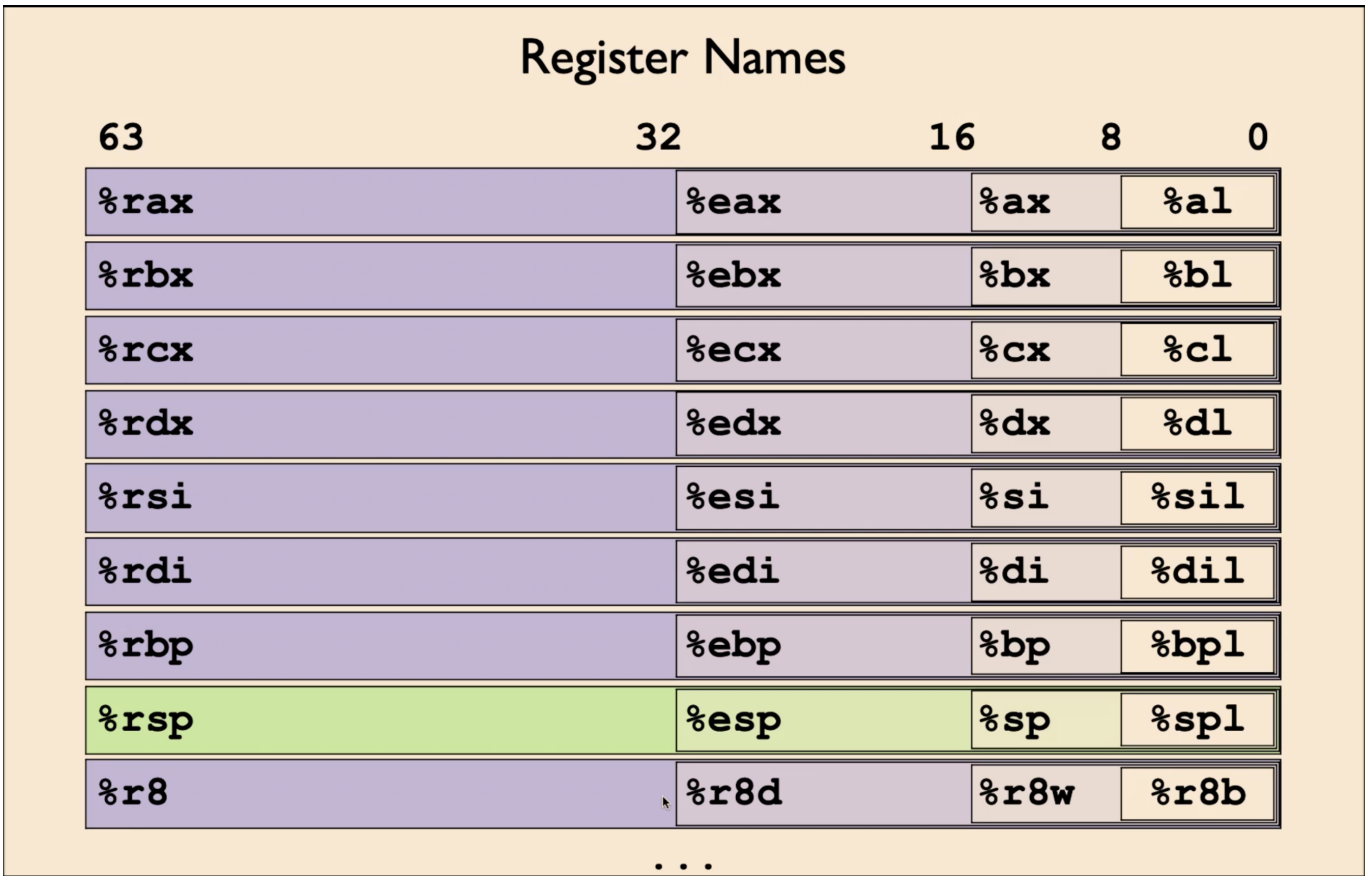
NOTES: Assembly

Assembly

Registers

- Registers live in the CPU and are faster access than memory
- There are 16 total registers to use
    - From #8 and on, they're named %r8, %r9, etc



- For backwards compatibility, there are 32, 16, and 8-bit registers. These are simply the last X bits of the main 64-bit registers

Syntax

- `$` means a number literal.
- `(%x)` means the memory value of the address stored in register x.
- A number without a $ means the value of the memory at the specified address literal.
- Indexing takes two arguments.
    - The memory address stored in the first register is like the pointer to the first element of an array

- The value in the second register is like the index

`%eax`	register	R[%eax]
`$0x2a3`	literal	0x2a3
`0x2a3`	absolute	M[0x2a3]
`(%eax)`	indirect	M[R[%eax]]
`7(%edx)`	base + displacement	M[7 + R[%edx]]
`(%eax,%ecx)`	indexed	M[R[%eax] + R[%ecx]]
`7(%eax,%ecx)`	indexed	M[7 + R[%eax] + R[%ecx]]
`(,%eax,4)`	scaled indexed	M[R[%eax] × 4]
`7(,%eax,4)`	scaled indexed	M[7 + R[%eax] × 4]
`(%eax,%ecx,4)`	scaled indexed	M[R[%eax] + R[%ecx] × 4]
`7(%eax,%ecx,4)`	scaled indexed	M[7 + R[%eax] + R[%ecx] × 4]

Instructions

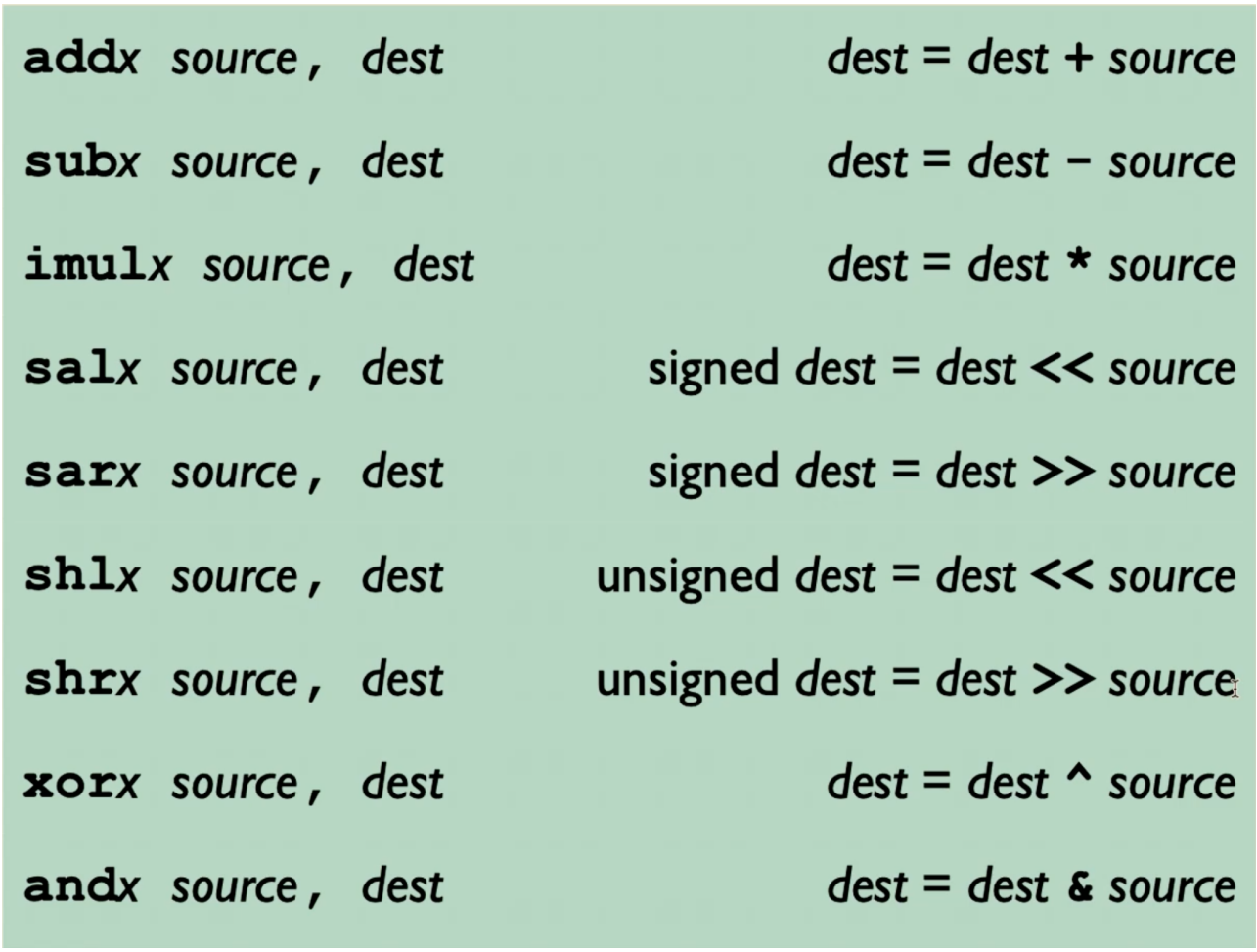
- `push#` increments `%rsp` (the stack pointer), then copies into the space where it points. `pop#` does a similar thing which you can infer.
  - The `#` indicates the number of bytes, i.e. `push1` pushes four bytes.

> Example: C Code
>
> ```
> int exchange(int *xp, int y) {
>     int x = *xp;
>     *xp = y;
>     return x;
> }
> ```
>
> ```
> movq -0x18(%rbp),%rax     ; put *xp arg into %rax
> mov1 -0x1c(%rbp),%edx     ; put y arg into %edx
> mov1 (%rax),%ecx          ; get the value of whatever %rax (*xp)
> points to
> mov1 %edx,(%rax)          ; put y into *xp
> mov1 %ecx,%eax            ; move to %eax, the return value register
> ```
>
> The first argument is the top

Arithmetic

- Arithmetic instructions always change one argument. So, if you want to store the result in a new variable, you must do a copy.

- For example, `sub $7, %rax` does `rax = rax - 7`.



Comparisons

Flags

- CF (carry flag): carry out of most-significant bit
- ZF (zero): produced zero
- SF (sign): produced negative
- OF (overflow): two's complement overflow

Comparison Instructions

- You can do a subtraction command, then check the zero flag, to see if two registers are equal.
- Or, you can use `cmpx`. It's the same thing as subtract, but it does not reassign the value of the first argument.
- After you compare, you can use `setg`, `setl`, etc to set if the result is greater, lesser, etc.
- The value calculated for `cmp a b` is `b - a`.

> Note: You can do something similar to ternary expressions with conditional moves.

```
long x = ((a < b) ? 17 : 42);
```

```
movql $42, %rax # Guess 42
cmpq %rsi, %rdi # Compare a to b
cmovlq $17, %rax # Maybe correct guess
```

- You can do conditional jump commands to make control flow.

je	equal / zero	**ZF** a.k.a. **jz**
jne	not equal / not zero	**~ZF** a.k.a. **jnz**
js	negative	**SF**
jns	non-negative	**~SF**
jg	greater **signed**	**~(SF^OF) &~ZF**
jge	greater or equal **signed**	**~(SF^OF)**
jl	less **signed**	**(SF^OF)**
jle	less or equal **signed**	**(SF^OF)|ZF**
ja	above **unsigned**	**~CF&~ZF**
jb	below **unsigned**	**CF**

`lea`

- `leax` computes the address of an expression, and puts it in a register.
- `movx`, on the other hand, actually writes to the computed address.

```
leaq 4(%rbx, %rsp), %rax
```

Calling Procedures

- `callx` source
  - Pushes the *next* value of `%rip` (program counter)
  - Jumps to `source` argument—sets `%rip` to `source`
- `retx`
  - Returns out of the function
  - Pops off the stack, and sets `%rip` to its value
  - Return is stored in `%rax`
- Arguments
  - 1st argument in `%rdi`

    - 2nd argument in `%rsi`
    - 3rd argument in `%rdx`
    - 4th argument in `%rcx`
    - 5th argument in `%r8`
    - 6th argument in `%r9`
    - 7th argument and later pusehd onto the stack
        - Pushing is in reverse; `%rsp` points to 7th argument; `%rsp - 1` points to 8th argument, etc

Memory Calls

- Each instruction executed automatically causes memory to be read, since the program is stored in memory
- `mov` can contribute additional read/writes, depending on if it's moving to/from a register or a memory address
- `add` can contribute 1, 2, or 3 total, since the variable being modified will have to be read and then written
- `ret` contributes 2 because it needs to get the return address from the stack
- `push` and `pop` are two memory calls