. S. Gupta
Arizona State University
8th lecture
Machine-Level Programming 3 (procedures)
(Slides adapted from CSAPP)
Announcements

- HW2 is uploaded:
  - Due date: Fri., Sept 30, 11:59 pm
Summary of previous class

- Optimization on the system architecture layer
  - Stored program concept
  - Harvard vs. Von Neumann Arch??

- Optimization on the instruction set layer
  - When to use registers or memory?
  - Stacks:
    - to handle procedures
    - to use registers both in caller and callee

- This class: More on instructions
  - Conditions, loop (Which one to use: for or while?)
  - More on stacks (register conventions for procedures, x86-64 optimizations on stack!)
agenda

- Operations on x86-64
- Control statement (Setting conditions; Reading conditions; Jumping)
- Loops (Do-while; While; for)
- Procedure call
  - Register conventions
  - Arguments
  - Return values
Some Arithmetic Operations

- Two Operand Instructions:

<table>
<thead>
<tr>
<th>Format</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Leal</td>
<td>$Src,Dest$</td>
</tr>
<tr>
<td>addl</td>
<td>$Src,Dest$</td>
</tr>
<tr>
<td>subl</td>
<td>$Src,Dest$</td>
</tr>
<tr>
<td>imull</td>
<td>$Src,Dest$</td>
</tr>
<tr>
<td>sall</td>
<td>$Src,Dest$</td>
</tr>
<tr>
<td>sarl</td>
<td>$Src,Dest$</td>
</tr>
<tr>
<td>shrl</td>
<td>$Src,Dest$</td>
</tr>
<tr>
<td>xorl</td>
<td>$Src,Dest$</td>
</tr>
<tr>
<td>andl</td>
<td>$Src,Dest$</td>
</tr>
<tr>
<td>orl</td>
<td>$Src,Dest$</td>
</tr>
</tbody>
</table>

- Watch out for argument order!
- No distinction between signed and unsigned int
- See book for more detail

Ex: \[\text{[eax contains } x]\]
\[x = x \times 12\]

leal (%eax,%eax,2), %eax ; x=x+2*x
sall $2, %eax ; x=x*4
# Data Representations: IA32 + x86-64

<table>
<thead>
<tr>
<th>C declaration</th>
<th>Intel data type</th>
<th>Assembly code suffix</th>
<th>X86-64 size(bytes)</th>
<th>IA32 size</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>Byte</td>
<td>b</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>short</td>
<td>Word</td>
<td>w</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>Double word</td>
<td>l</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>long int</td>
<td>Double word</td>
<td>q</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>long long int</td>
<td>- (needs generating sequence of operations)</td>
<td>q</td>
<td>8</td>
<td>-</td>
</tr>
<tr>
<td>char *</td>
<td>Double word</td>
<td>q</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>float</td>
<td>Single precision</td>
<td>s</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>Double precision</td>
<td>l</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>long double</td>
<td>Extended precision</td>
<td>t</td>
<td>10/16</td>
<td>10/12</td>
</tr>
</tbody>
</table>

CEN591 Fall 2011
Instructions

- Long word $l$ (4 Bytes) ↔ Quad word $q$ (8 Bytes)

- New instructions:
  - movl → movq
  - addl → addq
  - sall → salq
  - etc.

- 32-bit instructions that generate 32-bit results
  - Set higher order bits of destination register to 0
  - Example: addl
Control statement

Processor states
Comparing and testing
Reading conditions
Jump operations
loop
Processor State (IA32, Partial)

- Information about currently executing program
  - Temporary data (%eax, …)
  - Location of runtime stack (%ebp, %esp)
  - Location of current code control point (%eip, …)
  - Status of recent tests (CF, ZF, SF, OF)

- General purpose registers
  - %eax
  - %ecx
  - %edx
  - %ebx
  - %esi
  - %edi
  - %esp
  - %ebp

- Current stack top
  - %esp
  - %ebp

- Instruction pointer
  - %eip

- Condition codes
  - CF flag (Carry)
  - ZF flag (Zero)
  - SF flag (Sign)
  - OF flag (Overflow)
Implicit and explicit setting of Condition flags

- **Implicitly set** (think of it as **side effect**) by arithmetic operations
  
  Example: `addl/addq Src, Dest ← t = a+b`
  
  - **CF set** if carry out from most significant bit (unsigned overflow)
  - **ZF set** if `t == 0`
  - **SF set** if `t < 0` (as signed)
  - **OF set** if two’s-complement (signed) overflow
  
  Not set by `lea` instruction

- **Explicit set**: `comp, test`
  
  - `cmpl b,a` like computing `a-b` without setting destination
  - `Testl/q b,a` like computing `a&b` without setting destination
Reading / Jumping conditions

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

  Example:

<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>
</tbody>
</table>

- **jX instructions**
  - Jump to different part of code depending on condition codes

  Example:

<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>
</tbody>
</table>
### Conditional Branch Example

```c
int goto_ad(int x, int y) {
    int result;
    if (x <= y) goto Else;
    result = x - y;
    goto Exit;
Else:
    result = y - x;
Exit:
    return result;
}
```

- C allows “goto” as means of transferring control (Closer to machine-level programming style)

- Generally considered bad coding style

**absdiff:**

```assembly
pushl %ebp  
movl %esp, %ebp  
movl 8(%ebp), %edx  
movl 12(%ebp), %eax  
cmpl %eax, %edx  
jle .L6  
subl %edx, %eax  
movl %edx, %eax  
jmp .L7  
.L6:  
    subl %edx, %eax  
.L7:  
    popl %ebp  
    ret
```

### Setup

- Body1
- Body2a
- Body2b

### Finish
General “Do-While” Translation

C Code

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

- Body: 
  ```
  {
    Statement_1;
    Statement_2;
    ...
    Statement_n;
  }
  ```

- Test returns integer
  - = 0 interpreted as false
  - ≠ 0 interpreted as true

Goto Version

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

CEN591 Fall 2011
“Do-While” Loop Compilation

Goto Version

```c
int pcount_do(unsigned x) {
    int result = 0;
    loop:
        result += x & 0x1;
        x >>= 1;
        if (x)
            goto loop;
    return result;
}
```

<table>
<thead>
<tr>
<th>Registers:</th>
</tr>
</thead>
<tbody>
<tr>
<td>%edx   x</td>
</tr>
<tr>
<td>%ecx   result</td>
</tr>
</tbody>
</table>

```
movl  $0, %ecx       # result = 0
.L2:
    movl  %edx, %eax    # loop:
    andl  $1, %eax      # t = x & 1
    addl  %eax, %ecx    # result += t
    shrl  %edx          # x >>= 1
    jne   .L2           # If !0, goto loop
```
General “While” Translation

While version

\[
\text{while (Test)} \\
\text{Body}
\]

Do-While Version

\[
\text{if (!Test)} \\
\text{goto done;} \\
\text{do} \\
\text{Body} \\
\text{while (Test);} \\
\text{done:}
\]

Goto Version

\[
\text{if (!Test)} \\
\text{goto done;} \\
\text{loop:} \\
\text{Body} \\
\text{if (Test)} \\
\text{goto loop;} \\
\text{done:}
\]
"For" Loop → ... → Goto

For Version

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

Body

While Version

\[
\text{Init;}
\]

while (Test) {

Body

Update;

}

Init;

if (!Test)
goto done;

loop:

Body

Update

goto loop;

done:

Init;

if (!Test)
goto done;

done:

Body

while (Test);
Procedures

Stack frame review
Argument and return value
Register convention
IA32/Linux Stack Frame

- **Stack frame**: The portion of stack which is allocated to a procedure
- **Stack pointer (\%esp)**: moves while the procedure is executing
- **Frame pointer (\%ebp)**: procedure stack information is accessed relative to the frame pointer

**Current Stack Frame** ("Top" to Bottom)
- "Argument build:"
  - Parameters for function about to call
- 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

[Diagram showing stack frame structure with labels]

- **Stack pointer**: %esp
- **Frame pointer**: %ebp

CEN591 Fall 2011
Stack Frames

- **Contents**
  - Local variables
  - Return information
  - Temporary space

- **Management**
  - Space allocated when enter procedure
    - “Set-up” code
  - Deallocated when return
    - “Finish” code
Call Chain Example

Procedure `amI()` is recursive
Example

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

Stack

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

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

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

Stack

```
yoo
who
amI
amI
%ebp
%esp
```
Register Saving Conventions

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

- Can register be used for temporary storage?

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

who:
  . . .
  movl 8(%ebp), %edx
  addl $18243, %edx
  . . .
  ret
```
IA32/Linux+Windows 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 form of callee save
  - Restored to original values upon exit from procedure
Ex: Procedure argument (as pointers) and return values

- Use leal instruction to compute address of localx
- Retrieve localx from stack as return value

```c
int add3(int x) {
    int localx = x;
    incrk(&localx, 3);
    return localx;
}
```

Middle part of add3

- `movl $3, 4(%esp)` # 2nd arg = 3
- `leal -4(%ebp), %eax` # &localx
- `movl %eax, (%esp)` # 1st arg = &localx
- `call incrk`

Final part of add3

- `movl -4(%ebp), %eax` # Return val = localx
- `leave`
- `ret`
How about Recursion?

- **Handled Without Special Consideration**
  - Stack frames mean that each function call has private storage
    - Saved registers & local variables
    - Saved return pointer
  - Register saving conventions prevent one function call from corrupting another’s data
  - Stack discipline follows call / return pattern
    - If P calls Q, then Q returns before P
    - Last-In, First-Out

- **Also works for mutual recursion**
  - P calls Q; Q calls P
## x86-64 Integer Registers

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

- Twice the number of registers
- Accessible as 8, 16, 32, 64 bits
x86-64 Integer Registers: Usage Conventions

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
<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>

- %r8: Argument #5
- %r9: Argument #6
- %r10: Caller saved
- %r11: Caller Saved
- %r12: Callee saved
- %r13: Callee saved
- %r14: Callee saved
- %r15: Callee saved
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/%rbp`

- Other Registers
  - 6 callee saved
  - 2 caller saved
  - 1 return value (also usable as caller saved)
  - 1 special (stack pointer)
x86-64 Long Swap

void swap_l(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 (except ret)

Avoiding stack
- Can hold all local information in registers

swap:
    movq (%rdi), %rdx
    movq (%rsi), %rax
    movq %rax, (%rdi)
    movq %rdx, (%rsi)
    ret
Understanding x86-64 Stack Frame
-Example

```
movq  %rbx, -16(%rsp)  # Save %rbx
movq  %rbp, -8(%rsp)  # Save %rbp

subq  $16, %rsp  # Allocate stack frame

movq  (%rsp), %rbx  # Restore %rbx
movq  8(%rsp), %rbp  # Restore %rbp

addq  $16, %rsp  # Deallocate frame
```

```
%rsp  ——>  rtn addr
   -8  %rbp
   -16  %rbx

%rsp  ——>  rtn addr
   +8  %rbp
   %rbx
```
Interesting Features of Stack Frame

- Allocate entire frame at once
  - All stack accesses can be relative to %rsp
  - Do by decrementing stack pointer

- Simple deallocation
  - Increment stack pointer
  - No base/frame pointer needed
x86-64 Procedure Summary

- Heavy use of registers
  - Parameter passing
  - More temporaries since more registers

- Minimal use of stack
  - Sometimes none
  - Allocate/deallocate entire block

- Many tricky optimizations
  - What kind of stack frame to use
  - Various allocation techniques
Homework: buffer overflow
String Library Code

- Implementation of Unix function `gets()`

```c
/* Get string from stdin */
char *gets(char *dest)
{
    int c = getchar();
    char *p = dest;
    while (c != EOF && c != '\n') {
        *p++ = c;
        c = getchar();
    }
    *p = '\0';
    return dest;
}
```

- No way to specify **limit on number of characters** to read

- Similar problems with other library functions
  - `strcpy, strcat`: Copy strings of arbitrary length
  - `scanf, fscanf, sscanf`, when given `%s` conversion specification
Vulnerable Buffer Code

```c
/* Echo Line */
void echo()
{
    char buf[4]; /* Way too small! */
    gets(buf);
    puts(buf);
}

void call_echo()
{
    echo();
}
```

```
unix>
Type a string: 1234567
1234567

unix>
Type a string: 123456789ABC...
Segmentation Fault
```
Buffer Overflow Stack

Before call to gets

Stack Frame for main

Return Address
Saved %ebp
Saved %ebx
[3] [2] [1] [0]

Stack Frame for echo

echo:
  pushl %ebp  # Save %ebp on stack
  movl %esp, %ebp
  pushl %ebx  # Save %ebx
  subl $20, %esp  # Allocate stack space
  leal -8(%ebp),%ebx  # Compute buf as %ebp-8
  movl %ebx, (%esp)  # Push buf on stack
  call gets  # Call gets
  . . .

/* Echo Line */
void echo()
{
  char buf[4];  /* Way too small! */
  gets(buf);
  puts(buf);
}
Malicious Use of Buffer Overflow

Is it possible to alter the behavior of program? (E.g., Changing the flow)
**Malicious Use of Buffer Overflow**

- Input string contains byte representation of executable code
- Overwrite return address A with address of buffer B
- When `bar()` executes `ret`, will jump to exploit code

```c
void foo()
{
    bar();
    ...
}

int bar()
{
    char buf[64];
    gets(buf);
    ...
    return ...;
}
```

Stack after call to `gets()`

- Return address (B)
- Pad
- Exploit code

![Diagram of stack frames and buffer overflow](image-url)
Homework 2

- Exploiting buffer over flow
- Do the homework individually
- Use linux (all files are complied under –m32 option)
- Handout:
  - Instruction (read it very carefully)
  - Three binary files:
    - makecookies: to give you a code according to your studentid
    - hex2raw: to convert your hexadecimal formatted string to the ASCII values
    - bufbomb: the program you that you are asked to alter its behavior

The program asks you to enter a string, your string should be in such a way that the requested behavior in the question is performed.
Use disassembly to learn addresses and required instructions

```
objdump -d bufbom > text
```

echo:

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>80485c5</td>
<td>push %ebp</td>
<td>80485c6</td>
<td>mov %esp, %ebp</td>
</tr>
<tr>
<td>80485c6</td>
<td>89 e5</td>
<td>80485c8</td>
<td>push %ebx</td>
</tr>
<tr>
<td>80485c8</td>
<td></td>
<td>80485c9</td>
<td>sub $0x14,%esp (20)</td>
</tr>
<tr>
<td>80485c9</td>
<td></td>
<td>80485cc</td>
<td>lea -8(%ebp),%ebx</td>
</tr>
<tr>
<td>80485cc</td>
<td></td>
<td>80485cf</td>
<td>mov %ebx, (%esp)</td>
</tr>
<tr>
<td>80485cf</td>
<td></td>
<td>80485d2</td>
<td>call 8048575 &lt;gets&gt;</td>
</tr>
<tr>
<td>80485d2</td>
<td>e8 9e ff ff ff</td>
<td>80485d7</td>
<td>mov %ebx, (%esp)</td>
</tr>
<tr>
<td>80485d7</td>
<td>89 1c 24</td>
<td>80485da</td>
<td>call 80483e4 <a href="mailto:puts@plt">puts@plt</a></td>
</tr>
<tr>
<td>80485da</td>
<td>e8 05 fe ff ff</td>
<td>80485df</td>
<td>add $0x14,%esp</td>
</tr>
<tr>
<td>80485df</td>
<td>83 c4 14</td>
<td>80485e2</td>
<td>pop %ebx</td>
</tr>
<tr>
<td>80485e2</td>
<td>5b</td>
<td>80485e3</td>
<td>pop %ebp</td>
</tr>
<tr>
<td>80485e3</td>
<td>5d</td>
<td>80485e4</td>
<td>ret</td>
</tr>
<tr>
<td>80485e4</td>
<td>c3</td>
<td>-----------</td>
<td>----------------------</td>
</tr>
</tbody>
</table>
Homework 2 scores and deliveries

- **Score:**
  - Five levels, (see the instruction for each levels’ credit)
  - Full credit: 45
  - Bonus credit: 20

- **Delivery:**
  - Submit your inputs to the hex2raw as a text file for each level as follows
    - `studentid-levelx.txt`

- Share your experience on linux with other students in the class google group