# Section 1 / Register Sizes

## Overview

In each of the various sets of registers, each register can be referred to by different synonyms which determine how wide the register operation will be.

## General Purpose Registers

| Intended Width | Register Prefix | Instruction Postfix |
|---|---|---|
| 8 bytes | `x` | NA |
| 4 bytes | `w` | NA |
| 2 bytes | `w` | `h` |
| 1 byte | `w` | `b` |

### `ldr` (and `ldp`)

```
ldr     x0, [sp]    // load 8 bytes from address specified by sp
ldr     w0, [sp]    // load 4 bytes from address specified by sp
ldrh    w0, [sp]    // load 2 bytes from address specified by sp
ldrb    w0, [sp]    // load 1 byte  from address specified by sp
```

The address from which a load is taking *should* match the alignment of what is being loaded. That is, a long *should* be found only at addresses which are a multiple of 8 (the size of a long).

**When misaligned accesses to RAM are made, the processor must slow down and access each byte individually. This is a big performance hit. Properly aligned access is critical to performance.**

### `str` (and `stp`)

```
str     x0, [sp]    // store 8 bytes to address specified by sp
str     w0, [sp]    // store 4 bytes to address specified by sp
strh    w0, [sp]    // store 2 bytes to address specified by sp
strb    w0, [sp]    // store 1 byte  to address specified by sp
```

See above for comments about misaligned memory access.

### Example

Let's look at this program:

```
        .global    main                                             // 1
        .text                                                       // 2
        .align    2                                                 // 3
                                                                    // 4
```

```
main:   mov     x0, xzr                                      // 5
        ldr     x1, =ram                                     // 6
        strb    w0, [x1]                                     // 7
        strh    w0, [x1]                                     // 8
        str     w0, [x1]                                     // 9
        str     x0, [x1]                                     // 10
        ret                                                  // 11
                                                             // 12
        .data                                                // 13
ram:    .quad   0xFFFFFFFFFFFFFFFF                           // 14
                                                             // 15
        .end                                                 // 16
                                                             // 17
```

`Line 14` puts an identifiable pattern into 8 bytes of RAM and gives them the symbol `ram`.

`Line 6` gets the address of these bytes into `x1`.

The next four lines put zeros into that memory using progressively wider store instructions.

The following is a `gdb` session running the above program. Line numbers have been added to assist with the description of the session. Rather than describe all after a wall of text, descriptions will be provided inline.

```
(gdb) b main                                                      // 1
Breakpoint 1 at 0x740: file align.s, line 5.                      // 2
```

Immediately after entering `gdb` we set a breakpoint at `main`.

```
(gdb) run                                                         // 3
Starting program: /media/psf/Home/buffet/3510/pk_do/regs/a.out    // 4
                                                                  // 5
Breakpoint 1, main () at align.s:5                                // 6
5    main:    mov     x0, xzr                                     // 7
```

We launched the program and `gdb` stops its execution upon reaching the breakpoint.

```
(gdb) p/x $x0                                                     // 8
$1 = 0x1                                                          // 9
```

Before putting zero into `x0`, let's see what it currently holds... the value 1. Recall this is `argc`. The `p` command means `print` and is used to print the values in registers. The modifier `/x` says to print in hexadecimal.

```
(gdb) n                                                           // 10
6            ldr     x1, =ram                                     // 11
(gdb) p/x $x0                                                     // 12
$2 = 0x0                                                          // 13
```

After putting zero into `x0`, we confirm its contents.

```
(gdb) p/x $x1                                                     // 14
$3 = 0xfffffffff028                                               // 15
```

Prior to loading the address of 8 bytes found with the label `ram`, we print out the value already sitting in `x1`. The address it contains will be the address of the `C`-string containing the name of the program being run. Notice this value is 6-bytes long and not 8 as we might have expected. Why? The answer relates to the size of the virtual address each program is allowed. A full 64-bit virtual address space would make certain OS data structures too large for efficiency.

```
(gdb) n                                                       // 16
7            strb    w0, [x1]                                 // 17
(gdb) p/x $x1                                                 // 18
$4 = 0xaaaaaaab1010                                           // 19
```

After loading the address of `ram` into `x1`, we confirm its contents.

```
(gdb) p/x &ram                                                // 20
$5 = 0xaaaaaaab1010                                           // 21
```

Just for kicks, we confirm that the previous instruction really did get the address correctly.

```
(gdb) x/x &ram                                                // 22
0xaaaaaaab1010:    0xffffffff                                 // 23
```

We shift from print to examine to reach into memory and see what is found at `ram`.

```
(gdb) x/gx &ram                                               // 24
0xaaaaaaab1010:    0xffffffffffffffff                         // 25
```

Adding the `g` (for giant) we can see all 8 bytes.

```
(gdb) n                                                       // 26
8            strh    w0, [x1]                                 // 27
(gdb) x/gx &ram                                               // 28
0xaaaaaaab1010:    0xffffffffffffff00                         // 29
```

We just did a `strb` and looking at memory, we see one byte's worth of zeros.

*Note: this brings up an interesting question... which byte is actually sitting at the address of `ram`? We will have to look into this more later.*

```
(gdb) n                                                       // 30
9            str     w0, [x1]                                 // 31
(gdb) x/gx &ram                                               // 32
0xaaaaaaab1010:    0xffffffffffff0000                         // 33
```

After storing a `short`.

```
(gdb) n
10          str     x0, [x1]
(gdb) x/gx &ram
0xaaaaaaab1010:    0xffffffff00000000
```

After storing an `int`.

```
(gdb) n
11          ret
(gdb) x/gx &ram
0xaaaaaaab1010:    0x0000000000000000
(gdb) quit
```

And finally, after storing a `long`.

Let's circle back to the question asked above: Which byte is actually at the address `ram`? When we examined the `long` just after putting in one byte of zero, we saw this:

```
(gdb) x/gx &ram
0xaaaaaaab1010:    0xffffffffffffff00
```

Notice the zeros come at the end. Keep in mind, these bytes are printed as a `long`.

But what if we look at these 8 bytes individually?

```
(gdb) x/gx &ram
0xaaaaaaaabb010: 0xffffffffffffff00
(gdb) x/8bx &ram
0xaaaaaaaabb010: 0x00    0xff    0xff    0xff    0xff    0xff    0xff    0xff
```

Look at that... the *least significant* byte of a `long` comes **first**.

This is the definition of `little endian`.

The following image is from here:

### Little Endian in More Detail

Given this program (not intended for meaningful execution... just examining memory):

```
        .global    main
        .text
        .align     2

main:    mov     x0, xzr
        ret

        .data
```

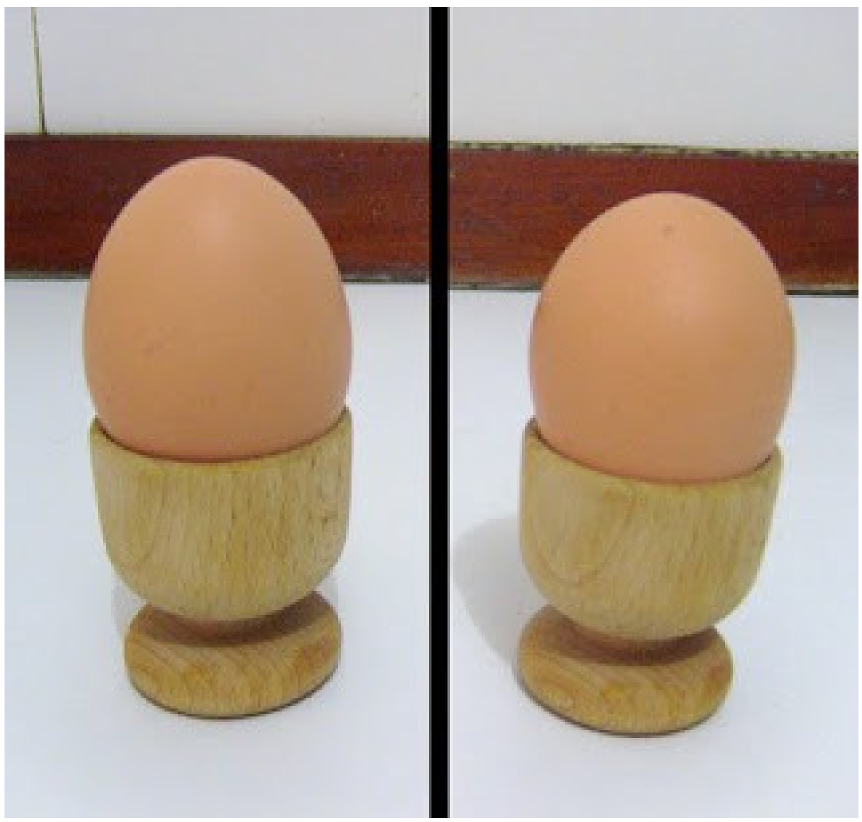

Figure 1: eggs

```
ram:     .quad   0xAABBCCDDEEFF0011                                 // 9
         .end                                                       // 10
```

let’s take a look at the memory at location `ram` in two ways. Once interpreted as a `long`:

```
(gdb) x/gx &ram
0x11010:    0xaabbccddeeff0011
```

and then interpreted as 8 bytes appearing in the order of lowest address to highest:

```
(gdb) x/8bx &ram
0x11010:    0x11    0x00    0xff    0xee    0xdd    0xcc    0xbb    0xaa
```

Compare the order of the bytes. They are least significant to most significant. Specifically:

- within a `long` the least significant `int` comes first
- within an `int`, the least significant `short` comes first
- within a `short` the least significant byte comes first

Endiannes isn’t an issue unless you’re exchanging data with a computer that has a different endedness and then only if the data being transferred is longer in native width than 1 byte. Text, expressed in single bytes, is immune from endedness issues - text is an array of bytes and is the same on all platforms.

### What Happens to the Rest of a Register When Only a Portion is Affected?

Whenever a narrower portion of a register is written to, the remainder of the register is zero’d out. That is: `ldrb` overwrites the least significant byte of an `x` register and zeros out the upper 7 bytes.

*There are dedicated instructions for manipulating bits in the middle of registers.*

### Casting Between `int` Type

Casting between integer types is in some cases accomplished by `anding` with `255` and `65535` (for `char` and `short`). Otherwise, see the previous section (What Happens to the Rest of a Register…).