SWEN430 - Compiler Engineering (2018)

Lecture 16 - Machine Code I

Lindsay Groves & David J. Pearce

School of Engineering and Computer Science
Victoria University of Wellington

What is ... Machine Code?

- Machine code is the **native language** of a microprocessor
- Each **microprocessor family** has it own machine language
- Machine languages of different families are **not compatible**
- Examples: x86, ARM, PowerPC, Motorola 68K, Z80
- Two main flavours
  - **Reduced Instruction Set Computing (RISC)**: favours simple instructions, but more of them required
  - **Complex Instruction Set Computing (CISC)**: favours fewer, more complex instructions

Machine Code vs Assembly Language

- Machine code is a binary format **directly executed** by microprocessor

- Generally speaking, humans don't read or write **machine code:**

```
0000 0000 0000 0000 6900 696e 2e74 0063
7263 7374 7574 6666 632e 5f00 4a5f 5243
...
```

- Normally, humans read and write **assembly language:**

```
    pushq   %rbp
    movq    %rsp, %rbp
    subq    $16, %rsp
    ...
```

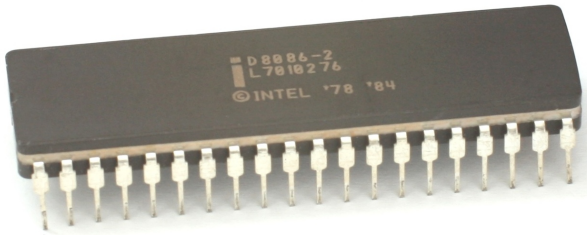
- Assembly language is the **human readable** form of machine code

Running on “Bare Metal”

- JVM provides **safe arena** because of bytecode verification and runtime checks
  - E.g. cannot read a variable before its defined
  - E.g. cannot operate on variable with incorrect type
  - E.g. cannot branch to invalid destination address
  - E.g. cannot access an array out-of-bounds

- Machine code provides **no such guarantees**!
  - If something bad happens, the machine might give a `segmentation fault` **or** it might just carry on
  - E.g. can **always** read from undefined variable and **garbage** is returned
  - E.g. can **always** operate on variable with incorrect type (because there’s no such thing as a type — it’s just a **bit pattern**)
  - E.g. can **sometimes** branch to an invalid address, and machine attempts to execute from there
  - E.g. can **sometimes** access an array out-of-bounds and **garbage** is returned

History of x86 Machine Code



- **1978**: Intel 8086 Microprocessor Released
- **1982**: Intel 80286 Microprocessor Released
- **1985**: Intel 80386 Microprocessor Released (and AMD clone)
- **1989**: Intel 80486 Microprocessor Released (and AMD clone)
- **1993**: Intel Pentium Microprocessor Released (and Cyrix 586)

(Image authored by Konstantin Lanzet, released under creative commons license)

Hello World (x86-64/Linux)

test.s

```
     .data                    /* start of data segment */
str:
     .string "Hello World\n"
     .text                    /* start of text segment */
     .globl main              /* export symbol main */
main:
     pushq   %rbp             /* save contents of rbp */
     movq    %rsp, %rbp       /* assign rsp to rbp */
     subq    $16, %rsp        /* allocate 16 bytes on stack */
     movq    %rdi, -8(%rbp)   /* save rdi into stack */
     movl    $str, %edi       /* assign str address to edi */
     movl    $0, %eax         /* ??? */
     call    printf           /* call printf function */
     leave                    /* restore stack */
     ret                      /* return from function */
```

- This is 64bit x86
- **NOTE:** This is our target architecture!

Hello World (x86-32/NetBSD)

test.s

```
        .data                   /* start of data segment */
str:
        .string "Hello World\n"
        .text                   /* start of text segment */
        .globl main             /* export symbol main */
main:
        pushl   %ebp            /* save contents of ebp */
        movl    %esp, %ebp      /* assign esp to ebp */
        subl    $4, %esp        /* allocate 4 bytes on stack */
        movl    $str, %eax      /* assign str address to eax */
        movl    %eax, (%esp)    /* indirectly assign eax through ebp */
        call    printf          /* call printf function */
        leave                   /* restore stack */
        ret                     /* return from function */
```

- BSD has different calling convention from Linux; this is 32bit x86

Hello World (x86-64/MacOS)

test.s

```
        .data                   /* start of data segment */
str:
        .asciz  "Hello World\n"
        .text                   /* start of text segment */
        .globl  _main           /* export symbol main */
_main:
        pushq %rbp
        movq %rsp, %rbp
        subq $16, %rsp
        movq %rdi, -8(%rbp)
        xorb %al, %al
        leaq str(%rip), %rcx
        movq %rcx, %rdi
        callq _printf
        movl -12(%rbp), %eax
        addq $16, %rsp
        popq %rbp
        ret
```

Portability

- x86 is **not portable** across different Operating Systems!
  - » Most common Operating Systems: *Windows, Linux, MacOS, BSD*
  - » x86-32/Linux code **won't necessarily** run on x86-32/MacOS
  - » Because e.g. OS **calling conventions** differ

- x86 is **backward compatible** across architectures!
  - » e.g. x86-32/Linux code probably **will run** on x86-64/Linux
  - » But, x86-64/Linux code probably **won't run** on x86-32/Linux

(Yes, this makes generating x86 code difficult)

Running Hello World

```
% gcc -o test test.s
% ./test
Hello World
%
```

- GCC can compile our **assembly language** programs!
- We can then execute them **directly on the machine**

Assembly Language from C

test.c

```
#include <stdio.h>
int main(char** args) { printf("Hello World"); }
```

```
% gcc -S test.c
% cat test.s
.file   "test.c"
        .section        .rodata

        ...

.globl main
        .type   main, @function
main:
        pushl   %ebp
        movl    %esp, %ebp
        ...
```

- GCC can also compile **C programs** to assembly language!

Debugging with GDB

```
There is absolutely no warranty for GDB.  Type
"show warranty" for details.  This GDB was
configured as "x86_64-apple-darwin"...Reading
symbols for shared libraries .. done

(gdb) r
Starting program: /Users/djp/test
Reading symbols for shared libraries done

Program received signal EXC_BAD_ACCESS, Could
not access memory.
Reason: 13 at address: 0x0000000000000000
0x00007fff888186cd in
misaligned_stack_error_entering_dyld_stub_binder ()
(gdb)
```

- The **GNU Debugger** is an important tool for debugging machine code — you will probably need to use it!!

Some x86 Instructions

`movl $c, %eax`	Assign constant `c` to `eax` register	eax = c
`movl %eax, %edi`	Assign register `eax` to `edi` register	edi = eax
`addl $c, %eax`	Add constant `c` to `eax` register	eax += c
`addl %eax, %ebx`	Add `eax` register to `ebx` register	ebx += eax
`subl $c, %eax`	Substract constant `c` from `eax` register	eax -= c
`subl %eax, %ebx`	Subtract `eax` register from `ebx` register	ebx -= eax
`cmpl $0, %edx`	Compare constant `0` register against `edx` register	
`cmpl %eax, %edx`	Compare `eax` register against `edx` register	

- General form: **Instr** src, dst
- Similar range of instructions as found in JVM Bytecode
- However, x86 is a **register-based** machine code

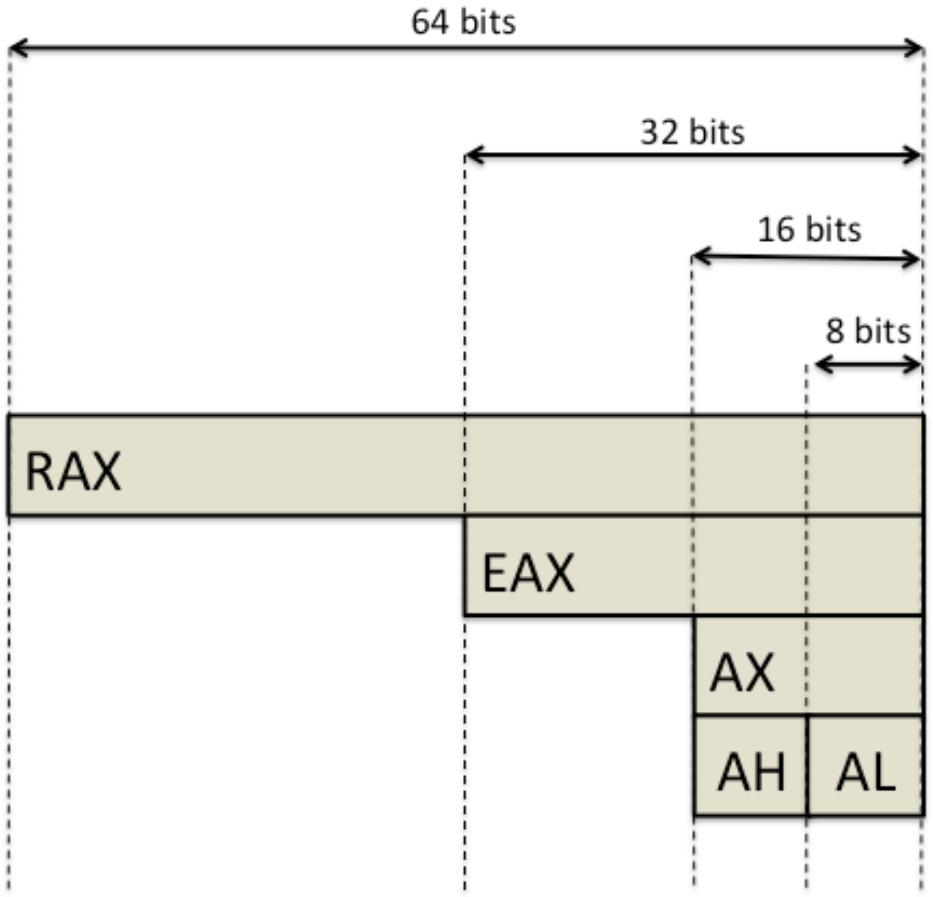
Instruction Suffixes

- GNU Assembler uses **AT&T** instruction format
- AT&T format uses **instruction suffixes**:

```
main:
      pushq   %rbp
      movq    %rsp, %rbp
      subq    $16, %rsp
      movq    %rdi, -8(%rbp)
      movl    $str, %edi
      ...
```

- Where:
  - `q` indicates quad word (8 bytes)
  - `l` indicates long (a.k.a. double) word (4 bytes)
  - `w` indicates word (2 bytes)
  - `b` indicates byte (1 byte)

Understanding x86 Registers



- Registers on x86 are unusual because they **overlap**
  - » e.g. `rax` overlaps with `eax`, which overlaps with `ax`, etc.
  - » Therefore, assigning to e.g. `ax` affects `eax` and `rax`, etc.

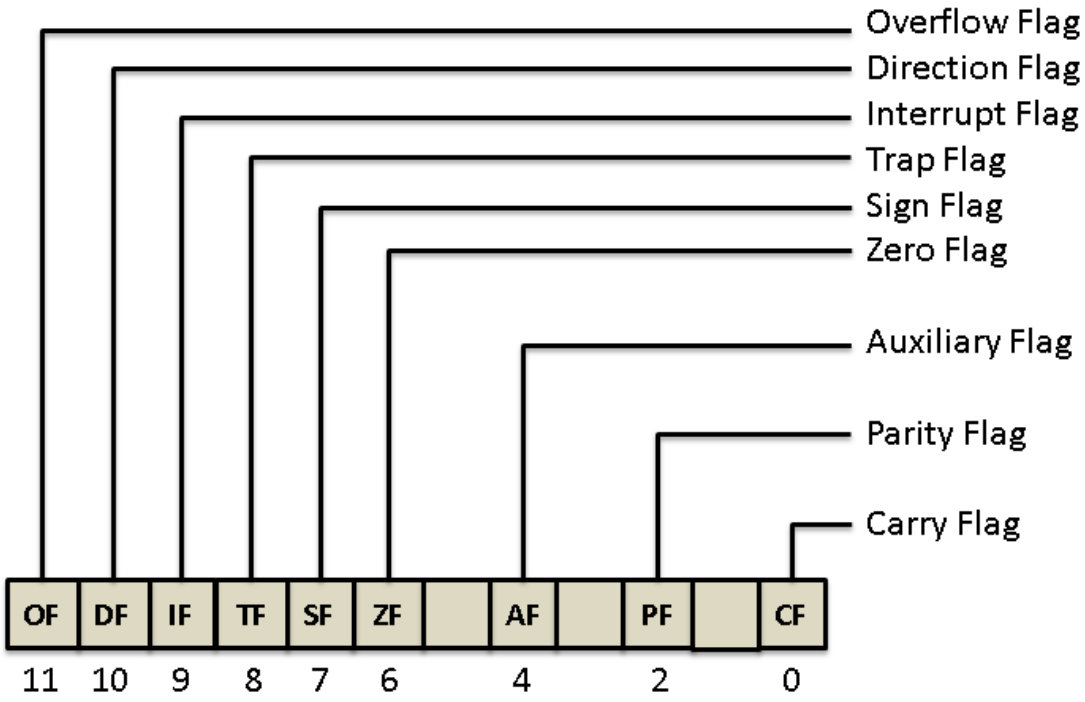
Overview of x86 Registers

64bits	32bits	16bits	8bits	Comments
`rax`	`eax`	`ax`	`al, ah`	General purpose. The “accumulator”
`rbx`	`ebx`	`bx`	`bl, bh`	General purpose.
`rcx`	`ecx`	`cx`	`cl, ch`	General purpose.
`rdx`	`edx`	`dx`	`dl, dh`	General purpose.
`rsi`	`esi`	`si`	-	Index register.
`rdi`	`edi`	`di`	-	Index register.
`rbp`	`ebp`	`bp`	-	Index register. Normally holds “Frame Pointer”
`rsp`	`esp`	`sp`	-	Index register. Normally holds “Stack Pointer”
-	-	`cs`	-	Segment register. Identifies “Code Segment”
-	-	`ds`	-	Segment register. Identifies “Data Segment”
-	-	`ss`	-	Segment register. Identifies “Stack Segment”

- These are the main registers, although there are others (e.g. for FPU, MMX, R8-15, etc)
- x86 architecture notable for having **very few** general purpose registers

Flags Register

- The `EFLAGS` register holds "processor state":



- Used (amongst other things) to implement **conditional branching**
- **Note:** there are more flags than shown here

Conditional Branching

- Conditional branch (equality) implemented as follows:

```
cmpl %eax,%ebx       /* compare eax against ebx */
jz target           /* branch if zero flag set */
```

- Conditional branch (less than or equal) implemented as follows:

```
cmpl %eax,%ebx       /* compare eax against ebx */
jle target          /* branch if sign or zero flags set */
```

- Conditional branch (not equals) implemented as follows:

```
cmpl %eax,%ebx       /* compare eax against ebx */
jnz target          /* branch if zero flag not set */
```

- Notes:
  - » **Zero Flag** set after comparison if items equal
  - » **Sign Flag** set after comparison if left operand less than right

Addressing Modes

`movl %eax, (%ebx)`	Assign `eax` register to dword at address `ebx`	`*ebx = eax`
`movl (%ebx),%eax`	Assign `eax` register from dword at address `ebx`	`eax = *ebx`
`movl 4(%esp),%eax`	Assign `eax` register from dword at address `esp+4`	`eax = *(esp+4)`
`movl (%esi,%eax),%cl`	Assign `cl` register from byte at address `esi+eax`	`cl = *(esi+eax)`
`movl %edx, (%esi,%ebx,4)`	Assign `edx` register to dword at address `esi+4*ebx`	`*(esi+4*ebs) = edx`

- Access the value at an address by a(%r1,%r2,b) $\rightarrow$ %r1 + a + b * %r2
- 64bit x86-compatible processors can access 2^{64} **bytes** of memory
- Can read or write memory **indirectly** using address stored in register
- Corresponds to reading / writing through **pointers** in C

Understanding the Stack

`pushq %rax`	Push `rax` register onto stack
`pushq %c`	Push constant `c` onto stack
`popq %rdi`	pop qword off stack and assign to register `rdi`

- Stack provided for additional **temporary storage**:

```
movq $0xFF, %rax     /* store 255 in rax */
pushq %rax           /* push contents of rax on stack */
pushq $0xEE          /* push 238 directly on stack */
movq 8(%rsp),%rax    /* assign 255 to rax */
popq %rdx            /* pop 238 and assign to rdx */
```

- Stack grows **downwards**!
- Stack used primarily for **local variables**, and **return address**

Visualising the Stack

- Consider **executing** these instructions:

```
movq $0xFF, %rax     /* store 255 in rax */
pushq %rax           /* push contents of rax on stack */
pushq $0xEE          /* push 238 directly on stack */
movq 8(%rsp),%rax    /* assign 255 to rax */
popq %rdx            /* pop 238 and assign to rdx */
```

- The effect on the stack can be **visualised** like so: