Kickstart

In this section, we'll kickstart our exploration of AARCH64 assembly language.

Registers

At the dawn of time, central processing units (CPUs) could operate upon memory directly as both were relatively the same speed. CPUs got smaller and faster, leaving the speed of memory in the dust but also memory got further and further away.

A CPU might be on one board while RAM was on another board and had to be accessed over a shared bus. CPUs got still smaller and faster and might be on one chip and RAM on another set of chips.

The idea of registers were introduced a very long time ago as being super fast storage that is implemented directly in the CPU. Because they are within the CPU, distance isn't really an issue. Similarly, because they are in the CPU, they operate as the speed of the CPU itself.

Registers don't have addresses because they are not in memory. Instead they have names and naming conventions. They have only a minimal concept of typing as apart from integer, floating point (single or double precision) and pointer, every other notion of "type" is syntactic sugar provided by your language and its compiler.

ARM processors are RISC processors (Reduced Instruction Set Computers). The rough idea behind RISC is to make instructions simpler so as to make room for more registers. AARCH64 (what we are studying) has a lot of registers. Thirty two integer registers and thirty two floating point registers. Also, there are thirty two vector or SIMD registers.

rn means register "of some type" number n.

The kind of register is specified by a letter. Which register within a given type is specified by a number. There are some exceptions to this. Here is an introductory summary:

Letter	Type
x	64 bit integer or pointer
w	32 bit *or smaller* integer
d	64 bit floats (doubles)
s	32 bit floats

Some register types have been left out.

Integers

This declares an integer	This IS an integer
char	wn
short	wn
int	wn
long	xn

Registers do not have to be declared. They simply ARE.

There are 32 integer registers but some, like x30 are used for specific purposes. x31 is also not available.

When you want a 64 bit integer operation you use an x register. For all other integer operations, you use a w register and further specify the size by using different instructions, as you'll see later.

The x and w registers occupy the same space. w0 is the lower half of x0 for example. Writing into x0 will overwrite w0 and vice versa.

Pointers

This declares a pointer	This IS a pointer
type *	xn

All pointers are stored in x registers. X registers are 64 bits long but many operating systems do not support 64 bit address spaces because keeping track of that big of an address space itself would use a lot of space. Instead OS's typically have 48 to 52 bit address spaces. So, while a pointer sits in a 64 bit register, it is possible that not all of the bits are actually used in making a pointer.

Floats

There are 32 additional registers for floating point values ranging from half floats to single precision to double precision and beyond. What is beyond double? Vector registers can support multiple values in a single "very large" register.

If you've never heard of *half floats*, don't worry. After this course you will likely never hear of them again.

Instructions

Remember what RISC means? *Reduced Instruction Set*? Well, that was then. This is now. AARCH64 has an enormous instruction set - hundreds of instructions

each potentially has many variations. You will be responsible for mastering every one.

Kidding - you won't be responsible for too too many. Relatively few. Even so, you will be able to write sophisticated programs.

EVERY AARCH64 instruction is 4 bytes wide. Everything the CPU needs to know about what the instruction is and what variation it might be plus what data it will use will be found in those 4 bytes.

You might immediately wonder, how could you load a big big number into a register if the whole instruction is exactly and always 4 bytes long? Be patient, campers.

But we can draw a distinction between the RISC nature of ARM and the CISC (Complex Instruction Set Computer) nature of the Intel processors. The x86 and x64 ISA has variable length instructions ranging from 1 to 15 bytes in length! Complex indeed!

Every instruction is specified by an mnemonic consisting of some letters which the *assembler* converts into numeric *op-codes*.

Most (but not all) AARCH64 instructions have three *operands*. These are read in the following way:

```
op      ra, rb, rc
```

means:

```
ra = rb op rc
```

For a concrete examples:

```
sub     x0, x0, x1
```

means

```
x0 = x0 - x1
```

An example of a two operand instruction is:

```
mov     x0, x1
```

This means *copy* the 64 bit contents of x1 into the 64 bit register x0.

Or:

```
x0 = x1
```

Mixing Register Types

With few exceptions, different register types cannot be part of the same instruction. For example adding a 64 bit register to a 32 bit register cannot be done. For example:

Given:

```
mov    w0, 10       // puts 32 bit 10 in w0
```

You cannot do this:

```
add    x1, w0, x1   // attempts to add w0 and x1 - BAD
```

But you can do this:

```
add    x1, x0, x1   // This is fine
```

Putting a smaller-than-64-bit value into an integer register zeros out the higher order bits (ignoring the sign bit - explained later). So, you can safely view the x register when a value was placed into its corresponding w register.

Two Instructions for Dealing with Memory

With minimal exception, the AARCH64 ISA permits operations to be performed only on data in registers. Two obvious instruction families are those for transferring data from RAM to register(s) and those for transferring data from register(s) to RAM.

Both loading and storing instructions must specify:

- The registers involved
- The address in RAM involved (always held in an x register)

Loading has the general form of:

```
ldr    rn, [xm]
```

This is like:

```
type * ptr = some address;
type var;

var = *ptr;
```

Storing has the general form of:

```
str    rn, [xm]
```

This is like:

```
type * ptr = some address;
type var;

*ptr = var;
```

The analogies are not exact but close.

Pairs of registers can also be stored and loaded with the `stp` and `ldp` op codes.

Post increment and decrement of the pointer is also supported.

Pre increment and decrement of the pointer is also supported.

Now You Are Ready to Proceed

Have fun!