Homework

• Reading
  – PAL, pp 201-216, 297-312

• Machine Projects
  – Finish mp2warmup
    • Questions?
  – Start mp2 as soon as possible

• Labs
  – Continue labs with your assigned section
Coding and Calling Functions

• An assembly language programmer handles a lot of details to coordinate the code for calling a function and the code in the function itself.

• There are two mechanisms in the instruction set for calling and returning from functions:
  • Linux system calls and returns
    int $0x80 and iret
  • C library style function calls and returns
    call and ret
Coding and Calling Functions

- A really “old school” way to pass data back and forth between assembly language functions is to leave all data in “global memory”
- This was really very efficient back when CPU’s were not very powerful and some did not have hardware supported stack mechanisms
- Today we understand the software maintenance problem that this choice creates and the CPU’s are powerful enough for us to not need to do it
Coding and Calling Functions

• A somewhat “old school” way to call functions:
  – Load up registers with input values before call
  – Unload return values from registers after return (if any)

• This is still in use in Linux system calls, such as:

  # <unistd> write as a Linux system call
  movl $4, %eax     # system call value
  movl $1, %ebx     # file descriptor
  movl $output, %ecx  # *buffer
  movl $len, %edx    # length
  int $0x80       # call to system
Coding and Calling Functions

• We won’t use the Linux system call and return mechanism in this course, but:
  – I feel that you should be aware of it and recognize it when the textbook uses it in an example
  – We’ll use the `iret` instruction later with hardware interrupts

• We will use the `call` and `ret` mechanism as is typically used for C library function calls
Call/Return to/from our C Function

# C compiler generated code for:
# static int z = mycode(x, y);
.text
...
pushl y     # put arg y on stack
pushl x     # put arg x on stack
call _mycode # call function mycode
addl $8, %esp # purge args from stack
movl %eax, z # save return value
...
.data
z:
.long 0     # location for variable z
C Library Coding Conventions

- Use same function name as used in the calling C program except add a leading underscore ‘_’
- Setup C compiler stack frame (optional)
- Use only %eax, %ecx, and %edx to not affect registers the C compiler expects to be preserved
- Save/restore any other registers on stack if used
- Put return value in %eax
- Remove C compiler stack frame (optional)
- Return
C Library Coding Conventions

• Example of Assembly code for C function:

```c
int mycode(int x, int y)
{
    /* automatic variables */
    int i;
    int j;
    ...  
    return result;
}
```
C Library Coding Conventions

• Start with basic calling sequence discussed earlier

```assembly
.text
.globl _mycode
_mycode:
    ... # entry point label
    movl xxx, %eax # code as needed
    ret # set return value
    ret # return to caller
.end
```
C Library Coding Conventions

• If function has arguments or automatic variables (that require n bytes), include this optional code

• Assembly language after entry point label (enter):
  
  pushl %ebp
  movl %esp,%ebp
  subl $n,%esp
  
  # set up stack frame
  # save %esp in %ebp
  # automatic variables

• Assembly language before ret (leave):
  
  movl %ebp, %esp
  popl %ebp
  
  # restore %esp from %ebp
  # restore %ebp
C Compiler Reserved Registers

• The C compiler assumes it can keep data in certain registers (%ebx, %ebp) when it generates code.

• If assembly code uses compiler’s reserved registers, it must save and restore the values for the calling C code.

• Example:

```
. . . # we can’t use %ebx yet
pushl %ebx # save register contents

. . . # we can use %ebx now
popl %ebx # restore %ebx

. . . # we can’t use %ebx any more
ret
```

Matching pair
C Library Coding Conventions

• State of the stack during function execution:

<table>
<thead>
<tr>
<th>Lower level Function Calls</th>
<th>%esp</th>
<th>%ebp</th>
<th>Points to previous stack frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>%ebx</td>
<td>j</td>
<td>i</td>
<td>%ebp</td>
</tr>
</tbody>
</table>

Lower Level Function Returns

Automatic Variables
- \( i = -4(%ebp) \)

Return Address
- \( x = 8(%ebp) \)

Argument Variables
- \( j = -8(%ebp) \)
- \( y = 12(%ebp) \)
Turning It Around

•Calling a C function from Assembly Language
  –Can use printf to help debug assembly code (although it’s better to use either tutor or gdb as a debugger)
  –Assume C functions “clobber” the contents of the %eax, %ecx, and %edx registers
  –If you need to save them across a C function call:
    • Push them on the stack before the call
    • Pop them off the stack after the return
Printing From Assembler

• The C calling routine (helloc.c according to our convention) to get things going is:

```c
extern void hello();
int main(int argc, char ** argv)
{
    hello();
    return 0;
}
```
Printing From Assembler

• Assembly code to print Hello:

```assembly
.globl _hello
.text
_hello:
pushl $hellostr # pass string argument
call   _printf   # print the string
addl  $4, %esp  # restore stack
ret
.data
hellostr:
  .asciz "Hello\n" # printf format string
.end
```

Printing from Assembler

• Assembly code to use a format statement and variable:

```
... pushl x          # x is a 32-bit integer
pushl $format     # pointer to format
call _printf      # call C printf routine
addl $8, %esp     # purge the arguments
...
```

```
x:       .long 0x341256
format:  .asciz “x is: %d”
```
Preserving Compiler Scratch Registers

• C compiler assumes that it can use certain registers when it generates code (%eax, %ecx, and %edx)
• A C function may or may not clobber the value of these registers
• If assembly code needs to preserve the values of these registers across a C function call, it must save/restore their:

```
  . . .  # if ecx is in use
  pushl %ecx  # save %ecx
  call _cFunction  # may clobber ecx
  popl %ecx  # restore %ecx
  . . .  # ecx is OK again
```
Function Calling Summary

• Passing information to functions in assembly
  – In global memory (Horrors!)
  – In registers (“old school” for assembly calling assembly)
  – On the stack (for C compatible functions)

• For C compatible functions, the stack is used to hold:
  – Arguments
  – Return address
  – Previous frame’s %ebp value
  – Automatic variables
  – Saved values for other compiler used registers (if needed)

• Passing information from function back to caller
  – In registers (%eax is C compiler return value convention)