CS3104 Practical 2

An Analysis of Paging

200007413

1 Page Table

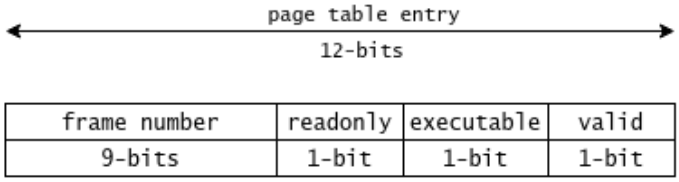

Figure 1: Page Table Entry

Given the requirement of 128-byte pages/frames, the number of bits required to reference each byte of each frame is $\log_2(128) = 7$. Therefore, as the architecture is using 16-bit addresses, 7 out of the 16-bits in each address are used as the page/frame offset. This leaves $16 - 7 = 9$ bits to reference each page and frame in the page table and physical memory respectively.

This gives us the maximum number of pages and frames which can be referenced. With 9 bits, $2^9 = 512$ pages/frames can be referenced, and so 512 rows are needed in our page table.

Given that the page table is used to get the frame number correspoding to a given page number, 9-bits are required per page table row to store the frame number. Additionally, the specification requires two protection bits for readonly, and executable flags. Finally, I decided to include one additional valid bit to help keep track of what pages are actually stored in memory. This gives a total of $9 + 2 + 1 = 12$ bits required to store each row.

The total size required to store our page table is therefore $512 * 12 = 6144$-bits (1536 bytes).

The total size of the virtual memory and physical memory are the same because our architecture only supports single-level paging. As we can reference a maximum of 512 pages/frames with a size of 128-bytes, the maximum virtual and physical memory is $512*128 = 65536$-bytes (64 KiB) of byte-addressable memory.

To convert a virtual memory address to a physical memory address, we first need to convert the value into binary (base-2). This isn't really necessary for machines, as they store all data

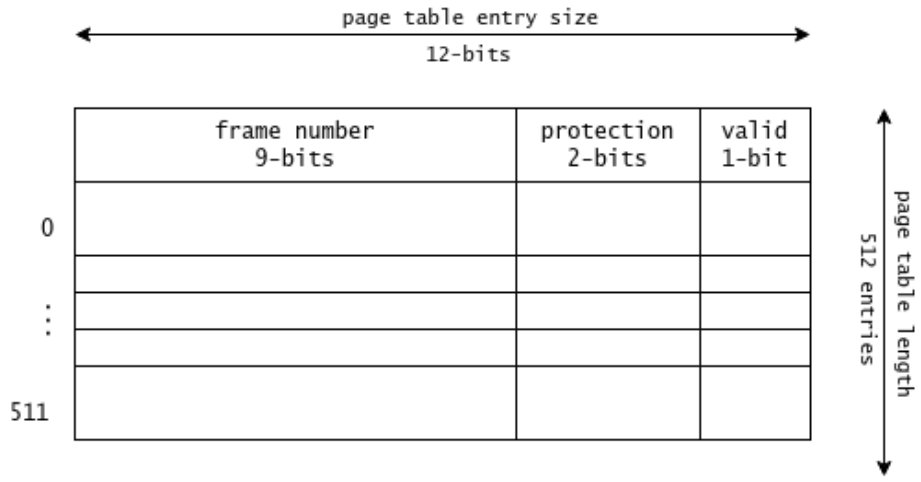

Figure 2: Page Table

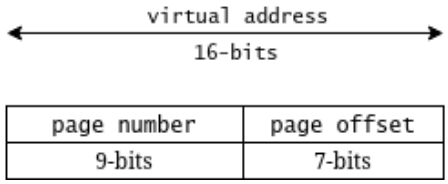

Figure 3: Virtual Memory Address

as binary, but if it were necessary would be done by the kernel and not the hardware.

- 0xbeef (base 16) = 1011111011101111 (base 2)
- 0xc001 (base 16) = 1100000000000001 (base 2)
- 07413 (base 10) = 0001110011110101 (base 2)

From here, we need to do a bitwise shift right by the number of bits storing the offset (in this case 7-bits) to get the most significant 9-bits representing the page number. This would be done by the kernel.

- $1011111011101111 >> 7 = 101111101$
- $1100000000000001 >> 7 = 110000000$
- $0001110011110101 >> 7 = 000111001$

This gives us the page number, which use to index the page table. The returning value is a 12-bit row. The first 9-bits are the frame number, the 10th bit is the readonly bit, the 11th bit is the executable bit, and the 12th bit is the valid bit. This page table lookup would be done by the hardware.

- table[101111101] = $f_1 r_1 e_1 v_1$ (binary)
- table[110000000] = $f_2 r_2 e_2 v_2$ (binary)
- table[000111001] = $f_3 r_3 e_3 v_3$ (binary)

To get the frame number from this row, we can do a bitwise shift right to remove the valid and protection bits (in our case 3-bits). This would be done by the hardware, but the kernel would be responsible for checking the protection and valid bits.

- table[101111101] $>> 3 = f_1$
- table[110000000] $>> 3 = f_2$
- table[000111001] $>> 3 = f_3$

As each frame in physical memory is 128-bytes, to index the frame and offset, we can multiply the frame number by 128 (as physical memory is byte-addressable), and add the offset. This is the same as doing a bitwise shift on the frame number by the offset size (7), and then adding the offset.

The offset is found by taking the last 7 bits of the virtual address, which is the same as taking the address bitwise and $2^7 - 1 = 127 = 0000000001111111$ (binary). This would be done by the hardware.

- $(f_1 << 7) + 1101111 = f_a 1101111$ (binary)
- $(f_2 << 7) + 0000001 = f_b 0000001$ (binary)
- $(f_3 << 7) + 1110101 = f_c 1110101$ (binary)

This gives us our final values for the conversion of the three given examples.

- 0xbeef $\rightarrow f_a 1101111$ (binary)
- 0xc001 $\rightarrow f_b 0000001$ (binary)
- 07413 $\rightarrow f_c 1110101$ (binary)

2 Research into ARM Architecture

This essay will attempt to given an overview of paging within the ARMv8-A architecture. It is relevant to note that the official ARMv8-A documentation refers to paging as translation and, as such, page tables as translation tables.

ARMv8-A supports two *execution states*: AArch64 and AArch32, allowing the processors to run 64-bit and 32-bit applications respectively. AArch32 mostly implements the previous 32-bit ARMv7-A design[1] and so, for this essay, I will be focusing on the ARMv8-A specific AArch64 execution state.

Given the 64-bit architecture, each virtual address is 64-bits wide. However, only 48-bits of each address are used in indexing the translation tables. This is referred to as the VA input rangeThe reason for this is because the virtual address space is split into two sections labelled TTBR0, and TTBR1. TTBR0 is allocated to applications, and spans the least significant 48-bit address range of virtual memory. TTBR1 is allocated to the OS and peripherals, and spans the most significant 48-bits of virtual address space. This allows applications and the OS/peripherals to be addressed using their own translation tables with different privilege levels, meaning user applications must ask the kernel for permission to access data stored in TTBR1[2].

This separation also reduces the cost of context switches. Each application is allocated it's own 48-bit virtual address space and transition tables. The virtual address space gets swapped in and out of the TTBR0 region of virtual memory by the kernel. This allows all applications to use the same virtual addresses, meaning they can run independently of knowledge of the memory used by other programs. The data stored by OS/peripherals is more persistently relevant for system operation, and is often used by different applications. This region does not get swapped in and out of virtual memory, and so can have it's own persistent translation table[3].

In using the most and least significant 48-bit address space means that the first 16-bits of mapped addresses are always either all 0s are 1s. The region between these two sections is unaccessible, as the addresses are unmapped, thus a fault is caused when addresses without all 0 or 1 starting bits are used[4].

Therefore, the number of physical bytes that can be referenced from the 64-bit addresses are $2 * 2^{48} = 2^{49}$ bytes of byte-addressable memory which is roughly 512TiB.

The translation tables themselves for ARMv8-A are hierarchical. There are a maximum of 4 levels of translation tables (0 to 3), but the number used depends on the page/frame

[1] Programmers Guide for ARMv8-A, p.1-1
[2] Programmers Guide for ARMv8-A, p.12-7
[3] Programmers Guide for ARMv8-A, p.12-7
[4] Programmers Guide for ARMv8-A, pp.12-7:12-8

size, as well as the physical memory available [5]. Page/frame size is referred to in official documentation as *granule size*. The kernel can switch between three sizes: 4KB, 16KB, or 64KB[6]. Each level is indexed by a portion of the virtual address, and gives a pointer to translation table for the next level, or a block of memory. Level 0 always references the level 1 table, and level 3 always references a block of memory. The translation is complete when a block is reached.

When the 4KB granule size is used, all 4 levels can be used. The least significant 12 bits represent the offset. The remaining 36-bits are split into 8/8/8/8 bit page numbers, where the level 0 table is selected by which region TTBR0 or TTBR1 the address is in. The frame number will be 36-bits wide[7].

When the 16KB granule size is used, all 4 levels can be used in the same way as with the 4KB granule size, except 14 bits represent the offset, and the other 34 bits are split into 1/11/11/11 bit page numbers. The frame number will be 34-bits wide [8].

When the 64KB granule size is used, the level 0 table is skipped, so only 3 levels can be used. The least significant 16-bits represent the offset, and the other 32 bits are split into 6/13/13 bit page numbers. The frame number will be 32-bits wide[9].

The format for the AArch64 specific translation tables is called the *Long Descriptor format*. This specifies that each entry in the translation table is 64-bits [10]. Therefore, the amount of memory required to store one set of translation tables with a 4KB granule size is $64 * 2^8 * 4 = 65536$-bits (64KiB). For a 16KB granule size, the memory required is $64 * (2^1 + (2^1 1 * 3)) = 393344$-bits ($\approx$ 48KiB). For a granule size of 64KB, the memory required is $64 * (2^6 + 2^1 3 + 2^1 3) = 1052672$-bits (128.5KiB).

If physical memory is small enough to allow the virtual address input range to be restricted to 42-bits, then the first level of the translation tables for each granule size can be skipped as every address will have the same index for this level[11].

Along with the address of the next page table level or frame number, each entry in the translation tables also includes several protection bits. The first is a Access Flag, which indictates whether a page has been used. This helps spot when a page has not been mapped[12]. Next, there are two access permission bits. The effect of their value depends on what privilege level the access has[13]:

[5]Programmers Guide for ARMv8-A, pp.12-9:12-11
[6]Programmers Guide for ARMv8-A, pp.12-15:12-16
[7]Programmers Guide for ARMv8-A, p.12-15
[8]Programmers Guide for ARMv8-A, pp.12-15:12-16
[9]Programmers Guide for ARMv8-A, p.12-16
[10]Programmers Guide for ARMv8-A, p.12-14
[11]Programmers Guide for ARMv8-A, p.12-15
[12]Programmers Guide for ARMv8-A, p.12-25
[13]Programmers Guide for ARMv8-A, p.12-23

bits	unprivileged	privileged
00	no-access	read-write
01	read-write	read-write
10	no-access	read-only
11	read-only	read-only

Another 2-bits are used for cacheability settings, determining whether not a page can be cached, and in what way[14]:

bits	caching policy
00	not cacheable
01	write-back and write-allocate cacheable
10	write-through cacheable
11	write-back cacheable

There are 2 bits reserved for shareability settings which is relevant to translation table walks[15]:

bits	shareability
00	non-shareable
01	unpredictable
10	outer shareable
11	inner shareable

Along with the hierarchical translation tables, ARMv8-A also implements a Translation Lookaside Buffer (TLB) which is a cached table of recently accessed translations. This can only store a limited number of entries. It acts somewhat as an inverse translation table, as the Address Space ID, and Virtual Machine ID are stored which store the application to which has the entry in it's translation tables. This can be matched with the particular application Address Space ID and virtual machine ID if necesssary[16]. The TLB, can be used more effectively when caching contigous areas of physical memory. When frames are contigous, only one cache is required which leads to a TLB hit whenever the contigous frames are searched for[17]. If the page does not occur in the TLB, then a TLB miss occurs, and a standard lookup is used.

[14]Programmers Guide for ARMv8-A, p.12-18
[15]Programmers Guide for ARMv8-A, p.12-18
[16]Programmers Guide for ARMv8-A, p.12-4
[17]Programmers Guide for ARMv8-A, p.12-6

2.1 Bibliography

- ARM Cortex A Series v1.0, "Programmers Guide for ARMv8-A" (ARM, 2015)

  ARM DEN0024A (ID050815)

3 Implementation

3.1 Design

The first major design decision for my implementation was how to implement the page table entries. As previously described in section 1, each entry is 12-bits. To help with this, I created a stucture called PageEntry, which used bitfields to create a 9-bit unsigned integer to store the frame number, and booleans for the three protection bits.

This meant that the size of a PageEntry would always be 12-bits wide, and allowed me to index the page tabled, which was created by creating a byte-addressable section of memory.

This simplified getting entries from the table, as I could cast the page table from a void to a PageEntry. This treats the table as an array of PageEntries, and so indexing returns 12-bit values with the PageEntry type which could then easily return the relevant attributes, thus allowing me to index directly just using a given page number.

For storing and reading data with a given byte size, I decided to loop over the buffers, storing and reading in byte chunks, because physical addresses address store data in bytes. I cast the store from a void to a char, because chars have a size of 1 byte. This let me use the byte length directly as I could loop over the length, and increment the physical address which has the effect of incrementing the offset, which indexes to the next byte in the frame.

I added in checks when storing and reading data to make sure that data cannot be read or stored outsize of the frame specified if the length of the data is too large given the frame size and the offset specified. This is because reading or storing beyond the given frame would pose a security risk as programs could read or overwrite data stored by other programs if mapped next to one another in physical memory.

I decided to use the errno library for my implementation. This is because for the future extensions, when faults occur, using errno allows me to return the correct frame number when converting a virtual address to a physical address, but also let the calling application know that a fault has occured, which may mean that the page is not loaded into phyical memory, but is on disk.

I decided not to add any checks into my implementation to make sure that the store was large enough when storing and reading data. This is because the simulation code is replicating just the hardware. Instead, I decided to trust that the store given would have enough space, and it was the calling codes fault if they map and address a part of physical memory out side of it's own range.

3.2 Testing

I implemented a unit testing design, which enabled me to first implement initializing the page table, and then converting virtual addresses to physical addresses, and then storing and reading from a store.

For the use cases tests, I decided to add unit tests for checking that small sized data can be stored, as well as large size data. I also addded in a test that data of different types could be stored adjacently, and then retrieved successfully.