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 (if any) before call
  – Unload return values (if any) from registers after return

• 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
# 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

```
.text
.globl _mycode
_mycode:
  ...
  movl xxx, %eax
  ret
.end
```

# entry point label
# code as needed
# set return value
# return to caller
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):

```assembly
pushl %ebp          # set up stack frame
movl %esp,%ebp     # save %esp in %ebp
subl $n,%esp       # automatic variables
```

• Assembly language before `ret` (leave):

```assembly
movl %ebp, %esp    # restore %esp from %ebp
popl %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
```
C Library Coding Conventions

- State of the stack during function execution:

  ![Stack Diagram]

  - Lower level Function Calls:
    - %esp
    - %ebp
    - %ebx
    - j
    - i
    - %ebp
    - %eip
    - x
    - y

  - Lower Level Function Returns:
    - i = -4(%ebp)
    - x = 8(%ebp)
    - j = -8(%ebp)
    - y = 12(%ebp)

  - Automatic Variables:
    - i = -4(%ebp)
  - Argument Variables:
    - x = 8(%ebp)
    - y = 12(%ebp)

  - Return Address:
    - %eip

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:

```assembly
... # if ecx is in use
pushl %ecx  # save %ecx
call _cFunction  # may clobber ecx
popl %ecx  # restore %ecx
... # ecx is OK again
```
Integrating C and Assembly

• Pick up the makefile from ~/bobw/cs341/mp2
• Always read the makefile for a program first!
• The makefile expects a “matched pair” of source names
  – C driver filename is mycodec.c
  – Assembly filename is mycode.s
• The make file uses macro substitutions for input:
  – The format of the make command is:
    make A=mycode
Note: Examples are located in: ~/bobw/cs341/examples/lecture06
Example: Function cpuid

- C “driver” in file cpuidc.c to execute code in cpuid.s

```c
#include <stdio.h>

extern char *cpuid(); /* our .s file is external*/

int main(int argc, char **argv)
{
    printf("The cpu ID is: %s\n", cpuid());
    return 0;
}
```
Example: Function cpuid

- Assembly code for function in file cpuid.s
  
  ```assembly
  # cpuid.s C callable function to get cpu ID value
  .data
  buffer:
      .asciz "Overwritten!"  # overwritten later
  .text
  .globl _cpuid
  _cpuid:
      movl $0, %eax          # zero to get Vendor ID
      cpuid                 # get it
      movl $buffer, %eax    # point to string buffer
      movl %ebx, (%eax)     # move four chars
      movl %edx, 4(%eax)    # move four chars
      movl %ecx, 8(%eax)    # move four chars
      ret                   # string pointer is in %eax
  .end
  ```
• Our assembler does not actually support cpuid instruction, so I made the code self-modifying:

```assembly
.cpuid:
    movb $0x0f, cpuid1  # patch in the cpuid first byte
    movb $0xa2, cpuid2  # patch in the cpuid second byte
    movl $0,%eax        # input to cpuid for ID value
    cpuid1:               # hex for cpuid instruction here
        nop                 # 0x0f replaces 0x90
    cpuid2:               # 0xa2 replaces 0x90
        nop
```

Self Modifying Code 😞

• Obviously, the self modifying code I used for this demonstration would not work if:
  – The code is physically located in PROM/ROM
  – There is an O/S like UNIX/Linux that protects the code space from being modified (A problem that we avoid using the Tutor VM)

• Try justifying the “kludge” in the next slide to the maintenance programmer!!
int main(int argc, char **args) {
    char function [100];  // array to hold the machine code bytes of the function

    // I put machine code instructions byte by byte into the function array:
    // Instruction 1: movl the &function[6] to the %eax (for return value)
    // Instruction 2: the machine code for a ret instruction (0xc3)
    // Following the ret instruction, I put the bytes of the string “Hello World”
    // with a null terminator into the array starting at function[6]

    function[0] = 0xb8;     // op code for movl immediate data to %eax
    function[1] = (int) &function[6] & 0xff;       // immediate data field
    function[2] = (int) &function[6] >>  8 & 0xff; // little endian format
    function[3] = (int) &function[6] >> 16 & 0xff; // four bytes for the
    function[4] = (int) &function[6] >> 24 & 0xff; // address of the string
    function[5] = 0xc3;    // op code for ret
    function[6] = 'H';     // string whose address is returned is stored here
    . . .                  // rest of characters in string omitted for clarity
    function[17] = 0;      // null terminator for the string

    // execute the function whose address is the array
    printf("%s\n", (* (char * (*)()) function)());
    return 0;
}