CS24: INTRODUCTION TO COMPUTING SYSTEMS

Spring 2015
Lecture 5
Began our tour of the IA32 instruction set architecture

IA32 provides 8 general-purpose registers
- `eax`, `ebx`, `ecx`, `edx` are used for general operations
- `esp` is the stack pointer, `ebp` is the frame pointer (a.k.a. “base pointer”)
- `esi`, `edi` used for string operations

Two additional registers:
- `eip` is the instruction pointer
- `eflags` contains status flags
IA32 Instructions

Instructions follow this pattern:
- opcode operand, operand, ...

Examples:
- add $5, %ax
- mov %ecx, %edx
- push %ebp

Important note!
- Above assembly-code syntax is called AT&T syntax
- GNU assembler uses this syntax by default
- Intel IA32 manuals, other assemblers use Intel syntax

Some big differences between the two formats!
- mov %ecx, %edx  # AT&T: Copies ecx to edx
- mov edx, ecx    # Intel: Copies ecx to edx
Some general categories of instructions:
- Data movement instructions
- Arithmetic and logical instructions
- Flow-control instructions
- (many others too, e.g. floating point, SIMD, etc.)

Data movement:
- \texttt{mov} Move data value from source to destination
- \texttt{movs} Move value with sign-extension
- \texttt{movz} Move value with zero-extension
- \texttt{push} Push value onto the stack
- \texttt{pop} Pop value off the stack
IA32 Data Movement Instructions

Data movement instructions can specify a suffix to indicate size of operand(s)
- b = byte, w = word, l = doubleword, q = quadword

Some instructions work with one data size:
- `movl %ecx, %edx`
  - Moves doubleword (4 byte) register `ecx` into `edx`
- `pushb %al`
  - Pushes register `al` (1 byte) onto stack

Move with sign/zero extension takes two sizes:
- `movsbl %al, %edx`
  - Moves byte `al` into doubleword (4 bytes) register `edx`, extending sign-bit of value into remaining bytes
- `movzqw %cx, %rax`
  - Moves word (2 bytes) `cx` into quadword (8 bytes) register `rax`, zeroing out higher-order bytes in destination
IA32 OPERAND TYPES

- Many different operand types and combinations supported by IA32 instruction set
- Immediate values – numeric constants:
  - Must specify \$ prefix to use a numeric constant
  - \$5, \$-37, \$0xF005B411
- Registers:
  - Specify \% prefix on register name
  - \%ebp, \%eax, \%rcx
- Example:
  - `movl $42, \%ebx`
    - Moves the value $42_{10}$ into ebx register
IA32 Memory-Reference Operands

- IA32 has very rich support for memory references
  - Denote memory access as M[Address]
- Absolute memory access
  - Immediate value with no $ prefix
  - `movl 0xE700, %edx`
    - Retrieves memory value M[0xE700] into edx
- Indirect memory access
  - Register name, enclosed with parens: `(Reg)`
  - `movw %cx, (%ebx)`
    - Stores word (2 bytes) in %cx into memory location M[%ebx]
- Base + displacement memory access
  - `Imm(Reg)` accesses M[Imm + Reg]
  - `movl -8(%ebp), %eax`
    - Retrieves dword M[-8 + %ebp] into %eax
    - (Presumably, %ebp − 8 > 0)
Indexed memory access

- \((\text{Reg}B, \text{Reg}I)\) accesses \(M[\text{Reg}B + \text{Reg}I]\)
  - \(\text{Reg}B\) is the base (i.e. starting) address of a memory array
  - \(\text{Reg}I\) is an index into the memory array
- \(\text{Imm}(\text{Reg}B, \text{Reg}I)\) accesses \(M[\text{Imm} + \text{Reg}B + \text{Reg}I]\)
- Assumes that array elements are bytes

Examples:

- \(\%eax = 150, \%ebx = 400\)
- \(\text{movl} (\%eax, \%ebx), \%edx\)
  - Retrieves value at \(M[150 + 400] = M[550]\) into \(\%edx\)
- \(\text{movl} \%ecx, -200(\%ebx, \%eax)\)
  - Stores \(\%ecx\) into location \(M[-200 + 400 + 150] = M[350]\)
IA32 MEMORY-REFERENCE OPERANDS (3)

Scaled indexed memory access
- With scale factor \( s = 1, 2, 4, 8 \):
  - \((, \text{ Reg}, s)\) \(\text{M}[\text{Reg} \times s]\)
  - \(\text{Imm}(, \text{ Reg}, s)\) \(\text{M}[\text{Imm} + \text{Reg} \times s]\)
  - \((\text{RegB}, \text{ RegI}, s)\) \(\text{M}[\text{RegB} + \text{RegI} \times s]\)
  - \(\text{Imm}(\text{RegB}, \text{ RegI}, s)\) \(\text{M}[\text{Imm} + \text{RegB} + \text{RegI} \times s]\)
- For arrays with elements that are 1/2/4/8 bytes

Examples:
- \(\%\text{eax} = 150, \%\text{ebx} = 400\)
- \(\text{movl}(, \%\text{eax}, 4), \%\text{edx}\)
  - Retrieves value at \(\text{M}[150 \times 4] = \text{M}[600]\) into \(\%\text{edx}\)
- \(\text{movl} \%\text{ecx}, 350(\%\text{ebx}, \%\text{eax}, 2)\)
  - Stores \(\%\text{ecx}\) into \(\text{M}[350 + 400 + 150 \times 2] = \text{M}[1050]\)
IA32 Memory-Reference Summary

Summary chart, from IA32 manual:

<table>
<thead>
<tr>
<th>Base</th>
<th>+</th>
<th>Index</th>
<th>×</th>
<th>Scale</th>
<th>+</th>
<th>Displacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>eax</td>
<td></td>
<td>eax</td>
<td></td>
<td>1</td>
<td></td>
<td>None</td>
</tr>
<tr>
<td>ebx</td>
<td></td>
<td>ebx</td>
<td></td>
<td>2</td>
<td></td>
<td>8-bit</td>
</tr>
<tr>
<td>ecx</td>
<td></td>
<td>ecx</td>
<td></td>
<td>4</td>
<td></td>
<td>16-bit</td>
</tr>
<tr>
<td>edx</td>
<td>+</td>
<td>edx (not esp!)</td>
<td>×</td>
<td>8</td>
<td></td>
<td>32-bit</td>
</tr>
<tr>
<td>esp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ebp</td>
<td></td>
<td>ebp</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>esi</td>
<td></td>
<td>esi</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>edi</td>
<td></td>
<td>edi</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Base, Index, Displacement are all optional
- Scale is only allowed when Index is specified
- Note that esp can only be used as a base value, but never as an index value
IA32 Operand Combinations

- Important constraints on combinations of operand types
  - Source argument can be:
    - Immediate, Register, Memory (direct or indirect)
  - Destination argument can be:
    - Register, Memory (direct or indirect)
  - Both arguments cannot be memory references
    - To move data from one memory location to another, must move Mem1 → Register, then Register → Mem2

- These constraints apply to data movement instructions, and most other instructions too
**IA32 Arithmetic/Logical Operations**

- **Unary arithmetic/logical operations:**
  - `inc Dst`  \[ Dst = Dst + 1 \]
  - `dec Dst`  \[ Dst = Dst - 1 \]
  - `neg Dst`  \[ Dst = -Dst \]
  - `not Dst`  \[ Dst = \sim Dst \]

- **Binary arithmetic/logical operations:**
  - `add Src, Dst`  \[ Dst = Dst + Src \]
  - `sub Src, Dst`  \[ Dst = Dst - Src \]
  - `xor Src, Dst`  \[ Dst = Dst \wedge Src \]
  - `or Src, Dst`  \[ Dst = Dst \lor Src \]
  - `and Src, Dst`  \[ Dst = Dst \land Src \]

- Specify byte-width of operands via suffixes, as usual
  - `decb %cl`
    - Decrements the 1-byte value in \texttt{cl} register
  - `addl 4(%ebp), %eax`
    - Adds \( M[4 + ebp] \) to contents of \texttt{eax}
IA32 Shift Operations

- Shift operations:
  - `shl k, Dst` \( \text{Dst} = \text{Dst} \ll k \)
  - `shr k, Dst` \( \text{Dst} = \text{Dst} \gg k \) (logical)
  - `sal k, Dst` \( \text{Dst} = \text{Dst} \ll k \)
  - `sar k, Dst` \( \text{Dst} = \text{Dst} \gg k \) (arithmetic)
  - `shl, sal` are identical
  - `k` is a constant, or `%cl` register
  - Can only shift values by up to 32 bits
    - ...even when Dst is a 64-bit register!

- Also rotate operations
  - See docs for `rol, ror, rcl, rcr`
  - Similar form, constraints as shift operations
IA32 Multiply, Divide Operations

- Multiplication and division are more challenging
  - 32-bit value \times 32-bit value = 64-bit value
  - 64-bit value ÷ 32-bit value = 32-bit quotient, 32-bit remainder

- Two-argument multiplication:
  - `imul Src, Dst`
  - Use width modifier, as usual: `imull (%ebx), %ecx`
  - For Src, Dst of bit-width \( w \), produces result also of width \( w \)
  - Dst = (Src \times Dst) \mod 2^w

- Also three-argument multiplication:
  - `imul Src1, Src2, Dst`
  - Dst = (Src1 \times Src2) \mod 2^w
IA32 Multiply, Divide Operations (2)

- One-argument multiplication:
  - `imull Src` – 32-bit signed multiplication
    - `edx:eax = Src \times eax`
    - `edx` is top 4 bytes of result, `eax` is bottom 4 bytes of result
  - `mull Src` – 32-bit unsigned multiplication

- One-argument division:
  - `idivl Src` – 32-bit signed division
    - `eax = edx:eax \div Src`
    - `edx = edx:eax \mod Src`
  - `divl Src` – 32-bit unsigned division

- Can use `cltd` to set up `edx:eax` for division
  - `cltd` – “convert long-word to double-word”
    - Sign-extends `eax` into `edx`, creating `edx:eax`
    - Note: in Intel manual, this instruction is called `cdq`
IA32 Multiply, Divide Operations (3)

- Can perform multiplication and division on varying input widths, too

Examples:

- **imulw Src** – 16-bit signed multiplication
  - \( dx:ax = Src \times ax \)
- **idivb Src** – 8-bit signed division
  - \( al = ax \div Src \)
  - \( ah = ax \mod Src \)

- Also variants of **cltd** to set up for division on different input widths
  - **cbtw** – sign-extends al into ax
  - **cwtl** – sign-extends ax into eax
  - **cwtd** – sign-extends ax into dx:ax
IA32 Multiply/Divide Examples

- **Values:**
  - \( x \) at location \( 8(\%ebp) \), \( y \) at location \( 12(\%ebp) \)
  - Both signed values, doublewords (4 bytes)

- **Compute signed product of** \( x \) **and** \( y \)

  ```
  movl 8(%ebp), %eax       # eax = x
  imull 12(%ebp)           # edx:eax = x * y
  pushl %edx               # Save 64-bit result
  pushl %eax               #     onto stack.
  ```

- **Compute signed quotient and remainder of** \( x \div y \)

  ```
  movl 8(%ebp), %eax       # eax = x
  cltd                    # edx:eax = x
  idivl 12(%ebp)           # Compute x \div y
  pushl %eax              #     eax = quotient
  pushl %edx              #     edx = remainder
  ```
Many different instructions for branching in IA32

Simplest version: unconditional jump

- `jmp Label` Direct jump to address
- `jmp *Operand` Indirect jump to address
  - `jmp *%eax` – jumps to address stored in `eax`
  - `jmp *(%eax)` – jumps to address stored at $\text{M}[\text{eax}]$

Can use indirect addressing with unconditional jumps! Very useful in many situations:
- Implementing switch statements: jump tables
- Object oriented programming: virtual function ptr tables
- Other flow-control mechanisms in high-level languages

Other jumps are **conditional** jumps

- Jump occurs based on flags in eflags status register
BRANCH LOGIC, CONDITIONAL JUMPS

- Previous branching logic:

  - Fed output of A to Branching Logic
    - Have one opcode: BRZ
  - Not very extensible to general branching tests
**Branch Logic, Conditional Jumps (2)**

- More powerful branching logic:
  - ALU generates status flags; feed to Branch Logic
  - Conditional jumps use status flags to drive branching
IA32 Conditional Operations

- Status bits in eflags register:
  - CF = carry flag (1 indicates unsigned overflow)
  - SF = sign flag (1 = result is negative)
  - OF = overflow flag (1 indicates signed overflow)
  - ZF = zero flag (1 = result is zero)

- Conditional jump instructions use these flags to control program flow

- All arithmetic and logical operations set these flags
  - Good to know how these instructions affect above flags!

- Can also update these flags with cmp, test
  - `cmp Src2, Src1`
    - Updates flags as for Src1 – Src2 (i.e. sub Src2, Src1)
  - `test Src2, Src1`
    - Updates flags as for Src1 & Src2 (i.e. and Src2, Src1)
  - Src1, Src2 are unchanged by comparison/test operation
  - Can specify size prefixes, as usual: `cmpl %ecx, $0`
IA32 Conditional Jumps

Conditional jumps can only use a label
- Can’t specify an indirect conditional jump

Some operations:
- `je Label`  
  “Jump if equal” (ZF = 1)
- `jne Label`  
  “Jump if not equal” (ZF = 0)
  - `sub Src2, Src1` produces zero result if `Src1 == Src2`
  - `cmp Src2, Src1` sets zero-flag in this case
- `js Label`  
  “Jump if sign” (SF = 1)
- `jns Label`  
  “Jump if not sign” (SF = 0)
  - Jump if answer is negative (SF = 1) or nonnegative (SF = 0)
- `jc/jnc Label`  
  “Jump if [not] carry”
  - Unsigned overflow tests
- `jo/jno Label`  
  “Jump if [not] overflow”
  - Signed overflow tests
IA32 Signed Conditional-Jumps

- **jg Label**  
  “Jump if greater” (signed >)
  - **jnle** is synonym – “Jump if not less or equal”
  - All comparison opcodes have synonyms like this

- Also:
  - **jge**  
    Jump if greater or equal
  - **jl**  
    Jump if less
  - **jle**  
    Jump if less or equal

- These look at sign flag, overflow flag, zero flag
  - Remember: OF = signed overflow, CF = unsigned overflow
  - ZF indicates if Src1 == Src2 (Src2 – Src1 == 0)
  - SF + OF indicate whether Src2 – Src1 > 0 or < 0
    (when nonzero)
    - Logic is slightly involved; see CS:APP §3.6.2 for details
Unsigned comparisons are similar:

- **ja Label** “Jump if above” (unsigned >)
  - **jnbe** is synonym – “Jump if not below or equal”
- Also:
  - **jae** Jump if above or equal
  - **jb** Jump if below
  - **jbe** Jump if below or equal

- These look at carry flag and zero flag
  - CF indicates whether unsigned overflow occurred from Src2 – Src1
  - If Src2 – Src1 generates unsigned overflow, (CF = 1) then Src2 < Src1
  - Again, ZF indicates if Src1 == Src2
IA32 Conditional-Set Instructions

Also a variety of conditional-set instructions

Examples:

- `sete Dst` “Set if equal”
  - Stores ZF into 8-bit target Dst
  - Result is 0 or 1
  - Synonym: `setz` “Set if zero”

- Others:
  - `sets/setns Dst` “Set if sign” / “Set if not sign”
  - `setg Dst` “Set if greater” (signed >)
  - `setl Dst` “Set if less” (signed <)
  - `seta Dst` “Set if above” (unsigned >)
  - `setb Dst` “Set if below” (unsigned <)
  - etc. (same as for conditional-jump instructions)

All instructions modify a single 8-bit destination
But Wait, There’s More!

- Really only scratched the surface of IA32
  - Covered a lot of what you will see in CS24...
  - ...but there’s a lot more where that came from!

- The book reading for Week 2 covers several more instructions, and goes into greater detail
  - Chapter 3 – 3.7

- If you see an instruction you don’t recognize, look it up in the IA32 manuals (provided on Moodle)
  - If it still doesn’t make sense, ask Donnie or a TA 😊
More Advanced Language Features

- Last time, introduced higher-level abstractions
  - Subroutines, the stack, stack frames, frame pointers
- Many different languages, calling conventions, computational models to choose from!
  - e.g. Scheme environment model allows functions to be created and passed around dynamically
  - When an environment or function is no longer used, it is garbage collected automatically

```scheme
Global Environment

make-meter:  
  params: x
  body: (lambda (op . args) ...)

(mmake-meter 3.3)  
  x: 3.3

m1:  
  params: op, args
  body: (cond ((eq? op ...

x 3.3  
```
C Functions

- Start with a simple abstraction: C functions
  - Relatively simple computational model
  - No trapped frames, no lambdas, no garbage collection
- Pretty easy to implement with IA32 assembly

To implement subroutines (tasks from last time):
  - Need a way to pass arguments and return values between caller and subroutine
  - Need a way to transfer control from caller to subroutine, then return back to caller
  - Need to isolate subroutine’s state from caller’s state
EXAMPLE C PROGRAM

- A simple accumulator:
- Uses a global variable to store current value
- Functions to update accumulator, or reset it
- Main function to exercise the accumulator

Three kinds of variables
- Global variables
- Function arguments
- Local variables

```c
int value;

int accum(int n) {
    value += n;
    return value;
}

int reset() {
    int old = value;
    value = 0;
    return old;
}

int main() {
    int i, n;
    reset();
    for (i = 0; i < 10; i++) {
        n = rand() % 1000;
        printf("n = %d\ttaccum = %d\n", n, accum(n));
    }
    return 0;
}
```
**Example C Program (2)**

- Three kinds of variables
  - Global variables
  - Function arguments
  - Local variables

- C computational model:
  - *(approximately...)*
  - A global environment at the top level
  - When a function is called, a new environment is created to hold args, local variables
  - All functions in the file can access the contents of the global environment

```c
int value;

int accum(int n) {
    value += n;
    return value;
}

int reset() {
    int old = value;
    value = 0;
    return old;
}

int main() {
    int i, n;
    reset();
    for (i = 0; i < 10; i++) {
        n = rand() % 1000;
        printf("n = %d\ttaccum = %d\n", n, accum(n));
    }
    return 0;
}
```
After `reset()` call:

- Also, `rand()` has returned 807

```c
int value;

int accum(int n) {
    value += n;
    return value;
}

int reset() {
    int old = value;
    value = 0;
    return old;
}

int main() {
    int i, n;

    reset();
    for (i = 0; i < 10; i++) {
        n = rand() % 1000;
        printf("n = %d \t accum = %d\n", n, accum(n));
    }
    return 0;
}
```
EXAMPLE C PROGRAM (4)

- **accum(807) call:**
  - Function invocation has its own local environment specifying n = 807

- **Global Environment**

```
int value;
int accum(int n) {
    value += n;
    return value;
}
int reset() {
    int old = value;
    value = 0;
    return old;
}
int main() {
    int i, n;
    reset();
    for (i = 0; i < 10; i++) {
        n = rand() % 1000;  
        printf("n = %d\taccum = %d\n", n, accum(n));
    }
    return 0;
}
```
**Representing C Model**

- **Global variables**
  - Store at specific location
  - Reference via absolute address

- **Function arguments**
  - Store on stack
  - Pushed by caller before invoking subroutine
  - IA32: frame pointer plus some offset

- **Local variables**
  - Also store on stack
  - Subroutine manages space for these variables
  - IA32: frame pointer minus some offset

```c
int value;

int accum(int n) {
    value += n;
    return value;
}

int reset() {
    int old = value;
    value = 0;
    return old;
}

int main() {
    int i, n;
    reset();
    for (i = 0; i < 10; i++) {
        n = rand() % 1000;
        printf("n = %d\taccum = %d\n", n, accum(n));
    }
    return 0;
}
```
IA32 Subroutine Calls

- IA32 provides specific features for subroutines

Registers:
- **esp** = stack pointer
  - Stack grows “downward” in memory
  - *push* decrements *esp*, then stores value to (%esp)
  - *pop* retrieves value at (%esp), then increments *esp*
- **ebp** = base pointer
  - IA32 name for frame pointer

Instructions:
- **call** Addr
  - Pushes *eip* onto stack (*eip* references next instruction)
  - Sets *eip* = Addr
- **ret**
  - Pops top of stack into *eip*
IA32 Subroutine Calls and gcc

- Many different ways to organize stack frames!
- A calling convention is a particular way of passing information to/from a subroutine
- The cdecl convention is frequently used on x86 for C subroutines
- Both the procedure caller and the callee have to coordinate the operation!
  - Shared resources: the stack, the register file
- Calling convention specifies:
  - Who sets up which parts of the call
  - What needs to be saved, and by whom
  - How to return values back to the caller
  - Who cleans up which parts of the call
CDECL: PASSING ARGUMENTS

- Caller is responsible for setting up arguments
- Arguments pushed onto stack in reverse order
  - Last argument is pushed first
  - 2nd argument pushed next-to-last
  - 1st argument is pushed last
- Two benefits:
  - Earlier arguments have a lower offset added to the frame pointer
  - If procedure is passed more args than it expects, it doesn’t break the procedure’s code
- Primitive values generally take up a doubleword (4 bytes) on stack
  - e.g. ints, floats, pointers (on 32-bit OS)
CDECL: INVOKING THE PROCEDURE

- Caller uses `call` to invoke the procedure
  - Pushes `eip` of `next` instruction onto the stack

- First task of callee:
  - Set up frame pointer for this function call
  - `ebp` is used for the frame pointer on IA32

- Must preserve caller’s frame pointer!

- Typical code:
  - `pushl %ebp`
  - `movl %esp, %ebp`

- Now:
  - `4 (%ebp)` = Return address
  - `8 (%ebp)` = Arg1 value
  - `12 (%ebp)` = Arg2 value
CDECL: SAVING REGISTERS

“Callee must save ebp before it modifies it”

A general issue:
- The register file is a shared resource
- Calling convention must specify how registers are managed

Callee-save registers:
- When callee returns, values must be same as when subroutine was invoked
- ebp, ebx, esi, edi are callee-save registers

Caller-save registers:
- Callee may change these registers without saving them!
- The caller must save these registers before the call, if the old values need to be preserved
- eax, ecx, edx are caller-save registers
CDECL: RETURNING RESULTS

- For now, only consider simple results
  - e.g. `int` or pointer
- In these cases, callee returns the result in `eax`
  - Set `eax` to result, restore `ebp`, then return to caller
- Who removes the arguments from the stack??
- In cdecl, the `caller` cleans up stack
  - Linux / GNU calling convention
  - e.g. can add a constant to `esp` to remove arguments
- In stdcall (Win32), the `callee` cleans up stack
  - Microsoft Visual C++ calling convention
  - IA32 includes version of `ret` that takes an argument
  - `ret n`
    - Sets `eip` to `(esp)`, then pops `n` bytes off stack
LOCAL VARIABLES

- Procedures sometimes need space for local variables
  - Compiler figures out how much space, from the source code
  - Sometimes allocates more than is strictly required
- Local variables reside just below the frame pointer
  - Accessed via \(-\text{off}(\%ebp)\)
- Common pattern:
  - Allocate \(n\) bytes on stack for local vars
  - \textit{subl} \(\$n, \%esp\) (or \textit{addl} \(-n, \%esp\))
- Example: allocate 8 bytes for local vars
  - \textit{subl} \$8, \%esp
- \textbf{Note:} these memory locations are \textit{not} initialized!
  - Contains whatever values were in that memory before the call…
CDECL AND FRAME POINTER

- **Note**: subroutines don’t *always* use `%ebp`
  - `%ebp` is primarily used when a function manages its own local state on the stack

- **Example**: a function that adds two values, and returns the result
  ```c
  int add(int a, int b) {
    return a + b;
  }
  ```
  - Doesn’t have any local variables!

- In this case, subroutine can access *args* `a` and `b` directly, via `%esp`:
  - `a` can be accessed via `4 (%esp)`
  - `b` can be accessed via `8 (%esp)`
  - Return address is at `(%esp)`

![Stack Diagram](image)
Look at how our simple C accumulator program is implemented in IA32 assembly language
- Memory layout strategy for global variables, local variables, and arguments
- `gcc` and the cdecl calling convention

Begin to look at other C language features
- C flow-control statements, and how they are translated into IA32 assembly