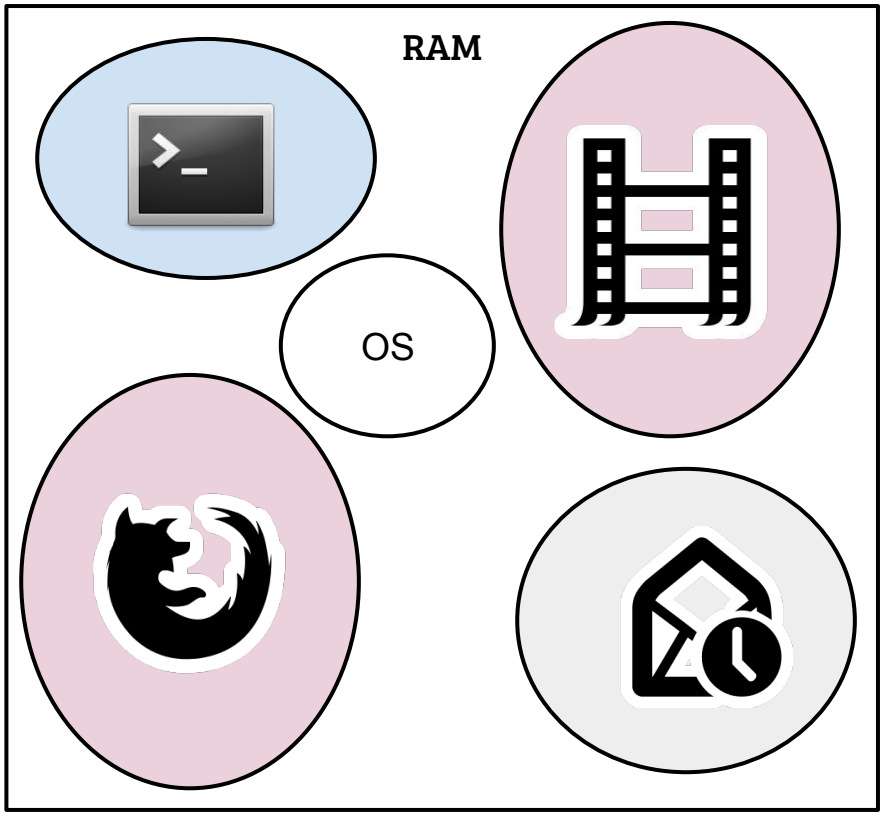
Operating Systems

Virtual Memory

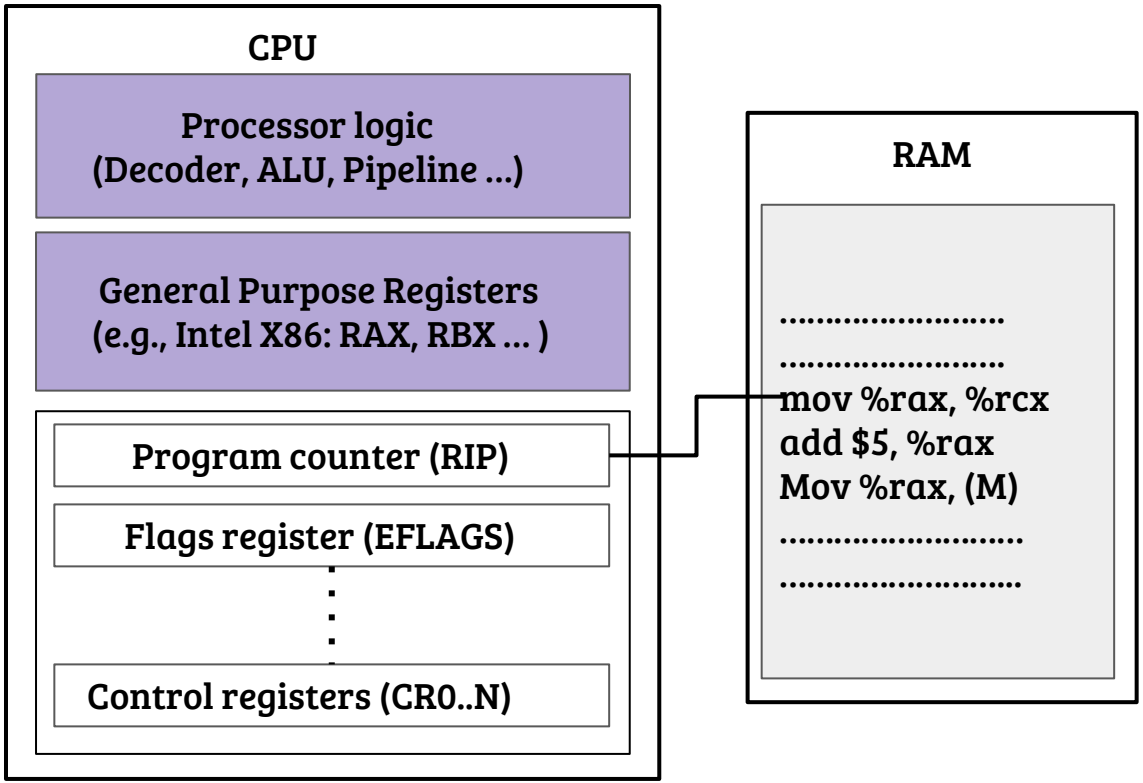
Debadatta Mishra, CSE, IITK

Memory partitioning



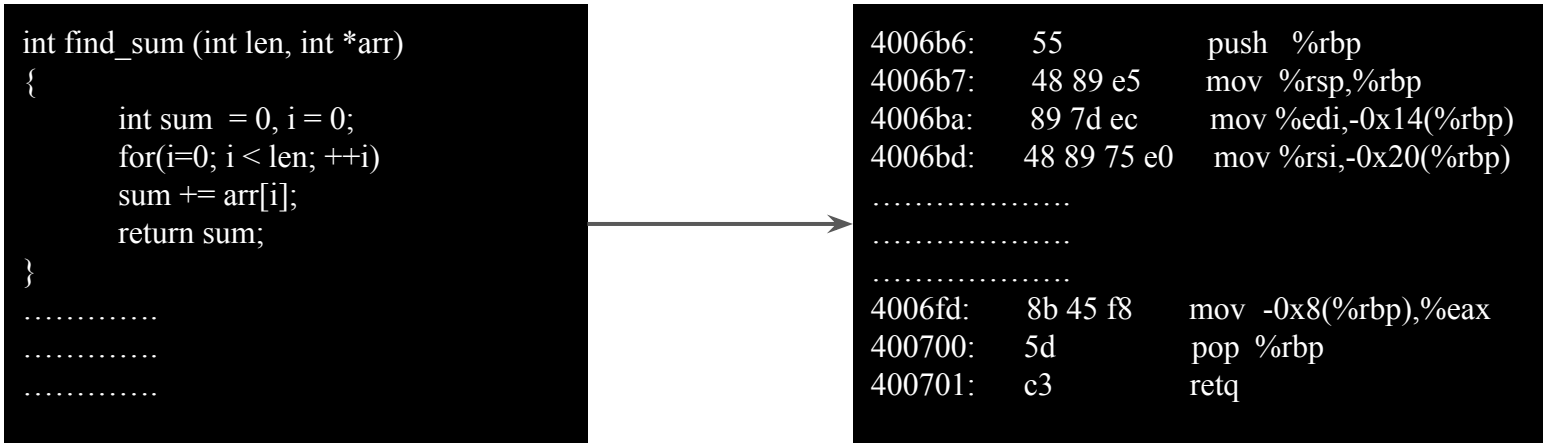
- ➔ Isolated memory allocation
  - ◆ Inter-application isolation
  - ◆ Intra-application isolation
- ➔ Static vs. Dynamic
  - ◆ Flexibility
  - ◆ Memory utilization
  - ◆ Degree of multiprogramming
- ➔ Memory used by OS need to be isolated from all applications
- ➔ However, system calls require access to OS memory

Computing model and ISA



- ➔ PC points to the instruction (pointing to memory address)
- ➔ Instructions can address
  - Registers
  - Memory (data)
  - Stack
- ➔ All memory accessed should be in memory, or else?

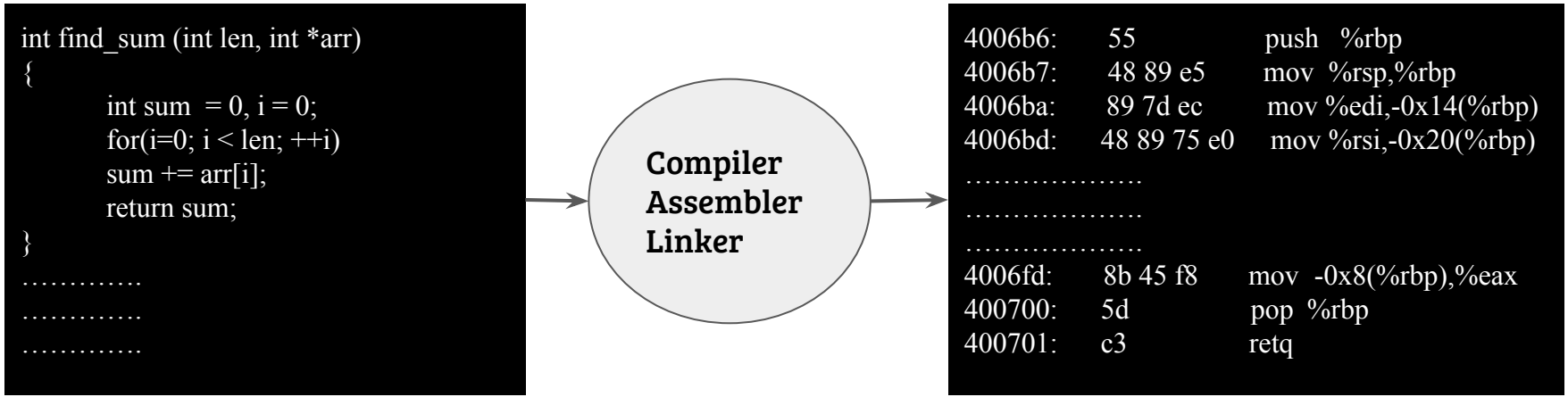
Application program → Execution



➔ High-level language → Assembly code → Machine code

➔ Application program: memory containing program, data and stack

Application program → Execution



- ➔ Program code, stack and data → Which addresses?
- ➔ Code, stack and data addresses are interdependent, why?
- ➔ When to decide the symbol to address assignment (a.k.a address binding)?

Example code

```
u64 value = 0x500;
u64 array[10];
main ( )
{
        u64 ctr = 0;
        for (ctr=0; ctr<10; ctr++)
             array[i] += value;
}
```

- ➔ Increments each element of array by “value”
- ➔ How exactly the symbols translate to address?

Address binding: compile time

```
0x1000:    mov $0, %rcx
0x1004:    mov  $0x20000, %rsp
0x1008:    push %rbp
0x100a:    mov %rsp, %rbp
0x100c:    mov ($0x10008, %rcx, 0x8), %rdi
0x1010:    add ($0x10000), %rdi
0x1014:    mov  %rdi, ($0x10008,%rcx, 0x8 )
0x1018:    inc %rcx
0x101a:    xor %rcx, $10
0x101c:    jnz $0x1010

/*Initialization*/
0x10000: .long 0x500
0x10008: .long 0    REP (10)
0x20000: .long 0x0
```

- ➔ Compiler assigns absolute addresses to instructions and variables during compilation
- ➔ Isolation
- ➔ Flexibility
- ➔ Resource utilization, degree of multiprogramming

Address binding: load time (OS)

```
0x0:    mov $0, %rcx
0x4:    mov  $0x19000, %rsp
0x8:    push %rbp
0xa:    mov %rsp, %rbp
0xc:    mov ($0x9008, %rcx, 0x8), %rdi
0x10:  add ($0x9000), %rdi
0x14:  mov  %rdi, ($0x9008,%rcx, 0x8 )
0x18:   inc %rcx
0x1a:   xor %rcx, $10
0x1c:   jnz $0x10

/*Initialization*/
0x9000:   .long 0x500
0x9008:   .long 0    REP (10)
0x19000: .long 0x0
```

- ➔ Compiler assigns relative addresses (relative to what?) to instructions and variables during compilation
- ➔ OS converts the addresses to absolute addresses while loading
- ➔ Isolation
- ➔ Flexibility
- ➔ Resource utilization, degree of multiprogramming

Address binding: runtime (Hw + OS)

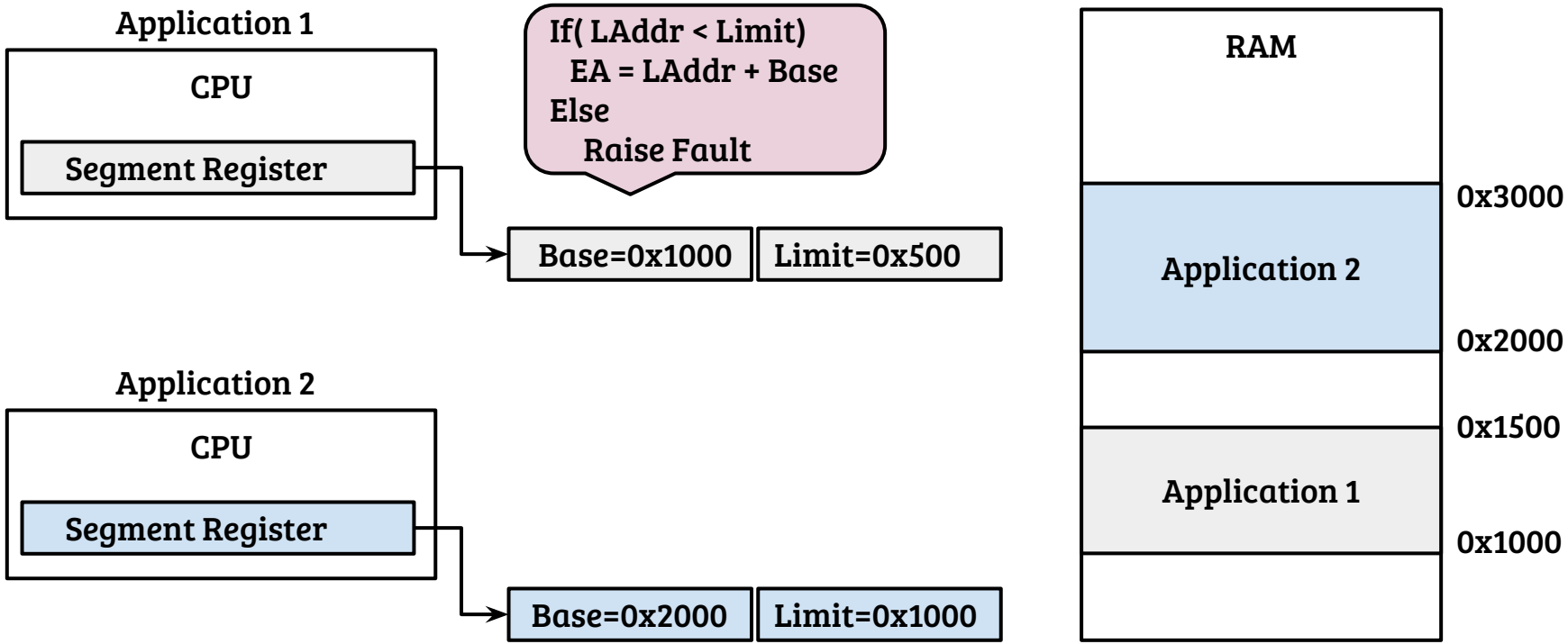
```
code + 0x0:    mov $0, %rcx
code + 0x4:    mov  %stack, %rsp
code + 0x8:    push %rbp
code + 0xa:    mov %rsp, %rbp
code + 0xc:    mov (%data + 0x8, %rcx, 0x8), %rdi
code + 0x10:  add (%data), %rdi
code + 0x14:  mov  %rdi, (%data + 0x8,%rcx, 0x8 )
code + 0x18:   inc %rcx
code + 0x1a:   xor %rcx, $10
code + 0x1c:   jnz (%code + $0x10)

/*Initialization*/
(%data)            .long 0x500
(%data + 0x8)   .long 0    REP (10)
(%stack)           .long 0x0
```

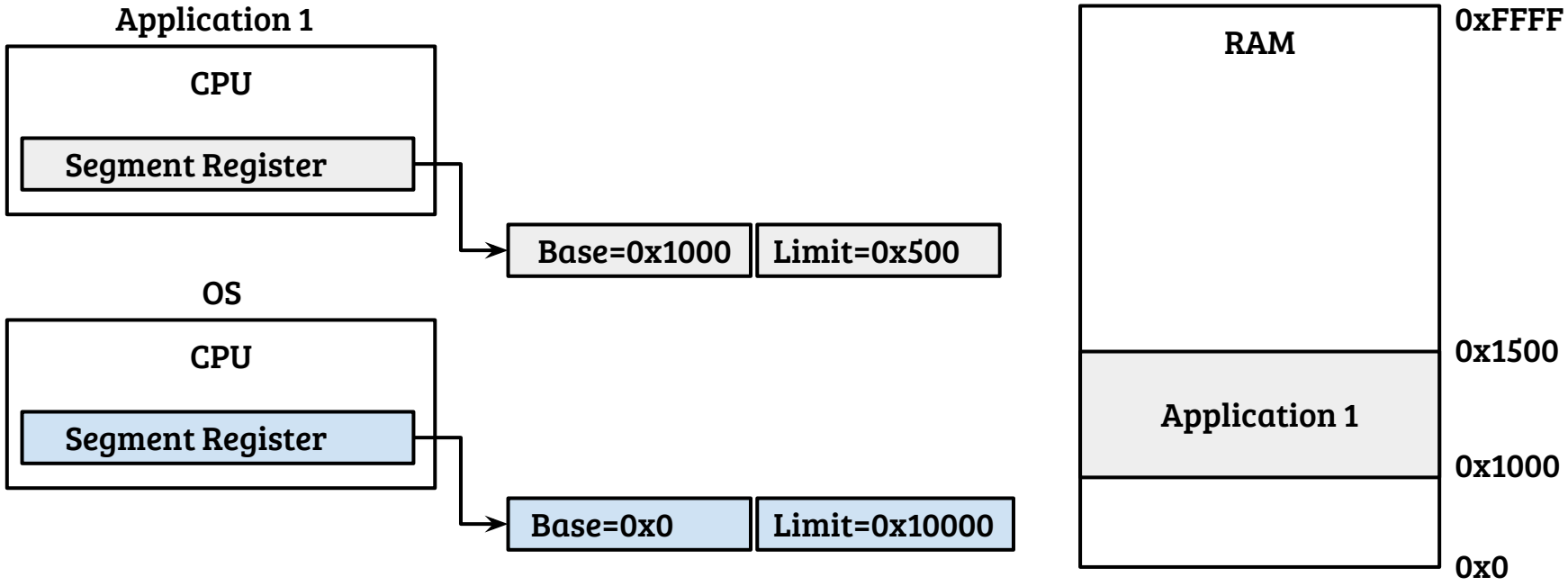
code = 0x1000, data = 0x10000, stack = 0x20000

- ➔ Compiler assigns relative addresses to instructions and variables during compilation
- ➔ Hardware registers contain the base, can change dynamically
- ➔ Isolation
- ➔ Flexibility
- ➔ Resource utilization, degree of multiprogramming

Memory segmentation



Memory segmentation: What about OS segment?



➔ OS segment should be a superset of all applications, why? alternates?

➔ In OS, application segment information is required to be accessed, why?

Segmentation

- ➔ Advantages
  - Simple operations to translate program address to effective address
  - Isolation enforcement
  - Reflects compilers view of a program
- ➔ Disadvantages
  - Conservative provisioning, fragmentation → Inefficient memory utilization
  - Multiplexing granularity is limited → Degree of multiprogramming suffers
  - Large overhead to achieve over-provisioning
  - Memory sharing is at segment granularity

Segmentation: granularity issue

- ➔ Theoretically, segmentation is not a problem
  - ◆ We can have “a lot of” segments for an active application
  - ◆ Issues like memory efficiency, degree of multiprogramming can be addressed
  - ◆ One caveat though, which is?

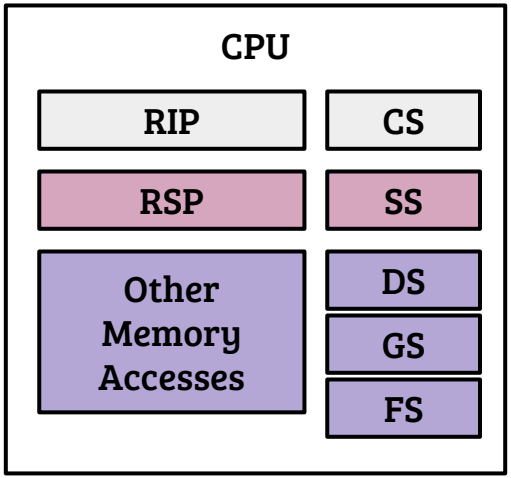
Segmentation: granularity issue

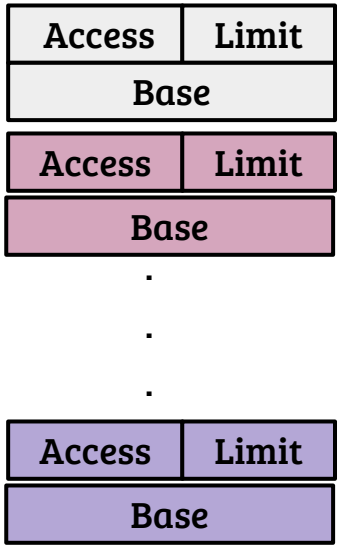
- ➔ Theoretically, segmentations is not a problem
  - ◆ We can have “a lot of” segments for an active application
  - ◆ Issues like memory efficiency, degree of multiprogramming can be addressed
  - ◆ One caveat though, which is?
- ➔ How can we address the hardware limitation?
  - ◆ Segmentation is like a guided (through pointers) one-step translation mechanism
  - ◆ Can we expand it to multi-step lookup?
  - ◆ Data structures?

Segmentation: granularity issue

- ➔ Theoretically, segmentations is not a problem
  - ◆ We can have “a lot of” segments for an active application
  - ◆ Issues like memory efficiency, degree of multiprogramming can be addressed
  - ◆ One caveat though, which is?
- ➔ How can we address the hardware limitation?
  - ◆ Segmentation is like a guided (through pointers) one-step translation mechanism
  - ◆ Can we expand it to multi-step lookup?
  - ◆ Data structures?
- ➔ Design attributes
  - ◆ Minimize translation overheads → lookup latency, memory usage
  - ◆ Support for sparse and dynamic mappings → lazy allocation, memory sharing
  - ◆ Protection, Isolation etc.

Memory segmentation (X86)



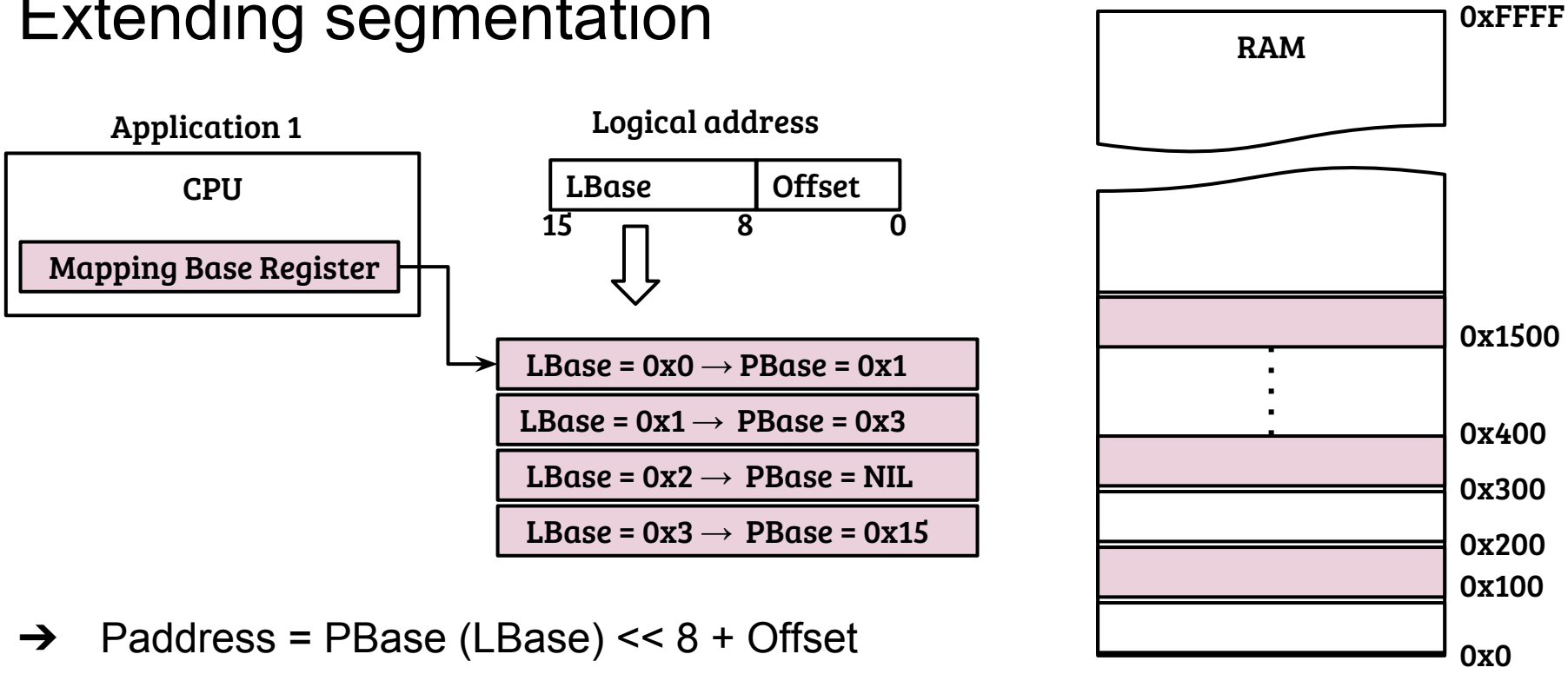


- ➔ In 64-bit, segmentation is minimally used
  - ◆ Flat segmentation model
  - ◆ No limit checks
  - ◆ Used to implement privileges, entry gates (for interrupt, exception etc.)
- ➔ Segment descriptor table
  - ◆ Accessible from ring-0
  - ◆ Global descriptor table and local descriptor table
    - Load (LGDT) and Store (SGDT)

A possible solution to limitations of segmentation

- ➔ Assume that
  - ◆ Segment size (limit) is fixed (e.g., 1 KB), segment base address is a multiple of segment limit
  - ◆ Offset within a segment remains same in logical and physical address
- ➔ Instead of segment registers, logical base address is used to map to physical base address of a segment
  - ◆ For example, with segment size 100 bytes, logical address 1005, requires mapping of logical base 10 to some physical base (say 900)
  - ◆ Final physical address is calculated to be 90005
- ➔ There should be a mapping from logical base to physical base
- ➔ How to maintain and use the mapping?

Extending segmentation



➔ Paddress = PBase (LBase) << 8 + Offset

➔ Can logical address be greater than physical address?

Paging

➔ Every process is provided an illusion of a huge memory, called the virtual memory

Paging

- ➔ Every process is provided an illusion of a huge memory, called the virtual memory
- ➔ Virtual address width determines the size of virtual address
  - ◆ For example, 32-bit wide virtual address → 4GB virtual address, 0x0 to 0xFFFFFFFF

Paging

- ➔ Every process is provided an illusion of a huge memory, called the virtual memory
- ➔ Virtual address width determines the size of virtual address
  - ◆ For example, 32-bit wide virtual address → 4GB virtual address, 0x0 to 0xFFFFFFFF
- ➔ Virtual address is partitioned into small chunks (mostly 4KB), called pages
- ➔ Compiler can place the program structures (code, data etc.) at any address in the virtual address range (ideally).

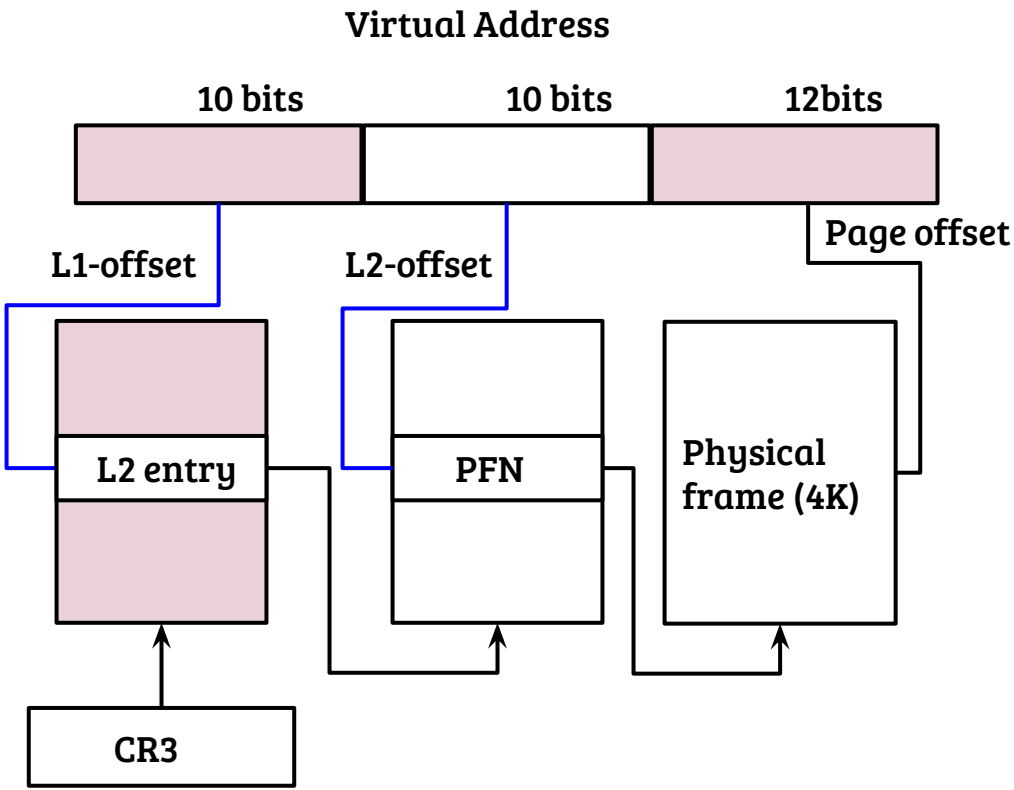
Paging

- ➔ Every process is provided an illusion of a huge memory, called the virtual memory
- ➔ Virtual address width determines the size of virtual address
  - ◆ For example, 32-bit wide virtual address → 4GB virtual address, 0x0 to 0xFFFFFFFF
- ➔ Virtual address is partitioned into small chunks (mostly 4KB), called pages
- ➔ Compiler can place the program structures (code, data etc.) at any address in the virtual address range (ideally).
- ➔ Physical address is partitioned into chunks of same size, called page frames

Paging

- ➔ Every process is provided an illusion of a huge memory, called the virtual memory
- ➔ Virtual address width determines the size of virtual address
  - ◆ For example, 32-bit wide virtual address → 4GB virtual address, 0x0 to 0xFFFFFFFF
- ➔ Virtual address is partitioned into small chunks (mostly 4KB), called pages
- ➔ Compiler can place the program structures (code, data etc.) at any address in the virtual address range (ideally).
- ➔ Physical address is partitioned into chunks of same size, called page frames
- ➔ When applications use virtual address to access memory,
  - ◆ Virtual address is translated to physical address
  - ◆ Translation is per-process and performed by the hardware
  - ◆ Translation may require multiple intermediate levels of translation (will see shortly)
  - ◆ A CPU register (CR3 in X86) points to a memory location containing the first level of translation

Paging example (32-bit virtual address)



➔ Example:

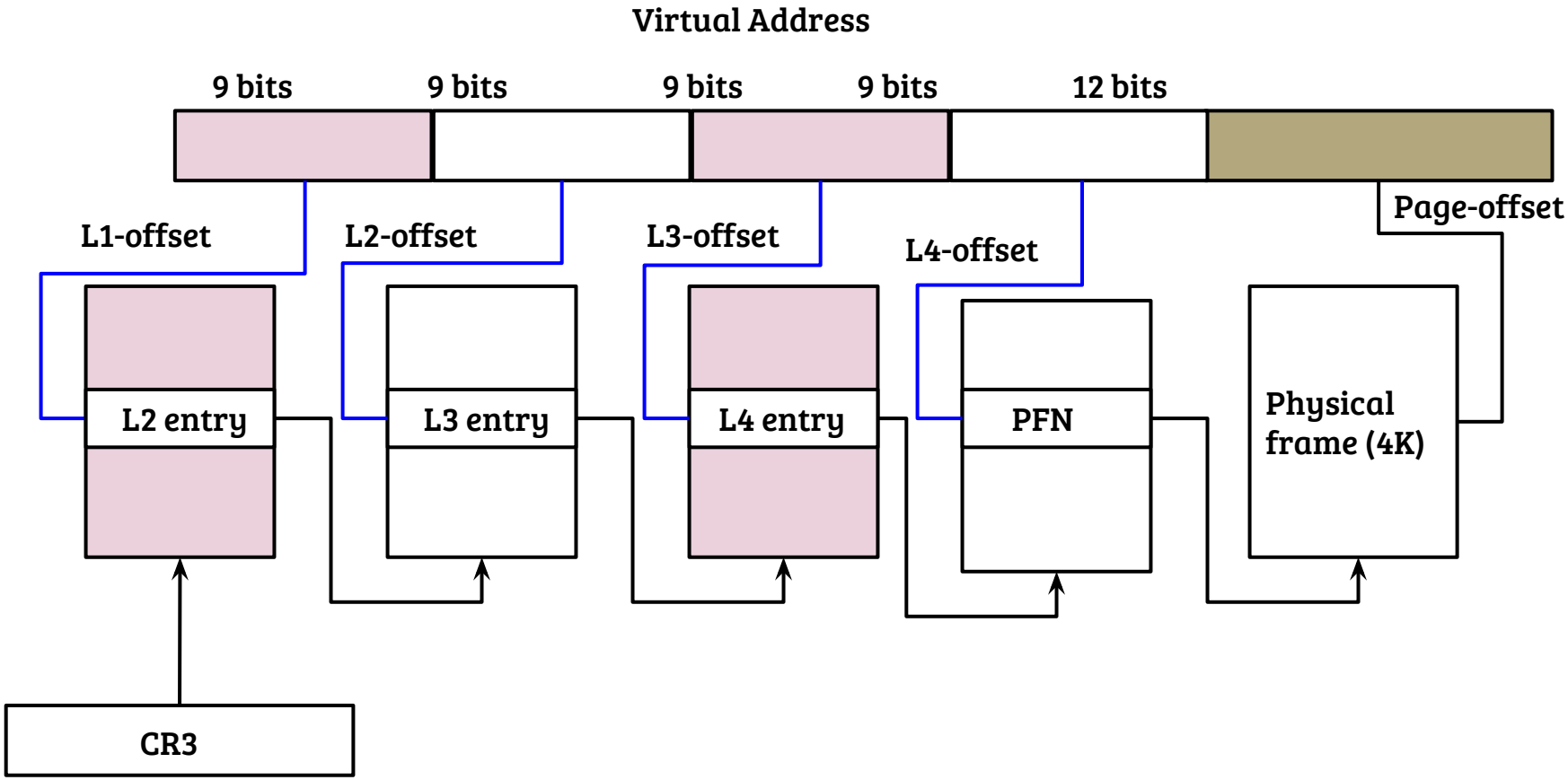
- Page size = 4KB
- Entry size = 4 Bytes (access flags + next level address)
- How virtual address 0x3F50A075 will be translated?
- What is the max. physical address supported if 16 bits in page table entry are reserved for access flags?
- #of memory accesses to perform translation?

Paging: design parameters

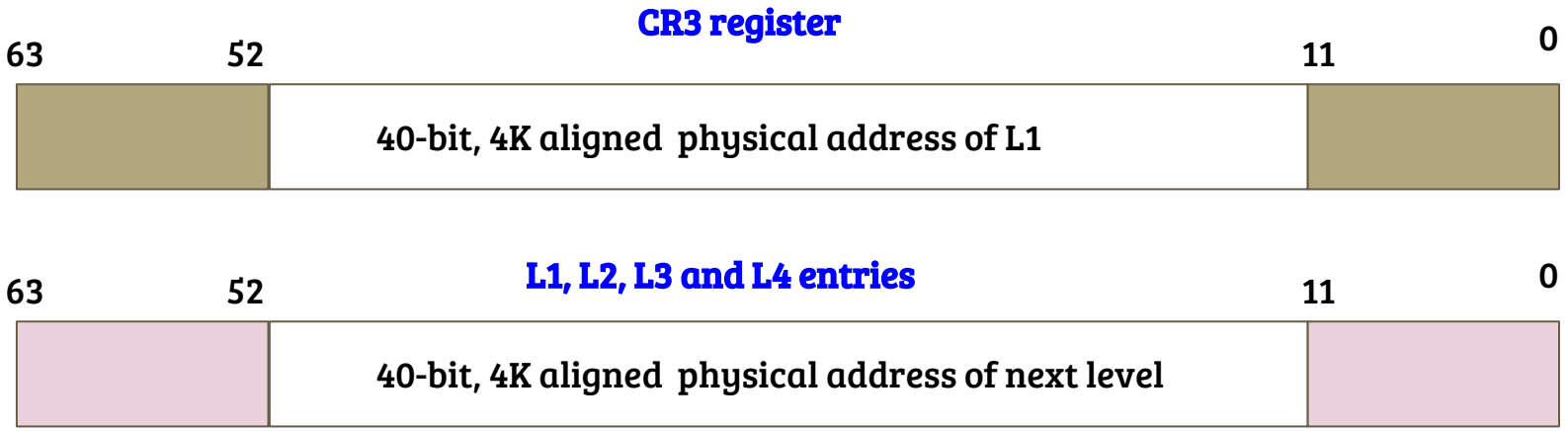
- Homework
  - 1. With page table layout explained in last slide, calculate the worst case memory consumption for maintaining page tables for 32 applications.
  - 2. Answer Q1 for a page size of 64 KB

- Virtual address width vs. page table levels
- #of levels vs. memory required for page tables
- Radix tree vs. hashed page tables
- Other ideas?

X86_64: 4-level page tables (48-bit virtual address)



X86_64 page table entries



Some important flags

0 (present/absent) 1 (read/write) {2} (user/kernel), 5(accessed) 63 (execute permission)

*Source: Intel manual Vol: 3A 4.5