SWEN430 - Compiler Engineering

Lecture 16 - Machine Code II

David J. Pearce & Alex Potanin & Roma Klapaukh

School of Engineering and Computer Science
Victoria University of Wellington

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

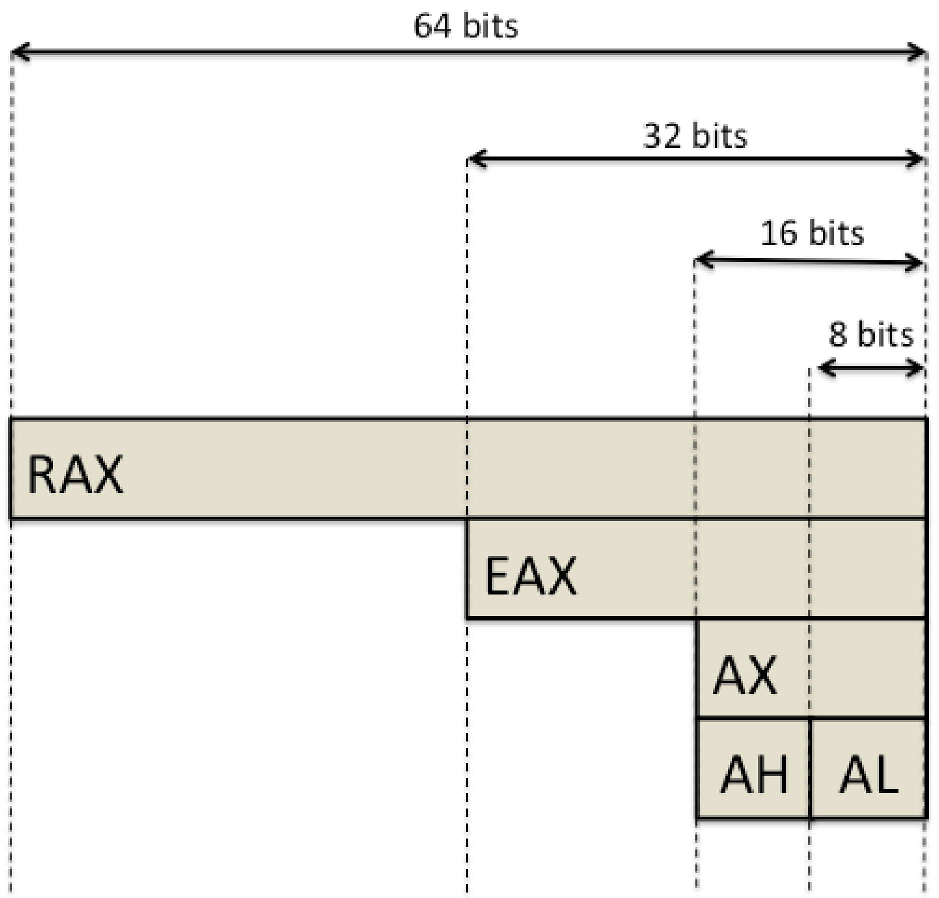
Instruction Suffixes

- 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.

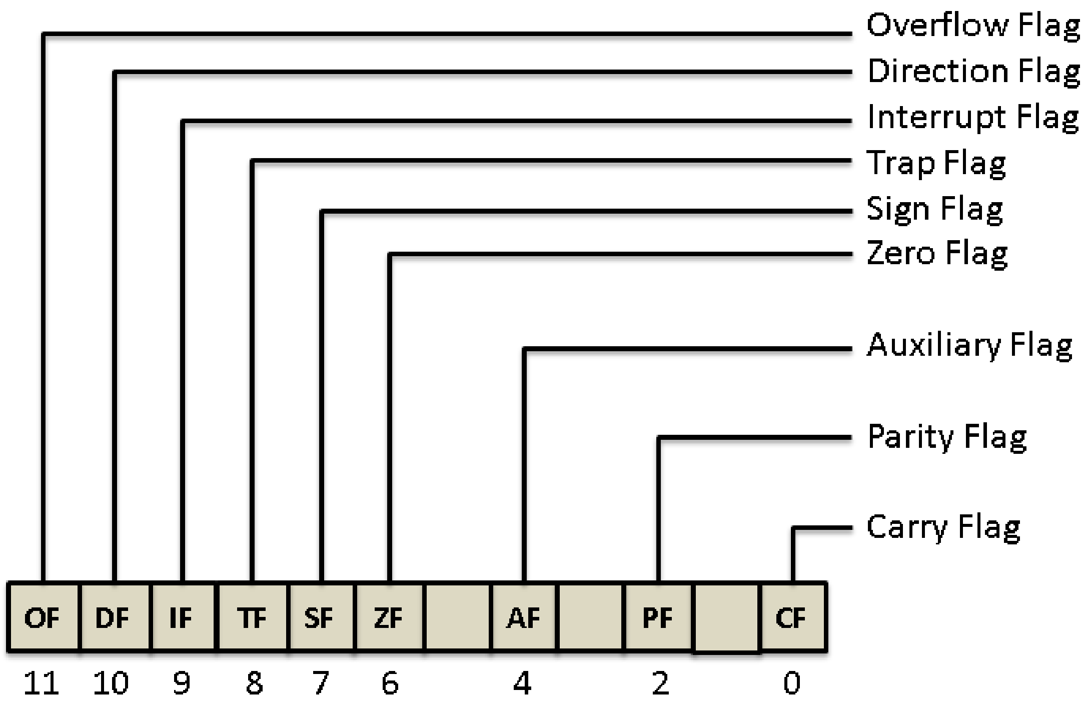
Overview of x86 Registers

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) $\longrightarrow$ %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 */
```

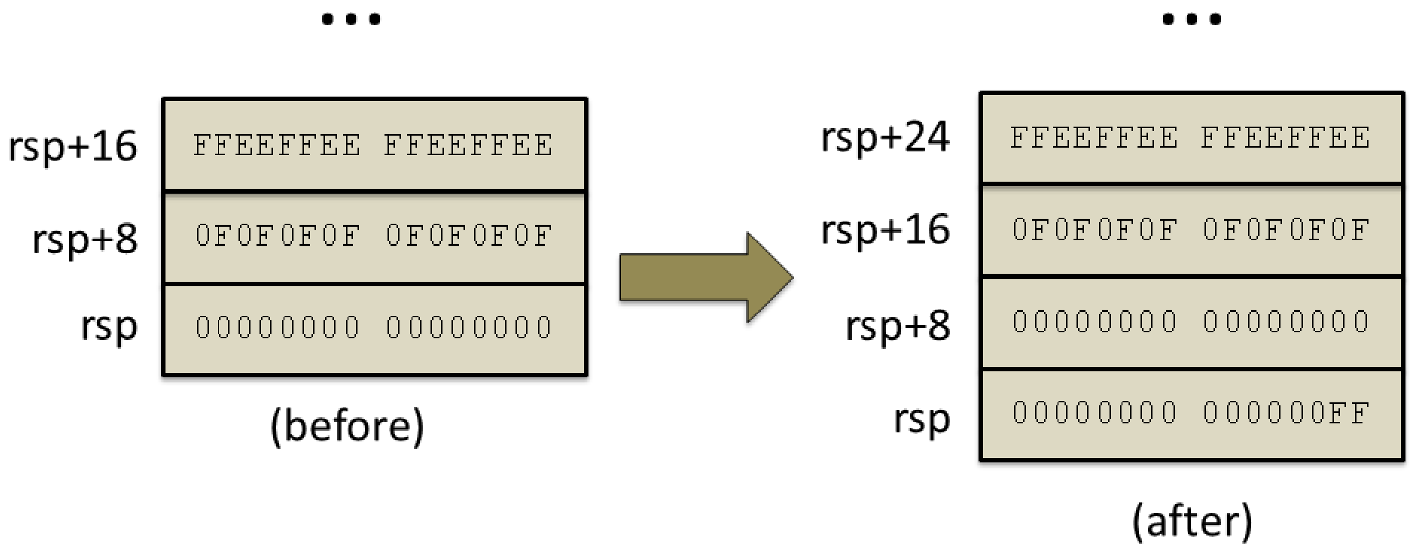
- Stack grows **downwards**!
- Stack used primarily for **local variables**, and **return address**

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:



...
rsp+16
FFEEFFEE FFEEFFEE
rsp+8
0F0F0F0F 0F0F0F0F
rsp
00000000 00000000
(before)
...
rsp+24
FFEEFFEE FFEEFFEE
rsp+16
0F0F0F0F 0F0F0F0F
rsp+8
00000000 00000000
rsp
00000000 000000FF
(after)