Adding v8-R64 Support to TF-A

TF-A Tech Forum

Gary Morrison and Lauren Wehrmeister
1 July 2021

Agenda

What to do:

- What's v8-R64?
- v8-R64 architecture differences
  - No-EL3
  - MPU vs. MMU
  - Secure-Mode Only
  - No AArch32 support
- Why add v8-R64 support to v8-A firmware?

How to do it:

- The immediate assignment
- No-EL3 ramifications
- MPU ramifications
- No non-secure ramifications
- Loading runtime system and transfer of control
- Porting approach
- Review approach

Armv8-R AArch64 Architecture ("v8-R64")

What's v8-R64?

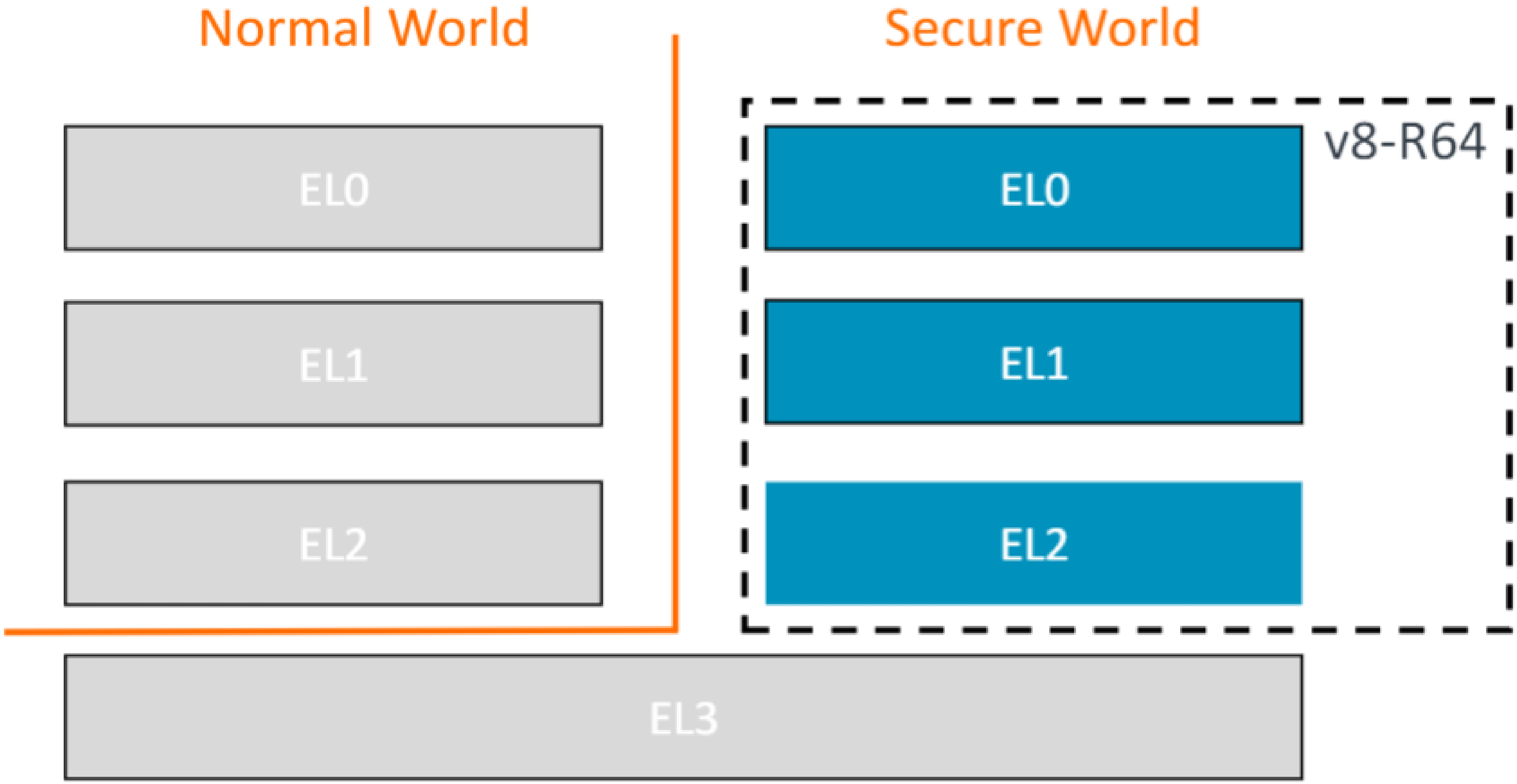
- Next generation of R-Class family: AArch64. Cortex-R82 is first implementation.
- R-Class cores are especially targeted toward *deterministic performance* (e.g., avoiding the unpredictable timing of table walks).
- Markets that have shown interest include Automotive and Computational Storage.
- The R-Class community is excited for all the reasons you'd expect:
  - Vastly-increased address space
  - Much-higher performance
  - v8-R64 can support rich-OSes in addition to comparatively-spartan RTOSes.
- More information available here: https://developer.arm.com/ip-products/processors/cortex-r/cortex-r82

v8-R64 Architecture Differences

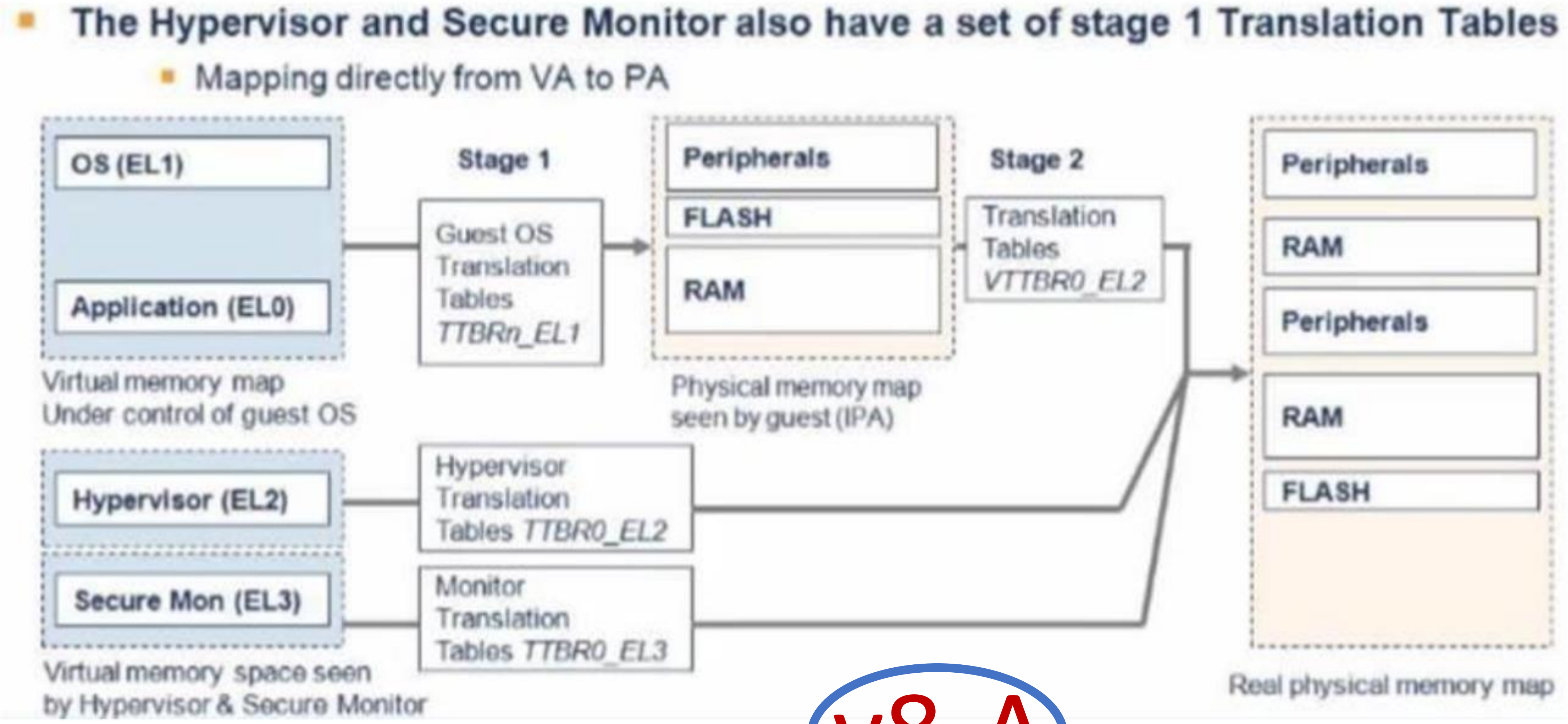
Most basic differences compared to A-Class cores:

- No EL3.
- Everything is Secure.
- For Stage-2 translation, MPU only.
- For Stage-1 translation, MMU or MPU.
- No AArch32 support.

v8-R64 Architecture Differences



v8-R64 Architecture Differences – MPU



- **The Hypervisor and Secure Monitor also have a set of stage 1 Translation Tables**
  - Mapping directly from VA to PA

OS (EL1)

Application (EL0)

Virtual memory map
Under control of guest OS

Stage 1

Guest OS
Translation
Tables
TTBRn_EL1

Peripherals

FLASH

RAM

Physical memory map
seen by guest (IPA)

Stage 2

Translation
Tables
VTTBR0_EL2

Peripherals

RAM

Peripherals

RAM

FLASH

Real physical memory map

Hypervisor (EL2)

Hypervisor
Translation
Tables *TTBR0_EL2*

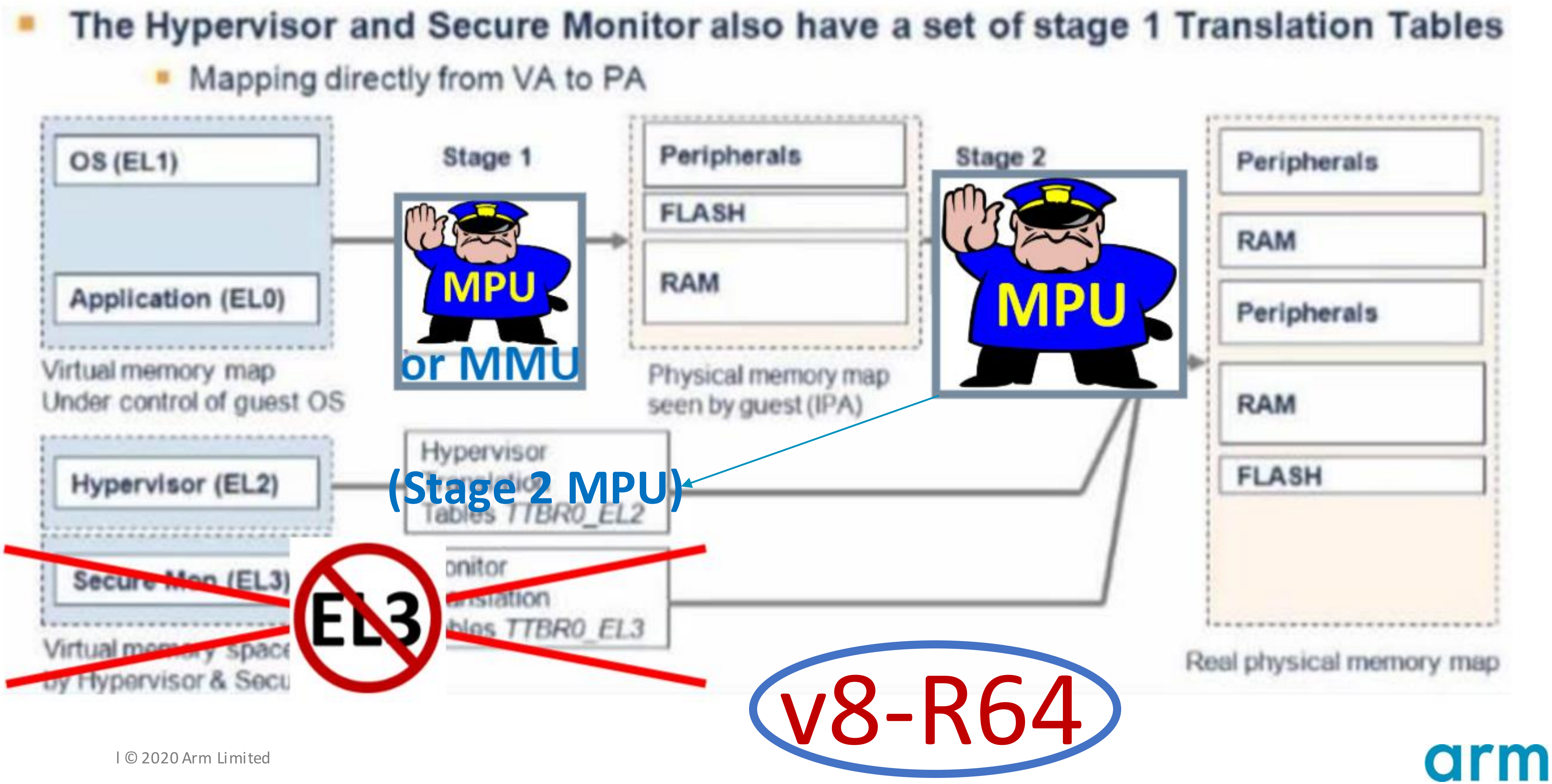
Secure Mon (EL3)

Monitor
Translation
Tables *TTBR0_EL3*

Virtual memory space seen
by Hypervisor & Secure Monitor

v8-A

v8-R64 Architecture Differences – MPU



- The Hypervisor and Secure Monitor also have a set of stage 1 Translation Tables
  - Mapping directly from VA to PA

The Assignment

Why Add v8-R64 Support to v8-A Firmware?

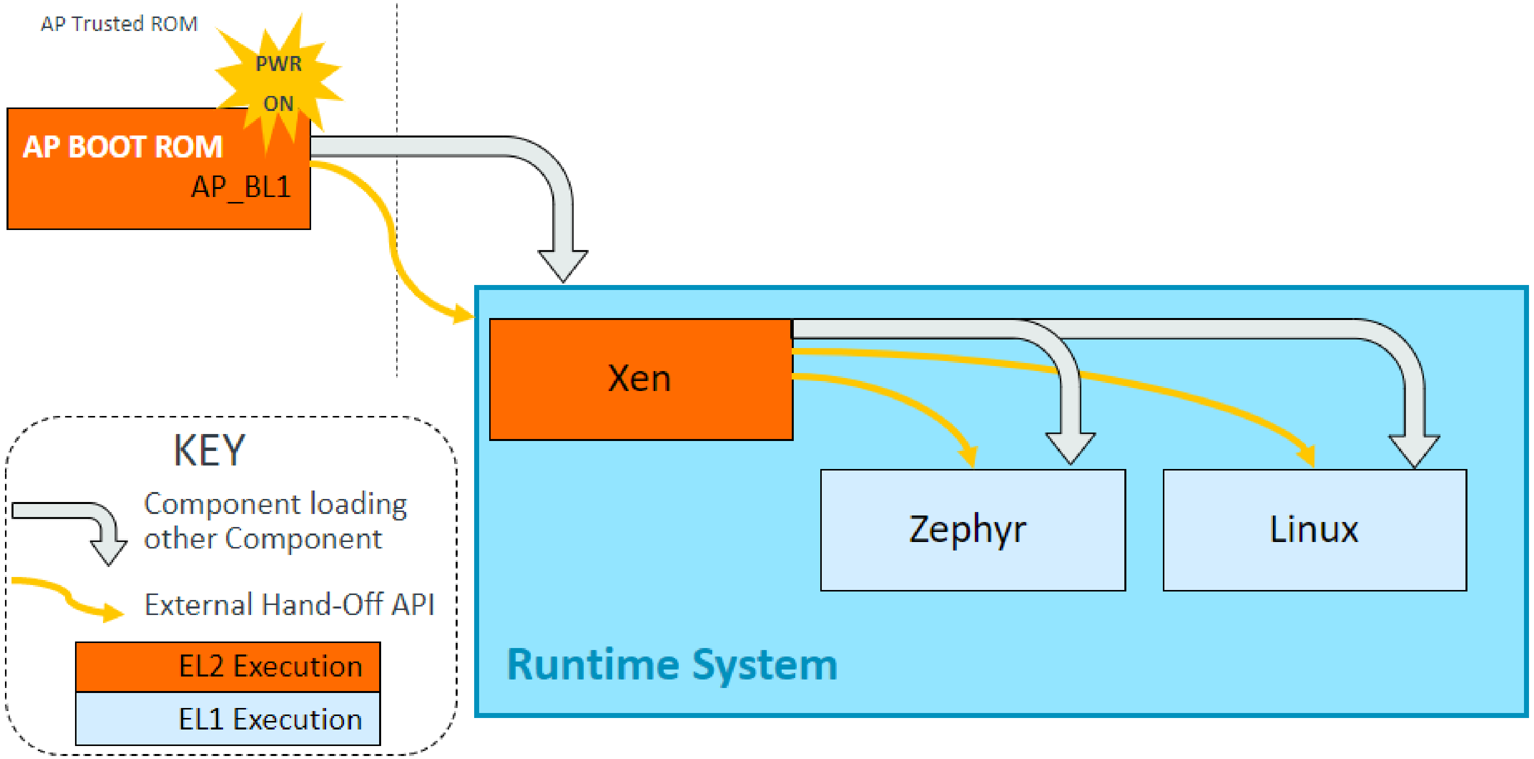
- v8-A and v8-R64 are far more similar than different.
- Most of the trusted-firmware code for v8-A is useful for v8-R64 as well.
- If a separate trusted-firmware project were created for v8-R64, we would have a huge parallel-maintenance headache.
- SystemReady IR certification for v8-R64 cores requires compliance with EBBR, and building from the TF-A framework puts us on the path toward that goal.

Perhaps the best reason though, is that open-source software is all about pooling our efforts. Adding v8-R64 developers support would recruit even more review and development talent to Trusted Firmware!

The Immediate Assignment

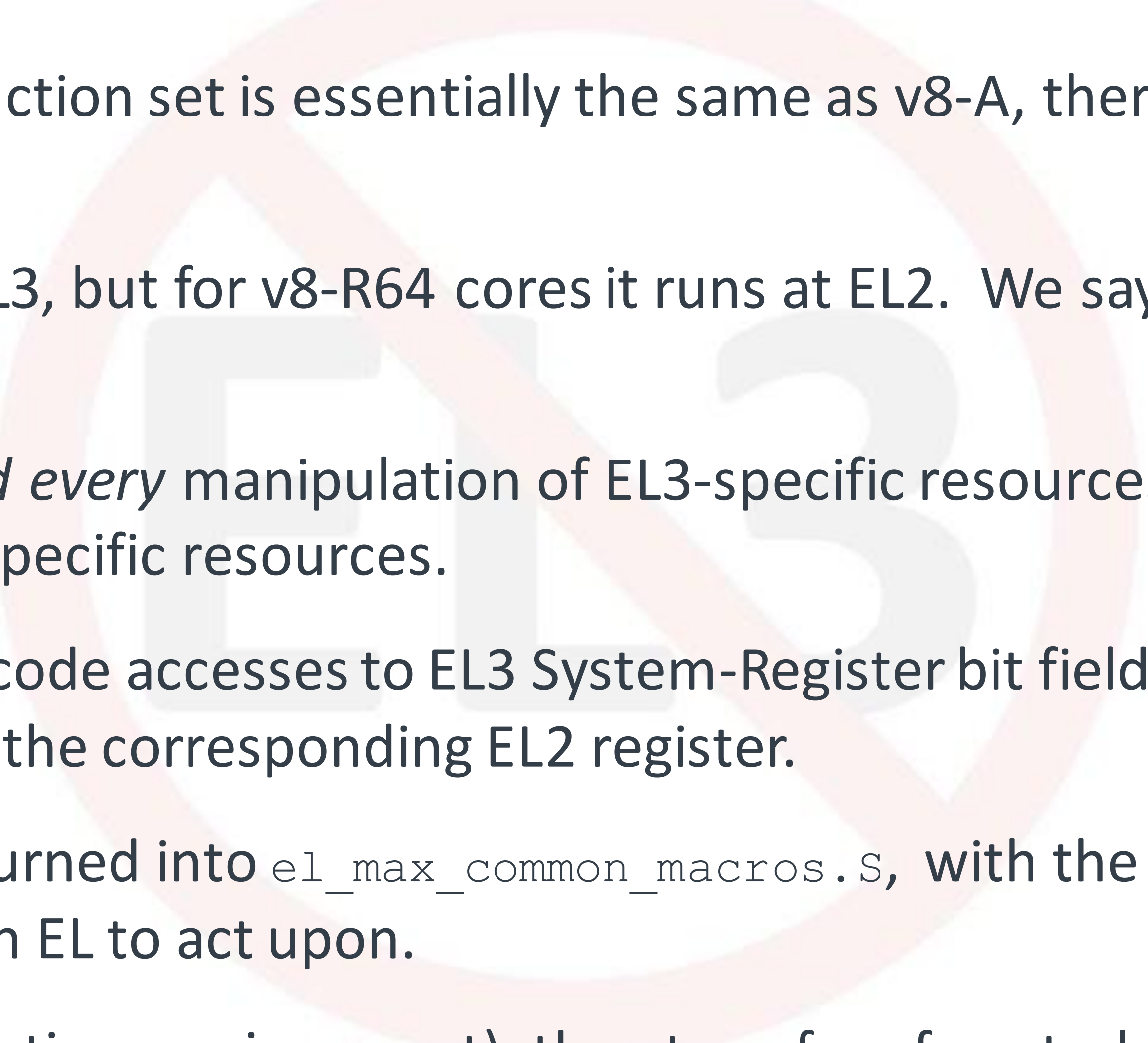
- The immediate assignment was ***not*** *to port all of TF-A* to v8-R64.
- The immediate assignment was:
  - To ***port BL1 only***, to ***EL2*** and ***MPU***,
  - to adapt BL1 authenticate, opaquely load, and transfer control to *a Customer/Partner run-time system instead of BL2*, and
  - to support Recovery-mode FWU.

The Immediate Assignment



Ramifications of v8-R64 Differences

No-EL3 Ramifications



- Although v8-R64 instruction set is essentially the same as v8-A, there are no EL3 System Registers.

- BL1 normally runs in EL3, but for v8-R64 cores it runs at EL2. We say BL1 runs in "ELmax".

- Ramifications: *Any and every* manipulation of EL3-specific resources is redirected to corresponding EL2-specific resources.

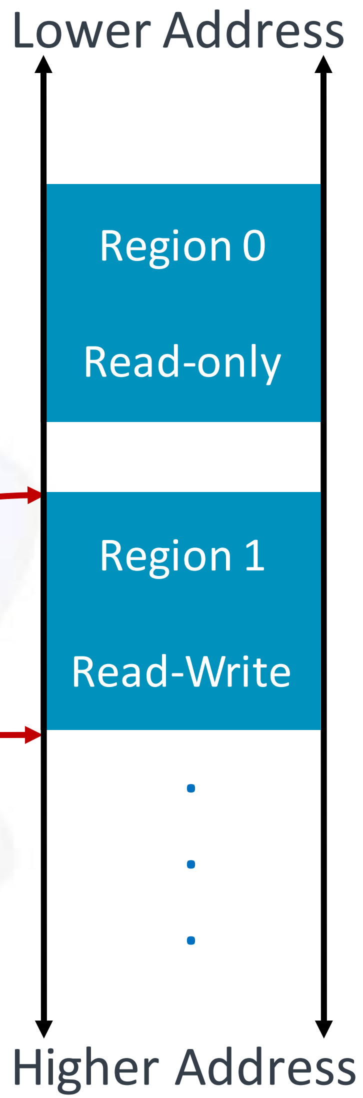
- In the majority of BL1-code accesses to EL3 System-Register bit fields, the same bit fields are the same for the corresponding EL2 register.

- `el3_common_macros.S` turned into `el_max_common_macros.S`, with the macros parameterized to which EL to act upon.

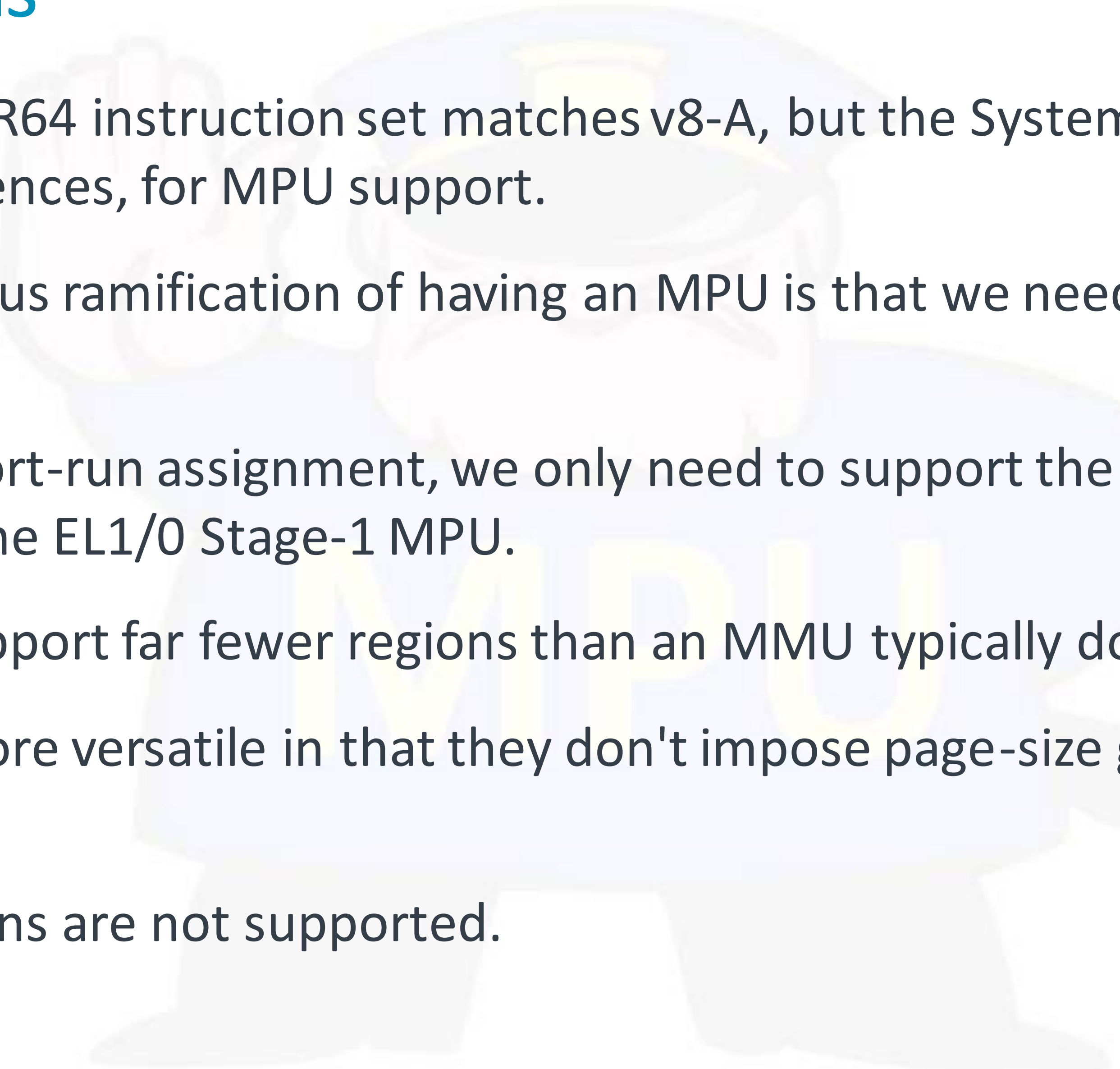
- EL2 BL1 loads BL33 (runtime environment), then transfer of control.

MPU Ramifications – MPU? Wuzzat?

- Memory-Protection Units are still said to "perform translations," but don't change addresses: PA = IPA = VA. Everything is strictly flat-, direct-mapped.
- MPUs only apply permissions to specified, contiguous address *regions*.
- Most importantly, they are ***register-based***, making timing *much more-deterministic* (no table walks).
- Regions are specified by System Registers:
  - **`PRBAR_ELn`**, Protection Region *Base*-Address Register
  - **`PRLAR_ELn`**, Protection Region *Limit* Address Register, also specifies permissions (which MAIR)
  - **`PRENR_ELn`**, Protection Region Enable Register (bitmap of which regions are enabled)
- Entirely *register-based*, so no non-deterministic-speed searching through page tables in memory.



MPU Ramifications



- As mention earlier, v8-R64 instruction set matches v8-A, but the System Registers do have substantial differences, for MPU support.
- Clearly, the most obvious ramification of having an MPU is that we need to add code to "drive" the MPU.
- For the immediate, short-run assignment, we only need to support the EL2/Stage-2 MPU, and not (yet) the EL1/0 Stage-1 MPU.
- MPUs also typically support far fewer regions than an MMU typically does.
- However, MPUs are more versatile in that they don't impose page-size granularity limitations.
- Overlapped MPU regions are not supported.

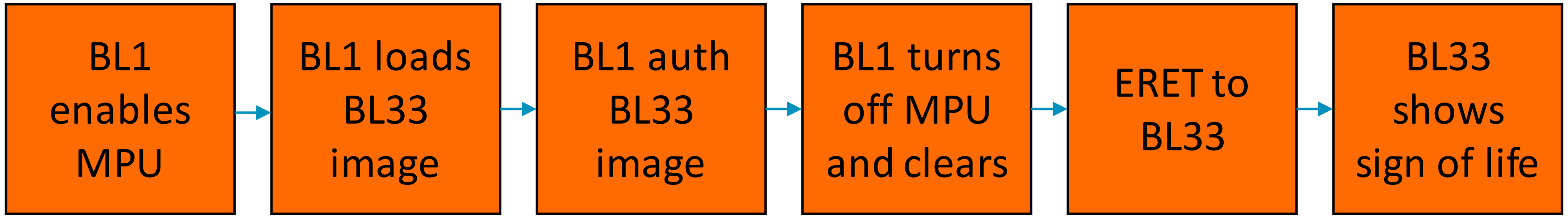
No Non-Secure Ramifications

- Having no EL3 to switch NS/Secure, v8-R64 mandates everything Secure mode.
- Arguably, making *everything* Secure makes *nothing* secure, or at least nothing any *more* secure than anything else.
- Still, this turns out to be lucky for TB-R purposes, because BL1 normally operates in Secure mode. The requested memory-translation permissions are already requested for Secure mode.
- Nevertheless, v8-R64 makes provisions for generating NS accesses from the Secure state:
  - The MMU region descriptors also have an equivalent NS bit.
  - We haven't seen need to use this feature, but it's available if at some point we do need it.

The Details

Loading Runtime System and Transfer of Control

- Load and Transfer from BL1 to BL33:
  - In EL2, BL1 enables MPU and loads and authenticates the BL33 image, which for v8-R64 purposes is the customer/partner runtime system, or bootwrapped-OS.
  - TBBR support is extended to fvp_r platform for BL1 for authentication.
  - After turning off the MPU and clearing regions, then an ERET is used to jump to the base address of BL33 for execution directly from BL1.
  - We used an internal bootwrapped-OS to test the transfer of control from BL1 to BL33 for a shell prompt on a UART as a sign of life.



BL1 enables MPU → BL1 loads BL33 image → BL1 auth BL33 image → BL1 turns off MPU and clears → ERET to BL33 → BL33 shows sign of life

- Transfer from BL1 to Recovery mode FWU:
  - The transfer from BL1 to Recovery mode FWU remains the same as BL1 copies the image from external interfaces and updates if the image can be authenticated.

TB-R Memory Map change requirements

- For addresses in the 0-0xffffffff range, the v8-R64 memory map is similar with v8-A, but the upper 2GB and lower 2GB are switched.
- In other words, bit 31 is reversed, so, for example, DRAM1 is at address 0 rather than address 0x80000000.
- See https://developer.arm.com/documentation/100964/1113/Base-Platform/Base---memory/Base-Platform-memory-map

  and https://developer.arm.com/documentation/100964/1113/Base-Platform/Base---memory/BaseR-Platform-memory-map
- The reasons are explained in the above documents, but for now, just FYI, the addresses are different from most other platforms.

Porting Approach

- Created a new `.../trusted-firmware-a/lib/xlat_mpu` library:
  - API is similar, where applicable.
  - For example, to program regions, you still set up a terminated array of `mmap_region_t`-type `struct`s, and call `setup_page_tables()` to create those regions. `mmap_add_region()` still creates.
  - Enable calls, however, named `enable_mpu_*()` rather than `enable_mmu_*()`. Same for `disable_*()`.
- `#define NO_EL3` (probably change to `BL1_AT_EL2`) to divert EL3-resource accesses to EL2.
- Created the `fvp_r` platform, patterned after Cortex R-82, that was ported to and tested on the v8-R64 FVP.
- BL1 image that stays in EL2 using MPU, loads the BL33 image, wipes MPU regions, and then jumps to the BL33 image.
- For our testing purposes we used an internal OS to test the transfer of control from BL1 to BL33.

Review Approach

- Gerrit reviews:
  - https://review.trustedfirmware.org/c/TF-A/trusted-firmware-a/+/10512 Platform definition
  - https://review.trustedfirmware.org/c/TF-A/trusted-firmware-a/+/10518 No-EL3 and MPU
  - https://review.trustedfirmware.org/c/TF-A/trusted-firmware-a/+/10519 Validate and load system
  - https://review.trustedfirmware.org/c/TF-A/trusted-firmware-a/+/10520 Documentation
- As with most new platforms, this is a rather large change, and it can't really be broken down much into smaller, stand-alone patches, and still compile.
- We hope to create a Phabricator site to help review *new files*:
  - Most of the "new" files are not really new, but derived from existing files.
  - The Gerrit review, however, has no way of knowing which existing files to compare them to.
  - If successful, we'll create a table listing each new file, which of the above patches it was created under, plus an `sdiff` output against a close-relative existing/familiar file.

Resources

- https://www.arm.com/products/silicon-ip-cpu/cortex-r/cortex-r82
- https://www.arm.com/company/news/2020/09/highest-performance-arm-cortex-r-processor
- https://community.arm.com/developer/tools-software/oss-platforms/w/docs/626/armv8-r-aarch64

arm

Thank You
Danke
Merci
谢谢
ありがとう
Gracias
Kiitos
감사합니다
धन्यवाद
شكرًا
ধন্যবাদ
תודה