Main Memory

ICS332
Operating Systems

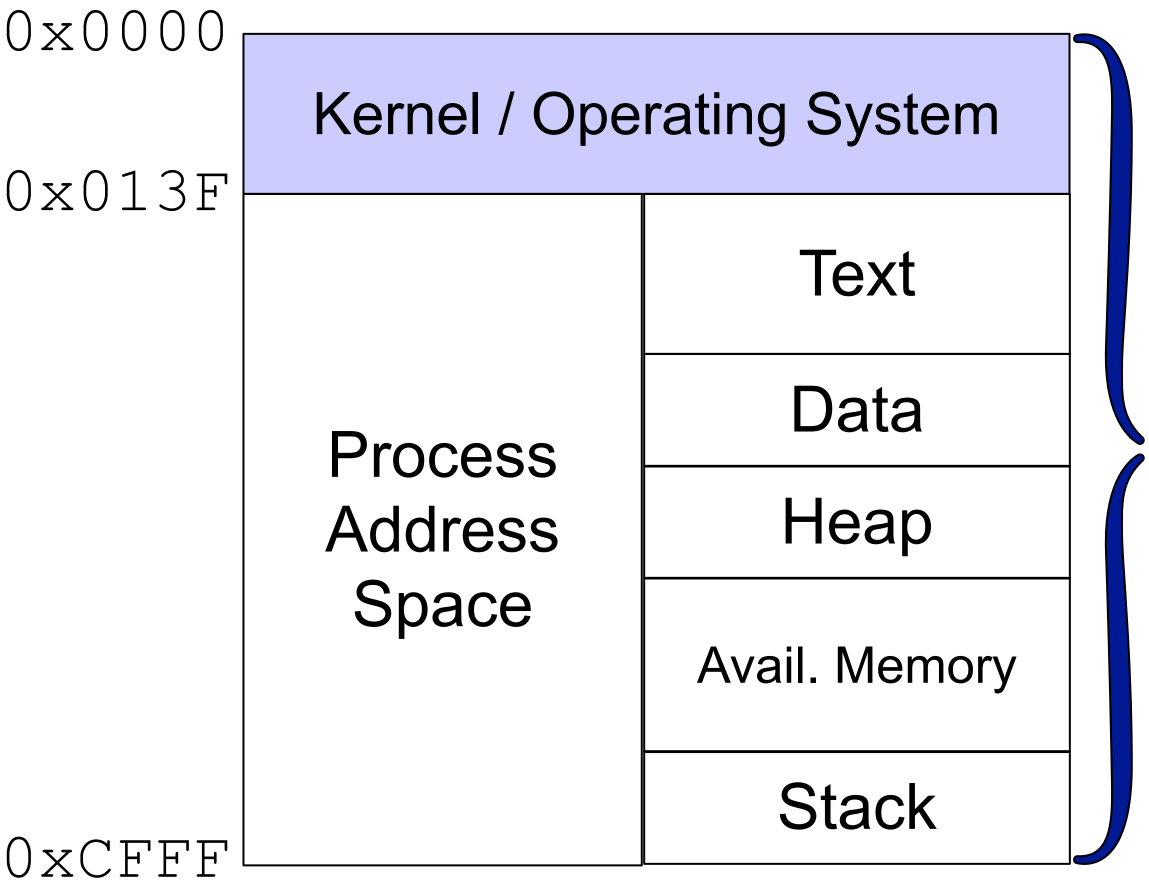
Henri Casanova (henric@hawaii.edu)

Main Memory: Basics

- Main Memory = Memory Unit (in Von Neumann model)
  - (Large) contiguous array of bytes/words, each with its own address
  - Stream of addresses coming in on the memory bus
  - Each incoming address is stored in the memory-address register of the memory unit
  - Which causes the memory unit to put the content at that address on the memory bus
  - And that content is then read in by the CPU
- Called the “Main” memory by contrast with registers, caches, which are all managed 100% by the hardware
- Processes share the main memory, therefore the OS must manage the main memory
- The CPU only works with registers, but it can issue addresses of locations (bytes) in main memory
  - Via load/store instructions (in MIPS assembly: LOAD and STORE; in x86 assembly: mov .., [..] and mov [..], ..)

Contiguous Memory Allocation

- Let us assume what we have always assumed so far: each process is allocated a contiguous zone of physical memory



In this example memory is fully utilized by the kernel and one process:

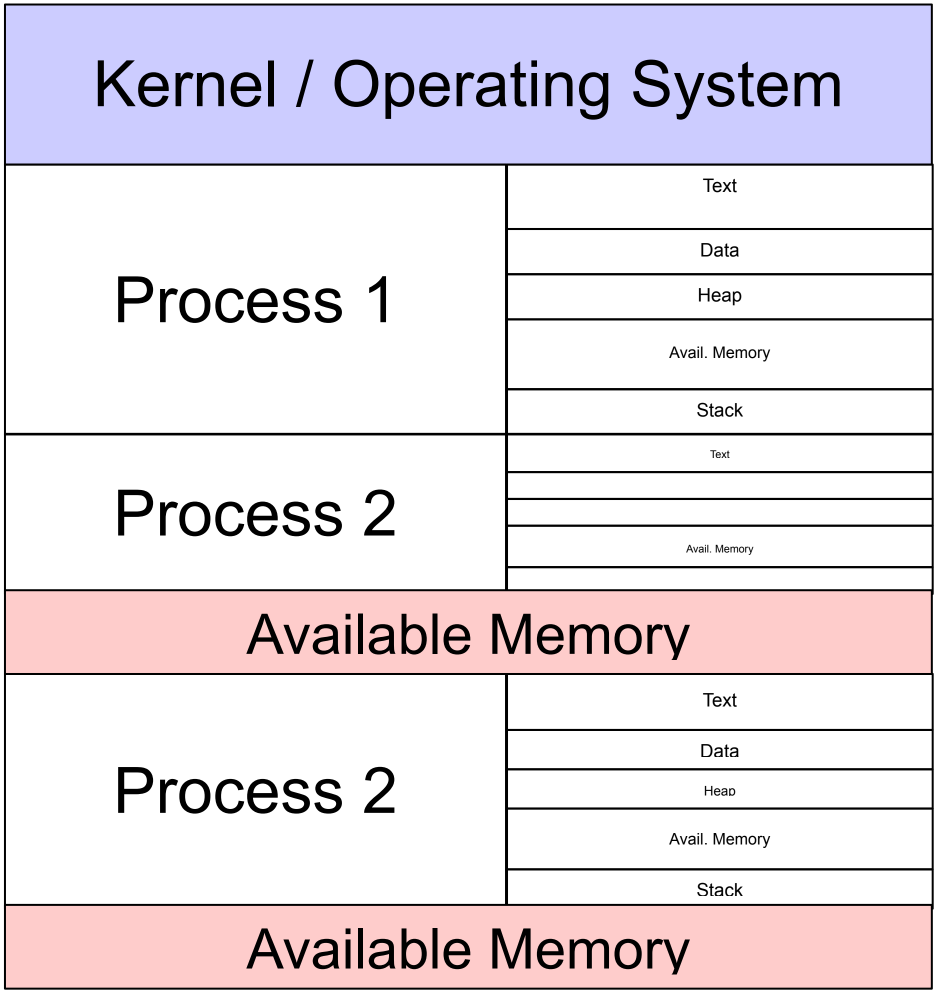
Total RAM size

= 0xCFFF + 1

= 53248d

= 52 KiB

Contiguous Memory Allocation + Multiprogramming



This is the typical picture shown with multiple processes in memory, each with its own contiguous address space, and perhaps some left over available memory

Address Binding

- One important question is that of address binding: when are physical addresses determined for bytes of data/instruction?
- In your high-level code you write something like:

C source code

```
int a; // global
. . .
if (a != 0) {
  a++;
}
. . .
```

- When is the address of where the value of `a` is located determined??
- When is the address of the instruction to which to jump in case the value of `a` is zero determined??
- Let’s look at the compiled version of this program…

Address Binding

- One important question is that of address binding: when are physical addresses determined for bytes of data/instruction?
- In your high-level code you write something like:

C source code

```
int a; // global
. . .
if (a != 0) {
  a++;
}
. . .
```

Compiled Assembly Code

```
    . . .
    cmp [label_a], 0
    jz Nope
    inc word [label_a]
Nope:
    . . .
```

- When is the address of where the value of **`a`** is located determined??
- When is the address of the instruction to which to jump in case the value of **`a`** is zero determined??
- The assembler transforms the assembly code into a binary executable
- Let’s look at the compiled version of this program…

Address Binding - Absolute Addressing?

C source code

```
int a; // global
. . .
if (a != 0) {
   a++;
}
. . .
```

Compiled Assembly Code

```
    . . .
    cmp [label_a], 0
    jz Nope
    inc word [label_a]
Nope:
    . . .
```

- One approach is to use absolute addressing so that the binary executable contains physical addresses:

```
    Address   Text
        ...   ...
 0x5623FAB2   AFFB 0x6677FFBB     // cmp [label_a], 0
 0x5623FAB4   DC32 0x5623FAB8     // jz Nope
 0x5623FAB6   E013 0x6677FFBB     // a++
 0x5623FAB8   ...
        ...   ...
 0x6677FFBB   0                   // "int a" is here
        ...   ...
```

Problems of Absolute Addressing

- Absolute addressing is simple, but is has not been used in decades
- Anybody sees what a problem is with it?

Problems of Absolute Addressing

- Absolute addressing is simple, but is has not been used in decades
- Anybody sees what a problem is with it?
- **With absolute addressing a program must be loaded exactly at the same place into memory each time we run it**
  - Otherwise the addresses will be wrong!

- Therefore we may not be able to run a program because another program is running and encroaches on the address range!
- **Corollary**: We cannot run multiple instances of a single program!
- One solution would be to recompile a program each time you need to run it
  - Because only when you're about to run a program can you know where it should fit in memory
  - But this has problems: while you're recompiling it somebody else starts another program…

- **Bottom-line:** absolute addressing is not a good idea and hasn't been used for a loooong time on general-purpose computers

Address Binding - Relative Addressing?

- We can solve the problem of absolute addressing with a very simple idea called relative addressing
- Assume the address space starts at some BASE address, and compute all addresses as an offset from the BASE:

```
   Address   Text
0x56230000   F43D 0x5623000               // set BASE = 0x56230000
       ...   ...
0x5623FAB2   AFFB BASE + 1054FFBB         // cmp [label_a], 0
0x5623FAB4   DC32 BASE + FAB8             // jz Nope
0x5623FAB6   E013 BASE + 1054FFBB         // a++
0x5623FAB8   ...
       ...   ...
0x6677FFBB   0                            // "int a" is here
       ...   ...
```

- The code is now completely **relocatable**: Only the BASE needs to be determined before running it
- The same program can be run anywhere in memory (at whatever BASE address)
- Multiple instances can run, each with a different BASE address, provided they don't overlap

RAM Virtualization

- All addresses in the process address space are expressed as an offset relative to the base value
- A program can be anywhere in RAM and doesn’t care where:
  - Instead of saying “the 4th byte in my address space is at address x”, it says “the 4th byte in my address space is at address BASE + 4”
- And just like that we have memory virtualization!
- OSTEP shows C programs that highlight this
  - OSTEP 2.2
- Let’s do another simple example here…

Memory-Virtualization Uncovered

Memory-Allocating Program

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
int main(int argc, char **argv) {
  if (argc != 2) {
    fprintf(stderr,"Usage: %s <int value>\n", argv[0]); exit(1);
  }
  int value;
  if (sscanf(argv[1], "%d", &value) != 1) {
    fprintf(stderr,"Invalid command-line argument\n");
    exit(1);
  }
  int *address = (int*)malloc(sizeof(int));
  *address = value;
  printf("I wrote value %d at address %p\n", value, address);
  sleep(10);
  printf("At address %p I see value %d\n", address, *address);
  exit(0);
}
```

Compile and run on Linux

```
gcc -o memory_virtualization memory_virtualization.c -fsanitize=address
```

Let’s run two instances in two terminals

Take Away

- Both programs print the same address, therefore **it cannot be a physical address!**
- Instead, the address issued by programs and handled by the CPU are logical addresses or virtual addresses (both terms are used)

- What was that `-fsanitize=address` thing???
  - This command-line option to gcc enables the use of AddressSanitizer
  - AddressSanitizer is an open-source tool developed by Google
    - It is supported by most compilers on Linux and MacOS
  - It detects memory errors for C/C++
    - buffer overflow, stack overflow, use after free, stack overflow
  - It turns out that to do its work AddressSanitizer disables address space layout randomization (ASLR)
  - Let’s remove that option and see what happens....

Memory Virtualization

- Thanks to virtualization: Each program instance has the illusion that it's alone in RAM and that its address space starts at address 0
- This gives us Memory Protection
  - A program doesn't need to know anything about other programs
  - It never has to think "ooh... I shouldn't write there in RAM because that address is used by another program"
  - This is good, because when you write the code you don't know what other programs will be running anyway!

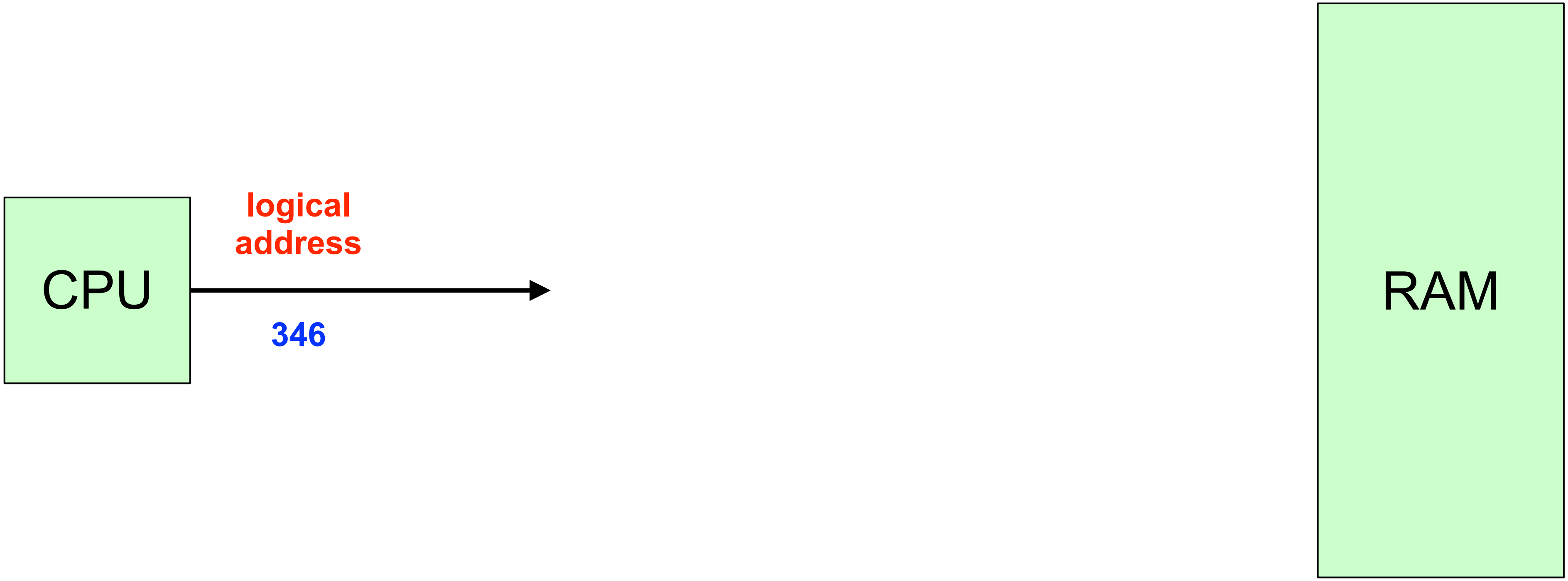
- **Bottom Line:** A program references a logical address space, which corresponds to a physical address space in the memory

- "Something" needs to tell the CPU how to translate from virtual to physical addresses, i.e., some address translation mechanism

Virtualizing Address Spaces

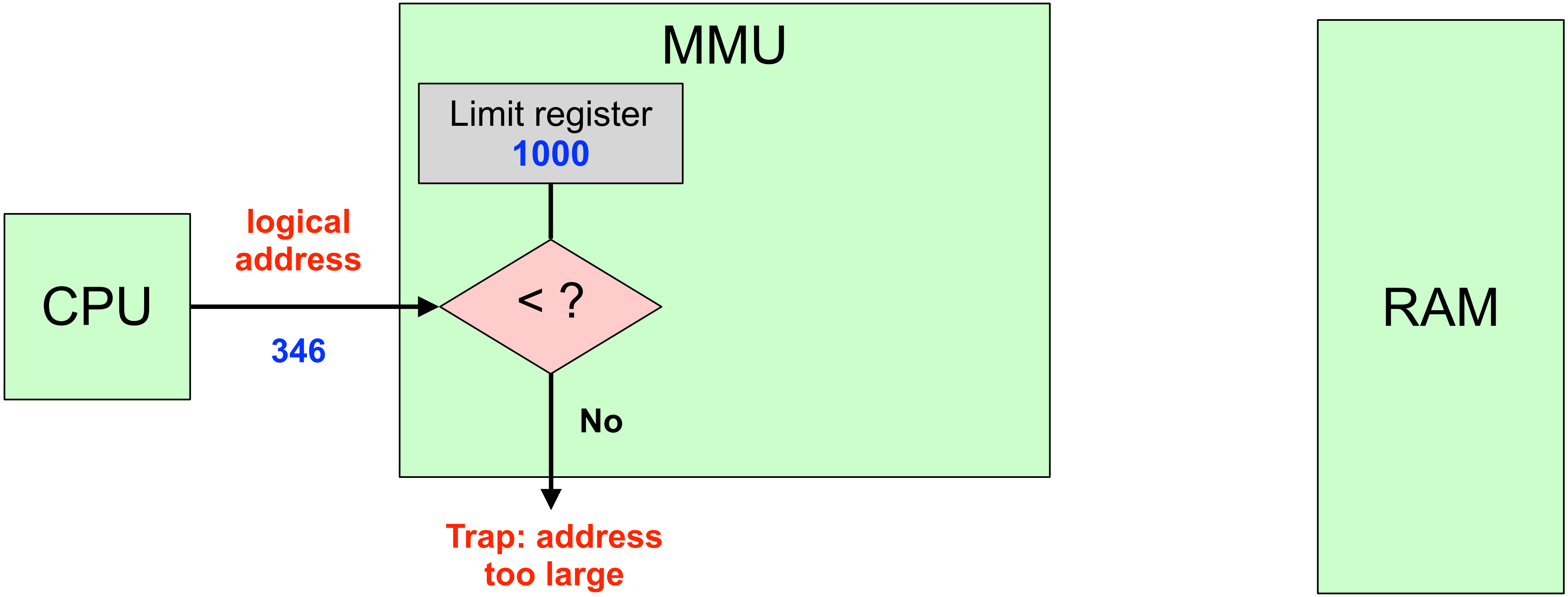
- Some component needs to translate virtual addresses into physical addresses: add an offset to the BASE
- **Address translation happens very frequently** (each load, store, jump)
- Therefore: The BASE Address is accessed very frequently
  - The memory translation component should store it in a register or something as fast as a register
- And: Offsets are added to the BASE address very frequently
  - Wasting even one CPU cycle to do the addition would be very expensive
- Furthermore: It would be nice if only valid logical addresses were translated
  - For memory protection: we don't want processes to step on each other's toes
- So we use a base register and a limit register that stores the base address and the largest possible logical address
- **And we implement the above as a super fast hardware component: the Memory Management Unit (MMU)**

Memory Management Unit



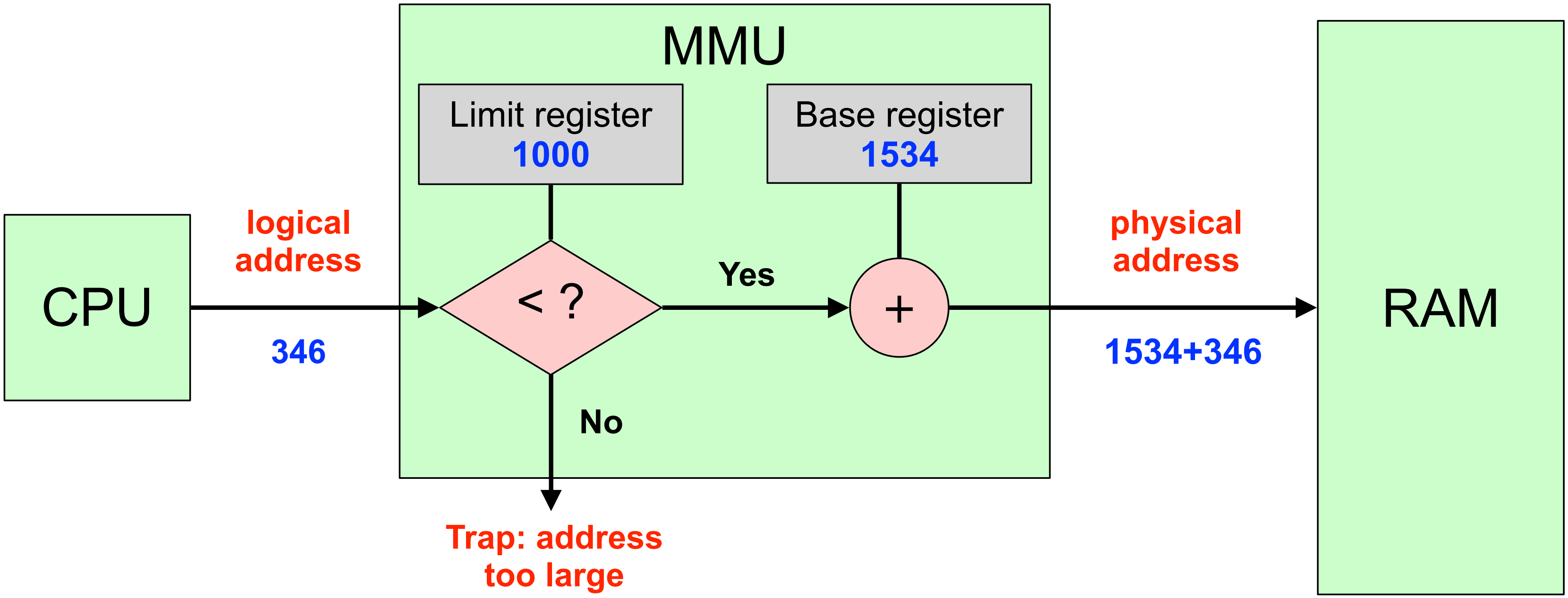
- Historically: A specialized circuit between the CPU and the memory
- Nowadays: Integrated with the CPU

Memory Management Unit



- Historically: A specialized circuit between the CPU and the memory
- Nowadays: Integrated with the CPU

Memory Management Unit



- Historically: A specialized circuit between the CPU and the memory
- Nowadays: Integrated with the CPU

Summary So Far

- Your program generates only logical addresses between 0 and some upper bound (the limit)
- Each such address is checked to see if it's beyond the limit
- If not, then the address is translated (just add it to the base)
- That translated physical address is then sent to the memory bus

- **Bottom line:**
  - Your CPU only "sees" logical addresses
  - Your RAM only "sees" physical addresses

Segmentation

- Recall the structure of the address space
  - The figure doesn't show the "data" part
- An address space is full of empty space
  - In which the heap/stack will grow
- Therefore having a single contiguous "segment" is wasteful

- Segmentation: Avoid waste by breaking up the address space into pieces
  - Each piece has its own base/limit register

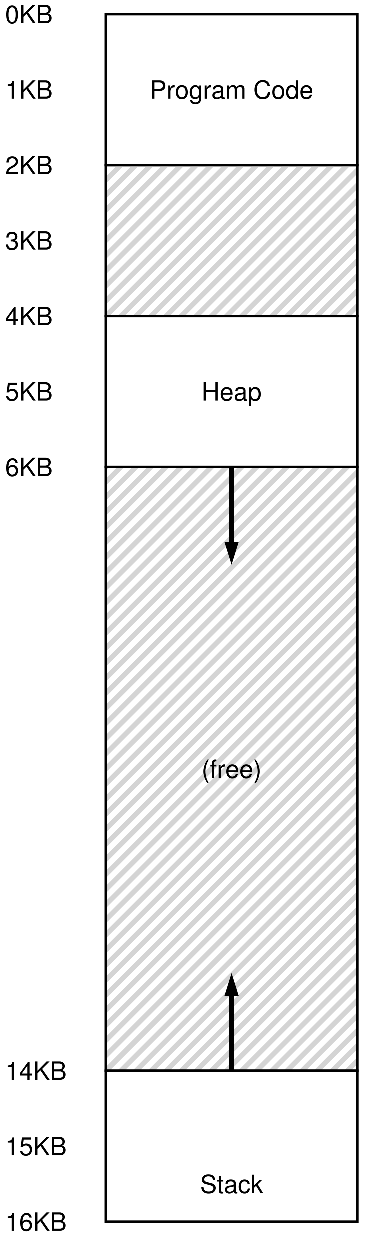
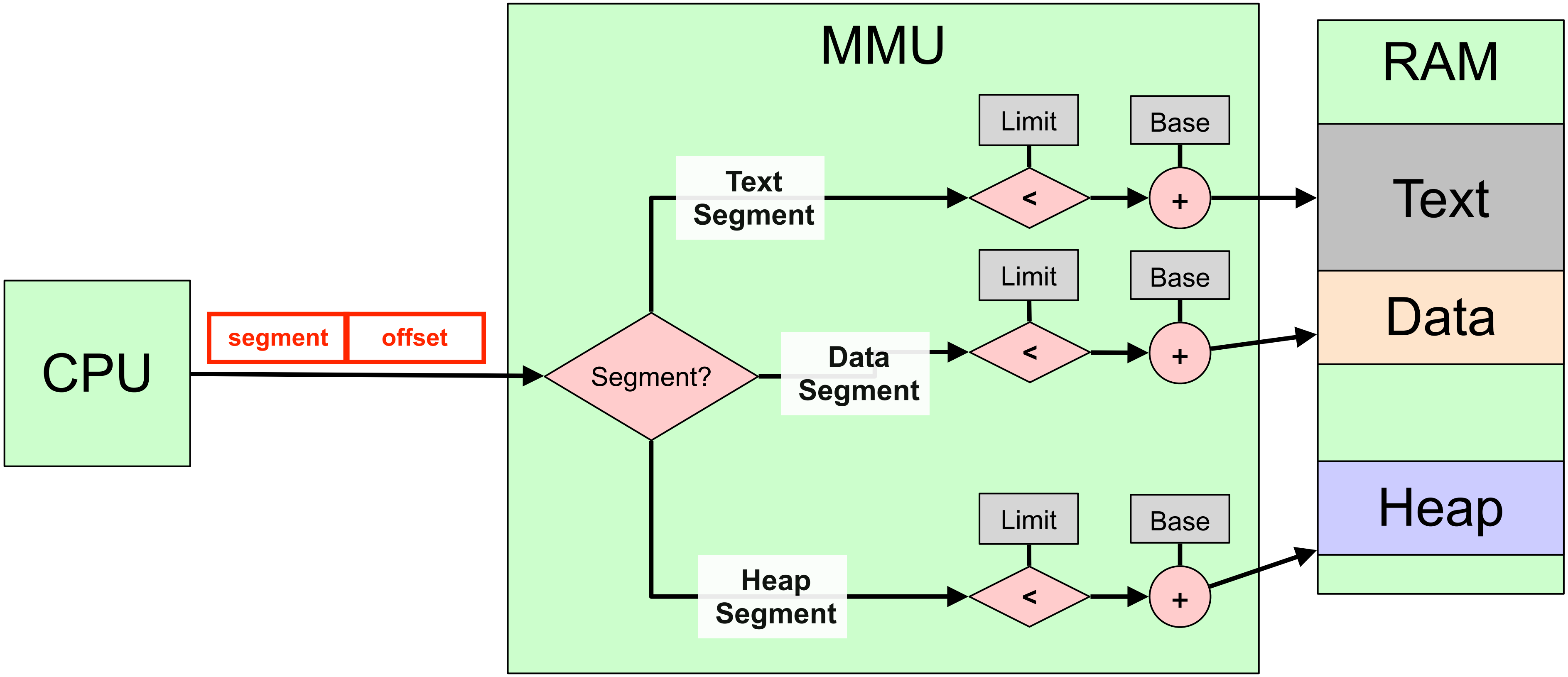


Figure 16.1: **An Address Space (Again)**

Segmentation

- The logical address space is now a collection of segments
- The compiler/language interpreter handles the segments and the logical addresses are built appropriately
  - If you write in assembly language, you may have to deal with segments manually
- Typical segments used by a C compiler
  - text
  - data
  - heap
  - stacks
  - standard C library
- The first bits of the logical address are used to identify which segment is being referenced
- Let’s see this on a picture

MMU Segmentation



- Implementing segmentation is easy
- Reserve bits (e.g., the left-most ones) in the logical address to reference a segment (the segment bits)
- Question: how do we know which segment is being referenced?

Segment Table

- A segment table with one entry per segment number is used to keep track of segments
- For each segment, its entry stores:
  - Base: Starting address of the segment
  - Limit: Length of the segment
- The segment table is stored in memory
  - (but cached on the CPU to avoid extra memory accesses… a common theme we'll come back to)
- A **Segment-Table Base Register** (STBR): Points to the segment table address
- A **Segment-Table Length Register** (STLR): Gives the length of the segment table
  - Makes it easy to detect an invalid segment offset
- These registers are saved/restored at each context switch

Segmentation for Protection

- Now that we have each “piece” of the address space in its own segment we can easily implement some protection mechanisms!
- The segment table can include bits that answer:
  - Is the segment readable?
  - Is the segment writable?
  - Is the segment executable?
  - Any combination of 3 bits: RWX
    - RX: Read and execute (e.g. text)
    - RW: Read and write (e.g. stack)
- This allows the CPU to detect errors/bugs such as “executing data as if it were code”, “overwriting code”, ...

Conclusion

- We now have a basic understanding of how memory addresses can be virtualized
- Main concept: the CPU sees logical addresses, and the MMU transforms them into physical addresses
  - Determines the segment
  - Look up the segment table to find the segment's base and limit values
  - Check that the logical address is within the limit, and if not generate a trap
  - Add the base to the logical address
  - And voila, we have the physical address

- Next up: what happens if our program does not fit in RAM?