ECE 498 – Linux Assembly Language Lecture 2

Vince Weaver

http://www.eece.maine.edu/~vweaver

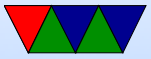
vincent.weaver@maine.edu

15 November 2012

Assembly Example: tb_asm

A small game coded entirely in x86 Linux Assembly Language:

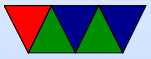
`http://www.deater.net/weave/vmwprod/tb1/tb_asm.html`



Assembly Example: ll

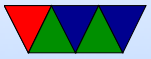
A system-printing utility in multiple assembly languages:

`http://www.deater.net/weave/vmwprod/asm/ll/ll.html`



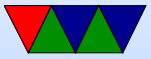
Sample programs

The code for these can be downloaded from the course website.



A first x86 assembly program: hello_exit

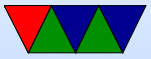
```
.equ SYSCALL_EXIT,     1

    .globl _start

_start:
    #=====
    # Exit
    #=====
exit:
    mov $5,%ebx
    mov $SYSCALL_EXIT,%eax  # put exit syscall number (1) in eax
    int $0x80               # and exit
```



`hello_exit` example

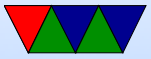
Assembling/Linking using `make`, running, and checking the output.

```
lecture2$ make hello_exit_x86
as --32 -o hello_exit_x86.o hello_exit_x86.s
ld -melf_i386 -o hello_exit_x86 hello_exit_x86.o
lecture2$ ./hello_exit_x86
lecture2$ echo $?
5
```

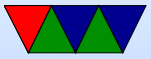


New Things in the `hello_exit` Example

- labels: text string, end with :
- comments: on x86/as it's #
- on x86/as $ specifies a constant
- Intel vs UNIX opcode argument order: be careful, on x86 they are reversed; the order in the Intel manuals is opposite of what we'll use in the GNU assembler
- `.equ` is like a `#define`

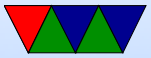


- `int $0x80` is a 32-bit x86 syscall



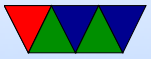
x86 32-bit syscall interface

- called with `int $0x80`
- system call number put in %eax
- system call numbers can be found in /usr/include/asm/unistd_32.h
- arguments go in %ebx, %ecx, %edx, %esi, %edi, %ebp
  Any extra arguments are passed on the stack
  Other operating systems on x86 typically pass *all*



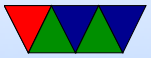
arguments on the stack.
Return value and errno is in %eax.

- see manpages 2 for system call documentation (for example `man 2 exit`)
- Anything you see a Linux box do is some combination of these syscalls!

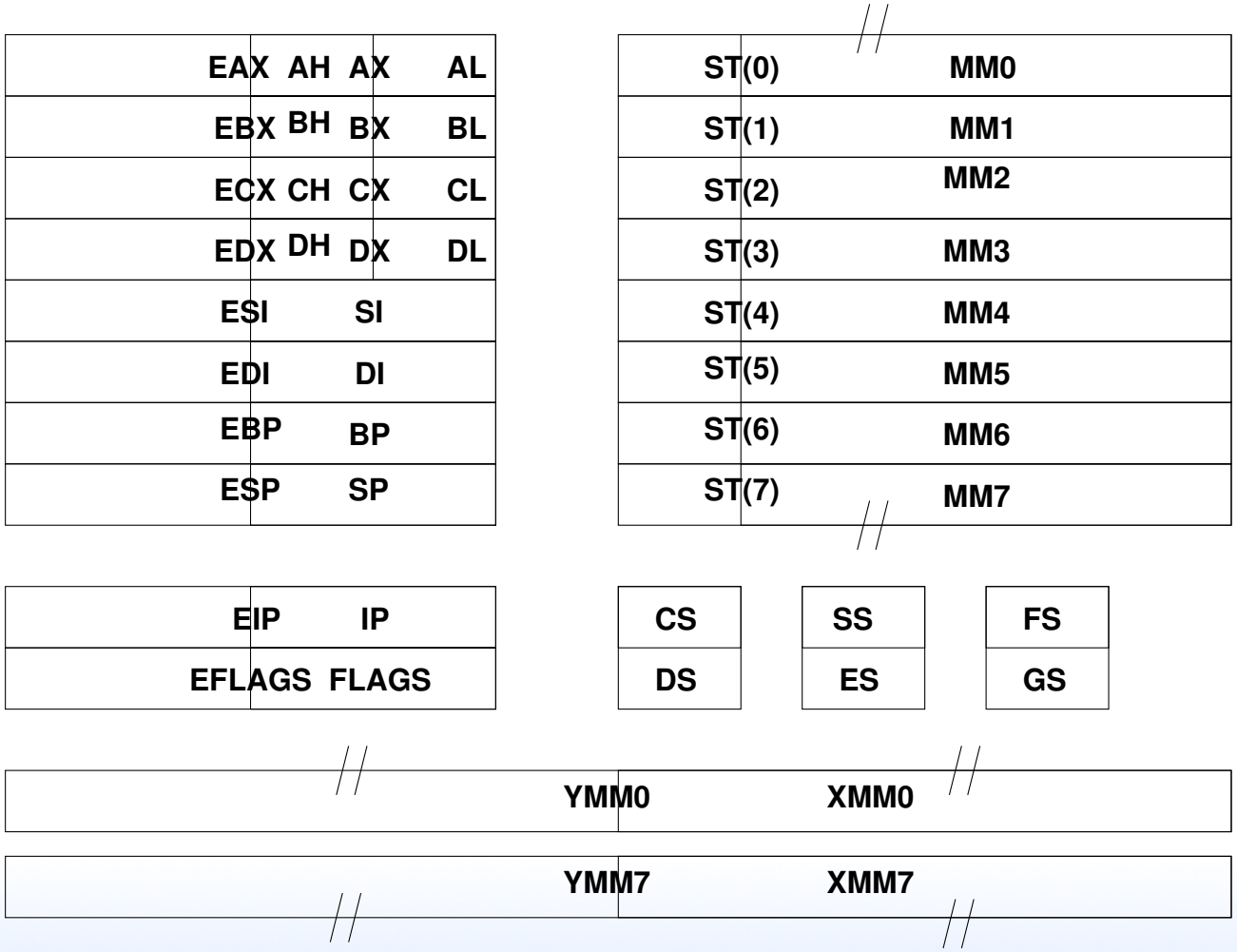


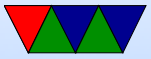
Let's look at our executable

- `ls -la ./hello_exit_x86`
  Check the size
- `readelf -a ./hello_exit_x86`
  Look at the ELF executable layout
- `objdump --disassemble-all ./hello_exit_x86`
  See the machine code we generated
- `strace ./hello_exit_x86`
  Trace the system calls as they happen.



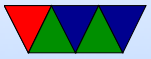
x86 32-bit registers



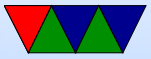


x86 32-bit registers

- 8 32-bit registers, EAX, EBX, ECX, EDX, EDI, ESI, EBP, ESP (ESP is the stack pointer)
- Lower 16-bits can be accessed separately (AX, BX, CX, DX, DI, SI, BP, SP)
- The low/high 8-bits of some can be accessed too (AH/AL, BH/BL, CH/CL, DH/DL)
- 8 80-bit x87 floating point registers, in a stack configuration ST(0) - ST(7)

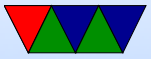


- 8 64-bit MMX MM0-MM7 (overlap with x87)
- 8 128-bit SSE XMM0-XMM7
- 8 256-bit AVX YMM0-YMM7 (overlap with SSE)
- EFLAGS and EIP
- 16-bit segment registers (mostly unused in modern systems)



hello_world example

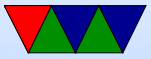
```
.equ SYSCALL_EXIT,  1
.equ SYSCALL_WRITE, 4

    .globl _start
_start:
    mov $1,%ebx          # stdout
    mov $hello,%ecx
    mov $13,%edx         # length
    mov $SYSCALL_WRITE,%eax
    int $0x80

        #================================
        # Exit
        #================================
exit:
    mov $5,%ebx
    mov $SYSCALL_EXIT,%eax  # put exit syscall number (1) in eax
    int $0x80               # and exit

.data
    hello:  .ascii "Hello World!\n"
```



hello_world

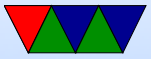
Assembling/Linking using make, running, and checking the output.

```
lecture2$ make hello_world_x86
as --32 -o hello_world_x86.o hello_world_x86.s
ld -melf_i386 -o hello_world_x86 hello_world_x86.o
lecture2$ ./hello_world_x86
Hello World!
```



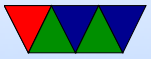
New things to note in `hello_world`

- assembler directives start with `.`
- Data segment directives, used for allocating space for variables.
  .ascii, .asciiz, .byte, .double, .float, .long, .short
- BSS (uninitialized variables), use .lcomm
  `.lcomm LABEL, size`
- `.align` can align your variables if needed

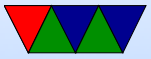


print_string Example

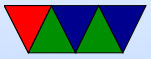
```
# Syscalls
.equ SYSCALL_EXIT,      1
.equ SYSCALL_WRITE,     4

# Other definitions
.equ STDOUT,            1

        .globl _start
_start:
        mov     $hello,%eax
        call    print_string            # Print Hello World
        mov     $mystery,%eax
        call    print_string            # Print Goodbye
        mov     $goodbye,%eax
        call    print_string            # Print Goodbye

exit:
        mov     $5,%ebx
        mov     $SYSCALL_EXIT,%eax      # put exit syscall number (1) in eax
        int     $0x80                   # and exit
```



```
        #====================
        # print string
        #====================
        # Null-terminated string to print pointed to by %eax
        # %eax is trashed by this routine

print_string:
        push    %ebx    # Save registers on stack
        push    %ecx
        push    %edx
        push    %esi

        mov     %eax,%ecx               # String to print is arg3 (%ecx)
        mov     $SYSCALL_WRITE,%eax     # Syscall number goes in %eax
        mov     $STDOUT,%ebx            # Destination fd is arg1 (%ebx)

        # Now we need the length in %edx
        xor     %edx,%edx               # Set %edx to 0
        mov     %ecx,%esi
count_loop:
        cmpb    $0,(%esi)
        je      zero                    # If zero, we are done
        inc     %edx                    # Otherwise increase length count
        inc     %esi                    # increment pointer
```

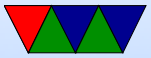


```
        jmp     count_loop              # And continue in loop

zero:   # Found Zero, we are done

        int     $0x80                   # Print the string

        pop     %esi    # Restore registers from stack
        pop     %edx    # In reverse order as we saved, stack is LIFO
        pop     %ecx
        pop     %ebx

        ret                             # Return to where we came from

.data
hello:          .string "Hello World!\n"    # includes null at end
mystery:        .byte 63,0x3f,63,10,0       # mystery string
goodbye:        .string "Goodbye!\n"        # includes null at end
```

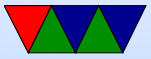


`print_string` Things to Note

- `call` instruction: stores return on stack
- `ret` instruction pulls address off stack and jumps to it
- () used for pointer indirection. CISC type instruction, RISC load/store would have to load it in first.
- 'b' postfix on the cmp instruction
- push/pop save things on stack

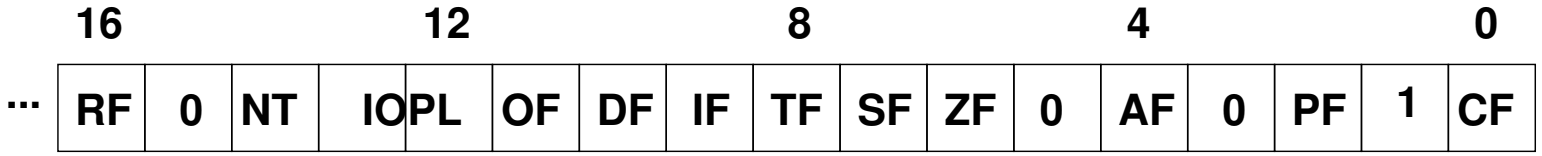


- `xor` – why use that instead of mov to clear register?



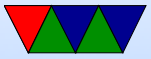
x86 flags

Commonly used ones are starred.



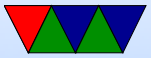
* CF = Carry Flag
PF = Parity Flag
AF = Adjust Flag
* ZF = Zero Flag
* SF = Sign Flag
TF = Trap Flag

IF = Interrupt Enable
DF = Direction Flag
OF = Overflow Flag
IOPL = I/O Priv Level
NT = Nested Task
RF = Resume Flag

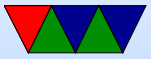


Using the debugger

- gdb ./print_string_x86
- break print_string, break *0xdeadbeef
- run
- info regis
- bt
- disassem

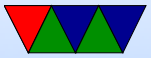


- stepi
- x 0xdeadbeef



x86 History

- 4004, 8008, 8080, 8086/8088, 186, 286
- 386, 486, Pentium (K5, K6)
- Pentium Pro, PII, PIII (Athlon)
- Pentium 4
- Opteron
- Core Duo, Core 2



- Nehalem, Westmere, SandyBridge, Ivy Bridge