X86 Assembly Programming
with GNU assembler

Lecture 7

Instructor:
Alvin R. Lebeck

Some Slides based on those from
Randy Bryant and Dave O’Hallaron

Admin

• Homework #3 Due Monday Feb 14 11:59pm
• Reading: Chapter 3
• Note about pointers: You must explicitly initialize/set to NULL

Assembly Programming (x86)
• Quick Instruction Review
• Assembly Language
• Simple one function program
• High level constructs (control)
• Interfacing to a C program
• Procedure Calling Conventions
Some Arithmetic and Logical Operations

- **Two Operand Instructions:**
  - Format: Computation
  - `addl` Src, Dest: \( \text{Dest} = \text{Dest} + \text{Src} \)
  - `subl` Src, Dest: \( \text{Dest} = \text{Dest} - \text{Src} \)
  - `imull` Src, Dest: \( \text{Dest} = \text{Dest} \times \text{Src} \) **Also called shll**
  - `sall` Src, Dest: \( \text{Dest} = \text{Dest} \ll \text{Src} \)
  - `sarl` Src, Dest: \( \text{Dest} = \text{Dest} \gg \text{Src} \) **Arithmetic**
  - `shr` Src, Dest: \( \text{Dest} = \text{Dest} \gg \text{Src} \) **Logical**
  - `xorl` Src, Dest: \( \text{Dest} = \text{Dest} \oplus \text{Src} \)
  - `andl` Src, Dest: \( \text{Dest} = \text{Dest} \& \text{Src} \)
  - `orl` Src, Dest: \( \text{Dest} = \text{Dest} \mid \text{Src} \)

- Watch out for argument order!
- No distinction between signed and unsigned int (why?)

Some Arithmetic Operations

- **One Operand Instructions**
  - `incl` Dest: \( \text{Dest} = \text{Dest} + 1 \)
  - `decl` Dest: \( \text{Dest} = \text{Dest} - 1 \)
  - `negl` Dest: \( \text{Dest} = -\text{Dest} \)
  - `notl` Dest: \( \text{Dest} = \sim\text{Dest} \)

- See book for more instructions
- Note: suffix “l” is for 32-bit values, “b” for byte, “w” for 16-bit
Address Computation Instruction

- **leal** `Src, Dest`  
  - `Src` is address mode expression  
  - Set `Dest` to address denoted by expression

- Uses
  - Computing addresses without a memory reference  
    - E.g., translation of `p = &x[i];`  
  - Computing arithmetic expressions of the form `x + k*y`  
    - `k = 1, 2, 4,` or `8`

- **Example**

```c
int mul12(int x)  
{  
    return x*12;  
}
```

**Converted to ASM by compiler:**

```
leal (%eax,%eax,2), %eax ; t <- x+x*2
sall $2, %eax ; return t<<2
```

Condition Codes (Implicit Setting)

- Single bit registers  
  - CF Carry Flag (for unsigned)  
  - SF Sign Flag (for signed)  
  - ZF Zero Flag  
  - OF Overflow Flag (for signed)

- Implicitly set (think of it as side effect) by arithmetic operations

- Not set by lea instruction

- Explicitly set by compare and test instructions

- Allow for conditional change of PC via jump instructions
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

X86 w/ Gnu Assembly Language

- One instruction per line.
- **Numbers** are base-10 integers or Hex w/ leading 0x.
- **Identifiers:** alphanumeric, _, . string starting in a letter or _
- **Labels:** identifiers starting at the beginning of a line followed by ":

- **Comments:** everything following # till end-of-line.
- **Directives:** convey information to the assembler
- **Instruction format:** Space and "," separated fields.
  - `[Label:] <op> src, dest` [ # comment]
  - `[Label:] .Directive [arg1], [arg2], . . ."
Assembly Language (cont.)

- **Directives**: tell the assembler what to do...
- **Format** “."<string> [arg1], [arg2] ... 

- **Examples**
  - `.data [address] # start a data segment. [optional begining address]
  - `.text [address] # start a code segment.
  - `.globl # declare a label externally visible
  - `.ascii <string> # store a string in memory.
  - `.asciz <string> # store a null terminated string in memory
  - `.long w1, w2, ..., wn # store n 32-bit values in memory.
  - `.align n # align segment on 2^n byte boundary.

---

A simple function

- **Add two numbers together x and y**

  ```assembly
  .text          # declare text segment
  .globl _sum    # declare function name for external call

  _sum:          # label for function
    movl x, %edx # load M[x] into %edx
    movl y, %eax # load M[y] into %eax
    addl %edx, %eax # %eax = %eax + %edx
    movl %eax, x # store %eax into M[x]
    ret          # return to calling function
  
  .data          # declare data segment
  x: .long 10    # initialize x to 10
  y: .long 2     # initialize y to 10
  ```
Typical Code Segments-- IF

if (x != y)
    x = x + y;
y = 2;

- General Rule is to invert condition
if (x == y) goto skip
    x = x + y
skip: y = 2;

- Assume %ecx contains x and %edx contains y
    cmpl %ecx, %edx
    je skip
    addl %edx, %ecx
skip:
    movl $2, %edx

Typical Code Segments-- IF-else

if (x != y)
    x = x + y;
else
    x = x - y;

- Invert condition check and use goto
if (x == y) goto L1
    x = x - y;
goto done
L1: x = x + y;
done:

- Assume %ecx contains x and %edx contains y
    cmpl %ecx, %edx           # compute condition
    je L1                   # checking !(condition)
    subl %edx, %ecx         # x = x - y
    jmp done
L1:
    addl %edx, %ecx         # x = x - y
done:
The C code

int sum(){
    int i;
    int sum = 0;
    for(i=0; i <= 100; ++i)
        sum = sum + i*i;
    return(sum);  // put sum into %eax
}

Let's write the assembly ... :)

Sum array

Task: sum together the integers stored in memory

.text
.globl _sum

_sum:
# Fill in what goes here

.data
num_array: .long 35, 16, 42, 19, 55, 91, 24, 61, 53
Assembly Programming in Eclipse

- Add source file of type <none>
- Name source file with .S suffix (must be capital S)
- We are using 32-bit (IA32), so we need to tell compiler & assembler
  - Project->properties->C/C++ Build->Settings
    - MAC OS C Linker: add –m32 after gcc
    - GCC Assembler: add –arch i386 after as
    - GCC C Compiler: add –m 32 after gcc

Calling an Assembly Function from C

- Main in normal C file
- Declare function using “extern”
  - E.g., extern int foo();
  - Foo is our assembly function in a .S file
- Function name (label) must start with _
  - E.g., _foo:
    - C program uses foo (compiler adds the _)
- Examples in Eclipse
Review: Procedure Call and Return

```c
int equal(int a1, int a2) {
    int tsame;
    tsame = 0;
    if (a1 == a2)
        tsame = 1;
    return(tsame);
}
```

```c
main() {
    int x,y,same;
    x = 43;
    y = 2;
    same = equal(x,y);
    // other computation
}
```

### Assembly Level
- **Must bridge gap between HLL and ISA**
- **Supporting Local Names**
- **Passing Arguments/Parameters (arbitrary number?)**
- **What data structure?**

### Procedure Call GAP

#### ISA Level
- call and return instructions

#### C Level
- Local Name Scope
  - change tsame to same
- Recursion
- Arguments/parameters and Return Value (functions)

#### Assembly Level
- Must bridge gap between HLL and ISA
- Supporting Local Names
- Passing Arguments/Parameters (arbitrary number?)
**Procedure Call (Stack) Frame**

- Procedures use a frame in the stack to:
  - Hold values passed to procedures as arguments.
  - Save registers that a procedure may modify, but which the procedure’s caller does not want changed.
  - To provide space for local variables.
  - To evaluate complex expressions.

**IA32/Linux Stack Frame**

- **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
Register Saving Conventions

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

- Can Register be used for temporary storage?

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

who:
  . . .
  movl 8(%ebp), %edx
  addl $18243, %edx
  . . .
  ret
```

- This could be trouble ➔ something should be done!
  - Need some coordination

Conventions

- “Caller Save”
  - Caller saves temporary values in its frame before the call
- “Callee Save”
  - Callee saves temporary values in its frame before using
IA32/Linux+Windows Register Usage

- `%eax, %edx, %ecx`
  - Caller saves prior to call if values are used later

- `%eax`
  - also used to return integer or pointer 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

---

IA32/GCC Procedure Calling Conventions

**Calling Procedure**

- **Step-1**: Save *caller-saved* registers
  - Save registers `%eax, %ecx, %edx` if they contain live values at the call site.

- **Step-2**: Setup the arguments:
  - Push arguments onto the stack in reverse order

- **Step-3**: Execute a *call* instruction.
IA32/GCC Calling Conventions (cont.)

Called Routine
- Step-1: Update the frame pointer
  ```
  pushl %ebp
  movl %esp, %ebp
  ```
- Step-2: Allocate space for frame
  - Subtract the frame size from the stack pointer
    ```
    subl $<frame-size>, %esp
    ```
  - Space is for local variables and saved registers
  - May often allocate more space than necessary.
- Step-3: Save callee-saved registers in the frame.
  - Registers %ebx, %esi, %edi are saved if they are used.

On return from a call
- Step-1: Put returned value in register %eax.
  (if value is returned)
- Step-2: Restore callee-saved registers.
  - Restore %ebx, %esi, %edi if needed
- Step-3: “Pop” the stack
  ```
  leave
  ```
  - Equivalent to
  ```
  movl %ebp, %esp
  popl %ebp
  ```
- Step-4: Return
  ```
  ret
  ```
  %eip = M[%esp]; %esp = %esp - 4
C Function call with one parameter

```c
#include <stdio.h>
#include <stdlib.h>

// declare the function as externally defined
// computes sum of elements 0 to i of an array defined in sum_array
extern int sum_array(int i);

int main(void) {
    int result;
    result = sum_array(7);
    printf("Array sum = %d\n", result);
    return EXIT_SUCCESS;
}
```

Sample Function

```asm
.text
.globl _sum_array
    pushl %ebp
    movl %esp, %ebp
    movl 8(%ebp), %eax
    leal num_array, %edx
    leal (%edx,%eax,4), %ecx
    movl $0, %eax
    loop:
    addl (%edx), %eax
    addl $4, %edx
    cmpl %ecx, %edx
    jl loop
    leave
    ret

.data
    num_array: .long 35, 16, 42, 19, 55, 91, 24, 61, 53
```
x86 Assembly Programming

- Assembly Language
  - Text file (with .S for eclipse)
  - One instruction per line
  - Labels, directives, etc.

- High-level Constructs
  - If
  - If-else
  - Loops
  - Memory (array) accesses

- Calling assembly from C

- Calling Conventions

- Examples in “docs” section of course web site

- Next time recursion & pointers!