UiT

THE ARCTIC UNIVERSITY OF NORWAY

Protected Mode

A brief introduction

Erlend Graff

INF-2201 Operating System Fundamentals

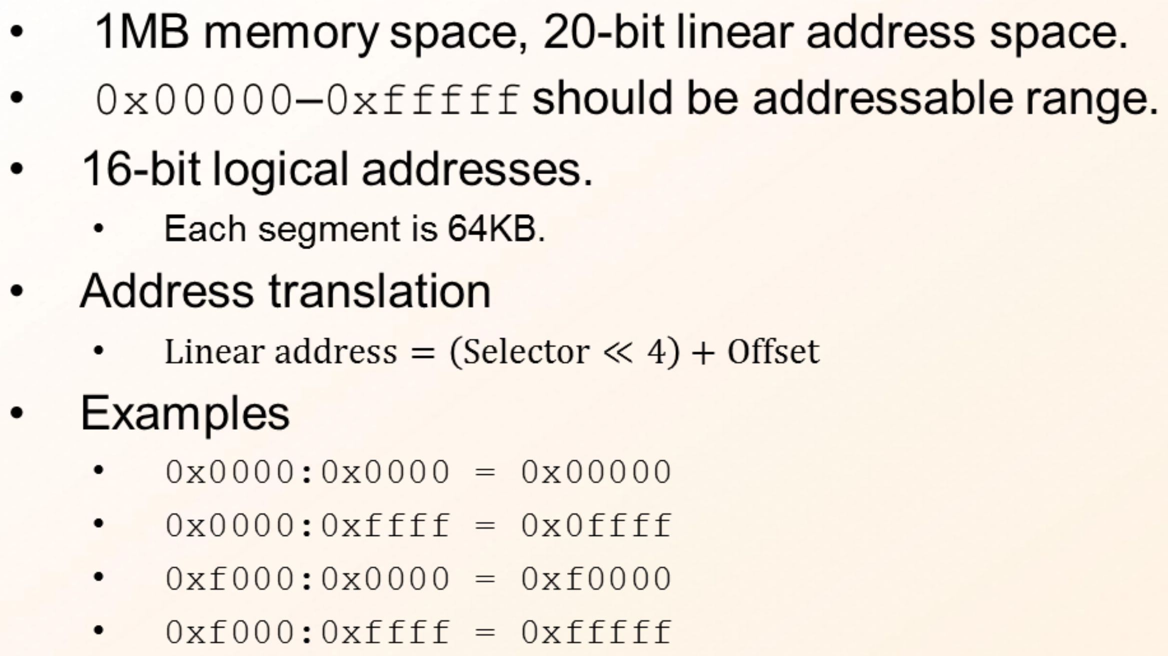
Department of Computer Science
University of Tromsø

UNIVERSITY OF TROMSØ

Real Mode – recap

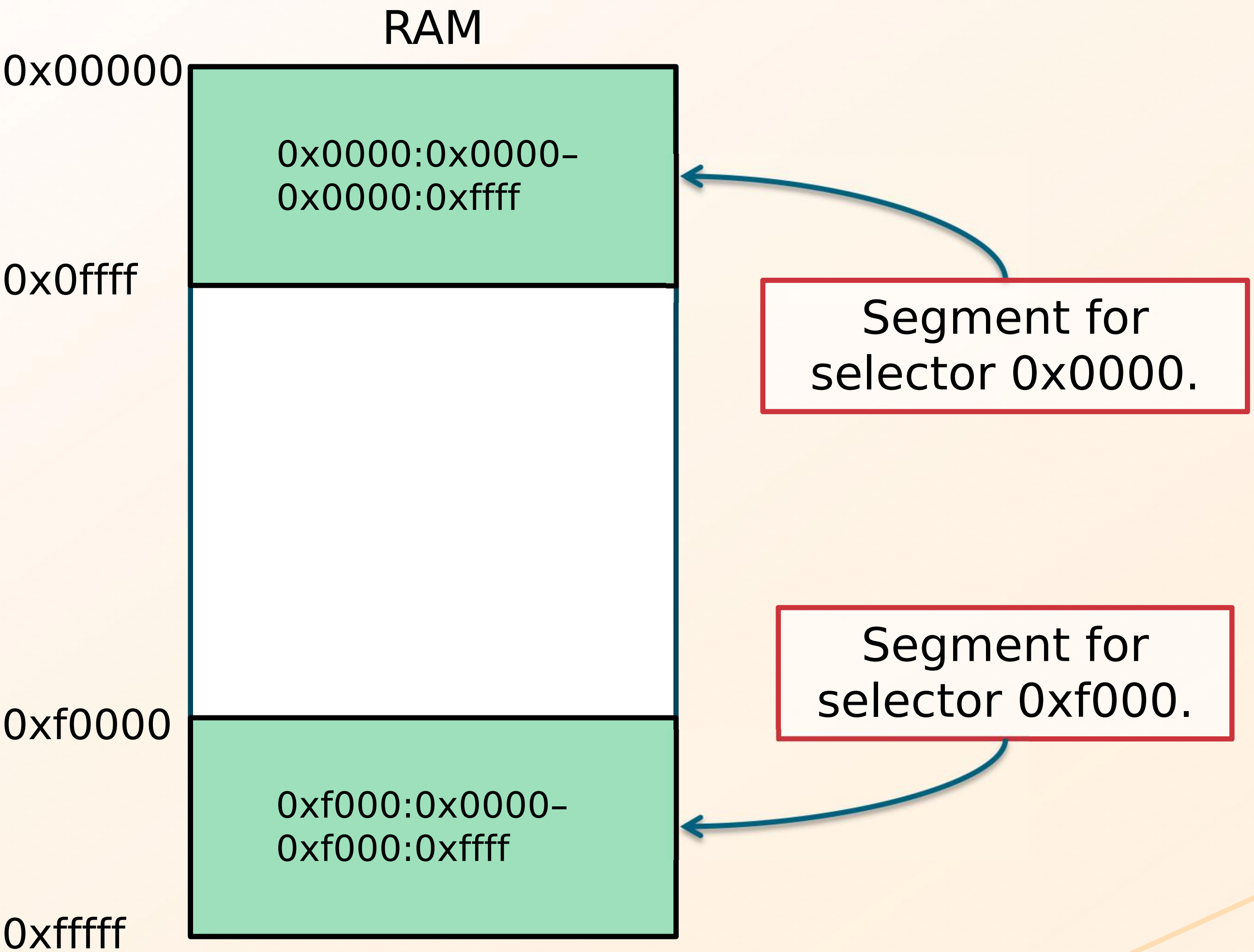
- Legacy mode – Intel 8086 architecture backwards-compatibility
- 16-bit segmented memory model
- 1MB memory (20-bit linear address space)

Real Mode segmentation



- 1MB memory space, 20-bit linear address space.
- `0x00000–0xfffff` should be addressable range.
- 16-bit logical addresses.
  - Each segment is 64KB.
- Address translation
  - Linear address = (Selector ≪ 4) + Offset
- Examples
  - `0x0000:0x0000 = 0x00000`
  - `0x0000:0xffff = 0x0ffff`
  - `0xf000:0x0000 = 0xf0000`
  - `0xf000:0xffff = 0xfffff`

Real Mode segmentation – example



© Erlend Graff, UiT

Uh…

But what about...?

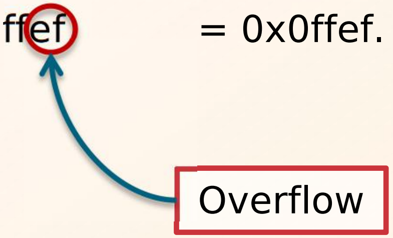
© Erlend Graff, UiT

A problem with segmentation (1)

- Selector 0x0000 addresses range 0x00000–0x0ffff.
- Selector 0xf000 addresses range 0xf0000–0xfffff.
- What about selectors 0xf001–0xffff?
- Shouldn’t 0xffff:0xffff be equal to 0x10ffef?
  - But 0x10ffef is outside of 1MB range 0x00000–0xfffff.

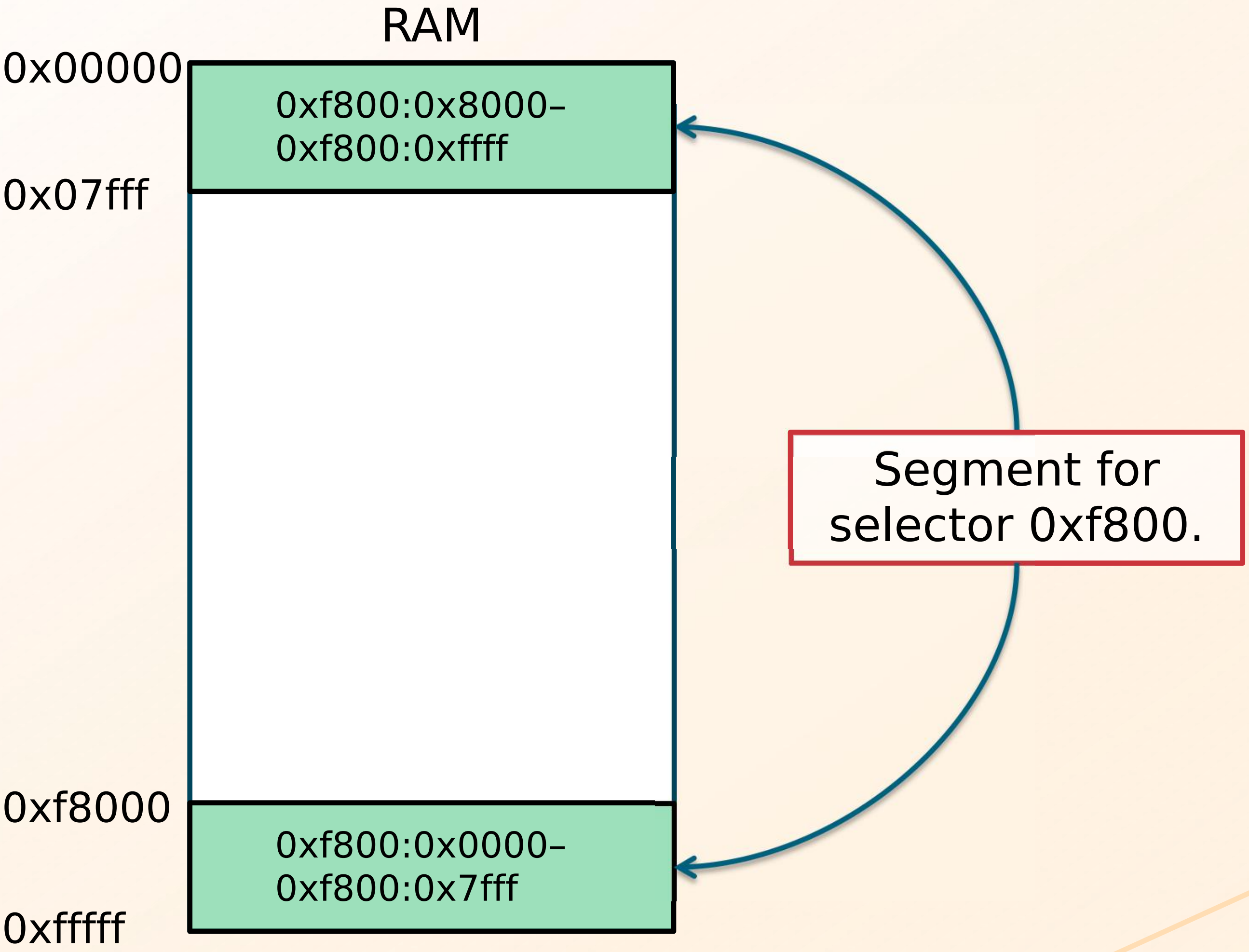
A problem with segmentation (2)

- 8086 had 20 address lines: A0–A19.
  - This is the reason for having 1MB (20-bit) linear address space in Real Mode.
  - Would be impossible to address memory outside 1MB range 0x00000–0xfffff.
  - Segments that cross 0xfffff wrap around. In essence, normal calculation of linear address, but with overflowing 20-bit addition.
  - E.g. 0xffff:0xffff = 0x10ffef = 0x0ffef.



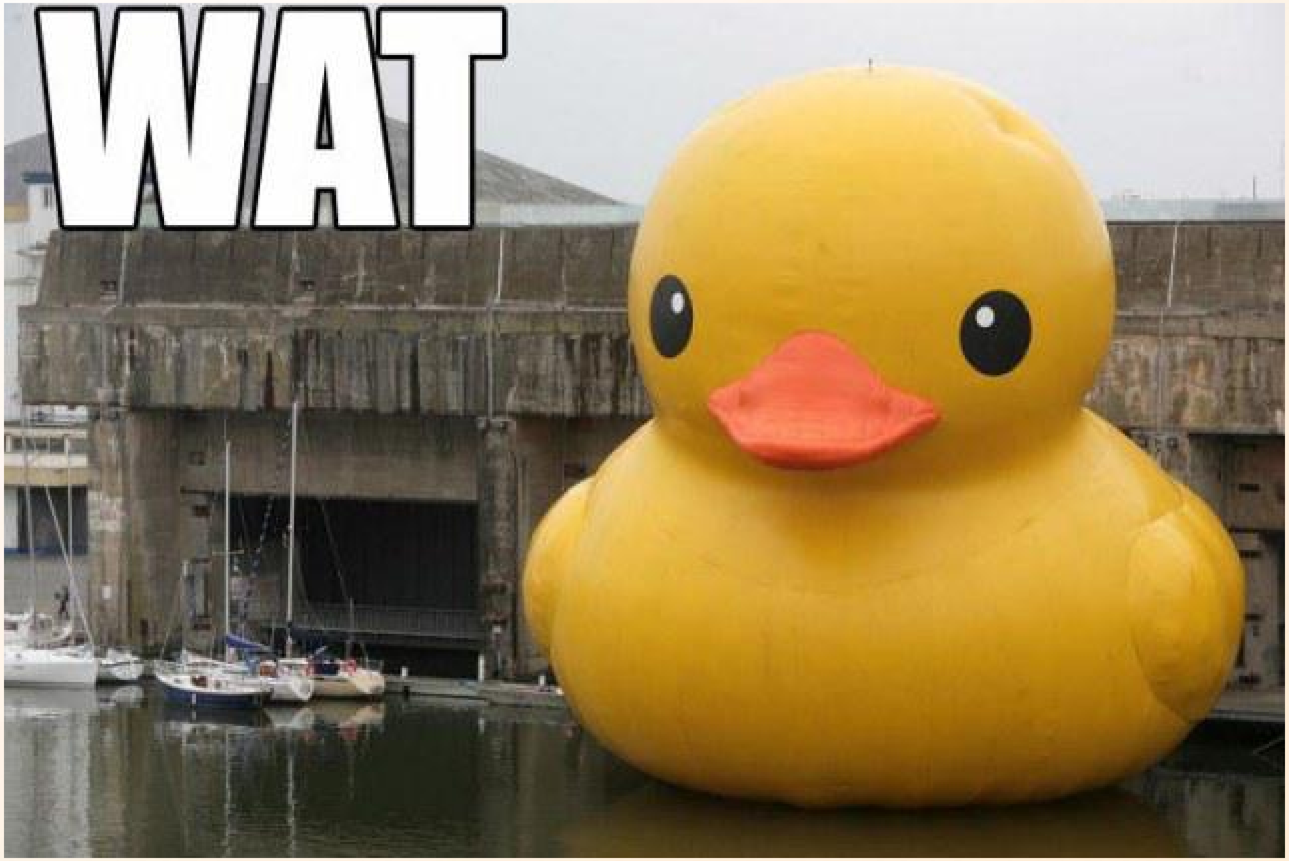
Overflow

Segmentation – example of wraparound



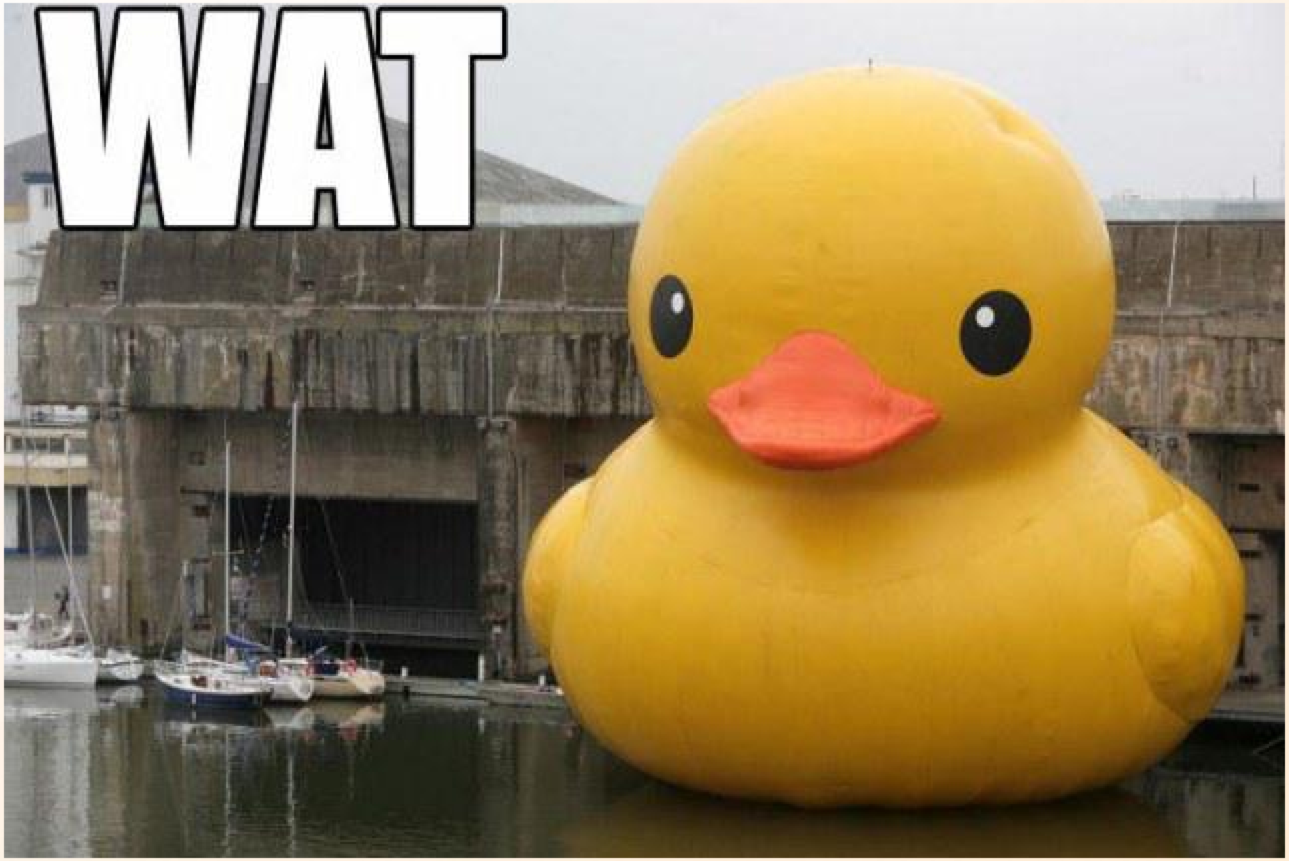
A problem with segmentation (3)

- 8086 had 20 address lines: A0–A19.
  - This is the reason for having 1MB (20-bit) linear address space in Real Mode.
  - Would be impossible to address memory outside 1MB range 0x00000–0xfffff.
  - Segments that cross 0xfffff wrap around. In essence, normal calculation of linear address, but with overflowing 20-bit addition.
  - E.g. 0xffff:0xffff = 0x10ffef = 0x0ffef.
- But 80286 had 24 address lines: A0–A23.
  - Could address 16MB (24-bit linear address space).
  - Still 64KB segments (16-bit logical addresses).
  - But segments crossing 1MB boundary do not wrap around!


WAT

A problem with segmentation (4)

- Intel 80286 would break 8086 Real Mode backwards compatibility.
- To prevent this, the 80486 introduced the A20M pin.
  - The A20M pin is asserted by default to disable the A20 address line and enforce wraparound.
  - It is part of the 8042 keyboard controller, because it (unfortunately) had a spare pin.


WAT

A problem with segmentation (4)

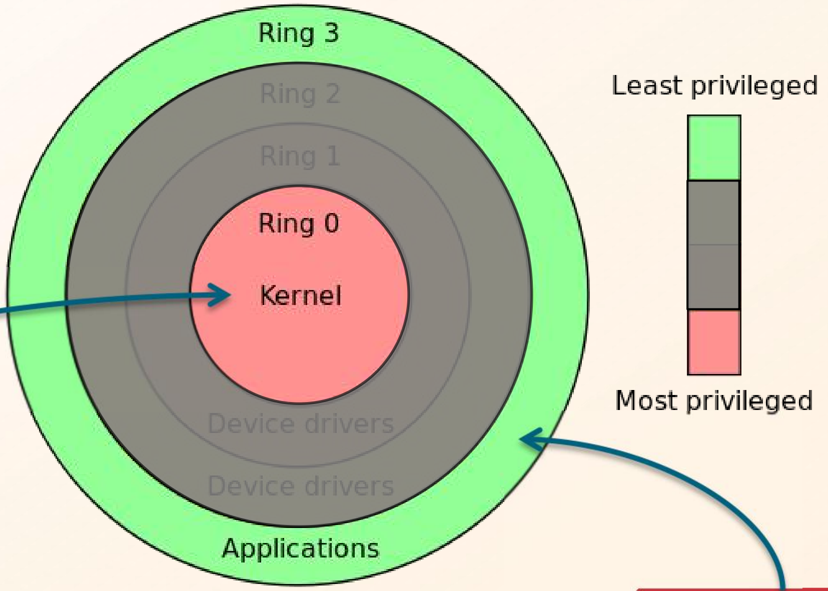
- Intel 80286 would break 8086 Real Mode backwards compatibility.
- To prevent this, the 80486 introduced the A20M pin.
  - The A20M pin is asserted by default to disable the A20 address line and enforce wraparound.
  - It is part of the 8042 keyboard controller, because it (unfortunately) had a spare pin.
- The pin must be asserted while in Real Mode, but may be deasserted to enable the A20 address line before changing processor mode.

Protected Mode

- Problems with Real Mode are the “features” of Real Mode
  - 1MB memory – too little
  - Segmented memory model – tedious
  - 16-bit is sooo 1978!
  - No protection
  - No isolation
  - No virtual memory
  - Etc.
- Protected mode
  - Introduced with 80286 as 16-bit operating mode.
  - Today’s 32-bit Protected Mode came with 80386.
  - Protected mode is the most widely used operating mode of 32-bit x86 processors.
  - Some features:
    - 4GB memory space.
    - Virtual memory / paging mechanisms with memory access rights.
    - 4 separate privilege levels for executing instructions.
    - Mechanisms for safe trapping and multi-tasking.

Protected Mode privilege levels

- Protected Mode offers 4 privilege levels
- Only 2 are used in practice!



Ring 0 used for kernel mode (Operating System)

Ring 3 used for user mode (applications)

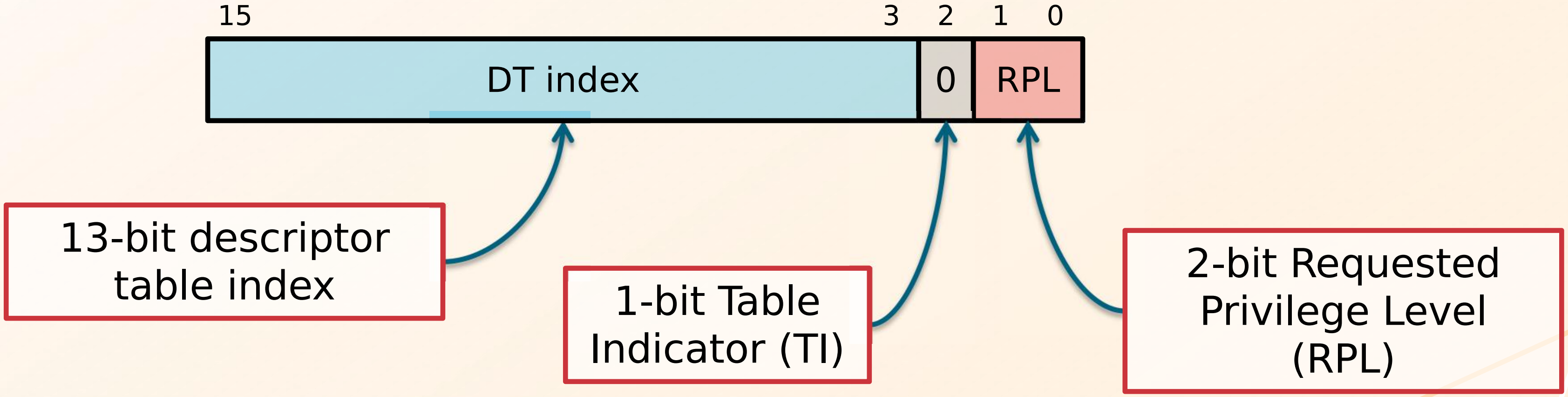
Protected Mode segmentation (1)

- More complex than in Real Mode.
- Still allows for a segmented memory model, but this is not used in practice. Instead, segments are set equal to entire 4GB address space.
- Then, why do we need segments?
  - Protection / access rights!
- Segments are no longer of fixed size. Instead, have both base and limit (size of segment).

Protected Mode segmentation (2)

- Segment registers are still in use
  - But segment registers no longer store segment base addresses in encoded form.
  - Segments registers consist of:
    - 16-bit selector (“visible”, may be written to and read from segment register).
    - 32-bit segment base address (“hidden”, in descriptor cache).
    - 32-bit segment limit (“hidden”, in descriptor cache).
    - 32-bit access rights (“hidden”, in descriptor cache).

16-bit segment selector


15
3
2
1
0
DT index
0
RPL
13-bit descriptor table index
1-bit Table Indicator (TI)
2-bit Requested Privilege Level (RPL)

Protected Mode segmentation (3)

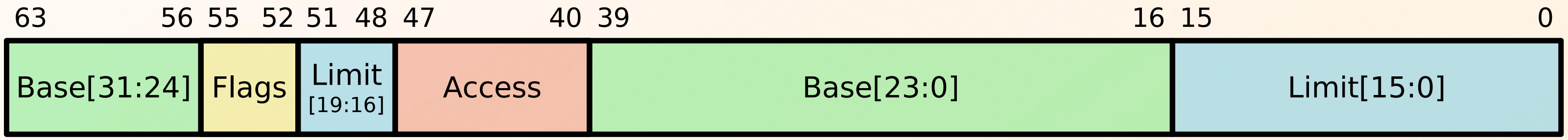
- Segment selector
  - The Requested Privilege Level (RPL) specifies the most privileged level (ring) that is allowed when loading a new segment (the segment that is loaded may only be equally or less privileged than the RPL). It corresponds to the ring number (0–3), so lower is more privileged.
  - The CS segment selector holds the Current Privilege Level (CPL) instead of an RPL.
  - The 13 MSB specifies the index into a descriptor table where a descriptor for the segment is found.
  - The Table Indicator specifies in which of two possible descriptor tables the segment descriptor is located:
    - TI = 0 – the descriptor is found in the Global Descriptor Table (GDT)
    - TI = 1 – the descriptor is found in the Local Descriptor Table (LDT)
  - In practice, the Local Descriptor Table is rarely used.

Global Descriptor Table (1)

- A fundamental data structure in x86 architecture.
- Only one instance of GDT!
- Contains 8-byte descriptors
  - One null descriptor (first descriptor, required but unused)
  - Segment descriptors
    - At least two – one for code and one for data.
    - Typically four – one pair of code and data descriptors for each privilege level.
  - Other descriptor types
    - Task State Segment (TSS) descriptors – at least (and most typically) one per CPU.
    - Local Descriptor Table (LDT) descriptors (rarely used anymore).
    - Call Gate descriptors (rarely used anymore).

Global Descriptor Table (2)

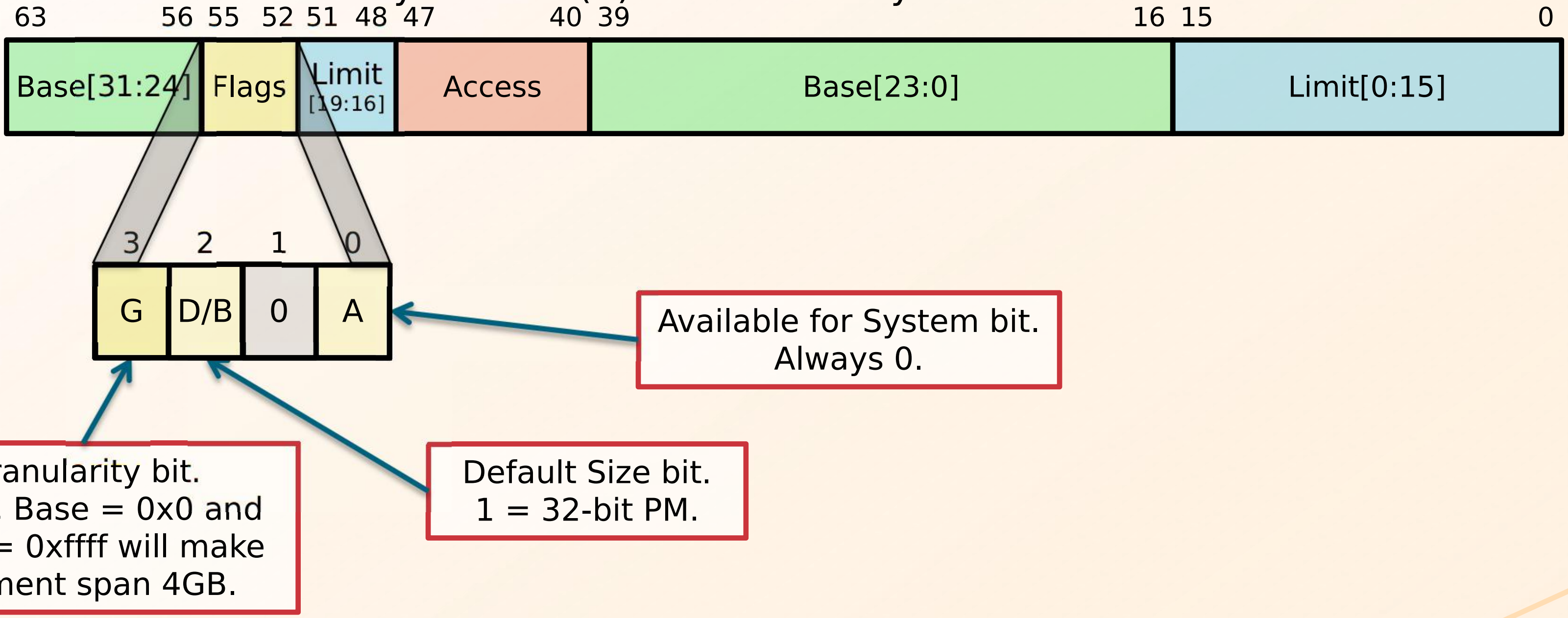
- Segment descriptors
  - Consist of
    - Segment base address (32 bit) – should be 0x00000000.
    - Segment limit (20 bit) – should be 0xfffff.
    - Flags
    - Access byte



63 56	55 52	51 48	47 40	39 16	15 0
Base[31:24]	Flags	Limit [19:16]	Access	Base[23:0]	Limit[15:0]

Global Descriptor Table (3)

- Flags
  - Granularity bit (G)
    - G = 0 – limit is in bytes, G = 1 – limit is in 4KB blocks. Should be 1.
  - Default Size bit (D/B) – should be 1 for 32-bit protected mode (0 used for 16-bit PM).
  - Available for System bit (A) – should always be 0.



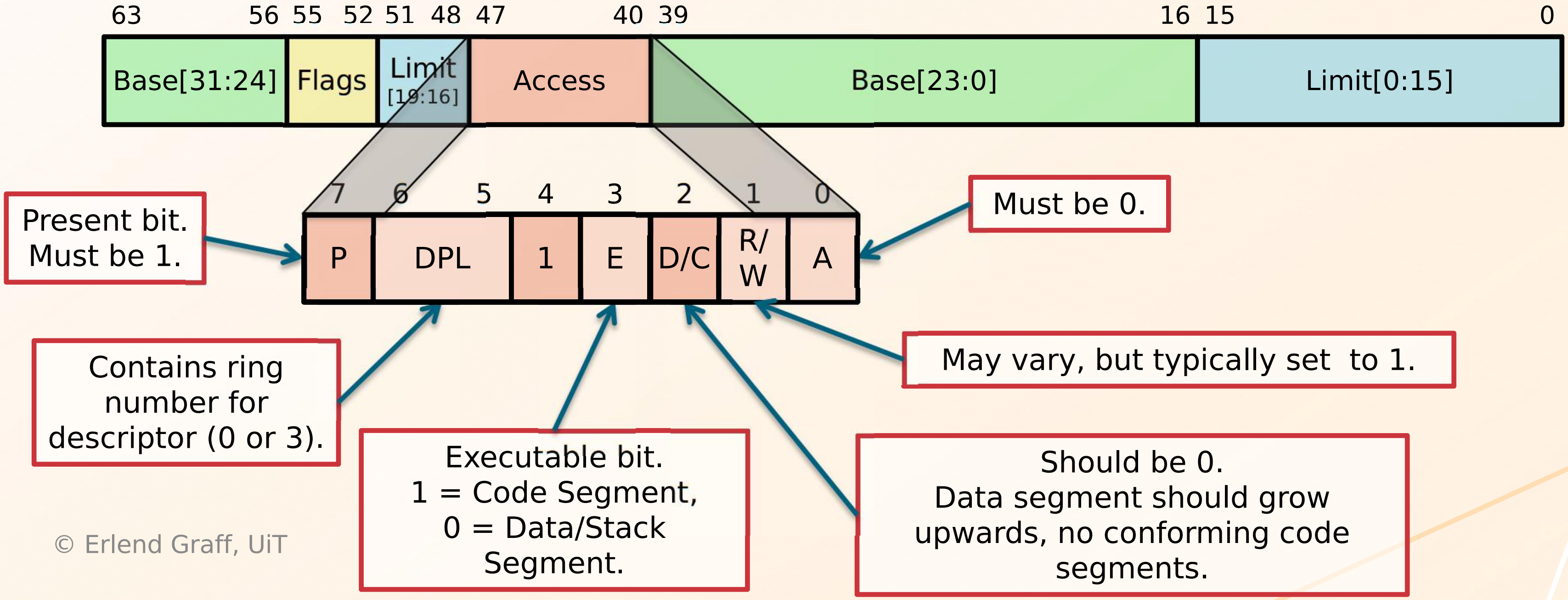
Available for System bit. Always 0.

Granularity bit. G = 1, Base = 0x0 and Limit = 0xffff will make segment span 4GB.

Default Size bit. 1 = 32-bit PM.

Global Descriptor Table (4)

- Access Byte
  - Present bit (P).
  - Descriptor Privilege Level (DPL).
  - Executable bit (E).
  - Direction/Conforming bit (D/C).
    - For stack/data segment, 0 = segment grows upwards, 1 = grows downwards.
    - If segment is code, 0 = non-conforming segment, 1 = conforming.
  - Readable/Writable bit (R/W). For code segment, 1 = readable, for data, 1 = writable.
  - Accessed bit (A).



Present bit. Must be 1.

Must be 0.

Contains ring number for descriptor (0 or 3).

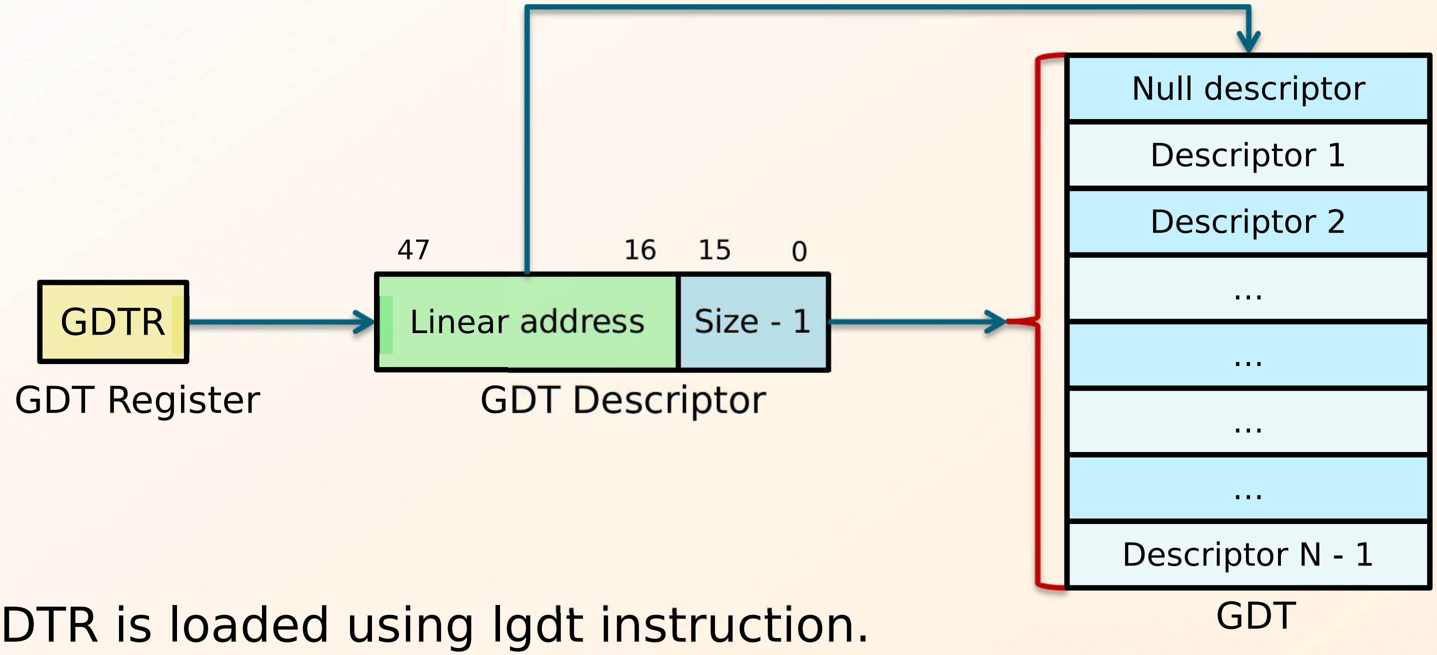
Executable bit. 1 = Code Segment, 0 = Data/Stack Segment.

Should be 0. Data segment should grow upwards, no conforming code segments.

May vary, but typically set to 1.

Global Descriptor Table (5)

- The GDT is held in memory. The GDT register (GDTR) identifies where.

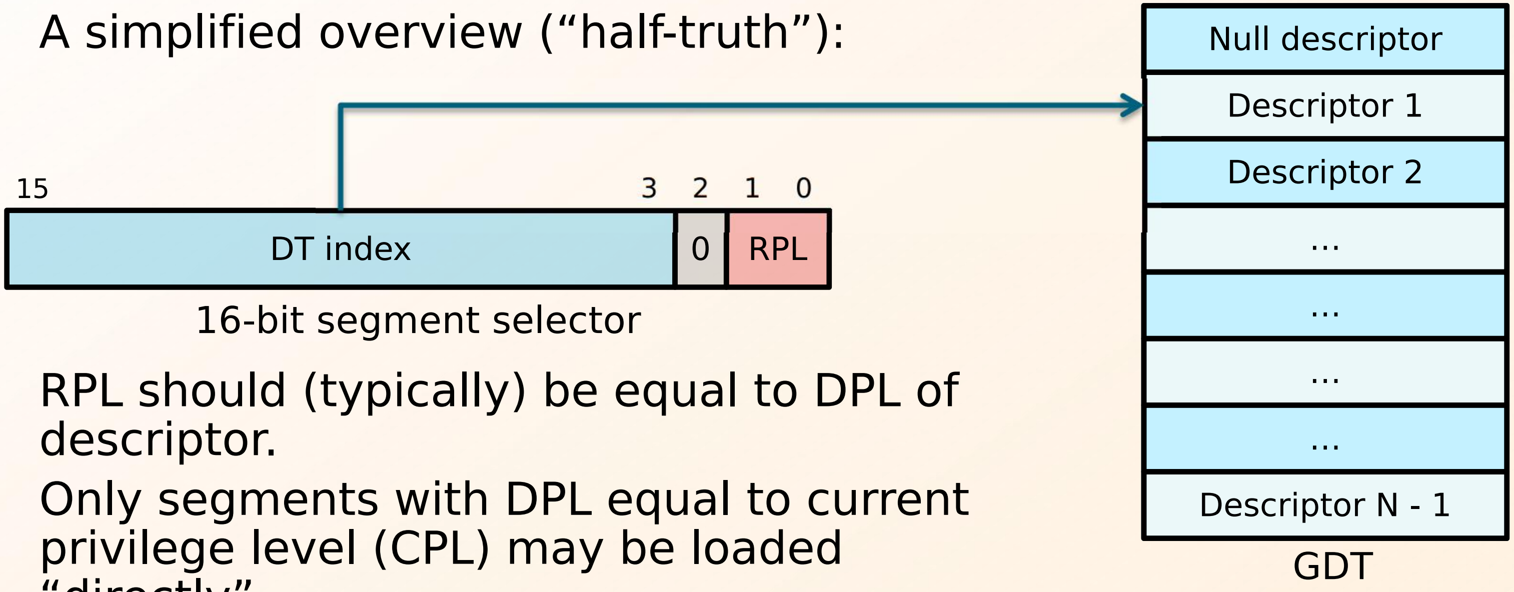


- GDTR is loaded using lgdt instruction.

© Erlend Graff, UiT

What happens when loading segment register?

- It’s complicated!
- So we ignore some of the hairy parts…
- A simplified overview (“half-truth”):



16-bit segment selector

- RPL should (typically) be equal to DPL of descriptor.
- Only segments with DPL equal to current privilege level (CPL) may be loaded “directly”.
- Base, limit and access rights of segment descriptor are stored in descriptor cache (“hidden” registers).
    - So GDT need not be accessed from memory repeatedly!

What about ring transitions?

- How do we change CPL if we only can transfer control directly to code segment with DPL equal to current CPL?
  - Trapping mechanisms
    - Interrupts (int/iret)
    - Exceptions
    - System calls (sysenter/sysleave)
    - Call gates (not used in modern OSs).

How to enable Protected Mode

1. Setup a valid GDT for Protected Mode.
2. Disable A20M to enable address line 20.
   - Allows us to use more than 1MB.
3. Disable interrupts.
4. Load GDT.
5. Enable Protection Enable (PE) bit (bit 0) in control register 0 (CR0).
6. Do a long jump to 32-bit code using valid code segment selector.
   - To clear CPU’s prefetch input queue.
7. Setup valid data segment (DS) – and preferably ES, FS and GS too!
8. Setup valid stack segment (SS) and stack for Protected Mode.
9. That’s it!

Any questions?

Any questions?

Resources

- http://www.logix.cz/michal/doc/i386/chp06-03.htm
- http://wiki.osdev.org/index.php?title=Global_Descriptor_Table&oldid=13668
- http://en.wikipedia.org/w/index.php?title=Global_Descriptor_Table&oldid=578074954
- http://duartes.org/gustavo/blog/post/cpu-rings-privilege-and-protection
- http://www.osdever.net/tutorials/view/protected-mode
- http://en.wikipedia.org/w/index.php?title=X86_memory_segmentation&oldid=581878645
- http://www.rcollins.org/ddj/Aug98/Aug98.html
- http://www.win.tue.nl/~aeb/linux/kbd/A20.html
- http://wiki.osdev.org/A20_Line