Exam

Q1) - Programming
on mutex, semaphore + condition variables
- general OS questions

Q2) Virtual memory + notes on linux + MMU

Q3) - Scheduling
- Basic algorithms (Round robin, First come, first served, SJF)
↳ shortest job first
- linux schedulers
O(n), O(1) + CFS (uses red black tree)
See notes

Q1

condition variable - used to synchronize between processes

wait(condition, mutex) -> put the to sleep until it gets a signal, releases mutex temporarily

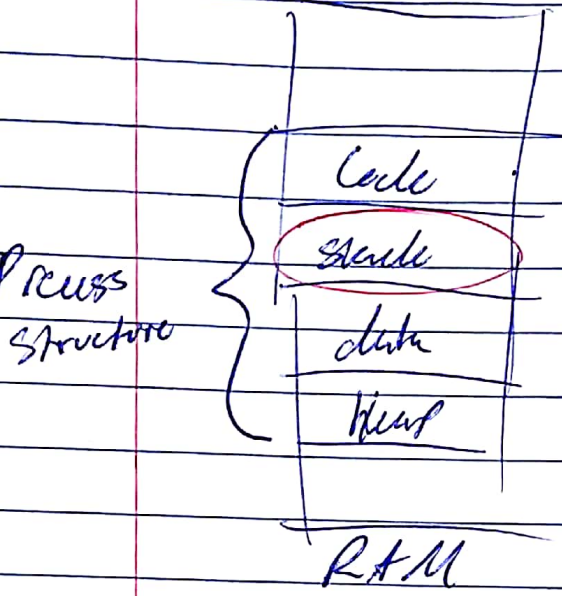
signal(condition) -> send signal to wake a sleeping process

Producer

```
lock(mutex)
if (flag == false)
    wait(condition, mutex)
process data
up mutex
```

Process

Q1

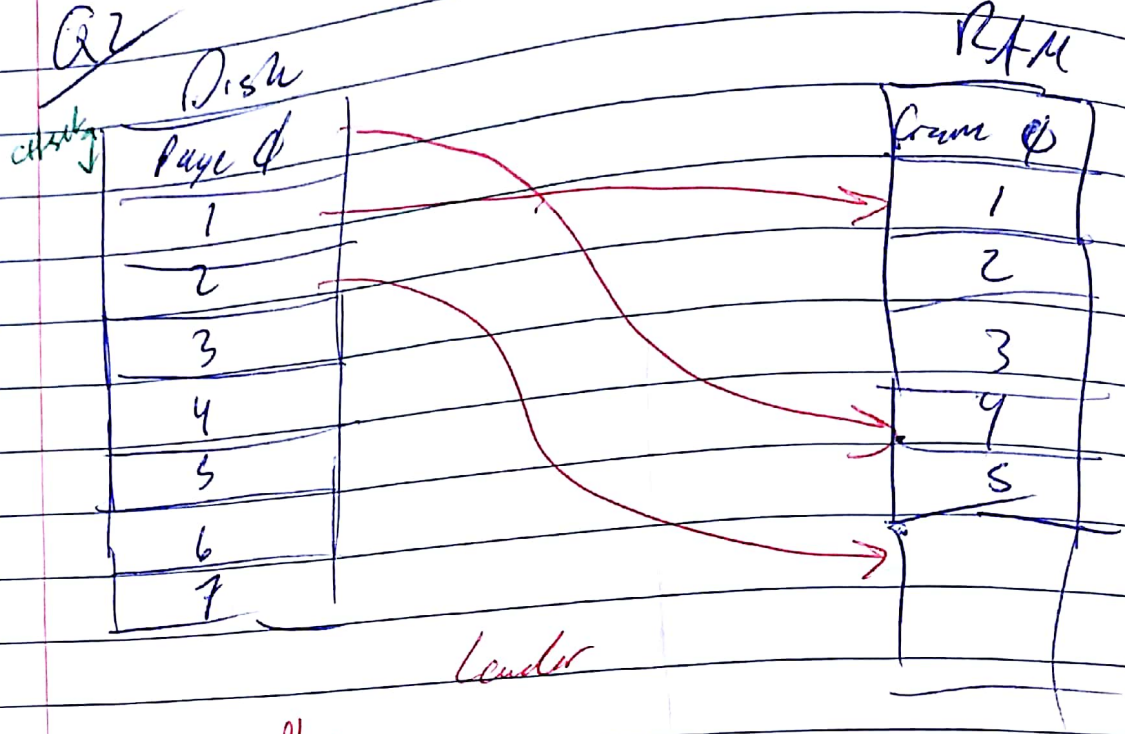


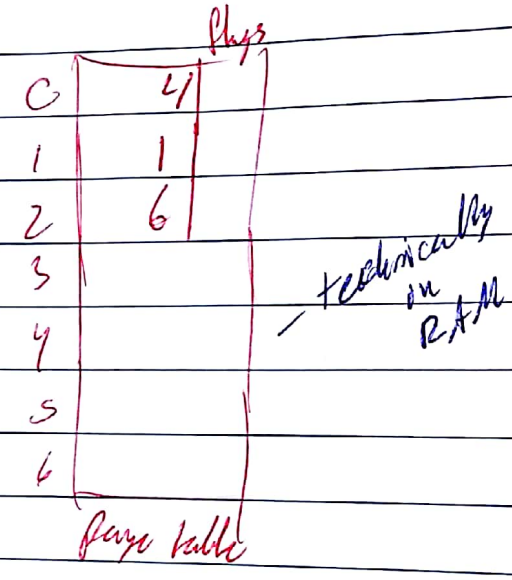
- User task calls system call eg read from disk
OS kernel runs code on behalf of user task
→ code runs in kernel mode.
When running in kernel mode OS switches to kernel stack from user stack

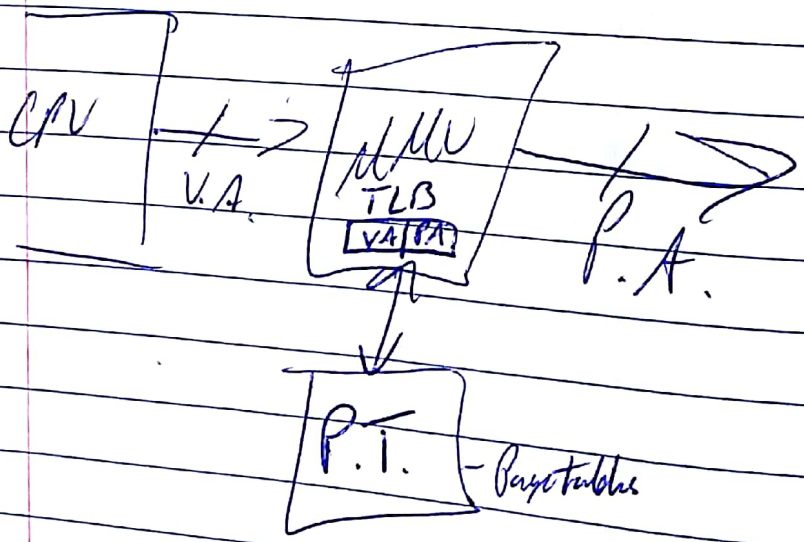
Need ~~to~~ switch to kernel stack because can't depend on user stack being correctly configured
If not → kernel code will crash

Each process has its own user + kernel stack
ie kernel stack is not shared between processes
∴ Makes it easy to switch between tasks.
Just save registers of current task in TCB (including kernel stack pointer) & related registers from TCB for new task.
Don't have to worry about kernel stack being changed while task is ~~sleeping~~ by another task

Q2







Cache = address / data pairs

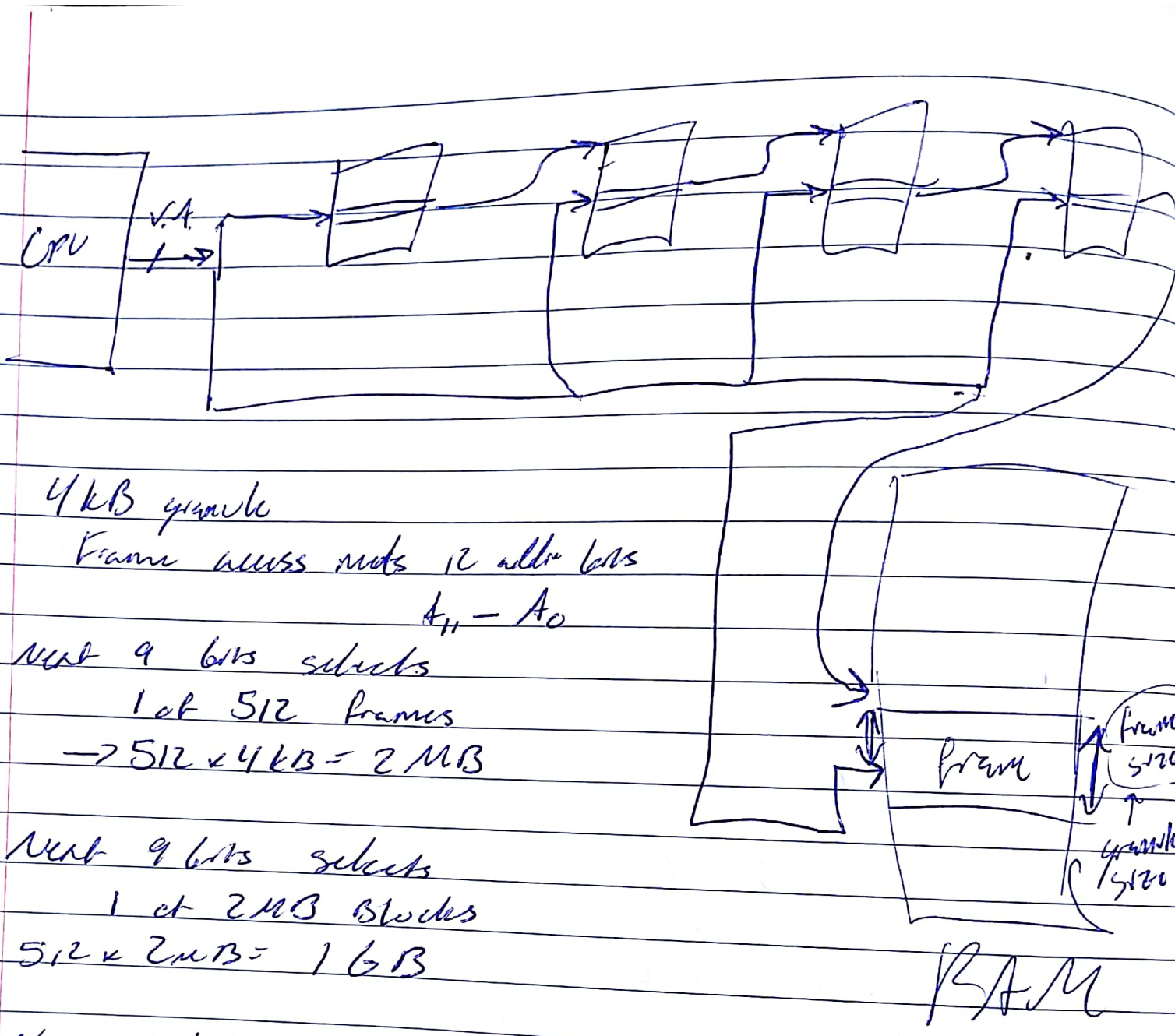
physically tagged i.e store P.A

Tag address	Cache line data

TLB options

① when switching tasks flush TLB
i.e all V.A. entries in TLB refer to V.A. space of 1 task

② Share TLB between tasks
→ More efficient on context switch
eg V.A. 0x ECCC
multiple entries in TLB for this address, each owned by different task



4 KB granule

Frame access needs 12 addr bits

$A_{11} - A_0$

Next 9 bits selects

1 of 512 frames

→ 512 × 4 KB = 2 MB

Next 9 bits selects

1 of 2 MB Blocks

512 × 2 MB = 1 GB

Next 9 bits selects

1 of ~~1 GB~~ 512 GB blocks

→ 512 × 1 GB = 512 GB

2 What is memory management?

Memory management describes how access to memory in a system is controlled. The hardware performs memory management every time that memory is accessed by either the OS or applications. Memory management is a way of dynamically allocating regions of memory to applications.

2.1. Why is memory management needed?

Application processors are designed to run a rich OS, such as Linux, and to support virtual memory systems. Software that executes on the processor only sees virtual addresses, which the processor translates into physical addresses. These physical addresses are presented to the memory system and point to the actual physical locations in memory.

3 Virtual and physical addresses

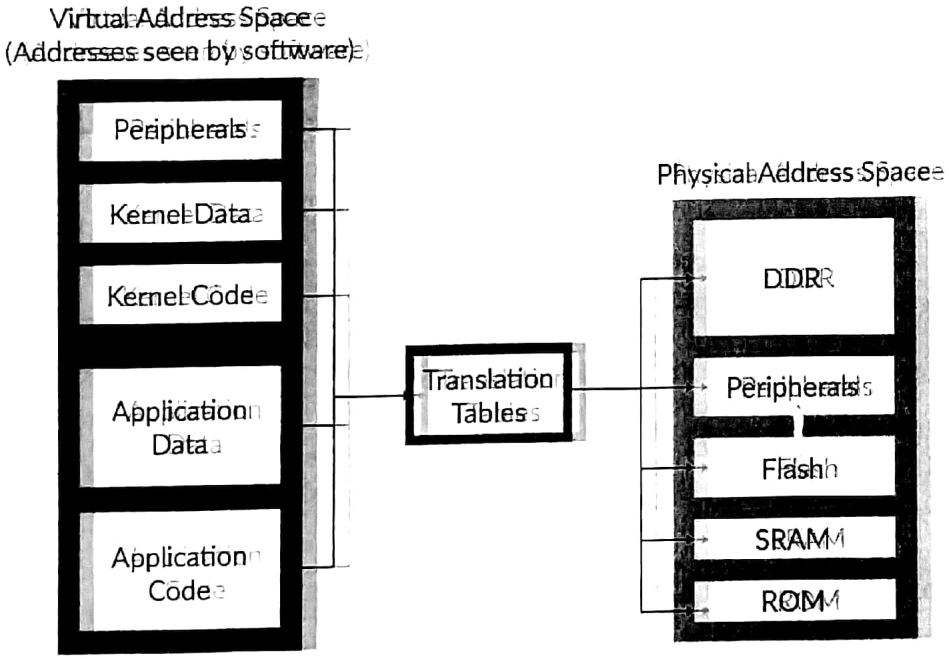
The benefit of using virtual addresses is that it allows management software, such as an *Operating System* (OS), to control the view of memory that is presented to software. The OS can control what memory is visible, the virtual address at which that memory is visible, and what accesses are permitted to that memory. This allows the OS to sandbox applications (hiding the resources of one application from another application) and to provide abstraction from the underlying hardware.

One benefit of using virtual addresses is that an OS can present multiple fragmented physical regions of memory as a single, contiguous virtual address space to an application.

Virtual addresses also benefit software developers, who will not know a system's exact memory addresses when writing their application. With virtual addresses, software developers do not need to concern themselves with the physical memory. The application knows that it is up to the OS and the hardware to work together to perform the address translation.

In practice, each application can use its own set of virtual addresses that will be mapped to different locations in the physical system. As the operating system switches between different applications it re-programs the map. This means that the virtual addresses for the current application will map to the correct physical location in memory.

Virtual addresses are translated to physical addresses through mappings. The mappings between virtual addresses and physical addresses are stored in translation tables (sometimes referred to as page tables) as this diagram shows:



Translation tables are in memory and are managed by software, typically an OS or hypervisor. The translations tables are not static, and the tables can be updated as the needs of software change. This changes the mapping between virtual and physical addresses.

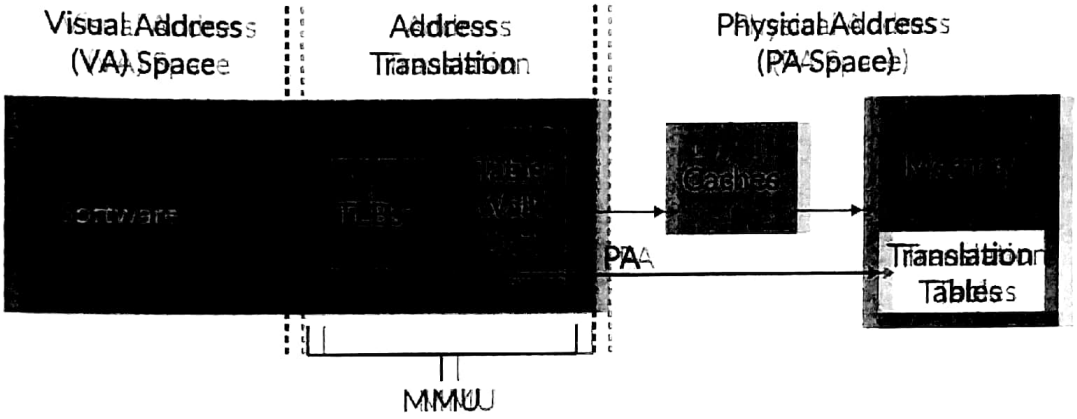
4 The Memory Management Unit (MMU)

The *Memory Management Unit* (MMU) performs translations.

The MMU contains the following:

- The table walk unit, which contains logic that reads the translation tables from memory.
- *Translation Lookaside Buffers* (TLBs), which cache recently used translations.

All memory addresses that are issued by software are virtual. These memory addresses are passed to the MMU, which checks the TLBs for a recently used cached translation. If the MMU does not find a recently cached translation, the table walk unit reads the appropriate table entry, or entries, from memory, as shown here:



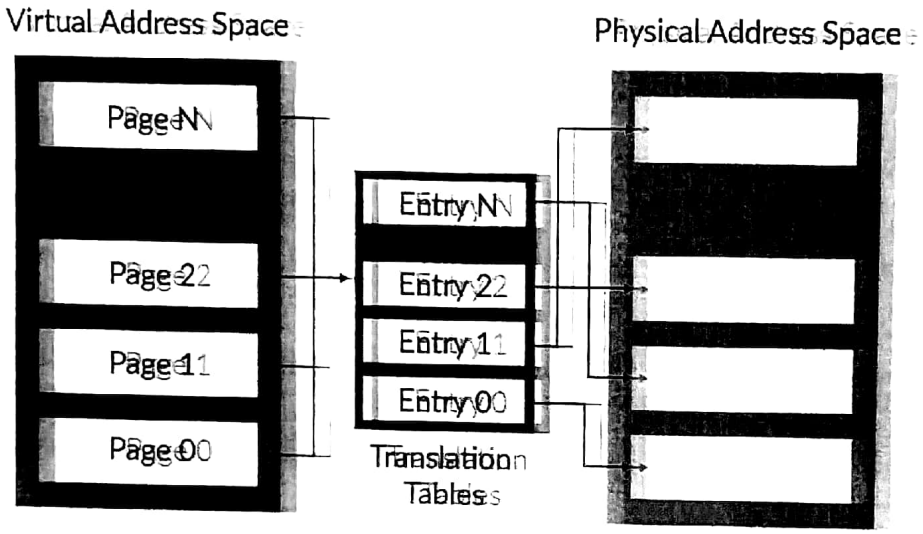
A virtual address must be translated to a physical address before a memory access can take place (because we must know which physical memory location we are accessing). This need for translation also applies to cached data, because on Armv6 and later processors, the data caches store data using the physical address (addresses that are physically tagged). Therefore, the address must be translated before a cache lookup can complete.

Note: Architecture is a behavioral specification. The caches must behave as if they are physically tagged. An implementation might do something different, as long as this is not software-visible.

4.1. Table entry

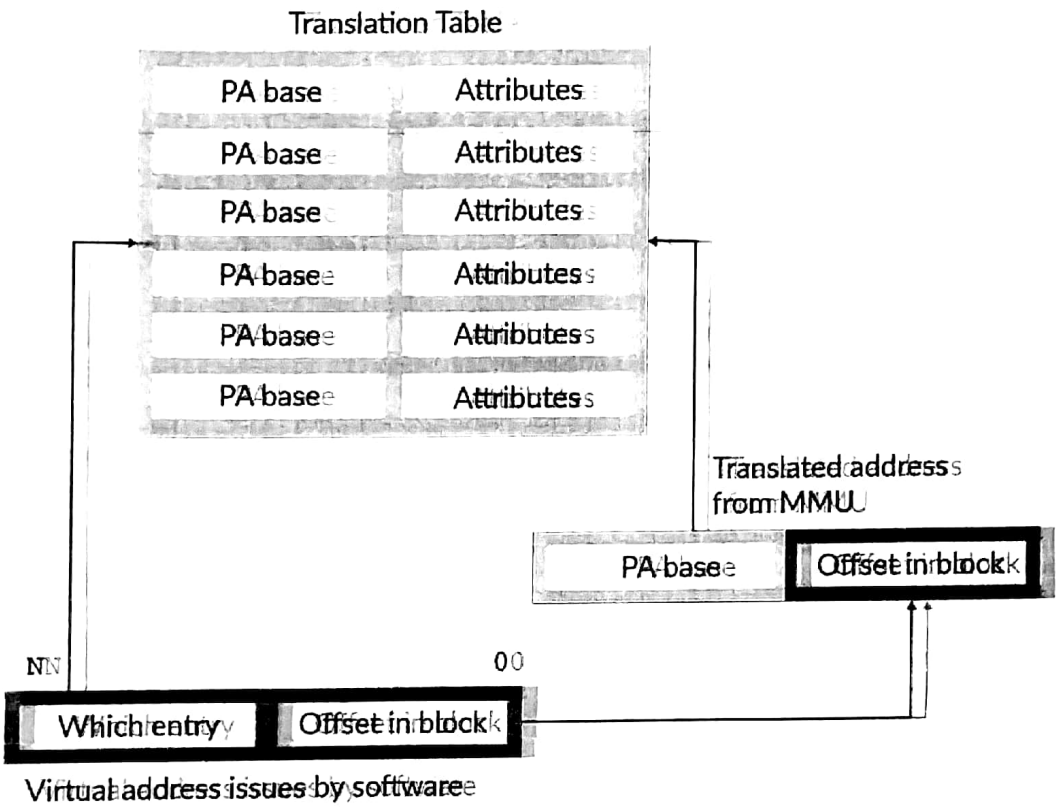
The translation tables work by dividing the virtual address space into equal-sized blocks and by providing one entry in the table per block.

Entry 0 in the table provides the mapping for block 0, entry 1 provides the mapping for block 1, and so on. Each entry contains the address of a corresponding block of physical memory and the attributes to use when accessing the physical address.



4.2 Table lookup

A table lookup occurs when a translation takes place. When a translation happens, the virtual address that is issued by the software is split in two, as shown in this diagram:



This diagram shows a single-level lookup.

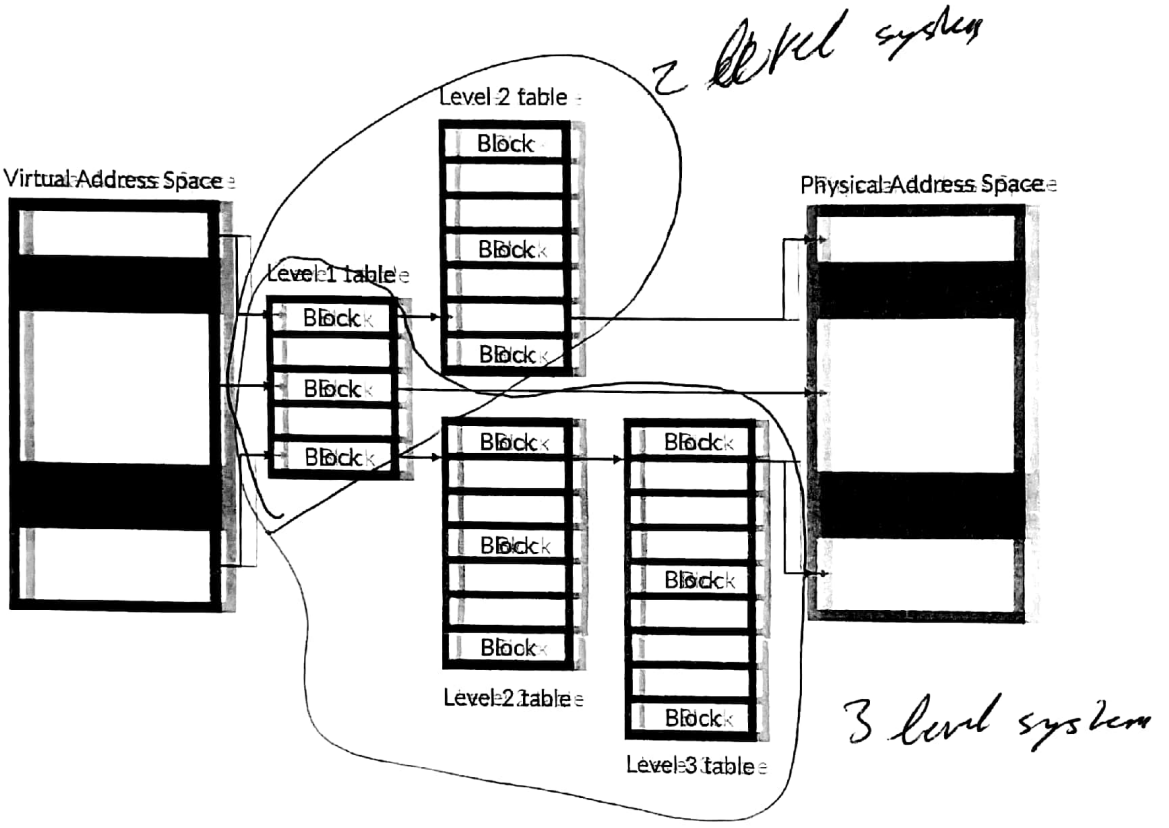
The upper-order bits, which are labeled 'Which entry' in the diagram, tell you which block entry to look in and they are used as an index into the table. This entry block contains the physical address for the virtual address.

The lower-order bits, which are labeled 'Offset in block' in the diagram, are an offset within that block and are not changed by the translation.

4.3 Multilevel translation

In a single-level lookup, the virtual address space is split into equal-sized blocks. In practice, a hierarchy of tables is used.

The first table (Level 1 table) divides the virtual address space into large blocks. Each entry in this table can point to an equal-sized block of physical memory or it can point to another table which subdivides the block into smaller blocks. We call this type of table a 'multilevel table'. Here we can see an example of a multilevel table that has three levels:



In Armv8-A, the maximum number of levels is four, and the levels are numbered 0 to 3. This multilevel approach allows both larger blocks and smaller blocks to be described. The characteristics of large and small blocks are as follows:

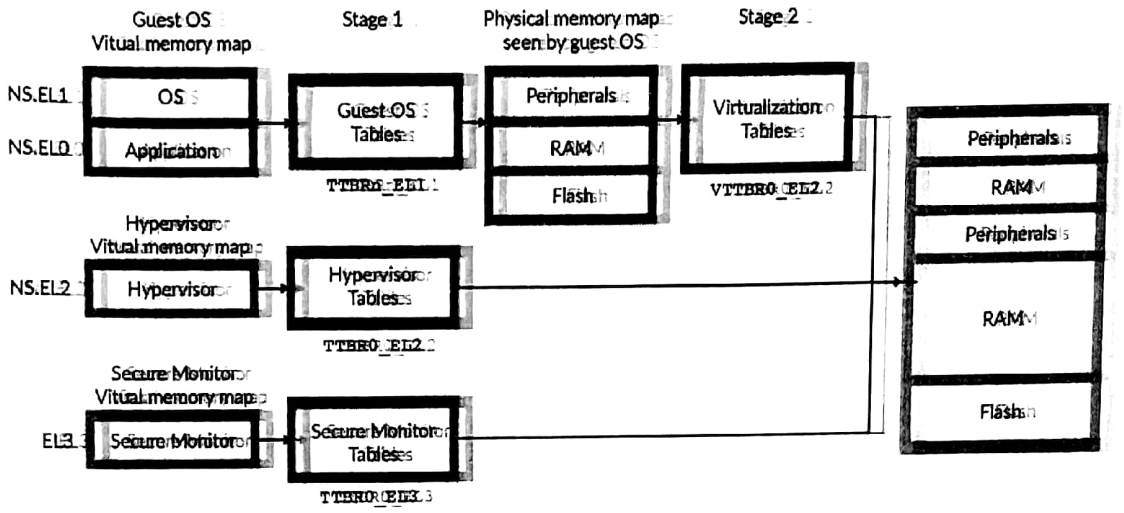
- Large blocks require fewer levels of reads to translate than small blocks. Plus, large blocks are more efficient to cache in the TLBs.
- Small blocks give software fine-grain control over memory allocation. However, small blocks are less efficient to cache in the TLBs. Caching is less efficient because small blocks require multiple reads through the levels to translate.

To manage this trade-off, an OS must balance the efficiency of using large mappings against the flexibility of using smaller mappings for optimum performance.

Note: The processor does not know the size of the translation when it starts the table lookup. The processor works out the size of the block that is being translated by performing the table walk.

5 Address spaces in Armv8-A

There are several independent virtual address spaces in Armv8-A. This diagram shows these virtual address spaces:



The diagram shows three virtual address spaces:

- NS.EL0 and NS.EL1 (Non-secure EL0/EL1).
- NS.EL2 (Non-secure EL2).
- EL3.

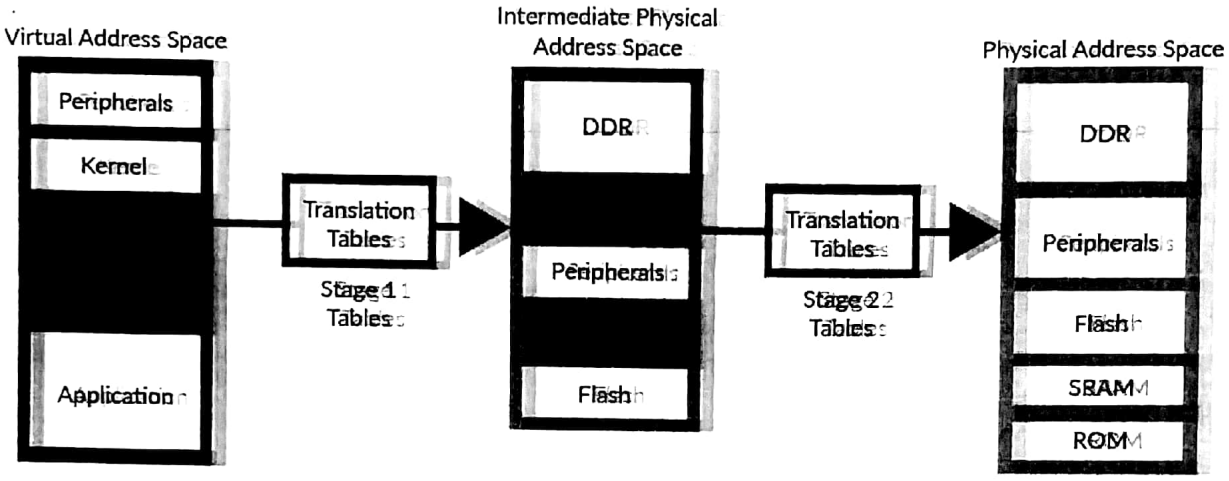
Each of these virtual address spaces is independent, and has its own settings and tables. We often call these settings and tables 'translation regimes'. There are also virtual address spaces for Secure EL0, Secure EL1 and Secure EL2, but they are not shown in the diagram.

Note: Support for Secure EL2 was added in Armv8.4-A.

Because there are multiple virtual address spaces, it is important to specify which address space an address is in. For example, NS.EL2:0x8000 refers to the address 0x8000 in the Non-secure EL2 virtual address space.

The diagram also shows that the virtual addresses from Non-secure EL0 and Non-secure EL1 go through two sets of tables. These tables support virtualization and allow the hypervisor to virtualize the view of physical memory that is seen by a virtual machine (VM).

In virtualization, we call the set of translations that are controlled by the OS, Stage 1. The Stage 1 tables translate virtual addresses to *intermediate physical addresses* (IPAs). In Stage 1 the OS thinks that the IPAs are physical address spaces. However, the hypervisor controls a second set of translations, which we call Stage 2. This second set of translations translates IPAs to physical addresses. This diagram shows how the two sets of translations work:



Although there are some minor differences in the table format, the process of Stage 1 and Stage 2 translation is usually the same.

Note: At Arm, we use the address 0x8000 in many of our examples. 0x8000 is also the default address for linking with the Arm linker, armlink. The address comes from an early microcomputer, the BBC Micro Model B, which had ROM (and sideways RAM) at the address 0x8000. The BBC Micro Model B was built by a company called Acorn, which developed the Acorn RISC Machine (ARM), and later became Arm.

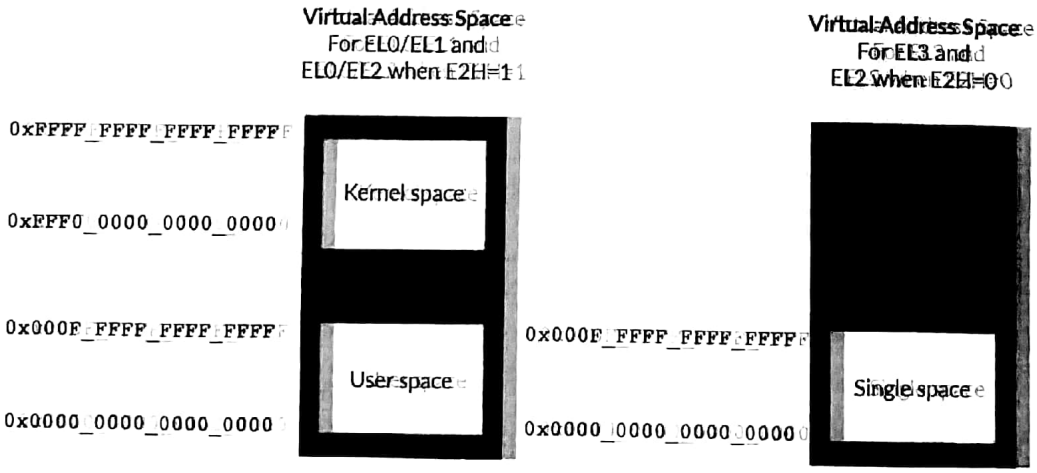
5.1. Address sizes

Armv8-A is a 64-bit architecture, but this does not mean that all addresses are 64-bit.

5.1.1 Size of virtual addresses

Virtual addresses are stored in a 64-bit format. As a result, the address in load instructions (LDR) and store instructions (STR) is always specified in an X register. However, not all of the addresses in the X register are valid.

This diagram shows the layout of the virtual address space in AArch64:

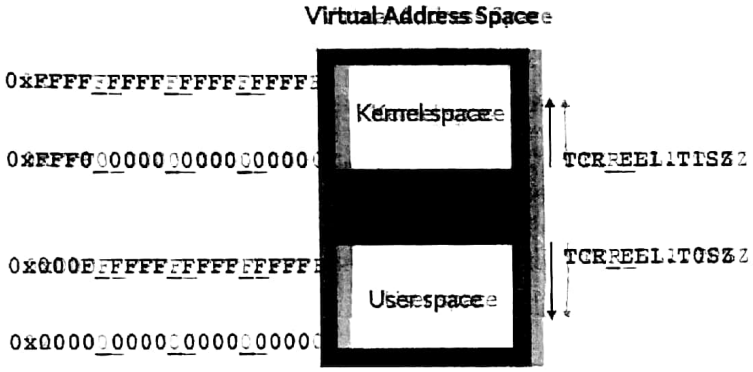


There are two regions for the EL0/EL1 virtual address space: kernel space and application space. These two regions are shown on the left-hand side of the diagram, with kernel space at the top, and application space, which is labeled 'User space', at the bottom of the address space. Kernel space and user space have separate translation tables and this means that their mappings can be kept separate.

There is a single region at the bottom of the address space for all other Exception levels. This region is shown on the right-hand side of the diagram and is the box with no text in it.

Note: If you set HCR_EL2.E2H to 1 it enables a configuration where a host OS runs in EL2, and the applications of the host OS run in EL0. In this scenario, EL2 also has an upper and a lower region.

Each region of address space has a size of up to 252 bytes. However, each region can be independently shrunk to a smaller size. The TnSZ fields in the TCR_ELx registers control the size of the virtual address space. For example, this diagram shows that TCR_EL1 controls the EL0/EL1 virtual address space:



The virtual address size is encoded as:

$$\text{virtual address size in bytes} = 2^{64-\text{TCR\_ELx.TnSZ}}$$

The virtual address size can also be expressed as a number of address bits:

$$\text{Number of address bits} = 64 - \text{TnSZ}$$

Therefore, if TCR_EL1.SZ1 is set to 32, the size of the kernel region in the EL0/EL1 virtual address space is 2^{32} bytes (0xFFFF_FFFF_0000_0000 to 0xFFFF_FFFF_FFFF_FFFF). Any address that is outside of the configured range or ranges will, when it is accessed, generate an exception as a translation fault. The advantage of this configuration is that we only need to describe as much of the address space as we want to use, which saves time and space. For example, imagine that the OS kernel needs 1GB of address space (30-bit address size) for its kernel space. If the OS sets T1SZ to 34, then only the translation table entries to describe 1GB are created, as 64 - 34 = 30.

Note: All Armv8-A implementations support 48-bit virtual addresses. Support for 52-bit virtual addresses is optional and reported by ID_AA64MMFR2_EL1. At the time of writing, none of the Arm Cortex-A processors support 52-bit virtual addresses.

5.1.2 Size of physical addresses

The size of a physical address is IMPLEMENTATION DEFINED, up to a maximum of 52 bits. The ID_AA64MMFR0_EL1 register reports the size that is implemented by the processor. For Arm Cortex-A processors, this will usually be 40 bits or 44 bits.

Note: In Armv8.0-A, the maximum size for a physical address is 48 bits. This was extended to 52 bits in Armv8.2-A.

5.1.3 Size of intermediate physical addresses

If you specify an output address in a translation table entry that is larger than the implemented maximum, the *Memory Management Unit* (MMU) will generate an exception as an address size fault.

The size of the IPA space can be configured in the same way as the virtual address space. VTCR_EL2.T0SZ controls the size. The maximum size that can be configured is the same as the physical address size that is supported by the processor. This means that you cannot configure a larger IPA space than the supported physical address space.

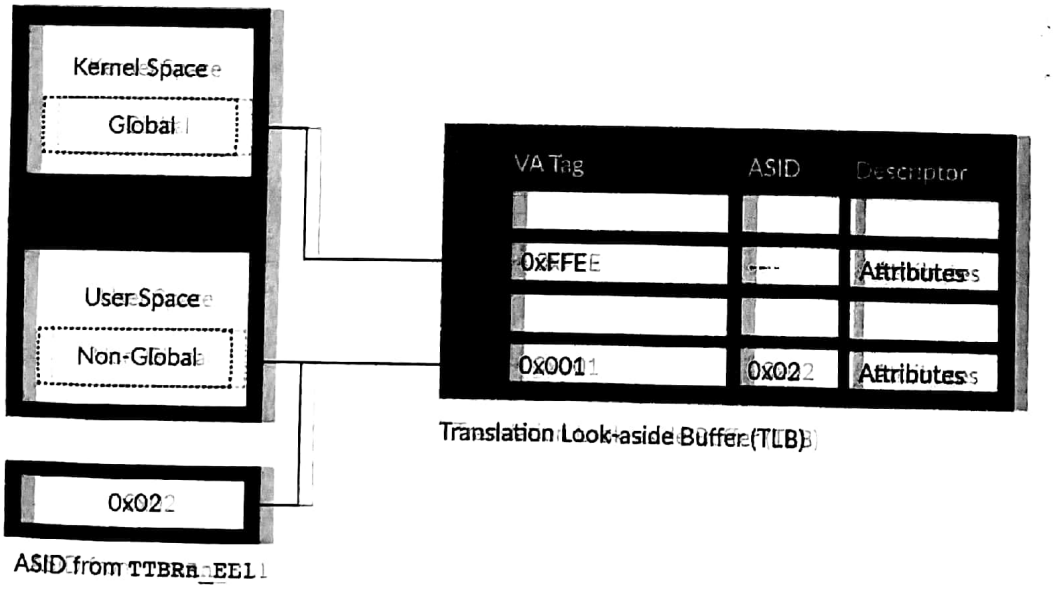
5.2 Address Space Identifiers - Tagging translations with the owning process

Many modern OSs have all applications that seem to run from the same address region, this is what we have described as user space. In practice, different applications require different mappings. This means, for example, that the translation for VA 0x8000 depends on which application is currently running.

Ideally, we would like the translations for different applications to coexist within the *Translation Lookaside Buffers* (TLBs), to prevent the need for TLB invalidates on a context switch. But how would the processor know which version of the VA 0x8000 translation to use? In Armv8-A, the answer is *Address Space Identifiers* (ASIDs).

For the EL0/EL1 virtual address space, translations can be marked as Global (G) or Non-Global (nG) using the nG bit in the attributes field of the translation table entry. For example, kernel mappings are Global translations, and application mappings are Non-Global translations. Global translations apply whichever application is currently running. Non-Global translations only apply with a specific application.

Non-Global mappings are tagged with an ASID in the TLBs. On a TLB lookup, the ASID in the TLB entry is compared with the currently selected ASID. If they do not match, then the TLB entry is not used. This diagram shows a Global mapping in the kernel space with no ASID tag and a non-Global mapping in user space with an ASID tag:



The diagram shows that TLB entries for multiple applications are allowed to coexist in the cache, and the ASID determines which entry to use.

The ASID is stored in one of the two TTBRn_EL1 registers. Usually TTBR0_EL1 is used for user space. As a result, a single register update can change both the ASID and the translation table that it points to.

Note: ASID tagging is also available in EL2, when HCR_EL2.E2H==1.

5.3. Virtual Machine Identifiers - Tagging translations with the owning VM

EL0/EL1 translations can also be tagged with a *Virtual Machine Identifier* (VMID). VMIDs allow translations from different VMs to coexist in the cache. This is similar to the way in which ASIDs work for translations from different applications. In practice, this means that some translations will be tagged with both a VMID and an ASID, and that both must match for the TLB entry to be used.

Note: When virtualization is supported for a security state, EL0/EL1 translations are always tagged with a VMID – even if Stage 2 translation is not enabled. This means that if you are writing initialization code and are not using a hypervisor, it is important to set a known VMID value before setting up the Stage 1 MMU.

5.4. Common not Private

If a system includes multiple processors, do the ASIDs and VMIDs used on one processor have the same meaning on other processors?

For Armv8.0-A the answer is that they do not have to mean the same thing. There is no requirement for software to use a given ASID in the same way across multiple processors. For example, ASID 5 might be used by the calculator on one processor and by the web browser on another processor. This means that a TLB entry that is created by one processor cannot be used by another processor.

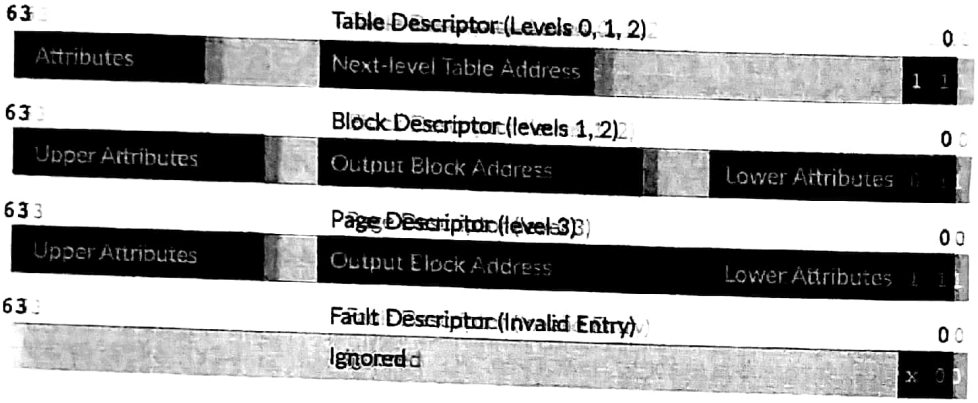
In practice, it is unlikely that software will use ASIDs differently across processors. It is more common for software to use ASIDs and VMIDs in the same way on all processors in a given system. Therefore, Armv8.2-A introduced the *Common not Private* (CnP) bit in the *Translation Table Base Register* (TTBR). When the CnP bit is set, the software promises to use the ASIDs and VMIDs in the same way on all processors, which allows the TLB entries that are created by one processor to be used by another.

Note: We have been talking about processors, however, technically, we should be using the term, *Processing Element* (PE). PE is a generic term for any machine that implements the Arm architecture. It is important here because there are microarchitectural reasons why sharing TLBs between processors would be difficult. But within a multithreaded processor, where each hardware thread is a PE, it is much more desirable to share TLB entries.

6 Controlling address translation

6.1. Translation table format

Here we can see the different formats that are allowed for translation table entries:



Each entry is 64 bits and the bottom two bits determine the type of entry.

Notice that some of the table entries are only valid at specific levels. The maximum number of levels of tables is four, which is why there is no table descriptor for level 3 (or the fourth level), tables. Similarly, there are no Block descriptors or Page descriptors for level 0. Because level 0 entry covers a large region of virtual address space, it does not make sense to allow blocks.

Note: The encoding for the Table descriptor at levels 0-2 is the same as the Page descriptor at level 3. This encoding allows 'recursive tables', which point back to themselves. This is useful because it makes it easy to calculate the virtual address of a particular page table entry so that it can be updated.

7 Translation granule

A translation granule is the smallest block of memory that can be described. Nothing smaller can be described, only larger blocks, which are multiples of the granule.

Armv8-A supports three different granule sizes: 4KB, 16KB, and 64KB.

The granule sizes that a processor supports are IMPLEMENTATION DEFINED and are reported by ID_AA64MMFR0_EL1. All Arm Cortex-A processors support 4KB and 64KB. The selected granule is the smallest block that can be described in the latest level table. Larger blocks can also be described. This table shows the different block sizes for each level of table based on the selected granule:

Level of table	4KB granule		16KB granule		64KB granule	
	Size per entry	Bits used to index	Size per entry	Bits used to index	Size per entry	Bits used to index
0	512GB	47:39*	128TB	47*	-	-
1	1GB	38:30	64GB	46:36	4TB	51:42
2	2MB	29:21	32MB	35:25	512MB	41:29
3	4KB	20:12	16KB	24:14	64KB	28:16

* There are restrictions on using 52-bit addresses. When the selected granule is 4KB or 16KB, the maximum virtual address region size is 48 bits. Similarly, output physical addresses are limited to 48 bits. It is only when the 64KB granule is used that the full 52 bits can be used.

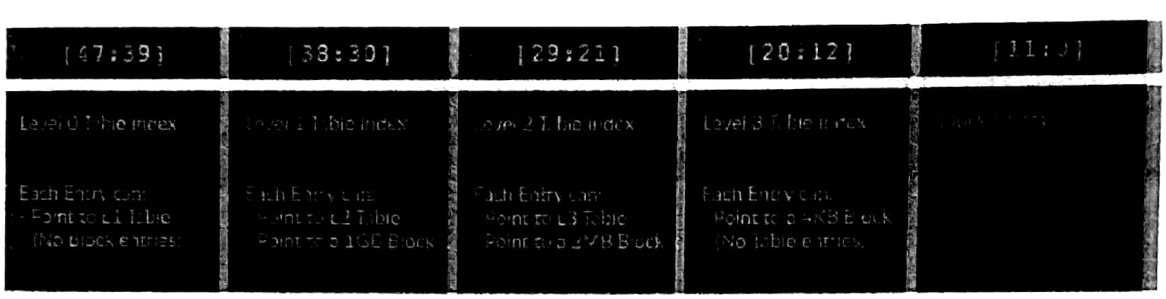
Note: TCR_EL1 has two separate fields that control the granule size for the kernel space and the user space virtual address ranges. These fields are called TG1 for kernel space and TG0 for user space. A potential problem for programmers is that these two fields have different encodings.

7.1. The starting level of address translation

Together, the granule and the size of the virtual address space control the starting level of address translation.

The previous table summarized the block size (size of virtual address range covered by a single entry) for each granule at each level of table. From the block size, you can work out which bits of the virtual address are used to index each level of table.

Let us take the 4KB granule as an example. This diagram shows the bits that are used to index the different levels of table for a 4KB granule:



Imagine that, for a configuration, you set the size of the virtual address space, TCR_EEx.T0SZ, to 32. Then the size of the virtual address space, in address bits, is calculated as:

64 - T0SZ = 32-bit address space (address bits 31:0)

If we look at the previous 4KB granule diagram again, level 0 is indexed by bits 47:39. With a 32-bit address space you do not have these bits. Therefore, the starting level of translation for your configuration is level 1.

Next, imagine you set T0SZ to 34:

64 - T0SZ = 30-bit address space (address bits 29:0)

This time, you do not have any other bits that are used to index the level 0 table or the level 1 table, so the starting level of translation for your configuration is level 2.

As the previous diagram shows, when the size of the virtual address space reduces, you need fewer levels of tables to describe it.

These examples are based on using the 4KB granule. The same principle applies when using 16KB and 64KB granules, but the address bits change.

7.2 Registers that control address translation

Address translation is controlled by a combination of system registers:

- SCTLR_ELx
  - M - Enable Memory Management Unit (MMU).
  - C - Enable for data and unified caches.
  - EE - Endianness of translation table walks.
- TTBR0_ELx and TTBR1_ELx
  - BADDR - Physical address (PA) (or intermediate physical address, IPA, for EL0/EL1) of start of translation table.
  - ASID - The Address Space Identifier for Non-Global translations.
- TCR_ELx
  - PS/IPS - Size of PA or IPA space, the maximum output address size.
  - TnSZ - Size of address space covered by table.
  - TGn - Granule size.
  - SH/IRGN/ORGN - Cacheability and shareability to be used by MMU table walks.
  - TBIn - Disabling of table walks to a specific table.
- MAIR_ELx
  - Attr - Controls the Type and cacheability in Stage 1 tables.

7.3 MMU disabled

When the MMU is disabled for a stage of translation, all addresses are flat-mapped. Flat-mapping means that the input and output addresses are the same.