1.3. MINOS KERNEL

System Startup

RPI (2) VC / firmware

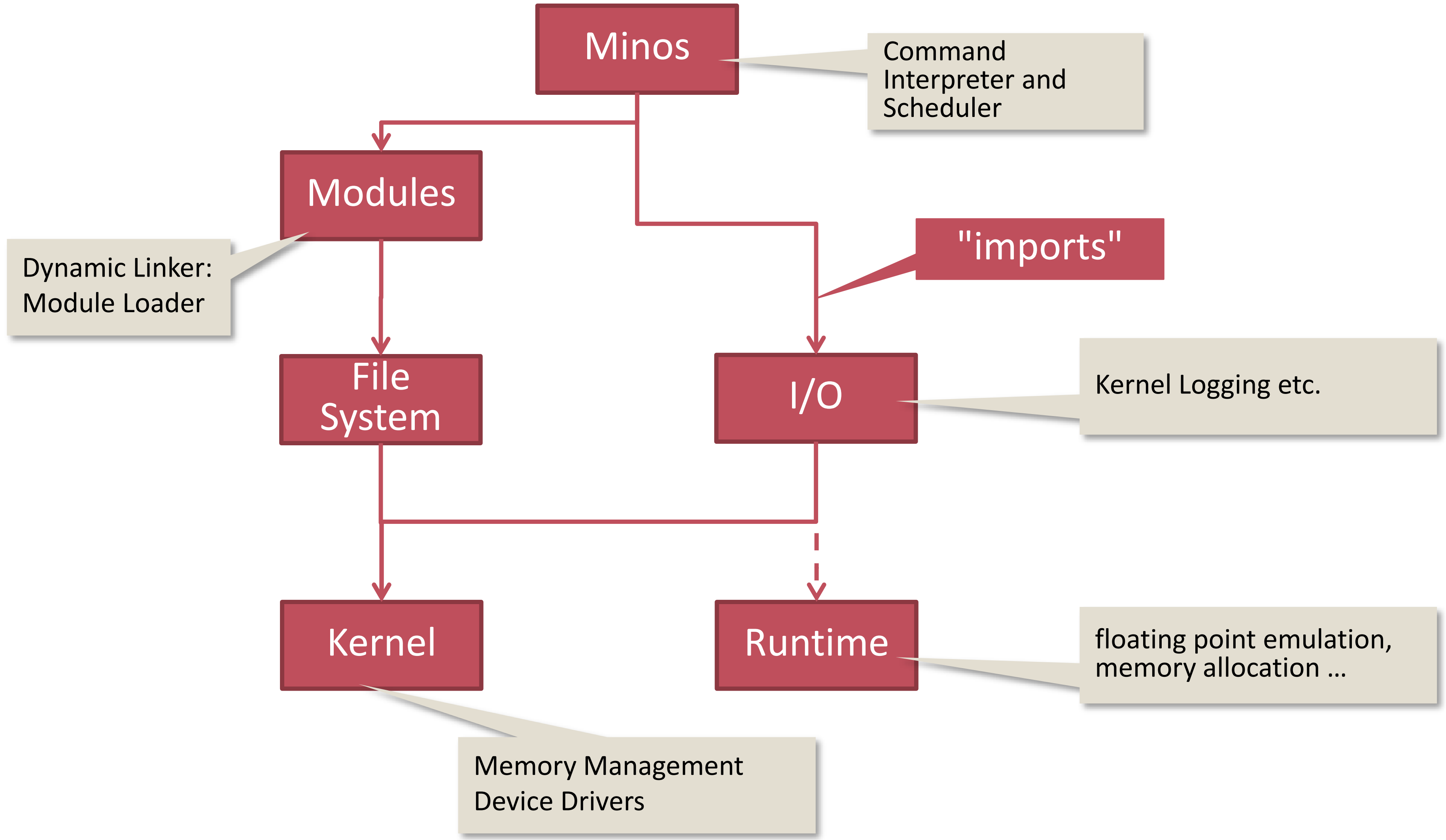
- Initialize hardware
- Copy boot image to RAM
- Jump to OS boot image (Initializer)

OS Initializer (we!)

- Set stack registers for all processor modes
- Setup free heap list and module list
- Initialize MMU & page table
- Setup interrupt handlers & runtime vectors
- Start timer & enable interrupts
- Initialize other runtime data structures
- Initialize UARTs
- Initialize RAM disk
- Enter scheduling loop on OS

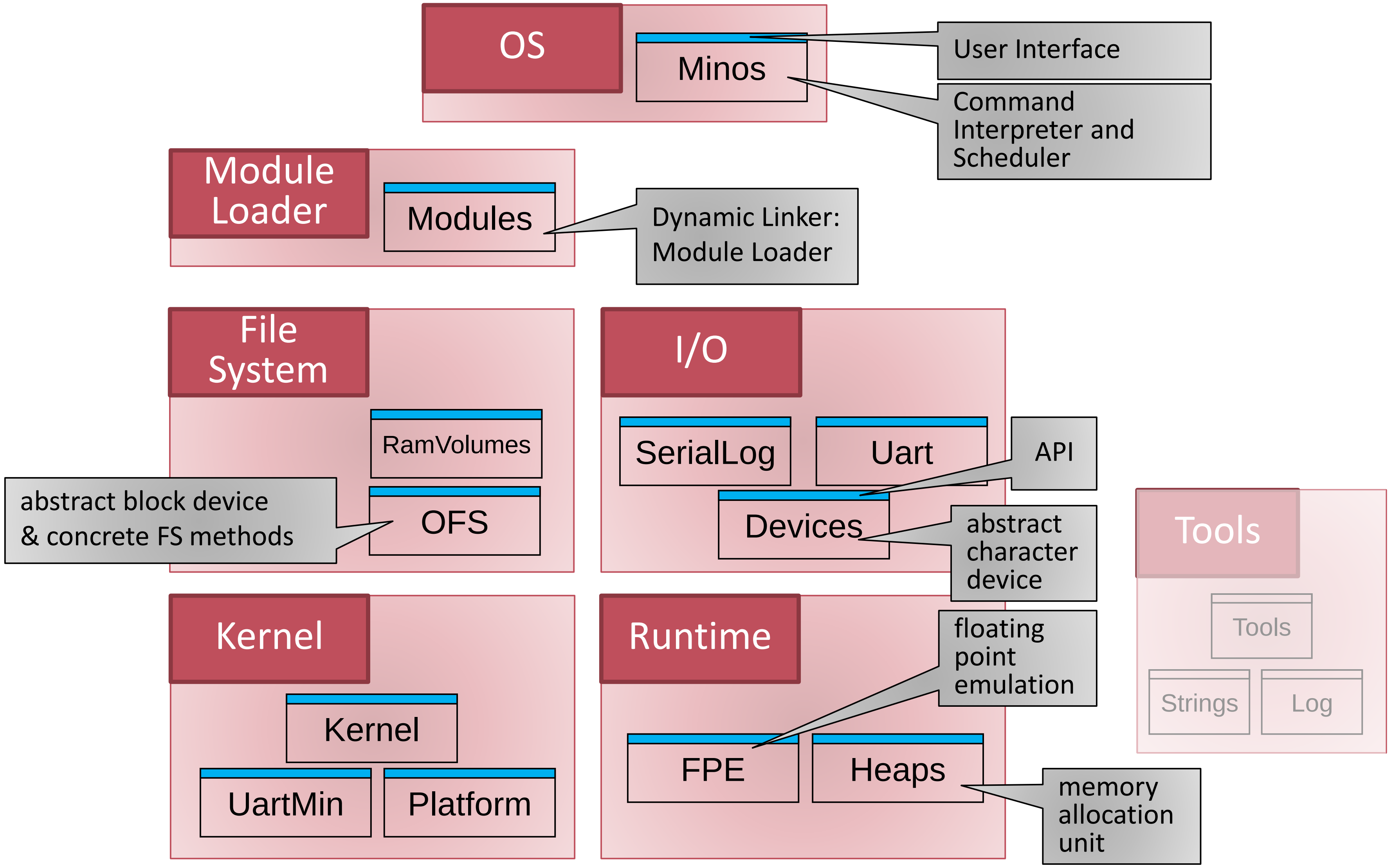
Modular Kernel Structure

The Big Picture


Minos
Command Interpreter and Scheduler
Modules
Dynamic Linker: Module Loader
"imports"
File System
I/O
Kernel Logging etc.
Kernel
Runtime
floating point emulation, memory allocation ...
Memory Management
Device Drivers

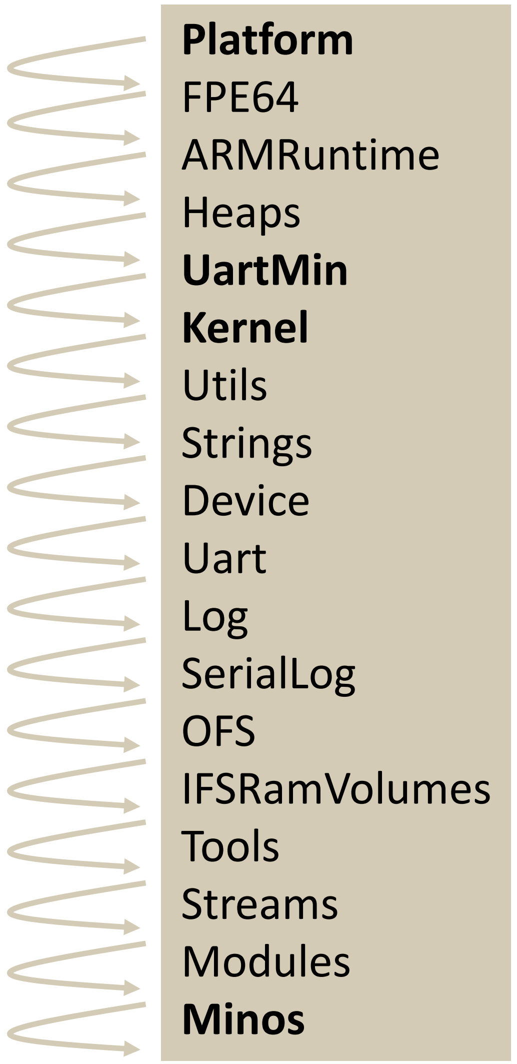
Modular Kernel Structure

Minos Modules in More Detail



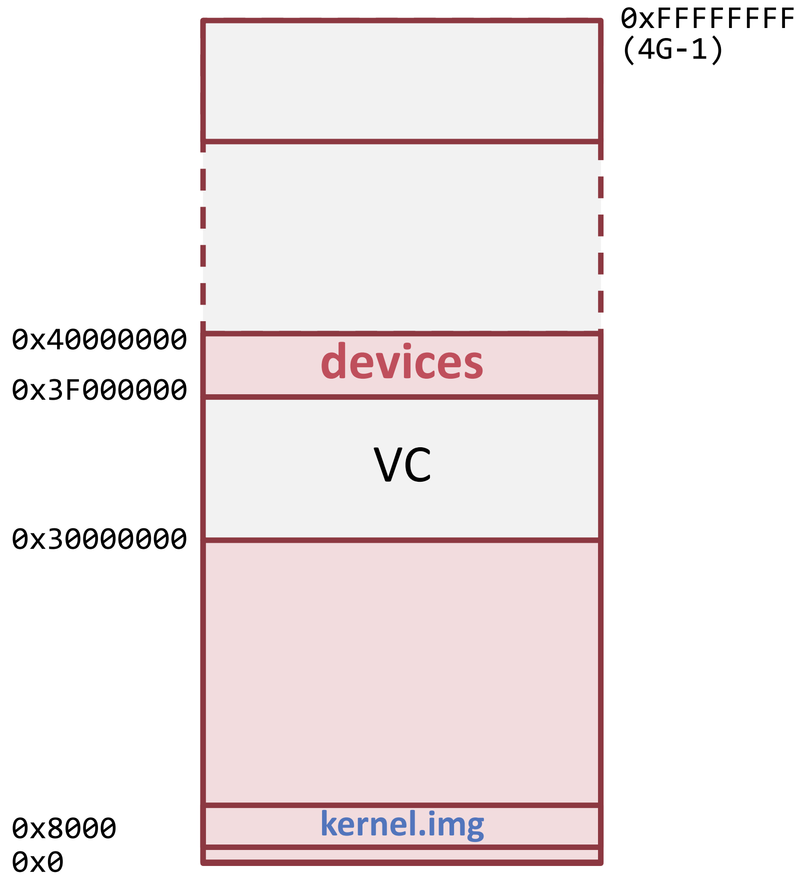
Kernel Module

```
MODULE Kernel;
  IMPORT SYSTEM, Platform;
  TYPE …
    (* types of runtime data structure *)
  VAR …
    (* global runtime data structures *)
  PROCEDURE P* (…); (* exported *)
  BEGIN …
    (* low level routine *)
  END …;
  PROCEDURE … (…); (* internal *)
    (* low level routine *)
  BEGIN …
  END …;
BEGIN …
  (* runtime initialization *)
END Kernel.
```

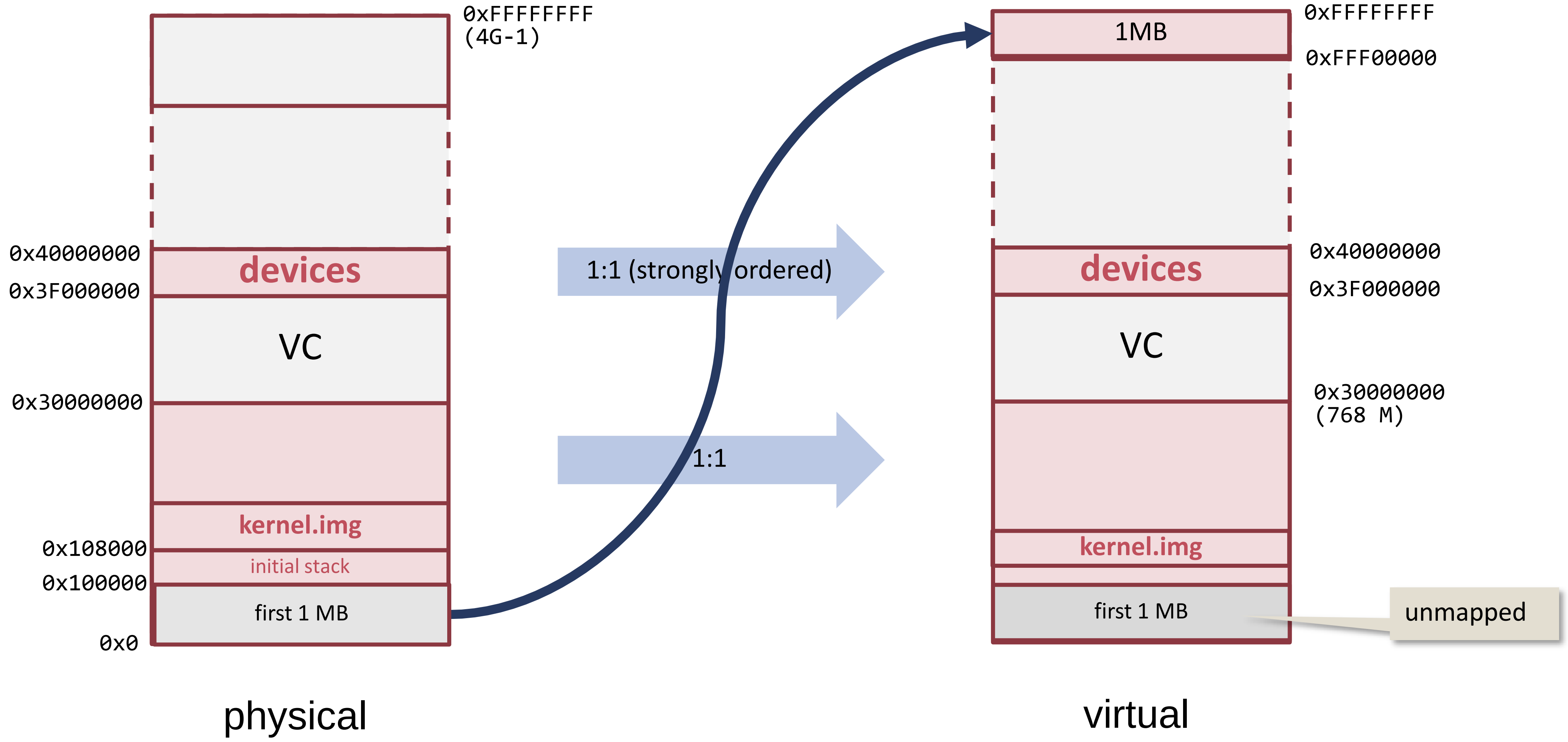


Memory Layout -- Objectives

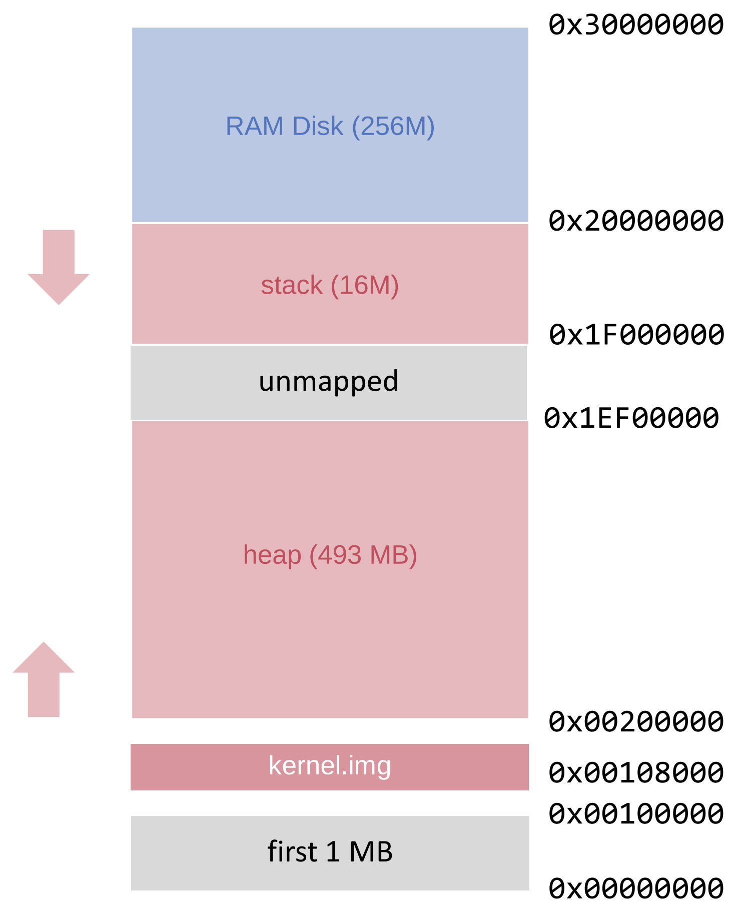
- As simple as possible
- Support null-pointer checks via MMU
- Classical Heap / Stack layout
- 1 MB pages



Memory Layout: big picture

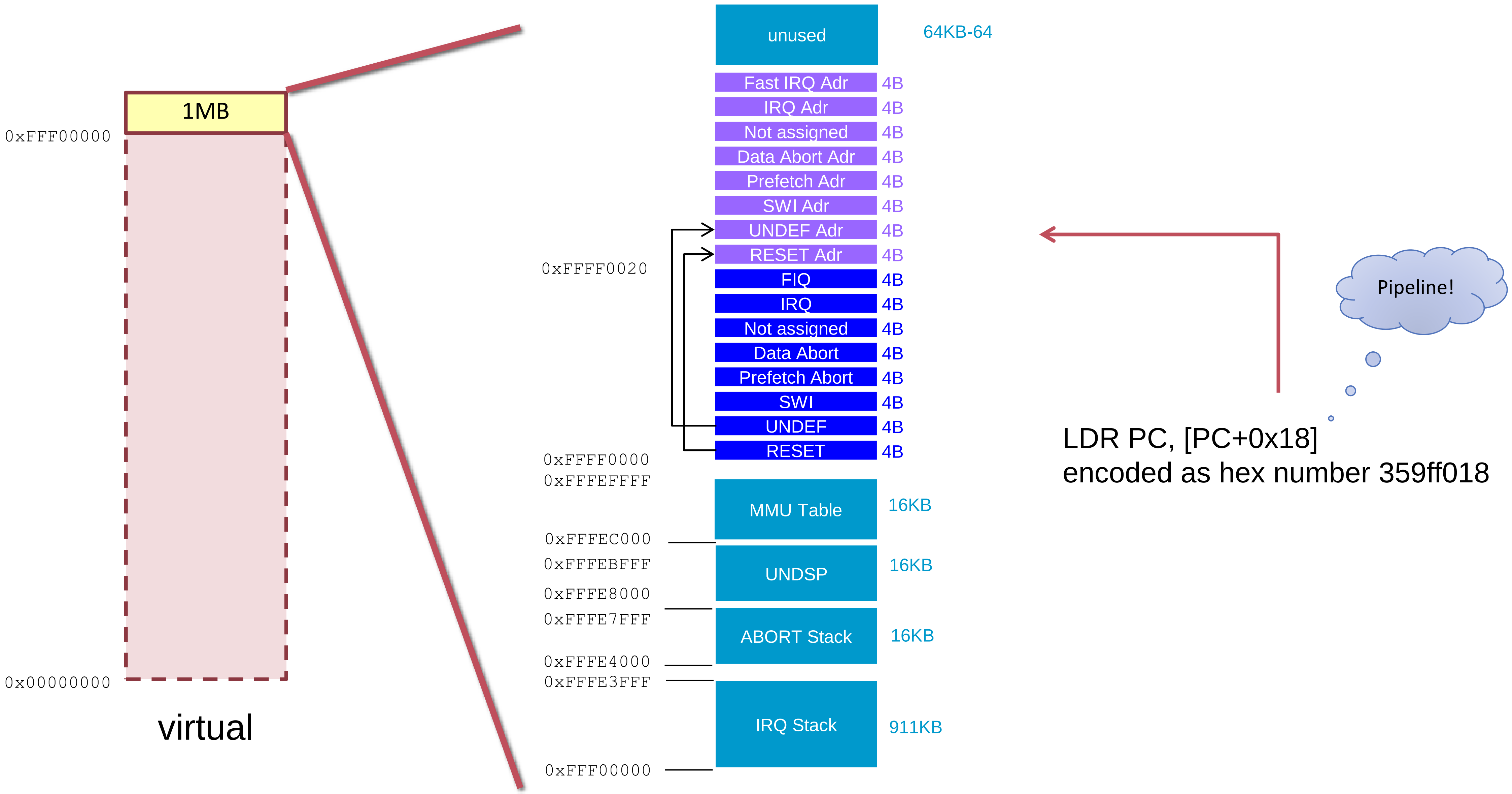


Virtual Memory Layout: Heap, Stack, RAMDisk



virtual

Virtual Memory Layout: IRQ Table / MMU



LDR PC, [PC+0x18]
encoded as hex number 359ff018

Initialization: Kernel (body)

SYSTEM.LDPSR(u, src) [instruction MSR]

u = 0	PSR of current processor mode
u = 1	PSR of saved processor mode
src	value to be loaded in PSR, an expression

```
VAR lnk: PROCEDURE;
...
BEGIN  (* do not enter any call here --> link register consistency ! *)

   lnk := SYSTEM.PUT32(ADDRESSOF(lnk), SYSTEM.LNK());

   SYSTEM.LDPSR( 0, Platform.SVCMode + Platform.FIQDisabled + Platform.IRQDisabled );
   SYSTEM.SETSP(Platform.SVCSP);
   SYSTEM.SETFP(Platform.SVCSP);

   SYSTEM.LDPSR( 0, Platform.IRQMode + Platform.FIQDisabled + Platform.IRQDisabled );
   SYSTEM.SETSP(Platform.IRQSP);
```

store link register globally – we are switching the stack!

disable IRQs, stay in SVC mode

new stack top for this mode

Initialization

Kernel (body)

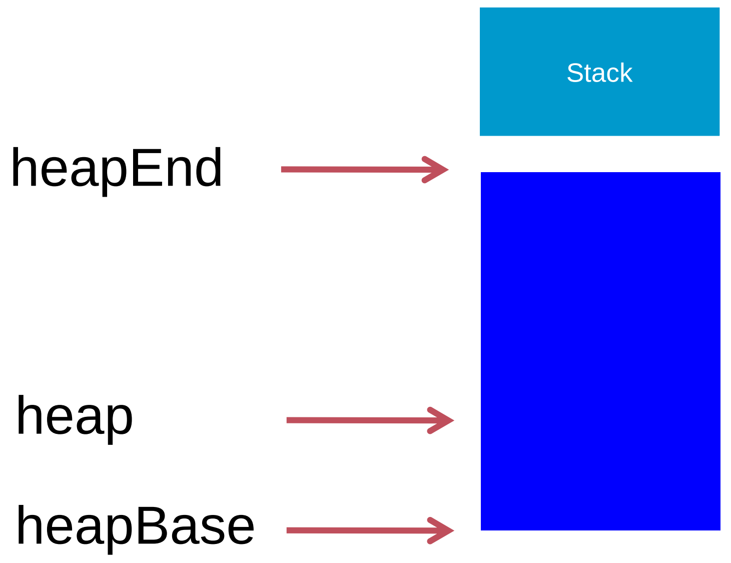
```
   SYSTEM.LDPSR( 0, Platform.FIQMode + Platform.FIQDisabled + Platform.IRQDisabled);
   SYSTEM.SETSP(Platform.FIQSP);

   ...

   SYSTEM.LDPSR( 0, Platform.SVCMode + Platform.FIQDisabled + Platform.IRQDisabled );

   InitMMU;
   SetupInterruptVectors;
   InitHandlers;
   EnableIRQs;
   OSTimer;
   lnk
END Kernel.
```

continue execution (next body)

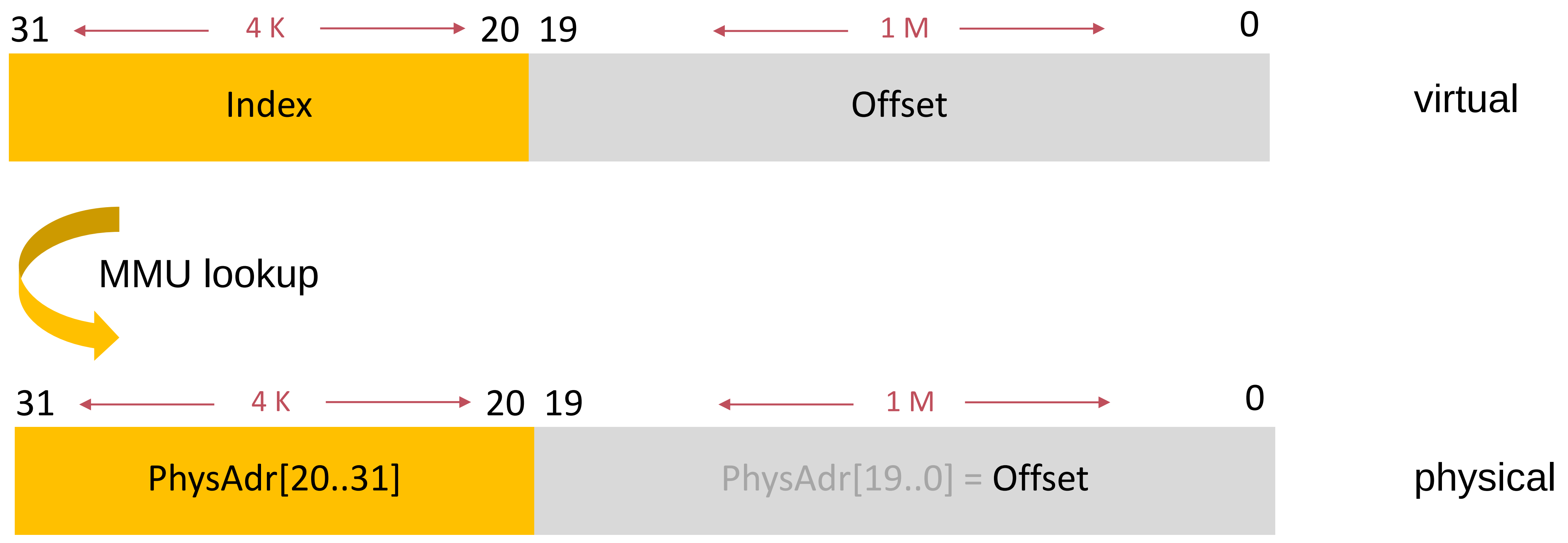
Initialization

Heap

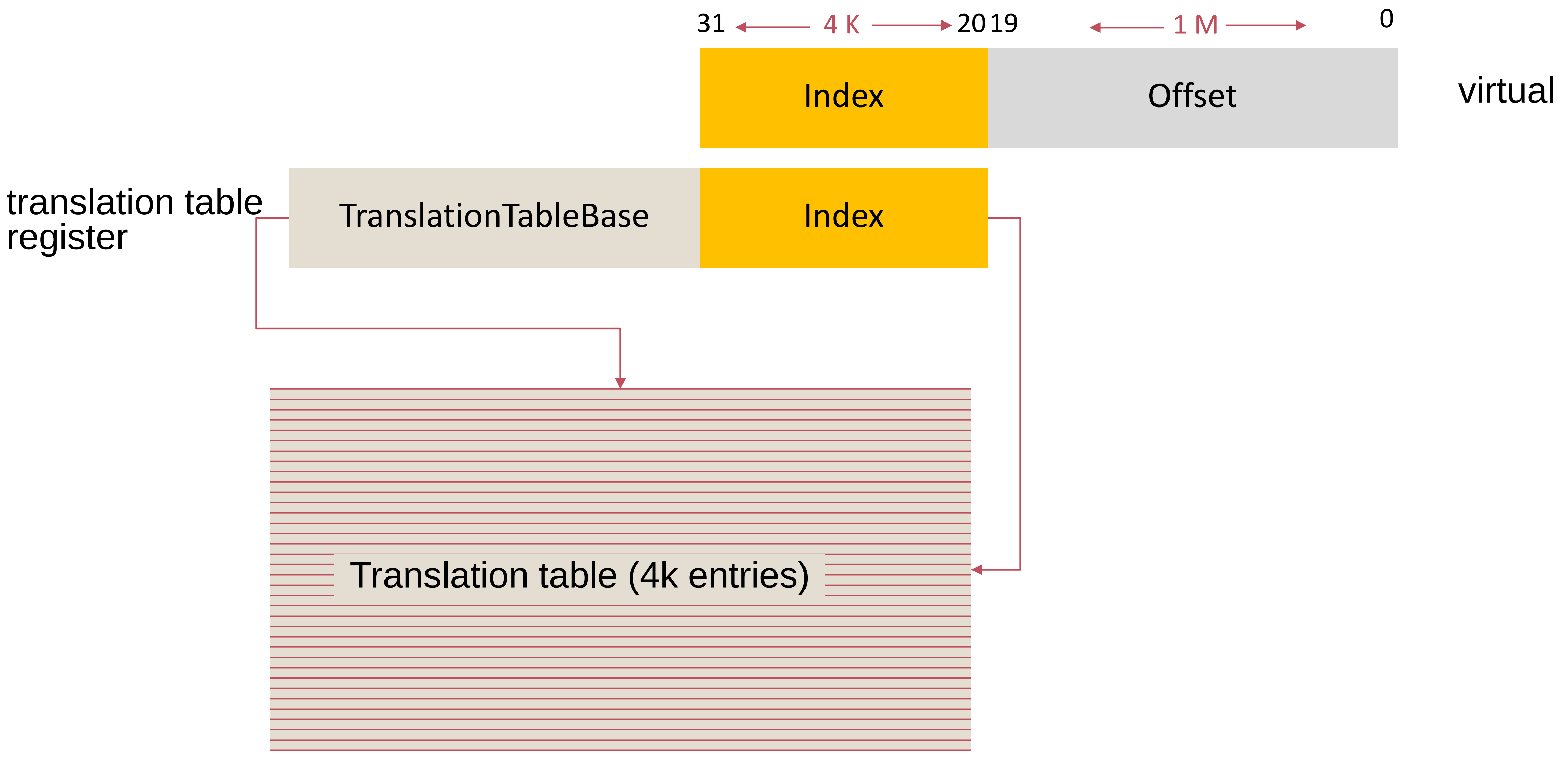
```
MODULE Heaps;

...

BEGIN
  heapStart := Platform.HeapBase;
  heap := Platform.HeapBase;
  heapEnd := Platform.HeapEnd;
END Heaps.
```



Address Translation (1 MB pages)



Translation Table



Page Table Entries

Possible entries:

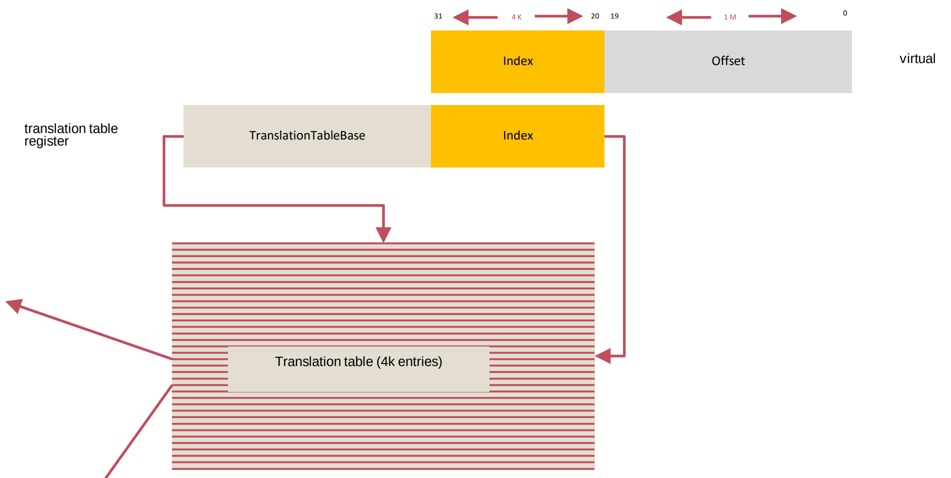
- Invalid

31 … 2	1 0
Ignored	0 0

- Section

31 … 20	19	18	17	16	15	14 … 12	11 10	9	8 … 5	4	3	2	1 0
Section base address		0		S	AP [2]	TEX [2:0]	AP [1:0]	SBZ	Domain	IMP	C	B	1 0

(Properties define memory type and sharing attributes)

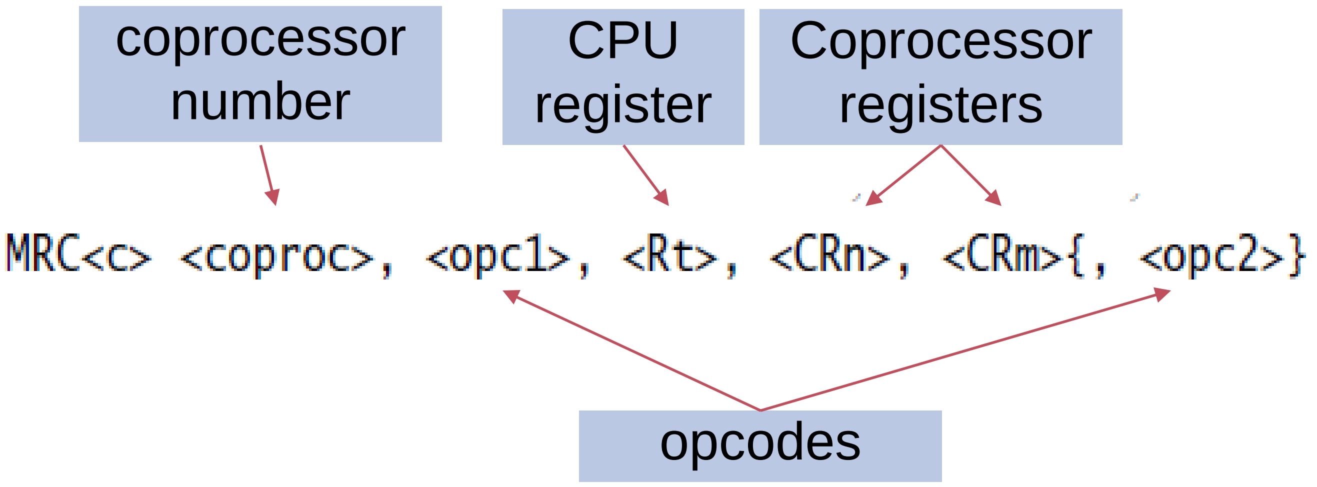


- Page table (2nd level: 4k or 64K pages), Supersection (16 MB pages)

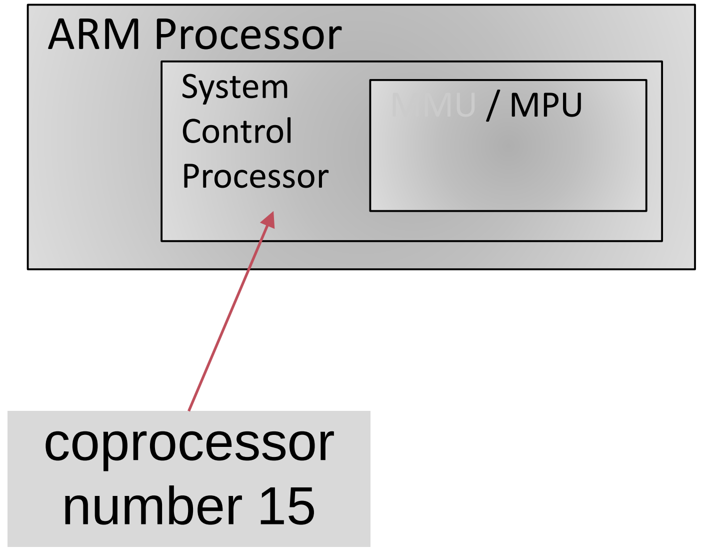
(cf. ARM Reference Manual v7-A Section B.3)

Accessing the System Control Processor

MCR (Move to Coprocessor from Arm Register)
MRC (Move to Arm Register from Coprocessor)



```
MRC<c> <coproc>, <opc1>, <Rt>, <CRn>, <CRm>{, <opc2>}
```

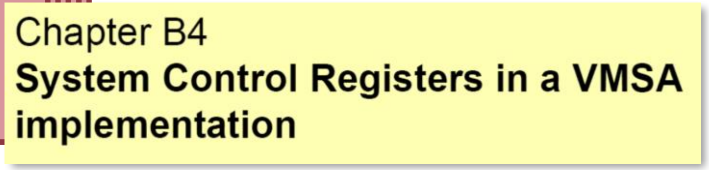


31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	1	0	opc1			1	CRn				Rt				coproc				opc2			1	CRm			

System Control Processor Registers

ARM Architecture Reference Manual ARM v7A

B4.1.154

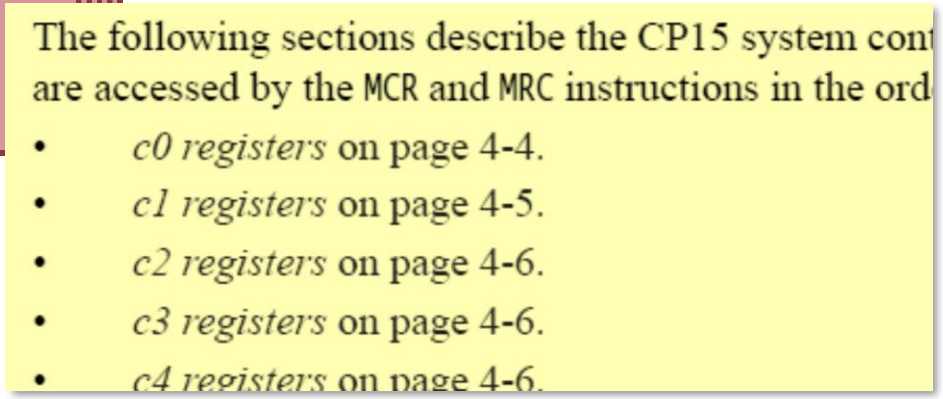


Chapter B4

System Control Registers in a VMSA implementation

Cortex A7 MPCore Technical Reference Manual

4.2.3.



The following sections describe the CP15 system cont
are accessed by the MCR and MRC instructions in the ord

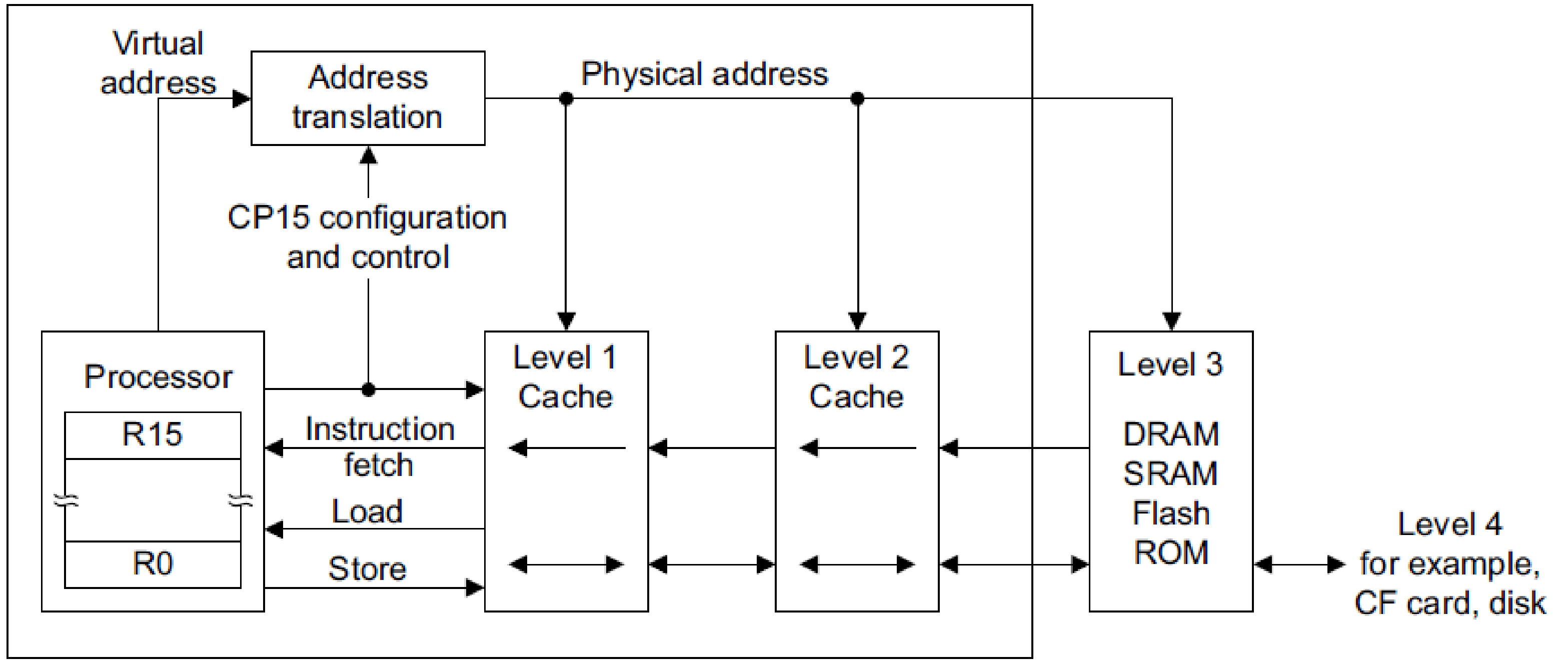
- *c0 registers* on page 4-4.
- *c1 registers* on page 4-5.
- *c2 registers* on page 4-6.
- *c3 registers* on page 4-6.
- *c4 registers* on page 4-6.

e.g.

"MCR P15, 0, R0, C2, C0, 0 ; set page table base address"

Cache

- ARM processors can support several levels of cache

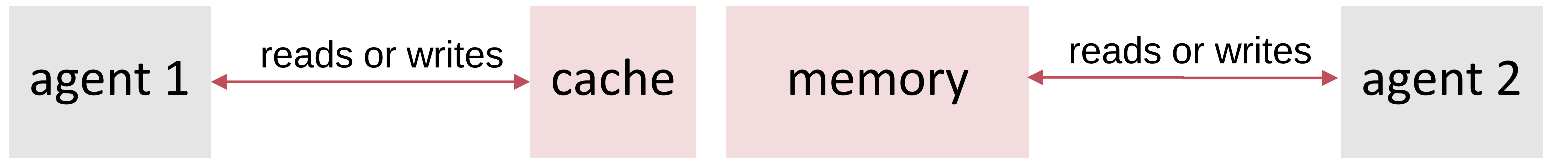


(from ARM Architecture Reference Manual ARM v7-A, Section A3.9.2

Cache Coherency

Caches largely invisible to the application programmer in normal operation.

Cache coherency breakdown:



Examples:

- agent 1 = processor, agent 2 = **DMA controller**
- agent 1 = processor (**instruction cache**), agent 2 = processor (**data cache**)
- agent 1 = processor x, agent 2 = processor y

relevant even for single-core systems

Ensuring Cache Coherency

- Not enabling caches in the system
  - RPI starts with MMU switched off and with caches disabled
- Use memory maintenance operations to manage cache coherency issues in software
  - E.g. when sharing information between processors
  - E.g. when changing instruction memory from data path
- Use hardware coherency mechanisms configurable for memory regions
  - E.g. strongly ordered memory for memory regions containing device registers

Memory types and attributes

Memory type attribute	Shareable attribute	Other attribute	Objective
Strongly-ordered	Shareable		Memory accesses to Strongly-ordered memory occur in program order.
Device	Shareable		Memory mapped peripherals that are shared by several processors.
	Non-Shareable		Memory mapped peripherals that are used only by a single processor.
Normal	Shareable	Non-cacheable Write-Through cacheable Write-Back cacheable	Normal memory that is shared between several processors.
	Non-Shareable	Non-cacheable Write-Through cacheable Write-Back cacheable	Normal memory that is used only by a single processor.

... we will revisit this topic again when discussing memory ordering in the context of a multi-core kernel.

Cache properties (ARM v7)

- Data memory cache: Reads and Writes from one observer to the same physical location, also from different virtual addresses, always happen in program order
  - No memory barriers required
- Instruction caches are never written to or read from by memory load / store operations
  - The guarantees from above do not necessarily apply to instruction cache, depending on the cache implementation
- Changing memory attributes in the page table can require a cache maintenance operation

Cache Coherency Issues

Memory Location Update by a Processor not visible to other observers because

1. new updates are still in the writing's processor cache
2. cache of observer contains stale copy of the memory

Two explicit mechanisms to address this

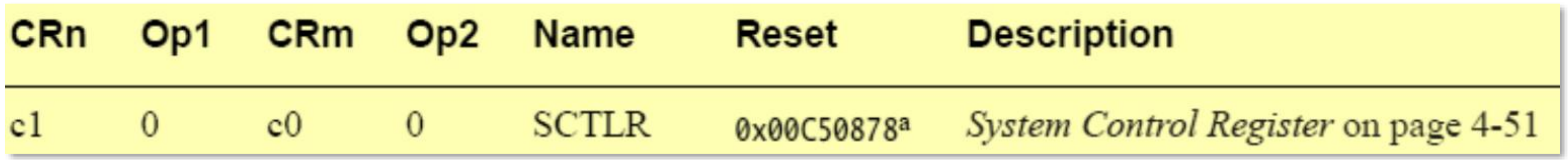
Clean: updates made by a writer made visible to other observers that can access memory at the point to which the operation is performed.

Invalidate: A cache invalidate operation ensures that updates made visible by writers that access memory are made visible to an observer that controls the cache.

[precise definitions in ARM Architecture Manual v7, B 2.2.6]

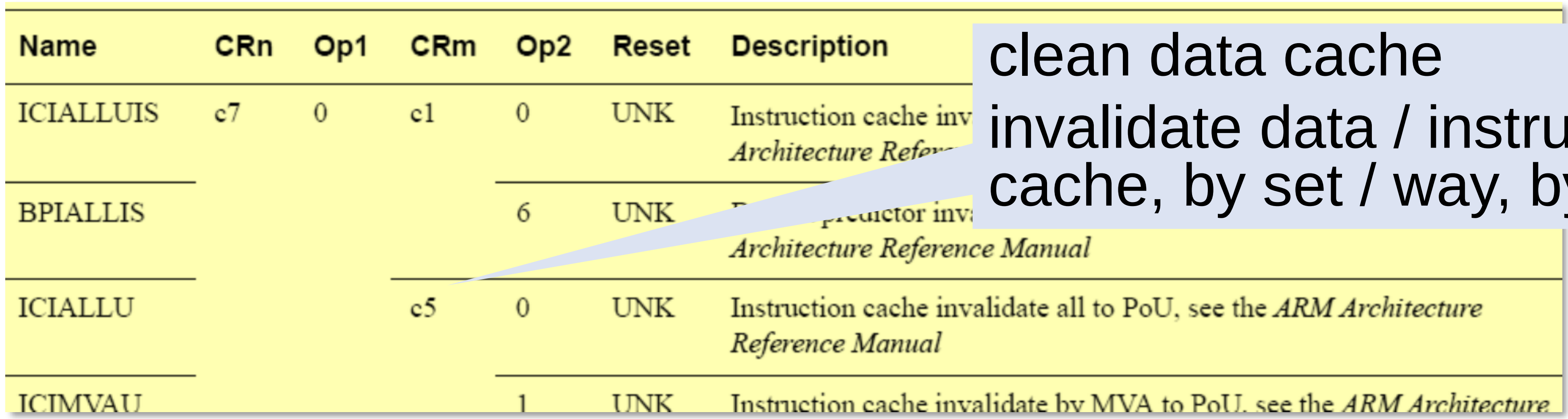
Cache Maintenance

- Instruction/Memory Cache can be selectively enabled / disabled



CRn	Op1	CRm	Op2	Name	Reset	Description
c1	0	c0	0	SCTLR	0x00C50878[a]	*System Control Register* on page 4-51

- Cache manipulation: CRn = 7, TLB mainpulation: CRn = 8



Name	CRn	Op1	CRm	Op2	Reset	Description
ICIALLUIS	c7	0	c1	0	UNK	Instruction cache inv *Architecture Refer*
BPIALLIS				6	UNK	predictor inv *Architecture Reference Manual*
ICIALLU			c5	0	UNK	Instruction cache invalidate all to PoU, see the *ARM Architecture Reference Manual*
ICIMVAU				1	UNK	Instruction cache invalidate by MVA to PoU, see the *ARM Architecture*

clean data cache

invalidate data / instruction cache, by set / way, by VA, ...

Barriers

- **ISB** -- Instruction Synchronization Barrier flushes the pipeline: all instructions following the ISB are fetched from cache or memory.
  - Important when code is written to data memory. Example: module loading.
  - Important when instruction memory changes, e.g. page table / TLB modifications
- **DMB** -- Data Memory Barrier: synchronizes memory accesses and provides memory ordering.
- **DSB** -- Data Synchronisation Barrier: data memory barrier that additionally synchronizes the execution stream with memory accesses.

to be revisited in the multicore context

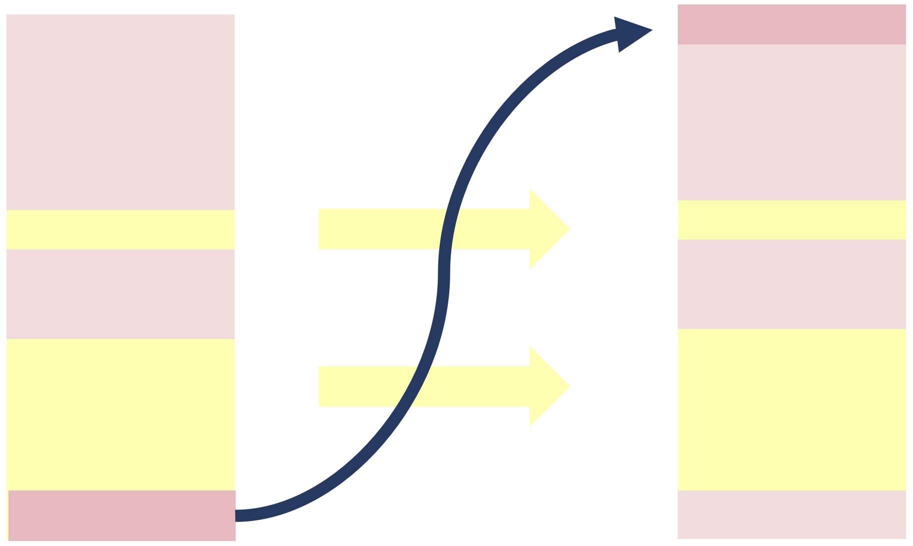
Initialization: Platform.IdentityMapMemory

```
VAR pageTable-: POINTER {UNSAFE} TO ARRAY 4096 OF ADDRESS;

PROCEDURE IdentityMapMemory-;
VAR index: SIZE;
BEGIN
   pageTable := MMUPhysicalTableBase;
   FOR index := 0 TO MemorySize DIV MB - 1 DO
      pageTable[index] := index * MB +  NormalMemory
   END;
   FOR index := MemorySize DIV MB TO LEN (pageTable) - 1 DO
      pageTable[index] := index * MB + StronglyOrderedMemory
   END;
END IdentityMapMemory;

BEGIN
   IdentityMapMemory;
   EnableMemoryManagementUnit;
END Platform.
```

Initialization: Kernel.InitMMU

```
(* Init the memory management unit *)
PROCEDURE InitMMU;
CONST
   Platform.DisableMemoryManagementUnit;
   Platform.pageTable[0] := 0; (* unmap page *)
   Platform.pageTable[4095] := 0*MB + StronglyOrderedMemory;
   Platform.EnableMemoryManagementUnit;
END InitMMU;
```

lengthy code because of
cache operations involved !



Enable Memory Management Unit

1. Set page table base address (register c2 c0)
2. Enable full access to domain 0 (register c3 c0)
3. set memory protection, data and unified cache, branch predicition, instruction cache and high vector bits in system control register
4. Flush And Invalidate DCache ← lengthy code
5. InvalidateTLB
6. InvalidateICache

Example (invalidate TLB)

```
MCR p15, 0, R0, c8, c7, 0    ; invalidate I+D TLB
DSB
ISB
```