ECE 498 – Linux Assembly Language Lecture 3

Vince Weaver

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

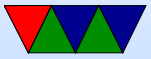
vincent.weaver@maine.edu

20 November 2012

Statically Linked C Hello-World

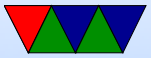
```
Disassembly of section .text:

08048320 <main>:
 8048320:       55                      push   %ebp
 8048321:       89 e5                   mov    %esp,%ebp
 8048323:       83 e4 f0                and    $0xfffffff0,%esp
 8048326:       83 ec 10                sub    $0x10,%esp
 8048329:       c7 04 24 b0 84 04 08    movl   $0x80484b0,(%esp)
 8048330:       e8 bb ff ff ff          call   80482f0 <puts@plt>
 8048335:       b8 05 00 00 00          mov    $0x5,%eax
 804833a:       c9                      leave
 804833b:       c3                      ret
```



Compile Hello World with C Compiler

- run `objdump --disassemble-all ./hello_world` and search all the sections: bss, data, rodata
- look for <main> and see what it does
- look for <puts>
- Why was `printf()` converted into `puts()`?
- Note the setting up of the arguments for the function on the stack.



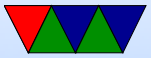
Statically Linked Hello World

- run

  `objdump --disassemble-all ./hello_world.static`

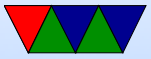
  and repeat

- This time the code for `puts` is included



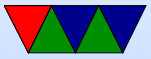
Use C compiler to create assembly

- `gcc -m32 -O2 -S hello_world.c`
- Look at generated hello_world.s



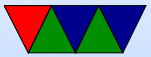
Compiler-generated assembly

```
        .file   "hello_world.c"
        .section        .rodata
.LC0:
        .string "Hello␣World!"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        movq    %rsp, %rbp
        subq    $16, %rsp
        movl    %edi, -4(%rbp)
        movq    %rsi, -16(%rbp)
        movl    $.LC0, %edi
        call    puts
        movl    $5, %eax
        leave
        ret
        .cfi_endproc
```

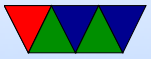


x86 Addressing Modes

- register : `mov %eax, %ebx`
- immediate : `mov $5, %eax`
- direct : `mov 0xdeadbeef,%eax`
- register indirect: `mov (%ebx),%eax`
- base scaled index w displacement:
  `mov 0xdeadbeef(%eax,%ebx,4),%ecx`
  gets value from 0xdeadbeef+(%eax+(%ebx*4))

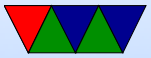


- IP relative (64-bit only):
  `mov 0x8(%rip), %eax`
  Useful for position independent code and keeping local variables nearby.



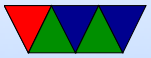
AMD64-bit extensions

- Registers now 64 bit (EAX→RAX, EBX→RBX, etc).
- 8 new general purpose registers, R8 - R15
  Can access low 32, 16, and 8-bits: R8D, R8W, R8L
  Instructions for accessing new 8 registers are encoded with extra REX prefix.
- Can no longer access high-bytes (AH, BH, CH, DL) if using a REX-prefixed (new) instruction but can now access the low bytes of RSI, RDI, RBP and RSP (SIL,



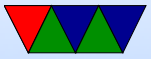
DIL, BPL, SPL)

- Some instructions dropped (`aaa`, single-byte `inc`/`dec`)
- 8 additional XMM registers
- 32-bit loads zero-extend into 64-bit (8 and 16 bit loads ignore top bits)
- RIP addressing – relative to RIP

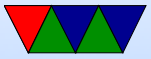


64-bit System Calls

- System Call numbers are all different.
  Done for "performance". Newer linux architectures use common generic syscall numbers.
- System call number in `%rax`
- Arguments in `%rdi` `%rsi` `%rdx` `%r10` `%r8` `%r9`
- `%r11` and `%rcx` are destroyed across syscall
- Return value in `%rax`

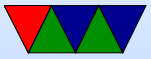


- `syscall` instruction used
- `int $0x80` can still be used to enter 32-bit syscalls



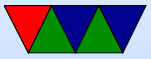
Size of int/long/pointer

- 32-bit Linux - ILP32 (integer/long/pointer all 32-bit)
- 64-bit Linux - LP64 (long and pointer 64)
- 64-bit Window IL32/P64 (only pointer 64-bit)
- new Linux “x32”: ILP32 but can use 64-bit instructions

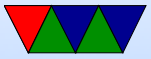


String Instructions

- b/w/l/q postfix (specify size) [note intel Manual uses b/w/d/q]
- auto increment (decrement if D (direction) flag set) after instruction
- `cmps` – compare `(%edi)` with `(%esi)`, increment
- `lods` – load value from `(%esi)` into `%eax`, increment
- `ins`/`outs` – input byte from i/o into `%eax`, increment



- `movs` – move `(%edi)` to `(%esi)`, incrememnt
- `scas` – scan `(%edi)` for `%eax`, increment
- `stos` – store %eax to `(%edi)`, increment
- `rep`/`repe`/`repz`/`repne`/`repnz` prefixes: repeat instruction ECX times

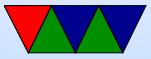


LEA Instruction

- `lea` – load effective address
  Computes the address calculation and stores calculated address into register

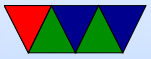
- what does `lea (%ebx,%ebx,4),%ebx` do?

- quick way to multiply %ebx by 5 (much faster than using %mul or discrete shift and add instructions)



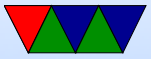
BCD Instructions

- `aaa`, `aad`, `aam`, `aas`, `daa`, `das`
- Adjust BCD results when doing Binary-Coded-Decimal arithmatic



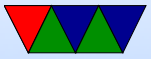
MOV instruction

- `mov` – move a value to or from a register
- `movzx` – move with zero extend
- `xchg` – exchange two registers.



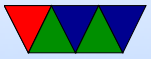
Stack Instructions

- `pop`, `push` – push or pop a register, constant, or memory location onto the stack, then decrement the stack by the appropriate amount
- `pusha`/`popa` (push/pop all)
- `pushf`/`popf` (push/pop flags)



ALU Instructions

- `add`, `adc` – add, add with carry
- `sub`, `sbb` – subtract, subtract with borrow
- `dec`, `inc` – decrement/increment
- `div`, `idiv` – divide AX or DX:AX with resulting Quotient in AL and Remainder in AH (or Quotient in AX and Remainder in DX)
  idiv is signed divide, div unsigned



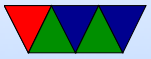
- `mul` – unsigned multiply.
  multiply by AX or DX:AX and put result in DX:AX

- `imul` – signed multiply. Can be like mul, or can also multiply two arbitrary registers, or even a register by a constant and store in a third.

- `cmp` – compare (subtract, but sets flags only, no result stored)

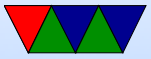
- `neg` – negate (2s complement)

- `nop` – same as `xchg %eax, %eax`.



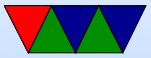
Why does this have to be special cased on 64-bit?
There are also fancier nops of various sizes.

- `cbw`/`cwde`/`cdwq` – sign extend `%eax`
- `cwd`/`cdq`/`cqo` – sign extend `%eax` into `%edx`
  also a quick way to clear `%edx`

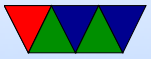


Bit Instructions

- `and` – bitwise and
- `bsf`, `bsr` – bit scan forward or reverse
- `test` – bit test (bitwise and, set flags, don't save result)
- `bt`/`btc`/`btr`/`bts` – bit test with complement/reset/set bit
- `not` – bitwise not

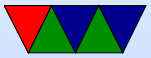


- `or` – bitwise or
- `xor` – bitwise xor. Fast way to clear a register is to xor with self
- `rcl`/`rcr`/`rol`/`ror` – rotate left/right, through carry
- `sal`/`sar`/`shl`/`shr` – shift left/right arithmatic/logical
- `shld`, `shrd` – doubler precision shift

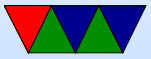


Control Flow

- `call`/`ret` – call by pushing next address on stack, jumping, return
- `call *%ebx` – call to address in register
- `enter` / `leave` – create stack frame
- Jcc – conditional jumps based on flags
  - ja, jna (above / not above)
  - jae, jnae (above equal)



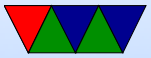
- jb, jnb (below)
- jbe, jnbe (below equal)
- jc, jnc (carry)
- jcxz (cx == 0)
- je, jne (equal)
- jg, jng (greater)
- jge, jnge (greater equal)
- jl, jnl (less)
- jle, jnle (less or equal)
- jo, jno (overflow)
- js, jns (sign)



  - jpe, jpo (parity)
  - jz, jnz (zero)

- `jmp` – unconditional jump

- `loop/loope/loopne` – decrement CX, loop if not 0 (with loope/loopne also check zero flag)

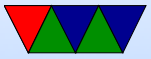


Conditional Moves/Sets

- CMOVcc (all of the postfixes of jmps)
  conditional move lets you do an `if (CONDITION) x=y;` construct without needing any jump instructions, which hurt performance

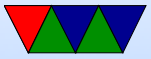
- i.e. `cmovc` = move if carry set

- SETcc – set byte on condition code



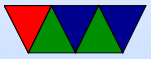
Flags

- `lahf` / `sahf` – load flags into or out of %ah
- `clc`, `cld`, `cmc`, `stc`, `std` – clear, complement or set the various flags



Other Misc

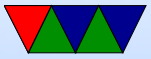
- `bound` – check arrary bounds
- `bswap` – byte swap (switch endian)
- `int` – software interrupt. Also single-step for debug
- `cmpxchg` – compare and exchange, useful for locks
- `cpuid` – get CPU info
- `rdmsr/rdtsc/rdpmc` – read model specific reg,



timestamp, perf counter

- `xadd` – xchange and add, useful for locks. Can use LOCK prefix

- `xlate` – do a table lookup



Summary

The proceeding was just a summary of integer x86 instructions.

There are numerous x86 floating point, SSE, MMX, 3Dnow! and AVX vector instructions, and others such as specific crypto instructions.