xv6©-rev10

(Copyright Frans Kaashoek, Robert Morris, and Russ Cox.)

Initialization, Segmentation

Carmi Merimovich

Tel-Aviv Academic College

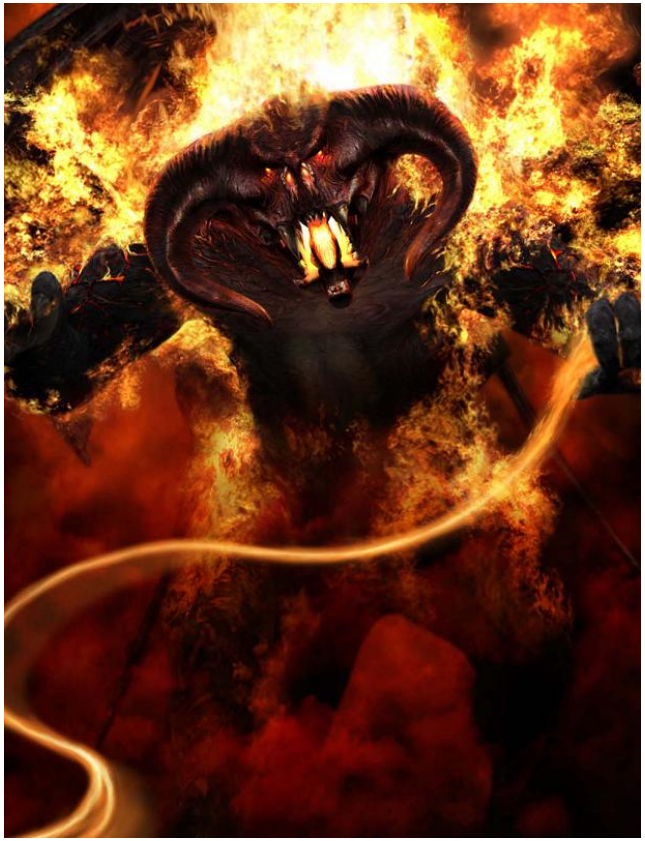
September 27, 2017

Context

```
kinit1(end, P2V(4*1024*1024)); // phys page alloca
kvmalloc();          // kernel page table
   ⋮
seginit();           // set up segments
   ⋮
pinit();             // process table
   ⋮
   ⋮
kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must co
userinit();          // first user process
mpmain();
```

Segmentation is a demon from the ancient world, just like the

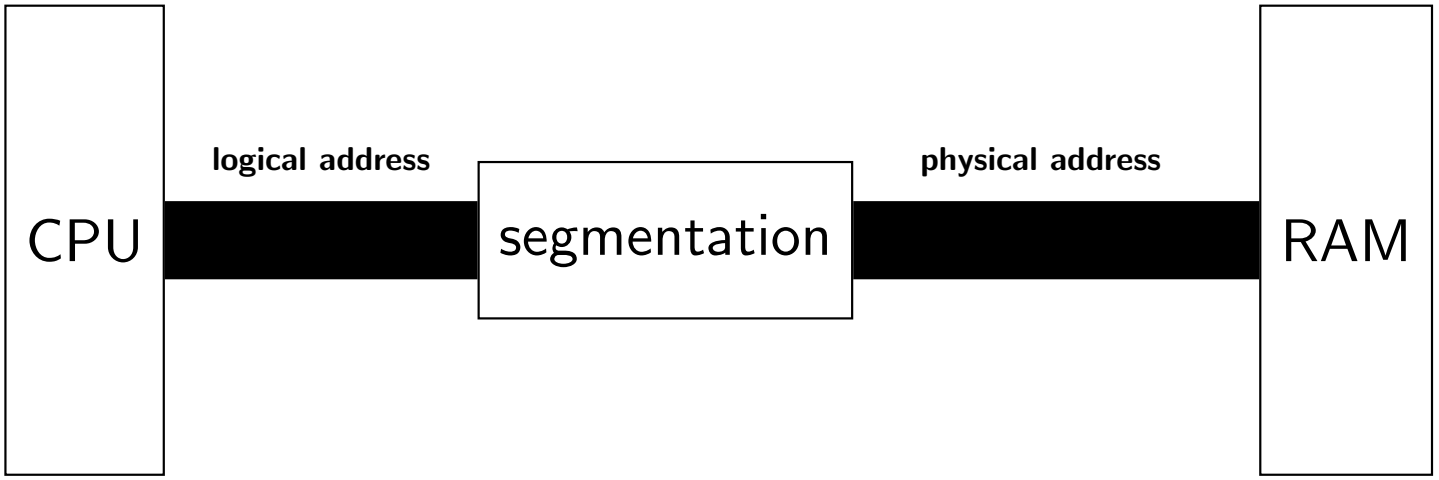
Balrog.



8086

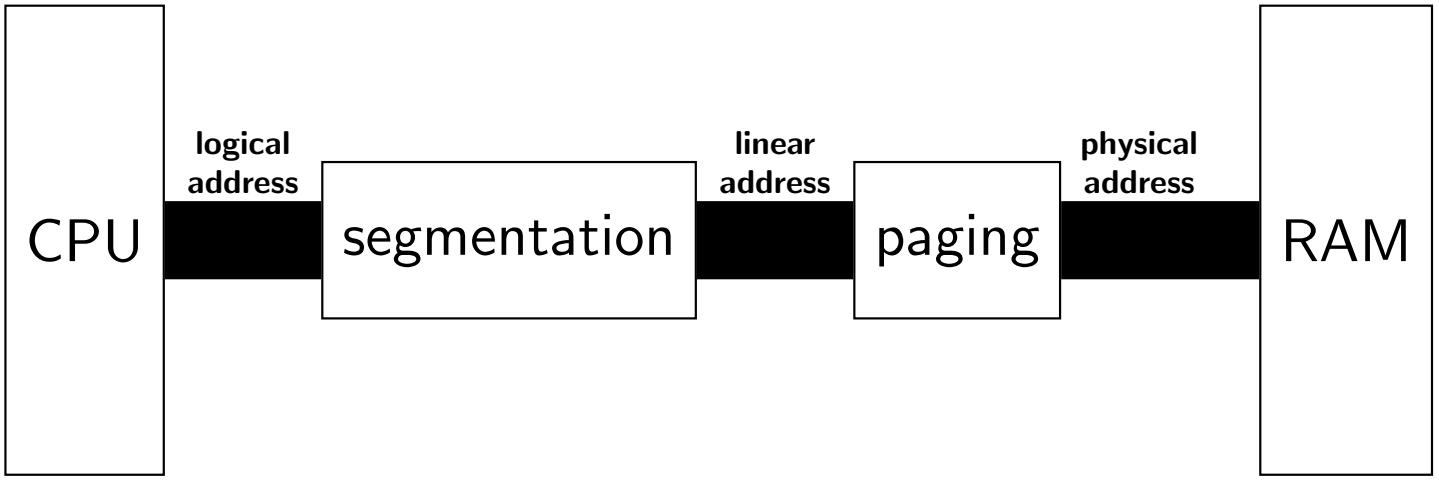
64KB sized segments aligned on 16B boundary.

80286



64KB sized segments mapped to physical addresses continuously.

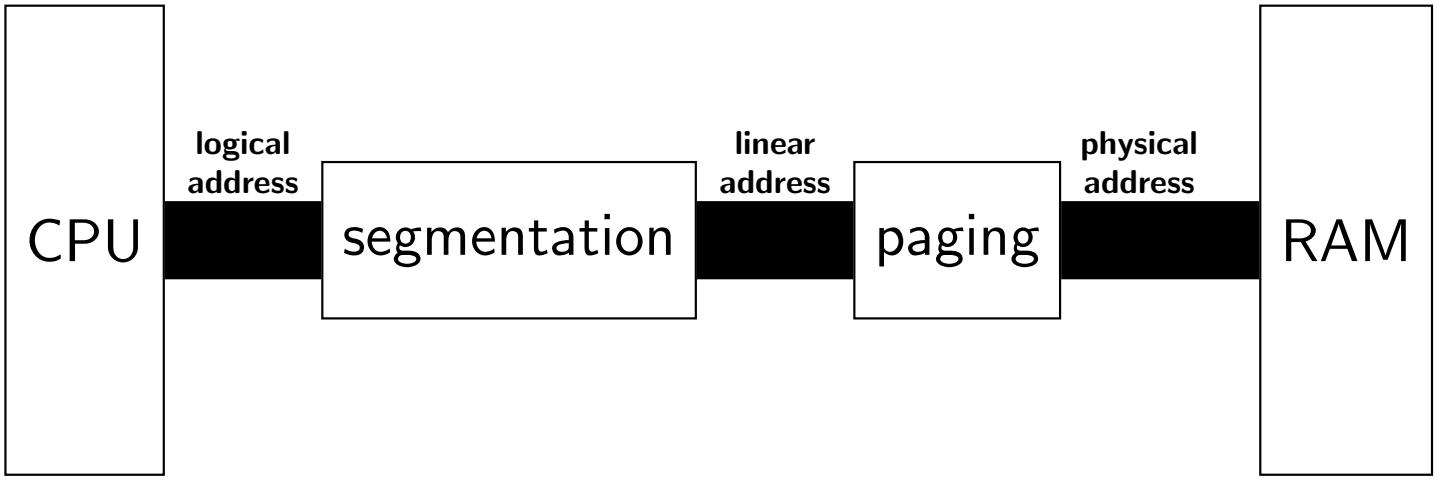
80386



4GB segments, of no need, mapped to linear addresses continuously.

Classical paging with 4KB pages. (4MB pages added later).

80386



- On boot both segmentation and paging are disabled.
- Paging cannot be enabled without first enabling segmentation.

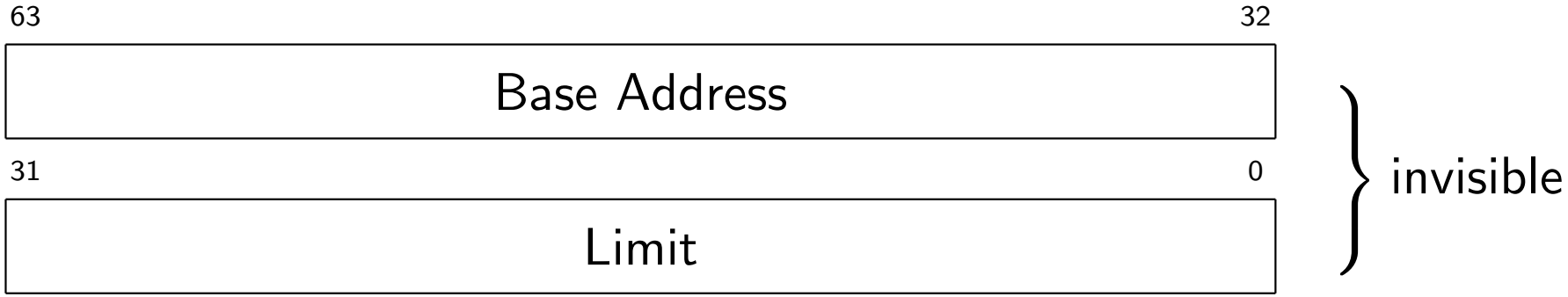
segments

- Segment is a block of linear addressable memory.
- Really a segmet is a kind of an address space.
- The substantial properties of a segment are the following two:
  - **Base Address**.
  - **Size**.
- Other properties are security/privilege/permissions etc...
- The 'usual' addresses we used until now are really addresses inside segments.
- The segment used is specified in one of the segment registers.

Segment registers

- There are six segment registers: cs, ds, ss, es, fs, gs.
- Their **visible** part is of 16 bits width:

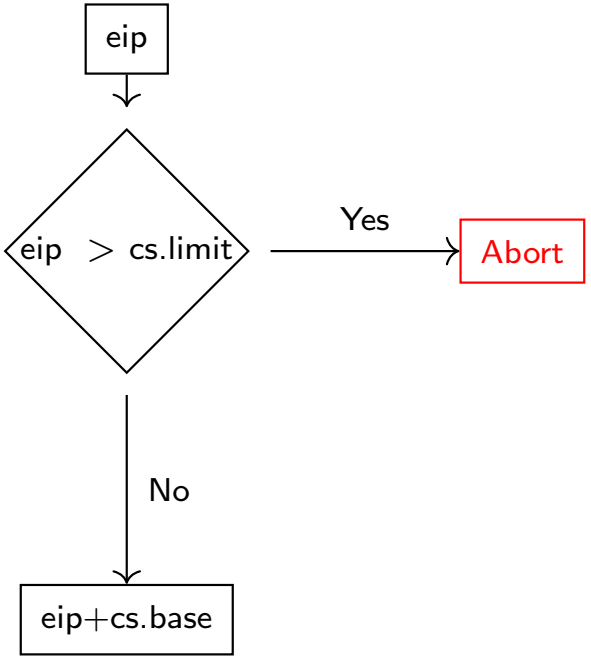
- The **invisible** part is 64-bit width as follows:



- Each **memory** operand in instruction **always** uses a segment register.
- The segment register used can be explicitly stated.
- If no segment register is specified an **implied** segment register is used.

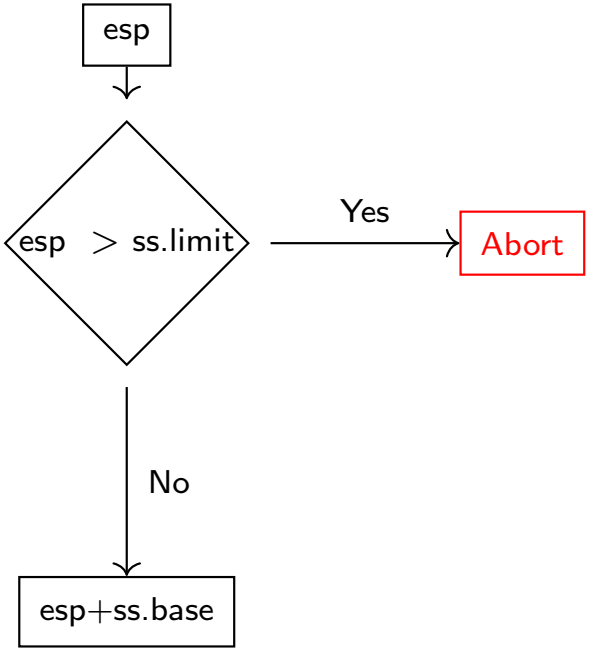
Example: Instruction fetch

- We are used to think **instruction address** is at **eip**.
- Actually it is in **cs:eip**.



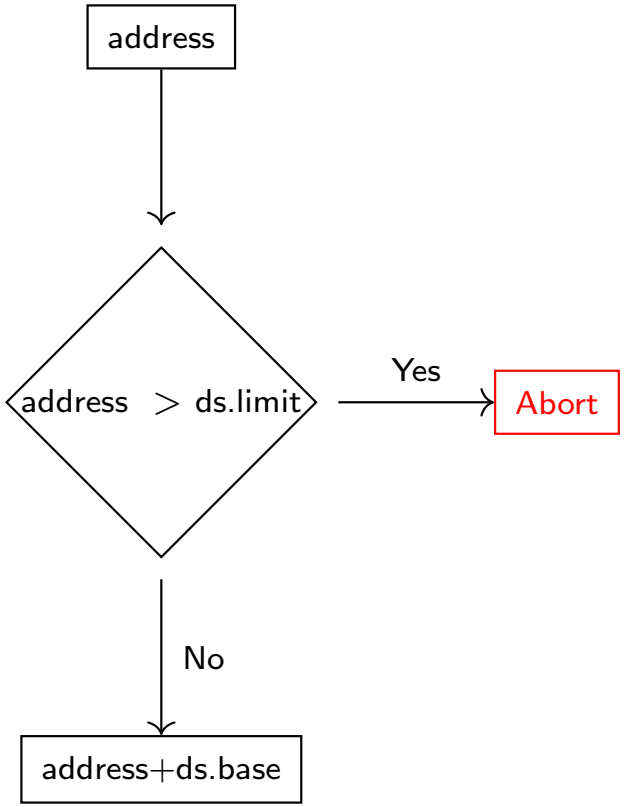
Example: Implied esp

- We are used to think **pushl/popl** use **esp**.
- Actually it uses **ss:eip**.



Example: Memory addressing modes without esp/ebp

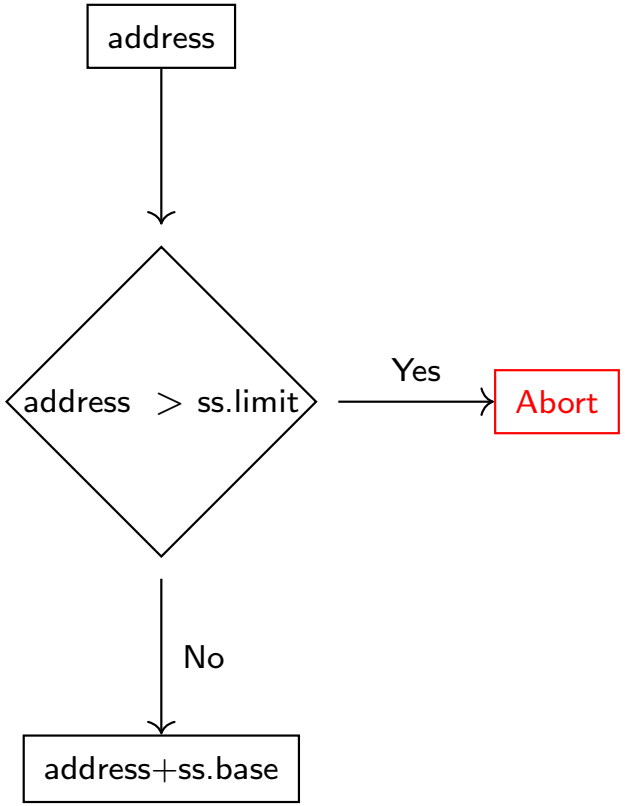
- E.g, label, 4(%edx), 8(%ecx,%ebx), 1057.
- The above are really %ds:label, %ds:4(%edx), %ds:8(%ecx,%ebx), %ds:1057.



- Override is possible: %cs:label, %es:4(%edx), %fs:8(%ecx,%ebx), %gs:1057.

Example: Memory addressing modes with either esp/ebp

- E.g, 4(%esp), 8(%ebp,%ebx).
- The above are really %ss:4(%esp), %ss:8(%ebp,%ebx).



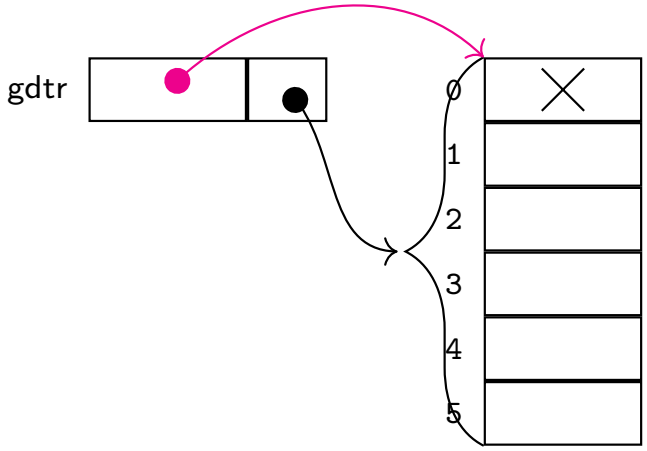
- Override is possible: %es:4(%esp), %fs:8(%ebp,%ebx).

Implied segment register rules

- Using **eip** means using **cs:eip**.
- Using implied **esp** means using **ss:esp**.
- Naturaly, there is no segment override for implied operands.
- Memory access using either **esp** or **ebp** uses the **ss** register.
- Memory access using neither the **esp** nor the **ebp** uses the **ds** register.
- Explicit memory references can always use segment override.

Global Descriptor Table

- The **GDT** is a vector pointed to by the **gdtr** register.
- Entry zero of the **GDT** is not used.
- Each entry of the **GDT** describes a memory block.
- The addresses in the **GDT** are physical or virtual.
- (Depending on the state of the paging unit).



General GDT entry

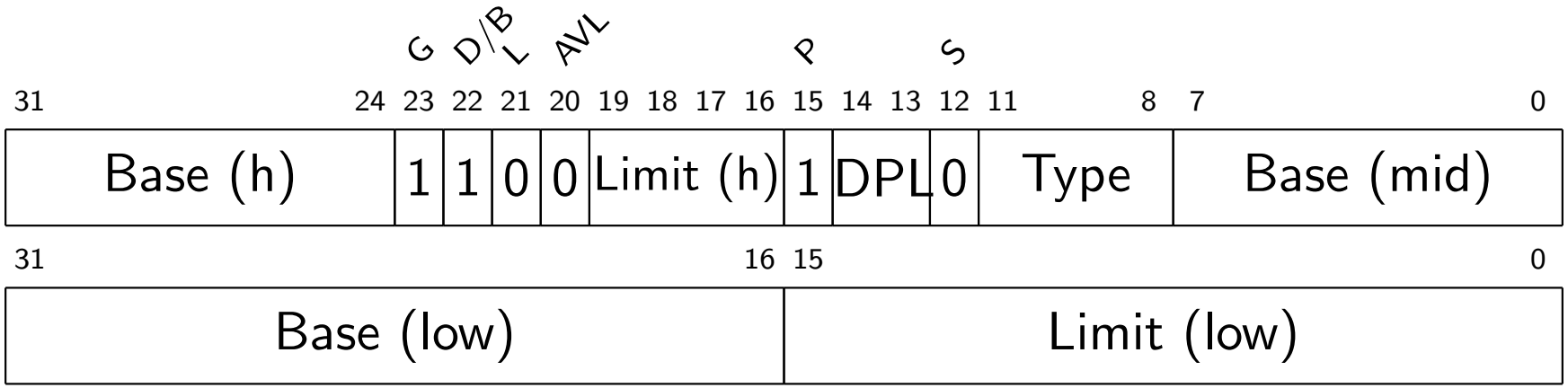


Figure: Descriptor

- DPL:
  - 00 - Kernel segment.
  - 11 - User segment.
- Type:
  - `0x2`: R/W data segment.
  - `0xA`: E/R code segment.

Segment register loading

- Program loads value into visible part of sreg.
- The processor loads the invisible parts from the GDT, permission allows. E.g.,

```
movw  $8,%ax
movw  %ax,%ds
```

Permission allows, **ds.base** and **ds.limit** are loaded from **gdt[1].base** and **gdt[1].limit**.

Basically we DO NOT WANT segmentation.
We will attempt to get the identity function.

Identity GDT entry

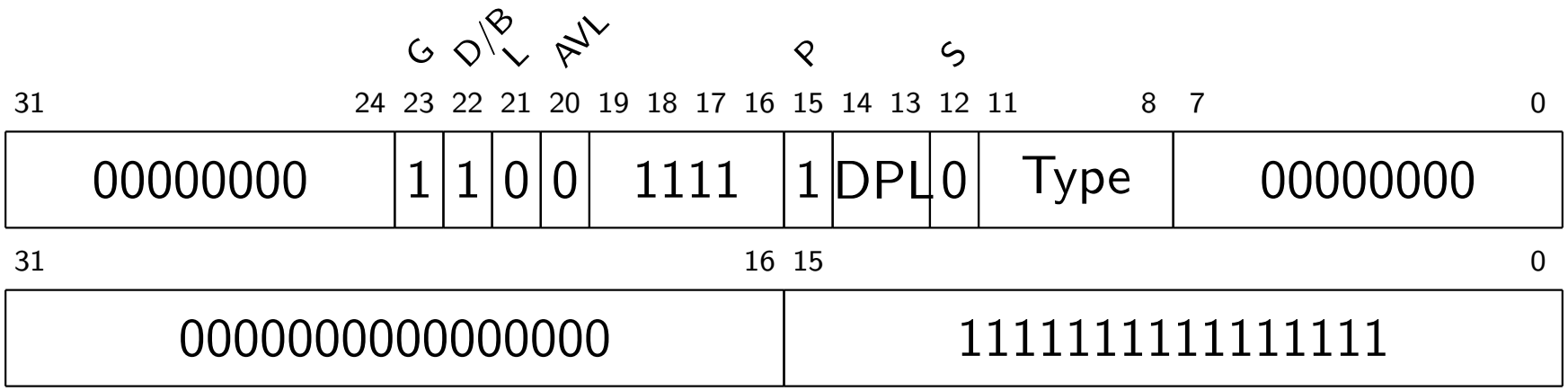


Figure: Descriptor

- Base: mininum, I.e., 0.
- Limit: Maximum, i.e., FFFFFFFF.

Kernel code GDT entry

	G	D/B	L	AVL		P		S		
31 … 24	23	22	21	20	19 … 16	15	14 … 13	12	11 … 8	7 … 0
00000000	1	1	0	0	1111	1	00	0	1010	00000000

31 … 16	15 … 0
0000000000000000	1111111111111111

Figure: Descriptor

```
#define STA_X 0x8 // Executable segment
#define STA_W 0x2 // Writeable (non executable seg
#define STA_R 0x2 // Readable (executable segments)

SEG(STA_X|STA_R,0,0xffffffff,0);
```

Kernel data GDT entry

	G	D/B	L	AVL		P		S		
31 … 24	23	22	21	20	19 … 16	15	14 … 13	12	11 … 8	7 … 0
00000000	1	1	0	0	1111	1	00	0	0010	00000000

31 … 16	15 … 0
0000000000000000	1111111111111111

Figure: Descriptor

- Base: mininum, I.e., 0.
- Limit: Maximum, i.e., FFFFFFFF.

```
#define STA_X 0x8 // Executable segment
#define STA_W 0x2 // Writeable (non executable seg
#define STA_R 0x2 // Readable (executable segments)

SEG(STA_W, 0,0xffffffff, 0);
```

User code GDT entry

	G	D/B	L	AVL		P		S		
31 … 24	23	22	21	20	19 … 16	15	14 … 13	12	11 … 8	7 … 0
00000000	1	1	0	0	1111	1	11	0	1010	00000000

31 … 16	15 … 0
0000000000000000	1111111111111111

Figure: Descriptor

```
#define STA_X 0x8 // Executable segment
#define STA_W 0x2 // Writeable (non executable seg
#define STA_R 0x2 // Readable (executable segments)

SEG(STA_X|STA_R,0,0xffffffff,DPL_USER);
```

User data GDT entry

	G	D/B	L	AVL		P		S		
31 … 24	23	22	21	20	19 … 16	15	14 … 13	12	11 … 8	7 … 0
00000000	1	1	0	0	1111	1	11	0	0010	00000000

31 … 16	15 … 0
0000000000000000	1111111111111111

Figure: Descriptor

```
#define STA_X 0x8 // Executable segment
#define STA_W 0x2 // Writeable (non executable seg
#define STA_R 0x2 // Readable (executable segments)

SEG(STA_W,        0,0xffffffff,DPL_USER);
```

xv6 SEG macro

```
752  struct segdesc {
      uint lim_15_0 : 16; // Low bits of segment limit
      uint base_15_0 : 16; // Low bits of segment base address
      uint base_23_16 : 8; // Middle bits of segment base address
      uint type : 4; // Segment type (see STS_ constants)
      uint s : 1; // 0 = system, 1 = application
      uint dpl : 2; // Descriptor Privilege Level
      uint p : 1; // Present
      uint lim_19_16 : 4; // High bits of segment limit
      uint avl : 1; // Unused (available for software use)
      uint rsv1 : 1; // Reserved
      uint db : 1; // 0 = 16 bit segment, 1 = 32 bit segment
      uint g : 1; // Granularity: limit scaled by 4K when set
      uint base_31_24 : 8; // High bits of segment base address
     };

     #define SEG(type, base, lim, dpl) (struct segdesc) \
     { ((lim) >> 12) & 0xffff, (uint)(base) & 0xffff, \
     ((uint)(base) >> 16) & 0xff, type, 1, dpl, 1, \
     (uint)(lim) >> 28, 0, 0, 1, 1, (uint)(base) >> 24 }
```

The **gdt** on entering **main**

```
#define SEG_KCODE 1 // kernel code
#define SEG_KDATA 2 // kernel data+stack

gdt[SEG_KCODE]=SEG(STA_X|STA_R,0,0xffffffff,0);
gdt[SEG_KDATA]=SEG(STA_W,      0,0xffffffff,0);

lgdtr(gdt, sizeof(gdt));
```

xv6 gdt to support user mode

Since we want Code/Data and Kernel/User we need at least 4 segments.

```
#define SEG_KCODE 1 // kernel code
#define SEG_KDATA 2 // kernel data+stack
#define SEG_UCODE 3 // user code
#define SEG_UDATA 4 // user data+stack

gdt[SEG_KCODE]=SEG(STA_X|STA_R,0,0xffffffff,0);
gdt[SEG_KDATA]=SEG(STA_W,      0,0xffffffff,0);
gdt[SEG_UCODE]=SEG(STA_X|STA_R,0,0xffffffff,DPL_USER
gdt[SEG_UDATA]=SEG(STA_W,      0,0xffffffff,DPL_USER

lgdtr(gdt, sizeof(gdt));
```

- Each cpu should execute the last instruction!

xv6 segment registers values

	Kernel	User
cs	8	27
ds	16	35
ss	16	35

xv6 Per-processor structure

```
2301 struct cpu {
      uchar apicid; // Local APIC ID;
      struct context *scheduler; // swtch() here to enter
      struct taskstate ts; // Used by x86 to find stack f
      struct segdesc gdt[NSEGS]; // x86 global descriptor
      volatile uint started; // Has the CPU started?
      int ncli; // Depth of pushcli nesting.
      int intena; // Were interrupts enabled before pushc
      struct proc *proc; // The currently running proce
     };

     extern struct cpu cpus[NCPU];
```

seginit

```
1715 void seginit(void) {
     struct cpus *c = &cpus[cpuid()];
     c->gdt[SEG_KCODE]=SEG(STA_X|STA_R,0,0xffffffff,0);
     c->gdt[SEG_KDATA]=SEG(STA_W,      0,0xffffffff,0);
     c->gdt[SEG_UCODE]=SEG(STA_X|STA_R,0,0xffffffff,DPL_
     c->gdt[SEG_UDATA]=SEG(STA_W,      0,0xffffffff,DPL_
     lgdtr(c->gdt, sizeof(c->gdt));
    }
```

cpuid and mycpu

```
2429 int cpuid() {
      return mycpu()-cpus;
     }

     struct cpu *mycpu(void) {
      int apicid, i;

      if (readeflags()&FL_IF)
       panic("mycpu called with interrupts enabled\n");

      apicid = lapicid();
      for (i = 0; i < ncpu; ++i) {
       if (cpus[i].apicid == apicid)
        return &cpus[i];
      }
      panic("unknown apicid\n");
     }
```