Data Sizes

Data sizes in assembly have slightly different terminology to get used to:

- A **byte** is 1 byte.
- A **word** is 2 bytes.
- A **double word** is 4 bytes.
- A **quad word** is 8 bytes.

Assembly instructions can have suffixes to refer to these sizes:

- `b` means **byte**
- `w` means **word**
- `l` means **double word**
- `q` means **quad word**

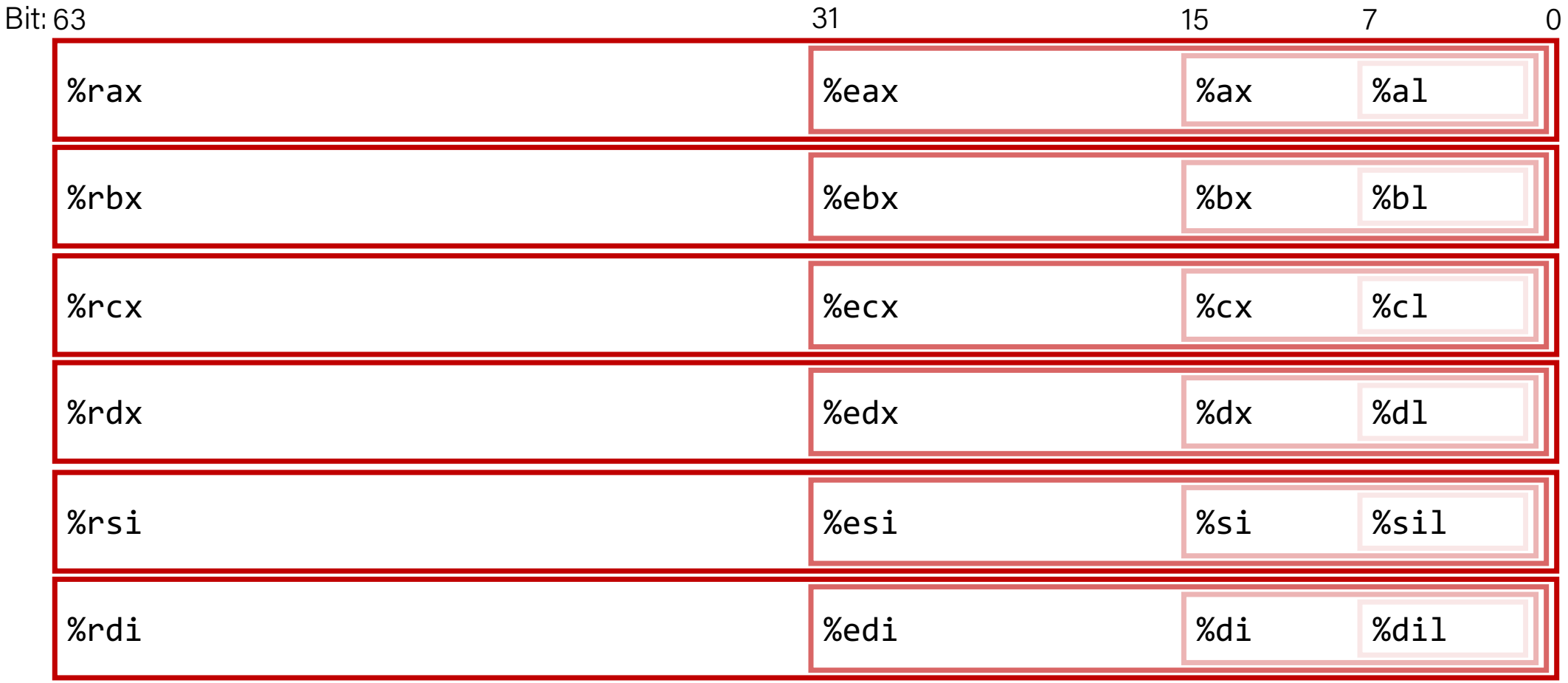
Data Sizes

Data sizes in assembly have slightly different terminology to get used to:

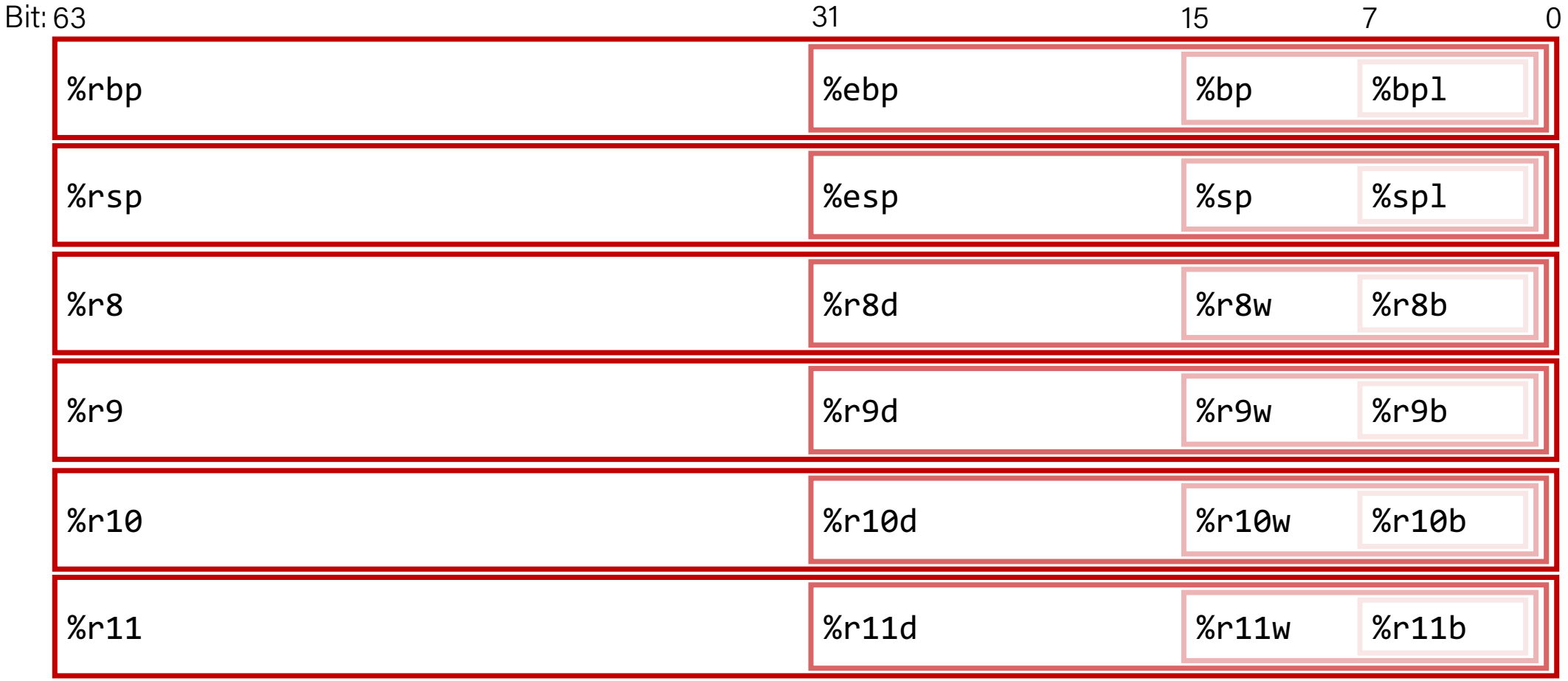
- A **byte** is 1 byte.
- A **word** is 2 bytes.
- A **double word** is 4 bytes.
- A **quad word** is 8 bytes.

C Type	Suffix	Byte	Intel Data Type
`char`	`b`	1	Byte
`short`	`w`	2	Word
`int`	`l`	4	Double word
`long`	`q`	8	Quad word
`char *`	`q`	8	Quad word
`float`	`s`	4	Single precision
`double`	`l`	8	Double precision

Register Sizes



Register Sizes



Register Sizes

Register Responsibilities

Some registers take on special responsibilities during program execution.

- **`%rax`** stores the return value
- **`%rdi`** stores the first parameter to a function
- **`%rsi`** stores the second parameter to a function
- **`%rdx`** stores the third parameter to a function
- **`%rip`** stores the address of the next instruction to execute
- **`%rsp`** stores the address of the current top of the stack

See **Stanford CS107 x86-64 Reference Sheet** on Resources page of the course website!
https://aykuterdem.github.io/classes/comp201/index.html#div_resources

Practice #1: `mov` And Data Sizes

For each of the following `mov` instructions, determine the appropriate suffix based on the operands (e.g. `movb`, `movw`, `movl` or `movq`).

```
1. mov__ %eax, (%rsp)
2. mov__ (%rax), %dx
3. mov__ $0xff, %bl
4. mov__ (%rsp,%rdx,4),%dl
5. mov__ (%rdx), %rax
6. mov__ %dx, (%rax)
```

```
movl %eax, (%rsp)
movw (%rax), %dx
movb $0xff, %bl
movb (%rsp,%rdx,4),%dl
movq (%rdx), %rax
movw %dx, (%rax)
```

mov

- The **movabsq** instruction is used to write a 64-bit Immediate (constant) value.
- The regular **movq** instruction can only take 32-bit immediates.
- 64-bit immediate as source, only register as destination.

```
movabsq $0x0011223344556677, %rax
```

Practice #2: `mov` And Data Sizes

For each of the following **`mov`** instructions, determine how data movement instructions modify the upper bytes of a destination register.

1. `movabs $0x0011223344556677, %rax`	`%rax = 0011223344556677`
2. `movb $-1, %al`	`%rax = 0011223344556FF`
3. `movw $-1, %ax`	`%rax = 0011223344FFFF`
4. `movl $-1, %eax`	`%rax = 00000000FFFFFFFF`
5. `movq $-1, %rax`	`%rax = FFFFFFFFFFFFFFFF`

`movz` and `movs`

- There are two **`mov`** instructions that can be used to copy a smaller source to a larger destination: **`movz`** and **`movs`**.
- **`movz`** fills the remaining bytes with zeros
- **`movs`** fills the remaining bytes by sign-extending the most significant bit in the source.
- The source must be from memory or a register, and the destination is a register.

movz and movs

```
MOVZ S,R          R ← ZeroExtend(S)
```

Instruction	Description
movzbw	Move zero-extended byte to word
movzbl	Move zero-extended byte to double word
movzwl	Move zero-extended word to double word
movzbq	Move zero-extended byte to quad word
movzwq	Move zero-extended word to quad word

movz and movs

MOVS S,R R ← SignExtend(S)

Instruction	Description
movsbw	Move sign-extended byte to word
movsbl	Move sign-extended byte to double word
movswl	Move sign-extended word to double word
movsbq	Move sign-extended byte to quad word
movswq	Move sign-extended word to quad word
movslq	Move sign-extended double word to quad word
cltq	Sign-extend %eax to %rax %rax ← SignExtend(%eax)

`lea`

The **`lea`** instruction copies an "effective address" from one place to another.

```
lea     src,dst
```

Unlike **`mov`**, which copies data at the address src to the destination, **`lea`** copies the value of src *itself* to the destination.

The syntax for the destinations is the same as **`mov`**. The difference is how it handles the **`src`**.

`lea` vs. `mov`

Operands	mov Interpretation	lea Interpretation
`6(%rax), %rdx`	Go to the address (6 + what's in `%rax`), and copy data there into `%rdx`	Copy 6 + what's in `%rax` into `%rdx`.
`(%rax, %rcx), %rdx`	Go to the address (what's in `%rax` + what's in `%rcx`) and copy data there into `%rdx`	Copy (what's in `%rax` + what's in `%rcx`) into `%rdx`.
`(%rax, %rcx, 4), %rdx`	Go to the address (`%rax` + 4 * `%rcx`) and copy data there into `%rdx`.	Copy (`%rax` + 4 * `%rcx`) into `%rdx`.
`7(%rax, %rax, 8), %rdx`	Go to the address (7 + `%rax` + 8 * `%rax`) and copy data there into `%rdx`.	Copy (7 + `%rax` + 8 * `%rax`) into `%rdx`.

Unlike **`mov`**, which copies data at the address src to the destination, **`lea`** copies the value of src itself to the destination.

Unary Instructions

The following instructions operate on a single operand (register or memory):

Instruction	Effect	Description
`inc D`	`D ← D + 1`	Increment
`dec D`	`D ← D - 1`	Decrement
`neg D`	`D ← -D`	Negate
`not D`	`D ← ~D`	Complement

Examples:

```
incq 16(%rax)
dec %rdx
not %rcx
```

Binary Instructions

The following instructions operate on two operands (both can be register or memory, source can also be immediate). Both cannot be memory locations. Read it as, e.g. "Subtract S from D":

Instruction	Effect	Description
`add S, D`	`D ← D + S`	Add
`sub S, D`	`D ← D - S`	Subtract
`imul S, D`	`D ← D * S`	Multiply
`xor S, D`	`D ← D ^ S`	Exclusive-or
`or S, D`	`D ← D | S`	Or
`and S, D`	`D ← D & S`	And

Examples:

```
addq %rcx,(%rax)
xorq $16,(%rax, %rdx, 8)
subq %rdx,8(%rax)
```

Large Multiplication

- Multiplying 64-bit numbers can produce a 128-bit result. How does x86-64 support this with only 64-bit registers?
- If you specify two operands to **`imul`**, it multiplies them together and truncates until it fits in a 64-bit register.

```
imul S, D      D ← D * S
```

- If you specify one operand, it multiplies that by **`%rax`**, and splits the product across **2** registers. It puts the high-order 64 bits in **`%rdx`** and the low-order 64 bits in **`%rax`**.

Instruction	Effect	Description
`imulq S`	`R[%rdx]:R[%rax] ← S x R[%rax]`	Signed full multiply
`mulq S`	`R[%rdx]:R[%rax] ← S x R[%rax]`	Unsigned full multiply

Division and Remainder

Instruction	Effect	Description
`idivq S`	`R[%rdx] ← R[%rdx]:R[%rax] mod S;` `R[%rax] ← R[%rdx]:R[%rax] ÷ S`	Signed divide
`divq S`	`R[%rdx] ← R[%rdx]:R[%rax] mod S;` `R[%rax] ← R[%rdx]:R[%rax] ÷ S`	Unsigned divide

- Terminology: **dividend / divisor = quotient + remainder**
- **x86-64** supports dividing up to a 128-bit value by a 64-bit value.
- The high-order 64 bits of the dividend are in **`%rdx`**, and the low-order 64 bits are in **`%rax`**. The divisor is the operand to the instruction.
- The quotient is stored in **`%rax`**, and the remainder in **`%rdx`**.

Division and Remainder

Instruction	Effect	Description
`idivq S`	`R[%rdx] ← R[%rdx]:R[%rax] mod S;` `R[%rax] ← R[%rdx]:R[%rax] ÷ S`	Signed divide
`divq S`	`R[%rdx] ← R[%rdx]:R[%rax] mod S;` `R[%rax] ← R[%rdx]:R[%rax] ÷ S`	Unsigned divide
`cqto`	`R[%rdx]:R[%rax] ← SignExtend(R[%rax])`	Convert to oct word

- Terminology: **dividend / divisor = quotient + remainder**
- The high-order 64 bits of the dividend are in **`%rdx`**, and the low-order 64 bits are in **`%rax`**. The divisor is the operand to the instruction.
- Most division uses only 64-bit dividends. The **`cqto`** instruction sign-extends the 64-bit value in **`%rax`** into **`%rdx`** to fill both registers with the dividend, as the division instruction expects.

Shift Instructions

The following instructions have two operands: the shift amount **k** and the destination to shift, **D**. **k** can be either an immediate value, or the byte register **`%cl`** (and only that register!)

Instruction	Effect	Description
`sal k, D`	`D ← D << k`	Left shift
`shl k, D`	`D ← D << k`	Left shift (same as `sal`)
`sar k, D`	`D ← D >>A k`	Arithmetic right shift
`shr k, D`	`D ← D >>L k`	Logical right shift

Examples:

```
shll $3,(%rax)
shrl %cl,(%rax,%rdx,8)
sarl $4,8(%rax)
```

Shift Amount

Instruction	Effect	Description
`sal k, D`	`D ← D << k`	Left shift
`shl k, D`	`D ← D << k`	Left shift (same as `sal`)
`sar k, D`	$D \leftarrow D >>_A k$	Arithmetic right shift
`shr k, D`	$D \leftarrow D >>_L k$	Logical right shift

- When using **`%cl`**, the width of what you are shifting determines what portion of **`%cl`** is used.
- For **`w`** bits of data, it looks at the low-order **`log2(w)`** bits of **`%cl`** to know how much to shift.
  - If **`%cl`** = 0xff (0b11111111), then: **`shlb`** shifts by 7 because it considers only the low-order log2(8) = 3 bits, which represent 7. **`shlw`** shifts by 15 because it considers only the low-order log2(16) = 4 bits, which represent 15.