32bit Protected Mode to Long Mode

GDT Creation

To handle memory on the system, the processor makes use of paging and segmentation as section 3.1 of the 3rd Volume of the Intel Documentation shows.

Under IA-32 and IA-32e in compatibility mode, segmentation allows the CPU to separate code, data and stack modules into memory segments (which are protected in Protected Mode).
In this case, the base addresses of the segments of a program may be given to the Segment Selectors of the OS while adding them to the offest in a Global Descriptor Table.

If paging is enabled (necessary to enable 64bit mode) and the CPU runs in 64bit mode, most Segment Selectors are ignore and CS is then used to switch the privileged modes between Ring 0 (Kernel Mode) and Ring 3 (User Mode).
We won’t make use of Ring 1 and Ring 2 for this project.
Section 3.2.4 and 3.5.2 of the 3rd Volume of the Intel Documentation provides more insight into how it functions.

The Global Descriptor table defines the properties of the segments used by the OS.
By default, we must define the kernel code segment that will be loaded in the Code Segment (CS) descriptor as well as a data segment for the data segment registers.
In 64bit mode, the data segments are ignored except for FS and GS and most parameters which must be defined in 32bit mode are ignored because address space is considered infinite.
The first entry of this table is always NULL.

Section 6.2.1 of the 3rd Volume of the Intel Documentation defines the following flags used by the Code Segment of a 64bit system.
Since we still haven't implemented the code running in the User Priviledged ring, we couldn't test the segments we have defined for the **code_user_segment**.

GDT Macros:

```
ACCESSED        equ 1 << 40 ; The CPU switches this bit to 1 when the specified segment is accessed. If the segment is stored in Read Only Data and the segment is accessed, it leads to failure.
RW              equ 1 << 41 ; Sets the Read Write bit. In code segments, it enables read. In data segments, it enables write.
EXEC            equ 1 << 43 ; Executable Bit. It defines the segment as executable.
DESCTYPE        equ 1 << 44 ; Descriptor type bit. When set to 0, it is a system segment. When set to 1, it is a data or code segment.
DPLLOW          equ 1 << 45 ; Defines the Ring of the segment. Ring 0 is kernel, Ring 3 is userland.
DPLHIGH         equ 1 << 46 ; Ring 1 and 2 are never used by a modern OS.
PRESENT         equ 1 << 47 ; Sets if a segment is valid or not. Must be switched on for all segments.
LONGBIT         equ 1 << 53 ; Enables long mode for a segment.
```

Kernel Page Table Definition

Under the x86_64 architecture, a 4-level paging (5-level paging on newer processors) is supported.
As it is our first experience with direct communication with the memory architecture of Intel CPUs, we decided to provide support for 4-level paging first.
Therefore, page tables will be defined as a tree of entries.

Paging transforms a physical address in RAM into a virtual address. This address is automatically decoded by the CPU thanks to its Memory Management Unit (MMU).
Control of the paging mechanisms is handled by Kernel Code and not the MMU itself as its only purpose is to decode addresses read by the CPU.

Section 5.5 of the 3rd Volume of the Intel Documentation discusses the address translation process.

In 64bit mode, only the first 48bits of the Virtual Address are used and a design decision made by manufacturers makes it so that the first 16bits must all be the same as the 47th bit.
We won't be using the Big Pages feature which allows 1GB sized pages as the leaves of the tree.
As such, all the page frames used by the system will be 4kb in size and all levels of the tree will have 512 entries each.
In order:

- The Page Map Level 4 (PML4) entry is defined by the bits 47 through 39.
- The Directory Pointer is defined by the bits 38 through 30.
- The Directory entry is defined by the bits 29 through 21.
- The Page Table entry is defined by the bits 20 through 12.
- The offset in the page is defined by the bits 11 through 0.

The page table defined in the .data section of the main.asm file can be explained like so.

After paging is enabled, the CPU will start reading all addresses as virtual addresses.
Therefore, we will identity map the first 64MB of physical RAM.
The process of identity mapping means that a Physical Address is equal to its Virtual Address.
For example, Physical Address 0x0 being encoded as Virtual Address 0x0.

The label named **page_table_level_4** will represent a pointer after compilation. As it is an array of 512 entries of 64bit values, we map entry 0 to the pointer of the Directory Pointer named **page_table_level_3**.
We then do the same with the Directory Pointer entry and the Directory entry.

Finally, we map all 512 entries of the Page Table with base addresses 0 through 511. Testing to assign more memory to the OS for the Memory Manager shows that we only need to use indexes instead of an address to index 4kb blocs.
As such, only using 0 through 511 will automatically be detected as base addresses of 4kb blocs.
These values are moved 12 bits to the right because the format loads the base address starting from bit 12.

All of the entries are loaded with 0b11 because Bit 0 and 1 are parameters of the page needed for proper kernel operations.
Bit 0 is the Present bit which allow mapping of a 4kb bloc and Bit 1 enables Read/Write permissions to the page.

NB: Read section Higher Half Setup to understand why the final entry of the Kernel Page Table has been duplicated.

More information may be found on this blog.

Multiboot Header Setup

As GRUB is a bootloader, it will run known procedures in order to initialise the system.
It will then exit 16bit Real Mode and return control to a **start** label as a 32bit process in Protected Mode.

Upon return, GRUB will store a pointer to a structure in the ebx register under a format defined in the GNU documentation.

There exist 2 formats, namely Multiboot 1 and Multiboot 2.
As we were unable to parse the Multiboot 2 structure, we temporarily decided to use Multiboot 1.
The header may be found under the path **./src/boot/header.asm** in the *COS Github Repository*.

In accordance with the GNU Documenation, the Multiboot 1 Header contains 3 fields, including:

- The magic number for a Multiboot 1 Structure.
- The flags of the structure represented as a longword.
- A checksum which must be equal to the 32bit additive inverse of the sum of the two other fields.

Upon return to **start** label in the file **./src/boot/main.asm**, we check whether or not the header was found by GRUB with the **check_for_multiboot_header** function.
If it was, the value *0x2BADB002* is stored in the eax register.

Long Mode Support Checking

As this project aims at running a 64bit compatible Operating System, we must first enable Long Mode (called IA-32e by Intel) support as Section 11.8.5 of the 3rd Volume of the Intel Documentation shows.

This section is WIP as there not only one way to check for Long Mode support. The method we chose for this POC was to test for CPUID support.

Chapter 2.3 of Volume 3A of the Intel Documentation shows that EFLAGS registers include bits to document CPU features.
If a processor supports the CPUID instruction, it is possible to leverage it to know if a processor supports long mode.
However, reports show that this wasn't standard on older processors. As such, they may support Long Mode but have no support for the CPUID instruction.

check_for_cpuid_support function

```
pushfd              ; We start by storing the Eflags in sthe stack.
pop eax             ; The first value is poped from the stack into eax, since the ID bit is the last of the eflags register, it means its the value at the top of the stack.
mov ecx, eax        ; The value of eax is temporarily backed up into ecx in order to check whether or not turning on the cpuid works.
or eax, 1 << 21     ; To enable support for CPUID, the ID bit of the eflags registers must be switched to 1.
push eax            ; Restores the stack by pushing the missing value of the eflags registers.
popfd               ; This instruction will try to switch the ID bit on by restoring the values from the stack.
pushfd
pop eax
push ecx            ; Pushes the backed up value to the stack to leave this function with the cpu state unchanged.
popfd
cmp eax, ecx        ; If eax and ecx are equal, it means that switching the ID bit doesn't work and most likely means the CPU doesn't have the CPUID instruction.
```

check_for_long_mode function

The CPUID instruction can return information about the processor depending on the value passed to it via the eax register.
The following code will check the result returned in the 29th bit as it documents long mode support.

```
mov eax, 0x80000000 ; When value 0x8000000 is passed as argument, CPUID will return in eax the maximum value that CPUID can handle on the current processor.
cpuid
cmp eax, 0x80000001 ; If eax is inferior to 0x80000001, it means the CPU most likely cannot enter long mode.
jb error

mov eax, 0x80000001 ; This value will return in ecx and edx information about the CPU.
cpuid
test edx, 1 << 29   ; Bit 29 in edx is flipped to 1 if 64bit support exists in the CPU.
```

Enabling paging

When long mode support is confirmed, one may then turn on the paging feature of the processor via the Control Registers and MSR 0xC0000080.

These registers are documented under Section 2.5 of 3rd Volume on System Programming of the Intel Documentation as:

```
Control registers (CR0, CR1, CR2, CR3, and CR4) determine operating mode of the processor and the characteristics of the currently executing task.
```

Control Register 4’s 5th bit switches on the Physical Address Extension.
This allows 32bit processes to be located above the 4GB RAM limit imposed by the 32bit architecture.
However, their virtual address space remains bound to a 4GB RAM limit.
As it is required to switch to long mode, it is set to 1.

Bit 31 and 1 of Control Register 0 controls the paging and protected mode features respectively.
While it would be unnecessary to enable Bit 1 as GRUB loaded the kernel in 32bit protected mode, we decided to do it as would like to provide support for a 32bit mode within the COS Kernel later down the line.

Section 11.4 of 3rd Volume of the Intel Documentation states that most IA-32 CPUs include Model Specific-Registers (MSRs).
These are used to provide control to hardware and software features

and MSR 0xC0000080 controls long mode features.
MSRs require usage of the RDMSR and WRMSR to ReaD and WRite values to them.

Finally, we must include a pointer to a paging structure in Control Register 3 to complete Paging initialisation.

enable_paging function

```
; Control Register 4's 5th bit switching.
mov eax, cr4
or eax, 1 << 5
mov cr4, eax

; The pointer to the level 4 page passed to Control Register 3.
mov eax, page_table_level_4 - VIRT_ADDR
mov cr3, eax

; Bit 8 enables or disables long mode.
; It is flipped to 1 to enable it.
; RDMSR and WRMSR take their arguments in the ecx and eax
registers respectively.
mov ecx, 0xC0000080
rdmsr
or eax, 1 << 8
wrmsr

; Bit 31 and 1 of control register 0 switching.
mov eax, cr0
or eax, 0x80000001
mov cr0, eax
```

Switch to 64bit mode

Once all prior routines are run, the GDT can be loaded into the GDT register with the LGDT instruction.

In Protected Mode, the Code Segment is loaded with the code to be executed. Even though the base address is always 0 for all Segment Selector once the CPU was switched into long mode, the Code Selector must be loaded with the proper value to finalise the switch to 64bit mode.

Section 6.4 of the 3rd Volume of the Intel Documentation states that Type Checking will be performed whenever a far call is made, a far jump is made or an interruption occurs.

At the beginning of the initialisation routine, we executed the CLI instruction which deactivated hardware interrupts.
It was needed because the Code Selector could have changed before the initialisation was finished.

A far jump may now be performed to switch to 64bit mode with the following instruction.

```
jmp gdt64.code_segment:long_mode_start - VIRT_ADDR
```

NB: Read the Higher Half Setup section to understand why VIRT_ADDR is added to this instruction.