Name: ___________________________ Entry no: ___________________

Overview

In this exam, we will reason about how to virtualize 64-bit ARMv8 architectures. Most concepts are similar to the Intel x86 architecture that we have been studying. But, there are some curious differences. The following excerpts are taken from [G]: "2015 - ARM Cortex-A Series Programmer's Guide for ARMv8-A". Please read them carefully to answer questions.

Design principles of virtualization support on ARM

ARM architecture virtualization extensions were introduced in 2010. Until then, virtualization on ARM systems was not widely used. However, as ARM CPUs continued to increase in performance, pushing from mobile devices into traditional servers, interest in ARM virtualization also grew.

[G 3] Fundamentals of ARMv8

In ARMv8, execution occurs at one of four Exception levels. The Exception level determines the privilege level, so an Exception level with a larger value of n than another one is at a higher Exception level.

Exception levels provide a logical separation of software execution privilege that applies across all operating states of the ARMv8 architecture. It is similar to and supports the concept of hierarchical protection domains common in computer science.

The following is a typical example of what software runs at each Exception level:

- EL0: Normal user applications.
- EL1: Operating system kernel typically described as privileged.
- EL2: Hypervisor.
- EL3: Low-level firmware.

In general, a piece of software, such as an application, the kernel of an operating system, or a hypervisor, occupies a single Exception level. An exception to this rule is in-kernel hypervisors such as KVM, which operate across both EL2 and EL1.

ARMv8 has the following privileged components:

- Guest OS kernels: Such kernels include Linux or Windows running in EL1. When running under a hypervisor, the rich OS kernels can be running as a guest or host depending on the hypervisor model.
- Hypervisor: This runs at EL2. The hypervisor, when present and enabled, provides virtualization services to rich OS kernels.

[G 3.2] Changing Exception levels

The processor mode can change under privileged software control or automatically when taking an exception. When an exception occurs, the core saves the current execution state and the return address, enters the required mode, and possibly disables hardware interrupts.

Movement between Exception levels follows these rules:

- Moves to a higher Exception level, such as from EL0 to EL1, indicate increased software execution privilege.
- An exception cannot be taken to a lower Exception level.
- There is no exception handling at level EL0, exceptions must be handled at a higher Exception level.
- An exception causes a change in the program flow. Execution of an exception handler starts, at an Exception level higher than EL0, from a defined vector that relates to the exception taken. Exceptions include
    - Interrupts.
    - Memory system aborts.
    - Undefined instructions.
    - System calls. These permit unprivileged software to make a system call to an operating system.
    - Hypervisor traps.
- Ending exception handling and returning to the previous Exception level is performed by executing the ERET instruction.
- Returning from an exception can stay at the same Exception level or enter a lower Exception level. It cannot move to a higher Exception level.

[2 marks] Trapping on undefined instructions sounds unnecessary. One should never have undefined instructions anyways. Why not just halt the CPU, i.e, it does not process any more instructions?

[2 marks] How come we do not allow interrupt handling at EL0? If an EL0 software was smart enough, we could just let it handle all the interrupts. This could be performant too since we will not have to change exception levels.

[G 4] ARMv8 registers

The execution state provides 31 64-bit general-purpose registers accessible at all times and in all Exception levels. Each register is 64 bits wide and they are generally referred to as registers X0-X30.

[G 4.1] Special registers

In addition to the 31 core registers, there are also several special registers.

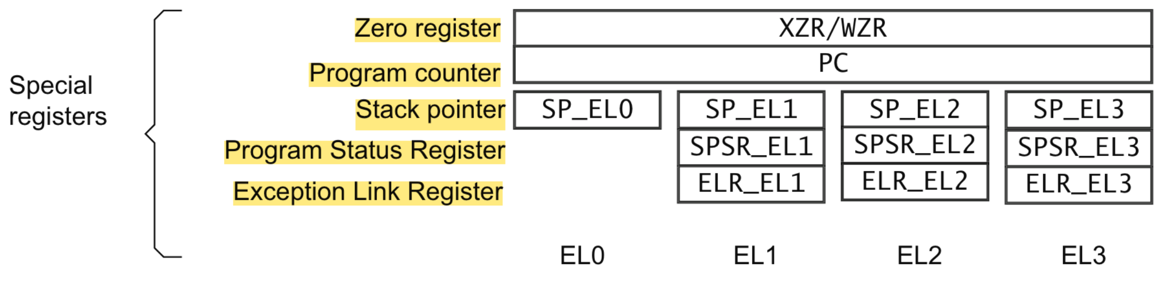

Figure 4-3 AArch64 special registers

When accessing the zero register, all writes are ignored and all reads return 0. The exception return state is held in the following dedicated registers for each Exception level:

- Exception Link Register (ELR).
- Saved Processor State Register (SPSR).

[G 4.1.2] Stack pointer

There is a dedicated SP per Exception level. The choice of the stack pointer to use is separated to some extent from the Exception level. By default, taking an exception selects the stack pointer for the target Exception level, SP_ELn. For example, taking an exception to EL1 selects SP_EL1. Each Exception level has its own stack pointer, SP_EL0, SP_EL1, SP_EL2, and SP_EL3.

At an Exception level other than EL0, the processor can use either:

- A dedicated 64-bit stack pointer associated with that Exception level (SP_ELn).
- The stack pointer associated with EL0 (SP_EL0).

EL0 can only ever access SP_EL0.

The SP cannot be referenced by most instructions. However, some forms of arithmetic instructions, for example, the ADD instruction, can read and write to the current stack pointer to adjust the stack pointer in a function. For example:

```
ADD SP, SP, #0x10 // Adjust SP to be 0x10 bytes before its current
value
```

[1 mark] Changing SP_EL1 must be a privileged instruction if we want to build a trap-and-emulate hypervisor. (True / False)

____

[G 4.1.3] Program Counter

The PC is never accessible as a named register. Its use is implicit in certain instructions such as PC-relative load and address generation. The PC cannot be specified as the destination of a data processing instruction or load instruction.

[1 mark] The processor has one physical PC register that is multiplexed across applications, guest OSes, and the hypervisor. (True / False)

____

[G 4.1.5] Saved Process Status Register (SPSR)

The SPSR holds the PSTATE before taking an exception and is used to restore the PSTATE when executing an exception return. Some bits:

- 31st: Negative flag (N)
- 30th: Zero flag (Z)
- 7th: IRQ mask bit (I)
- 6th: FIQ mask bit (I)
- Bits 3:0: Exception level that an exception was taken from

In ARMv8, the SPSR written to depends on the Exception level. If the exception is taken in EL1, then SPSR_EL1 is used. If the exception is taken in EL2, then SPSR_EL2 is used, and if the exception is taken in EL3, SPSR_EL3 is used. The core populates the SPSR when taking an exception.

[G 4.2] Processor state

You return from an exception by executing the ERET instruction, and this causes the SPSR_ELn to be copied into PSTATE. This restores the ALU flags, execution state, Exception level, and processor branches. From here, you continue execution from the address in ELR_ELn.

[1 mark] EL0 can be allowed to set the {N, Z} bits of SPSR_EL1. (True / False)

____

[1 mark] EL0 can be allowed to set the {I} bit of SPSR_EL1. (True / False)

____

[G 12] The Memory Management Unit

An important function of the Memory Management Unit (MMU) is to enable the system to run multiple tasks, as independent programs running in their own private virtual memory space.

Virtual Addresses are those used by you, the compiler and the linker when placing code in memory. *Physical Addresses* are those used by the actual hardware system. The MMU uses the most significant bits of the Virtual Address to index entries in a *translation table* and establishes which block is being accessed. The MMU translates the Virtual Addresses of code and data to the Physical Addresses in the actual system. The translation is carried out automatically in hardware and is transparent to the application.

[G 12.2] Separation of kernel and application Virtual Address spaces

Operating systems typically have a number of applications or tasks running concurrently. Each of these has its own unique set of translation tables and the kernel switches from one to another as part of the process of switching context between one task and another. However, much of the memory system is used only by the kernel and has fixed virtual to Physical Address mappings where the translation table entries rarely change.
The table base addresses are specified in the Translation Table Base Registers (TTBR0_EL1) and (TTBR1_EL1). The translation table pointed to by TTBR0 is selected when the upper bits of the VA are all 0. TTBR1 is selected when the upper bits of the VA are all set to 1. The Virtual Address from the processor of an instruction fetch or data access is 64 bits.

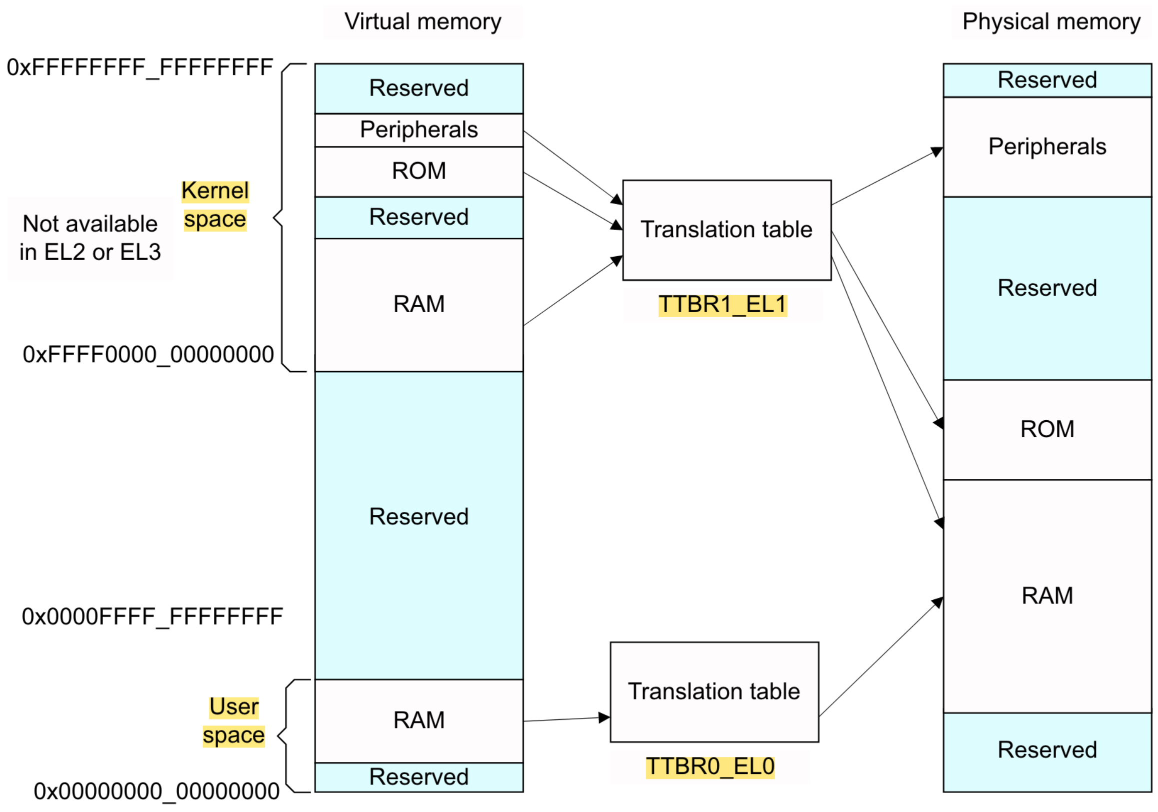

Figure 12-3 Address translation using translation tables

[G 12.3] Translating a Virtual Address to a Physical Address

When the processor issues a 64-bit Virtual Address for an instruction fetch, or data access, the MMU hardware translates the Virtual Address to the corresponding Physical Address. For a Virtual Address, the top 16 bits [63:47] must be all 0s or 1s, otherwise, the address triggers a fault.

The least significant bits are then used to give an offset within the selected section so that the MMU combines the Physical Address bits from the block table entry with the least significant bits from the original address to produce the final address.

The architecture also supports tagged addresses. This is where the most significant eight bits of the address are ignored (treated as not being part of the address). This means that the bits can be used for something else, for example, recording information about a pointer.

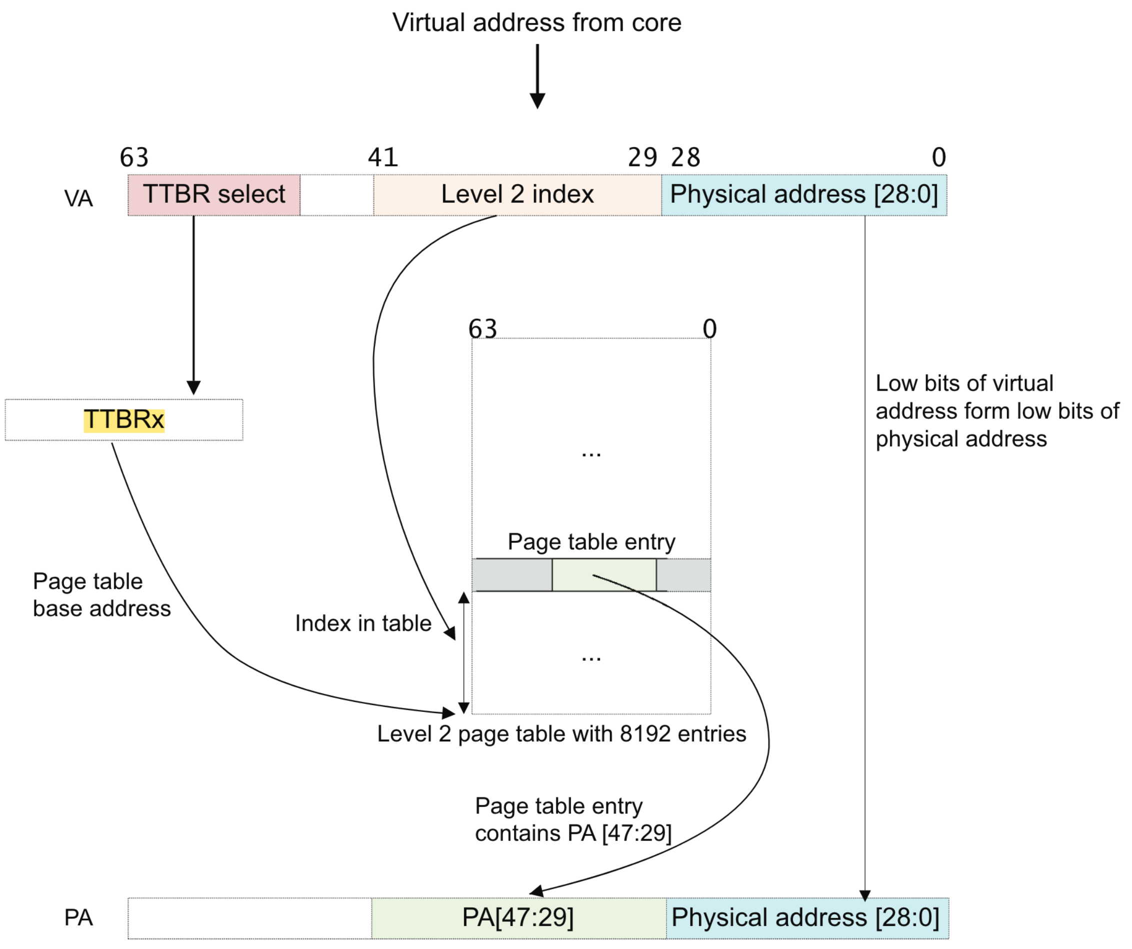

Figure 12-7 Virtual to Physical Address translation for a 512MB block

[5 marks] The above translation scheme has only one level of look-up. It assumes that we have 64KB pages in the page table and each page table entry is 64 bits (=8 bytes). So each page in the page table is storing 8192 entries. It is translating to 512MB blocks (physical frames). Redraw the above figure if we were translating the virtual address to 16KB pages. Pages in the page table are also 16KB. Assume that PTEs are still 64 bits.

[1 mark] Translating to larger pages uses less number of TLB entries. (True / False)

____

[G 12.5.1] Virtual Address tagging

The Translation Control Register, TCR_ELn has an additional field called Top Byte Ignore (TBI) that provides tagged addressing support. General-purpose registers are 64 bits wide, but the most significant 16 bits of an address must be all 0xFFFF or 0x0000. Any attempt to use a different bit value triggers a fault.

When tagged addressing support is enabled, the top eight bits, that is [63:56] of the Virtual Address are ignored by the processor. The top eight bits of a Virtual Address can then be used to pass data. These bits are ignored for addressing and translation faults. The TCR_EL1 has separate enable bits for EL0 and EL1. ARM does not specify or mandate a specific use case for tagged addressing.

[3 marks] Describe how Rust's RC might leverage ARM's virtual address tagging.

[4 marks] What are the pros and cons of using virtual address tagging for RC?

[G 12.6] Translations at EL2 and EL3

The virtualization extensions to the ARMv8-A architecture introduce the second stage of translation. When a hypervisor is present in the system, one or more guest operating systems might be present. These continue to use TTBRn_EL1 as previously described and MMU operation appears unchanged.

The hypervisor must perform some extra translation steps in a two-stage process to share the physical memory system between the different guest operating systems. In the first stage, a Virtual Address (VA) is translated into an Intermediate Physical Address (IPA). This is usually under OS control. A second stage, controlled by the hypervisor, then performs translation of the IPA to the final Physical Address (PA).

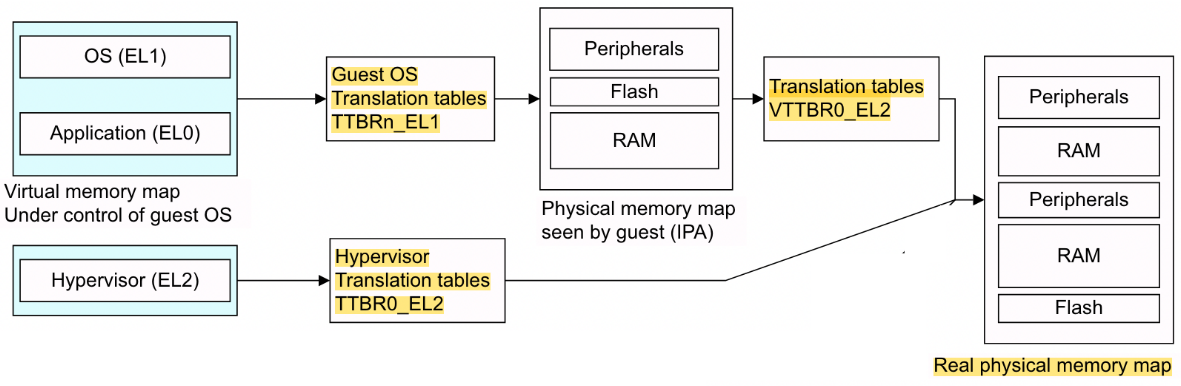

Figure 12-15 Two stage translation process

The stage 2 translations, which convert an intermediate physical address to a Physical Address, use an extra set of tables under the control of the hypervisor. These must be explicitly enabled by writing to the Hypervisor Configuration Register HCR_EL2. This process only applies to EL1/0 accesses. The base address of this stage 2 translation table is specified in the Virtualization Translation Table Base Register VTTBR0_EL2.

It is also possible to set stage 1 translation table for hypervisor's code and data by setting TTBR0_EL2.

For the following problems, assume that we are maintaining the following address translations. For simplicity, you can assume just two levels of translation tables in each stage. Feel free to define new address mapping as you see fit.

Virtual address (GVA)	Intermediate physical address (GPA)	Physical address (HPA)
0x1000	0x3000	0x5000
0x2000	0x4000	0x6000

[4 marks] Compare the number of memory references required to service a TLB miss in the two stages of address translations vs in a single stage of address translations. Give a walkthrough using an example.

Nested virtualization

In this problem, we wish to support *nested virtualization* on ARM architecture, i.e, run hypervisors on top of another hypervisor (which we call *ultravisor* to avoid confusion). This can be useful in the following scenario: AWS rents its servers to Uber, and then Uber further rents to its internal services. This allows Uber hypervisor to monitor each service and do its own load balancing by migrating Guest OSes. For this problem, we assume that the Ultravisor is running in EL3, as shown in the following schematic.

Rental service	Payment service	Account service	Photos service	EL0
Guest OS 1	Guest OS 2	Guest OS 3	Guest OS 4	EL1
Hypervisor-1 (Uber)		Hypervisor-2 (Snapchat)		EL2
Ultravisor (AWS)				EL3

Name: ____________________________ Entry no: ___________________

[8 marks] ARM MMU only supports two stages of address translation whereas we would like to support three stages: Guest virtual address (GVA) -> Guest physical address (GPA) -> Hypervisor physical address (HPA) -> Ultravisor physical address (UPA). Describe how we can support this without modifying the guest OS or the hypervisor and without adding another address translation stage to the hardware. Assume that the hypervisor was already using VTTBR0_EL2 to do 2 stages of address translation. Feel free to make reasonable assumptions on which instructions shall trap to EL3 to support good performance for your translation scheme. Discuss pros and cons of your approach.

[7 marks] How does your translation scheme handle the case where the hypervisor writes to HCR_EL2 to disable two-stage address translation. In other words, hypervisor made GPA = HPA by writing to HCR_EL2.

Name: ____________________________ Entry no: ___________________

Rough sheet