

KTRW

The journey to build a debuggable iPhone

Brandon Azad

OBTS v3.0





```
noname — astris — 76x33
imac-noname:~ noname$ astris
astris v2.6.5 [Astris-827.50.7~274 (Electric tools)]

Probe address: KongSWD-
Probe type: kong
Probe firmware: 0.52
Probe tckrate: 1950000

Listening on port 8000 for CPU0, CPU1
Listening on port 8002 for IOP
Listening on port 8003 for AE2
Listening on port 8004 for ISP
Detected HSP C0
KongSWD- - CPU0:Run CPU1:PowerOff IOP:Reset AE2:Reset ISP:PowerOff
NO CPU > cpu cpu0
KongSWD- - CPU0:Run CPU1:PowerOff IOP:Reset AE2:Reset ISP:PowerOff

CPU0 > halt
CPU0: ASTRIS_ERR_OK
r0: 0x00000001 r1: 0xbfffc080 r2: 0x00000004 r3: 0x00000000
r4: 0x00000001 r5: 0xbfffc080 r6: 0xbfffc084 r7: 0xbff60700
r8: 0xbfffc080 r9: 0x00000000 r10: 0x00000001 r11: 0x00000400
ip: 0x3f200020 sp: 0xbff606f4 lr: 0xbff0e709 pc: 0xbff0e768
cpsr: 0x80000073 N-----FT Supervisor
0xbff0e768: 0xbf28 it cs
KongSWD- - CPU0:Halt CPU1:PowerOff IOP:Reset AE2:Reset ISP:PowerOff

CPU0 > save ~/Desktop/iBoot 0xbff00000 0x50000
.....
327680 bytes received in 9.527 sec, 34395 bytes per second
KongSWD- - CPU0:Halt CPU1:PowerOff IOP:Reset AE2:Reset ISP:PowerOff

CPU0 > https://pbs.twimg.com/media/DyP1Q\_qX0AA838S?format=png&name=large
```

I do not use dev-fused
devices

I do not use dev-fused
devices

But they sure would make
my security research easier...

Goal: Build my own "home-brewed dev phone"

- Patch kernel memory (`__TEXT_EXEC`)
- Breakpoints, watchpoints
- Use with LLDB / IDA Pro
- Can update iOS version
- Only parts you can get at an Apple store



axi0mX

@axi0mX

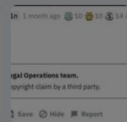
Bootrom exploit philanthropist. Advanced persistent troublemaker. Thought follower. Bringing iOS jailbreaking back from the dead. #checkm8 #alloc8 #kloader64

checkra.in

github.com/axi0mX

Joined October 2015

36 Photos and videos



axi0mX

@axi0mX



Follow

EPIC JAILBREAK: Introducing checkm8 (read "checkmate"), a permanent unpatchable bootrom exploit for hundreds of millions of iOS devices.

Most generations of iPhones and iPads are vulnerable: from iPhone 4S (A5 chip) to iPhone 8 and iPhone X (A11 chip).



axi0mX/ipwndfu

open-source jailbreaking tool for many iOS devices -
axi0mX/ipwndfu
github.com

4:15 AM - 27 Sep 2019

7,477 Retweets 16,436 Likes



970 7.5K 16K



axi0mX @axi0mX · Sep 27

1/ The last iOS device with a public bootrom exploit until today was iPhone 4, which was released in 2010. This is possibly the biggest news in iOS jailbreak community in years. I am releasing my exploit for free for the benefit of iOS

Follow



Want to take advantage of all the new Twitter features?

It's simple – just log in.

Log in

Sign up

You may also like · Refresh



checkra1n
@checkra1n



Pwn20wnd
@Pwn20wnd



Siguza

KTRR

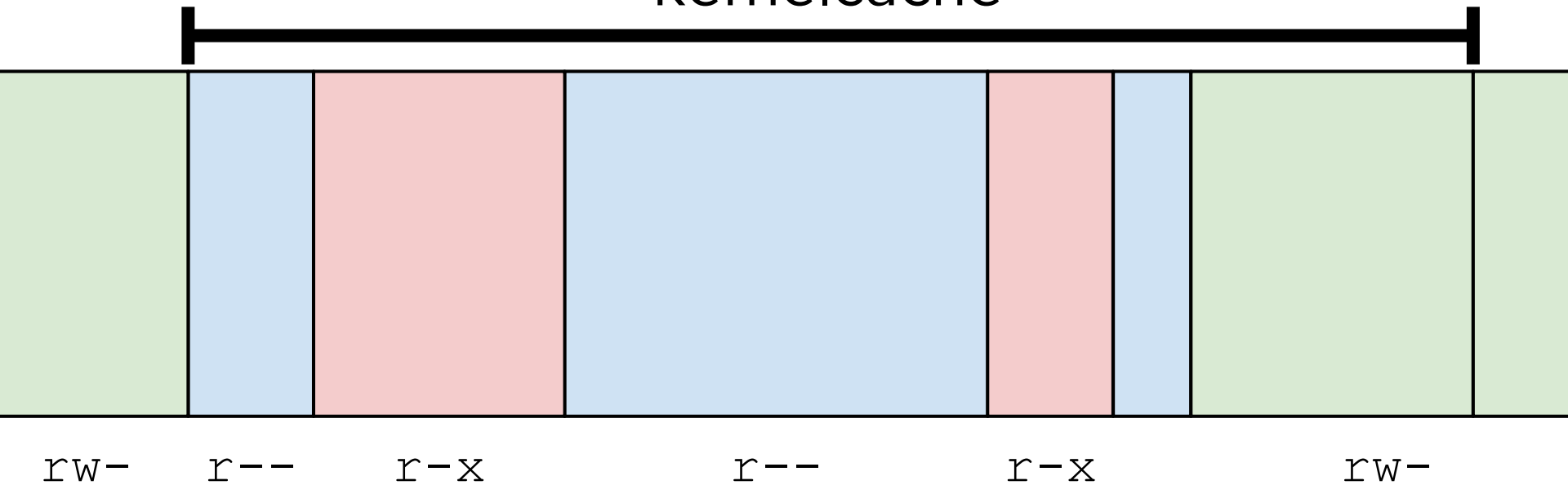
Goal: Build my own "home-brewed dev phone"

- **Patch kernel memory (`__TEXT_EXEC`)**
- Breakpoints, watchpoints
- Use with LLDB / IDA Pro
- Can update iOS version
- Only parts you can get at an Apple store

low addresses

high addresses

kernelcache

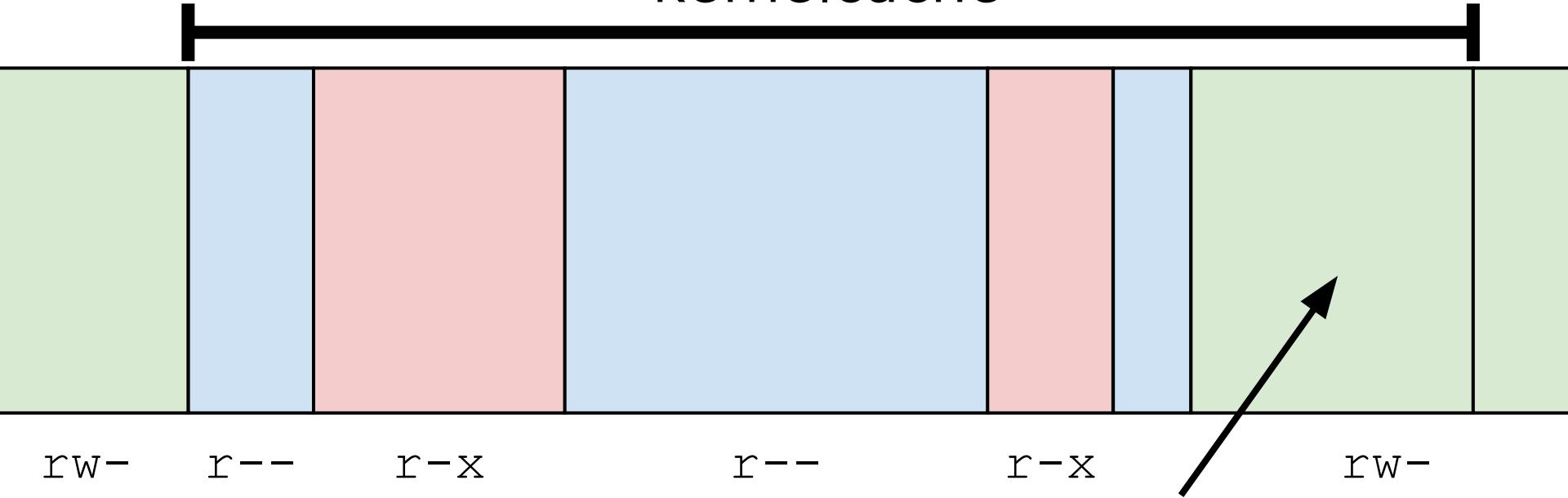


iPhone10,1 16G77 kernelcache

low addresses

high addresses

kernelcache



rw-

r--

r-x

r--

r-x

rw-

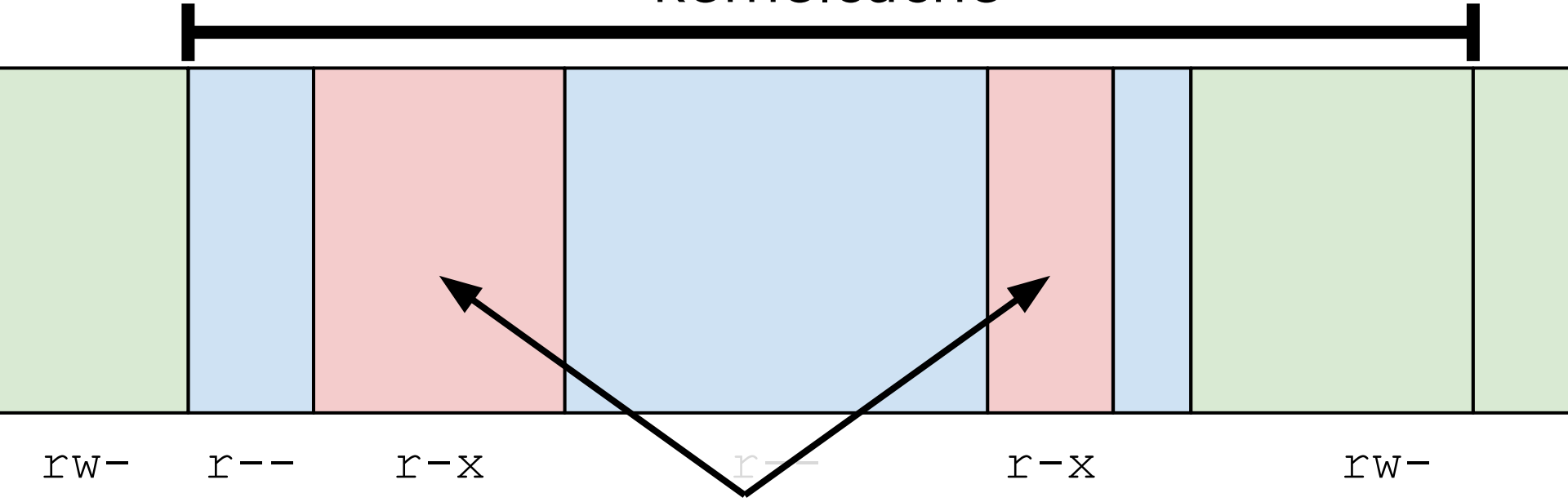
Read/write data

iPhone10,1 16G77 kernelcache

low addresses

high addresses

kernelcache



rw-

r--

r-x

r--

r-x

rw-

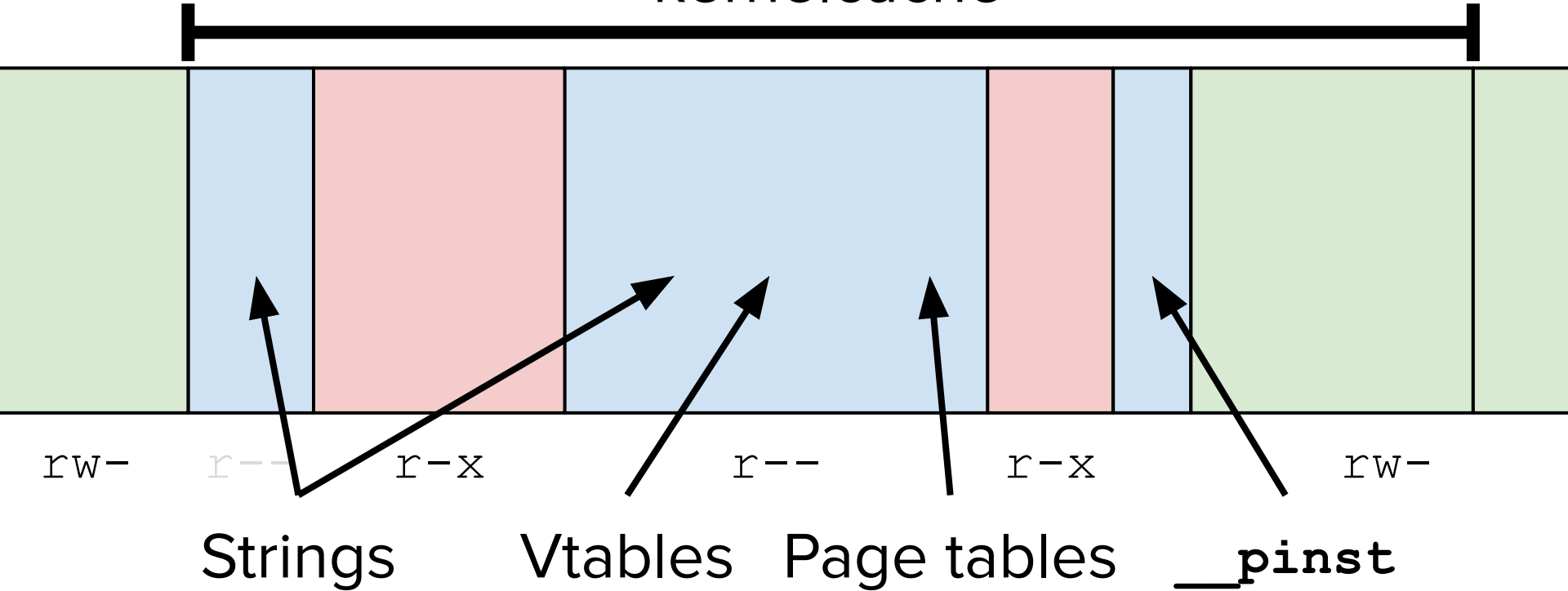
Executable code

iPhone10,1 16G77 kernelcache

low addresses

high addresses

kernelcache

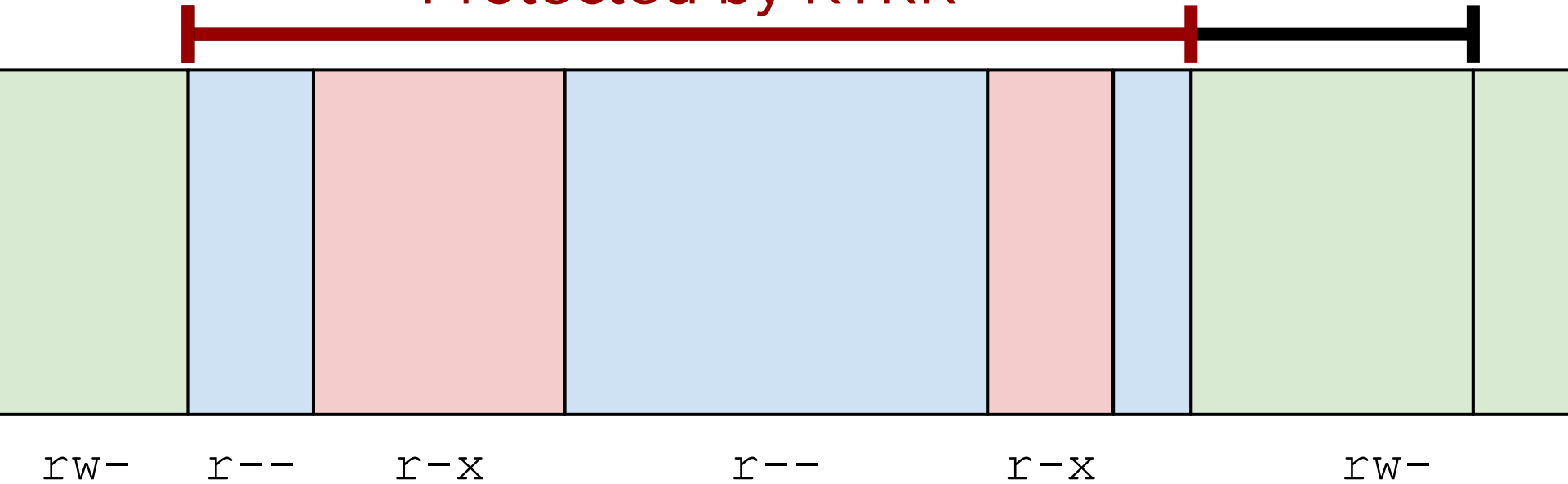


iPhone10,1 16G77 kernelcache

low addresses

high addresses

Protected by KTRR



iPhone10,1 16G77 kernelcache

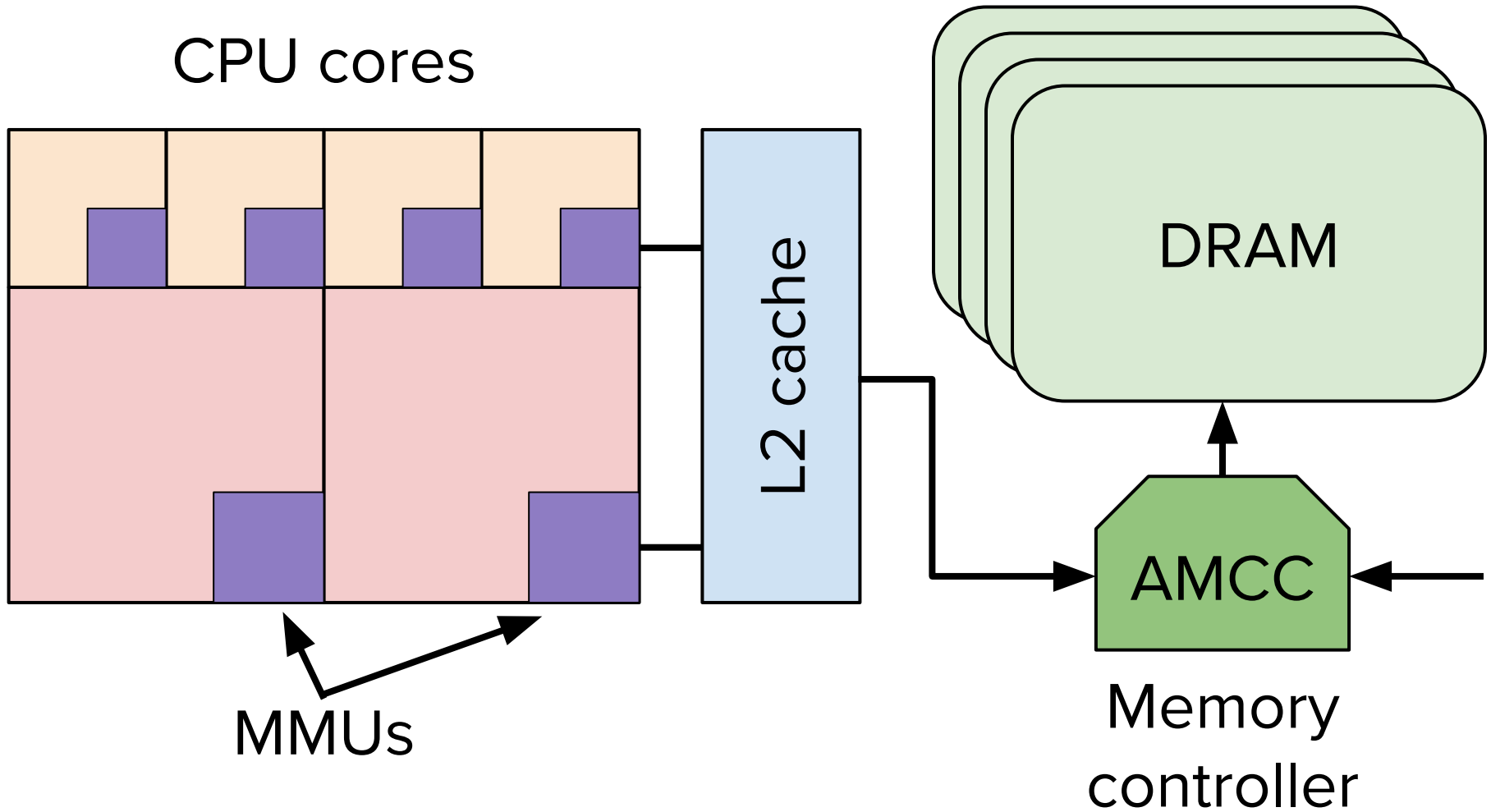
KTRR (Kernel Text Readonly Region)

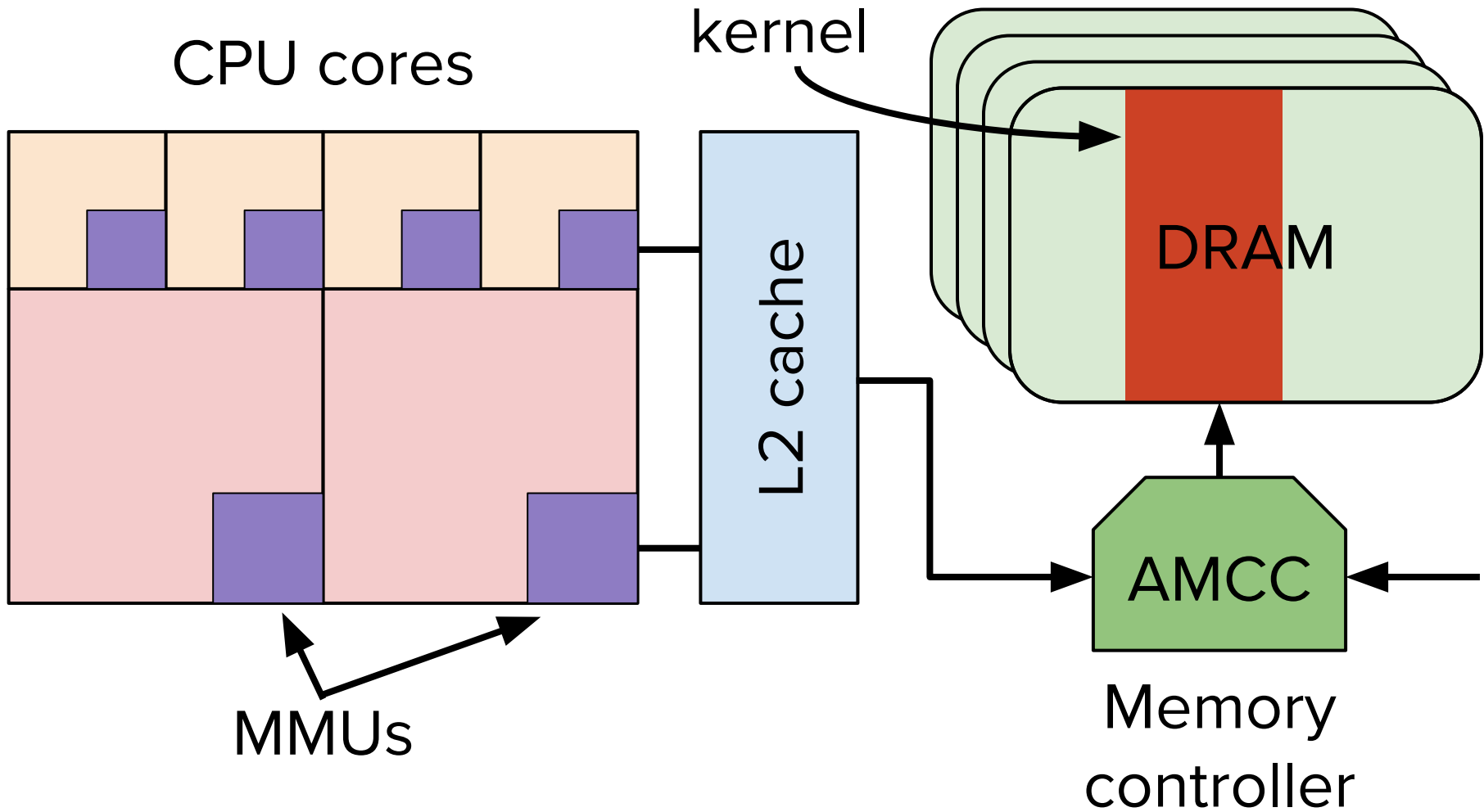
Strong form of W^X (write-xor-execute) protection

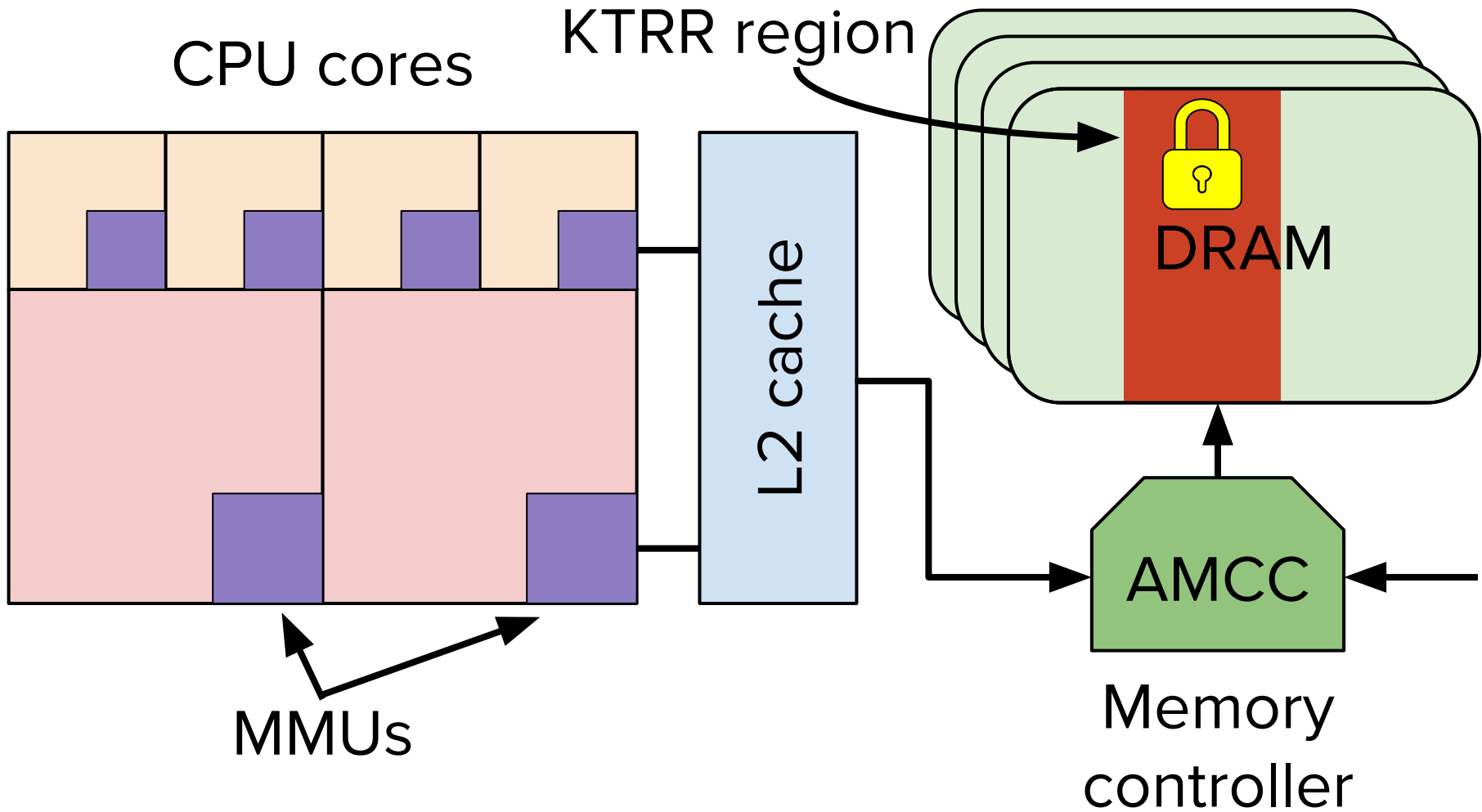
Apple A10 and later

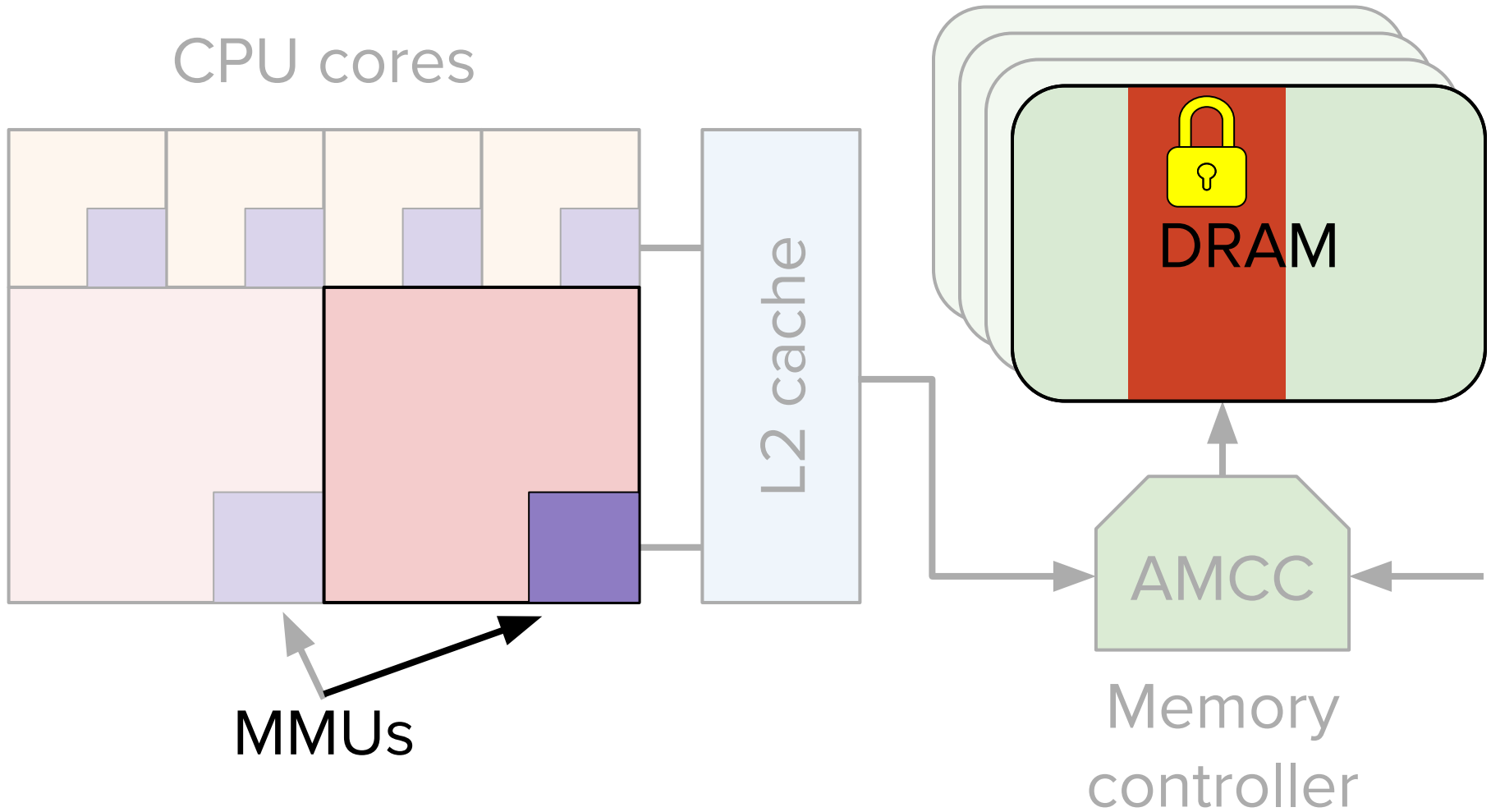
All writes to memory in the KTRR region fail

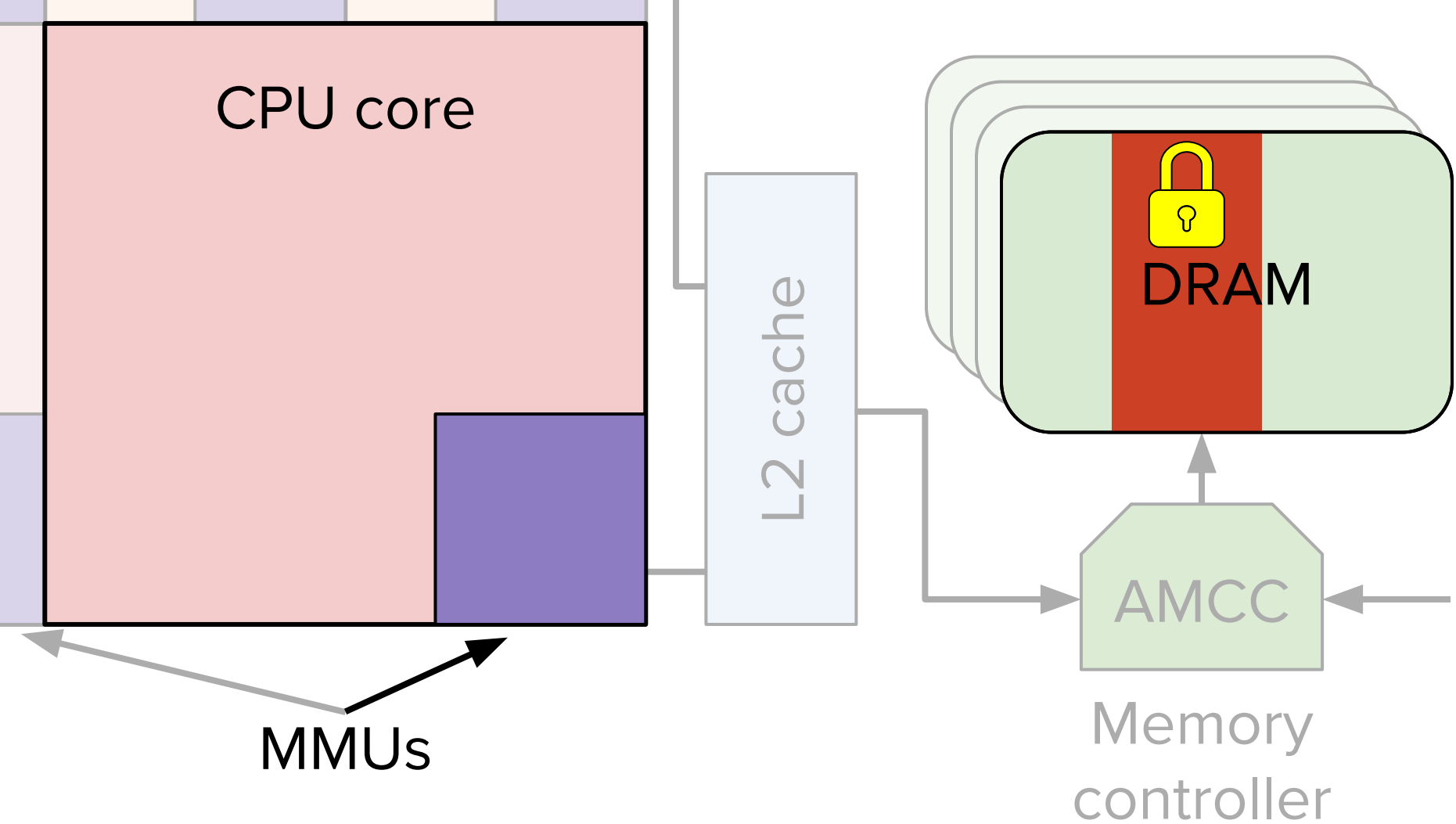
All instruction fetches from outside the KTRR region fail

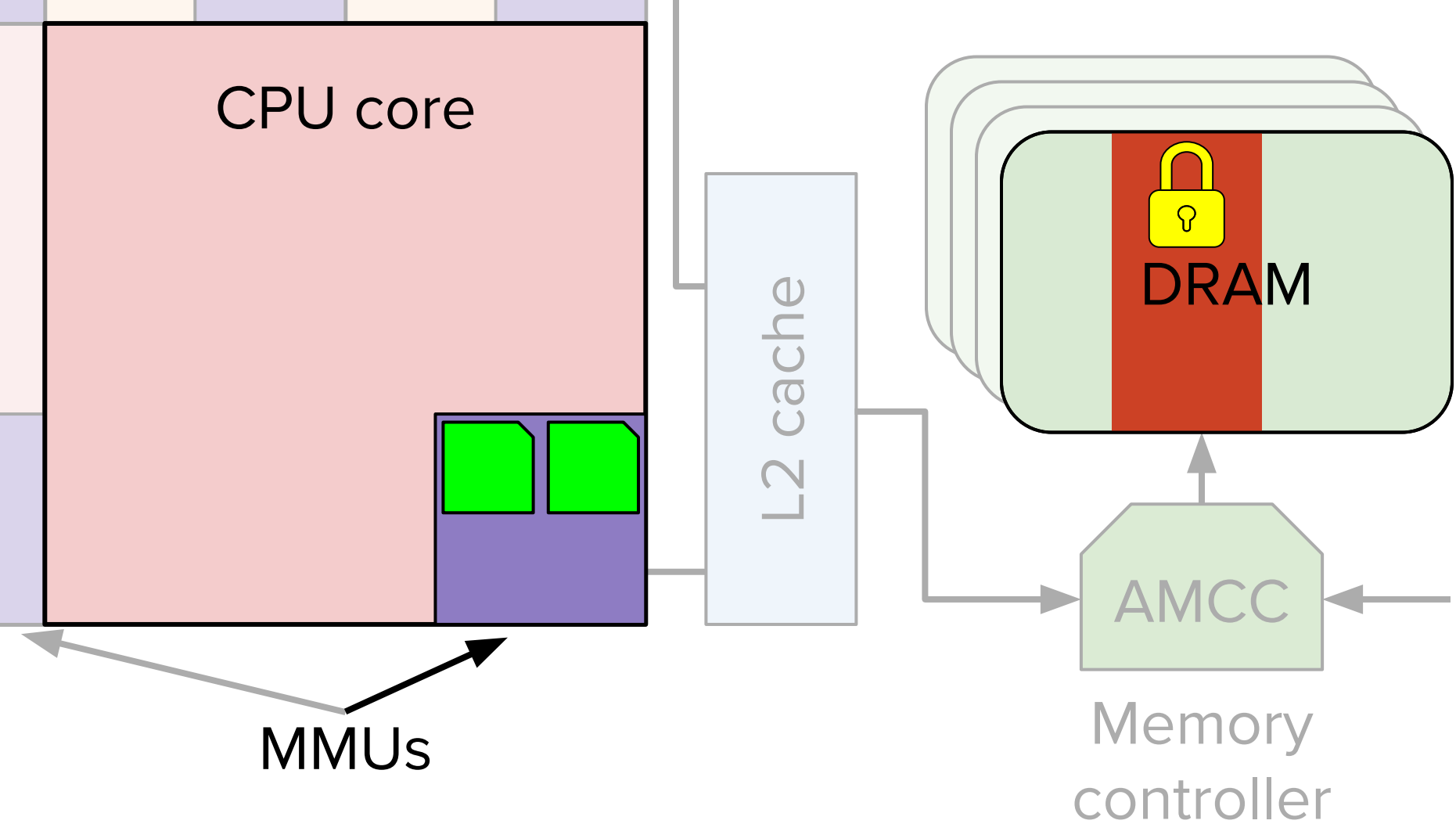


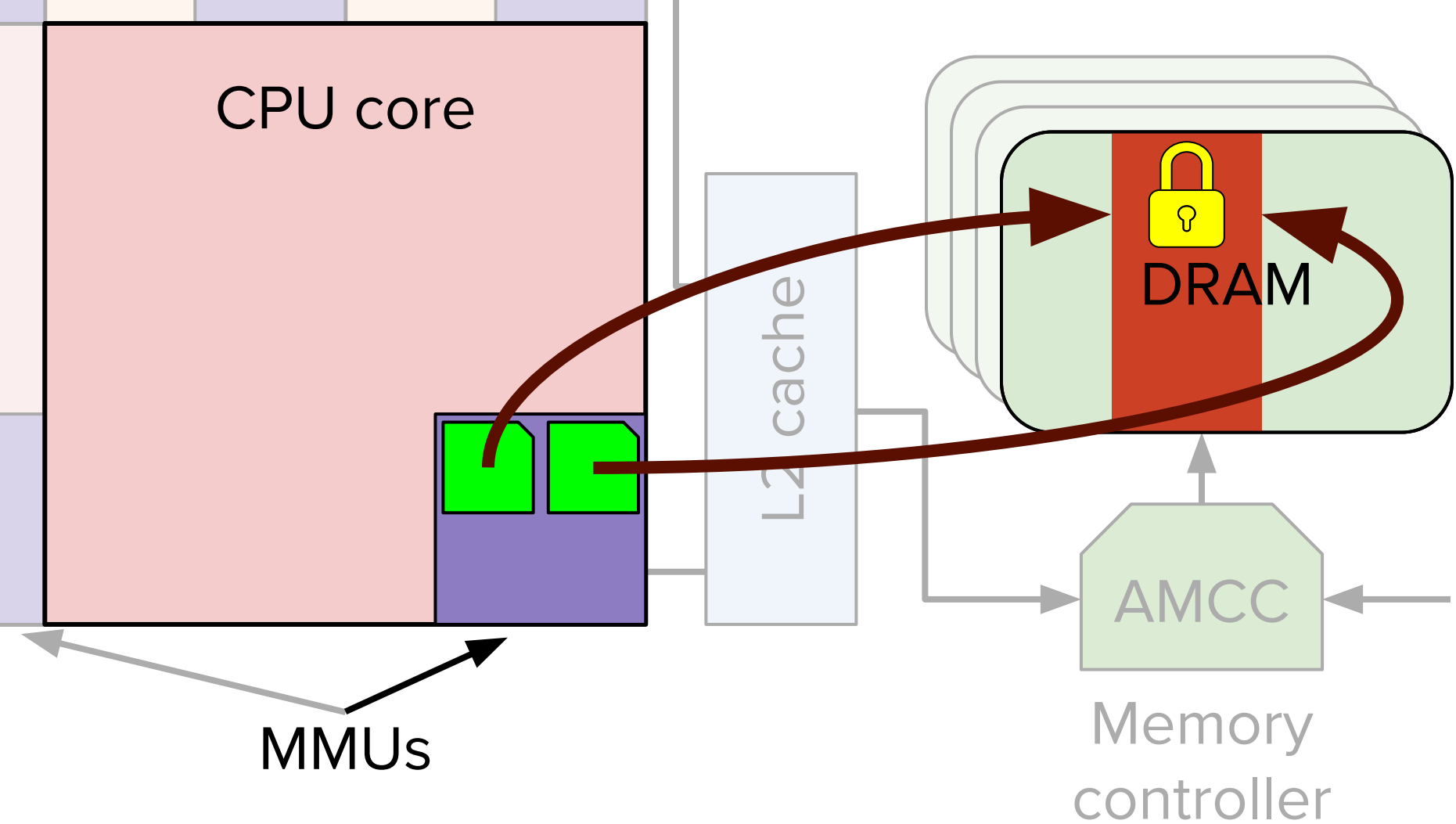


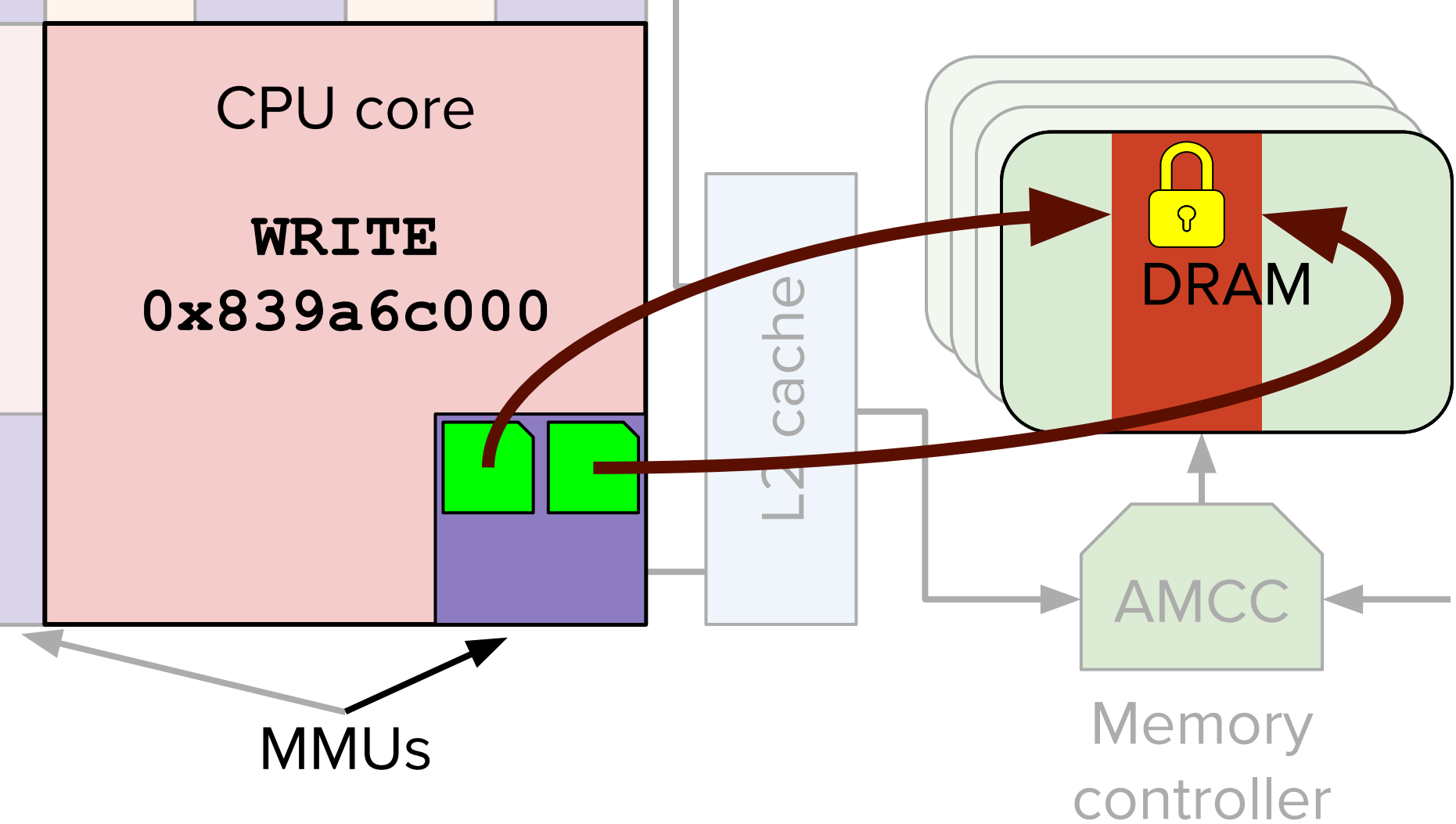


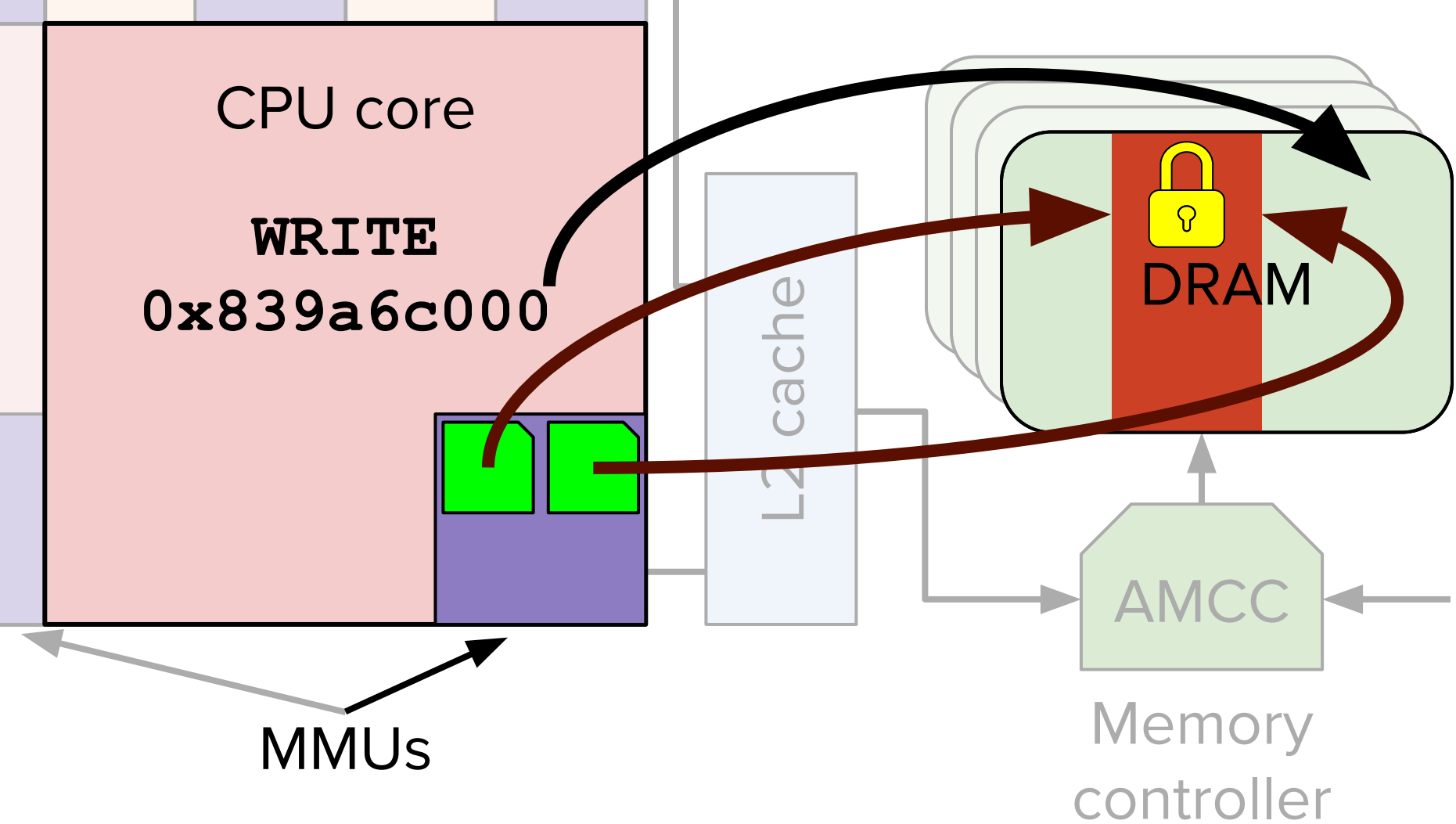


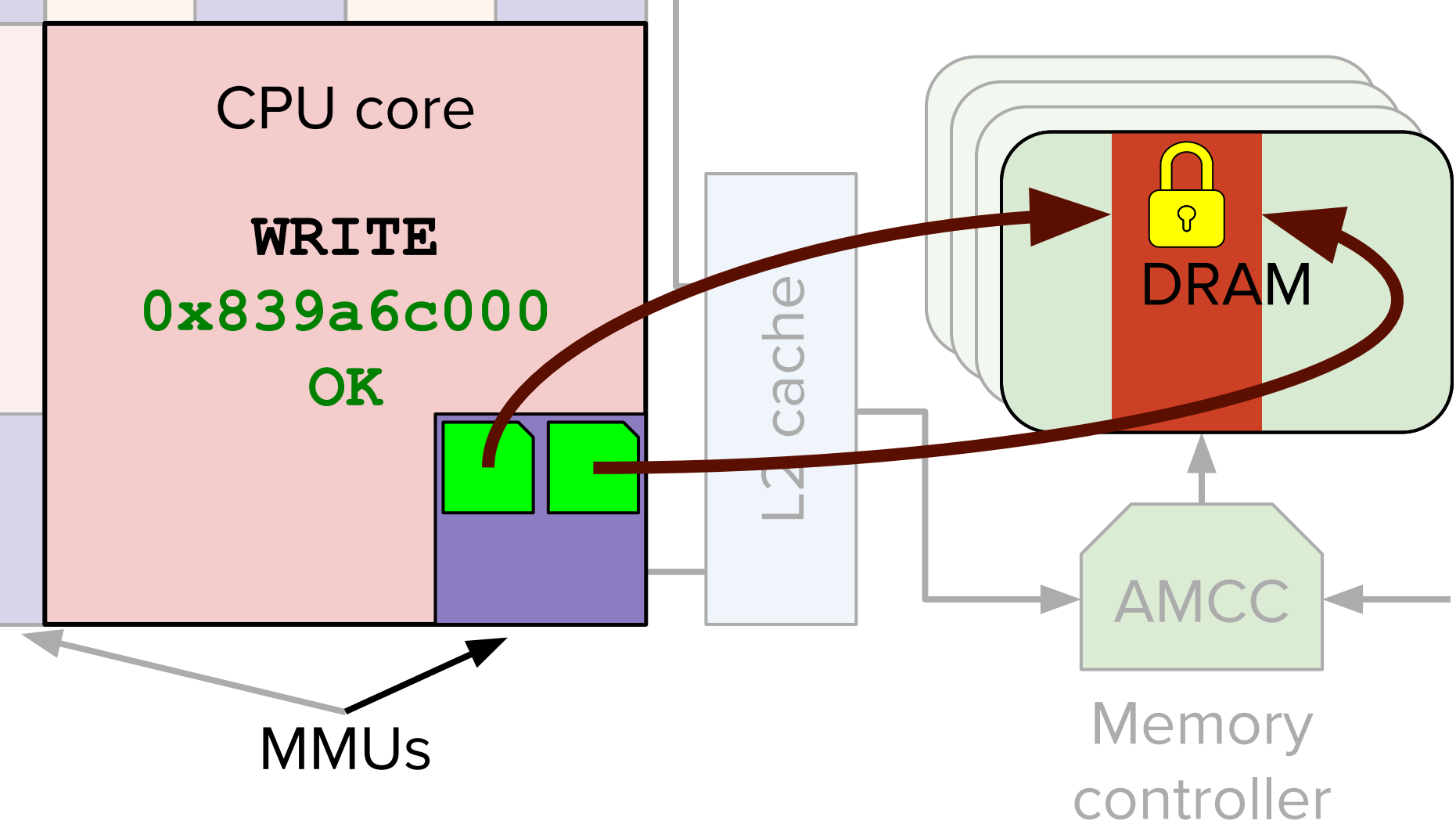


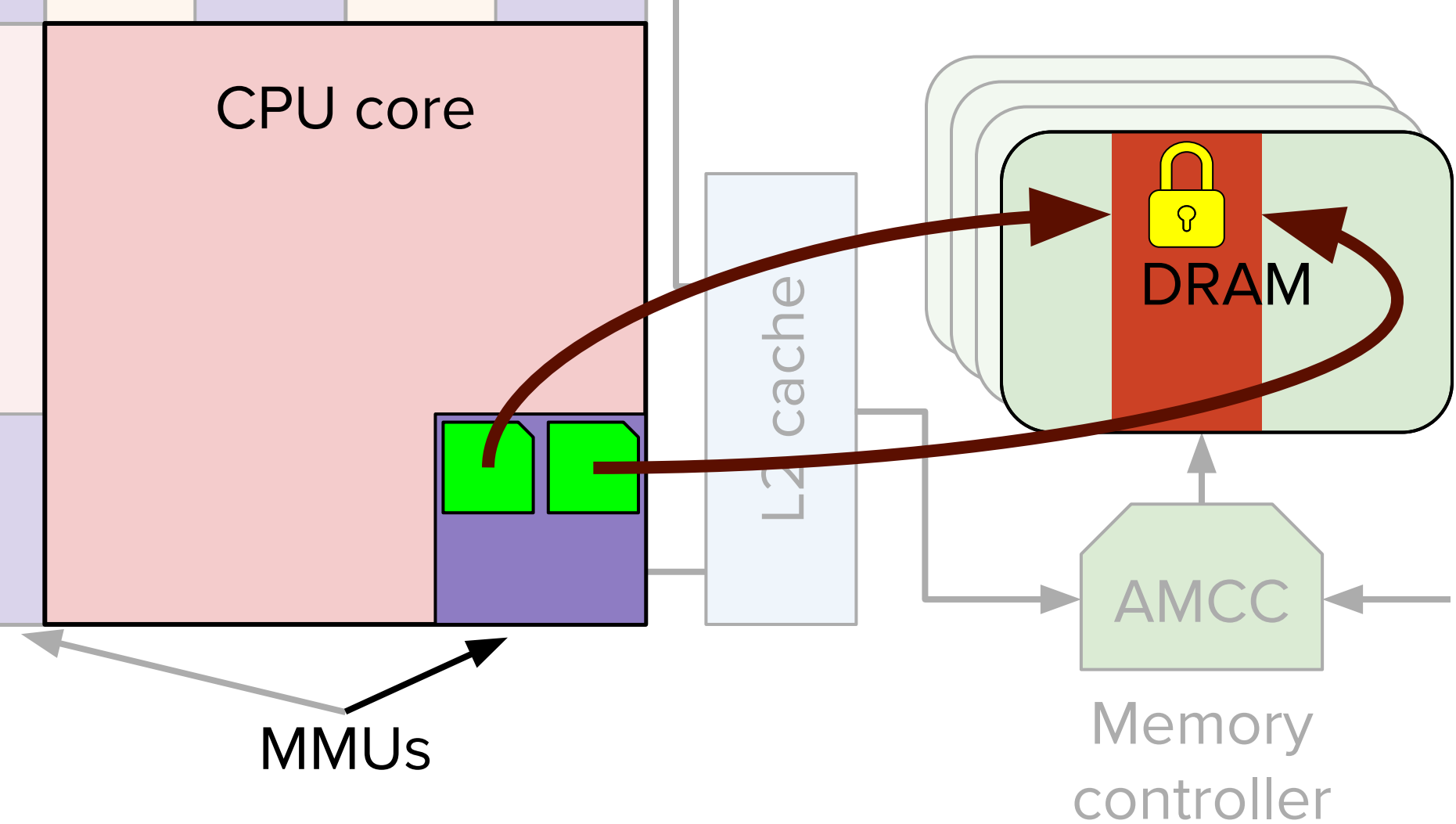


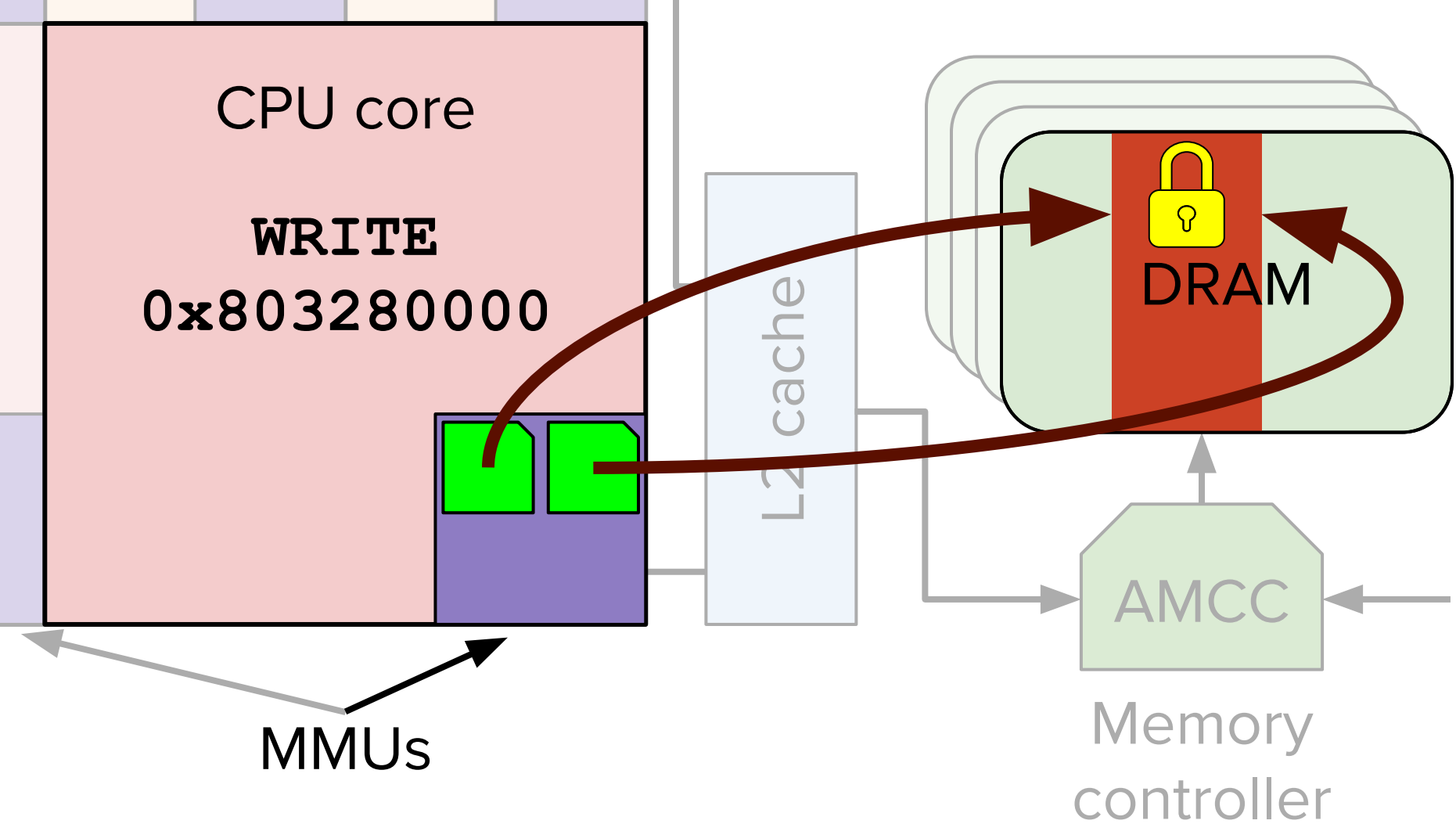


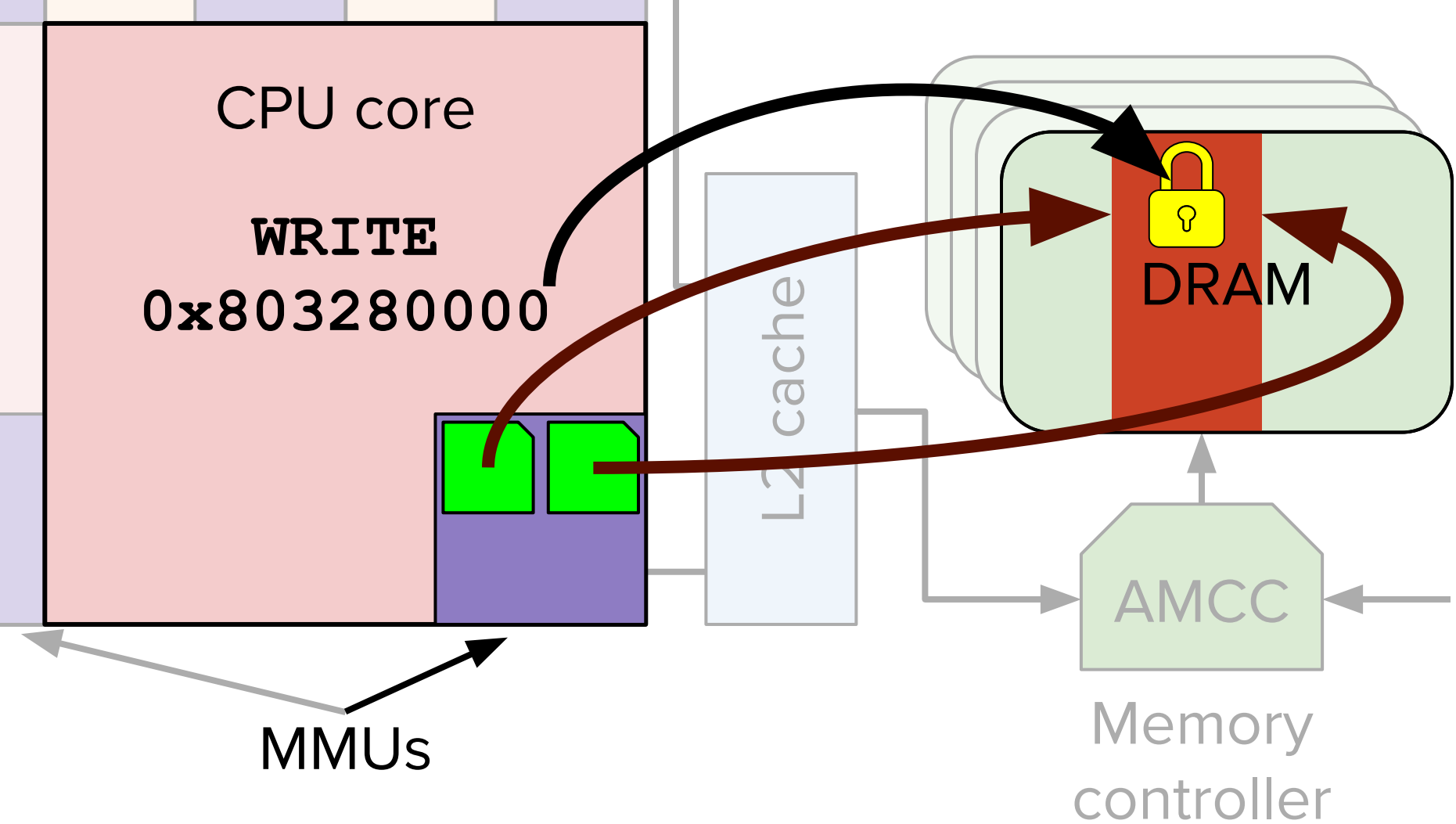


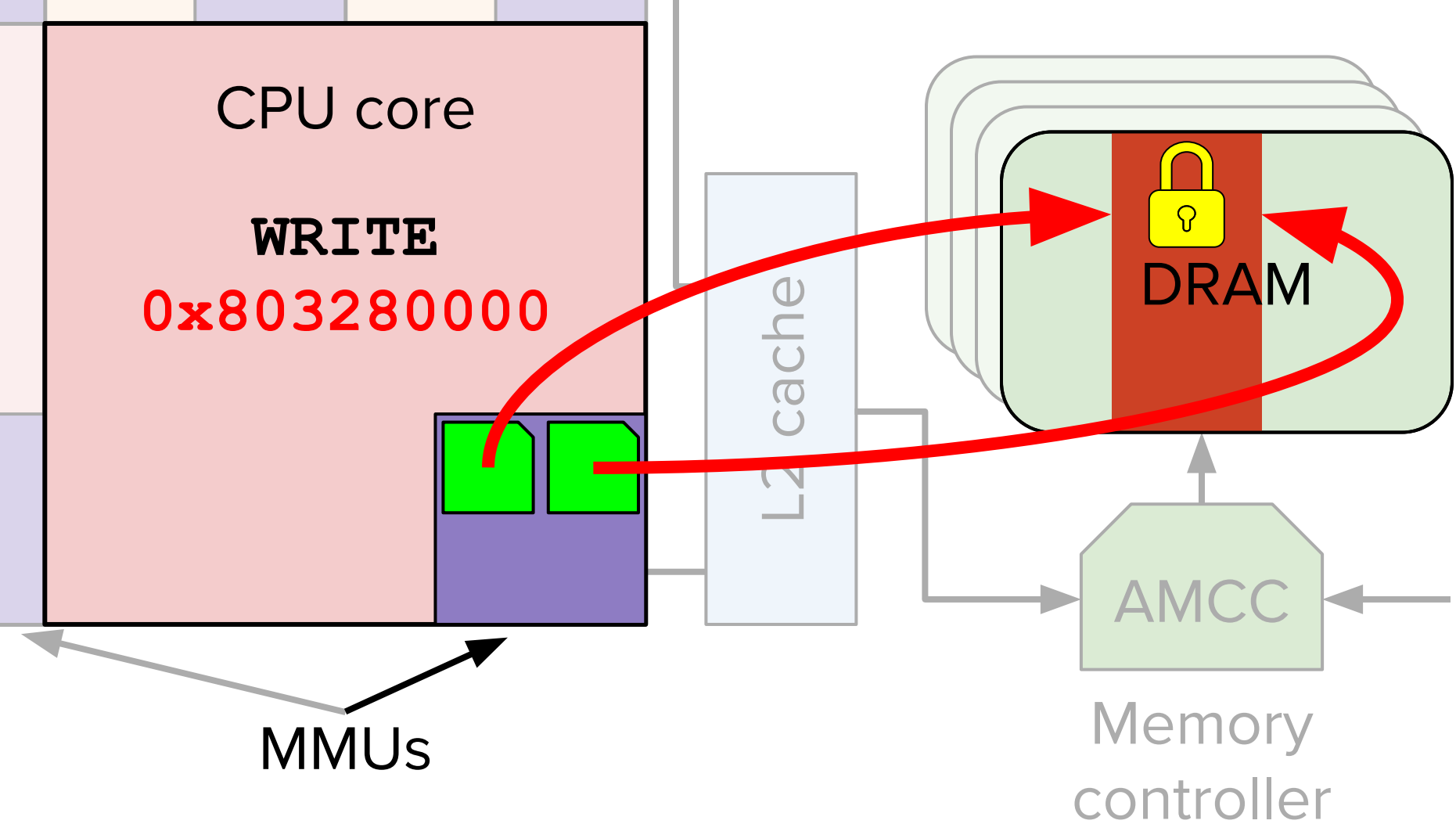


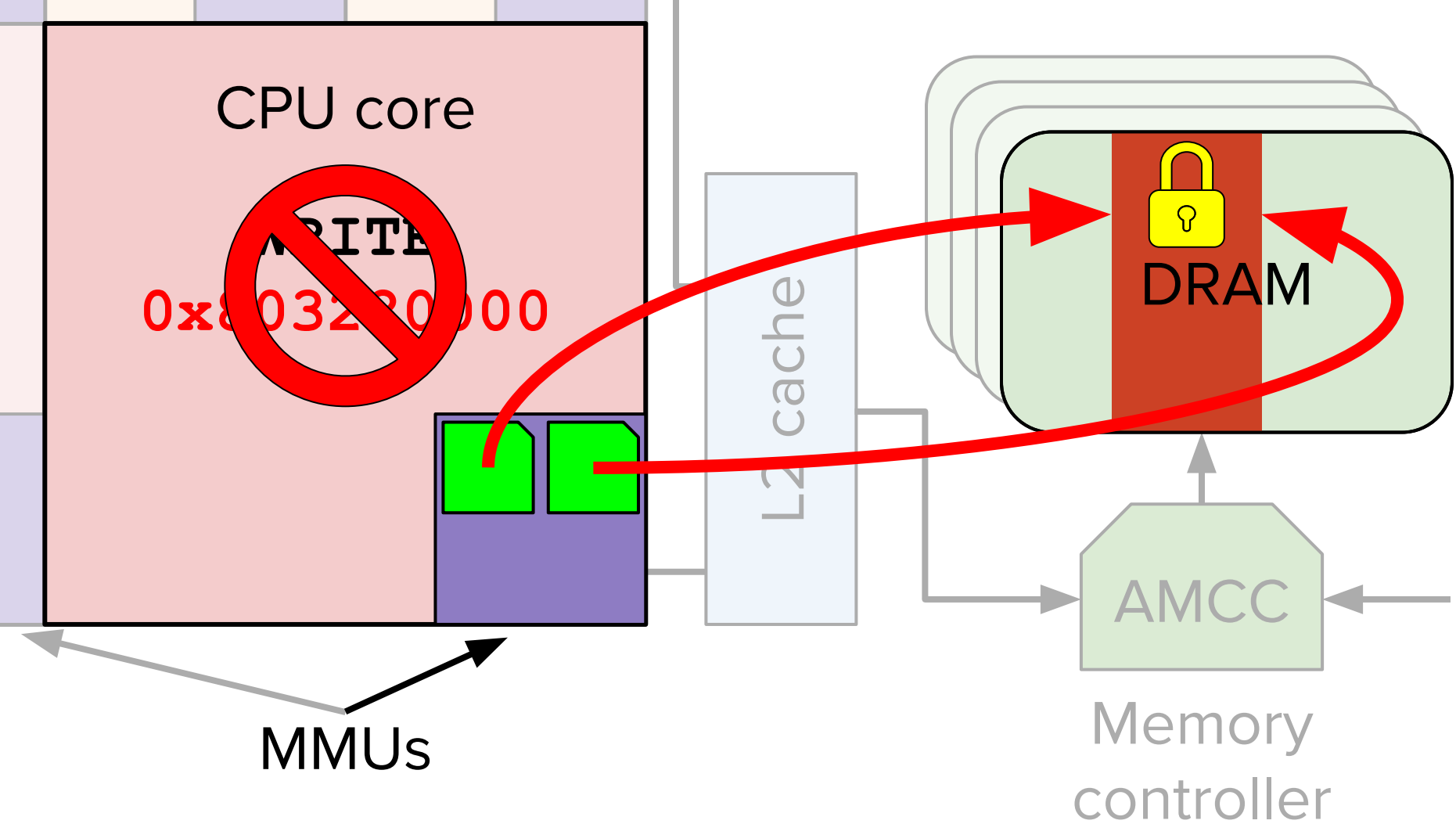


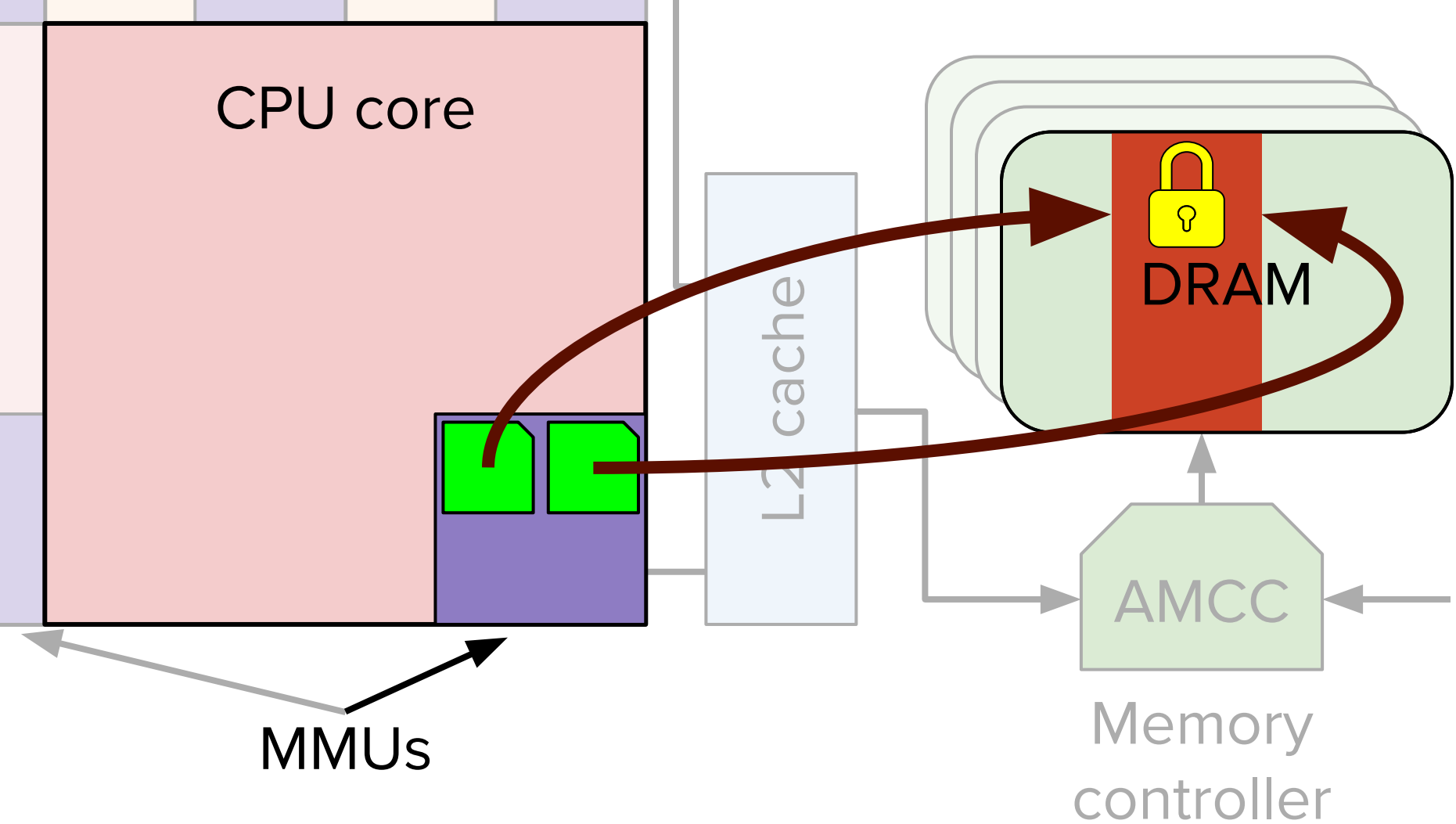


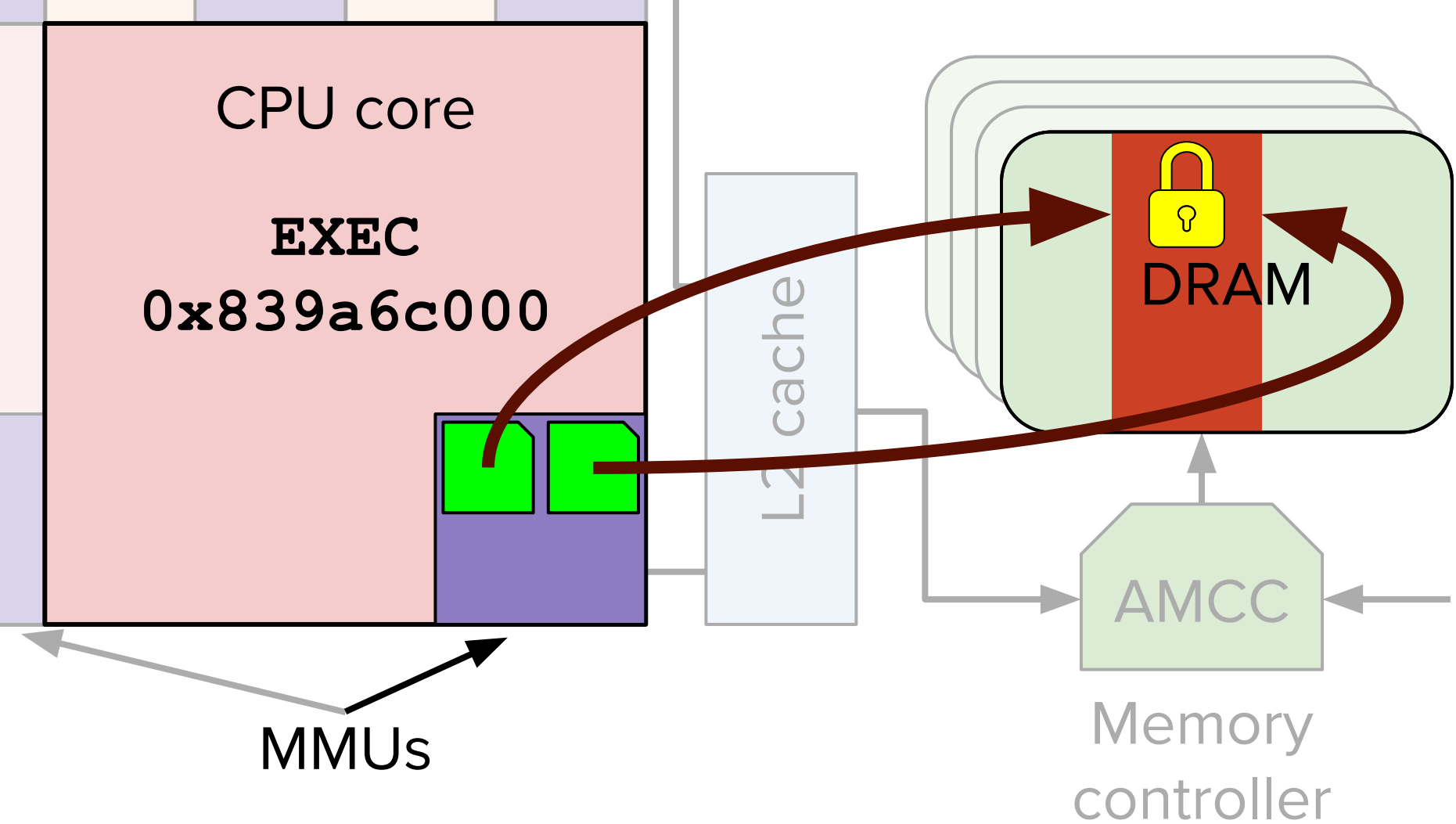


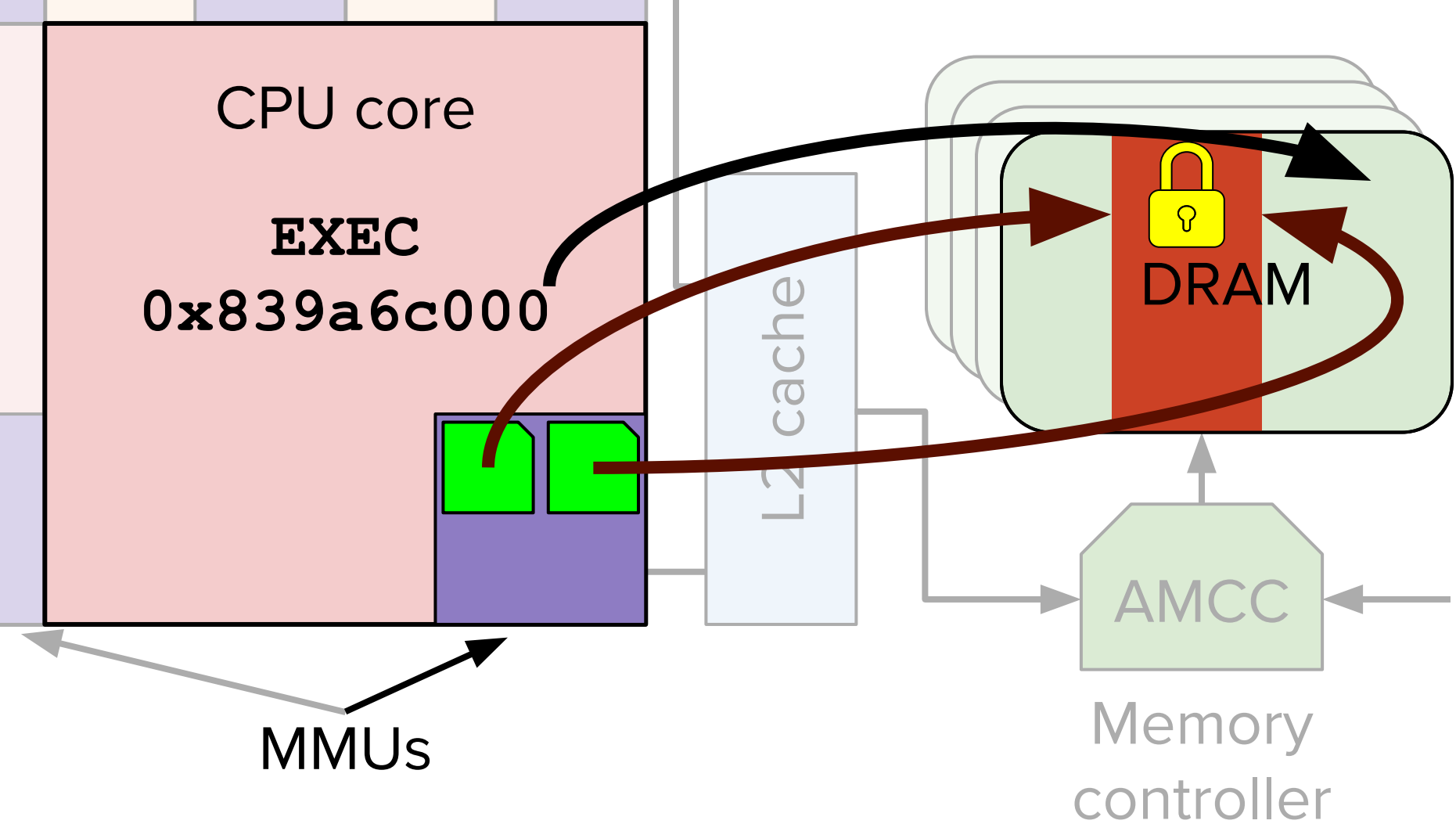


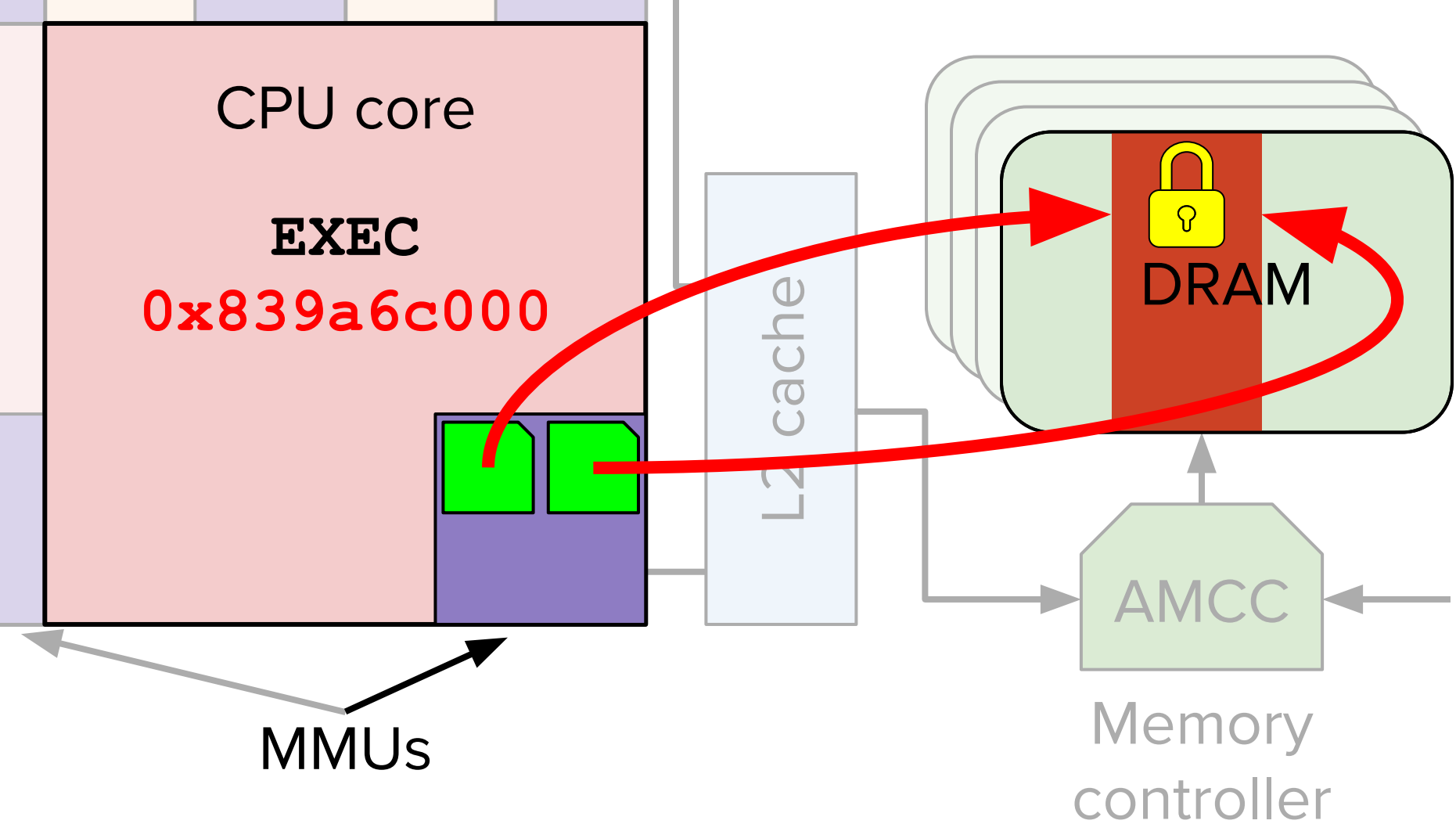


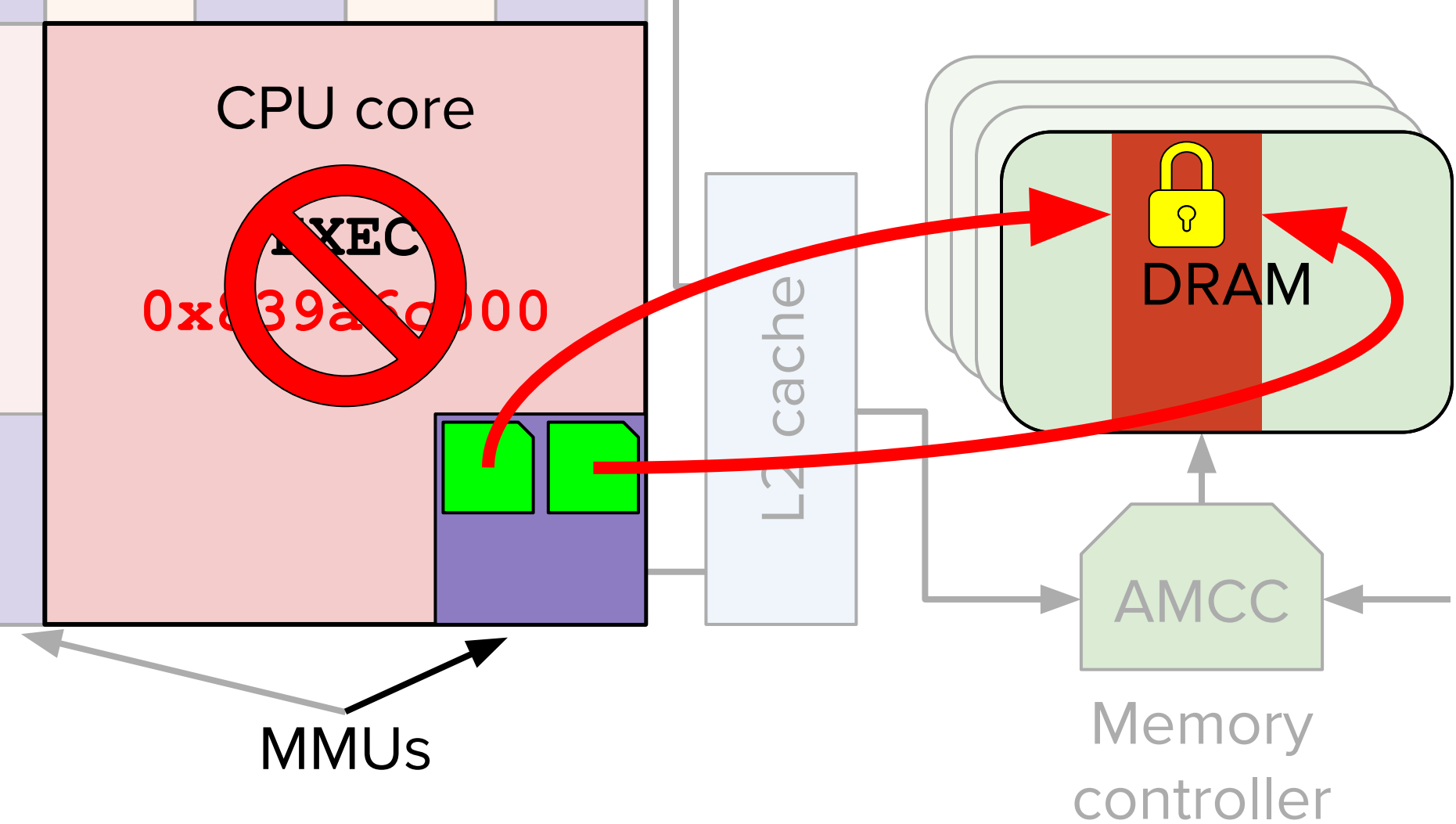


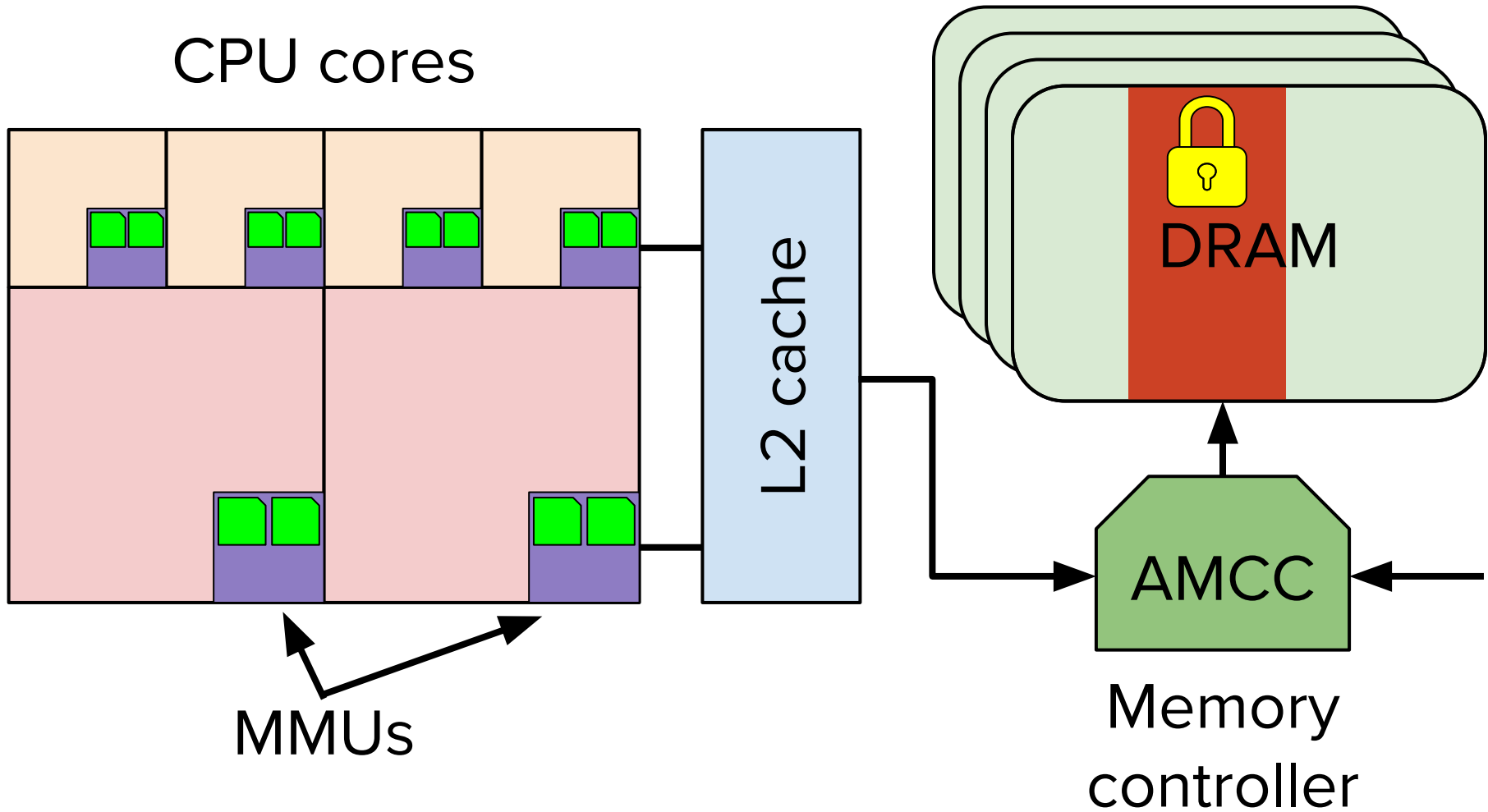


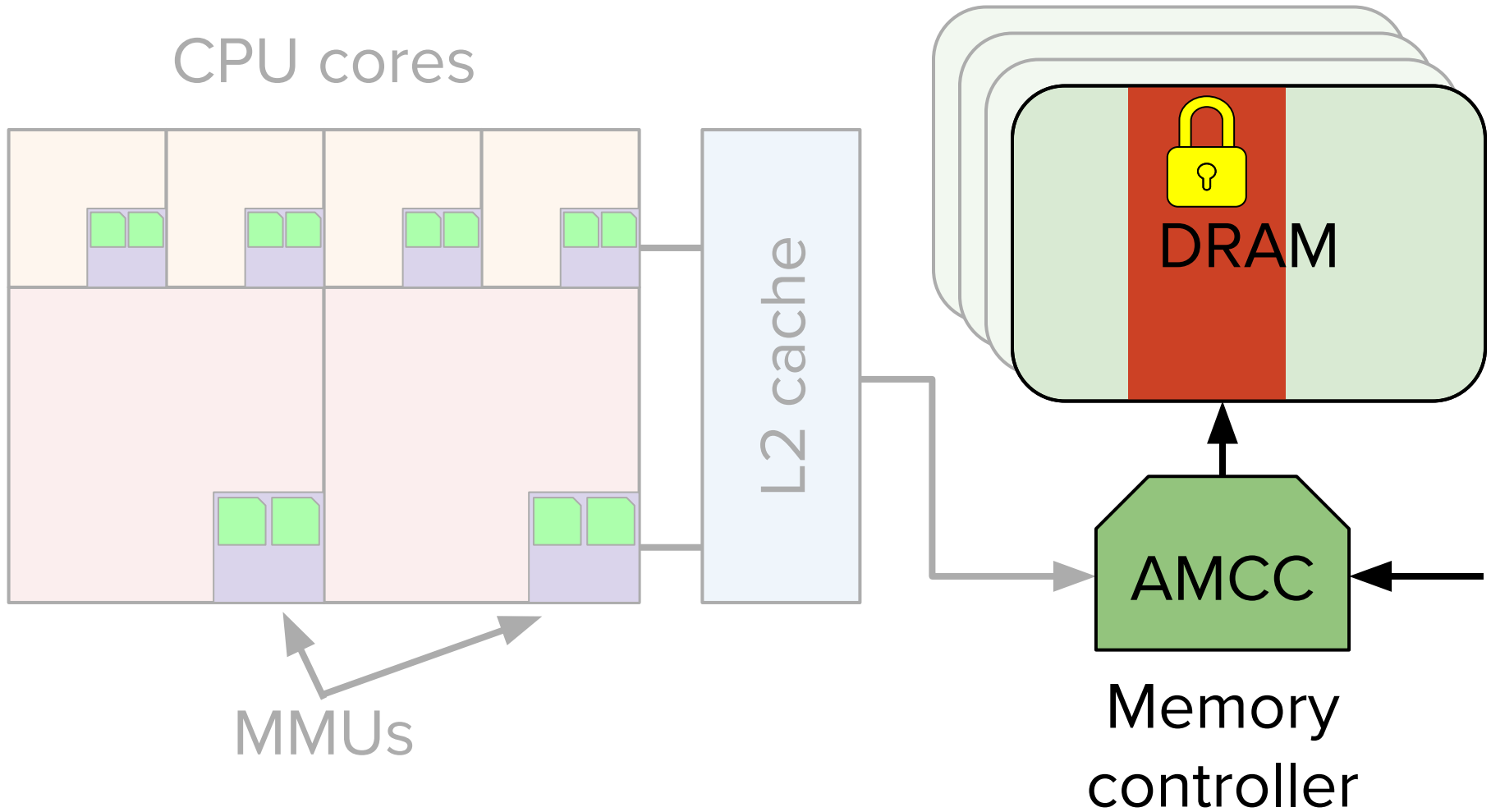












CPU cores

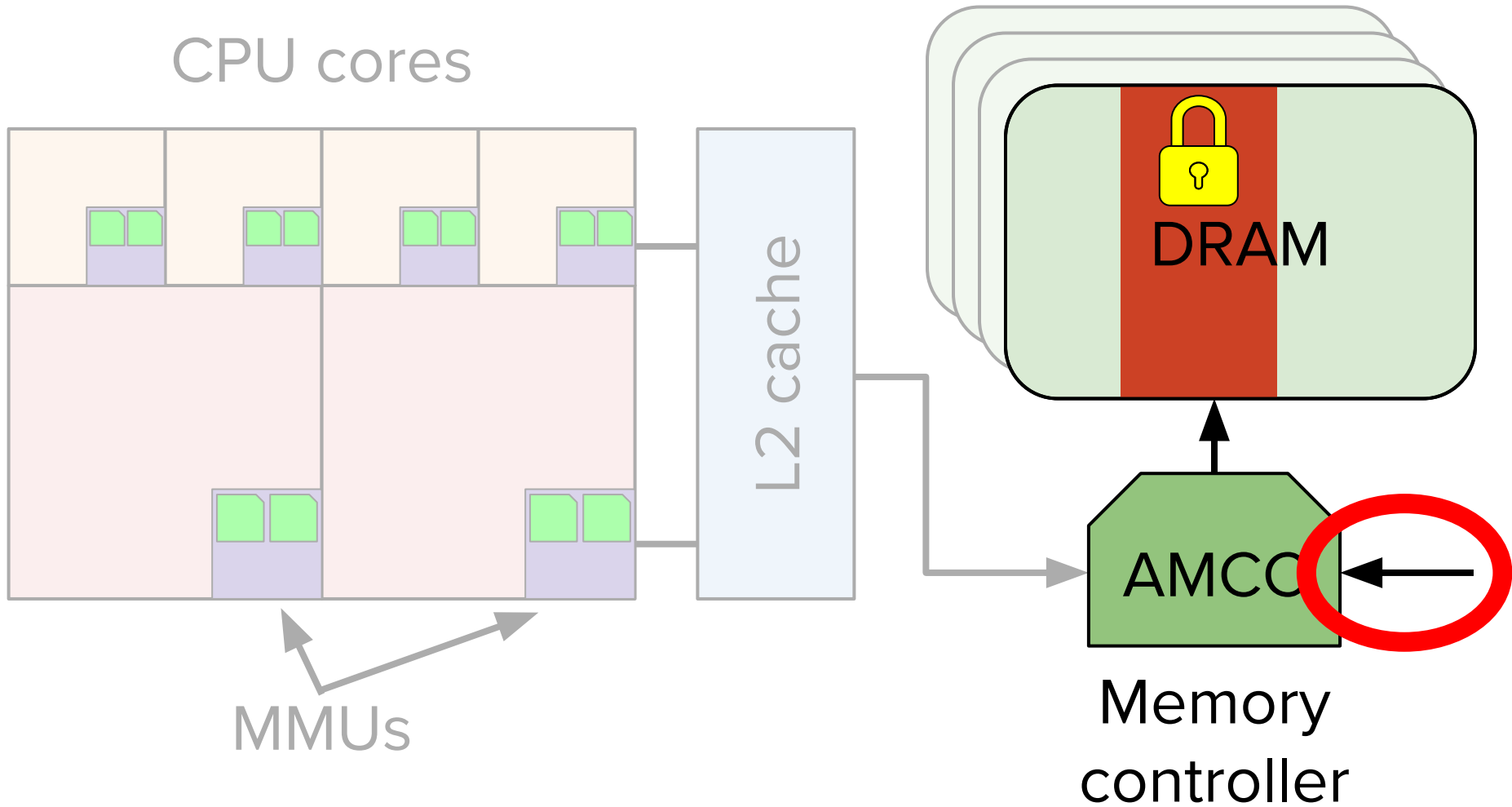
L2 cache

DRAM

AMCC

Memory controller

MMUs



CPU cores

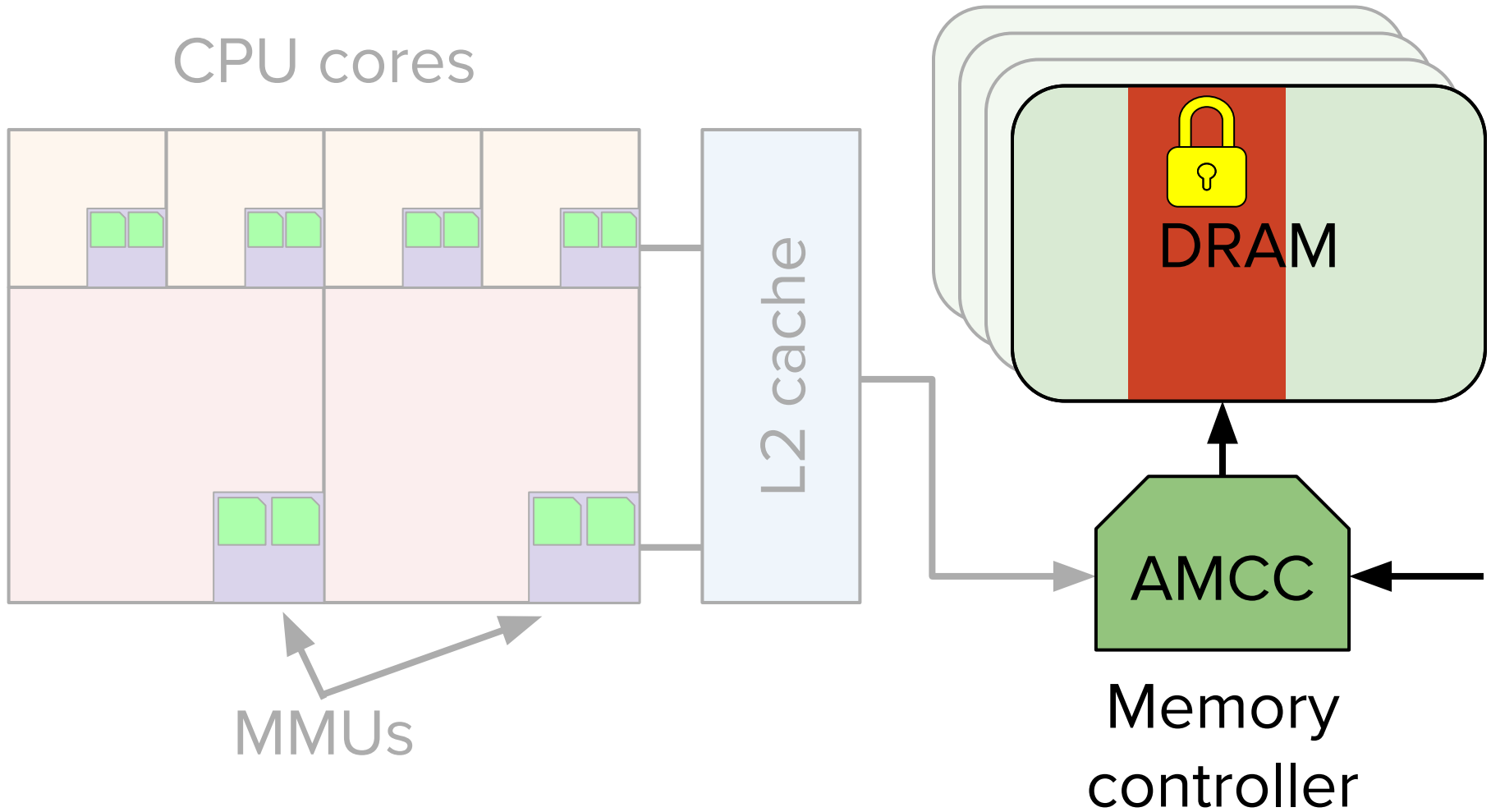
L2 cache

DRAM

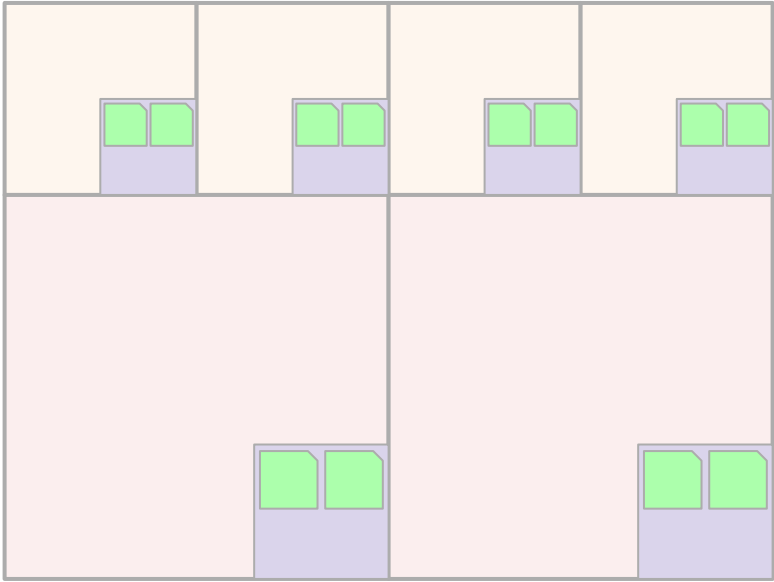
AMCC

Memory controller

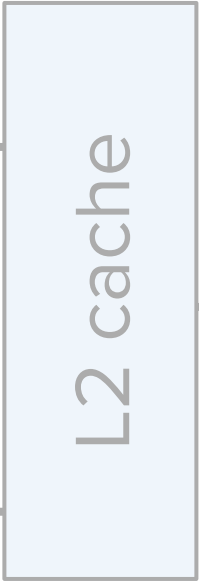
MMUs



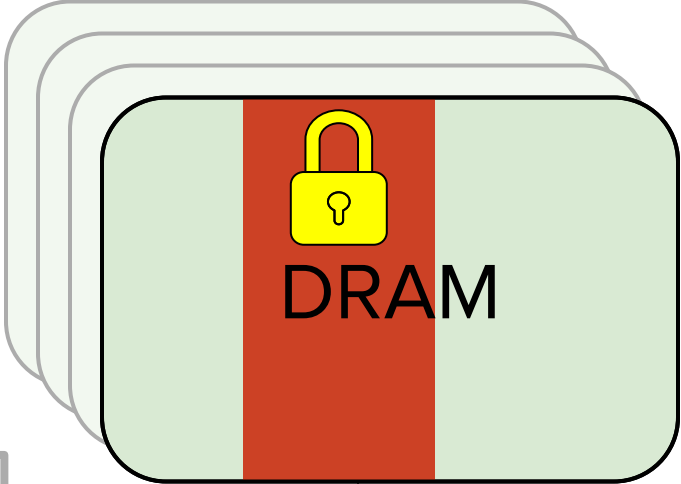
CPU cores



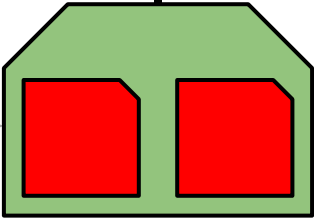
MMUs



L2 cache

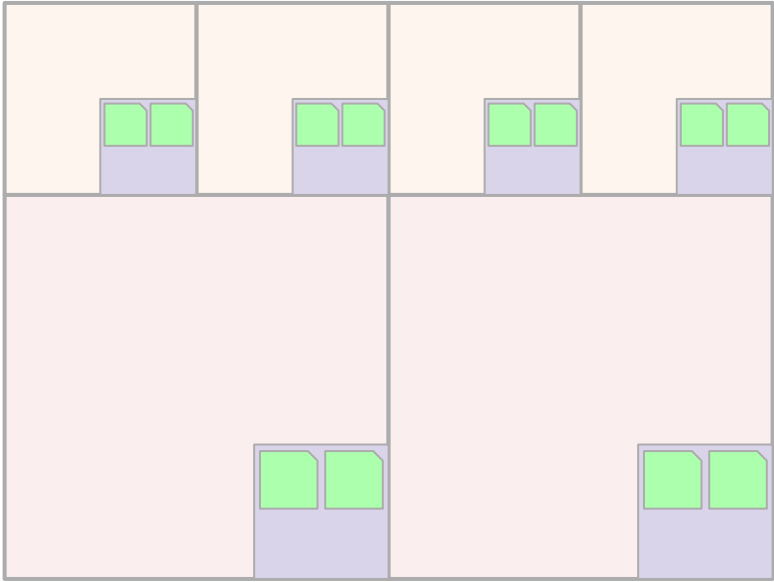


DRAM

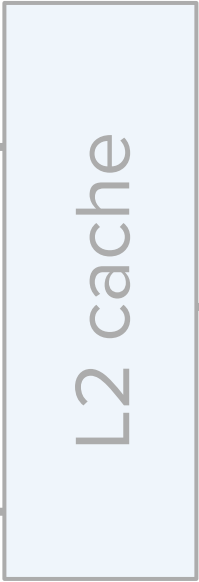


AMCC

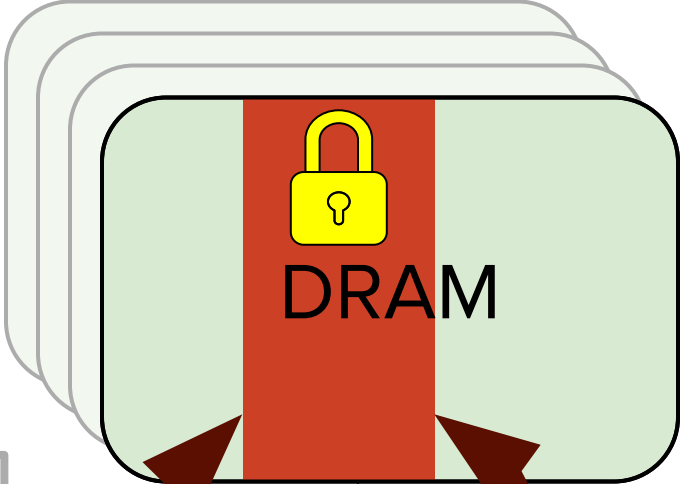
CPU cores



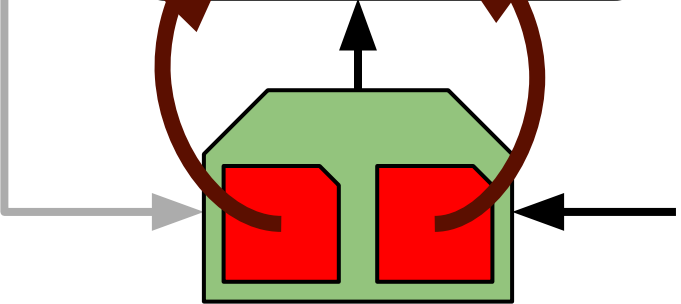
MMUs



L2 cache

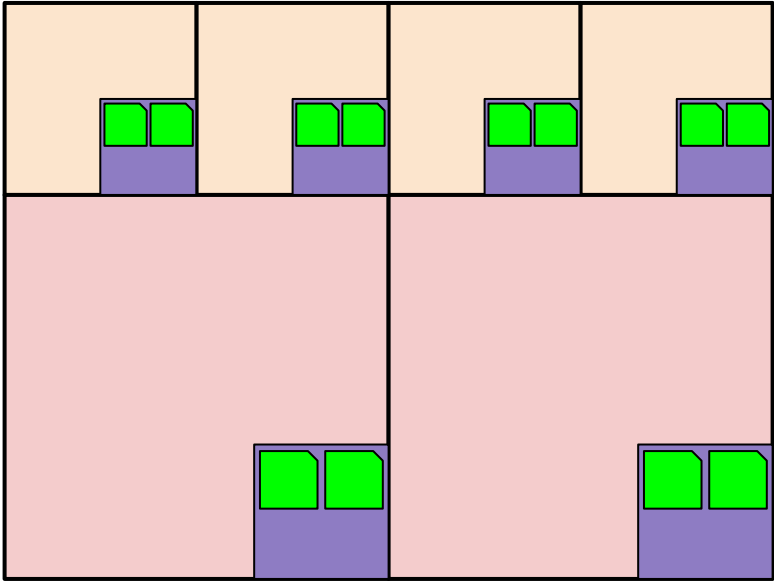


DRAM

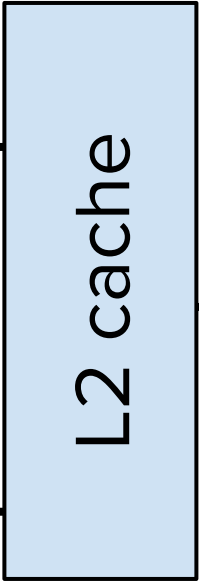


AMCC

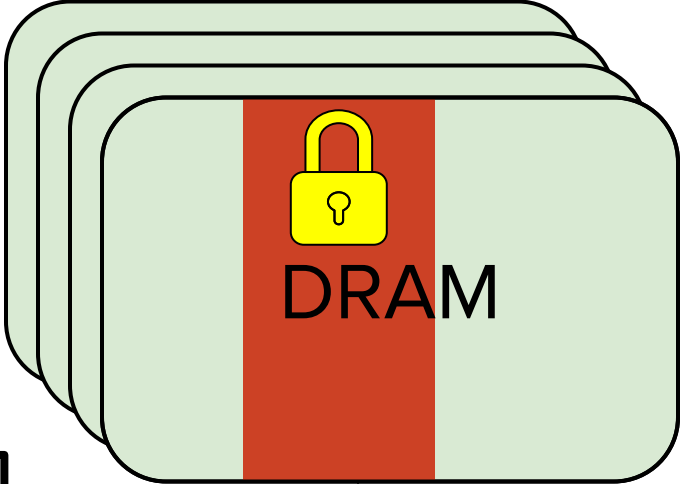
CPU cores



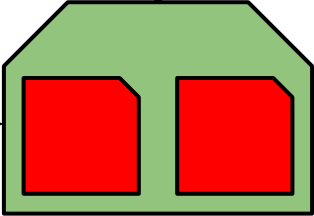
MMUs



L2 cache



DRAM

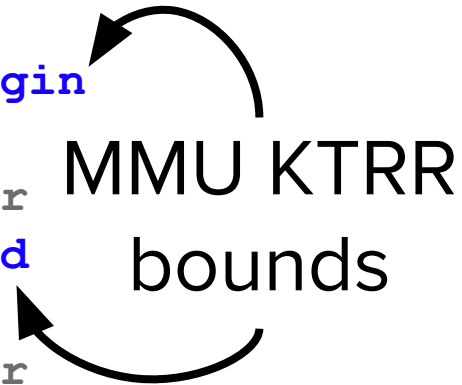


AMCC

MMU registers lose state
during core sleep

```
FFFFFFFF0070E4000 LowResetVectorBase
FFFFFFFF0070E4000 MSR #0, c1, c0, #4
FFFFFFFF0070E4004 MSR #6, #0xF
...
FFFFFFFF0070E4080 ADRL X17, _rorgn_begin
FFFFFFFF0070E4088 LDR X17, [X17]
FFFFFFFF0070E408C CBZ X17, Lskip_ktrr
FFFFFFFF0070E4090 ADRL X19, _rorgn_end
FFFFFFFF0070E4098 LDR X19, [X19]
FFFFFFFF0070E409C CBZ X19, Lskip_ktrr
FFFFFFFF0070E40A0 MSR ARM64_REG_KTRR_LOWER_EL1, X17
FFFFFFFF0070E40A4 SUB X19, X19, #4, LSL#12
FFFFFFFF0070E40A8 MSR ARM64_REG_KTRR_UPPER_EL1, X19
FFFFFFFF0070E40AC MOV X17, #1
FFFFFFFF0070E40B0 MSR ARM64_REG_KTRR_LOCK_EL1, X17
FFFFFFFF0070E40B4
FFFFFFFF0070E40B4 Lskip_ktrr ; CODE XREF: LowResetVectorBase+8C↑j
```

```
FFFFFFFF0070E4000 LowResetVectorBase
FFFFFFFF0070E4000 MSR #0, c1, c0, #4
FFFFFFFF0070E4004 MSR #6, #0xF
...
FFFFFFFF0070E4080 ADRL X17, _rorgn_begin
FFFFFFFF0070E4088 LDR X17, [X17]
FFFFFFFF0070E408C CBZ X17, Lskip_ktrr
FFFFFFFF0070E4090 ADRL X19, _rorgn_end
FFFFFFFF0070E4098 LDR X19, [X19]
FFFFFFFF0070E409C CBZ X19, Lskip_ktrr
FFFFFFFF0070E40A0 MSR ARM64_REG_KTRR_LOWER_EL1, X17
FFFFFFFF0070E40A4 SUB X19, X19, #4, LSL#12
FFFFFFFF0070E40A8 MSR ARM64_REG_KTRR_UPPER_EL1, X19
FFFFFFFF0070E40AC MOV X17, #1
FFFFFFFF0070E40B0 MSR ARM64_REG_KTRR_LOCK_EL1, X17
FFFFFFFF0070E40B4
FFFFFFFF0070E40B4 Lskip_ktrr ; CODE XREF: LowResetVectorBase+8C↑j
```



```

FFFFFFFF0070E4000 LowResetVectorBase
FFFFFFFF0070E4000 MSR #0, c1, c0, #4
FFFFFFFF0070E4004 MSR #6, #0xF
...
FFFFFFFF0070E4080 ADRL X17, _rorgn_begin
FFFFFFFF0070E4088 LDR X17, [X17]
FFFFFFFF0070E408C CBZ X17, Lskip_ktrr
FFFFFFFF0070E4090 ADRL X19, _rorgn_end
FFFFFFFF0070E4098 LDR X19, [X19]
FFFFFFFF0070E409C CBZ X19, Lskip_ktrr
FFFFFFFF0070E40A0 MSR ARM64_REG_KTRR_LOWER_EL1, X17
FFFFFFFF0070E40A4 SUB X19, X19, #4, LSL#12
FFFFFFFF0070E40A8 MSR ARM64_REG_KTRR_UPPER_EL1, X19
FFFFFFFF0070E40AC MOV X17, #1
FFFFFFFF0070E40B0 MSR ARM64_REG_KTRR_LOCK_EL1, X17
FFFFFFFF0070E40B4
FFFFFFFF0070E40B4 Lskip_ktrr ; CODE XREF: LowResetVectorBase+8C↑j

```

MMU KTRR
registers



Breaking KTRR

iOS 10.1.1: The Yalu KTRR bypass

Luca Todesco found that Apple accidentally left an MSR TTBR1_EL1 instruction executable

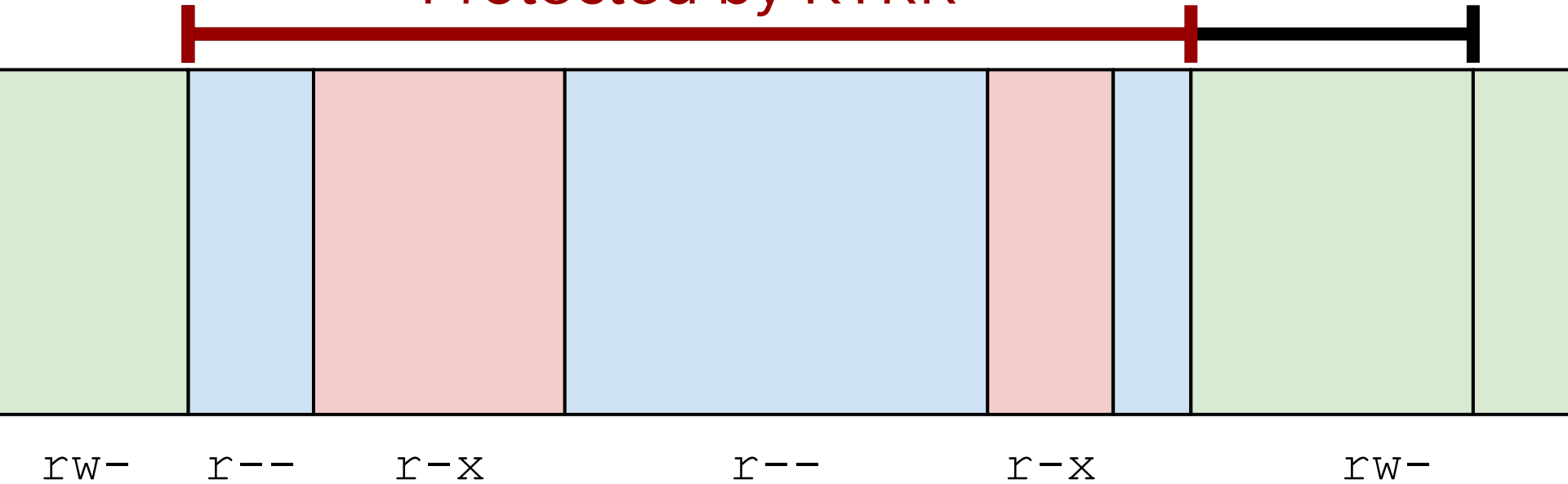
Use that instruction to set a new page table base

KTRR is still initialized (so no new kernel code), but readonly pages could now be made read/write

low addresses

high addresses

Protected by KTRR

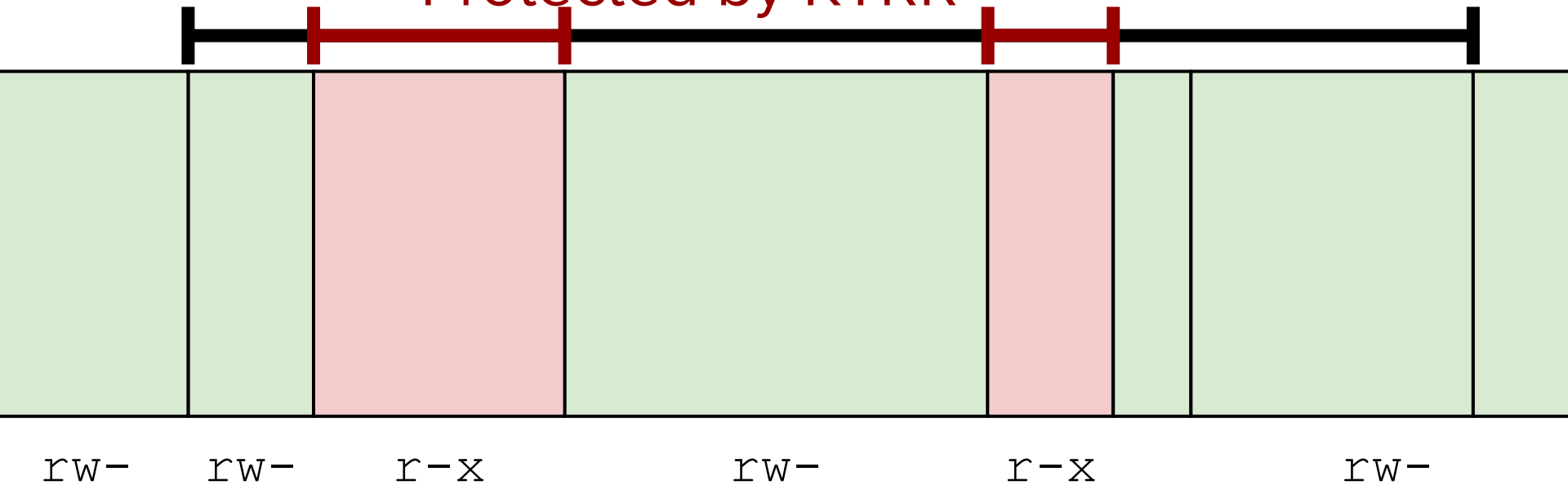


iPhone10,1 16G77 kernelcache

low addresses

high addresses

Protected by KTRR



iPhone10,1 16G77 kernelcache

iOS 11.1.2: Build your own iOS kernel debugger

Ian Beer found that self-hosted debugging could be enabled using ROP

Built an iOS kernel debugger with breakpoints for iOS 11.1.2 that works with LLDB

KTRR is still fully enabled, but existing instructions can be executed in arbitrary order

Attempts to bypass KTRR

iBoot bug: no

TLB "corruption": no

L2 cache "corruption": no



panic-base-2019-08-30-143759.i...



```
"kernel" : "Darwin Kernel Version 18.2.0: Mon Nov 12 20:32:02 PST 2018;
root:xnu-4903.232.2~1/RELEASE_ARM64_T8015",
"incident" : "479333C7-BAA2-4D77-8ABD-FD6C3B36B40C",
"crashReporterKey" : "ce81d5ec16fcde03b706d6a939240745a65782a2",
"date" : "2019-08-30 14:37:52.07 -0700",
"panicString" : "Attempting to forcibly halt cpu 1\ncpu 1 failed to halt with
error -5: halt not supported for this configuration\nDebugger
synchronization timed out; waited 10000000 nanoseconds\npanic(cpu 0
caller 0xffffffff00b5c96bc): \"WDT timeout: CPU 1 failed to respond\"@\
BuildRoot/Library/Caches/com.apple.xbs/Sources/AppleARMPlatform/
AppleARMPlatform-700.230.2/AppleARMWatchDogTimer.cpp:
501\nDebugger message: panic\nMemory ID: 0xff\nOS version:
16C101\nKernel version: Darwin Kernel Version 18.2.0: Mon Nov 12
20:32:02 PST 2018; root:xnu-4903.232.2~1/
RELEASE_ARM64_T8015\nKernelCache UUID:
EE1D833E1AF3B367B764DA9719881315\nKernel UUID:
94463A80-7B38-3176-8872-0B8E344C7138\niBoot version:
iBoot-4513.230.10\nsecure boot?: YES\nPaniclog version: 11\nKernel slide:
```



panic-base-2019-08-30-143759.i...



```
"kernel" : "Darwin Kernel Version 18.2.0: Mon Nov 12 20:32:02 PST 2018;
root:xnu-4903.232.2~1/RELEASE_ARM64_T8015",
"incident" : "479333C7-BAA2-4D77-8ABD-FD6C3B36B40C",
"crashReporterKey" : "ce81d5ec16fcde03b706d6a939240745a65782a2",
"date" : "2019-08-30 14:37:52.07 -0700",
"panicString" : "Attempting to forcibly halt cpu 1\ncpu 1 failed to halt with
error -5: halt not supported for this configuration\nDebugger
synchronization timed out; waited 10000000 nanoseconds\npanic(cpu 0
caller 0xffffffff00b5c96bc): \"WDT timeout: CPU 1 failed to respond\"@\
BuildRoot/Library/Caches/com.apple.xbs/Sources/AppleARMPlatform/
AppleARMPlatform-700.230.2/AppleARMWatchDogTimer.cpp:
501\nDebugger message: panic\nMemory ID: 0xff\nOS version:
16C101\nKernel version: Darwin Kernel Version 18.2.0: Mon Nov 12
20:32:02 PST 2018; root:xnu-4903.232.2~1/
RELEASE_ARM64_T8015\nKernelCache UUID:
EE1D833E1AF3B367B764DA9719881315\nKernel UUID:
94463A80-7B38-3176-8872-0B8E344C7138\niBoot version:
iBoot-4513.230.10\nsecure boot?: YES\nPaniclog version: 11\nKernel slide:
0x0000000000000000\nKernel text: 0x0000000000000000\nKernel binary
version: 0x0000000000000000"
```




panic-base-2019-08-30-143759.i...



```
"kernel" : "Darwin Kernel Version 18.2.0: Mon Nov 12 20:32:02 PST 2018;
root:xnu-4903.232.2~1/RELEASE_ARM64_T8015",
"incident" : "479333C7-BAA2-4D77-8ABD-FD6C3B36B40C",
"crashReporterKey" : "ce81d5ec16fcde03b706d6a939240745a65782a2",
"date" : "2019-08-30 14:37:52.07 -0700",
"panicString" : "Attempting to forcibly halt cpu 1\ncpu 1 failed to halt with
error -5: halt not supported for this configuration\nDebugger
synchronization timed out; waited 10000000 nanoseconds\npanic(cpu 0
caller 0xffffffff00b5c96bc): \"WDT timeout: CPU 1 failed to respond\"@\
BuildRoot/Library/Caches/com.apple.xbs/Sources/AppleARMPlatform/
AppleARMPlatform-700.230.2/AppleARMWatchDogTimer.cpp:
501\nDebugger message: panic\nMemory ID: 0xff\nOS version:
16C101\nKernel version: Darwin Kernel Version 18.2.0: Mon Nov 12
20:32:02 PST 2018; root:xnu-4903.232.2~1/
RELEASE_ARM64_T8015\nKernelCache UUID:
EE1D833E1AF3B367B764DA9719881315\nKernel UUID:
94463A80-7B38-3176-8872-0B8E344C7138\niBoot version:
iBoot-4513.230.10\nsecure boot?: YES\nPaniclog version: 11\nKernel slide:
0x0000000000000000\nKernel text base: 0xfffffff000000000\nSystem
..."
```



Library function Regular function Instruction Data Unexplored External symbol Lumina function

Functions window IDA View-A Pseudocode-A Strings window Structures Program Segmentation Local Types Hex View-1

```

Function name
sub_FFFFFFFF0076274EC
sub_FFFFFFFF007627788
sub_FFFFFFFF00762786C
sub_FFFFFFFF007627964
sub_FFFFFFFF007627A30
sub_FFFFFFFF007627B0C
sub_FFFFFFFF007627CFC
sub_FFFFFFFF00762C000
sub_FFFFFFFF00762C064
sub_FFFFFFFF00762C088
sub_FFFFFFFF00762C11C
sub_FFFFFFFF00762C128
sub_FFFFFFFF00762C12C
sub_FFFFFFFF00762C180
sub_FFFFFFFF00762C298
sub_FFFFFFFF00762CA54
sub_FFFFFFFF00762CD54
sub_FFFFFFFF00762CE28
sub_FFFFFFFF00762D020
sub_FFFFFFFF00762D3A4
sub_FFFFFFFF00762D55C
_KLD_InitFunc_0
_KLD_TermFunc_0

```

```

TEXT EXEC. text:FFFFFFFF0070E4000
TEXT EXEC. text:FFFFFFFF0070E4000 ; Attributes: noreturn
TEXT EXEC. text:FFFFFFFF0070E4000
TEXT EXEC. text:FFFFFFFF0070E4000 LowResetVectorBase
TEXT EXEC. text:FFFFFFFF0070E4000 MSR #0, c1, c0, #4
TEXT EXEC. text:FFFFFFFF0070E4004 MSR #6, #0xF
TEXT EXEC. text:FFFFFFFF0070E4008 MOVK X0, #0x4455, LSL#48
TEXT EXEC. text:FFFFFFFF0070E400C MOVK X0, #0x4455, LSL#32
TEXT EXEC. text:FFFFFFFF0070E4010 MOVK X0, #0x6466, LSL#16
TEXT EXEC. text:FFFFFFFF0070E4014 MOVK X0, #0x6677
TEXT EXEC. text:FFFFFFFF0070E4018 BL sub_FFFFFFFF0070E85A0
TEXT EXEC. text:FFFFFFFF0070E401C ADRP X4, #_aprr_jit@PAGE
TEXT EXEC. text:FFFFFFFF0070E4020 LDRB W5, [X4, #_aprr_jit@PAGEOFF]
TEXT EXEC. text:FFFFFFFF0070E4024 W5, #0
TEXT EXEC. text:FFFFFFFF0070E4028 B.NE loc_FFFFFFFF0070E4044
TEXT EXEC. text:FFFFFFFF0070E402C MOVK X0, #0x4545, LSL#48
TEXT EXEC. text:FFFFFFFF0070E4030 MOVK X0, #0x101, LSL#32
TEXT EXEC. text:FFFFFFFF0070E4034 MOVK X0, #0x6767, LSL#16
TEXT EXEC. text:FFFFFFFF0070E4038 MOVK X0, #0x101
TEXT EXEC. text:FFFFFFFF0070E403C BL sub_FFFFFFFF0070E8578
TEXT EXEC. text:FFFFFFFF0070E4040 B loc_FFFFFFFF0070E4058
TEXT EXEC. text:FFFFFFFF0070E4044 ; -----
TEXT EXEC. text:FFFFFFFF0070E4044 loc_FFFFFFFF0070E4044 ; CODE XREF: LowResetVectorBase+28tj
TEXT EXEC. text:FFFFFFFF0070E4048 MOVK X0, #0x4545, LSL#48
TEXT EXEC. text:FFFFFFFF0070E404C MOVK X0, #0x101, LSL#32
TEXT EXEC. text:FFFFFFFF0070E4050 MOVK X0, #0x6567, LSL#16
TEXT EXEC. text:FFFFFFFF0070E4054 MOVK X0, #0x101
TEXT EXEC. text:FFFFFFFF0070E4058 BL sub_FFFFFFFF0070E8578
TEXT EXEC. text:FFFFFFFF0070E4058 loc_FFFFFFFF0070E4058 ; CODE XREF: LowResetVectorBase+40tj
TEXT EXEC. text:FFFFFFFF0070E405C MOVK X0, #0, LSL#48
TEXT EXEC. text:FFFFFFFF0070E4060 MOVK X0, #0, LSL#32
TEXT EXEC. text:FFFFFFFF0070E4064 MOVK X0, #0, LSL#16

```

Line 80633 of 80634 000E0000 FFFFFFFF0070E4000: LowResetVectorBase (Synchronized with Hex View-1)

Output window

```

FFFFFFFF0070B9478: using guessed type __int64 *kernel_pmap;
FFFFFFFF0070B96E8: using guessed type __int64 gPhysBase;
FFFFFFFF0070B9700: using guessed type __int64 qword_FFFFFFFF0070B9700;
FFFFFFFF007663080: using guessed type __int64 kernel_map;
FFFFFFFF007663088: using guessed type __int64 page_size;

```

Python

bazad@bazad-macbookpro: ~/Developer/source/xnu -- zsh

```
Last login: Mon Dec 16 15:05:24 on ttys025
```

```
bazad@bazad-macbookpro ~ % cd ~/Developer/source/xnu
```

```
bazad@bazad-macbookpro ~/Developer/source/xnu (git)-[master] % grep -RF "Attempting to fo  
rcibly halt" .
```

```
./osfmk/arm/model_dep.c:                                paniclog_append_noflush("Attempti  
ng to forcibly halt cpu %d\n", cpu);
```

```
bazad@bazad-macbookpro ~/Developer/source/xnu (git)-[master] % █
```

Line 8

Ou

FFFF

FFFF

FFFFFFFF0070B9700: using guessed type __int64 qword_FFFFFFFF0070B9700;

FFFFFFFF007663080: using guessed type __int64 kernel_map;

FFFFFFFF007663088: using guessed type __int64 page_size;

Python

```
101 ml_dbgwrap_halt_cpu(int cpu_index, uint64_t timeout_ns)
102 {
103     cpu_data_t *cdp = cpu_datap(cpu_index);
104     if ((cdp == NULL) || (cdp->coresight_base[CORESIGHT_UTT] == 0))
105         return DBGWRAP_ERR_UNSUPPORTED;
106
107     /* Only one cpu is allowed to initiate the halt sequence, to prevent cpus from cr
108        oss-halting
109        * each other. The first cpu to request a halt may then halt any and all other c
110        pus besides itself. */
111     int curcpu = cpu_number();
112     if (cpu_index == curcpu)
113         return DBGWRAP_ERR_SELF_HALT;
114 }
```

```
1 osfmk/arm64/dbgwrap.c [c,utf-8,unix]
```

0

112,0-1

36%

```
FFFFF0070B9700: using guessed type __int64 qword_FFFFFF0070B9700;
FFFFF007663080: using guessed type __int64 kernel_map;
FFFFF007663088: using guessed type __int64 page_size;
```



```
114     (halt_from_cpu ≠ (uint32_t)curcpu))
115     return DBGWRAP_ERR_INPROGRESS;
116
117     volatile dbgwrap_reg_t *dbgWrapReg = (volatile dbgwrap_reg_t *) (cdp->coresight_base[CORESIGHT_UTT] + DBGWRAP_REG_OFFSET);
118
119     if (m1_dbgwrap_cpu_is_halted(cpu_index))
120         return DBGWRAP_WARN_ALREADY_HALTED;
121
122     /* Clear all other writable bits besides dbgHalt; none of the power-down or reset bits must be set. */
123     *dbgWrapReg = DBGWRAP_DBGHALT;
124
125     if (timeout_ns ≠ 0) {
```

1 osfmk/arm64/dbgwrap.c [c,utf-8,unix]

9

125,1-4

41%

```
FFFFF0070B9700: using guessed type __int64 qword_FFFFFF0070B9700;
FFFFF007663080: using guessed type __int64 kernel_map;
FFFFF007663088: using guessed type __int64 page_size;
```

```
114     (halt_from_cpu ≠ (uint32_t)curcpu))
115     return DBGWRAP_ERR_INPROGRESS;
116
117     volatile dbgwrap_reg_t *dbgWrapReg = (volatile dbgwrap_reg_t *) (cdp->coresight_base[CORESIGHT_UTT] + DBGWRAP_REG_OFFSET);
118
119     if (m1_dbgwrap_cpu_is_halted(cpu_index))
120         return DBGWRAP_WARN_ALREADY_HALTED;
121
122     /* Clear all other writable bits besides dbgHalt; none of the power-down or reset bits must be set. */
123     *dbgWrapReg = DBGWRAP_DBGHALT;
124
125     if (timeout_ns ≠ 0) {
```



**MMIO
register**

1 osfmk/arm64/dbgwrap.c [c,utf-8,unix] 9 125,1-4 41%

FFFFF0070B9700: using guessed type __int64 qword_FFFFFF0070B9700;
FFFFF007663080: using guessed type __int64 kernel_map;
FFFFF007663088: using guessed type __int64 page_size;

```
114     (halt_from_cpu ≠ (uint32_t)curcpu))
115     return DBGWRAP_ERR_INPROGRESS;
116
117     volatile dbgwrap_reg_t *dbgWrapReg = (volatile dbgwrap_reg_t *) (cdp->coresight_base[CORESIGHT_UTT] + DBGWRAP_REG_OFFSET);
118
119     if (ml_dbgwrap_cpu_is_halted(cpu_index))
120         return DBGWRAP_WARN_ALREADY_HALTED;
121
122     /* Clear all other writable bits besides dbgHalt; none of the power-down or reset bits must be set. */
123     *dbgWrapReg = DBGWRAP_DBGHALT;
124
125     if (timeout_ns ≠ 0) {
```

CoreSight?



Line 8 | 1 osfmk/arm64/dbgwrap.c [c,utf-8,unix]

9

125,1-4

41%

```
FFFF
FFFF
FFFF0070B9700: using guessed type __int64 qword_FFFFFF0070B9700;
FFFF007663080: using guessed type __int64 kernel_map;
FFFF007663088: using guessed type __int64 page_size;
```

```
dbgwrap_status_t
```

```
m1_dbgwrap_halt_cpu_with_state(int cpu_index,
```

```
    uint64_t timeout_ns, dbgwrap_thread_state_t *state) {
```

```
    cpu_data_t *cdp = cpu_datap(cpu_index);
```

```
...
```

```
/* Ensure memory-mapped coresight registers can be written */
```

```
*((volatile uint32_t *) (cdp->coresight_base[CORESIGHT_ED]  
    + ARM_DEBUG_OFFSET_DBGLAR)) = ARM_DBG_LOCK_ACCESS_KEY;
```

```
...
```

```
for (unsigned int i = 0; i < ...; ++i) {
```

```
    instr = (0xD51U << 20) | ... | i; // msr DBGDTR0, x<i>
```

```
    m1_dbgwrap_stuff_instr(cdp, instr, ...);
```

```
    state->x[i] = m1_dbgwrap_read_dtr(cdp, ...);
```

```
}
```

```
...
```

```
return status;
```



```
dbgwrap_status_t
```

```
m1_dbgwrap_halt_cpu_with_state(int cpu_index,  
    uint64_t timeout_ns, dbgwrap_thread_state_t *state) {  
    cpu_data_t *cdp = cpu_datap(cpu_index);
```

```
..
```

```
/* Ensure memory-mapped coresight registers can be written */  
*((volatile uint32_t *) (cdp->coresight_base[CORESIGHT_ED]  
    + ARM_DEBUG_OFFSET_DBGLAR)) = ARM_DBG_LOCK_ACCESS_KEY;
```

```
...
```

```
for (unsigned int i = 0; i < ...; ++i) {  
    instr = (0xD51U << 20) | ... | i; // msr DBGDTR0, x<i>  
    m1_dbgwrap_stuff_instr(cdp, instr, ...);  
    state->x[i] = m1_dbgwrap_read_dtr(cdp, ...);  
}
```

```
...
```

```
return status;
```

```
dbgwrap_status_t
```

```
m1_dbgwrap_halt_cpu_with_state(int cpu_index,
```

```
    uint64_t timeout_ns, dbgwrap_thread_state_t *state) {
```

```
    cpu_data_t *cdp = cpu_datap(cpu_index);
```

```
...
```

```
/* Ensure memory-mapped coresight registers can be written */
```

```
*((volatile uint32_t *) (cdp->coresight_base[CORESIGHT_ED]  
    + ARM_DEBUG_OFFSET_DBGLAR)) = ARM_DBG_LOCK_ACCESS_KEY;
```

```
...
```

```
for (unsigned int i = 0; i < ...; ++i) {  
    instr = (0xD51U << 20) | ... | i; // msr DBGDTR0, x<i>  
    m1_dbgwrap_stuff_instr(cdp, instr, ...);  
    state->x[i] = m1_dbgwrap_read_dtr(cdp, ...);  
}
```

```
...
```

```
return status;
```

```
dbgwrap
```

```
m1_dbgwrap
```

```
m1_dbgwrap_stuff_instr()
```

```
is executing
```

```
dynamically-generated
```

```
instructions!
```

```
*/
```

```
*(
```

```
...  
for (unsigned int i = 0; i < ...; ++i) {  
    instr = (0xD51U << 20) | ... | i; // msr DBGDTR0, x<i>  
    m1_dbgwrap_stuff_instr(cdp, instr, ...);  
    state->x[i] = m1_dbgwrap_read_dtr(cdp, ...);  
}
```

```
...  
return status;
```

```

896 // Set EDLAR to unlock the CoreSight registers.
897 uint32_t edlsr = kernel_call_7(ldr_w0_x0_ret, 1, (ed_mmio + 0xFB4));
898 INFO(" edlsr = %x", edlsr);
899 kernel_call_7(str_w0_x1_ret, 2, 0xc5acce55, (ed_mmio + 0xFB0));
900 edlsr = kernel_call_7(ldr_w0_x0_ret, 1, (ed_mmio + 0xFB4));
901 INFO(" edlsr = %x", edlsr);
902 }
903
904 // Try to call ml_dbgwrap_halt_cpu_with_state.
905 uint64_t dbgwrap_thread_state = kernel_vm_allocate(0x4000);
906 uint32_t halt_status = kernel_call_7(ml_dbgwrap_halt_cpu_with_state, 3, 2, 10000000,
    dbgwrap_thread_state);
907 INFO("halt_status = %x", halt_status);
908 arm_thread_state64_t state = {};
909 kernel_read(dbgwrap_thread_state, &state, sizeof(state));
910 for (int i = 0; i < sizeof(state.__x) / sizeof(state.__x[0]); i++) {
911     printf(" x[%2d] = %016llx\n", i, state.__x[i]);
912 }
913 printf(" fp = %016llx\n", state.__fp);
914 printf(" lr = %016llx\n", state.__lr);
915 printf(" sp = %016llx\n", state.__sp);
916 printf(" pc = %016llx\n", state.__pc);
917 printf(" cpsr = %08x\n", state.__cpsr);
918 sleep(1);
919

```

No Debug Session

```

x[23] = 0000000000000000
x[24] = 0000000000000000
x[25] = 0000000000000000
x[26] = 0000000000000000
x[27] = 0000000000000000
x[28] = 0000000000000000
fp = 0000000000000000
lr = 0000008042e406c
sp = 0000000000000000
pc = 0000008042e4120
cpsr = 600003c5

```

```
896 // Set EDLAR to unlock the CoreSight registers.
897 uint32_t edlrs = kernel_call_7(ldr w0_x0_ret, 1, (ed_mmio + 0xFB4));
```

```
acce55, (ed_mmio + 0xFB0));
1, (ed_mmio + 0xFB4));
```

```
locate(0x4000);
trap_halt_cpu_with_state, 3, 2, 10000000,
```

```
feof(state));
of(state.__x[0]); i++) {
ate.__x[i]);
```

```

x[23] = 0000000000000000
x[24] = 0000000000000000
x[25] = 0000000000000000
x[26] = 0000000000000000
x[27] = 0000000000000000
x[28] = 0000000000000000
fp = 0000000000000000
lr = 00000008042e406c
sp = 0000000000000000
pc = 00000008042e4120
cpsr = 600003c5
  
```

All Output ▾

```
x[26] = 0000000000000000
x[27] = 0000000000000000
x[28] = 0000000000000000
fp = 0000000000000000
lr = 00000008042e406c
sp = 0000000000000000
pc = 00000008042e4120
cpsr = 600003c5
```

```
896 // Set EDLAR to unlock the CoreSight registers.
897 uint32_t edlrs = kernel_call_7(ldr w0_x0_ret, 1, (ed_mmio + 0xFB4));
```

```
acce55, (ed_mmio + 0xFB0));
1, (ed_mmio + 0xFB4));
```

```
locate(0x4000);
trap_halt_cpu_with_state, 3, 2, 10000000,
```

```
feof(state));
of(state.__x[0]); i++) {
ate.__x[i]);
```

```

x[23] = 0000000000000000
x[24] = 0000000000000000
x[25] = 0000000000000000
x[26] = 0000000000000000
x[27] = 0000000000000000
x[28] = 0000000000000000
fp = 0000000000000000
lr = 0000008042e406c
sp = 0000000000000000
pc = 0000008042e4120
cpsr = 600003c5
  
```

All Output ▾



Kernel mode

```
896 // Set EDLAR to unlock the CoreSight registers.  
897 uint32_t edlrs = kernel_call_7(ldr w0_x0_ret, 1, (ed_mmio + 0xFB4));
```

```
acce55, (ed_mmio + 0xFB0));  
1, (ed_mmio + 0xFB4));
```

```
locate(0x4000);  
trap_halt_cpu_with_state, 3, 2, 10000000,
```

```
feof(state));  
of(state.__x[0]); i++) {  
ate.__x[i]);
```

```
x[23] = 0000000000000000  
x[24] = 0000000000000000  
x[25] = 0000000000000000  
x[26] = 0000000000000000  
x[27] = 0000000000000000  
x[28] = 0000000000000000  
fp = 0000000000000000  
lr = 0000008042e406c  
sp = 0000000000000000  
pc = 0000008042e4120  
cpsr = 600003c5
```

All Output ▾



Reset vector!

```
x[26] = 0000000000000000  
x[27] = 0000000000000000  
x[28] = 0000000000000000  
fp = 0000000000000000  
lr = 0000008042e406c  
sp = 0000000000000000  
pc = 0000008042e4120  
cpsr = 600003c5
```

We've halted execution in the reset vector, before the MMU has been turned on!

We've halted execution in the reset vector, before the MMU has been turned on!

How do we use this?

We

the

MM

Ho

What exactly is this
CoreSight thing
anyway?

H8.4 Memory-mapped accesses to the external debug interface

Support for memory-mapped access to the external debug interface is OPTIONAL. When memory-mapped access to the external debug interface is supported, the external debug interface is accessed as a little-endian memory-mapped peripheral.

If the external debug interface is **CoreSight** compliant, then an OPTIONAL Software Lock can be implemented for memory-mapped accesses to each component.

The Software Lock is OPTIONAL and deprecated. If **ARMv8.4-Debug** is implemented, the Software Lock is not implemented. If it is not implemented, the behavior is as if it is unlocked. The Software Locks are controlled by **EDLSR** and **EDLAR**, **PMLSR** and **PMLAR**, and **CTILSR** and **CTILAR**. See *Management registers and CoreSight compliance on page K2-7237*.

With the exception of these registers and the effect of the Software Lock, the behavior of the memory-mapped accesses is the same as for other accesses to the external debug interface.

———— **Note** —————

The recommended memory-mapped accesses to the external debug interface are not compatible with the memory-mapped interface defined in ARMv7. In particular:

- The memory map is different.
- Memory-mapped accesses do not behave differently to Debug Access Port accesses when **OSLSR.OSLK** = 1, meaning that the OS Lock is locked.

The following sections give more information about these memory-mapped accesses:

Nailgun Attack

Break the privilege isolation in ARM devices

Overview

Processors nowadays are consistently equipped with debugging features to facilitate the program debugging and analysis. Specifically, the ARM debugging architecture involves a series of **CoreSight** components and debug registers to aid the system debugging, but the security of the debugging features is under-examined since it normally requires physical access to use these features in the traditional debugging model.

The idea of Nailgun Attack is to misuse the debugging architecture with the inter-processor debugging model. In the inter-processor debugging model, a processor

CoreSight External Debug Interface

On-chip debug architecture

Per-CPU debug registers accessible via MMIO

Extensively documented in the ARMv8 manual

Can set breakpoints/watchpoints, execute instructions, etc

The memory-mapped version of the debug registers used by Ian Beer in "Build your own iOS kernel debugger"

Use External Debug
to single-step
LowResetVectorBase
and skip KTRR lockdown

LowResetVectorBase

```
FFFFFFFF0070E4000 MSR #0, c1, c0, #4
FFFFFFFF0070E4004 MSR #6, #0xF
...
FFFFFFFF0070E4080 ADRL X17, _rorgn_begin
FFFFFFFF0070E4088 LDR X17, [X17]
FFFFFFFF0070E408C CBZ X17, Lskip_ktrr
FFFFFFFF0070E4090 ADRL X19, _rorgn_end
FFFFFFFF0070E4098 LDR X19, [X19]
FFFFFFFF0070E409C CBZ X19, Lskip_ktrr
FFFFFFFF0070E40A0 MSR ARM64_REG_KTRR_LOWER_EL1, X17
FFFFFFFF0070E40A4 SUB X19, X19, #4, LSL#12
FFFFFFFF0070E40A8 MSR ARM64_REG_KTRR_UPPER_EL1, X19
FFFFFFFF0070E40AC MOV X17, #1
FFFFFFFF0070E40B0 MSR ARM64_REG_KTRR_LOCK_EL1, X17
FFFFFFFF0070E40B4 Lskip_ktrr ; CODE XREF: LowResetVectorBase+8C↑j
FFFFFFFF0070E40B4
```



Challenges

We can halt a CPU core, and we can execute instructions on it to modify state, but how do we resume execution after making our modifications?

We are using one CPU core to hijack another as it executes the reset vector, so how do we handle subsequent core resets?

- voucher_swap
 - app
 - AppDelegate.h
 - AppDelegate.m
 - ViewController.h
 - ViewController.m
 - Main.storyboard
 - Assets.xcassets
 - LaunchScreen.storyboard
 - Info.plist
 - main.m
 - headers
 - IOKitLib.h
 - ipc_port.h
 - mach_vm.h
 - voucher_swap
 - kernel_call
 - kc_parameters.h
 - kc_parameters.c
 - pac.h
 - pac.c
 - user_client.h
 - user_client.c
 - kernel_alloc.h
 - kernel_alloc.c
 - kernel_call.h
 - kernel_call.c
 - kernel_memory.h
 - kernel_memory.c
 - kernel_slide.h
 - kernel_slide.c
 - log.h
 - log.c
 - parameters.h
 - parameters.c
 - platform.h
 - platform.c
 - platform_match.h
 - platform_match.c
 - voucher_swap.h

```

344     } while ((dbgwrap & (1 << 28)) == 0);
345
346     // The CPU is now halted in Debug state at reset.
347     DEBUG_TRACE(1, "Halted CPU %u in Debug State", cpu_id);
348     DEBUG_TRACE(1, "DBGWRAP = %llx", (dbgwrap = kernel_ioread64(dbgwrap_reg)));
349     print_edprsr();
350
351     // TODO: If this operation actually prevents the CPU from resetting, we should observe no
352     // panic later.
353
354     // Restart.
355     DEBUG_TRACE(1, "Restarting CPU %u", cpu_id);
356     kernel_iowrite64(dbgwrap_reg, (1 << 30) | (1 << 29) | (1 << 26));
357     DEBUG_TRACE(1, "DBGWRAP = %llx", kernel_ioread64(dbgwrap_reg));
358     print_edprsr();
359     kernel_iowrite64(dbgwrap_reg, (1 << 29) | (1 << 26));
360     DEBUG_TRACE(1, "DBGWRAP = %llx", kernel_ioread64(dbgwrap_reg));
361
362     sample_cpus();
363
364     for (int i = 0; i < 20; i++) {
365         print_edprsr();
366         sleep(1);
367     }
368 }

```

```

[D] Restarting CPU 2
[D] DBGWRAP = 4000000
[D] EDPRSR: SDR=1 H=0 SR=0 R=0 SPD=0 PU=1
[D] DBGWRAP = 24000000
[D] Available cpus: 3f
[D] EDPRSR: SDR=0 H=0 SR=0 R=0 SPD=0 PU=1
[D] EDPRSR: SDR=0 H=0 SR=0 R=0 SPD=0 PU=1
[D] EDPRSR: SDR=0 H=0 SR=0 R=0 SPD=0 PU=1
[D] EDPRSR: SDR=0 H=0 SR=0 R=0 SPD=0 PU=1
[D] EDPRSR: SDR=0 H=0 SR=0 R=0 SPD=0 PU=1
[D] EDPRSR: SDR=0 H=0 SR=0 R=0 SPD=0 PU=1
[D] EDPRSR: SDR=0 H=0 SR=0 R=0 SPD=0 PU=1

```

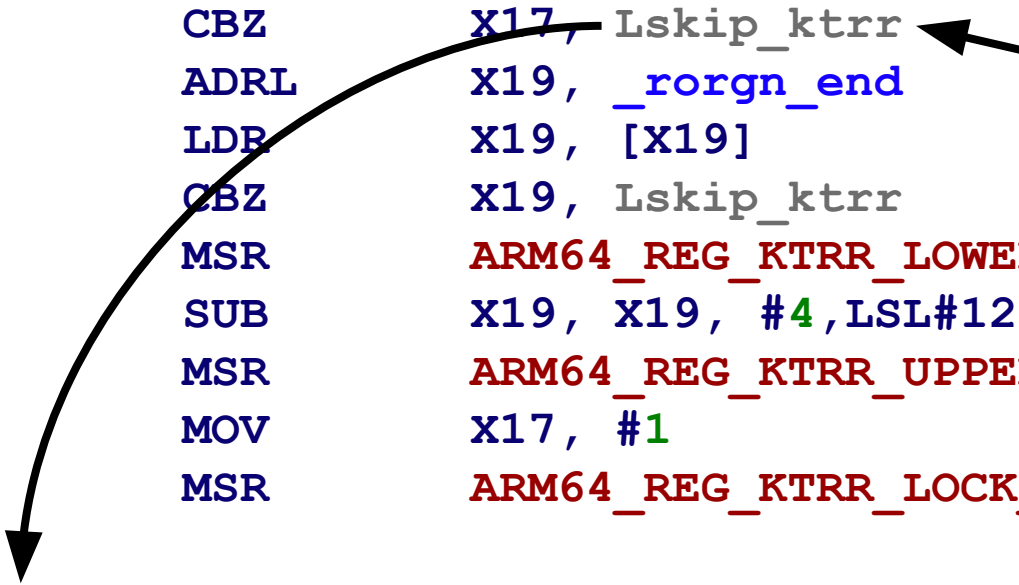
```

FFFFFFFF0070E4000 LowResetVectorBase
FFFFFFFF0070E4000      MSR      #0, c1, c0, #4
FFFFFFFF0070E4004      MSR      #6, #0xF
...
FFFFFFFF0070E4080      ADRL     X17, _rorgn_begin
FFFFFFFF0070E4088      LDR     X17, [X17]
FFFFFFFF0070E408C      CBZ     X17, Lskip_ktrr ← Take this
FFFFFFFF0070E4090      ADRL     X19, _rorgn_end      branch
FFFFFFFF0070E4098      LDR     X19, [X19]
FFFFFFFF0070E409C      CBZ     X19, Lskip_ktrr
FFFFFFFF0070E40A0      MSR     ARM64_REG_KTRR_LOWER_EL1, X17
FFFFFFFF0070E40A4      SUB     X19, X19, #4, LSL#12
FFFFFFFF0070E40A8      MSR     ARM64_REG_KTRR_UPPER_EL1, X19
FFFFFFFF0070E40AC      MOV     X17, #1
FFFFFFFF0070E40B0      MSR     ARM64_REG_KTRR_LOCK_EL1, X17
FFFFFFFF0070E40B4
FFFFFFFF0070E40B4 Lskip_ktrr ; CODE XREF: LowResetVectorBase+8C↑j

```

```
FFFFFFFF0070E4000 LowResetVectorBase
FFFFFFFF0070E4000 MSR #0, c1, c0, #4
FFFFFFFF0070E4004 MSR #6, #0xF
...
FFFFFFFF0070E4080 ADRL X17, _rorgn_begin
FFFFFFFF0070E4088 LDR X17, [X17]
FFFFFFFF0070E408C CBZ X17, Lskip_ktrr
FFFFFFFF0070E4090 ADRL X19, _rorgn_end
FFFFFFFF0070E4098 LDR X19, [X19]
FFFFFFFF0070E409C CBZ X19, Lskip_ktrr
FFFFFFFF0070E40A0 MSR ARM64_REG_KTRR_LOWER_EL1, X17
FFFFFFFF0070E40A4 SUB X19, X19, #4, LSL#12
FFFFFFFF0070E40A8 MSR ARM64_REG_KTRR_UPPER_EL1, X19
FFFFFFFF0070E40AC MOV X17, #1
FFFFFFFF0070E40B0 MSR ARM64_REG_KTRR_LOCK_EL1, X17
FFFFFFFF0070E40B4 Lskip_ktrr ; CODE XREF: LowResetVectorBase+8C↑j
FFFFFFFF0070E40B4
```

Take this branch



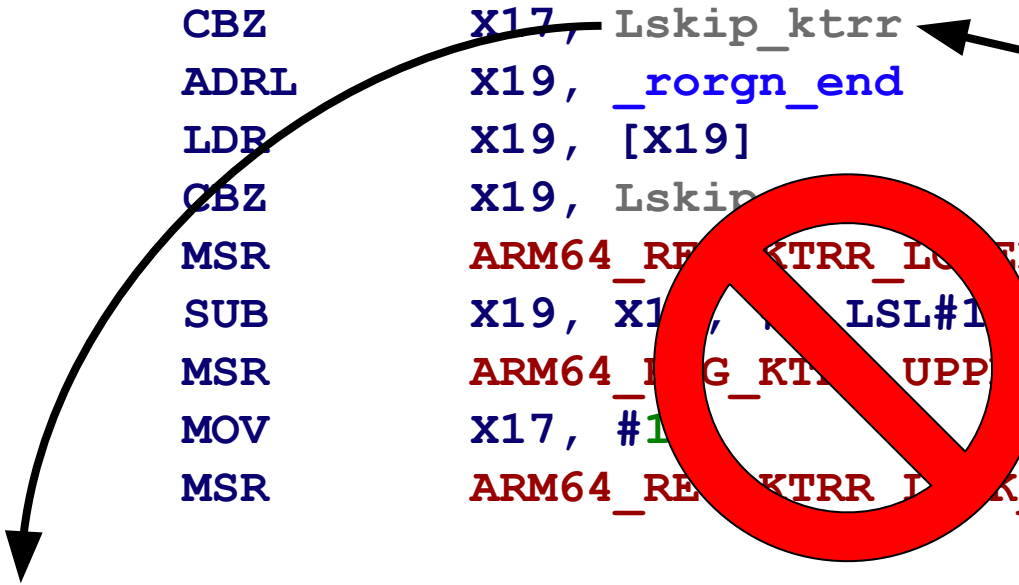
LowResetVectorBase

```

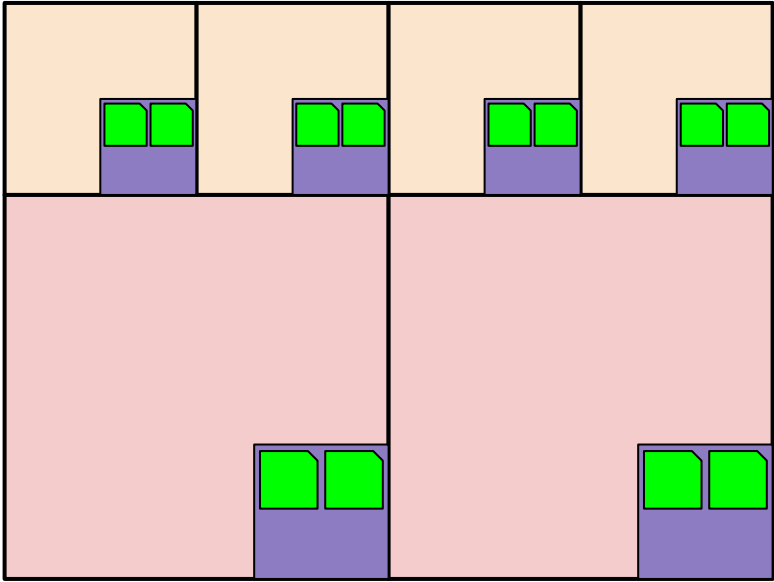
FFFFFFFF0070E4000 MSR      #0, c1, c0, #4
FFFFFFFF0070E4004 MSR      #6, #0xF
...
FFFFFFFF0070E4080 ADRL    X17, _rorgn_begin
FFFFFFFF0070E4088 LDR     X17, [X17]
FFFFFFFF0070E408C CBZ     X17, Lskip_ktrr
FFFFFFFF0070E4090 ADRL    X19, _rorgn_end
FFFFFFFF0070E4098 LDR     X19, [X19]
FFFFFFFF0070E409C CBZ     X19, Lskip_ktrr
FFFFFFFF0070E40A0 MSR     ARM64_REG_KTRR_LOWER_EL1, X17
FFFFFFFF0070E40A4 SUB     X19, X19, #1, LSL#1
FFFFFFFF0070E40A8 MSR     ARM64_REG_KTRR_UPPER_EL1, X19
FFFFFFFF0070E40AC MOV     X17, #1
FFFFFFFF0070E40B0 MSR     ARM64_REG_KTRR_LOWER_EL1, X17
FFFFFFFF0070E40B4 Lskip_ktrr ; CODE XREF: LowResetVectorBase+8C↑j

```

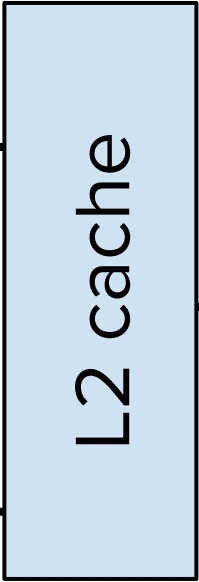
Take this branch



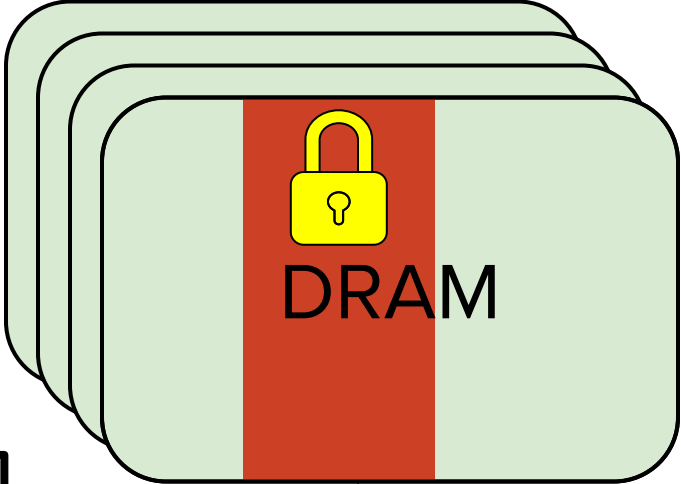
CPU cores



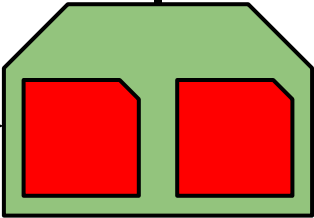
MMUs



L2 cache

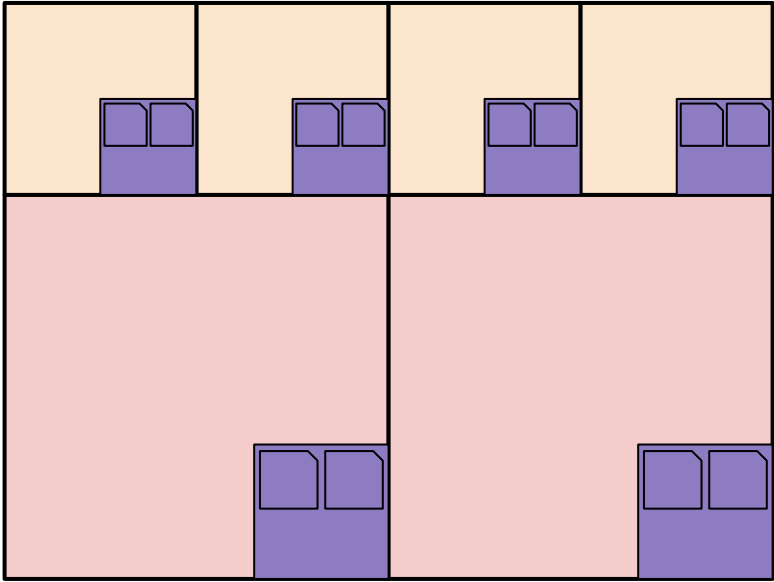


DRAM

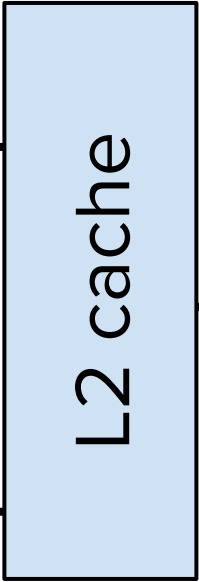


AMCC

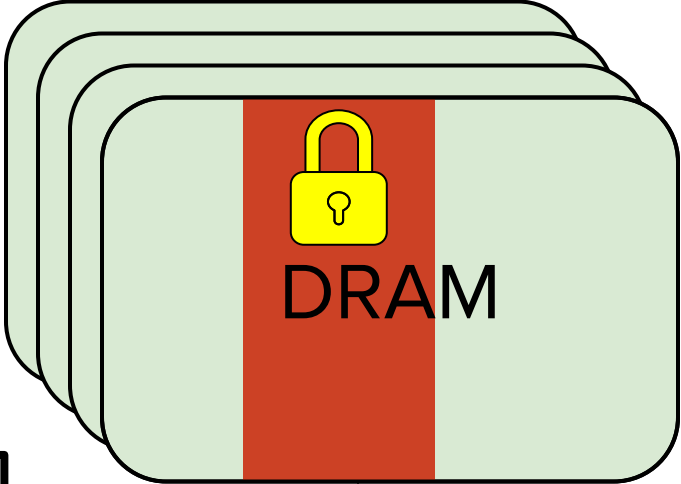
CPU cores



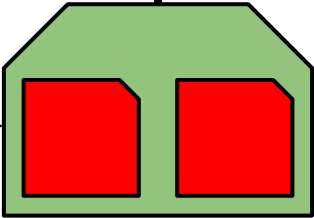
MMUs



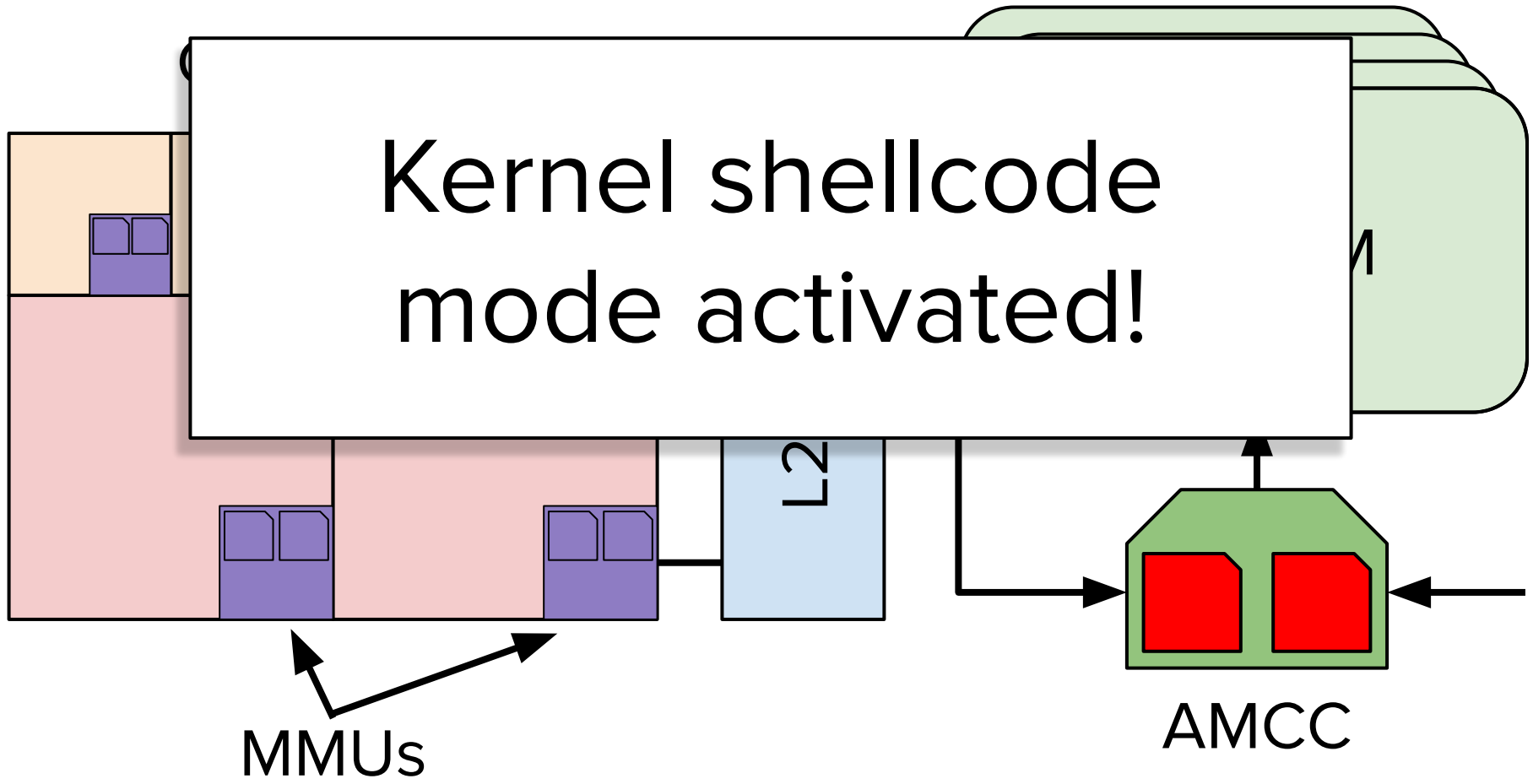
L2 cache



DRAM



AMCC



Kernel shellcode
mode activated!

MMUs

L2

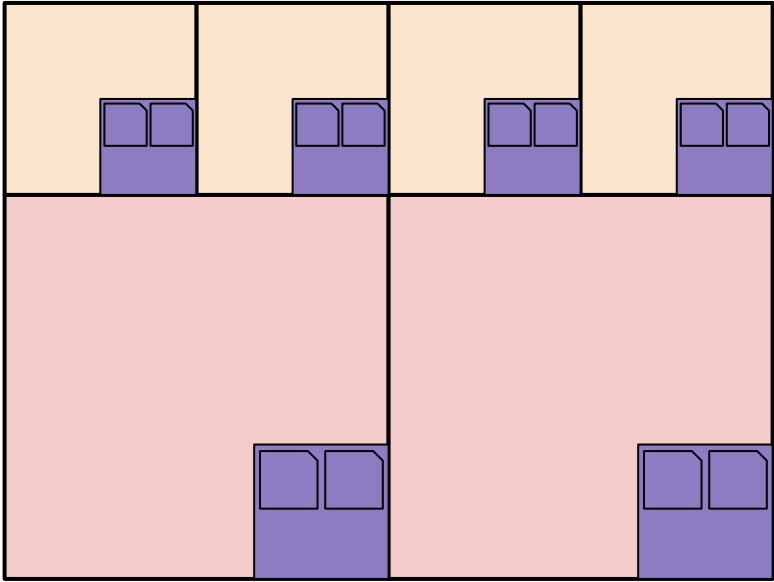
AMCC

Building KTRW

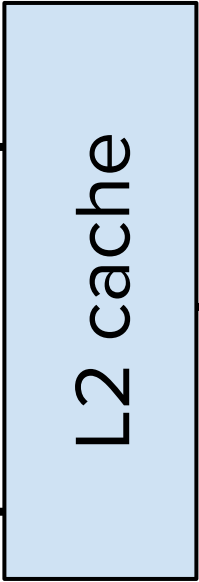
Steps to building a kernel debugger

- Remapping the kernel
- Loading a kernel extension
- Interrupt handling
- Communication channel
- GDB stub

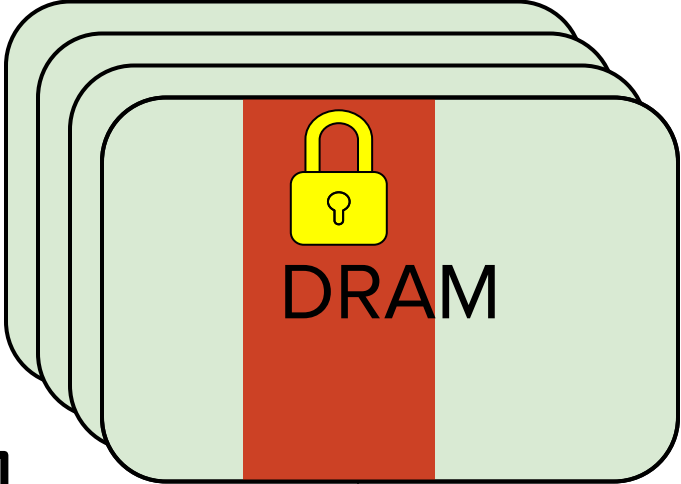
CPU cores



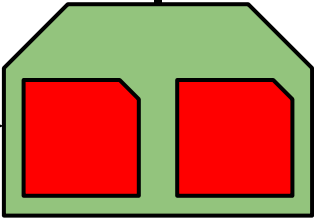
MMUs



L2 cache

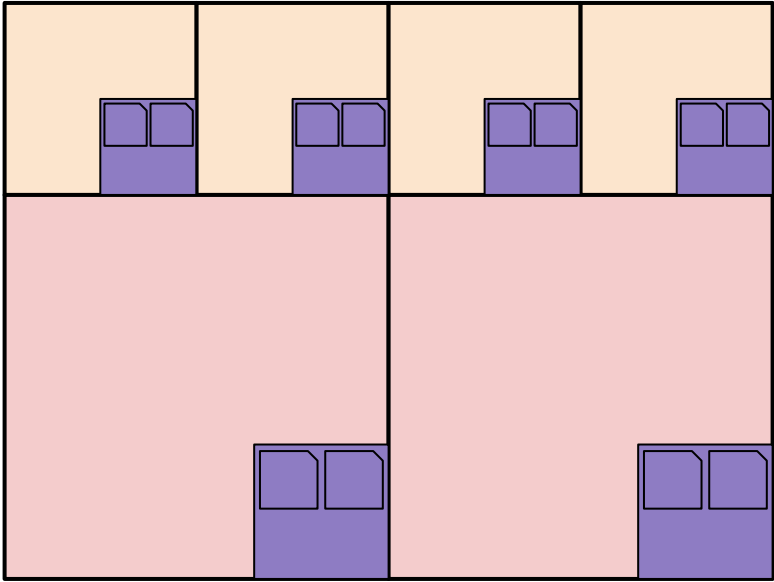


DRAM

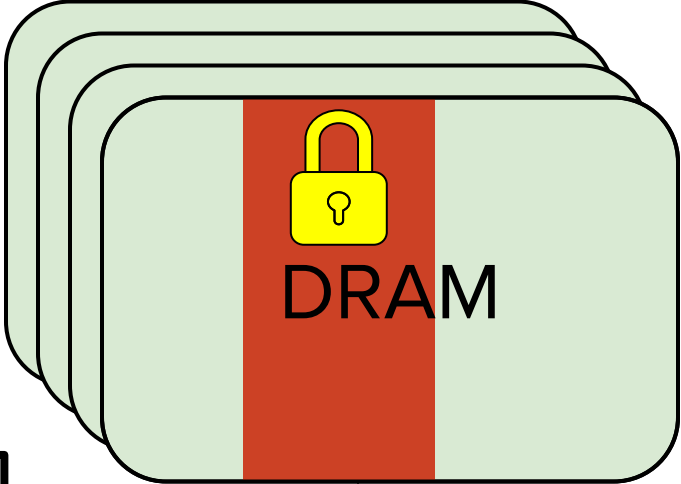
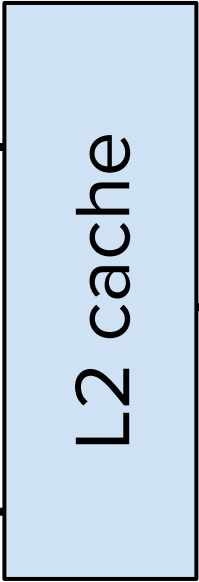


AMCC

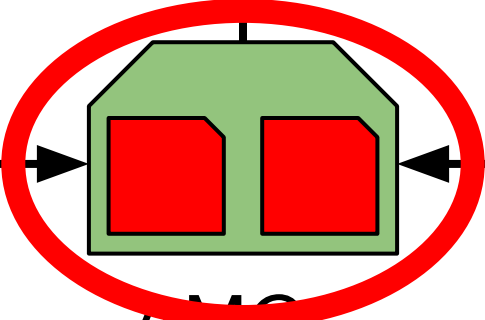
CPU cores



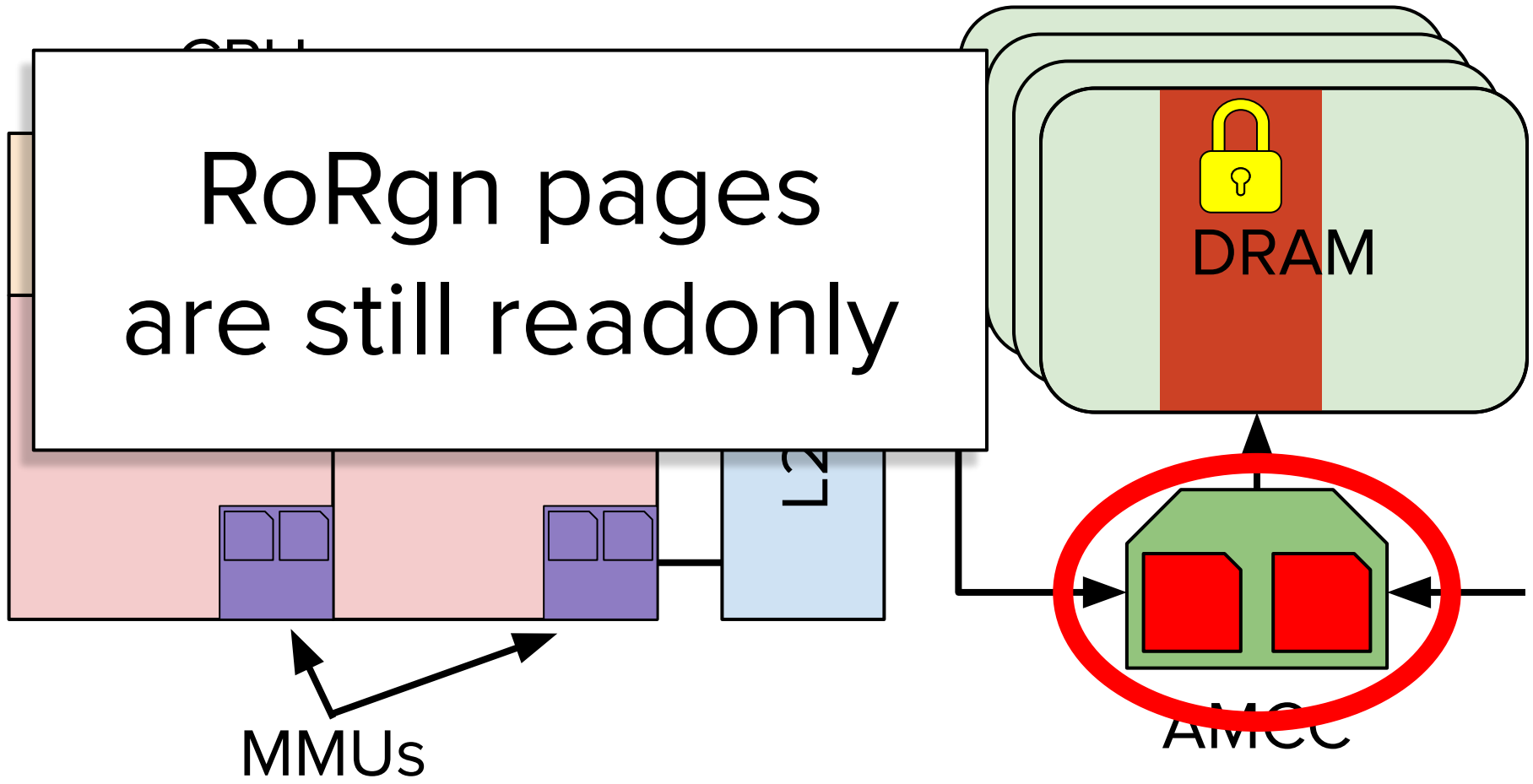
MMUs



DRAM



AMCC

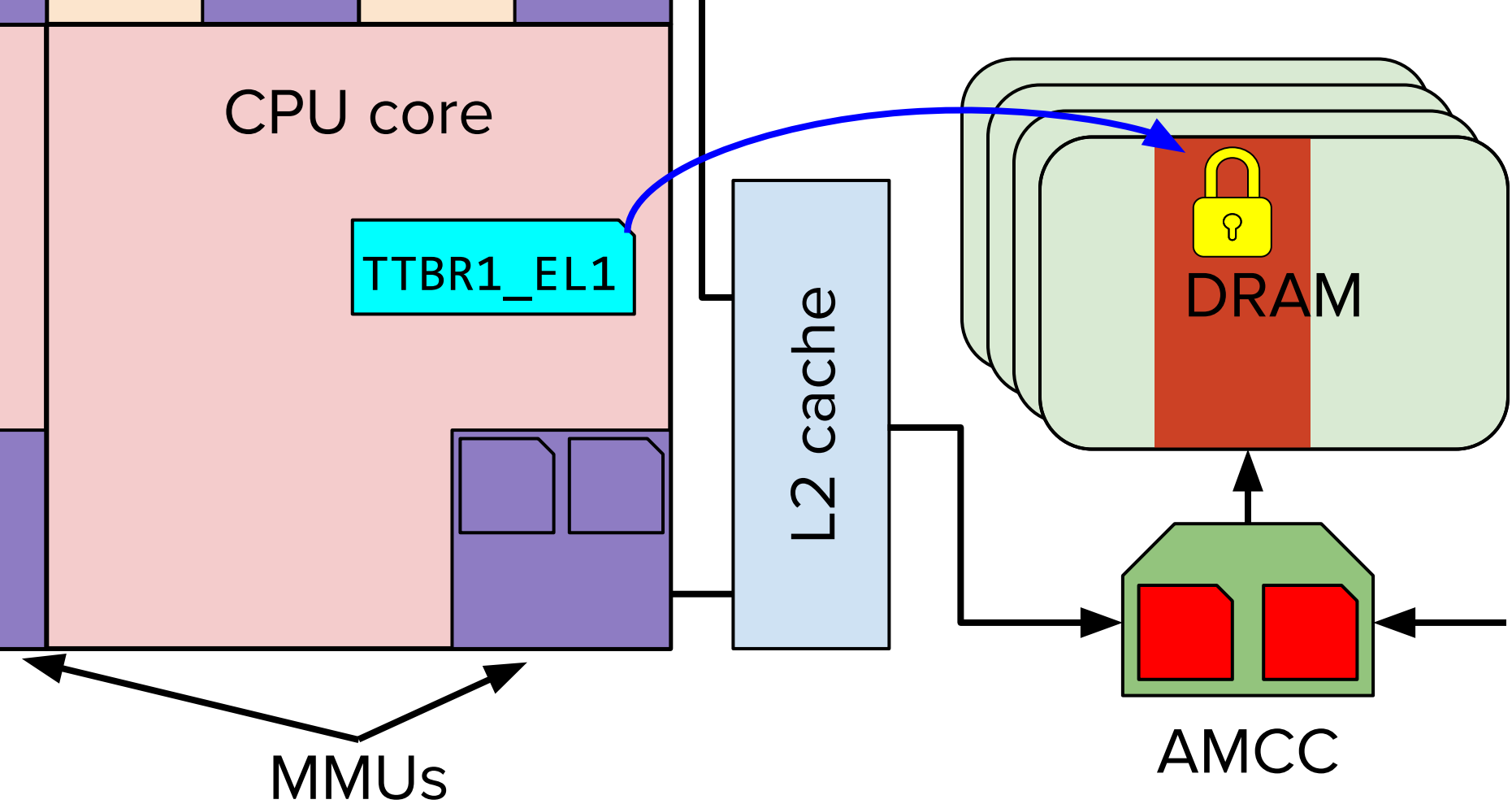


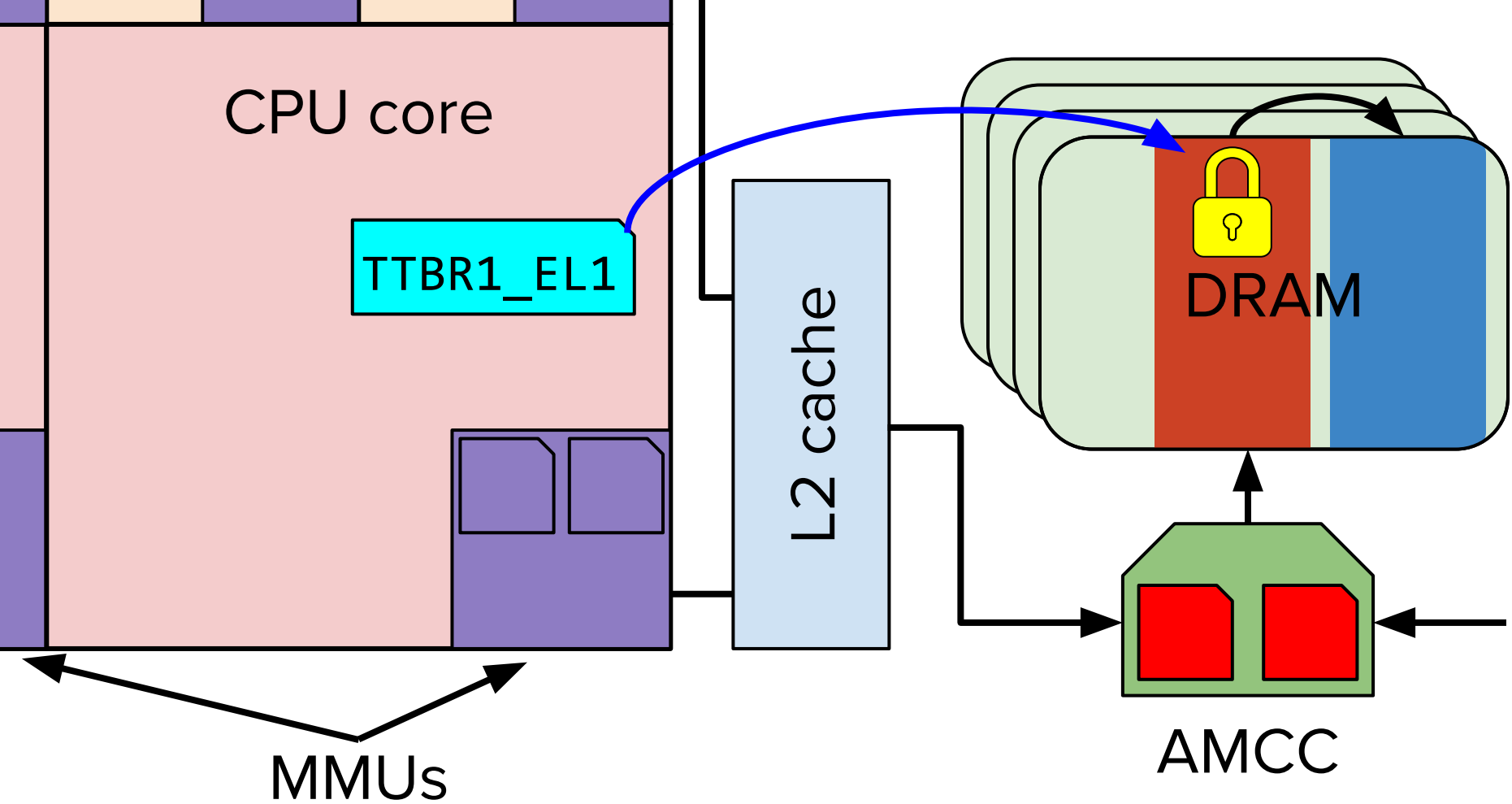
Remapping the kernel

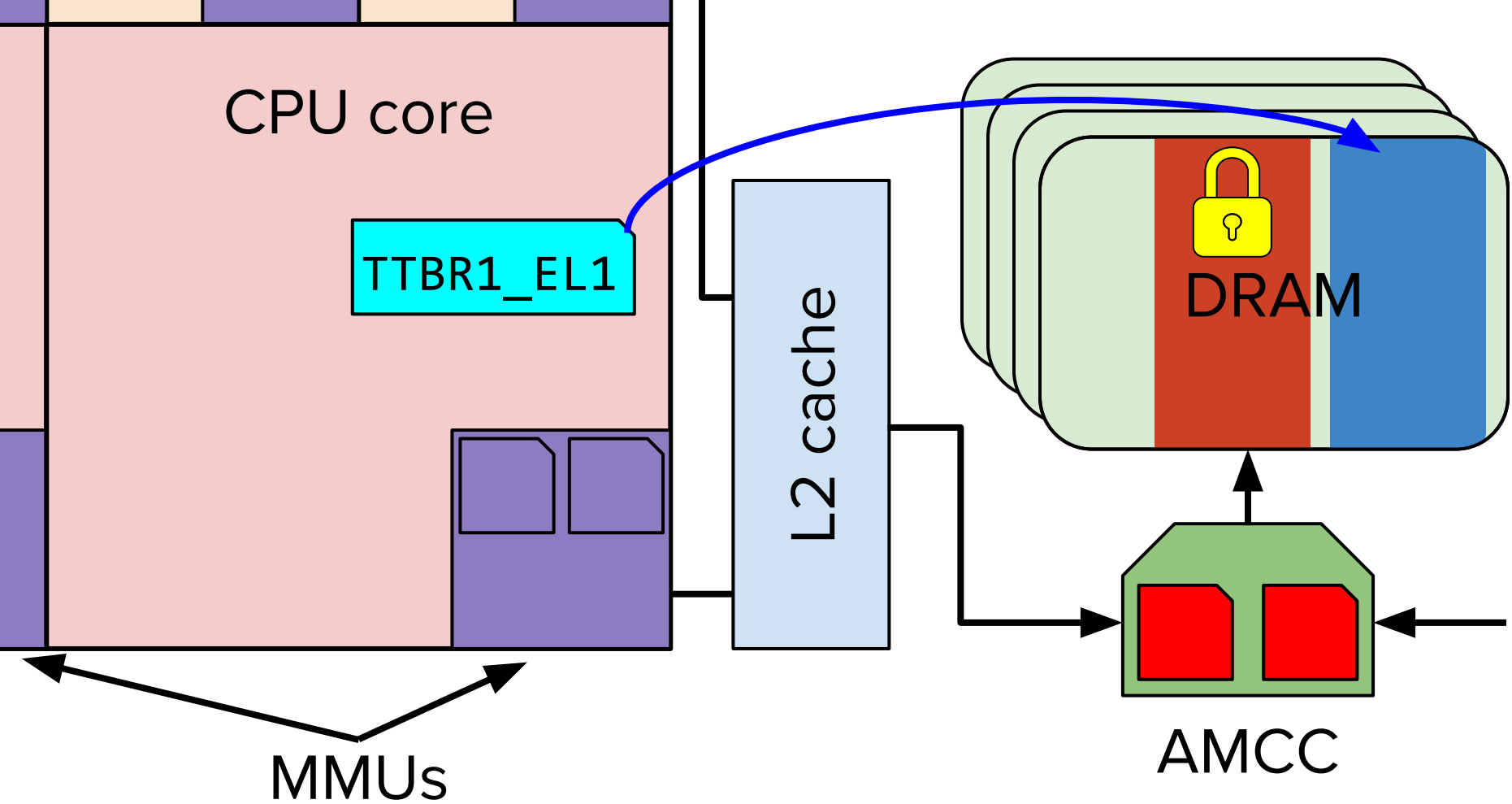
We need to modify page table permissions to make the kernel's memory executable

Root page tables are in the KTRR region (still readonly)

Solution: Remap the kernel onto fresh, writable pages and set `TTBR1_EL1` to point to the new page tables







Loading kernel extensions

Allocate kernel memory for the kext

Copy in the kext binary

Dynamically link against kernel symbols

Modify page tables to make it executable

Call the `kext_start()` function

KTRW: High-level design

One (monitor) core reserved for KTRW

Other (debugged) cores run XNU normally

Breakpoints/watchpoints cause the debugged core to halt and enter Debug state

Monitor core polls for entry to Debug state and notifies LLDB over some communication channel

Need some channel for
LLDB to communicate with
the KTRW kext

Communication channels

Serial	USB	WiFi

Communication channels

Serial	USB	WiFi
Easy	Fast	An option

Communication channels

Serial	USB	WiFi
Easy	Fast	An option
Requires <i>special</i> hardware	Write a custom USB stack	Write a custom WiFi driver

Communication channels

Serial	USB	WiFi
Easy	Fast	An option
Requires <i>special</i> hardware	Write a custom USB stack	Write a custom WiFi driver

DesignWare Hi-Speed USB 2.0 On-the-Go Controller

The DesignWare® Hi-Speed USB 2.0 On-The-Go (HS OTG) Controller provides designers with high-quality USB IP for the most demanding USB 2.0 peripherals. The controller performs as a standard Hi-Speed Dual-Role Device (DRD), operating as either a USB 2.0 Hi-Speed peripheral, or Hi-Speed USB 2.0 Host. Based on Synopsys' success in building and deploying Hi-Speed USB 2.0 Host, Device and PHY designs, the DesignWare USB 2.0 HS OTG Controller incorporates Synopsys expertise in Reuse Methodology, Constrained Random Verification, and USB PHY interoperability to deliver flexible, quality IP in Verilog source. The controller is optimized for area- and power-sensitive markets such as Internet of Things (IoT).

 [DesignWare IP Prototyping Kit for USB 2.0 HS OTG](#)

 [DesignWare IP Prototyping Kits](#)

SolvNetPlus

Sign In

 Please enter a username

[Need help signing in?](#)

[REGISTER - CREATE ACCOUNT](#)

[FORGOT PASSWORD](#)



raspberrypi / linux

Watch 721

Star 6.3k

Fork 3.1k

Code

Issues 243

Pull requests 28

Projects 0

Wiki

Security

Insights

Tree: c0e4ca1745

linux / drivers / usb / host / dwc_otg / dwc_otg_regs.h

Find file

Copy path



popcornmix Add dwc_otg driver

132ff67 on May 13

1 contributor

2550 lines (2376 sloc) 70.3 KB

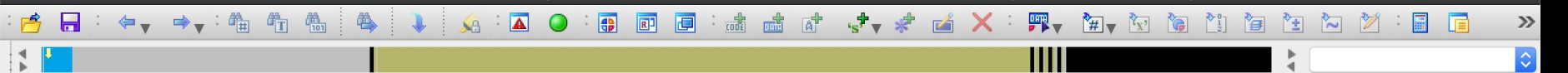
Raw

Blame

History



```
1173 * This union represents the bit fields in the Device Control
1174 * Register. Read the register into the <i>d32</i> member then
1175 * set/clear the bits using the <i>b</i> elements.
1176 */
1177 typedef union dctl_data {
1178     /** raw register data */
1179     uint32_t d32;
1180     /** register bits */
1181     struct {
1182         /** Remote Wakeup */
1183         unsigned rmtwkupsig:1;
1184         /** Soft Disconnect */
1185         unsigned sftdiscon:1;
1186         /** Global Non-Periodic IN NAK Status */
1187         unsigned gnpinnaksts:1;
1188         /** Global OUT NAK Status */
1189         unsigned goutnaksts:1;
```



Library function Regular function Instruction Data Unexplored External symbol Lumina function

Functions window IDA Vie... Pseudocod... Structu... Local Ty... Hex Vie... Strings win... Program Segmentat...

- Function name
 - syn_otg_stall_ep
 - syn_otg_reset_data_0
 - syn_otg_is_ep_stalled
 - syn_otg_do_xxx_
 - syn_otg_abort_ep
 - syn_otg_deactivate_ep
 - init_usb_task
 - usb_task_func
 - syn_otg_do_initial_reset
 - syn_otg_begin
 - retire_ioreq_chain
 - find_next_tx_fifo
 - begin_rx_dma
 - sub_1000052F8
 - syn_otg_int_handler**
 - send_current_ioreq_on_ep
 - add_just_finished_ioreq_to_finish
 - do_ep_in_timeout
 - recv_current_ioreq_on_ep
 - sub_100005AA8
- Line 140 of 583

```

50 out_daint = (unsigned int)daint >> 16;
51 in_daint = (unsigned __int16)daint;
52 for ( diepint_reg_off = 0x908LL; ; diepint_reg_off += 0x20LL )
53 {
54     if ( !(in_daint | out_daint) || ep_num > 5 )
55         goto DONE;
56     if ( in_daint & 1 )
57     {
58         diepint = *( _DWORD * )(diepint_reg_off | 0x23010000LL); // DIEPINT
59         syn_otg.endpoints[ep_idx_in].depint |= diepint;
60         *( _DWORD * )(diepint_reg_off | 0x23010000LL) = diepint;
61         if ( syn_otg.endpoints[ep_idx_in].is_active )
62         {
63             if ( diepint & 1 ) // xfercompl -- when is this signalled? When all packets have been
64                 //
65                 // Well I'm dealing with control transfers, so it's on each transacti
66             {
67                 endpoint_off = (int * __shifted(syn_otg_endpoint, -0x10))(&syn_otg.pending_setup_pkt
68                                     + ep_idx_in * 80);
69                 cur_ioreq = syn_otg.endpoints[ep_idx_in].current_io_req;
70                 if ( cur_ioreq )
71                 {
72                     cur_ioreq->transferred_size += ADJ(endpoint_off)[1].current_transfer_size;
73                     *( _QWORD * )&ADJ(endpoint_off)[1].current_transfer_size = 0LL;
74                     cur_ioreq = ADJ(endpoint_off)[1].current_io_req;
75                     if ( cur_ioreq->transferred_size >= cur_ioreq->size )
76                     {
77                         cur_ioreq->state = 0;
78                         cur_ioreq = ADJ(endpoint_off)[1].current_io_req;
79                         endpoint_off = (int * __shifted(syn_otg_endpoint, -0x10))(&syn_otg.pending_setup_pl

```

UNKNOWN syn_otg_int_handler:53 (1000053A4)

Output window

In general, the decompiler checks the segment permissions, class, and name to determine if it is read-only.
 -> OK
 1000052FC: restored microcode from idb

Python

USB 3.1 Bus

▼ USB 3.1 Bus

KTRW

KTRW:

Product ID:	0x1337
Vendor ID:	0x05ac (Apple Inc.)
Version:	0.00
Serial Number:	Google Project Zero
Speed:	Up to 480 Mb/s
Manufacturer:	Brandon Azad
Location ID:	0x14200000 / 3
Current Available (mA):	500
Current Required (mA):	500
Extra Operating Current (mA):	0

```
.. (up a dir)
</Developer/exploits/ktrw/
▼ ktrw_gdb_stub/
  ▼ source/
    ▼ gdb_stub/
      gdb_cpu.c
      gdb_cpu.h
      gdb_internal.c
      gdb_internal.h
      gdb_packets.c
      gdb_packets.h
      gdb_platform.c
      gdb_platform.h
      gdb_rsp.c
      gdb_rsp.h
      gdb_state.h
      gdb_stub.c
      gdb_stub.h
    ► third_party/
```

```
799
800 // ---- m packet -----
801
802 static sends_a_packet
803 gdb_pkt__m(struct packet *pkt) {
804     uint64_t address;
805     uint64_t length;
806     bool ok = pkt_read_hex_u64(pkt, &address)
807         && pkt_read_match(pkt, ",")
808         && pkt_read_hex_u64(pkt, &length)
809         && pkt_empty(pkt);
810     if (!ok) {
811         return send_error_bad_packet("m");
812     }
813     uint8_t data[GDB_RSP_MAX_PACKET_SIZE / 2];
814     if (length > sizeof(data)) {
815         return send_error_invalid_length("m");
816     }
817     size_t read = gdb_stub_read_memory(gdb.current_cpu, ad
818     if (read == 0 && length > 0) {
```

```
</bazad/Developer/exploits/ktrw 1 ktrw_gdb_stub/source/gdb_stub/gdb_packets.c [c,utf-8,unix] 0>
```


oob_timestamp

DEMO

<https://github.com/googleprojectzero/ktrw>

<https://googleprojectzero.blogspot.com/2019/10/ktrw-journey-to-build-debuggable-iphone.html>