Sandeep K. S. Gupta
Arizona State University

7th lecture

Machine-Level Programming II: Basics
(Slides adapted from CSAPP)
Announcements

- Assignment # 2 will be given in the next class
- Midterm exam
  - First exam
  - Second exam
Summary of previous class

- Intel processor speed growth exceeds the Moor law in the two past decades

- RISC vs. CISC tradeoffs
  - **Performance tradeoff**: Processor eq.: \[ \text{CPU time} = \left( \sum_{i=1}^{n} \text{IC}_i \times \text{CPI}_i \right) \times \text{Clock cycle time} \]
    - CISC reduces number of instructions compared to CISC
    - RISC has lower cycle per instruction and clock cycle time due to shorter critical path
  - **Cost-performance tradeoff**

- Single-core vs. multi-core tradeoffs
  - A single core sys. needs to run at higher freq. to achieve the same speed of a multi-core sys.
    - Remember cube relationship of freq. and power consumption
  - Overhead of multi-core sys.??

- No memory-to memory transfer?
  - How often this functionality would be used?
  - How much complexity it would add?

- **This class**: Stored program concept, how the logic of programs is interpreted in the machine?
  - Which one is better: `for` or `while` loop?
Agenda

- Stored program concept
- More on machine level instructions and operations:
  - Addressing, Operations (add, …)
Instructions are bits

Programs are stored in memory — to be read or written just like data

Fetch & Execute Cycle

- Instructions are fetched and put into a special register
- Bits in the register "control" the subsequent actions
- Fetch the “next” instruction and continue

memory for data, programs, compilers, editors, etc. in.

Instruction representation in the memory
Harvard vs. Von Numann Arch.

- Faster: Simultaneous reading of instruction and reading/writing data to memory
- Safer: Protection of codes while executing
- Application: Embedded systems

- Higher flexibility
  - Data and code memory space can be interchangeably used
  - Higher memory efficiency
- Application: Computer system?

Slides adapted from the book: an introduction to digital signal processor, Bruno Paillard, 2007
Simple Memory Addressing Modes

- **Normal**  
  \( (R) \) \( \text{Mem}[\text{Reg}[R]] \)  
  - Register R specifies memory address

  \texttt{movl} (\%ecx),\%eax

- **Displacement**  
  \( D(R) \) \( \text{Mem}[\text{Reg}[R]+D] \)  
  - Register R specifies start of memory region
  - Constant displacement D specifies offset

  \texttt{movl} 8(\%ebp),\%edx
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 8(%ebp), %edx
movl 12(%ebp), %ecx
movl (%edx), %ebx
movl (%ecx), %eax
movl %eax, (%edx)
movl %ebx, (%ecx)
popl %ebx
popl %ebp
ret
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

```
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
IA32/Linux Stack Frame

- **Stack frame:** The portion of the 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.

```
Frame pointer
%ebp

Stack pointer
%esp
```

```
...  
Arguments

Return Addr  
Old %ebp

Saved  
Registers
+
Local
Variables

Argument
Build
```
IA32 Stack

- Region of memory managed with stack discipline
- Grows toward lower addresses

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

Stack Pointer: `%esp`
IA32 Stack: Push

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

Stack Pointer: %esp

Stack “Bottom”

Stack “Top”

Increasing Addresses

Stack Grows Down
IA32 Stack: Pop

Stack Pointer: %esp

Stack "Bottom"

Increasing Addresses

Stack Grows Down

Stack "Top"
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 8(%ebp), %edx
movl 12(%ebp), %ecx
movl (%edx), %ebx
movl (%ecx), %eax
movl %eax, (%edx)
movl %ebx, (%ecx)

popl %ebx
popl %ebp
ret
void swap(int *xp, int *yp) {
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}

movl 8(%ebp), %edx  # edx = xp
movl 12(%ebp), %ecx  # ecx = yp
movl (%edx), %ebx  # ebx = *xp (t0)
movl (%ecx), %eax  # eax = *yp (t1)
movl %eax, (%edx)  # *xp = t1
movl %ebx, (%ecx)  # *yp = t0
Extra slides
Understanding Swap (3)

<table>
<thead>
<tr>
<th>EEG</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>0x124</td>
</tr>
<tr>
<td>%edx</td>
<td>0x120</td>
</tr>
<tr>
<td>%ecx</td>
<td>0x11c</td>
</tr>
<tr>
<td>%ebx</td>
<td>0x118</td>
</tr>
<tr>
<td>%esi</td>
<td>0x114</td>
</tr>
<tr>
<td>%edi</td>
<td>0x10c</td>
</tr>
<tr>
<td>%esp</td>
<td>0x108</td>
</tr>
<tr>
<td>%ebp</td>
<td>0x104</td>
</tr>
<tr>
<td></td>
<td>0x100</td>
</tr>
</tbody>
</table>

```
movl 8(%ebp), %edx  # edx = xp
movl 12(%ebp), %ecx  # ecx = yp
movl (%edx), %ebx  # ebx = *xp (t0)
movl (%ecx), %eax  # eax = *yp (t1)
movl %eax, (%edx)  # *xp = t1
movl %ebx, (%ecx)  # *yp = t0
```
Understanding Swap(3)

| %eax | %edx 0x124 |
| %ecx | %ebx |
| %esi | %edi |
| %esp | %ebp 0x104 |

```
movl 8(%ebp), %edx  # edx = xp
movl 12(%ebp), %ecx  # ecx = yp
movl (%edx), %ebx  # ebx = *xp (t0)
movl (%ecx), %eax  # eax = *yp (t1)
movl %eax, (%edx)  # *xp = t1
movl %ebx, (%ecx)  # *yp = t0
```
Understanding Swap(3)

### Variables and Addresses

<table>
<thead>
<tr>
<th>Offset</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>0x120</td>
</tr>
<tr>
<td>8</td>
<td>0x124</td>
</tr>
<tr>
<td>4</td>
<td>0x10c</td>
</tr>
<tr>
<td>0</td>
<td>0x104</td>
</tr>
<tr>
<td>-4</td>
<td>0x100</td>
</tr>
</tbody>
</table>

### Registers

- `%eax` → 0x124
- `%edx` → 0x124
- `%ecx` → 0x118
- `%ebx` → 0x104
- `%esi` → 0x100
- `%edi` → 0x10c
- `%esp` → 0x114
- `%ebp` → 0x110

### Code Snippet

```assembly
movl 8(%ebp), %edx  # edx = xp
movl 12(%ebp), %ecx # ecx = yp
movl (%edx), %ebx   # ebx = *xp (t0)
movl (%ecx), %eax   # eax = *yp (t1)
movl %eax, (%edx)   # *xp = t1
movl %ebx, (%ecx)   # *yp = t0
```
Understanding Swap(3)

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td></td>
</tr>
<tr>
<td>%edx</td>
<td>0x124</td>
</tr>
<tr>
<td>%ecx</td>
<td>0x120</td>
</tr>
<tr>
<td>%ebx</td>
<td>123</td>
</tr>
<tr>
<td>%esi</td>
<td></td>
</tr>
<tr>
<td>%edi</td>
<td></td>
</tr>
<tr>
<td>%esp</td>
<td></td>
</tr>
<tr>
<td>%ebp</td>
<td>0x104</td>
</tr>
</tbody>
</table>

movl 8(%ebp), %edx  # edx = xp
movl 12(%ebp), %ecx  # ecx = yp
movl (%edx), %ebx  # ebx = *xp (t0)
movl (%ecx), %eax  # eax = *yp (t1)
movl %eax, (%edx)  # *xp = t1
movl %ebx, (%ecx)  # *yp = t0
Understanding Swap(3)

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>456</td>
</tr>
<tr>
<td>%edx</td>
<td>0x124</td>
</tr>
<tr>
<td>%ecx</td>
<td>0x120</td>
</tr>
<tr>
<td>%ebx</td>
<td>123</td>
</tr>
<tr>
<td>%esi</td>
<td></td>
</tr>
<tr>
<td>%edi</td>
<td></td>
</tr>
<tr>
<td>%esp</td>
<td></td>
</tr>
<tr>
<td>%ebp</td>
<td>0x104</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Offset</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>yp 12</td>
<td>0x120</td>
</tr>
<tr>
<td>xp 8</td>
<td>0x124</td>
</tr>
<tr>
<td>4</td>
<td>0x110</td>
</tr>
<tr>
<td>0</td>
<td>0x10c</td>
</tr>
<tr>
<td>-4</td>
<td>0x108</td>
</tr>
<tr>
<td></td>
<td>0x104</td>
</tr>
<tr>
<td></td>
<td>0x100</td>
</tr>
</tbody>
</table>

```
movl 8(%ebp), %edx  # edx = xp
movl 12(%ebp), %ecx  # ecx = yp
movl (%edx), %ebx   # ebx = *xp (t0)
movl (%ecx), %eax   # eax = *yp (t1)
movl %eax, (%edx)   # *xp = t1
movl %ebx, (%ecx)   # *yp = t0
```
Understanding Swap(3)

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>456</td>
</tr>
<tr>
<td>%edx</td>
<td>0x124</td>
</tr>
<tr>
<td>%ecx</td>
<td>0x120</td>
</tr>
<tr>
<td>%ebx</td>
<td>123</td>
</tr>
<tr>
<td>%esi</td>
<td></td>
</tr>
<tr>
<td>%edi</td>
<td></td>
</tr>
<tr>
<td>%esp</td>
<td></td>
</tr>
<tr>
<td>%ebp</td>
<td>0x104</td>
</tr>
</tbody>
</table>

Offset:
- yp: 12
- xp: 8
- %ebp: 0

Address:
- 0x124
- 0x120
- 0x11c
- 0x118
- 0x114
- 0x10c
- 0x108
- 0x104
- 0x100

Assembly Code:
- `movl 8(%ebp), %edx`  # edx = xp
- `movl 12(%ebp), %ecx`  # ecx = yp
- `movl (%edx), %ebx`  # ebx = *xp (t0)
- `movl (%ecx), %eax`  # eax = *yp (t1)
- `movl %eax, (%edx)`  # *xp = t1
- `movl %ebx, (%ecx)`  # *yp = t0
### Understanding Swap(3)

<table>
<thead>
<tr>
<th>Register</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%eax</code></td>
<td>456</td>
</tr>
<tr>
<td><code>%edx</code></td>
<td>0x124</td>
</tr>
<tr>
<td><code>%ecx</code></td>
<td>0x120</td>
</tr>
<tr>
<td><code>%ebx</code></td>
<td>123</td>
</tr>
<tr>
<td><code>%esi</code></td>
<td></td>
</tr>
<tr>
<td><code>%edi</code></td>
<td></td>
</tr>
<tr>
<td><code>%esp</code></td>
<td></td>
</tr>
<tr>
<td><code>%ebp</code></td>
<td>0x104</td>
</tr>
</tbody>
</table>

#### Offset

<table>
<thead>
<tr>
<th>Offset</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>0x120</td>
</tr>
<tr>
<td>8</td>
<td>0x124</td>
</tr>
<tr>
<td>4</td>
<td>Rtn adr</td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>-4</td>
<td></td>
</tr>
</tbody>
</table>

#### Assembly Code

```
movl 8(%ebp), %edx  # edx = xp
movl 12(%ebp), %ecx  # ecx = yp
movl (%edx), %ebx   # ebx = *xp (t0)
movl (%ecx), %eax   # eax = *xp (t1)
movl %eax, (%edx)   # *xp = t1
movl %ebx, (%ecx)   # *yp = t0
```
Complete Memory Addressing Modes

- **Most General Form**
  \[ D(Rb,Ri,S) \rightarrow Mem[Reg[Rb]+S*Reg[Ri]+D] \]
  - **D:** Constant “displacement” 1, 2, or 4 bytes
  - **Rb:** Base register: Any of 8 integer registers
  - **Ri:** Index register: Any, except for \%esp
    - Unlikely you’d use \%ebp, either
  - **S:** Scale: 1, 2, 4, or 8

- **Special Cases**
  - \((Rb,Ri)\) \rightarrow Mem[Reg[Rb]+Reg[Ri]]
  - \(D(Rb,Ri)\) \rightarrow Mem[Reg[Rb]+Reg[Ri]+D]
  - \((Rb,Ri,S)\) \rightarrow Mem[Reg[Rb]+S*Reg[Ri]]