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
- 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`: `Dest = Dest + Src`
  - `subl` `Src, Dest`: `Dest = Dest - Src`
  - `imull` `Src, Dest`: `Dest = Dest * Src`
  - `sall` `Src, Dest`: `Dest = Dest << Src`  Also called `shll`
  - `sarl` `Src, Dest`: `Dest = Dest >> Src`  Arithmetic
  - `shrl` `Src, Dest`: `Dest = Dest >> Src`  Logical
  - `xorl` `Src, Dest`: `Dest = Dest ^ Src`
  - `andl` `Src, Dest`: `Dest = Dest & Src`
  - `orl` `Src, Dest`: `Dest = Dest | Src`

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

Some Arithmetic Operations

One Operand Instructions

- `incl` `Dest`: `Dest = Dest + 1`
- `decl` `Dest`: `Dest = Dest - 1`
- `negl` `Dest`: `Dest = - Dest`
- `notl` `Dest`: `Dest = ~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:

```asm
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>
```

- **Return address:**

```
8048553: 50  pushl %eax
```

- **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**

```
.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 2
```
Typical Code Segments-- IF

if \( x \neq y \)
   \[ x = x + y; \]
   \[ y = 2; \]
- **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 \( x \neq y \)
   \[ x = x + y; \]
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
  `addl %edx, %ecx` # \( x = x + y \)
  `jmp done`
L1:
  `subl %edx, %ecx` # \( x = x - y \)
done:
The C code

```c
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

```assembly
.text
.globl sum

sum:
    # Fill in what goes here

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

Review: Procedure Call and Return

```c
int equal(int a1, int a2) {
    int tsame;
    tsame = 0;
    if (a1 == a2)
        tsame = 1;
    return(tsame);
}
main()
{
    int x, y, same;
    x = 43;
    y = 2;
    same = equal(x, y);
    // other computation
}```
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?)
- What data structure?

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. (variables with local scope)
  - 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?**

  ```
  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
Register Saving Conventions

- When procedure \texttt{yoo} calls \texttt{who}:
  - \texttt{yoo} is the \texttt{caller}
  - \texttt{who} is the \texttt{callee}

- Can Register be used for temporary storage?

- 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.
IA32/GCC Calling Conventions (cont.)

**On return from a call**

- **Step-1:** Put return 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;
}
```

27
Sample Function

```
.text
.globl sum_array
# declare the text segment
# declare the function label (note the _ in this label)
# the C program calls sum(int)

sum_array:
pushl $ebp
# save old frame pointer
movl $esp, $ebp
# set new stack pointer
movl $8(%ebp), $eax
# read arg1 from stack, put into $eax
lea num_array, %edx
# load address of num_array into %edx (p = &num_array)
lea (%edx,%eax,4), %ecx
# load address of num_array+arg into %ecx
movl $0, %eax
# move 0 to running sum (%eax)
loop:
# label for loop structure
addl (%edx), %eax
# add value *p to running sum (%eax)
addl $4, %edx
# increment pointer in memory (p++)
cmpl %ecx, %edx
# compare pointer to termination (p < (num_array+arg1))
jl loop
# jump to loop if (p < (num_array+arg1))
leave
# prepare stack for return (movi $esp, $ebp; popl $ebp)
ret
# return to calling routine (result is in %eax)

.data
# declare data segment and array with 9 32-bit integers
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!