## Review addressing modes

<table>
<thead>
<tr>
<th>Op</th>
<th>Src</th>
<th>Dst</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>movl</td>
<td>$0, %rax</td>
<td></td>
<td>Register</td>
</tr>
<tr>
<td>movl</td>
<td>$0, 0x605428</td>
<td></td>
<td>Direct address</td>
</tr>
<tr>
<td>movl</td>
<td>$0, (%rcx)</td>
<td></td>
<td>Indirect address</td>
</tr>
<tr>
<td>movl</td>
<td>$0, 20(%rsp)</td>
<td></td>
<td>Indirect with displacement</td>
</tr>
<tr>
<td>movl</td>
<td>$0, -8(%rdi, %rax, 4)</td>
<td></td>
<td>Indirect with scaled-index</td>
</tr>
</tbody>
</table>

### lea

```
lea -8(%rdi, %rax, 4), %rax
```

"Load effective address" — compute target address and stop (no access memory)

**Used for:**

- Pointer math, address of, e.g. `p = &arr[i];`
- Simple linear equations, e.g. `dst = x + k*y`  \(k = 1, 2, 4, \text{ or } 8\)
What are registers?

Small set of named "data cubbies" on CPU itself

- CPU can directly manipulate values in register (reaching out to memory is much slower)
- 16 general-purpose integer registers

Each register stores a 64-bit data value

- Anything in integer family (long, int, char, address, signed/unsigned)
- (floating point registers are separate)
- Virtual sub-registers %rax -> %eax -> %ax -> %al

Some registers have special role

- Dictated by ISA/convention
- If/when not in use for special role, register may be used for other purposes

int binky(int arg)
Function binky is going to read arg from %edi and write return value to %eax

<table>
<thead>
<tr>
<th>Register</th>
<th>Special role</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>Return value from function</td>
</tr>
<tr>
<td>%rdi</td>
<td>1st argument to function</td>
</tr>
<tr>
<td>%rsi</td>
<td>2nd argument to function</td>
</tr>
<tr>
<td>%rdx</td>
<td>3rd argument to function</td>
</tr>
<tr>
<td>%rsp</td>
<td>Stack pointer</td>
</tr>
<tr>
<td>%rip</td>
<td>Instruction pointer</td>
</tr>
<tr>
<td>%eflags</td>
<td>Processor status/condition cod</td>
</tr>
<tr>
<td>...</td>
<td>see full list in x86 guide on web site</td>
</tr>
</tbody>
</table>
### Sample ALU instructions

#### Two operand instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Source, Destination</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>src, dst</td>
<td>dst = dst + src</td>
</tr>
<tr>
<td>sub</td>
<td>src, dst</td>
<td>dst = dst - src</td>
</tr>
<tr>
<td>imul</td>
<td>src, dst</td>
<td>dst = dst * src</td>
</tr>
<tr>
<td>and</td>
<td>src, dst</td>
<td>dst = dst &amp; src</td>
</tr>
<tr>
<td>xor</td>
<td>src, dst</td>
<td>dst = dst ^ src</td>
</tr>
<tr>
<td>sal</td>
<td>count, dst</td>
<td>dst = dst &lt;&lt; count</td>
</tr>
<tr>
<td>sar</td>
<td>count, dst</td>
<td>dst = dst &gt;&gt; count</td>
</tr>
</tbody>
</table>

#### One operand instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Destination</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>neg</td>
<td>dst</td>
<td>dst = -dst</td>
</tr>
<tr>
<td>not</td>
<td>dst</td>
<td>dst = ~dst</td>
</tr>
</tbody>
</table>

No distinction between signed/unsigned operands — why?
Fun tool to interactively examine C->asm translation!  

https://godbolt.org
What does it mean for a program to execute?

Instructions loaded into memory
Stack configured (more on that later…)
%rip stores address of current instruction, proceeds sequentially
program code entered at main function

What does it mean for a program to execute?

Instructions loaded into memory
Stack configured (more on that later…)
%rip stores address of current instruction, proceeds sequentially
program code entered at main function

00000000004004d6 <loop>:

4004d6: 8b 07                mov    (%rdi),%eax
4004d8: 83 c0 01            add    $0x1,%eax
4004db: 89 07               mov    %eax,(%rdi)
4004dd: eb f7               jmp    4004d6 <loop>

jmp is akin to mov <target>, %rip
(not valid to directly access %rip in this way though…)
Information about currently executing program

Temporary data
%rax, %rdi, ...
current parameters, local variables

Location of runtime stack
%rsp

Location of current instruction
%rip

Status of recent operation
CF ZF SF OF

Processor state

General purpose registers
%rax
%rdi
...
%r14
%r15

Stack pointer
%rsp

Instruction pointer
%rip

Condition codes
%eflags

CF ZF SF OF
Control flow

Controlling flow
Instructions proceed sequentially by default
Jmp instruction changes %rip (unconditionally)
if/loops/switch need a conditional jmp — "branch"

Branch is 2-step process
1. Previous instruction **writes** condition codes
   Codes report whether operation resulted in zero, overflow, etc
2. Branch instruction **reads** condition codes
   Whether takes branch or falls through depends on state of condition codes

Test result is "passed" through %eflags register
## Condition codes

<table>
<thead>
<tr>
<th>Op</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>cmp</strong></td>
<td>op1, op2</td>
</tr>
<tr>
<td><strong>test</strong></td>
<td>op1, op2</td>
</tr>
<tr>
<td><strong>sub</strong></td>
<td>op1, op2</td>
</tr>
<tr>
<td><strong>add</strong></td>
<td>op1, op2</td>
</tr>
</tbody>
</table>

*eflags register used as set of boolean values*

- **ZF** = zero flag
- **SF** = sign flag
- **CF** = carry flag, unsigned overflow (out of MSB)
- **OF** = overflow flag, signed overflow (into MSB)

*Codes explicitly set by cmp/test, implicitly set by many instructions*

*Codes read by jx instructions*
### Example branch instructions

<table>
<thead>
<tr>
<th>Op</th>
<th>Description</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>jmp</td>
<td>unconditional</td>
<td></td>
</tr>
<tr>
<td>je</td>
<td>equal/zero</td>
<td>ZF=1</td>
</tr>
<tr>
<td>jne</td>
<td>not equal/not zero</td>
<td>ZF=0</td>
</tr>
<tr>
<td>js</td>
<td>negative</td>
<td>SF=1</td>
</tr>
<tr>
<td>jl</td>
<td>less (signed)</td>
<td>SF! =OF</td>
</tr>
<tr>
<td>jle</td>
<td>less or equal (signed)</td>
<td>SF! =OF or ZF=1</td>
</tr>
<tr>
<td>jb</td>
<td>below (unsigned)</td>
<td>CF=1</td>
</tr>
</tbody>
</table>

#### Examples:

Assume previous instruction was `cmp op1, op2`. Computed "result" `op2-op1`

**je:** Jump if ZF is 1
- result `op2-op1` is zero means `op1` is **equal** to `op2`

**jl:** Jump if SF != 0F
- result `op2-op1` is negative means `op2` is **less** than `op1`
- other case: if result ended up positive due to overflow, `op2` is also less than `op1`
int if_then(int arg)
{
    if (arg == 6)
        arg++;  
    arg *= 35;
    return arg + 7;
}

Consider:
How does assembly change if test on line 1 is:  arg == 9?  arg <= 6?
What if put line 3 inside else clause?
(test ? expr : expr)
Can be implemented by similar if/else assembly sequence
Loops

```c
int for_loop(int n)
{
    int sum = 0;
    for (int i = 0; i < n; i++)
        sum += i;
    return sum;
}
```

400504: mov $0x0,%edx
400509: mov $0x0,%eax
40050e: jmp 400515 <for_loop+0x11>
400510: add %edx,%eax
400512: add $0x1,%edx
400515: cmp %edi,%edx
400517: jl 400510 <for_loop+0xc>
400519: repz retq

Translation re-arranged from what you might expect
First iteration jumps over body to get to test/branch — why?
Assembly tools

- **Objdump**
  Extracts code from compiled executable, displays instructions in assembly
  `myth51> objdump -d myprogram`

- **Gdb**
  Can debug at source or assembly level!
  Single step by instruction, read/write register values
  `(gdb) disassemble main`
  `(gdb) info reg`
  `(gdb) layout split`

- **Compiler explorer**
  [https://godbolt.org/g/NYuQKY](https://godbolt.org/g/NYuQKY)

- **References on web site**
  [http://cs107.stanford.edu/guide/x86-64.html](http://cs107.stanford.edu/guide/x86-64.html)
  [http://cs107.stanford.edu/resources/onepage_x86-64.pdf](http://cs107.stanford.edu/resources/onepage_x86-64.pdf)