4 January 2021

AArch64 Exception Levels

by Mike Krinkin

I'm continuing my exploration of the AArch64 architecture and this time I will touch on the AArch64 priviledge levels.

Note that AArch64 priviledge model is not exactly the same as the previous iterations of ARM. While there are plenty of similarities, and there is a level of backward compatibility, at the same time, there are some differences as well. So do not assume that things covered here will work the same way for all ARMs.

Finally, I assume that you're familiar with general GNU Assembler synatax or willing to figure things out as you go. Familiarity with ARM assmebly language will help, though I try to explain all the things I use.

As always the code is available on GitHub.

AArch64 Exception Levels (a.k.a. EL)

ARM specification defines four priviledge levels (with a caveat) that are called Exception Levels and numbered from 0 to 3, where `EL0` is the lowest privledge level and `EL3` is the highest privledge level.

It's not that different from the priviledge levels existing for x86, but there are a few caveats. ARM chips aren't required to implement all four privledge levels. That makes a lot of sence given that the range of applications for the ARM chips appear to be quite a bit wider than that of the x86. It's not uncommon to see ARMs that don't even have an MMU and don't really need multiple privledge levels to begin with.

The rough purposes of each EL are as follows:

- EL3 the highest priveldge level is typically used for so called `Secure Monitor`, and I'm still quite ignorant of the overarching purpose of the `Secure Monitor`, so I cannot provide more than the basic information:
  - EL3 firmware typically implements the Power State Coordination Interface (PSCI) for the lower ELs to use;
  - EL3 firmware typically involved into trusted boot - a technology that, as far as I can understand, targets establishing a chain of trust starting rooting at the hardware level with the goal to make sure that the hardware runs the code that it was intended to run and there was no unauthorized changes (which totally makes sense for servers, but the use of such restriction for consumer electronics is controversial).
- EL2 seem to target the virtualization use-case specifically and that's the EL at which hypervisors would normally use for virtualization purposes.
- EL1 is the level that priveldged parts of the OS kernels use, so for example, Linux Kernel code will run with EL1 priveledges.
- EL0 is the most unprivledged level and therefore that's where the most unpriviledged code runs (userspace application, userspace drivers, etc).

As I mentioned above not all levels are required. For example, a particular hardware can implement only EL3, EL1 and EL0, but not implement EL2 (and therefore all the ARM extensions supporting virtualization) or the hardware can go with the simplest approach and just implement EL1 and EL0.

So we should have an idea of what different ELs are used for, but what is the actual difference in priveldges between those levels? For example, what makes EL2 higher privildged level than EL0?

The levels aren't exactly uniform, as you may have guessed, so the complete answer to that question that will cover all the levels is going to be quite complicated. That being said I will cover a few basic ideas and examples below.

To start with, a higher EL has access to all the registers of the lower levels while the opposite is not true.

For example, code running in EL2 can modify `TTBR0_EL1` and `TTBR1_EL1` registers that contain page table pointers used for address translation on EL1 and EL0. So potentially it's possible to control address translation on the EL1 and EL0 from the EL2.

Moreover higher ELs may control trapping of certain conditions on lower levels. By trapping here I mean that when the condition happens the higher EL gets a notification in a form of exception/interrupt.

For example, on EL2 we may configure trapping of accesses to `TTBR0_EL1` and `TTBR1_EL1` registers, so that every time a code running on EL1 or EL0 tries to read or write from/to them it generates an exception/interrupt processed on the EL2.

One way to look at this is that higher levels can enable or disable certain features. Another way to look at it is that if the higher EL is notified every time lower EL uses a feature, the higher EL can emulate this feature for virtualization purposes.

The final example is interrupt routing. Given that one of the primary purposes of the OS is to manage hardware resources it's quite important who decides who will process signals from hardware. It would be quite bad for isolation in the OS if any userspace application can just decide that it should handle the hardware interrupts on its own without consulting with the OS kernel.

So naturally higher ELs can decide whether they are going to handle hardware signals themselves or allow the lower ELs handle them.

Jumping between ELs

Not surprisingly, when the CPU just starts it starts at the highest privledge level supported by that CPU. So for example, if you CPU supports EL3 then the first code executed on that CPU will be executing in EL3.

However at some point we need to switch to the lower EL. For example, once the OS kernel has been initialized it may want to start the first userspace process of the OS that will run in a non-privledged EL0 level. So naturally we need to know how to do that.

ARM specification is pretty clear that there are only two ways to change the EL:

- take an exception/interrupt - this may switch CPU from a lower EL to a higher EL
- return from an exception/interrupt - this may swtich CPU from a higher EL to a lower EL.

As I mentioned above higher EL may decide if they want to handle interrupts themselves or allow lower ELs handle them. If they decide to handle interrupts themselves, when an interrupt happen CPU automatically interrupts (thus the name) currently executing code and if necessary transfers CPU to the higher EL to handle the interrupt.

Returning from interrupt performs the action opposite to taking an interrupt and therefore can swtich the CPU from a higher EL to a lower one.

If only iterrupts can change EL how can we switch from a higher EL to a lower EL for the first time? There is no interrupt that brought CPU from a lower EL to the higher EL, so there is nowhere to return. For example, how can an OS kernel running in EL1 start a new userspace application at EL0 and pass control to it?

Well, ARM has a specific instruction used to return from an interrupt: `eret`. This instruction can be executed even if there was no interrupt. So what we can do is to prepare all the state that `eret` instuction needs and run the `eret` instruction without an actual interrupt or exception.

The state required by the `eret` instruction is stored in a few registers. For example, when running on EL2 and executing `eret` instruction takes the address of the instruction to return to from the `ELR_EL2` register and some other state of the processor from `SPSR_EL2` register.

Among other things `SPSR_EL2` register contains the EL to which the `eret` instruction should swtich the processor to. As you may have guessed already, the `_EL2` suffix means that those registers are accesible from EL2 and EL3 only and not accesible from EL1 and EL0. Other levels, except EL0, have their own version of those registers. For example, there is `SPSR_EL3` and `ELR_EL3` registers.

Unsurprisingly, there are restrictions on the values of `SPSR_EL2` that only allow `eret` to "return" to EL0-EL2, but not to EL3. Similarly, values `SPSR_EL1` are limited to only allow to "return" to EL0-EL1, but not EL2 and EL3.

Example: jumping to lower EL

Now it's time to for some practice. However, before we proceed I want to put forward an important disclaimer: I was not able to test the following code in QEMU or a real hardware.

The following code shows a jump from EL3 to EL2 and somehow the version of UEFI firmware that I use with QEMU doesn't work in EL3. Additionally, I don't have an actual physical hardware that supports EL3. So my confidence in the following code is quite low.

Anyways, I started learning the AArch64 architecture with the goal to learn more about the virtualization support in ARM, so I want my code to run on EL2 to explore the virtualization extensions - that's what I'm going to target in my example.

When my code starts there are few possibilities:

- my code is already running in EL2, so there is nothing to do
- my code runs in EL1 or EL0, so it's already to late to do anything as we cannot increase the priviledge leve
- my code runs in EL3, so we need to jump from EL3 to EL2.

> *NOTE:* a particular hardware will always start in the same EL and a particular firmware will switch to a specific EL, so it's not like my code can randomly endup in any of those three possibilities. So potentially I could just claim that my code always start at EL2 and look for hardware that supports this option (or use QEMU with the right configuration).

How can we figure out what EL we are currently running on?

AArch64 has a dedicated register (`CurrentEL`) that we can read to figure out our the current EL. `CurrentEL` is a system register and we should use a special instruction `mrs` to read such system registers, so the code starts like this:

```
mrs x0, CurrentEL
```

This instruction will read the content of the `CurrentEL` to a general purpose register `x0`. The register `CurrentEL` is a 64-bit one, but most of the bits are hardwired to be 0 and only bits 2 and 3 contain the actual EL value (see this doc).

> *NOTE:* I have no idea why they decided to use bits 2 and 3 instead of 0 and 1, but that's hardly a particularly bad example of hardware quirks.

So now we need to test if the value we read from `CurrentEL` is:

- equal to 8 - nothing for us to do, we are already at EL2
- smaller than 8 - to late for us to do anything, we are in EL1 or EL0
- higher than 8 - we are in EL3 and have to jump to EL2.

> *NOTE:* remember that the EL value is stored in bits 2 and 3, that's why we compare to 8 and not to 2 (8 is just 2 shifted by 2 bits).

Putting everything together we get the following:

```
    mrs x0, CurrentEL
    cmp x0, #0b1000  // remember the EL value is stored in bits 2 and 3
    beq in_el2
    blo in_el1

in_el3:
    // Put code switching from EL3 to EL2 here

in_el2:
    // This code will run at EL2

in_el1:
    // We are in EL0 or EL1, so we probably should report an error here
```

The `cmp` instruction above compares the value in register `x0` with 8 (`0b1000` is just 8 written in binary form). Depending on the result of comparision it will set some flags that can be used by other instructions.

The `beq` instruction for example looks at the flags and jump to the provided label if the result of the comparision indicated equality. In other words `beq` will transfer control to `in_el2` if the value in `x0` is equal 8. If they are not equal, then this instruction will pass control to the next instruction, as usual.

The `blo` is similar in behavior to `beq`, but it jumps to `in_el1` if the value in `x0` turned out to be lower (thus the name) than 8.

Finally, if both instructions didn't jump to `in_el2` or `in_el1`, then EL is neither equal to 8 nor lower than 8, so it must be higher than 8. That means that we are in EL3 and have to change the EL.

So starting from the `in_el3` label we need to put code that prepares us for the jump to EL2 from EL3 and exectues `eret` instruction to do the actual jump.

First, we need to figure out where exactly we want to jump. We know that we want to endup in EL2, but what code do we want to run in EL2?

As shown in the snippet above I have created a label for EL2 code called `in_el2`, so that's where we are going to jump. We need to write this address to the `ELR_EL3` register. `ELR_EL3` register is, like `CurrentEL`, a special system register and the instruction used to write a value there is `msr` (`mrs` to read and `msr` to write):

```
in_el3:
    adr x0, in_el2
    msr ELR_EL3, x0

    // TO BE CONTINUED ...

in_el2:
    // This code will run at EL2
```

The `adr` instruction above stores an absolute address of the `in_el2` to the register `x0`, so after this instruction `x0` will hold the address denoted by `in_el2` label. That's what we want to write to the `ELR_EL3` using the `msr` instruction.

The second part of the state that `eret` uses is stored in `SPSR_EL3` register. This register contains quite a few various flags that represent saved CPU state (and `eret` restores the saved CPU state from the register):

- condition flags - flags that instructions like `cmp` set to describe the result;
- interrupt masks - flags that prevent generation of interrupts/exceptions at a certain EL;
- target EL - what level we want to swtich to;
- a few other things…

We don't really care about the condition flags and a bunch of other things in the register, but we do want to specify what EL we want to swtich to.

In order to switch to EL2 we need to store in bits 0-4 of `SPSR_EL3` either `0b01000` or `0b01001`. There is difference betweent those two values, but I will cover it a little bit later. For now I'm just going to state that we will use `0b01001` and set all other bits to 0s:

```
in_el3:
    adr x0, in_el2
    msr ELR_EL3, x0

    mov x0, xzr
    orr x0, #0b01001
    msr SPSR_EL3, x0

    // TO BE CONTINUED ...

in_el2:
    // This code will run at EL2
```

The `xzr` in the snippet above is a special register that contains all 0s. I'm using it to zero out the value in x0 and then using the "bitwise or" instruction (`orr`) is set the bits I want.

Finally, when we configured `ELR_EL3` and `SPSR_EL3` we can actually do the jump:

```
in_el3:
    adr x0, in_el2
    msr ELR_EL3, x0

    mov x0, xzr
    orr x0, #0b01001
    msr SPSR_EL3, x0

    eret

in_el2:
    // This code will run at EL2
```

The `eret` instruction should jump to `in_el2` label, which, when you look at the code, is not a particularly impressive jump. However what's more important is not what code we jump to, but the fact that during this jump processor will switch from a higher priviledge level to a lower priviledge level.

Execution State

Chaning priviledge level, as you saw above, is not really complicated. However, normally we don't just want to switch to the lower EL, we want to run some code at a lower EL and that code may need some additional environment to work.

For example, code may need stack to run, or we may want to mask interrupts, at least until the interrupt handling has been set up properly. So let's cover those two things as well.

Let's start with stack. In AArch64 there are dedicated stack pointer registers for each EL. Specifically, there are `SP_EL3`, `SP_EL2`, `SP_EL1` and `SP_EL0`. However there is a bit of additional complexity with those registers.

When code actually works with stack it doesn't expclicitly specify `SP_EL2` or `SP_EL0` for the level it runs in. Instead it just uses `sp`, and processor maps `sp` to one of those EL specific registers.

The store doesn't end there however. In AArch64 there are two modes of operation:

- all ELs use `SP_EL0` - `sp` mapped to `SP_EL0` for all ELs
- each EL use a dedicated stack pointer - `sp` on EL0 is mapped to `SP_EL0`, on EL1 to `SP_EL1`, and so on.

In other words, even though there is a dedicated stack pointer for each EL it doesn't mean that each EL will use their own stack pointer. We can control which mode of operation we will use after `eret` via the `SPSR` register.

Remember that to switch to EL2 there were two possible values of bits 0-4 in `SPSR_EL3` we could use:

- `0b01000` - this value will swtich us to EL2 using `SP_EL0` (so `sp` will be mapped to `SP_EL0`)
- `0b01001` - this value will switch us to EL2 using `SP_EL2` (sp `sp` will be mapped to `SP_EL2`).

We actually could have picked any of them, but the second option where each EL uses a dedicated stack pointer makes slightly more sense to me and, therefore, that's what I picked.

With all that out of the way, what we actually want to do is to make sure that the stack pointer that will be used in EL2 will contain a valid value. The esiest way to do that is to just take the current value of `sp` and save it to `SP_EL2` register. Then when we switch to the EL2, the register `sp` will be mapped to `SP_EL2` and will start using the value we saved there:

```
in_el3:
    adr x0, in_el2
    msr ELR_EL3, x0

    mov x0, xzr
    orr x0, #0b01001
    msr SPSR_EL3, x0

    mov x0, sp
    msr SP_EL2, x0

    eret

in_el2:
    // This code will run at EL2
```

The `mov` instruction copies value from the register `sp` to the register `x0`. We need to make this copy because, evidently, we cannot use register `sp` directly with the `msr` instruction.

The `msr` instruction just stores the value of `x0` to `SP_EL2`, as we saw with other system registers above - nothing special here.

We dealt with stack now, let's take a look at interrupts. I will probably cover general interrupt handling in a separate post. For now it's worth mentioning that before we can handle any interrupts there are some preparations to be done.

As a result, since we didn't do any preparations to handle interrupts at the EL2, we should not assume that we can safely take an interrupt and we should mask them.

Masking interrupts doesn't prevent external hardware from signaling to the processor, it just tells the processor to ignore those signals for a time being.

Again, we can mask interrupts in EL2 when jumping there by writing the right value to `SPSR_EL3`. The bits 6, 7, 8 and 9 are used to mask various types of interrupts. Without going in much details we can just mask them all together:

```
in_el3:
    adr x0, in_el2
    msr ELR_EL3, x0

    mov x0, xzr
    // Mask interrupts
    orr x0, #0b1111000000
    // Set the target EL and stack
    orr x0, #0b0000001001
    msr SPSR_EL3, x0

    mov x0, sp
    msr SP_EL2, x0

    eret

in_el2:
    // This code will run at EL2
```

You might be curious why I used two separate `orr` instructions to update `x0`. Even though it has nothing to do with the EL or interrupt handling it's a rather typical quirk.

All instructions in the base ARM instruction set are 4 bytes long. Those 4 bytes need to inculde the code of the operation itself as well as the arguments.

So if one of the arguments of the instruction is a number (not a number in a register, but an actual number), it has to be packed into 4 bytes together with the code of the operation and other arguments.

Because of this you cannot use arbitrary numbers as arguments of instructions in ARM and have to deal with some limits. ARM encoding of such arguments is somewhat funky, so the limit is not as simple as just a range of values, but nevertheless the limit is always there.

Because of this limit I had to split one `orr` instruction in two `orr` instructions setting different parts.

Instead of conclusion

That's it for the post. I intend to cover general interrupt handling in AArch64, however while writing the text I realized that if I have to cover all the topics I wanted to cover in one post it will be quite long. So this post gives a hopefully gentle introductions into exception leves, some basic ARM assembly language and the basic registers involved into interrupt handling.

Again, if you think that some parts aren't explained well enough or could be explained better, I welcome any suggesstions.

tags: *aarch64 - arm - system-programming*