Machine-Level Representation: Procedure

CSCI 2021: Machine Architecture and Organization
Pen-Chung Yew
Department Computer Science and Engineering
University of Minnesota

With Slides from Bryant, O'Hallaron and Antonia Zhai

Review: Assembly/Machine Code View

CPU (Central Processing Unit)

Addresses

Memory

PC

Registers

Data

Codes

Instructions

Memory

Programmer-Visible State
- PC: Program counter
  - Address of next instruction
  - Called $rip$ in x86-64
- Register file
  - Heavily used program data
- Condition codes
  - Store status information about most recent arithmetic or logical operation
  - Used for conditional branching

Condition Flags

CF
ZF
SF
OF

Review: Jump and Conditional Move

<table>
<thead>
<tr>
<th>X</th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Unconditional</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Equal/Zero</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Not Equal/Not Zero</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Negative</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Greater (Signed)</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Greater or Equal (Signed)</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Less or Equal (Signed)</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Less (Signed)</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Greater (Unsigned)</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Greater or Equal (Unsigned)</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Less or Equal (Unsigned)</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Less (Unsigned)</td>
<td></td>
</tr>
</tbody>
</table>

Conditional Move

```
val = True? Then_Expr : Else_Expr;
val = x>y ? x-y : y-x;
```

Review: Loops

"While" Loop

```
while (Test)

Body
```

"Do-While" Loop

```
do

Body

while (Test);
```

"For" Loop

```
for (Init; Test; Update)

Body
```

Review: Jump Table Structure

```
switch(x) {
  case val_0:
    Block 0
  case val_1:
    Block 1
    ...
  case val_n-1:
    Block n-1
}
```

Jump Table

```
Jump Targets
Targ0:  Code Block 0
Targ1:  Code Block 1
Targ2:  Code Block 2

Targn-1: Code Block n-1
```

Outline

- Procedures
  - Stack Structure
    - Calling Conventions
    - Passing control
    - Passing data
    - Managing local data
  - Illustration of Recursion

Review: Jump Table Structure
Mechanisms in Procedures

- **Passing control**
  - To beginning of procedure code
  - Back to return point

- **Passing data**
  - Procedure arguments
  - Return value

- **Memory management**
  - Allocate during procedure execution
  - Deallocate upon return

- Mechanisms all implemented with machine instructions

- x86-64 implementation of a procedure uses only those mechanisms required

```c
P(…) {
  y = Q(x);
  print(y);
}
```

```c
int Q(int i) {
  int t = 3*i;
  int v[10];
  return v[t];
}
```

Review: Assembly/Machine Code View

- **CPU (Central Processing Unit)**
- **Programmer-Visible State**
  - PC: Program counter
  - Address of next instruction
  - Called %rip in x86-64
  - Register file
  - Condition codes
    - Store status information about most recent arithmetic or logical operation
    - Used for conditional branching

- **Stack**
  - Region of memory managed with stack discipline, i.e., last-in-first-out, or LIFO.
  - Grows toward lower addresses
  - Register %rsp contains lowest stack address
  - Address of "top" element

- **x86-64 Stack**
  - Stack "Top"
  - Stack "Bottom"

- **x86-64 Stack: Push**
  - pushq Src
    - Fetch operand at Src
    - Decrement %rsp by 8
    - Write operand at address given by %rsp

- **x86-64 Stack: Pop**
  - popq Dest
    - Read value at address given by %rsp
    - Increment %rsp by 8
    - Store value at Dest (must be register)

Outline

- **Procedures**
  - Stack Structure
  - Calling Conventions
    - Passing control
    - Passing data
    - Managing local data
    - Illustration of Recursion
Code Examples

```c
long mult2(long a, long b)
{
    long s = a * b;
    return s;
}

void multstore(long x, long y, long *dest)
{
    long t = mult2(x, y);
    *dest = t;
}
```

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 the next instruction right after call
    - Example from disassembly
  - Procedure return: ret
    - Pop address from stack
    - Jump to address

Control Flow Example #1

```assembly
0000000000400540 <multstore>:
    push %rbx  # Save %rbx
    mov %rdx,%rbx  # Save dest
    callq 400550 <mult2>  # mult2(x,y)
    mov %rax,(%rbx)  # Save at dest
    pop %rbx  # Restore %rbx
    retq  # Return
```

Control Flow Example #2

```assembly
0000000000400540 <multstore>:
    push %rbx  # Save %rbx
    mov %rdx,%rbx  # Save dest
    callq 400550 <mult2>  # mult2(x,y)
    mov %rax,(%rbx)  # Save at dest
    pop %rbx  # Restore %rbx
    retq  # Return
```

Control Flow Example #3

```assembly
0000000000400540 <multstore>:
    push %rbx  # Save %rbx
    mov %rdx,%rbx  # Save dest
    callq 400550 <mult2>  # mult2(x,y)
    mov %rax,(%rbx)  # Save at dest
    pop %rbx  # Restore %rbx
    retq  # Return
```

Control Flow Example #4

```assembly
0000000000400540 <multstore>:
    push %rbx  # Save %rbx
    mov %rdx,%rbx  # Save dest
    callq 400550 <mult2>  # mult2(x,y)
    mov %rax,(%rbx)  # Save at dest
    pop %rbx  # Restore %rbx
    retq  # Return
```
Outline

- Procedures
  - Stack Structure
  - Calling Conventions
    - Passing control
    - Passing data
    - Managing local data
  - Illustrations of Recursion & Pointers

Procedure Data Flow

Registers | Stack
--- | ---
rsi | Arg
rbx | Arg
rcx | Arg
rdx | Arg
r8  | Arg
r9  | Arg
rax | Return
rcx | Only allocate stack space when needed
rdi | x
rsi | y
rdx | z
rdi | dest
rcx | t
rdx | s
rcx | r
rsi | q
rdi | p

Code Examples

```c
long mult2(long a, long b)
{
    long s = a * b;
    return s;
}
```

```c
void multstore(long x, long y, long *dest) {
    long t = mult2(x, y);
    *dest = t;
}
```

Data Flow Examples

```c
long mult2(long a, long b) {
    long t = mult2(x, y); // t = a * b
    return t;
}
```

```c
void multstore(long x, long y, long *dest) {
    long t = mult2(x, y);
    *dest = t;
}
```
Stack-Based Languages

- Languages that support recursion
  - e.g., C, Pascal, Java
  - 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
  - Caller returns before caller does
- Stack allocated in Frames
  - State for single procedure instantiation

Call Chain Example

Example

Stack Frames

- Contents
  - Return information
  - Local storage (if needed)
  - Temporary space (if needed)
- Management
  - Space allocated when enter procedure
    - “Set-up” code
      - Includes push by call instruction
    - Deallocated when return
      - “Finish” code
      - Includes pop by ret instruction

Example
Outline

- Procedures
  - Stack Structure (Last-in-First-out, LIFO)
- Calling Conventions
  - Passing control
  - Passing data
  - Managing local data
- Illustration of Recursion

Review: Procedure Passing Control

- Use stack to support procedure call and return (part of Application Binary Interface, ABI)
- Procedure call: call label
  - Push return address on stack
  - Jump to label
- Return address:
  - Address of the next instruction right after call
- Procedure return: ret
  - Pop return address from stack
  - Jump to the return address

Review: Procedure Data Flow

- First 6 arguments
  - %edi
  - %esi
  - %ecx
  - %edx
  - %ebx
  - %ebp
- Return value
  - %eax
- Only allocate stack space when needed

Review: Mechanisms in Procedures

- Passing control
  - To beginning of procedure code
  - Back to return point
- Passing data
  - Procedure arguments
  - Return value
- Memory management
  - Allocate during procedure execution
  - Deallocate upon return
Review: Structure of Stack

- 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
  - Caller returns before callee does (LIFO)
- Stack allocated in Frames
  - State for single procedure instantiation

Example: incr

```c
long incr(long *p, long val) {
    long x = *p;
    long y = x + val;
    *p = y;
    return x;
}
```

Example: Calling incr #1

```c
long call_incr() {
    long v1 = 15213;
    long v2 = incr(avl, 3000);
    return v1+v2;
}
```

Example: Calling incr #2

```c
long call_incr() {
    long v1 = 15213;
    long v2 = incr(avl, 3000);
    return v1+v2;
}
```

Example: Calling incr #3

```c
long call_incr() {
    long v1 = 15213;
    long v2 = incr(avl, 3000);
    return v1+v2;
}
```
Example: Calling incr #4

```c
long call_incr() {
  long v1 = 15213;
  long v2 = incr(v1, 3000);
  return v1 + v2;
}
```

```
call_incr:
  subq $16, trap
  movq $15213, rdx
  movl $3000, rsi
  leaq rdx(rdi), trax
  call incr
  addq $8(trap), trax
  addq $16, trax
  ret
```

```
Example: Calling incr #5

```c
long call_incr() {
  long v1 = 15213;
  long v2 = incr(v1, 3000);
  return v1 + v2;
}
```

```
call_incr:
  subq $16, trap
  movq $15213, rdx
  movl $3000, rsi
  leaq rdx(rdi), trax
  call incr
  addq $8(trap), trax
  addq $16, trax
  ret
```

Register Saving Conventions

- When procedure `yoo` calls `who`:
  - `yoo` is the caller
  - `who` is the callee
- Can register be used for temporary storage?
  - `yoo`:
    ```
    ... 
    subq $16, rdx
    call who 
    ... 
    ret
    ```
  - Contents of register `rdx` overwritten by `who`
  - This could be trouble: something should be done!
    - Need some coordination

x86-64 Linux Register Usage (Caller Saved)

```
<table>
<thead>
<tr>
<th>Register</th>
<th>Usage</th>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>rax</td>
<td>Return value</td>
<td></td>
</tr>
<tr>
<td>rdx, rsi, rdi</td>
<td>caller-saved</td>
<td></td>
</tr>
<tr>
<td>rdi</td>
<td>can be changed by procedure</td>
<td></td>
</tr>
<tr>
<td>rsi</td>
<td>caller-saved</td>
<td></td>
</tr>
<tr>
<td>rdx</td>
<td>caller-saved</td>
<td></td>
</tr>
<tr>
<td>trb</td>
<td>caller-saved</td>
<td></td>
</tr>
<tr>
<td>trc</td>
<td>caller-saved</td>
<td></td>
</tr>
<tr>
<td>trd</td>
<td>caller-saved</td>
<td></td>
</tr>
<tr>
<td>trf</td>
<td>caller-saved</td>
<td></td>
</tr>
<tr>
<td>trg</td>
<td>caller-saved</td>
<td></td>
</tr>
<tr>
<td>trh</td>
<td>caller-saved</td>
<td></td>
</tr>
</tbody>
</table>
```

x86-64 Linux Register Usage (Callee Saved)

```
<table>
<thead>
<tr>
<th>Register</th>
<th>Usage</th>
<th>Special</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rbx, %r12, %r13, %r14</td>
<td>callee-saved</td>
<td>caller-saved</td>
</tr>
<tr>
<td>%rbp</td>
<td>callee-saved</td>
<td>caller save</td>
</tr>
<tr>
<td>%rsi</td>
<td>can mix &amp; match</td>
<td>caller-saved</td>
</tr>
<tr>
<td>%rdi</td>
<td>special form of callee save</td>
<td></td>
</tr>
<tr>
<td>%rdx</td>
<td>restored to original value upon exit from procedure</td>
<td></td>
</tr>
</tbody>
</table>
x86-64 Integer Registers: Usage Conventions

<table>
<thead>
<tr>
<th>Register</th>
<th>Usage</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 #8</td>
</tr>
<tr>
<td>%rdx</td>
<td>Argument #6</td>
</tr>
<tr>
<td>%rsi</td>
<td>Callee saved</td>
</tr>
<tr>
<td>%rdi</td>
<td>Argument #2</td>
</tr>
<tr>
<td>% rsp</td>
<td>Callee saved</td>
</tr>
<tr>
<td>%rbp</td>
<td>Stack pointer</td>
</tr>
<tr>
<td>% r8</td>
<td>Callee saved</td>
</tr>
<tr>
<td>% r9</td>
<td>Callee saved</td>
</tr>
<tr>
<td>% r10</td>
<td>Callee saved</td>
</tr>
<tr>
<td>% r11</td>
<td>Callee saved</td>
</tr>
<tr>
<td>% r12</td>
<td>Callee saved</td>
</tr>
<tr>
<td>% r13</td>
<td>Callee saved</td>
</tr>
<tr>
<td>% r14</td>
<td>Callee saved</td>
</tr>
<tr>
<td>% r15</td>
<td>Callee saved</td>
</tr>
</tbody>
</table>

Callee-Saved Example #1

```c
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x + v2;
}
```

Callee-Saved Example #2

```c
call_incr2:
pushq %rbx
subq $16, %rsp
movq %rdi, %rbx
movq $15213, 8(%rsp)
call incr
movl $3000, %esi
leaq 8(%rsp), %rdi
call incr
addq %rbx, %rax
addq $16, %rsp
popq %rbx
ret
```

Recursive Function

```c
/* Recursive popcount */
long popcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1) + popcount_r(x >> 1);
}
```

Recursive Function Terminal Case

```c
/* Recursive popcount */
long popcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1) + popcount_r(x >> 1);
}
```

Outline

- Procedures
  - Stack Structure
  - Calling Conventions
    - Passing control
    - Passing data
    - Managing local data
  - Illustration of recursion

Register Use(s)

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>x</td>
<td>Argument</td>
</tr>
<tr>
<td>%rax</td>
<td></td>
<td>Return value</td>
</tr>
</tbody>
</table>

Carnegie Mellon
Review: Jump

<table>
<thead>
<tr>
<th>JK</th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jp</td>
<td>1</td>
<td>Unconditional</td>
</tr>
<tr>
<td>Jz</td>
<td>0</td>
<td>Equal / Zero</td>
</tr>
<tr>
<td>Jx</td>
<td>xF</td>
<td>Not Equal / Not Zero</td>
</tr>
<tr>
<td>Jn</td>
<td>-xF</td>
<td>Negative</td>
</tr>
<tr>
<td>Ja</td>
<td>-xF</td>
<td>Nonnegative</td>
</tr>
<tr>
<td>J5</td>
<td>xS</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>J6</td>
<td>-xS</td>
<td>Less or Equal (Signed)</td>
</tr>
<tr>
<td>J7</td>
<td>xSF</td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td>J8</td>
<td>-xSF</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td>J9</td>
<td>xGF</td>
<td>Greater (Unsigned)</td>
</tr>
<tr>
<td>J10</td>
<td>-xGF</td>
<td>Less or Equal (Unsigned)</td>
</tr>
<tr>
<td>J11</td>
<td>xAF</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>J12</td>
<td>-xAF</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td>J13</td>
<td>xUF</td>
<td>Greater (Unsigned)</td>
</tr>
<tr>
<td>J14</td>
<td>-xUF</td>
<td>Less (Unsigned)</td>
</tr>
</tbody>
</table>

Recursive Function Register Save

```c
/* Recursive popcount */
long popcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1) + popcount_r(x >> 1);
}
```

Register | Use(s) | Type
---------|--------|------
rdi      | x      | Argument
```

Recursive Function Call Setup

```c
/* Recursive popcount */
long popcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1) + popcount_r(x >> 1);
}
```

Register | Use(s) | Type
---------|--------|------
rdi      | x      | Argument
```

Review: Recursive Function Call

```c
/* Recursive popcount */
long popcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1) + popcount_r(x >> 1);
}
```

Register | Use(s) | Type
---------|--------|------
rdi      | x      | Argument
```

Recursive Function Call Completion

```c
/* Recursive popcount */
long popcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1) + popcount_r(x >> 1);
}
```

Register | Use(s) | Type
---------|--------|------
rdi      | x      | Argument
```

Recursive Function Result

```c
/* Recursive popcount */
long popcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1) + popcount_r(x >> 1);
}
```

Register | Use(s) | Type
---------|--------|------
rdi      | x      | Argument
```

Recursive Function Completion

```c
/* Recursive popcount */
long popcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1) + popcount_r(x >> 1);
}
```

Register | Use(s) | Type
---------|--------|------
rdi      | x      | Argument
```
Observations 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
    - Unless the C code explicitly does so (e.g., buffer overflow)
  - 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 Procedure Summary

- Stack is the right data structure for procedure call / return
  - If P calls Q, then Q returns before P
- Recursion (& mutual recursion) handled by normal calling conventions
  - Can safely store values in local stack frame and in callee-saved registers
  - Put function arguments at top of stack
  - Result return in %rax
- Pointers are addresses of values
  - On stack or global