Module no: CS3014
Matriculation no: 200017941

P2 – An analysis of paging

Introduction

The objective of this practical was to demonstrate an understanding of paging. The first part deals with calculations regarding paging and establishes a specific architecture for this report to build on top of. The second part requires us to take a deeper dive into paging and to really look at how exactly a specific architecture is implemented. Lastly, part three is where all the accumulated knowledge is put to the test as the structures of a paging table and how virtual addresses are converted into physical addresses are required to be implemented in the form of a program. I achieved this and ended up replicating the paging process from a slightly higher level. This involved being able to allocate memory for the page table, adding entries into this page table which maps a page number to a frame number and being able to read to and write from virtual addresses using the provided page table and store.

Part 1: Page table layout

Calculations for an imaginary architecture:

Given a 16-bit architecture,

each address has 16 bits; therefore, it can store 2^{16} = 65536 different addresses which gives the size of the RAM.

And since physical address space = RAM, the size of the physical memory space for this architecture is **65536 bytes.**

This also means that the size of the virtual memory space is 2^{16} = **65536 bytes**.

Given 128 bytes for the page size,

physical address space = total number of frames * page size

Therefore, total number of frames = 65536/ 128 = **512 frames.**

And since frame size = page size, each page is **128 bytes**.

Size of page table = number of entries in page table * page table entry size.

Page table entry size = number of bits in frame number + number of bits used for optional fields.

Since there are 512 frames which is equivalent to 2^9 bits, the frame number is **9 bits**.

Therefore, the size of the page table entry = 9 + 2 (for the protection bits) = **11 bits**.

And since size of page number = size of frame number, the page offset = 16 – frame offset = 16 – 9 = **7-bits**

number of entries in page table = $2^{\text{number of bits in the frame number}}$

Therefore, the number of entries in the page table is 2^9 = 512 **entries.**

And the size of the page table is 512 * 11 = 5632 **bits (704 bytes).**

Layout of the page table:

Page offset (d)	Protection bits

An example row:

010010100 (9 bits for the page offset)	0 (1 protection bit for whether it is read-only)	1 (1 protection bit for whether it is executable)

How the address translation works:

A virtual address is an address in virtual memory which is used by a process in main memory to use more memory than is available by storing some contents back in hard disk. This is controlled by the operating system. Physical address is the actual address of the data inside the memory. For processes to be able to execute, virtual addresses need to be translated into physical addresses, which is done by the memory management unit (MMU). It makes use of a page table which stores the mapping from the virtual address to the physical address. This process is shown in the figure 1:

Virtual address
Page table
Physical address

figure 1

The page table consists of several page table entries (PTE). Each of these entries consists of a physical address which is indexed. And it is this indexing that is used to locate the matching physical address for the given virtual address. The virtual address is split into two parts, the page number and the page offset. The page number specifies where in the page table the physical address for the given virtual address is. The offset simply stores the actual value which needs to be transferred to the physical address. Below are a few examples which illustrate how the above steps would take place:

For 0xbeef, translating it to binary would give 1011111011101111. This is what is stored as the virtual address which is assigned by the kernel. This contains the page number and the offset which is where the actual data is stored. Following the above architecture, 10111 (23 in denary) would be the page number and 111011101111 (3823 in denary) would be the page offset. During the CPU execution, the MMU uses the page number to navigate to the appropriate row in the page table to access the physical address and the data is then stored here. This is shown in figure 2:

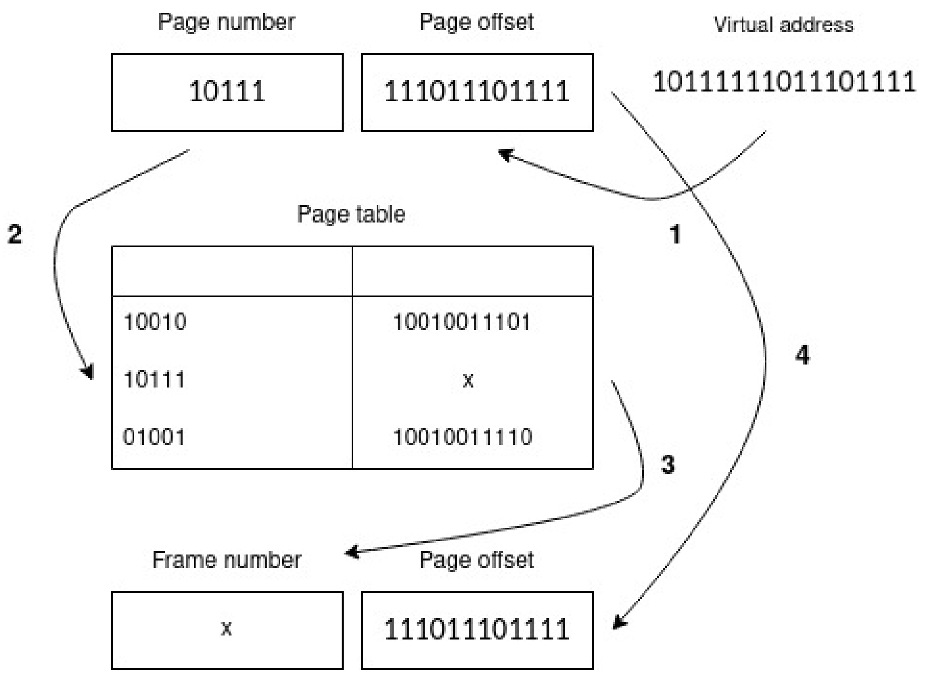


figure 2

For 0xc001, translating it to binary would give 1100000000000001. This is again stored somewhere in the virtual address space. 11000 (24 in denary) is the page number and 00000000001 is the page offset (1 in denary). During execution the MMU takes this address and translates it to a physical address by going to the index in the page table as specified in the page number. This data is then stored in the physical address space where it can be accessed by the process. The steps are shown in figure 3:

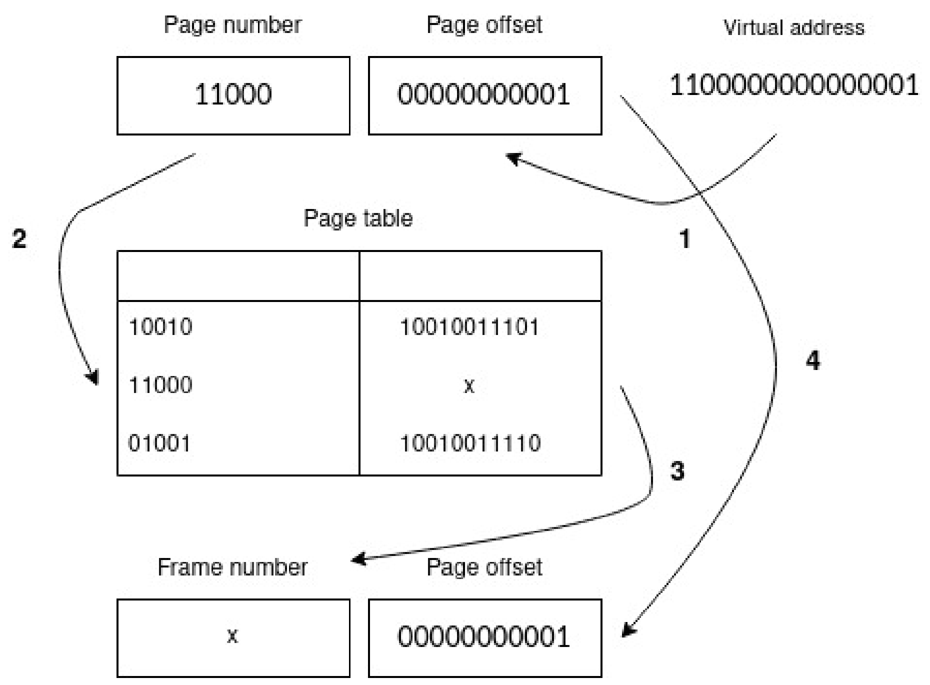


figure 3

For the last five numbers of my student ID (17941), in binary that would be 0100011000010101. This would make the page number 01000 and the page offset 11000010101. The MMU then takes this page number to access the physical address in the page table where it is stored. This is illustrated in figure 4:

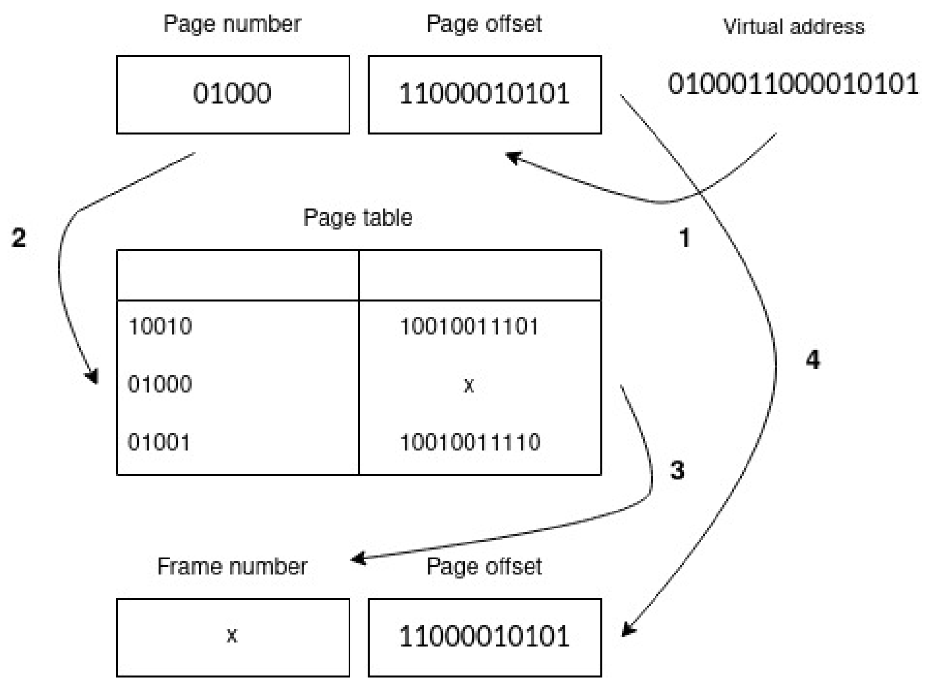


figure 4

Part 2: Research into ARM architecture

ARM stands for Advanced RISC Machines and is a family of reduced instruction set computer (RISC) architectures for processors. These architectures use paging as the memory allocation method. There are several different versions in the ARM architecture family, and each does paging differently, but this essay will only be focused on explaining how paging works on ARMv8-A.

ARMv8-A has two execution states: AArch32 and AArch64. These are 32-bit and 64-bit architectures respectively. This version is the first to extend the architecture to 64 bits but at the same time allowing backwards compatibility with existing 32-bit architectures such as ARMv7-A. The new instruction set is called A64 which uses 64-bit general purpose registers. The result of this is a large physical address space which goes beyond 4GB where each address is 48 bits. And since virtual addresses are stored in 64-bit registers, 64-bit virtual addressing is used. This enables the virtual memory space to go beyond 4GB as well.

During the execution of a process, the instructions can contain a data address. This is usually a virtual address which is stored in virtual memory system and can be in registers such as the program counter. But whenever the data is accessed, it is done in the physical memory system where it has a physical address. Therefore, address translation is needed which maps an input address (the virtual address) to an output address (the physical address). It is important to note that in practice, there are multiple translations involved when mapping a virtual address to a physical address. As a result, each process being executed does not have any knowledge of physical memory and just has its own virtual memory space allocated to it. Programs are written, compiled, linked and run in virtual memory.

For the translation to occur, a translation table is required where each table entry (called a page descriptor) contains the memory mapping. Each process will have its own translation table in their virtual memory space. Each table stores the corresponding physical address for the given virtual address, whether the physical address access is to the secure or non-secure address map, the physical address memory access permissions and the physical address memory region attributes. These tables are accessed by the Memory Management Unit (MMU) which also controls the memory access permissions, memory attribute determination and memory attribute checking. It reads an entry from a translation table and resolves a subset of the virtual address. Multiple translation table entries may be required to resolve a single virtual address as this architecture uses multi-level paging where each level has a translation table. So, during the translation process, the MMU access multiple page tables to get the corresponding physical address. But when the MMU cannot translate a given virtual address, a fault is generated. This can occur anywhere within each of the levels.

Since there can be several processes concurrently and each has its own translation table, the kernel needs to switch between tables. The MMU accesses each table using the base address of a table which is specified in the Translation Table Base Registers and there are two of these: TTBR0 and TTBR1. TTBR0 is selected when the upper bits of the virtual address are all set to 0 and TTBR1 is selected when the bits are set to 1. The Translation Control Register defines the exact number of most significant bits that are used for this, although 8 bits are usually used. It is important to note that there can be even more of these registers since this architecture also allows 3 and 4 level paging.

A translation granule is the smallest block of memory that can be described, and it can be either 4KB, 16KB or 64KB (only the 64KB will be covered in this essay, but for each of these values there the design of the architecture is fixed). The exact size is specified by the Translation Granule (TG) which uses two bits from the virtual address. 00 = 4KB, 01 = 16KB and 11 = 64KB. This value is important because it determines the size of a single page table, the size of each page, the number of bits required to address a page, the number of bits that can be resolved in a single translation table lookup and the translation process used. The size of this is also important because it determines the number of translation levels. A larger translation granule reduces the number of translation table levels because each table would have more entries.

For a 64KB granule, the page size is 64KB and therefore also the size of the frame since they both need to be the same. The translation table will have 8K entries where the first level table size is 4TB, the second level table size 512MB and the third level table size is 64KB. Also, a 3-level lookup would be used where a lower-level table is accessed from an index in the higher-level table. The entry for the first table can only output the address of the second table. The third table can only output the physical address. Given a virtual address, bits 47 to 42 are used as the index for the first level table, bits 41 to 29 are used for the second level table and bits 28 to 16 are used for the third level table. The last 16 bits are used as the offset. The entry for the tables can have several attributes. Depending on the level of the table the layout is slightly different. In general, it can have an access flag, shareable attribute, access permission, security bit and the index into the Memory Attribute Indirection Register. Furthermore, parts of the tables can be cached and the MMU controls the cache policy.

To put it all together, if an instruction contains certain data, this is stored in the virtual address space and has a virtual address. During execution, this needs to be converted into its corresponding physical address using a 3-level translation process. The MMU takes the virtual address and looks at the index for the table in the first level and goes to the specified place in the table which partially resolves the address and points to the next table. The MMU also checks the table entry for validity. These steps are then repeated for the second and third tables. As a result, the physical address is output by the third table and so the offset bits are then transferred to that specified location in the physical address space. However, this is not the only way paging is implemented in ARMv8-A. Depending on certain design decisions, such as the granule size to use, the exact implementation can vary. So, there can be an implementation where 4-level paging is used or where the page size is 4KB. But regardless of this, it is mostly the same.

Part 3: Implementation

Design

The architecture has been defined using the values calculated in part 3 of this practical. Therefore, it is a 16-bit architecture, so the virtual and physical addresses are 16-bit long, where 9 bits are used for the page number/frame number and 7 bits are used for the offset. Each page and frame are 128 bytes in size. There are 512 frames, so the page table is 512 entries in size. Each page table has two additional bits along with the page and frame numbers, one which specifies whether a frame is read-only and one for whether a frame can contain data.

The two main data structures are for the page table and store. The page table is defined as an array of struct type, where each row is a struct. This struct contains all the necessary fields for each table entry. For the store, it is also an array, but of type char. It works conveniently for this implementation since the size of each char is a byte which makes it relatively simple to index into the array to store the required data. These were the two main design decisions that had to be made to set the foundation of this program.

Implementation

The implementation of this program is split into five separate methods which were provided as starter code. First off is the pt_init() method which simply initializes the page table by dynamically defining an array of structs. An important point to note is that the table is casted to the void* type and whenever the table is being used in any other methods, this is casted back to an array of structs.

The next step after this is to fill the table with entries which is precisely what mape_page_to_frame() does. This is where values are assigned to the variables in the struct type, and since it is an array, each struct is simply accessed by indexing into the array. A complementary method is also included called unmap_page(). This does the opposite by deleting an entry. This is achieved by going through the table until the specified row is found. Then from here, this row is replaced by the next row which is replaced by the row after it and vice versa. So, the deletion is done indirectly by overriding the row to be deleted where each variable in the struct is copied individually.

The virtual_to_physical() method does the conversion from page number to frame number, and by doing so it returns the physical address which will be used to store the data. This is done by taking the virtual address, which is a number stored as 16 bits since the program is emulating a

16-bit architecture. As mentioned above 9 bits are used for the page number and 7 bits for the page offset. These bits are accessed separately and stored in their respective variables. The frame number is obtained by simply bit-shifting the bits to the right by the length of the page offset, which leaves the 9 most significant bits giving the value for the page number. On the other hand, the page offset is accessed using a bitmask, where the relevant bits (which are the least significant 7 bits) go through a bitwise & operation with the binary number 0000000001111111, which is 127 in decimal. Following this, the page number is used to find the relevant row to get the frame number it is mapped to. This value and the page offset are combined to form the physical address. Bitwise concatenation is used here to achieve this, where the binary values are simply concatenated.

For storing a given data, the store_data() method is used, which has the virtual_to_physical() method to obtain the physical address. Here there are two paths which can be taken based on whether the data to be stored can fit in one frame or not. This is calculated by taking the length of the text and checking to see if it is smaller than the size of a frame minus the page offset. If it is then it can fit in one frame so the first path is taken. The frame number and page offset is then retrieved in the same way as before from the physical address. These two values are used to calculate where to index into in the store array, where the frame number is used to locate the specific frame and page offset is used to locate the specific byte within the frame. Then using a simple loop, which is repeated until the max length is reached, each char is copied using memcpy(). This is done from the buffer array provided to the store array. But if more than one frame is needed, the second path is taken where it is dealt with differently. The way I implemented this was by having multiple rows in the page table for the same page number since only one virtual address is given. But this same page number is mapped to several different frame numbers. So initially when the table is initialized and entries are added, I made sure there are enough mappings for that page number for the given virtual address. Then virtual_to_physical() is called for each of these rows and this loop runs until the last row is reached. Within each loop after a frame number is retrieved, another loop is used to copy each of the characters in the buffer to the retrieved frame number in the store array. To do this properly, some calculations are done so that the buffer is partitioned each time it is copied into the store array. An important thing to note here is that, when a virtual address is assigned to a data that is to be either written or read, the page number within the address needs to match a page number in one of the entries. Otherwise, the mapping wouldn't occur. This is the 9-bits most significant bits as mentioned above, and the remaining is used as the offset later on.

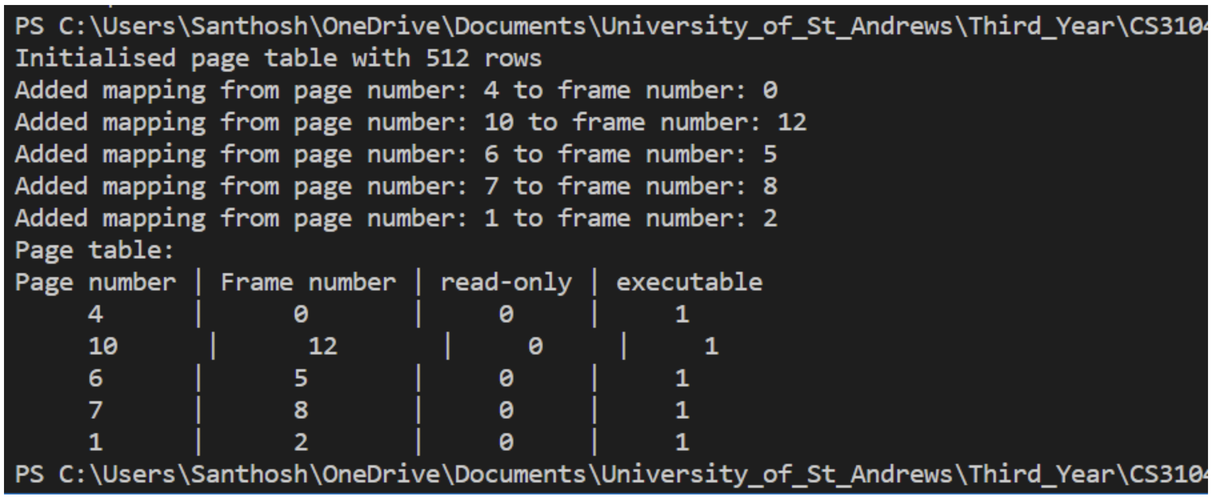
For the read_data() method, most of the code here is the same as in the previous method except for one major change, The buffer array and store array in the memcpy() method are swapped since we are reading from data which has already been stored. The final method is print_table() which is mainly just used for testing.

Testing

To ensure that my program executes properly with as little bugs as possible, I made use of the test.c file which was provided and added several tests which cover a wide range of scenarios. I added 14 tests in total where each test is a method with a description of what exact feature or functionality it intends to test.

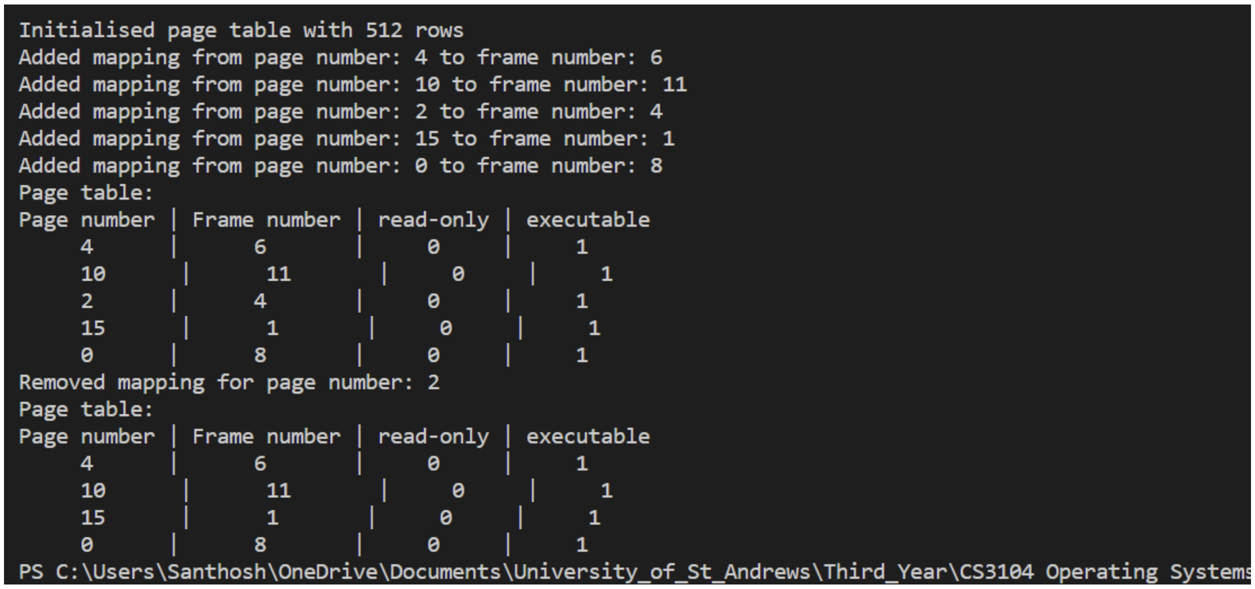
Tests:

- Whether rows can be added to the page table
    - Input: included in add_mapping_to_table().
    - Expected output: a complete table with all its rows in place.
    - Actual output:



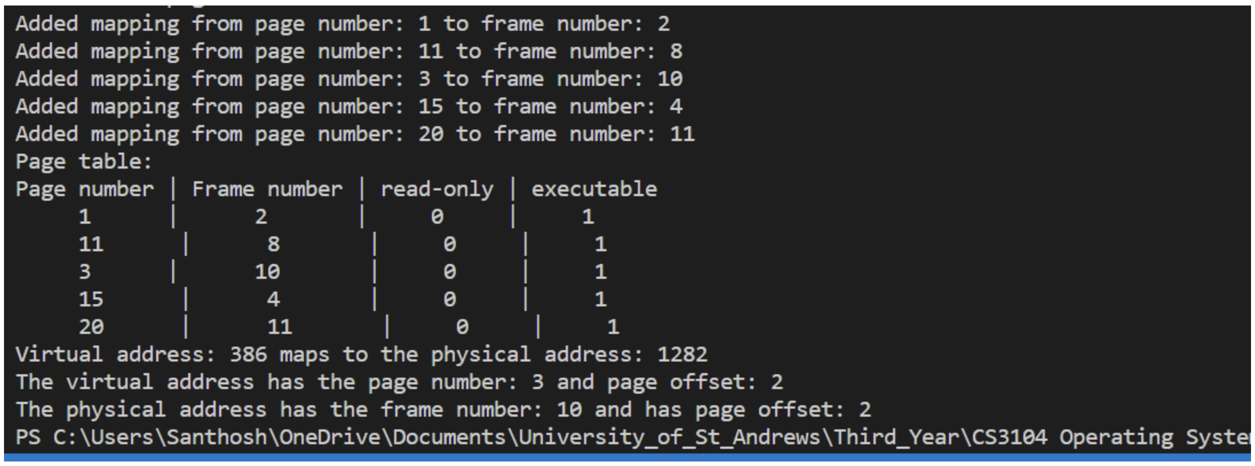
```
PS C:\Users\Santhosh\OneDrive\Documents\University_of_St_Andrews\Third_Year\CS310
Initialised page table with 512 rows
Added mapping from page number: 4 to frame number: 0
Added mapping from page number: 10 to frame number: 12
Added mapping from page number: 6 to frame number: 5
Added mapping from page number: 7 to frame number: 8
Added mapping from page number: 1 to frame number: 2
Page table:
Page number | Frame number | read-only | executable
     4      |      0       |     0     |     1
     10      |      12       |     0     |     1
     6      |      5       |     0     |     1
     7      |      8       |     0     |     1
     1      |      2       |     0     |     1
PS C:\Users\Santhosh\OneDrive\Documents\University_of_St_Andrews\Third_Year\CS310
```

- Whether rows can be deleted from the page table
    - Input: included in delete_mappings_from_table().
    - Expected output: a table with the updated rows.
    - Actual output:



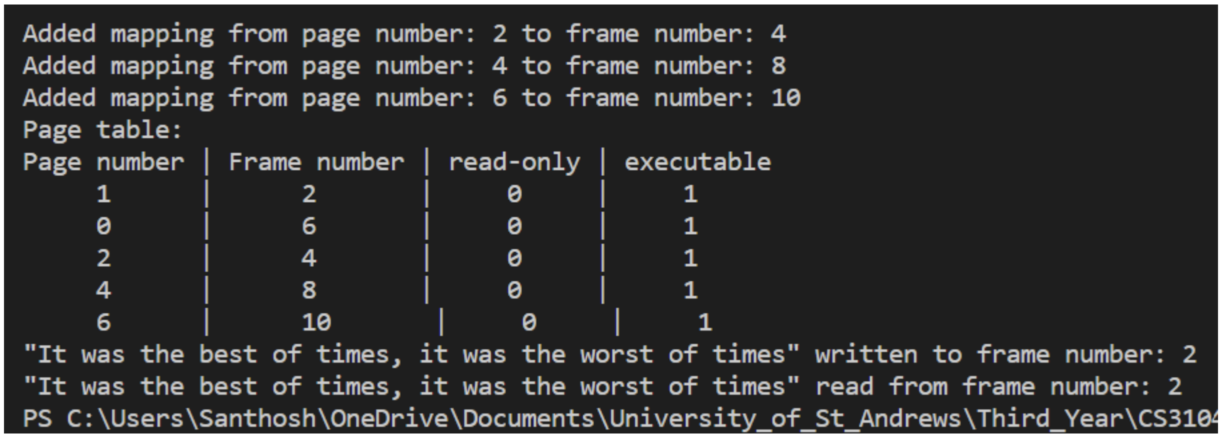
```
Initialised page table with 512 rows
Added mapping from page number: 4 to frame number: 6
Added mapping from page number: 10 to frame number: 11
Added mapping from page number: 2 to frame number: 4
Added mapping from page number: 15 to frame number: 1
Added mapping from page number: 0 to frame number: 8
Page table:
Page number | Frame number | read-only | executable
     4      |      6       |     0     |     1
     10      |      11       |     0     |     1
     2      |      4       |     0     |     1
     15      |      1       |     0     |     1
     0      |      8       |     0     |     1
Removed mapping for page number: 2
Page table:
Page number | Frame number | read-only | executable
     4      |      6       |     0     |     1
     10      |      11       |     0     |     1
     15      |      1       |     0     |     1
     0      |      8       |     0     |     1
PS C:\Users\Santhosh\OneDrive\Documents\University_of_St_Andrews\Third_Year\CS3104 Operating Systems
```

- Whether a given virtual address can be converted to a physical address
  - Input: included in map_virtual_address_to_physical_address().
  - Expected output: the correct physical address.
  - Actual output:



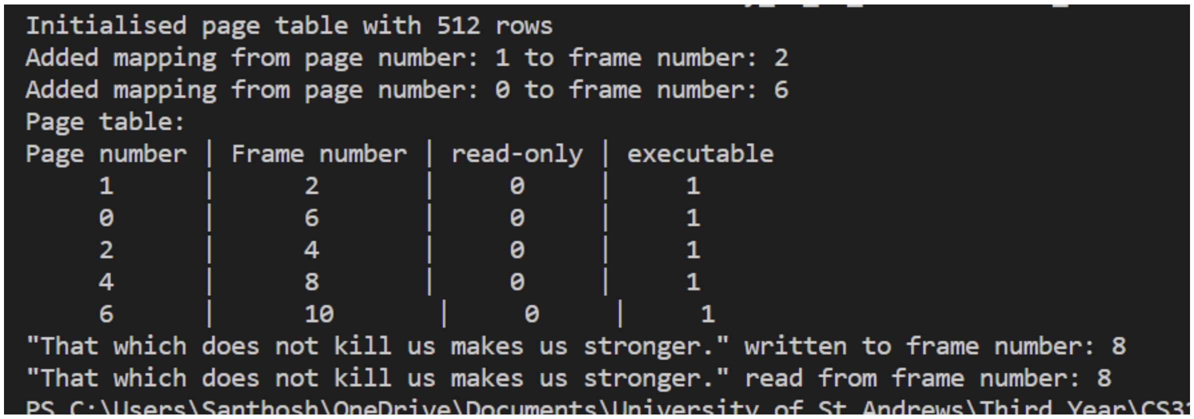
```
Added mapping from page number: 1 to frame number: 2
Added mapping from page number: 11 to frame number: 8
Added mapping from page number: 3 to frame number: 10
Added mapping from page number: 15 to frame number: 4
Added mapping from page number: 20 to frame number: 11
Page table:
Page number | Frame number | read-only | executable
     1      |      2       |     0     |     1
     11      |      8       |     0     |     1
     3      |      10       |     0     |     1
     15      |      4       |     0     |     1
     20      |      11       |     0     |     1
Virtual address: 386 maps to the physical address: 1282
The virtual address has the page number: 3 and page offset: 2
The physical address has the frame number: 10 and has page offset: 2
PS C:\Users\Santhosh\OneDrive\Documents\University_of_St_Andrews\Third_Year\CS3104 Operating Syste
```

- Whether the given text can be stored and read from a single frame (1)
  - Input: included in test1_read_write_data_in_single_frame().
  - Expected output: a message showing the text written to and read from store.
  - Actual output:



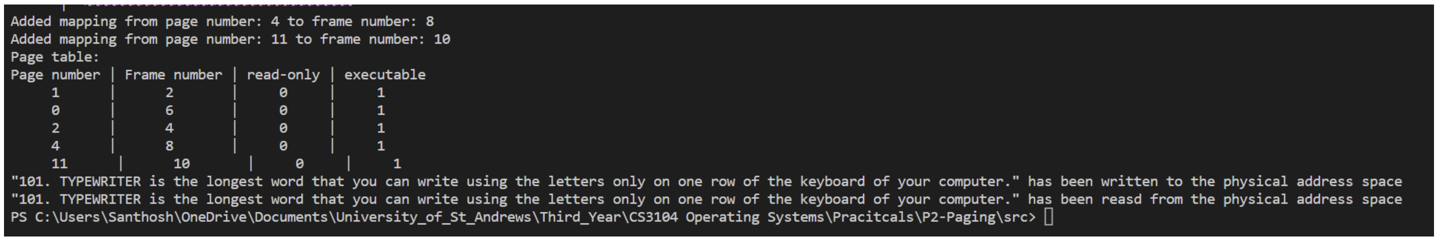
```
Added mapping from page number: 2 to frame number: 4
Added mapping from page number: 4 to frame number: 8
Added mapping from page number: 6 to frame number: 10
Page table:
Page number | Frame number | read-only | executable
     1      |      2       |     0     |     1
     0      |      6       |     0     |     1
     2      |      4       |     0     |     1
     4      |      8       |     0     |     1
     6      |      10       |     0     |     1
"It was the best of times, it was the worst of times" written to frame number: 2
"It was the best of times, it was the worst of times" read from frame number: 2
PS C:\Users\Santhosh\OneDrive\Documents\University_of_St_Andrews\Third_Year\CS310
```

- Whether the given text can be stored and read from a single frame (2)
  - Input: included in test2_read_write_data_in_single_frame ().
  - Expected output: a message showing the text written to and read from store.
  - Actual output:



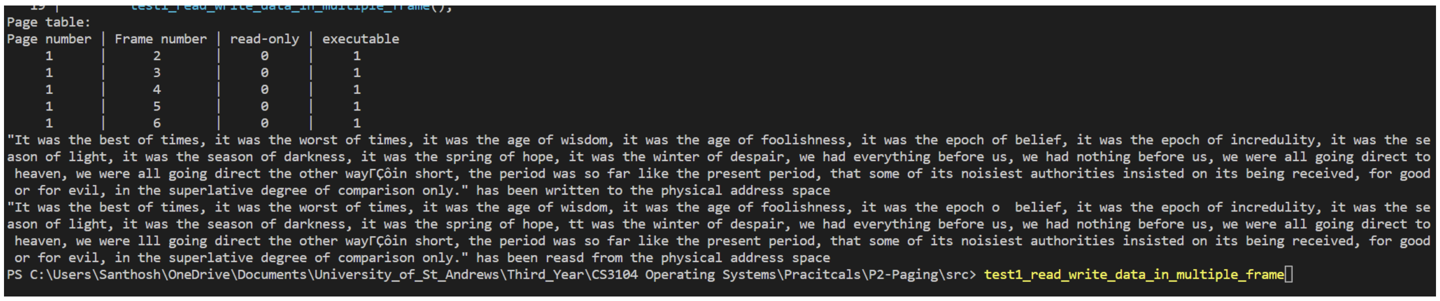
```
Initialised page table with 512 rows
Added mapping from page number: 1 to frame number: 2
Added mapping from page number: 0 to frame number: 6
Page table:
Page number | Frame number | read-only | executable
     1      |      2       |     0     |     1
     0      |      6       |     0     |     1
     2      |      4       |     0     |     1
     4      |      8       |     0     |     1
     6      |      10       |     0     |     1
"That which does not kill us makes us stronger." written to frame number: 8
"That which does not kill us makes us stronger." read from frame number: 8
PS C:\Users\Santhosh\OneDrive\Documents\University_of_St_Andrews\Third_Year\CS3
```

- Whether the given text can be stored and read from a single frame (3)
  - Input: included in test3_read_write_data_in_single_frame().
  - Expected output: a message showing the text written to and read from store.
  - Actual output:



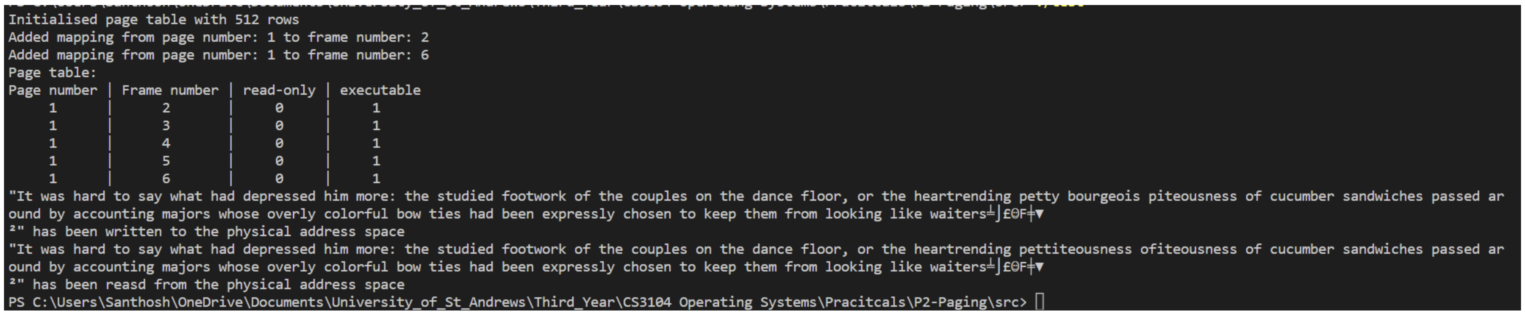
```
Added mapping from page number: 4 to frame number: 8
Added mapping from page number: 11 to frame number: 10
Page table:
Page number | Frame number | read-only | executable
     1      |      2       |     0     |     1
     0      |      6       |     0     |     1
     2      |      4       |     0     |     1
     4      |      8       |     0     |     1
     11      |      10       |     0     |     1
"101. TYPEWRITER is the longest word that you can write using the letters only on one row of the keyboard of your computer." has been written to the physical address space
"101. TYPEWRITER is the longest word that you can write using the letters only on one row of the keyboard of your computer." has been reasd from the physical address space
PS C:\Users\Santhosh\OneDrive\Documents\University_of_St_Andrews\Third_Year\CS3104 Operating Systems\Pracitcals\P2-Paging\src> 
```

- Whether the given text can be stored and read from multiple frames (1)
    - Input: included in test1_read_write_data_in_multiple_frame ().
    - Expected output: a message showing the text written to and read from store.
    - Actual output:



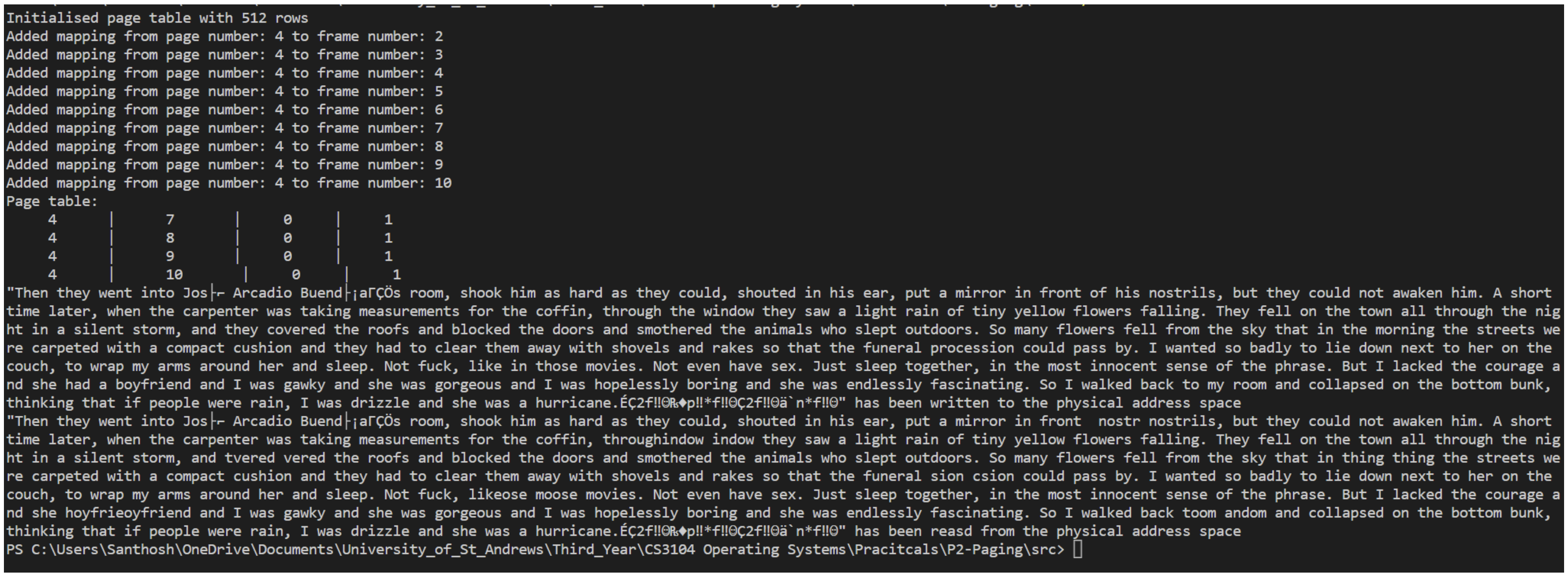
```
Page table:
Page number | Frame number | read-only | executable
     1      |      2       |     0     |     1
     1      |      3       |     0     |     1
     1      |      4       |     0     |     1
     1      |      5       |     0     |     1
     1      |      6       |     0     |     1
"It was the best of times, it was the worst of times, it was the age of wisdom, it was the age of foolishness, it was the epoch of belief, it was the epoch of incredulity, it was the se
ason of light, it was the season of darkness, it was the spring of hope, it was the winter of despair, we had everything before us, we had nothing before us, we were all going direct to
 heaven, we were all going direct the other wayΓÇôin short, the period was so far like the present period, that some of its noisiest authorities insisted on its being received, for good
 or for evil, in the superlative degree of comparison only." has been written to the physical address space
"It was the best of times, it was the worst of times, it was the age of wisdom, it was the age of foolishness, it was the epoch o  belief, it was the epoch of incredulity, it was the se
ason of light, it was the season of darkness, it was the spring of hope, tt was the winter of despair, we had everything before us, we had nothing before us, we were all going direct to
 heaven, we were lll going direct the other wayΓÇôin short, the period was so far like the present period, that some of its noisiest authorities insisted on its being received, for good
 or for evil, in the superlative degree of comparison only." has been reasd from the physical address space
PS C:\Users\Santhosh\OneDrive\Documents\University_of_St_Andrews\Third_Year\CS3104 Operating Systems\Pracitcals\P2-Paging\src> test1_read_write_data_in_multiple_frame
```

- Whether the given text can be stored and read from multiple frames (2)
    - Input: included in test2_read_write_data_in_multiple_frame ().
    - Expected output: a message showing the text written to and read from store
    - Actual output:



```
Initialised page table with 512 rows
Added mapping from page number: 1 to frame number: 2
Added mapping from page number: 1 to frame number: 6
Page table:
Page number | Frame number | read-only | executable
     1      |      2       |     0     |     1
     1      |      3       |     0     |     1
     1      |      4       |     0     |     1
     1      |      5       |     0     |     1
     1      |      6       |     0     |     1
"It was hard to say what had depressed him more: the studied footwork of the couples on the dance floor, or the heartrending petty bourgeois piteousness of cucumber sandwiches passed ar
ound by accounting majors whose overly colorful bow ties had been expressly chosen to keep them from looking like waiters╧⌡£ΘF╪▼
²" has been written to the physical address space
"It was hard to say what had depressed him more: the studied footwork of the couples on the dance floor, or the heartrending pettiteousness ofiteousness of cucumber sandwiches passed ar
ound by accounting majors whose overly colorful bow ties had been expressly chosen to keep them from looking like waiters╧⌡£ΘF╪▼
²" has been reasd from the physical address space
PS C:\Users\Santhosh\OneDrive\Documents\University_of_St_Andrews\Third_Year\CS3104 Operating Systems\Pracitcals\P2-Paging\src> 
```

- Whether the given text can be stored and read from multiple frames (3)
    - Input: included test3_read_write_data_in_multiple_frame ().
    - Expected output: a message showing the text written to and read from store
    - Actual output:



```
Initialised page table with 512 rows
Added mapping from page number: 4 to frame number: 2
Added mapping from page number: 4 to frame number: 3
Added mapping from page number: 4 to frame number: 4
Added mapping from page number: 4 to frame number: 5
Added mapping from page number: 4 to frame number: 6
Added mapping from page number: 4 to frame number: 7
Added mapping from page number: 4 to frame number: 8
Added mapping from page number: 4 to frame number: 9
Added mapping from page number: 4 to frame number: 10
Page table:
     4     |     7     |     0     |     1
     4     |     8     |     0     |     1
     4     |     9     |     0     |     1
     4     |     10     |     0     |     1
"Then they went into Jos├─ Arcadio Buend├¡aΓÇÖs room, shook him as hard as they could, shouted in his ear, put a mirror in front of his nostrils, but they could not awaken him. A short
time later, when the carpenter was taking measurements for the coffin, through the window they saw a light rain of tiny yellow flowers falling. They fell on the town all through the nig
ht in a silent storm, and they covered the roofs and blocked the doors and smothered the animals who slept outdoors. So many flowers fell from the sky that in the morning the streets we
re carpeted with a compact cushion and they had to clear them away with shovels and rakes so that the funeral procession could pass by. I wanted so badly to lie down next to her on the
couch, to wrap my arms around her and sleep. Not fuck, like in those movies. Not even have sex. Just sleep together, in the most innocent sense of the phrase. But I lacked the courage a
nd she had a boyfriend and I was gawky and she was gorgeous and I was hopelessly boring and she was endlessly fascinating. So I walked back to my room and collapsed on the bottom bunk,
thinking that if people were rain, I was drizzle and she was a hurricane.ÉÇ2f‼ΘRs♦p‼*f‼ΘÇ2f‼Θä`n*f‼Θ" has been written to the physical address space
"Then they went into Jos├─ Arcadio Buend├¡aΓÇÖs room, shook him as hard as they could, shouted in his ear, put a mirror in front  nostr nostrils, but they could not awaken him. A short
time later, when the carpenter was taking measurements for the coffin, throughindow indow they saw a light rain of tiny yellow flowers falling. They fell on the town all through the nig
ht in a silent storm, and tvered vered the roofs and blocked the doors and smothered the animals who slept outdoors. So many flowers fell from the sky that in thing thing the streets we
re carpeted with a compact cushion and they had to clear them away with shovels and rakes so that the funeral sion csion could pass by. I wanted so badly to lie down next to her on the
couch, to wrap my arms around her and sleep. Not fuck, likeose moose movies. Not even have sex. Just sleep together, in the most innocent sense of the phrase. But I lacked the courage a
nd she hoyfrieoyfriend and I was gawky and she was gorgeous and I was hopelessly boring and she was endlessly fascinating. So I walked back toom andom and collapsed on the bottom bunk,
thinking that if people were rain, I was drizzle and she was a hurricane.ÉÇ2f‼ΘRs♦p‼*f‼ΘÇ2f‼Θä`n*f‼Θ" has been reasd from the physical address space
PS C:\Users\Santhosh\OneDrive\Documents\University_of_St_Andrews\Third_Year\CS3104 Operating Systems\Pracitcals\P2-Paging\src> 
```

An important thing to mention here is that I have also written tests for the read-only and executable bits. However, I ran into some issues with them at the last minute and couldn't fix them due to time constraints. This is mainly the case with the executable bit but this will not be elaborated any further due to a lack of knowledge.

Conclusion

In conclusion, through this practical I have developed a deeper understanding of paging, especially in terms of how it's applied in practice. Not only by reading about certain existing architectures, but also by defining and implementing my own version. I was able to perform certain calculations to determine certain properties of a given architecture. And to use this information to come up with a working page mechanism. Although I did find it a bit challenging to get going initially since I found the practical specification to be a bit too vague for part 3. This resulted in code that isn't very neat and elegant because I ended up making last minute changes, which I would have to take the responsibility fr

Bibliography

[1] ARM architecture family, Wikipedia, 22 November 2022, https://en.wikipedia.org/wiki/ARM_architecture_family.

[2] Memory translation granule size, 22 November 2022, https://armv8-ref.codingbelief.com/en/chapter_d4/d42_3_memory_translation_granule_size.html.

[3] https://developer.arm.com/documentation/101811/0102/Translation-granule.

[4] *Armv8-A Address Translation*, 22 November 2022, armv8_a_address_translation.pdf.

[5]*Arm® Architecture Reference Manual*, 22 November 2022, DDI0487_I_a_a-profile_architecture_reference_manual.pdf.

[6] *ARMv8-A Architecture Overview*, 22 November 2022, ARMv8_Overview.pdf.