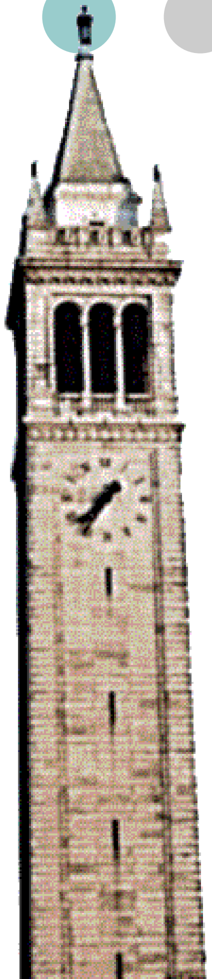


Introduction to Embedded Systems

Branden Ghena

UC Berkeley

EECS 149/249A

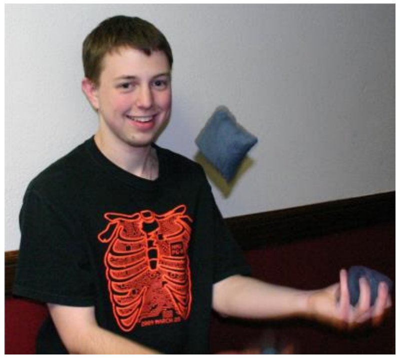
Fall 2019

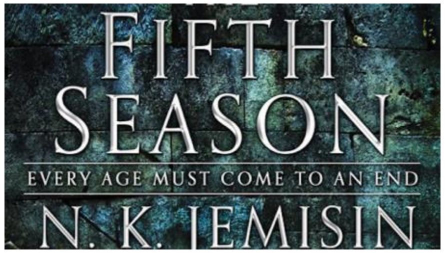
Content by Prabal Dutta, Edward Lee, and others

Chapter 9: Memory Architectures

Introducing Branden

- **Education**: 7th year PhD student 😱
  - Undergrad: Michigan Tech
  - Master's: University of Michigan
- **Teaching**:
  - Summer 2019: CS61C
  - Fall 2018: EE149/CS249A
  - On the job market for lecturer positions this fall!
- **Research**: Low-power wireless communications





Today's Topic: Memory Architectures

Computer Memory

- Physical Types
  - SRAM, Flash, Disks
- Hardware Architectures
  - Registers, Caches, Primary Memory
- Software
  - Stack, Heap, Code

Mostly Review (for some of you)

- With embedded systems twists
- For more details on normal non-embedded computers see CS61C
  - https://inst.eecs.berkeley.edu/~cs61c/su19/#lectures
  - Particularly "C Memory Management" and "Caches"

Outline

- Memory Overview
  - Types of Memory
  - Memory Hierarchy
- Embedded Systems Memory
  - Memory Maps
  - Memory-Mapped I/O
  - Lab Hardware Examples
- Memory Organization
  - Stacks & Heaps
  - Code examples

TYPES OF MEMORY

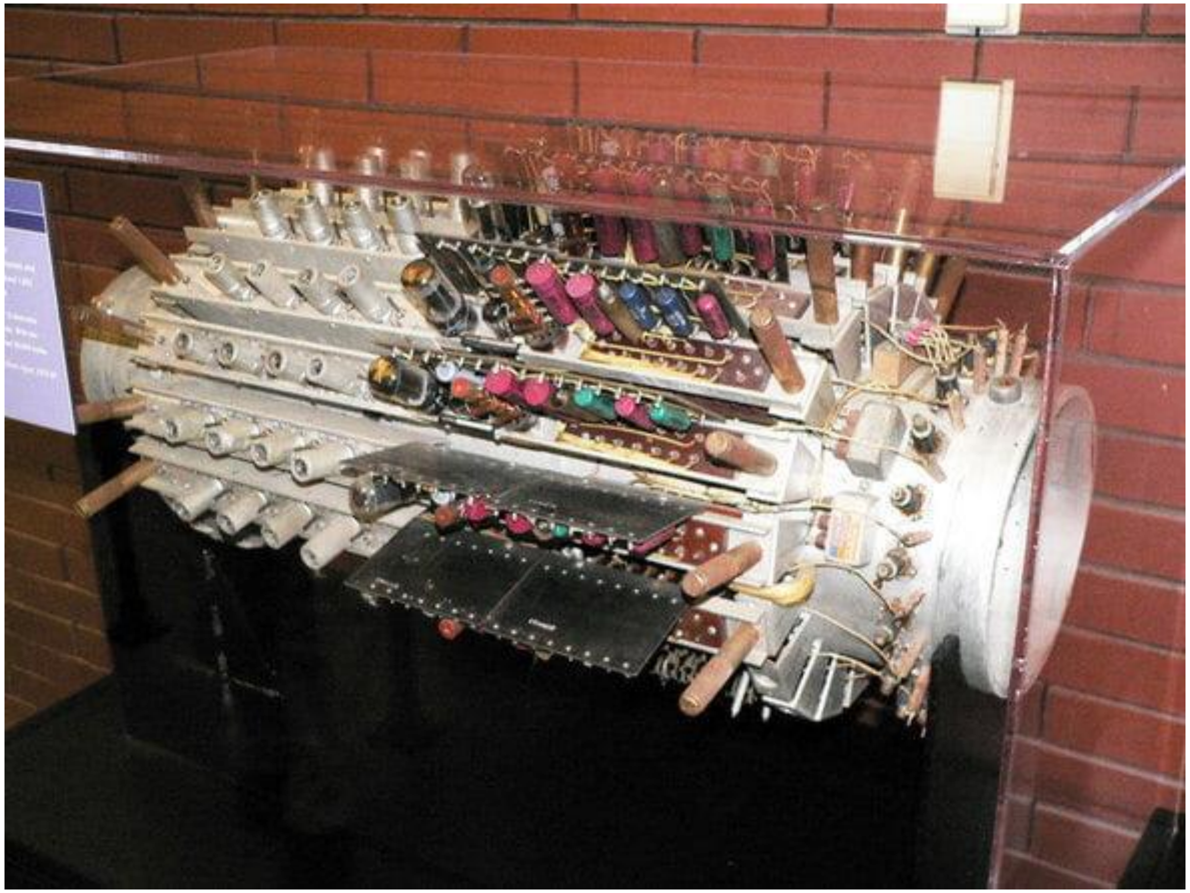
Memory Classes

Two major types

1. Memory which is temporary
   Volatile Memory

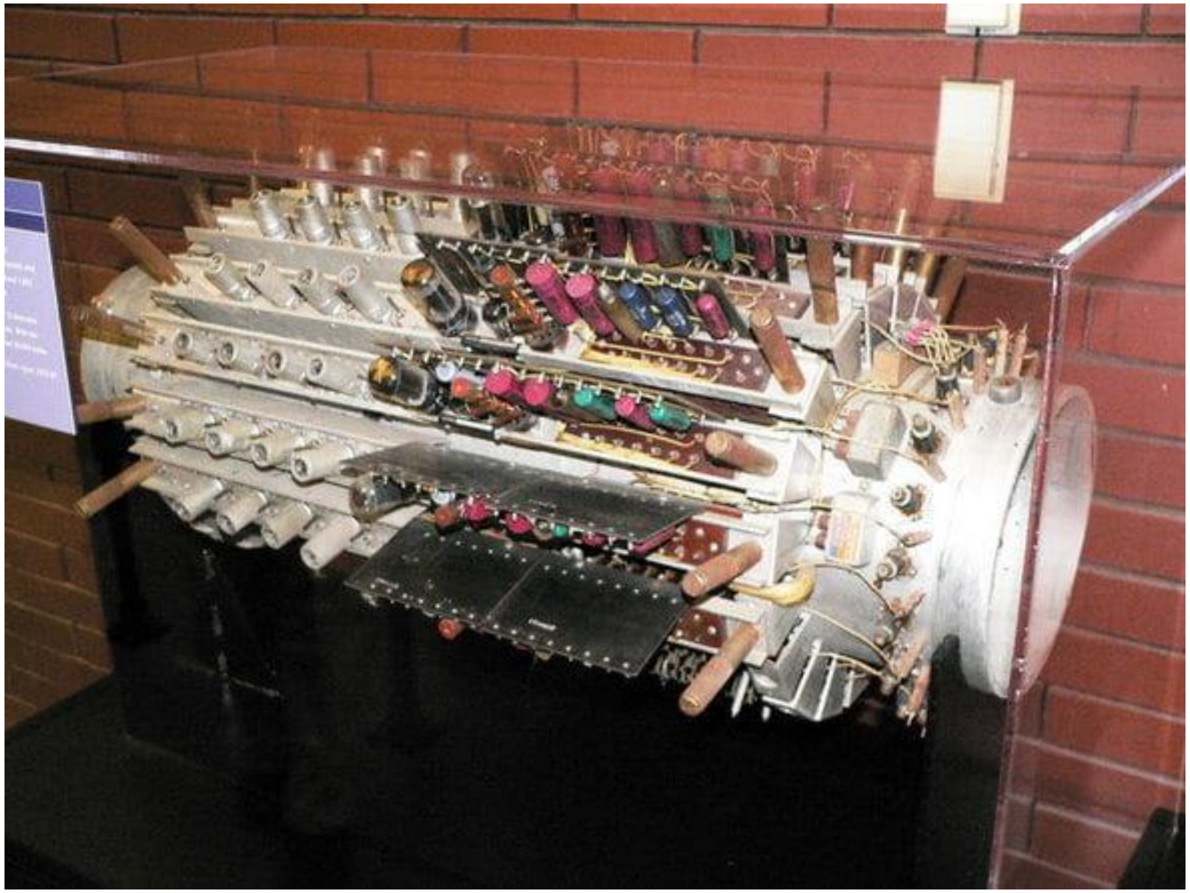
2. Memory which is permanent
   Non-Volatile Memory

Volatile Memory
Loses contents when power is off.



- How much memory do you think this stores?

Volatile Memory
Loses contents when power is off.



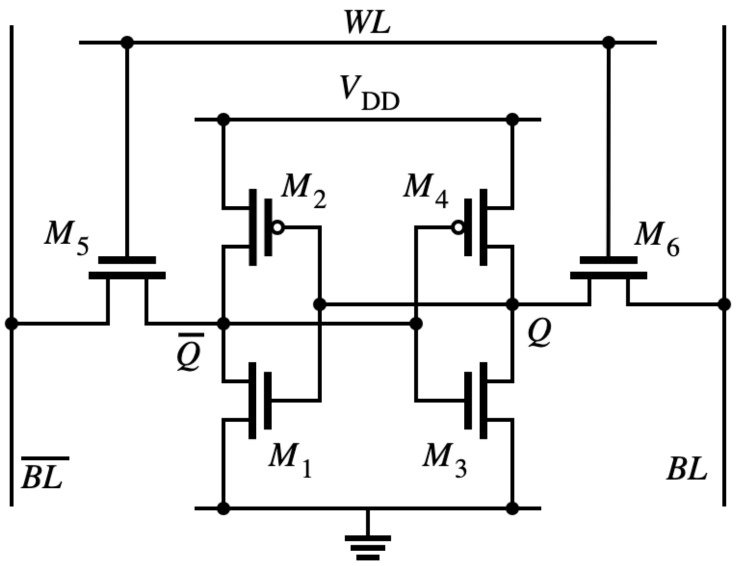
- Mercury Delay Line (UNIVAC)
  - Roughly 2000 bytes for the entire tank

Volatile Memory

Loses contents when power is off.

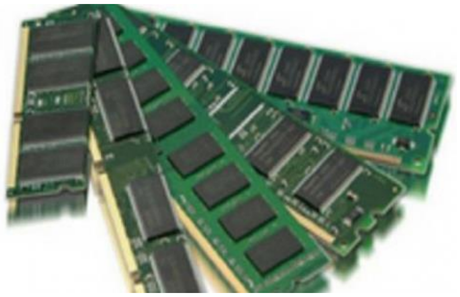
- SRAM: static random-access memory
  - Fast, deterministic access time
  - Used for registers, caches, and small embedded memories
  - But more power hungry and less dense than DRAM

One bit of SRAM

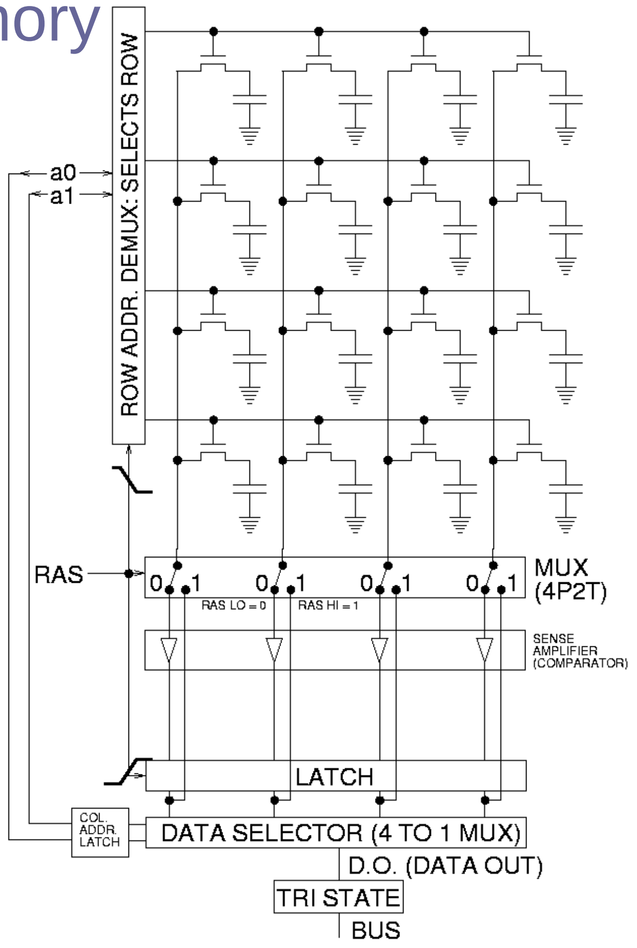


Volatile Memory

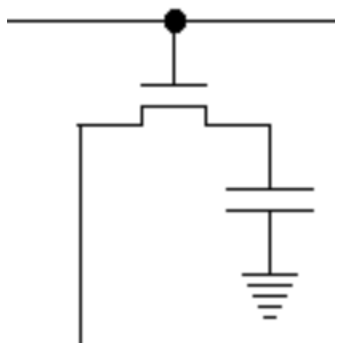
Loses contents when power is off.



- DRAM: dynamic random-access memory
  - Slower than SRAM
  - Access time depends on the sequence of addresses
  - Denser than SRAM (higher capacity)
  - Requires periodic refresh (typically every 64 milliseconds)
  - Typically used for main memory



One bit of DRAM



Non-Volatile Memory
Preserves contents when power is off

- EPROM: erasable programmable read only memory
  - Invented by Dov Frohman of Intel in 1971
  - Erase by exposing the chip to strong UV light
- EEPROM: electrically erasable programmable read-only memory
  - Invented by George Perlegos at Intel in 1978

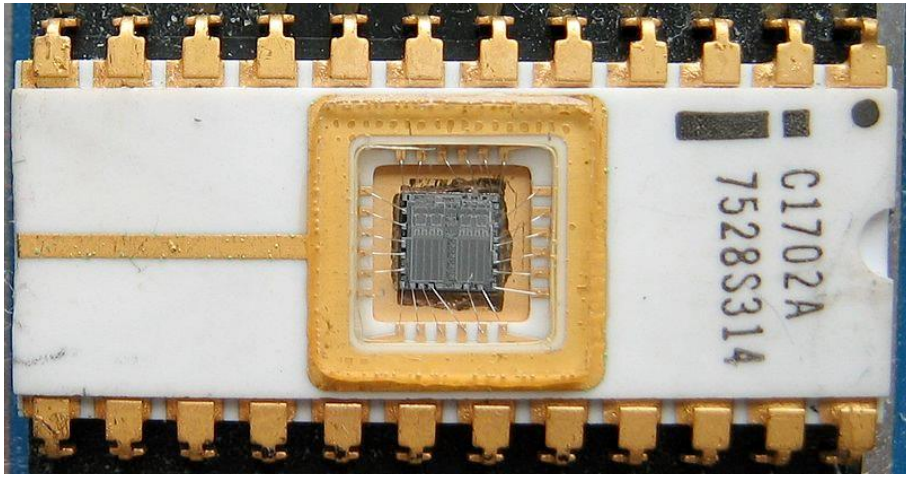
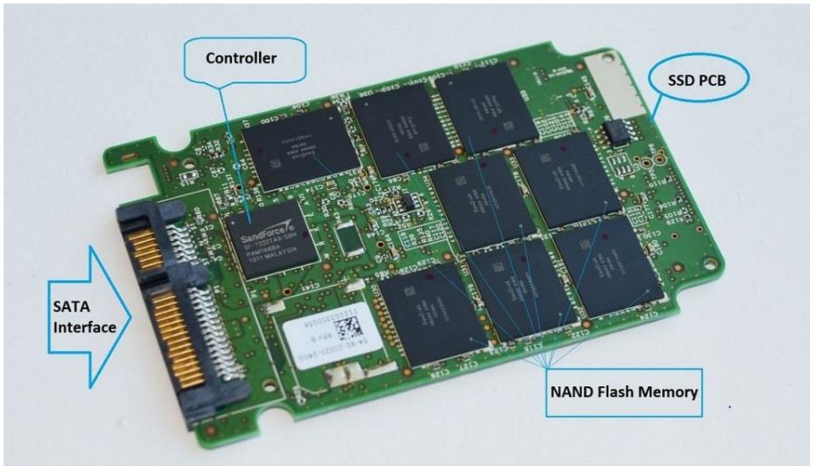


Image from the Wikimedia Commons

Non-Volatile Memory

Preserves contents when power is off

- Disk drives
  - Not as well suited for embedded systems
- Flash memory
  - Invented by Dr. Fujio Masuoka at Toshiba around 1980
  - Erased a “block” at a time
  - Limited number of program/erase cycles (~100,000)
  - Controllers can get quite complex



Example:

Die of a STM32F103VGT6 ARM Cortex-M3 microcontroller with 1 megabyte flash memory by STMicroelectronics.

Which part is the memory?

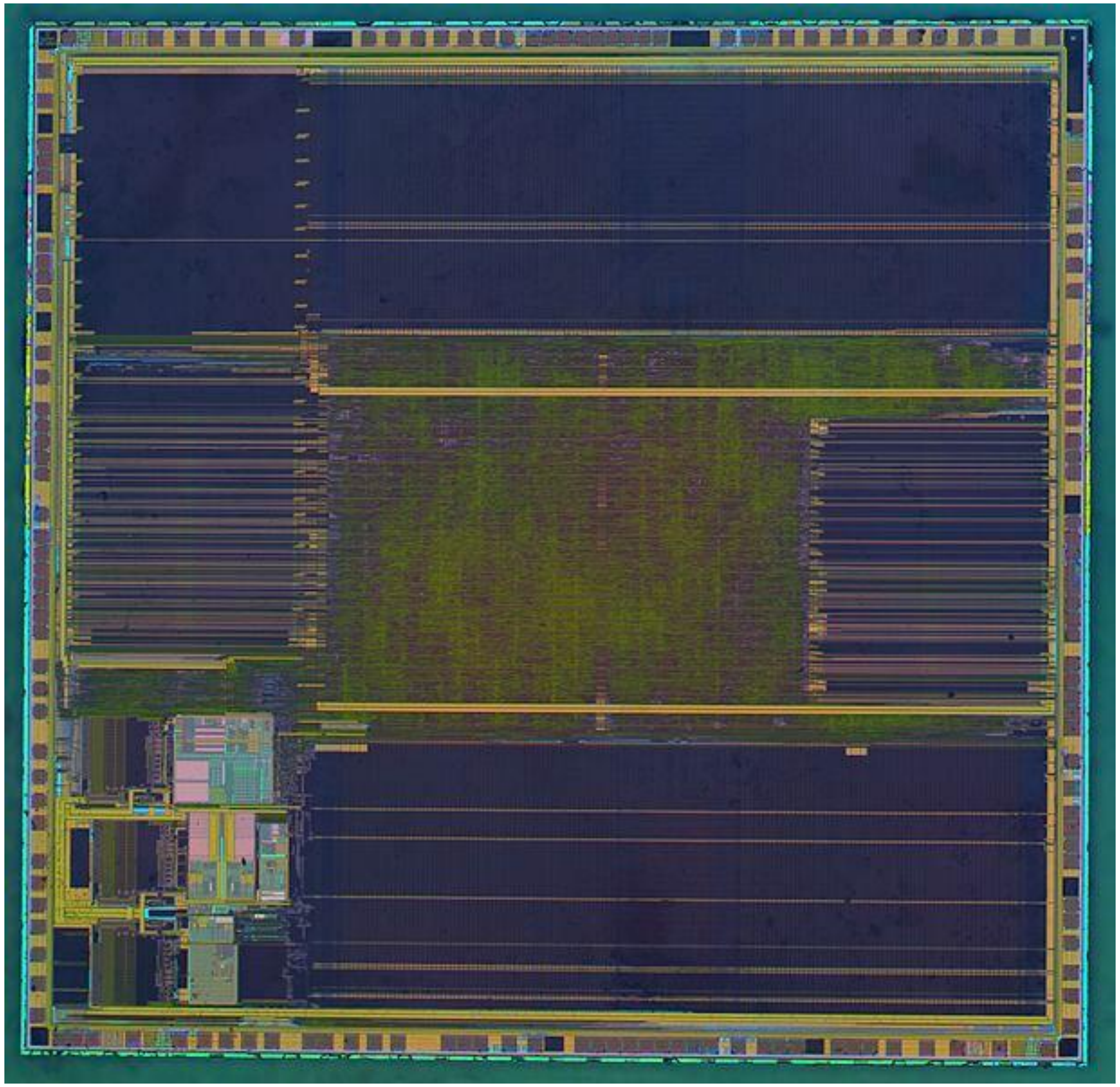


Image from Wikimedia Commons

Example:

Die of a STM32F103VGT6 ARM Cortex-M3 microcontroller with 1 megabyte flash memory by STMicroelectronics.

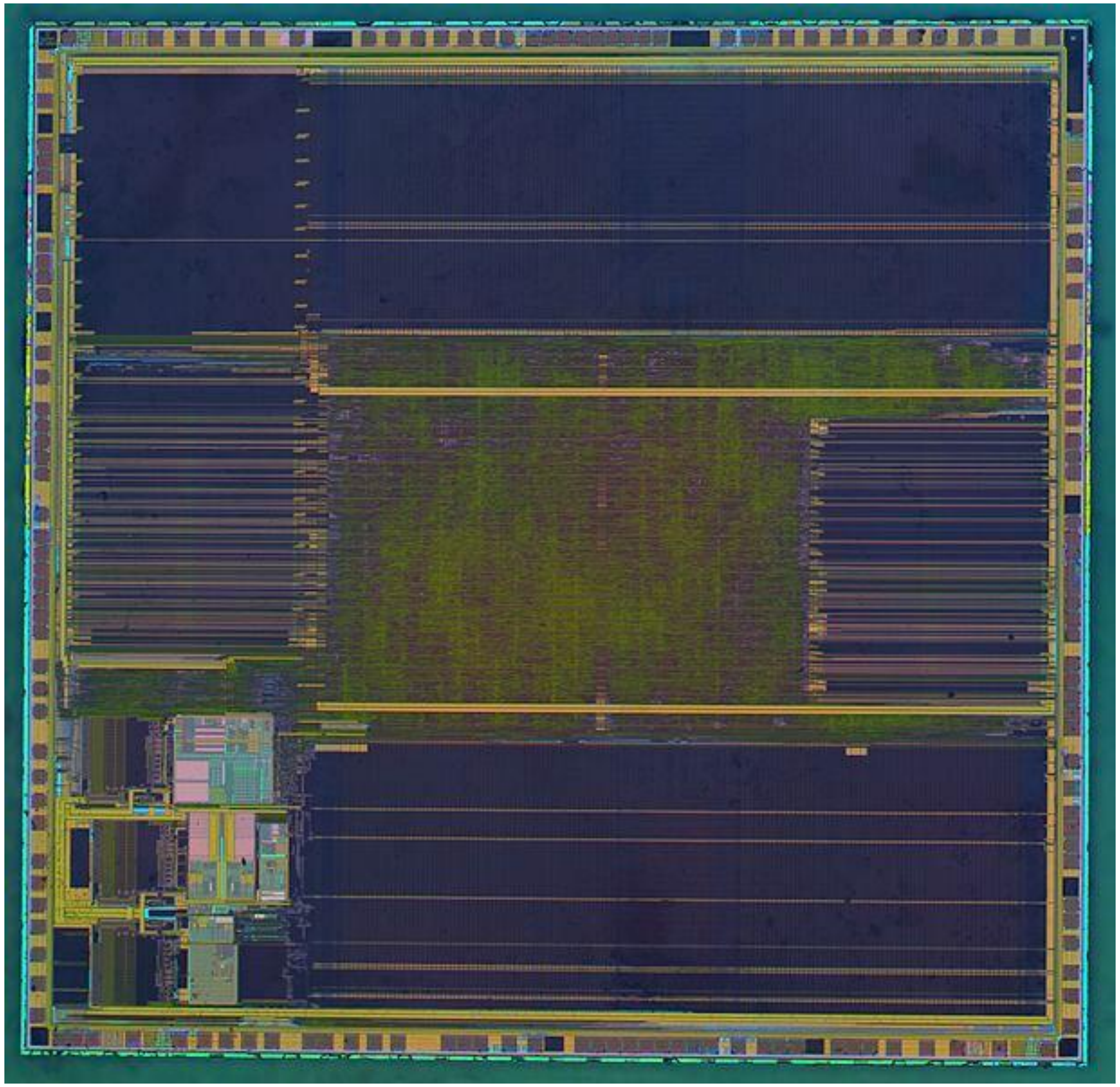


Image from Wikimedia Commons

Which part is the memory?

Just about everything but the bottom right corner

MEMORY HIERARCHIES

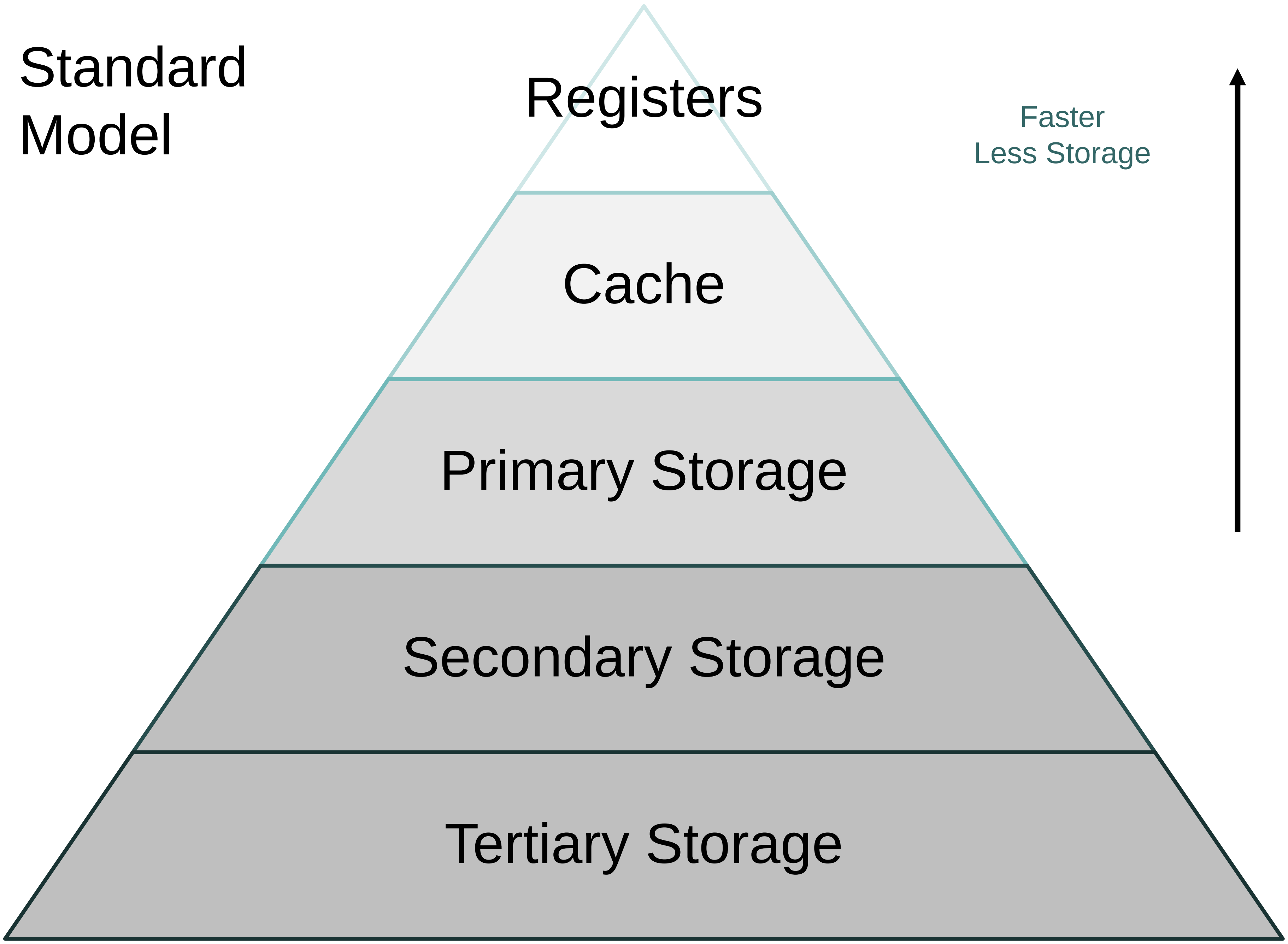
Memory Hierarchy

- Memories on a system can be arranged as a pyramid
  - Top is the most frequently used memory
  - Bottom is the least frequently used

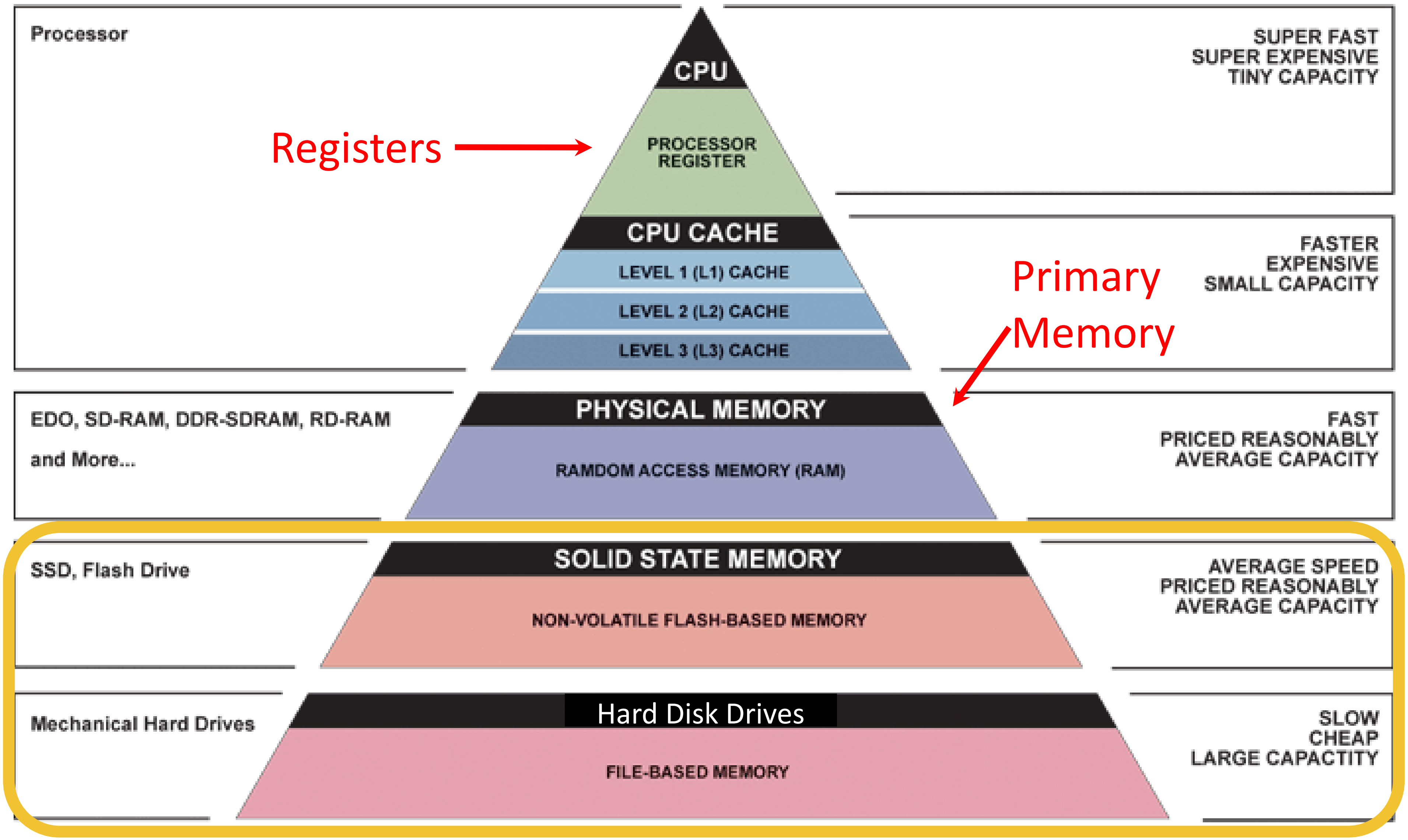
- Let’s draw the hierarchy pyramid

- What are the capabilities and constraints as you move up and down the hierarchy? Which are volatile/non-volatile?

Standard Model



Personal Computer Memory Hierarchy



Accessing Memories

How do you access:

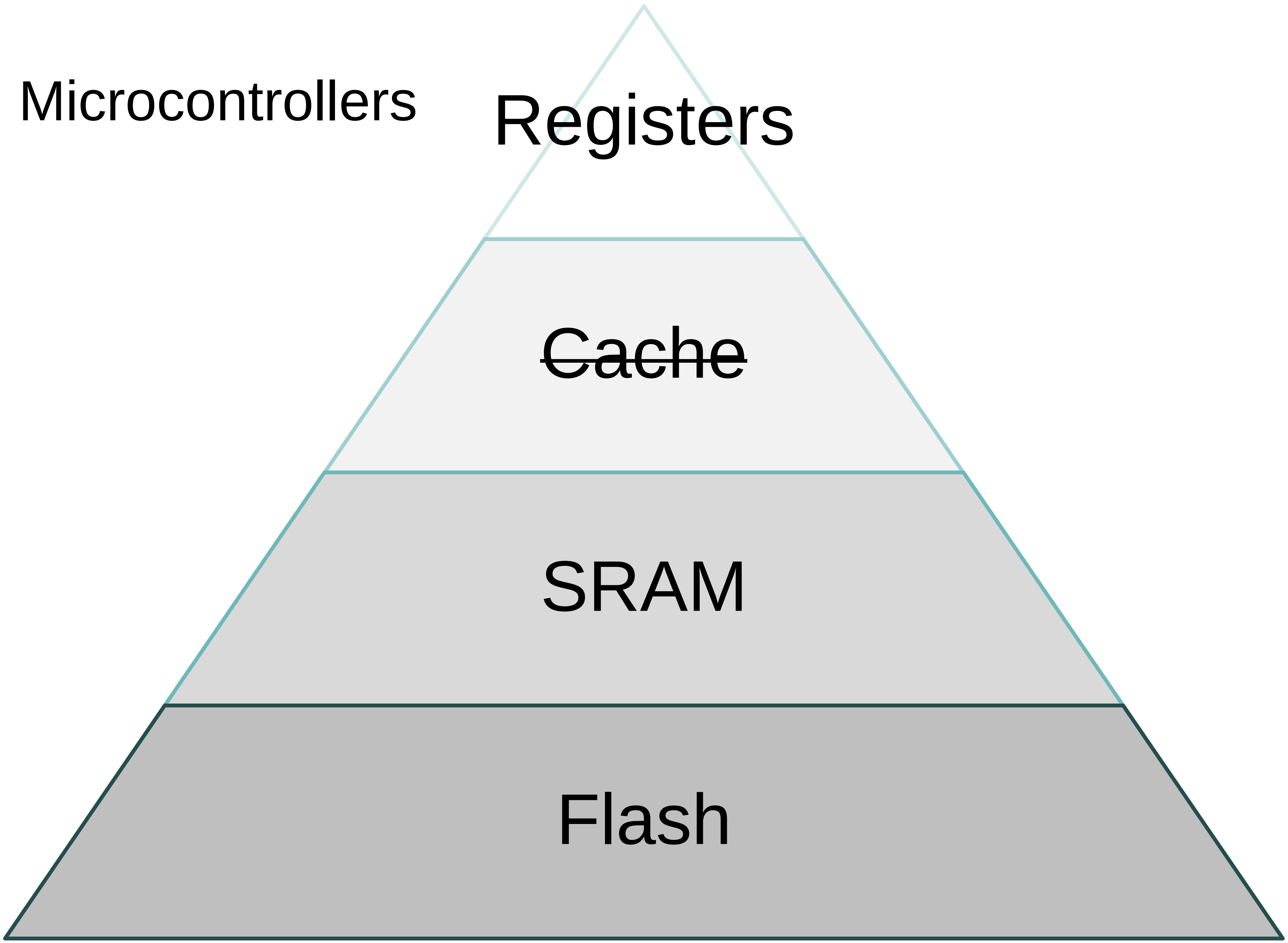
- Registers
  - Directly in assembly code
- Cache
  - Automatically handled by hardware
- RAM
  - Load/Store instructions in assembly code
  - This is the main memory for the system
- Disk
  - The OS handles copying pages into RAM

Outline

- Memory Overview
  - Types of Memory
  - Memory Hierarchy
- Embedded Systems Memory
  - Memory Maps
  - Memory-Mapped I/O
  - Lab Hardware Examples
- Memory Organization
  - Stacks & Heaps
  - Code examples

MEMORY MAPS

Microcontrollers



Caches in Embedded Systems

Why do embedded systems avoid using caches?

Caches in Embedded Systems

Why do embedded systems avoid using caches?

Caches improve performance, but making timing unreliable (could be faster or slower in any given case)

A Fact About the 20th Century Notion of Computing: Timing is not Part of Software Semantics

***Correct** execution of a program in C, C#, Java, Haskell, OCaml, Esterel, etc. has nothing to do with how long it takes to do anything. Nearly all our computation and networking abstractions are built on this premise.*

Caches improve *performance* for a fixed cost, at the expense of making it very difficult to control timing.

Accessing Embedded Memories

How do you access:

- Registers
  - Directly in assembly code
- SRAM (Volatile)
  - Load/Store instructions in assembly code
  - Variables are stored in here
- Flash (Non-Volatile)
  - Load instructions in assembly code (Read-Only)
  - Code executes from here

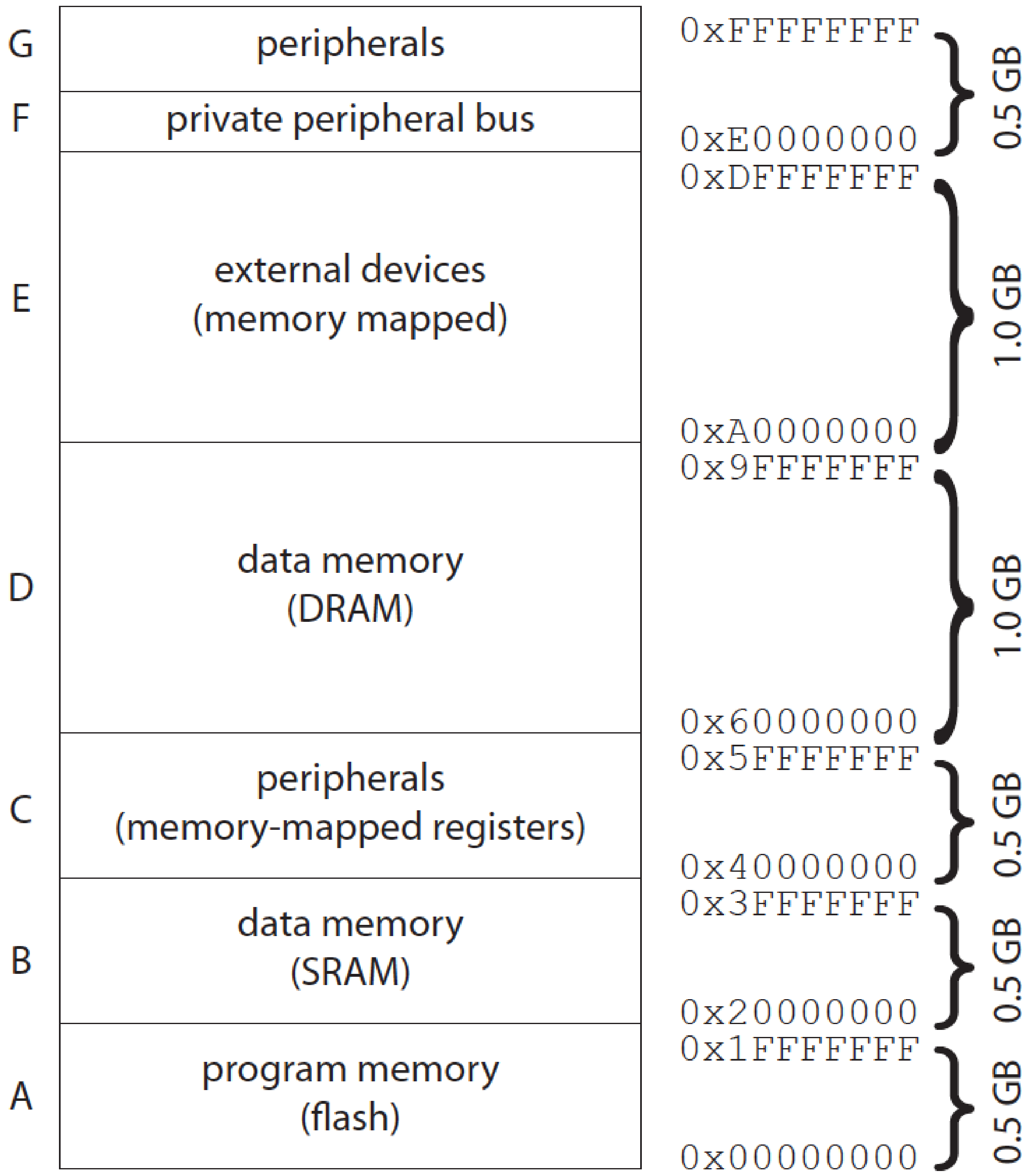
Note: no virtual memory!!

Memory Map of an ARM Cortex - M3 architecture

Defines the mapping of addresses to physical memory.

Why do this?

Note that this does not define how much physical memory there is!



Main Memory on Personal Computers

- Applications on personal computers don’t see a memory map like the Cortex-M3 one
  - Why not?
  - What does their memory look like?

Main Memory on Personal Computers

- Applications on personal computers don’t see a memory map like the Cortex-M3 one
  - Why not?
  - What does their memory look like?

Applications are provided virtual memory spaces, where it appears as if they own all addresses and start at address 0. This makes them easier to create and more secure.

- How this is implemented quickly and securely are major topics of Operating Systems and Computer Architecture

MEMORY-MAPPED I/O

Things That Aren’t Memory

- Microcontrollers have a lot of peripherals
  - General Purpose I/O (GPIO) pins
  - Analog to Digital Converters
  - Digital to Analog Converters
  - Pulse-Width Modulation Generators
  - Timers
  - Various communication buses: UART, SPI, I^{2}C
- How do they access the peripherals?
- Why not create special assembly functions to access them?

Things That Aren't Memory

- Microcontrollers have a lot of peripherals
  - General Purpose I/O (GPIO) pins
  - Analog to Digital Converters
  - Digital to Analog Converters
  - Pulse-Width Modulation Generators
  - Timers
  - Various communication buses: UART, SPI, I^{2}C

- How do they access the peripherals?

  With memory reads and writes

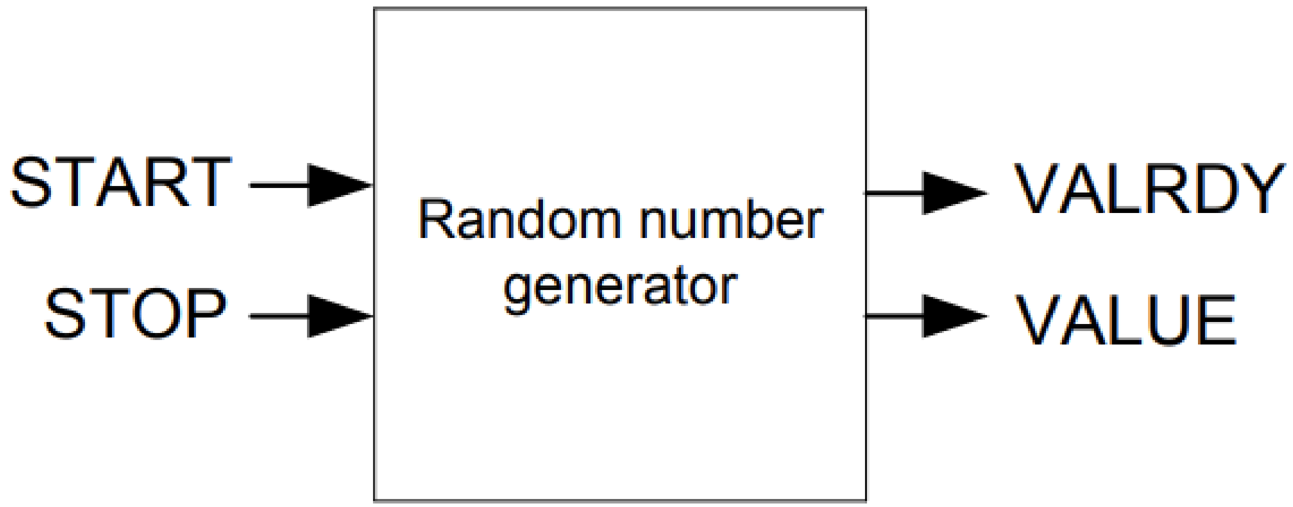
- Why not create special assembly functions to access them?

That would make the processor harder to design. In the memory-mapped case, one processor can use an arbitrary selection of peripherals and doesn't have to know anything about them.

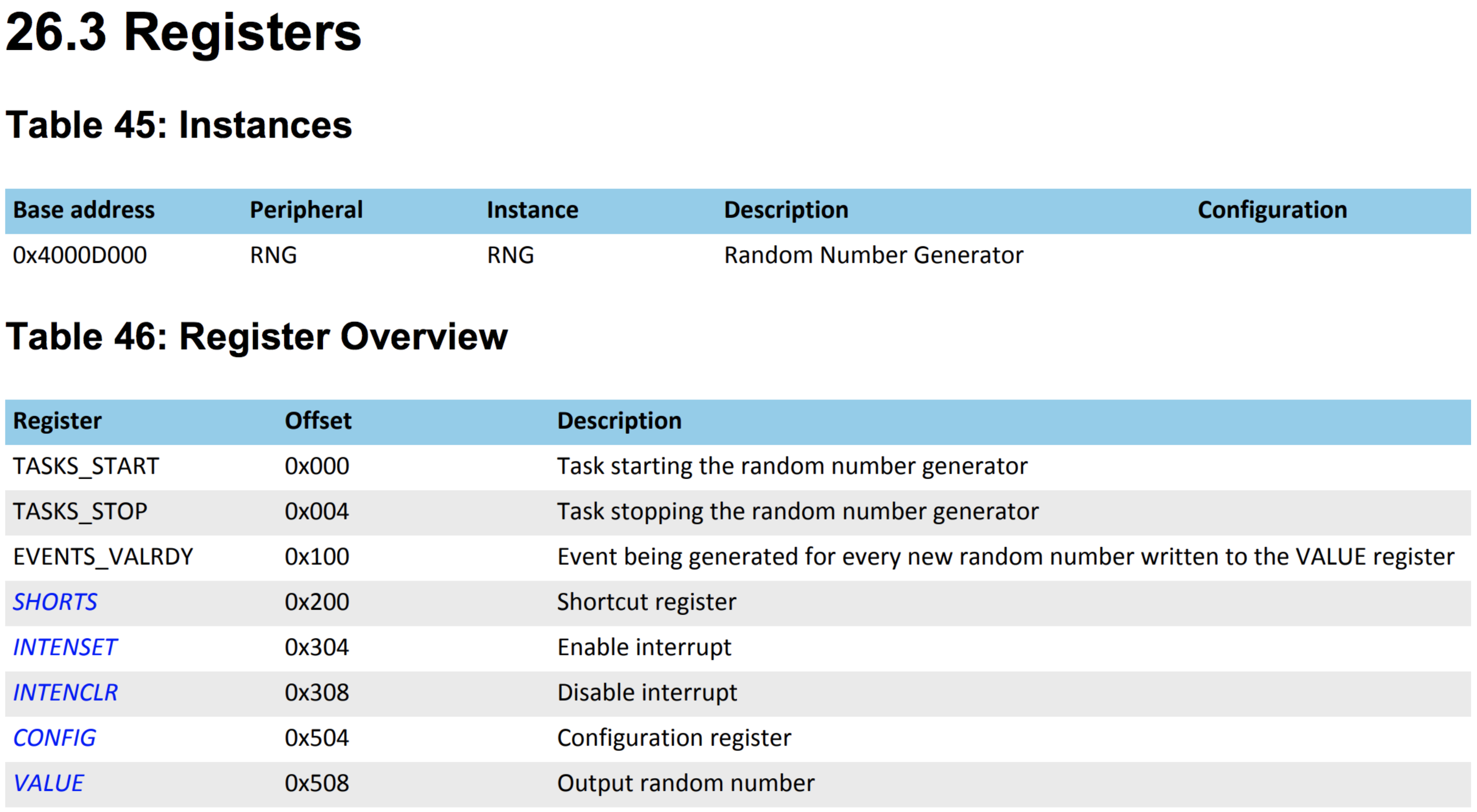
Example: Random Number Generator

Example RNG peripheral from the nRF52832

Interface:



Example: Random Number Generator



26.3 Registers

Table 45: Instances

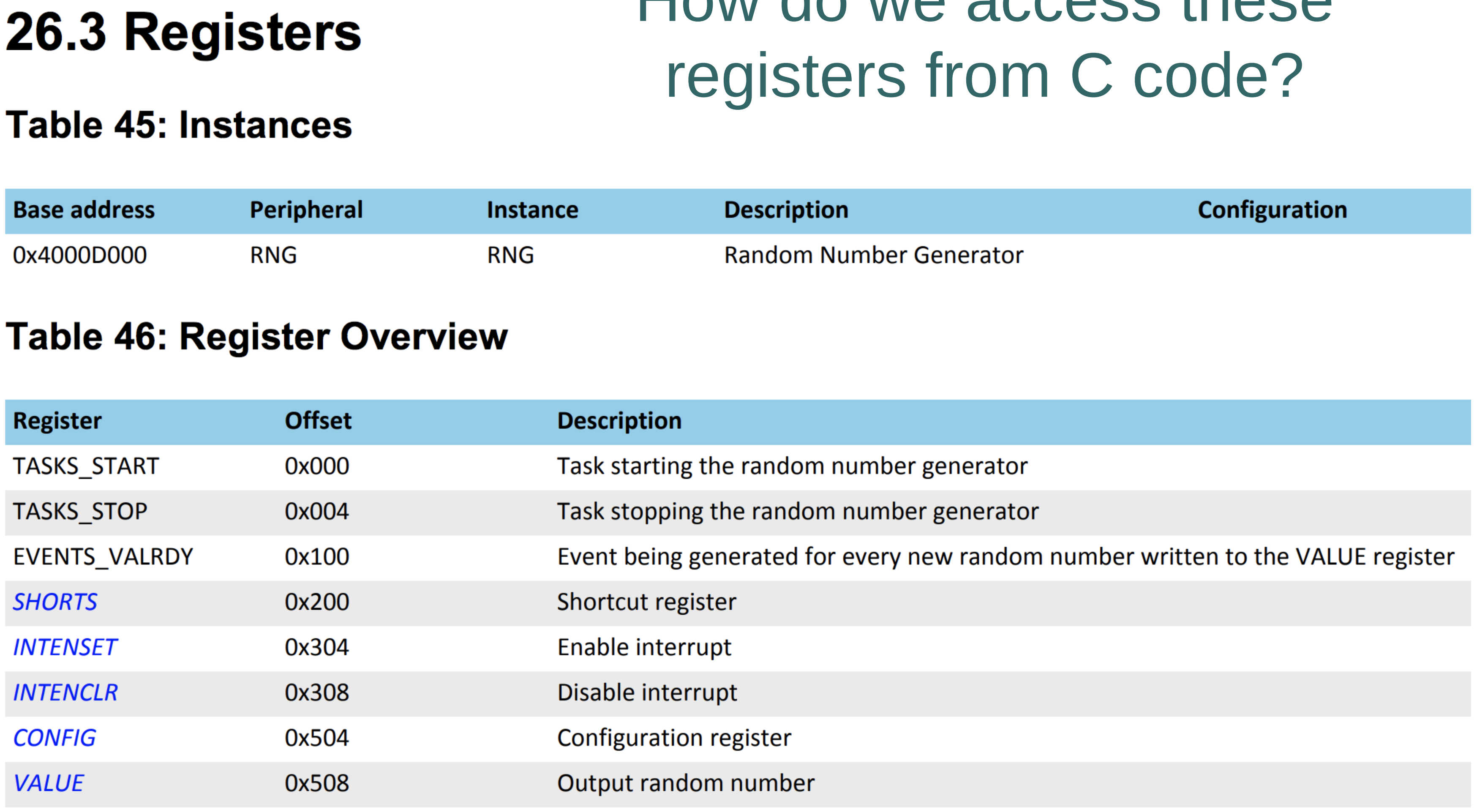
Base address	Peripheral	Instance	Description	Configuration
0x4000D000	RNG	RNG	Random Number Generator	

Table 46: Register Overview

Register	Offset	Description
TASKS_START	0x000	Task starting the random number generator
TASKS_STOP	0x004	Task stopping the random number generator
EVENTS_VALRDY	0x100	Event being generated for every new random number written to the VALUE register
SHORTS	0x200	Shortcut register
INTENSET	0x304	Enable interrupt
INTENCLR	0x308	Disable interrupt
CONFIG	0x504	Configuration register
VALUE	0x508	Output random number

Example: Random Number Generator

How do we access these registers from C code?



26.3 Registers

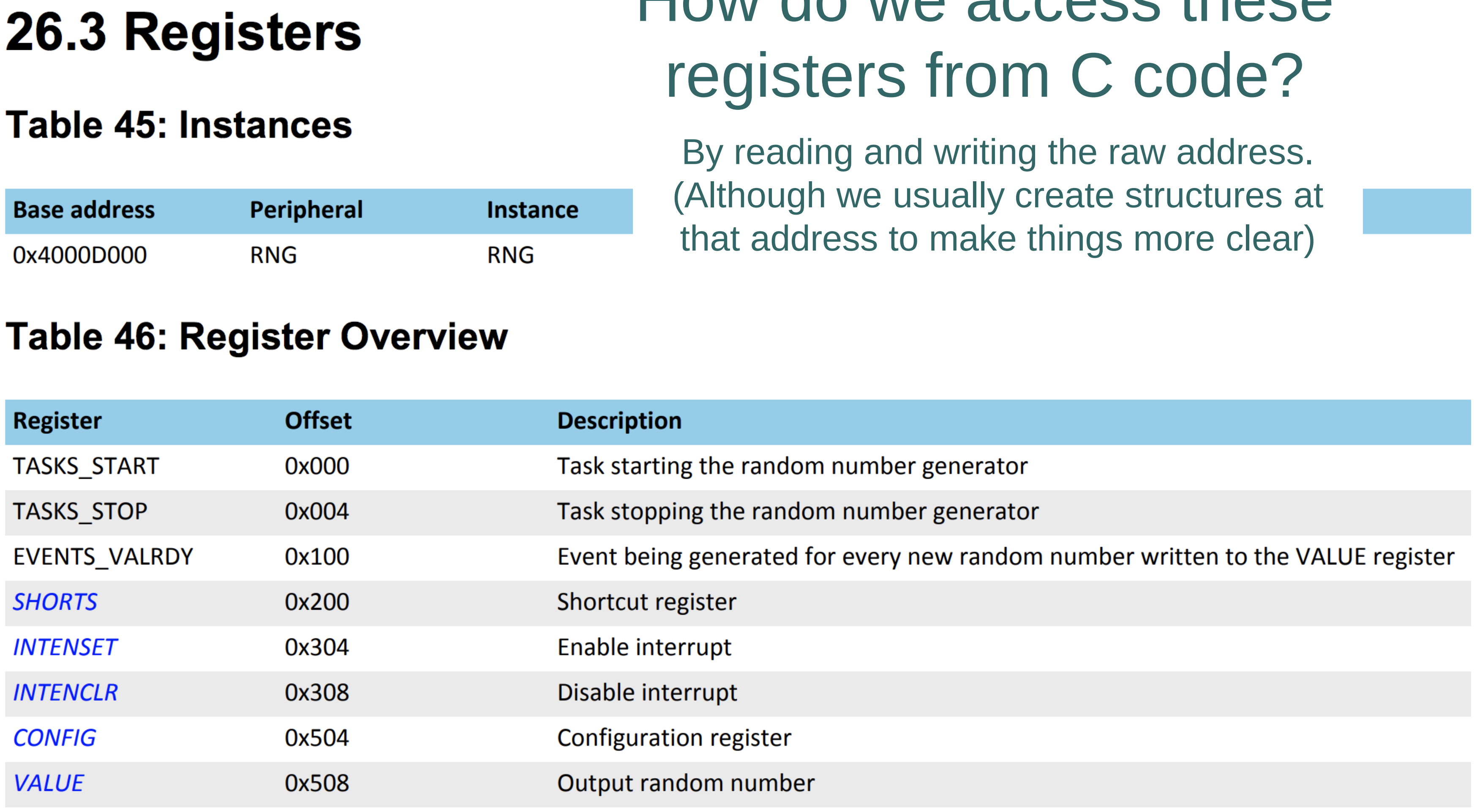
Table 45: Instances

Base address	Peripheral	Instance	Description	Configuration
0x4000D000	RNG	RNG	Random Number Generator	

Table 46: Register Overview

Register	Offset	Description
TASKS_START	0x000	Task starting the random number generator
TASKS_STOP	0x004	Task stopping the random number generator
EVENTS_VALRDY	0x100	Event being generated for every new random number written to the VALUE register
SHORTS	0x200	Shortcut register
INTENSET	0x304	Enable interrupt
INTENCLR	0x308	Disable interrupt
CONFIG	0x504	Configuration register
VALUE	0x508	Output random number

Example: Random Number Generator



26.3 Registers

Table 45: Instances

Base address	Peripheral	Instance
0x4000D000	RNG	RNG

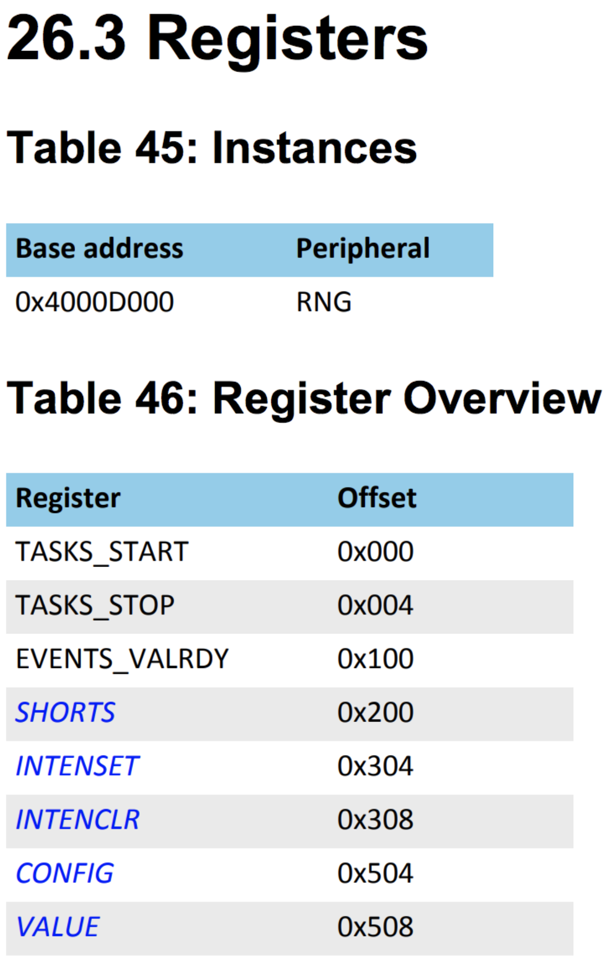
Table 46: Register Overview

Register	Offset	Description
TASKS_START	0x000	Task starting the random number generator
TASKS_STOP	0x004	Task stopping the random number generator
EVENTS_VALRDY	0x100	Event being generated for every new random number written to the VALUE register
SHORTS	0x200	Shortcut register
INTENSET	0x304	Enable interrupt
INTENCLR	0x308	Disable interrupt
CONFIG	0x504	Configuration register
VALUE	0x508	Output random number

How do we access these registers from C code?

By reading and writing the raw address.
(Although we usually create structures at that address to make things more clear)

Example: Random Number Generator



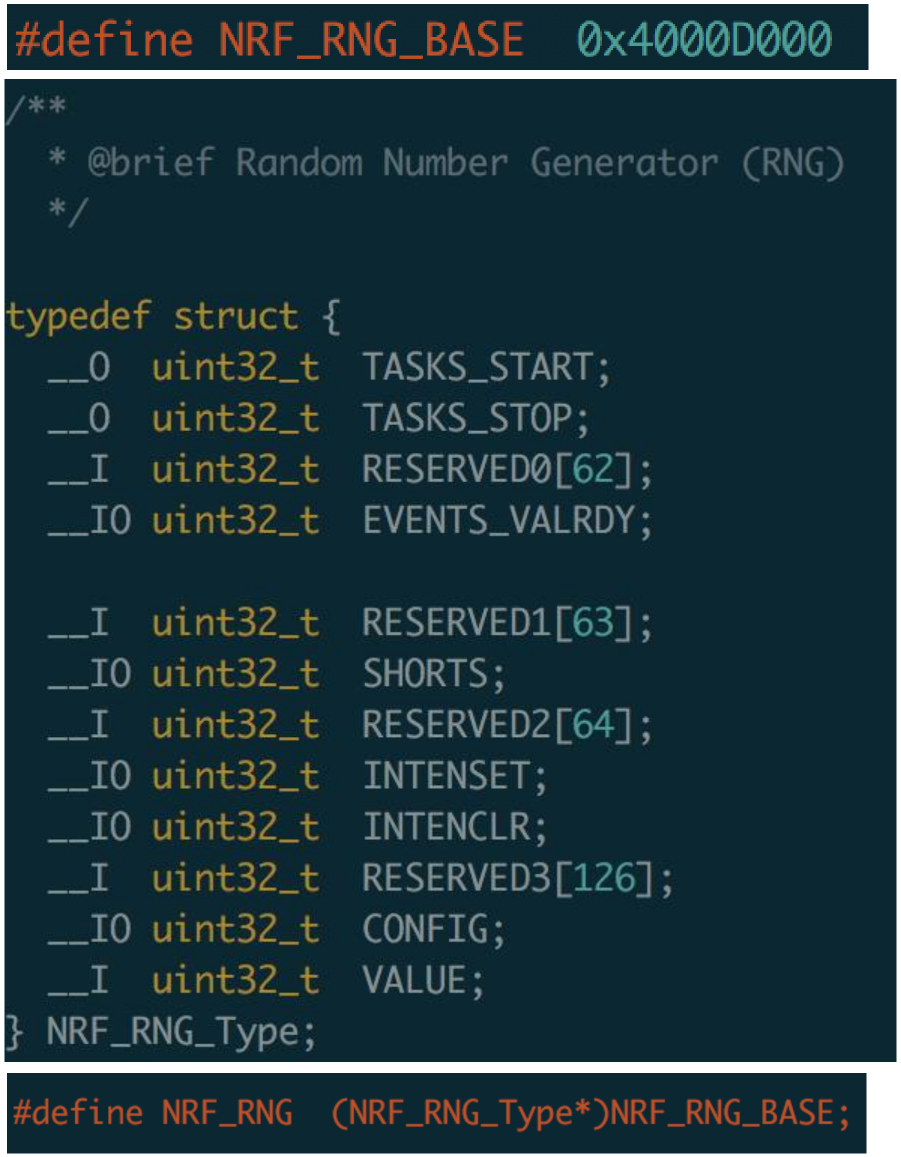
26.3 Registers

Table 45: Instances

Base address	Peripheral
0x4000D000	RNG

Table 46: Register Overview

Register	Offset
TASKS_START	0x000
TASKS_STOP	0x004
EVENTS_VALRDY	0x100
SHORTS	0x200
INTENSET	0x304
INTENCLR	0x308
CONFIG	0x504
VALUE	0x508



```
#define NRF_RNG_BASE   0x4000D000
```

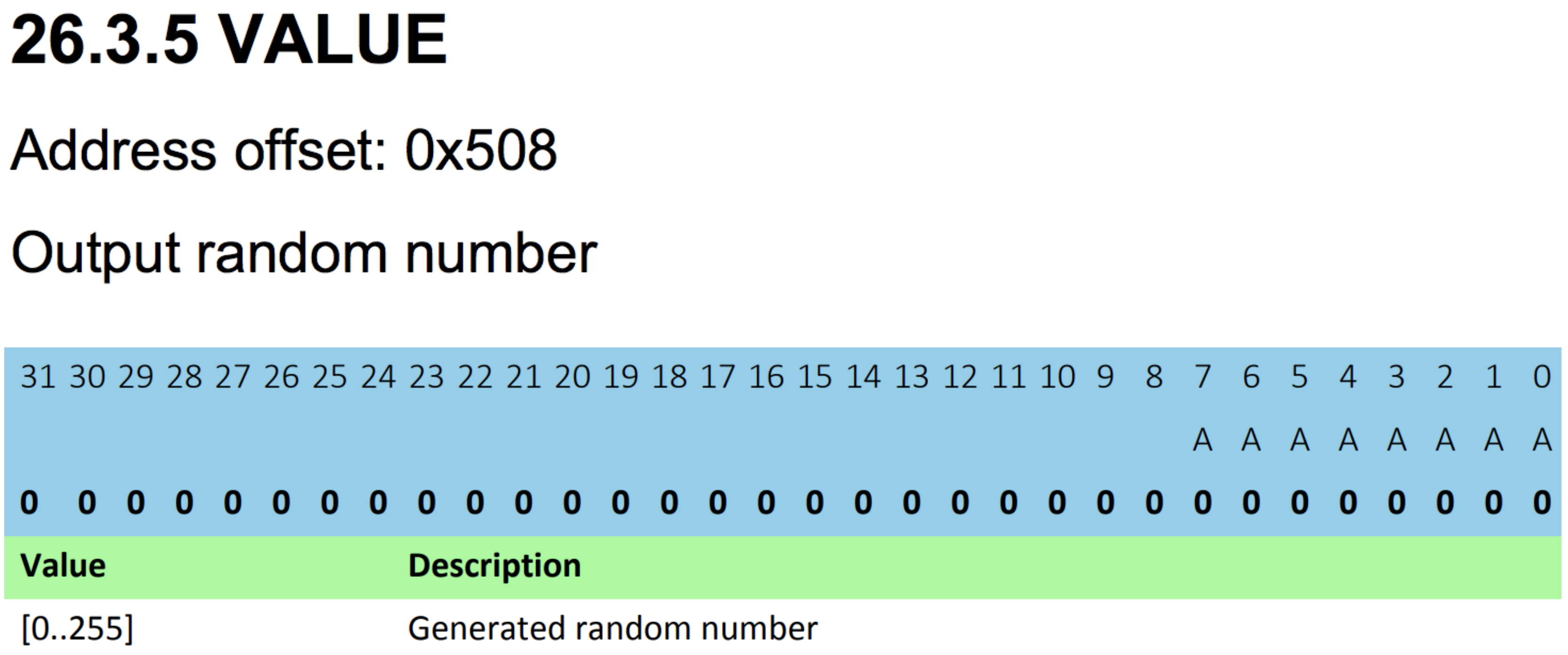
```
/**
  * @brief Random Number Generator (RNG)
  */

typedef struct {
  __O  uint32_t  TASKS_START;
  __O  uint32_t  TASKS_STOP;
  __I  uint32_t  RESERVED0[62];
  __IO uint32_t  EVENTS_VALRDY;

  __I  uint32_t  RESERVED1[63];
  __IO uint32_t  SHORTS;
  __I  uint32_t  RESERVED2[64];
  __IO uint32_t  INTENSET;
  __IO uint32_t  INTENCLR;
  __I  uint32_t  RESERVED3[126];
  __IO uint32_t  CONFIG;
  __I  uint32_t  VALUE;
} NRF_RNG_Type;
```

```
#define NRF_RNG  (NRF_RNG_Type*)NRF_RNG_BASE;
```

Example: Random Number Generator



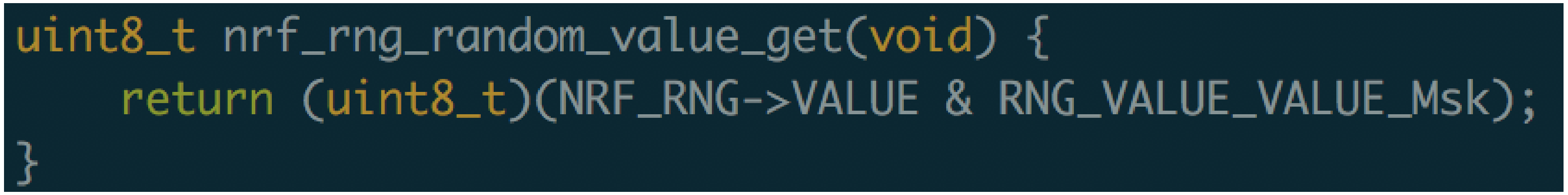
26.3.5 VALUE

Address offset: 0x508

Output random number

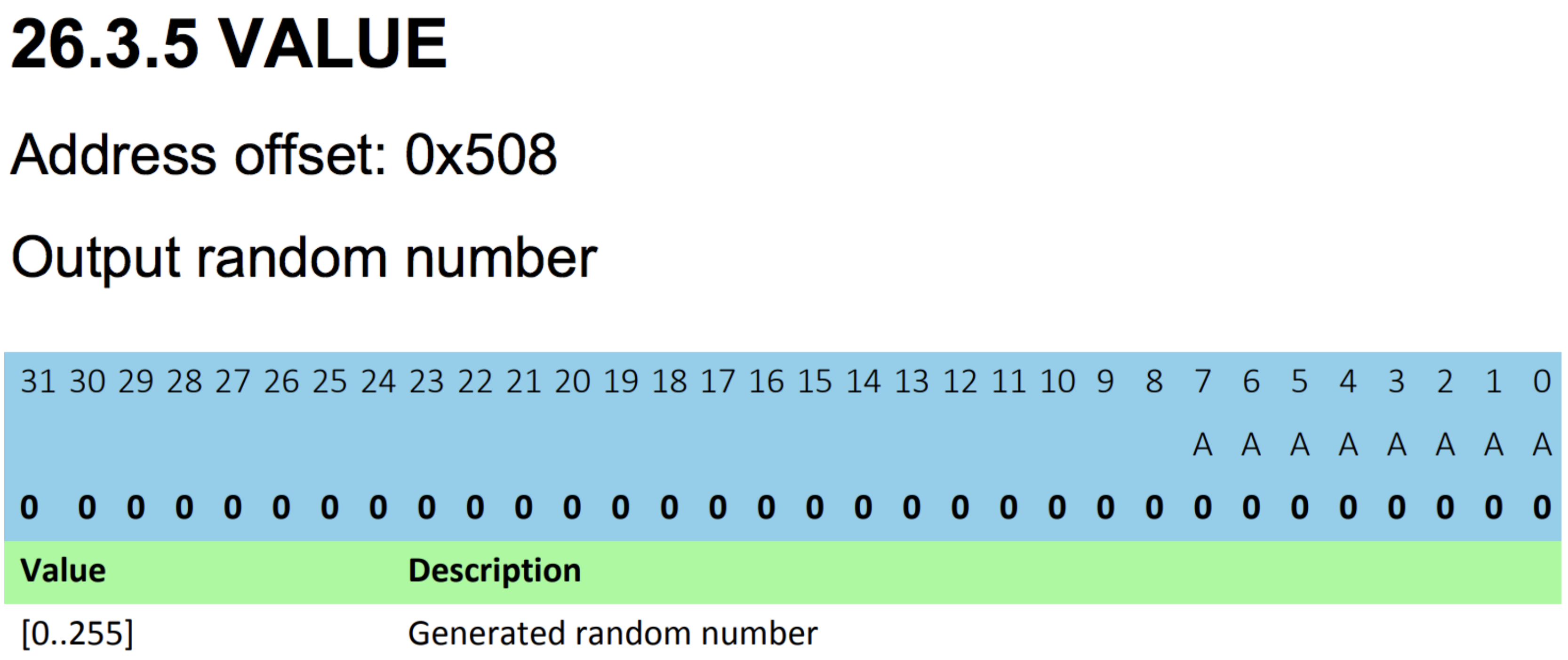
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
																								A	A	A	A	A	A	A	A
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**

Value	**Description**
[0..255]	Generated random number



```
uint8_t nrf_rng_random_value_get(void) {
    return (uint8_t)(NRF_RNG->VALUE & RNG_VALUE_VALUE_Msk);
}
```

Example: Random Number Generator



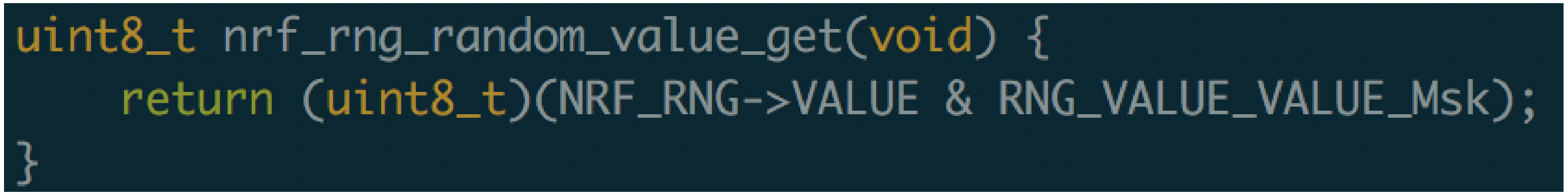
26.3.5 VALUE

Address offset: 0x508

Output random number

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
																								A	A	A	A	A	A	A	A
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**

Value	**Description**
[0..255]	Generated random number



```
uint8_t nrf_rng_random_value_get(void) {
    return (uint8_t)(NRF_RNG->VALUE & RNG_VALUE_VALUE_Msk);
}
```

Example: Random Number Generator

Remember: at the very bottom these are still just memory reads and writes!

```
uint32_t value = NRF_RNG->Value;
```

Is equivalent to

```
uint32_t value = *(uint32_t*)(0x4000D508);
```

LED Blinking Code Example

```
void main(void) {
   const unsigned int LED_NUM  = 1;
   // Sets the LED pin to be an output
   *((volatile unsigned int*)(0x400DC400)) |= (1 << LED_NUM);
   while(1) {
       volatile int i;
       // LED on
       *((volatile unsigned int*)(0x400DC000 + ((1 << LED_NUM) << 2))) = 0x00;
       for (i=0; i<400000; i++);
       // LED off
       *((volatile unsigned int*)((0x400DC000) + ((1 << LED_NUM) << 2))) = 0xFF;
       for (i=0; i<400000; i++);
   }
}
```

Memory Mapped I/O Address!

Compiler: don't optimize me!!!

Delay for a while

LAB HARDWARE

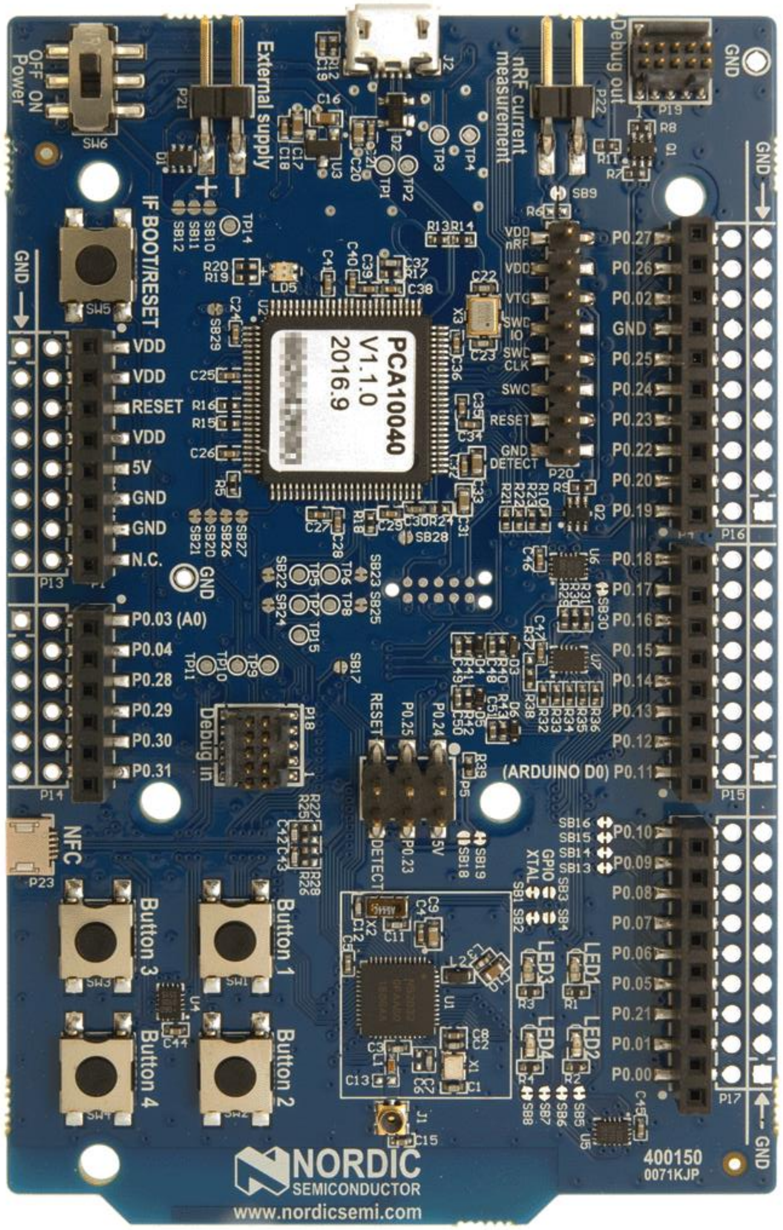
nRF52832 Microcontroller

Processor

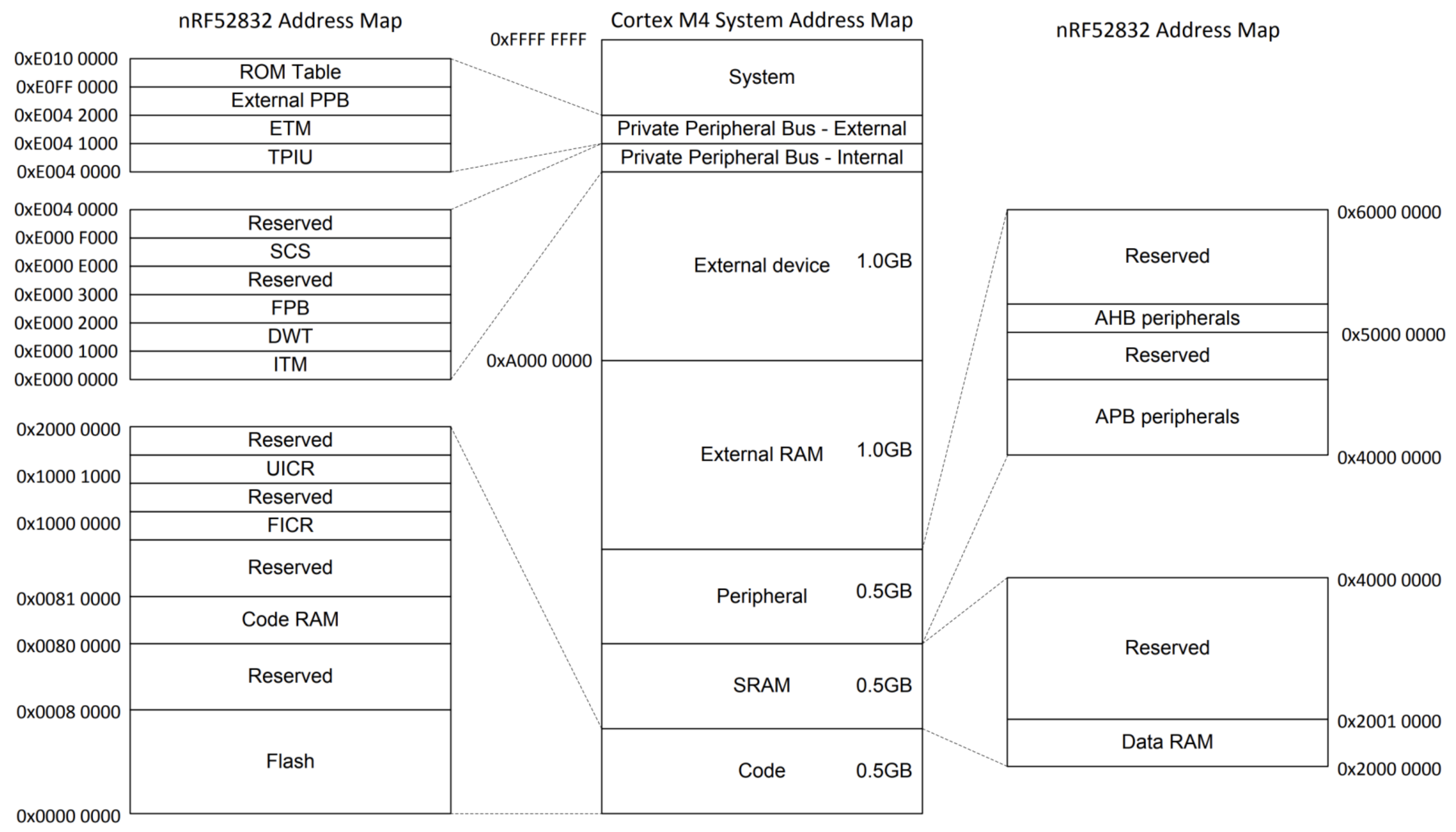
- ARM Cortex-M4F
- 3-stage pipeline!
- Floating point support

Memory

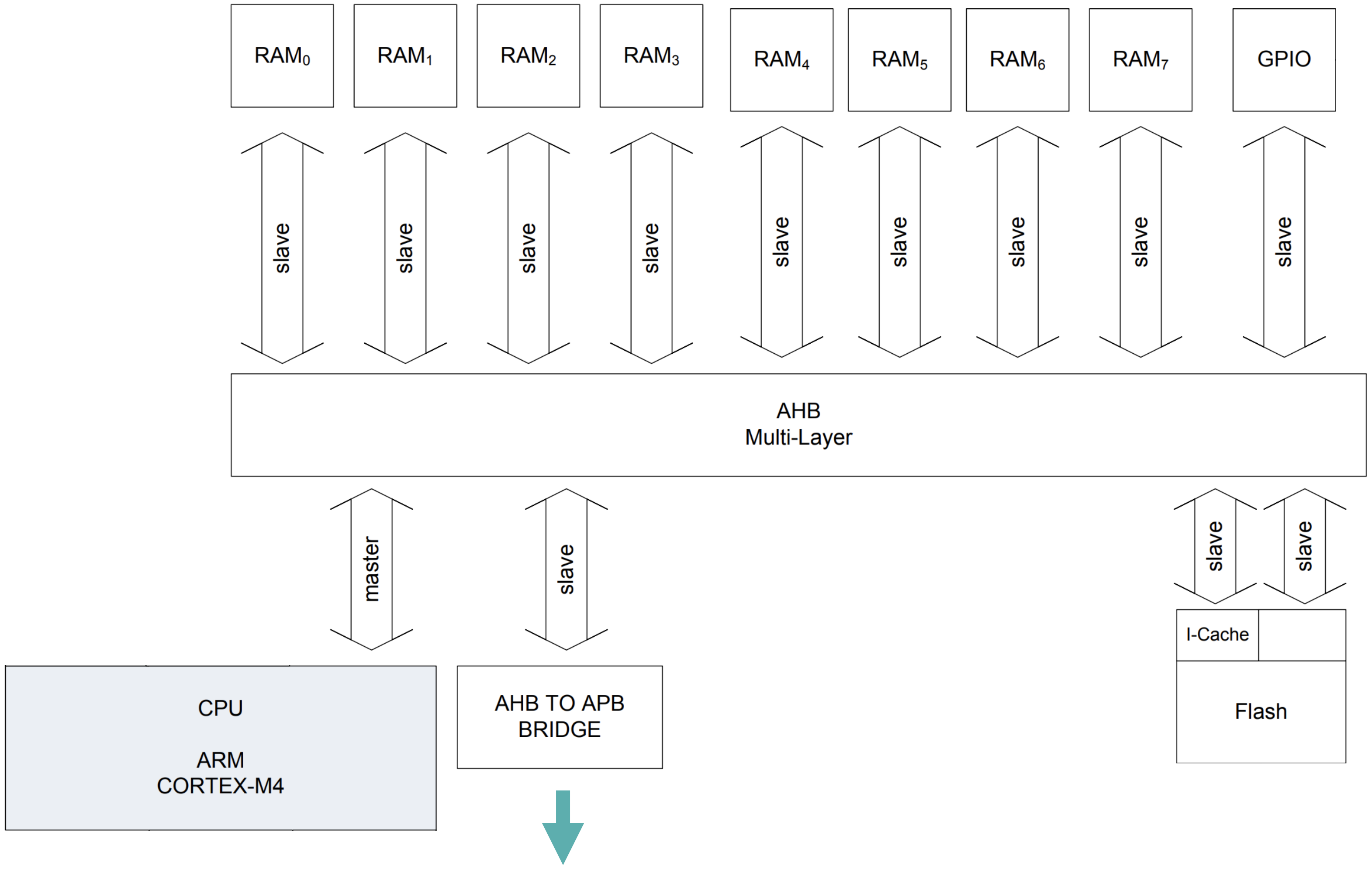
- Instruction Cache
  - Off by default
- 64 kB SRAM
- 512 kB Flash



Memory Map


nRF52832 Address Map
0xE010 0000
ROM Table
0xE0FF 0000
External PPB
0xE004 2000
ETM
0xE004 1000
TPIU
0xE004 0000
0xE004 0000
Reserved
0xE000 F000
SCS
0xE000 E000
Reserved
0xE000 3000
FPB
0xE000 2000
DWT
0xE000 1000
ITM
0xE000 0000
0x2000 0000
Reserved
UICR
0x1000 1000
Reserved
0x1000 0000
FICR
Reserved
0x0081 0000
Code RAM
0x0080 0000
Reserved
0x0008 0000
Flash
0x0000 0000
Cortex M4 System Address Map
0xFFFF FFFF
System
Private Peripheral Bus - External
Private Peripheral Bus - Internal
External device 1.0GB
0xA000 0000
External RAM 1.0GB
Peripheral 0.5GB
SRAM 0.5GB
Code 0.5GB
nRF52832 Address Map
0x6000 0000
Reserved
AHB peripherals
0x5000 0000
Reserved
APB peripherals
0x4000 0000
0x4000 0000
Reserved
0x2001 0000
Data RAM
0x2000 0000

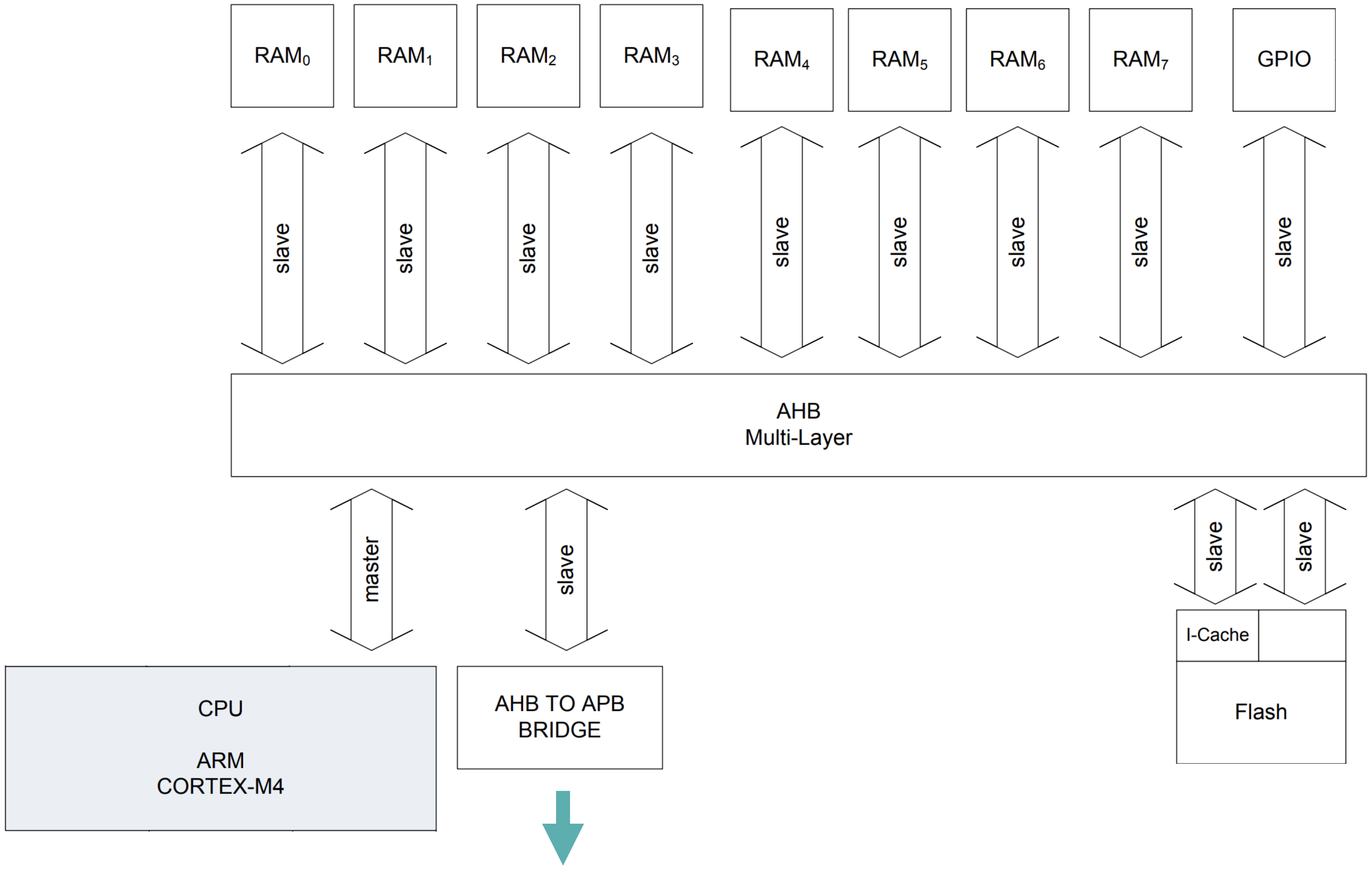
nRF52832 Block Diagram



To Peripherals

nRF52832 Block Diagram

Why have 8 separate RAM banks?

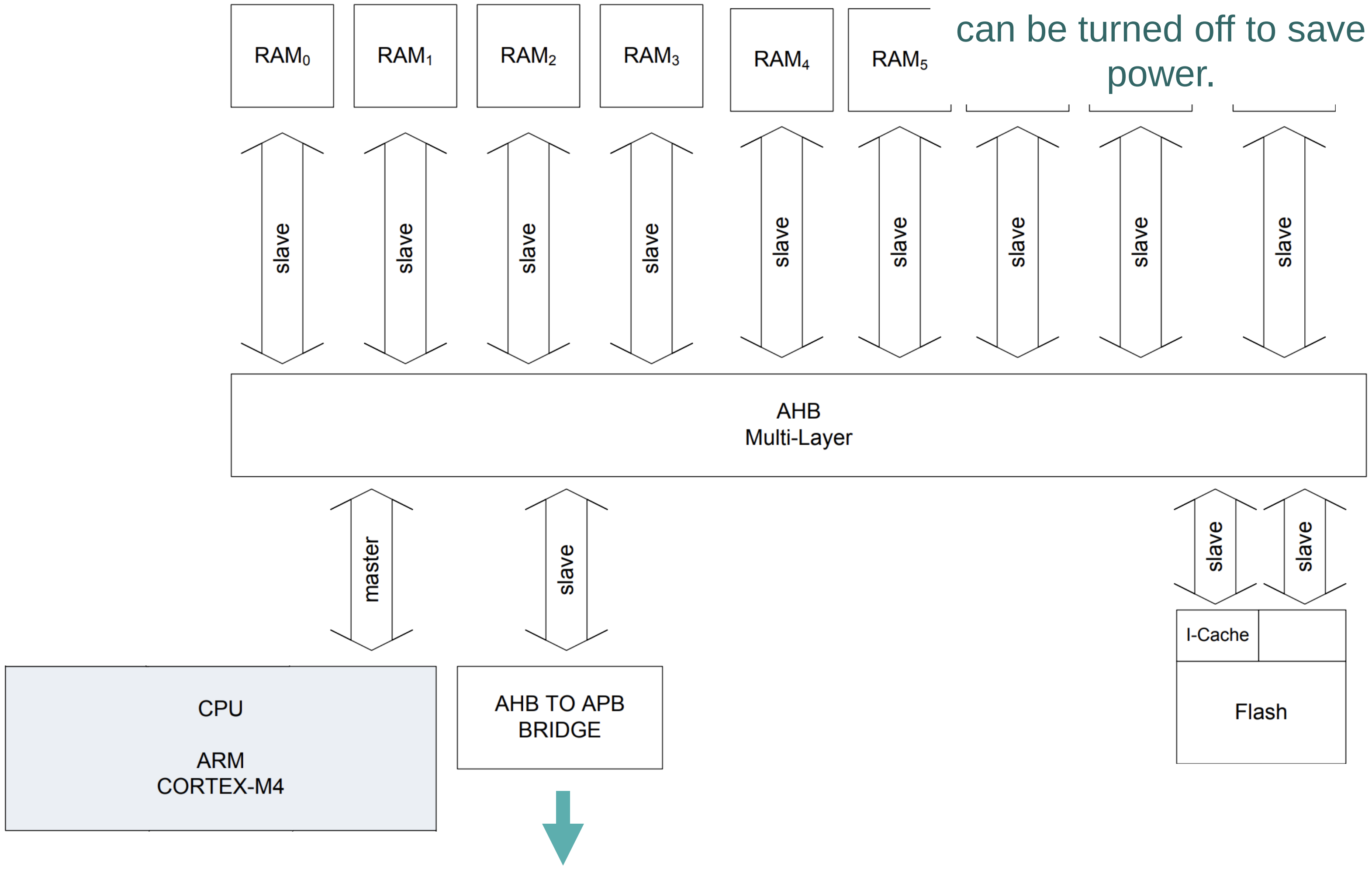


To Peripherals

nRF52832 Block Diagram

Why have 8 separate RAM banks?

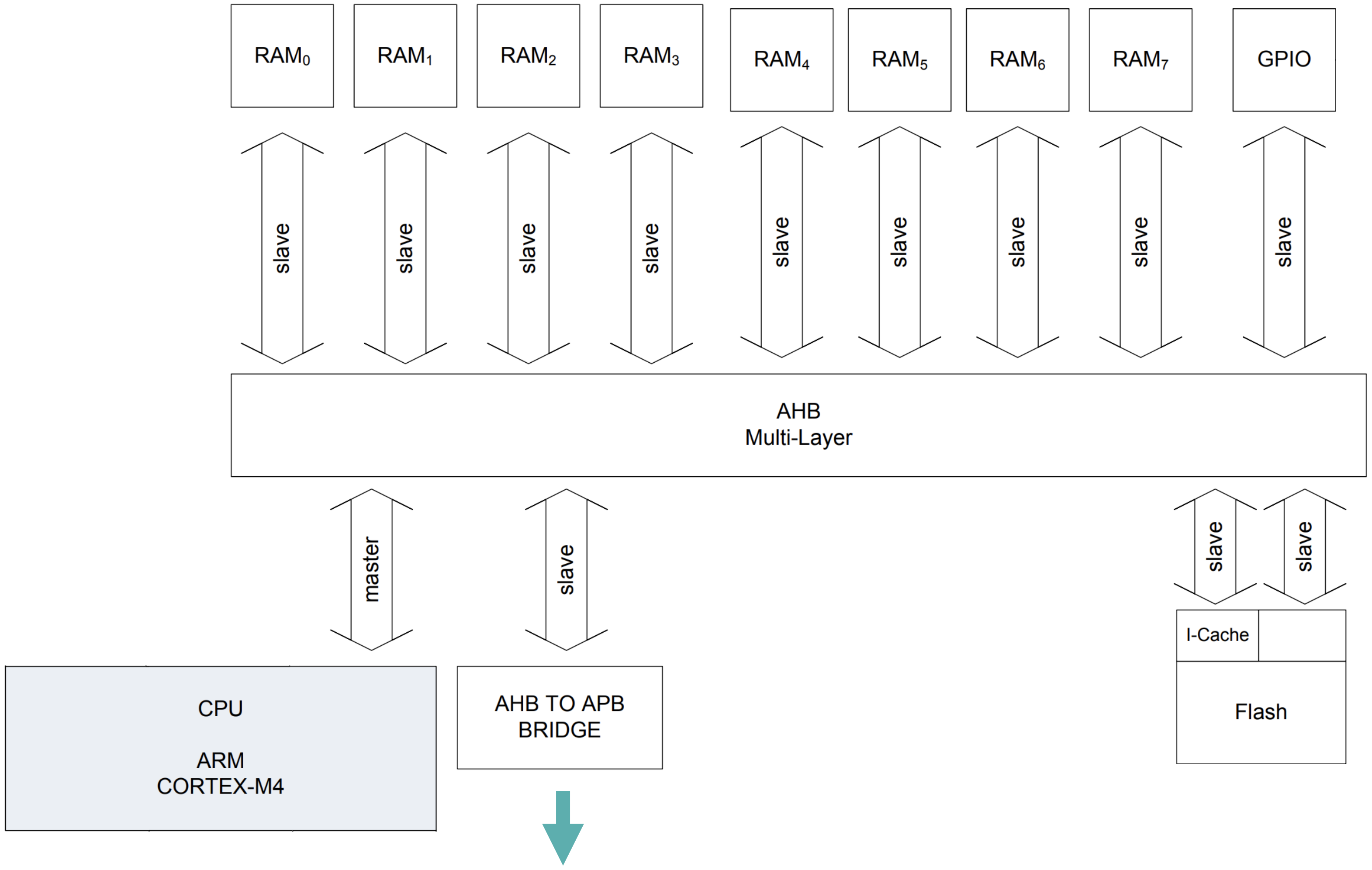
So that extraneous RAM can be turned off to save power.



To Peripherals

nRF52832 Block Diagram

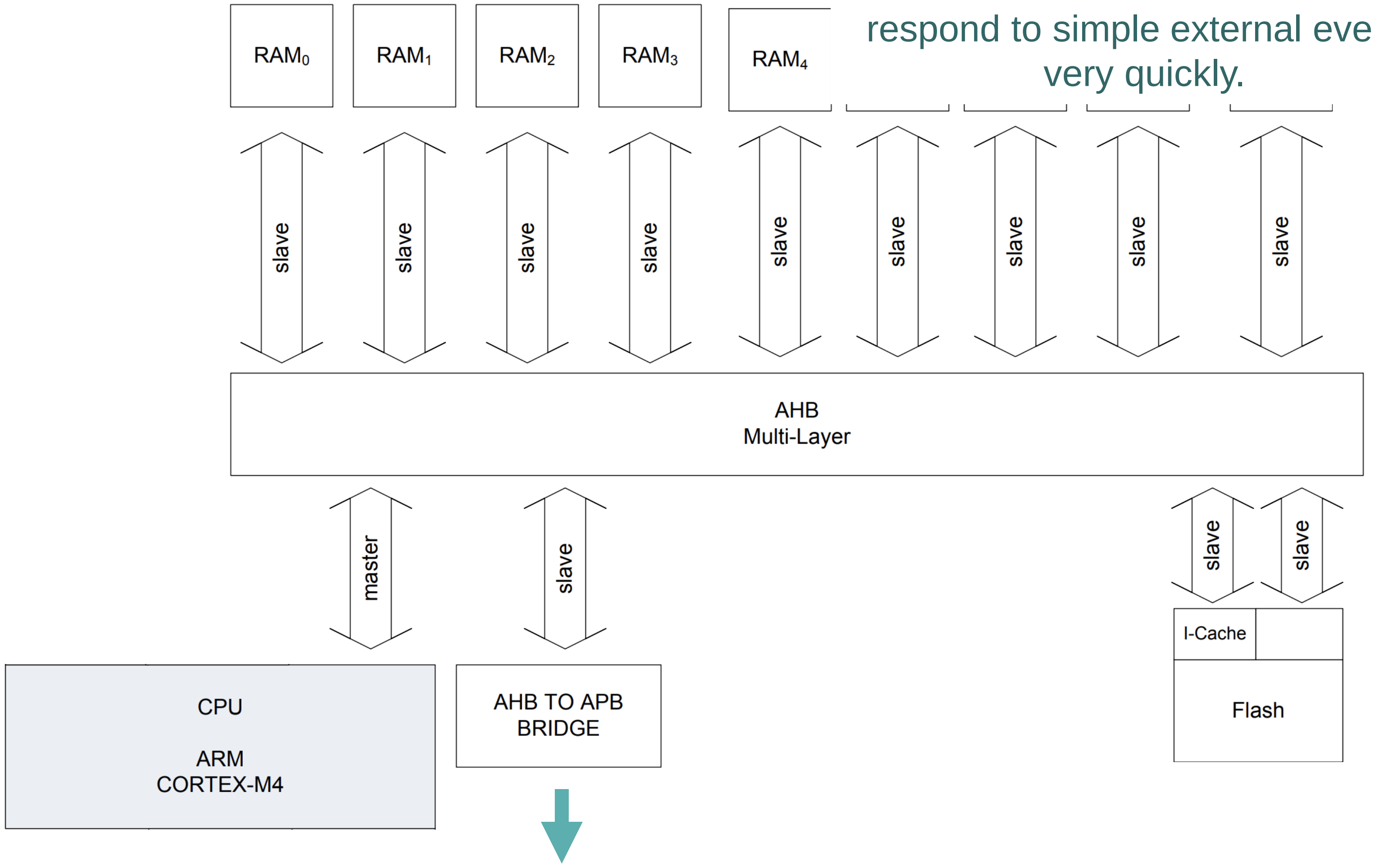
Why is GPIO special?



To Peripherals

nRF52832 Block Diagram

Why is GPIO special?



Because we want to be able to respond to simple external events very quickly.

To Peripherals

Announcements Placeholder

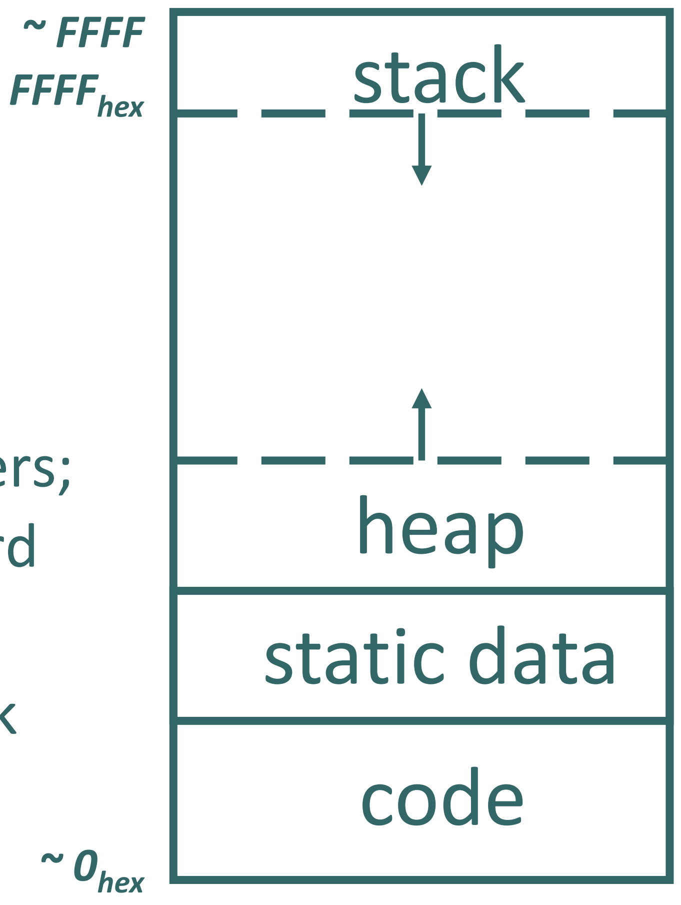
Outline

- Memory Overview
  - Types of Memory
  - Memory Hierarchy
- Embedded Systems Memory
  - Memory Maps
  - Memory-Mapped I/O
  - Lab Hardware Examples
- Memory Organization
  - Stacks & Heaps
  - Code examples

MEMORY LAYOUT

C Memory Layout

- Program's *address space* contains 4 regions:
  - Stack: local variables, grows downward
  - Heap: space requested via `malloc()` and used with pointers; resizes dynamically, grows upward
  - Static Data: global and static variables, does not grow or shrink
  - Code: loaded when program starts, does not change



Where Do the Variables Go?

- Declared outside a function:

  Static Data

- Declared inside a function:

  Stack

  - `main()` is a function
  - Freed when function returns

- Dynamically allocated:

  Heap

  - i.e. malloc

```
#include <stdio.h>

int varGlobal;

int main() {
  int varLocal;
  int *varDyn =
    malloc(sizeof(int));
}
```

Statically-Allocated Memory in C

```
char x;
void foo(void) {
     x = 0x20;
     …
}
```

Compiler chooses what address to use for x, and the variable is accessible across procedures. The variable's lifetime is the total duration of the program execution.

Statically-Allocated Memory with Limited Scope

```
void foo(void) {
      static char y;
      y = 0x20;
      …
}
```

Compiler chooses what address to use for y, but the variable is meant to be accessible only in foo(). The variable’s lifetime is the total duration of the program execution (values persist across calls to foo()).

Statically-Allocated Memory with Limited Scope

```
char x;
void foo(void) {
      x = 0x20;
      …
}
```

```
void foo(void) {
      static char y;
      y = 0x20;
      …
}
```

What is the difference between x and y when code is loaded on the device?

Statically-Allocated Memory with Limited Scope

```
char x;
void foo(void) {
     x = 0x20;
     …
}
```

```
void foo(void) {
     static char y;
     y = 0x20;
     …
}
```

What is the difference between x and y when code is loaded on the device?

There is no difference! Accessibility of a variable is a compile-time concept, not a run-time one.

Variables on the Stack (“automatic variables”)

```
void foo(void) {
     char x;
     x = 0x20;
     …
}
```

When the procedure is called, x is assigned an address on the stack (by decrementing the stack pointer). When the procedure returns, the memory is freed (by incrementing the stack pointer). The variable persists only for the duration of the call to foo().

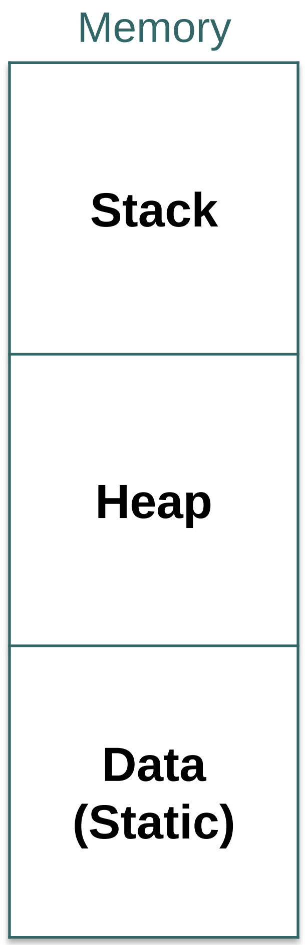
Memory Layout Question 1

Assume a 32-bit ARM microcontroller

```
char x;
void foo(void) {
   x = 0x20;
   …
}
```

How many bytes does **x** take,

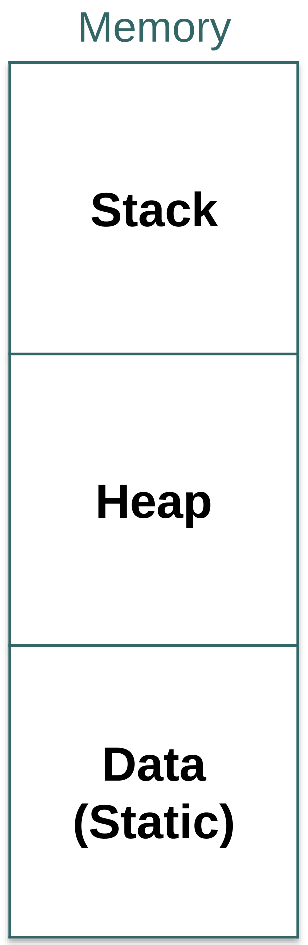
and in which section of the memory layout?



Memory Layout Question 1

Assume a 32-bit ARM microcontroller

```
char x;
void foo(void) {
   x = 0x20;
   …
}
```



How many bytes does **x** take,

and in which section of the memory layout?

1 byte in the data section

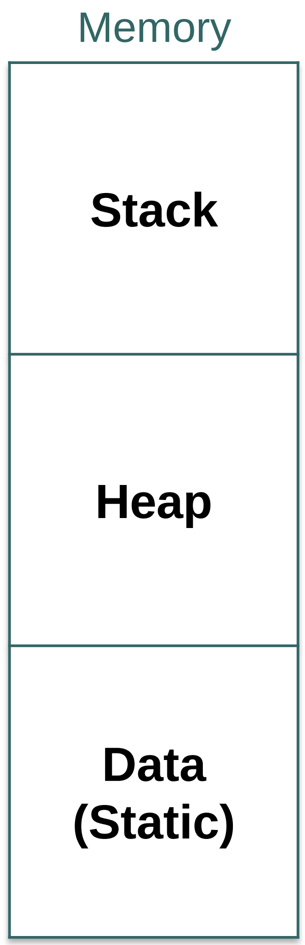
Memory Layout Question 2

Assume a 32-bit ARM microcontroller

```
char* x;
void foo(void) {
  x = 0x20;
  …
}
```

How many bytes does **x** take,

and in which section of the memory layout?



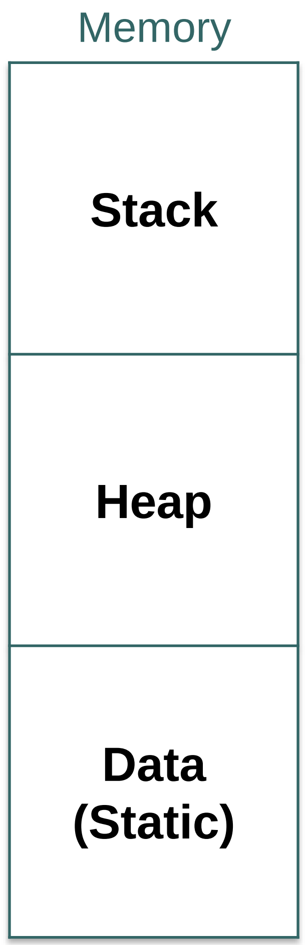
Memory Layout Question 2

Assume a 32-bit ARM microcontroller

```
char* x;
void foo(void) {
   x = 0x20;
   …
}
```

How many bytes does **x** take,
and in which section of the memory layout?

4 bytes in the data section (for 32-bit processors)

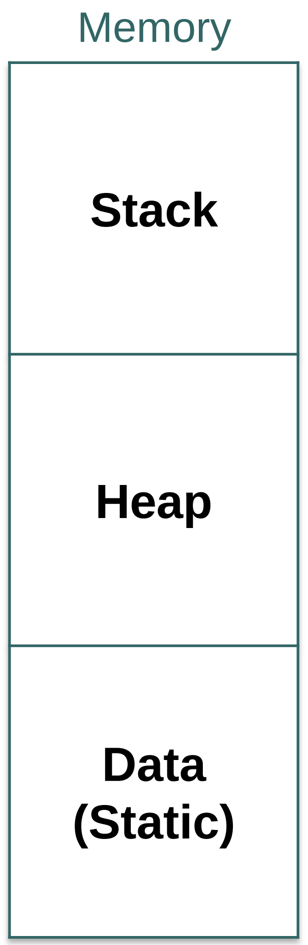


Memory Layout Question 3

Assume a 32-bit ARM microcontroller

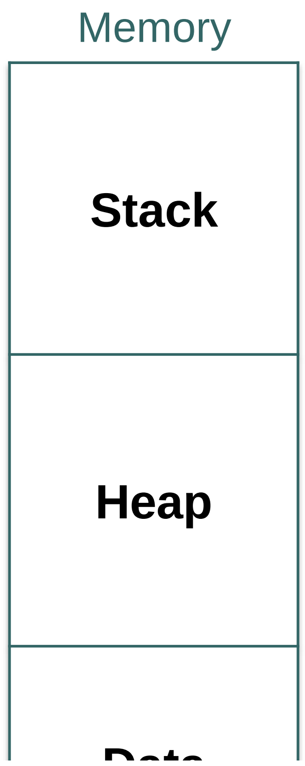
```
int a;
void foo(short b) {
	static int c = 3;
	char* d;
	d = (char*) malloc(4);
	printf("Hello EECS149\n");
}
```

What about **a**, **b**, **c**, and **d**?



Memory Layout Question 3

Assume a 32-bit ARM microcontroller



```
int a;
void foo(short b) {
      static int c = 3;
      char* d;
      d = (char*) malloc(4);
      printf(“Hello EECS149\n”);
}
```

What about **a**, **b**, **c**, and **d**?

a – 4 bytes in the data section
b – 2 bytes in the stack
c – 4 bytes in the data section
d – 4 bytes in the stack
contents of d – 4 bytes in the heap

Find the flaw in this program

(begin by thinking about where each variable is allocated)

```
int x = 2;

int* foo(int y) {
  int z;
  z = y * x;
  return &z;
}

int main(void) {
  int* result = foo(10);
  ...
}
```

Solution: Find the flaw in this program

```
int x = 2;

int* foo(int y) {
  int z;
  z = y * x;
  return &z;
}

int main(void) {
  int* result = foo(10);
  ...
}
```

statically allocated: compiler assigns a memory location.

arguments on the stack

automatic variables on the stack

program counter, argument 10, and z go on the stack (and possibly more, depending on the compiler).

The procedure foo() returns a pointer to a variable on the stack. What if another procedure call (or interrupt) occurs before the returned pointer is de-referenced?

The embedded systems perspective

The embedded systems perspective

The Heap is EVIL!!!!

Why?

Dynamically-Allocated Memory
The Heap

An operating system typically offers a way to dynamically allocate memory on a “heap”.

Memory management (malloc() and free()) can lead to many problems with embedded systems:

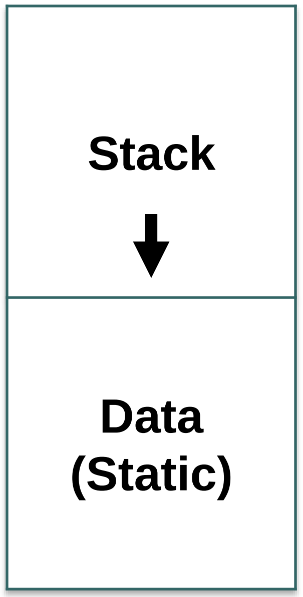
- Memory leaks (allocated memory is never freed)
- Memory fragmentation (allocatable pieces get smaller)

Automatic techniques (“garbage collection”) often require stopping everything and reorganizing the allocated memory. This is deadly for real-time programs.

The embedded systems perspective

How do we handle memory faults?

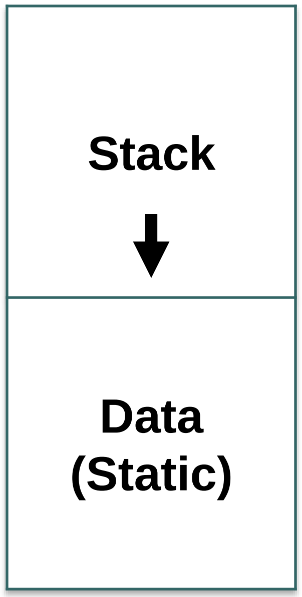
What if the stack grows too much?



The embedded systems perspective

How do we handle memory faults?

What if the stack grows too much?



Nothing stops it!

Hopefully the failure is easy to understand…

Conclusion

Memories (non-volatile and volatile) are essential to computers

Embedded systems use a simplified memory architecture with only registers, SRAM, and Flash (no caches)

Memory-Mapped I/O allows interactions with embedded peripherals to look like normal memory accesses

Software creates Stack, Heap, Static, and Code sections in memory

BONUS SLIDES ON CACHES

See CS61C Lectures on Caches for more information

https://inst.eecs.berkeley.edu/~cs61c/su19/#lectures

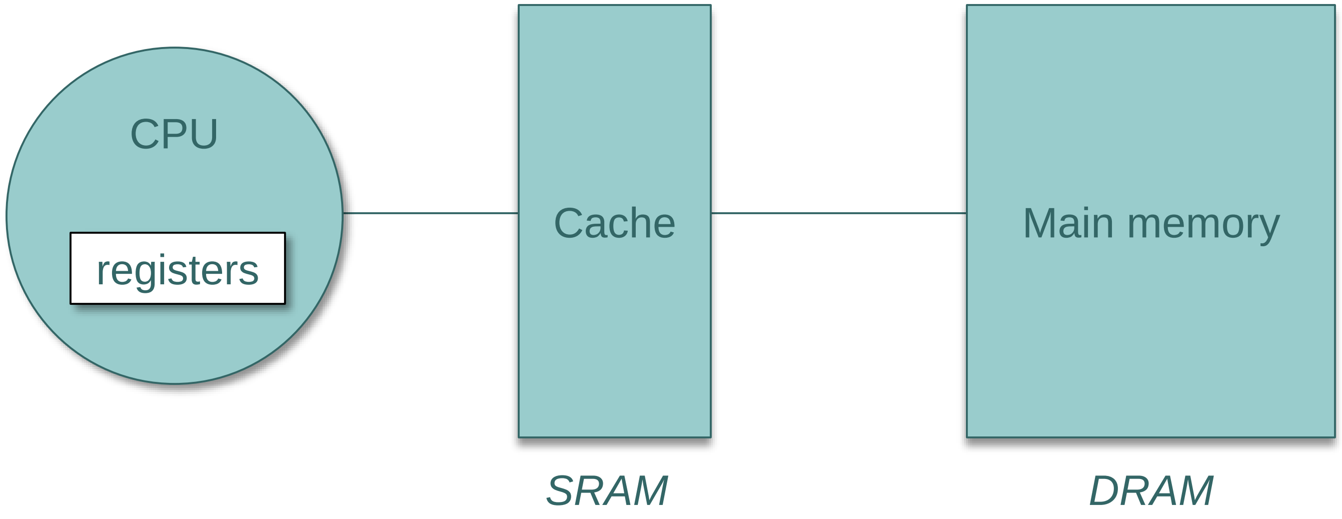
Caches

- How do we read or write to a cache?

Caches

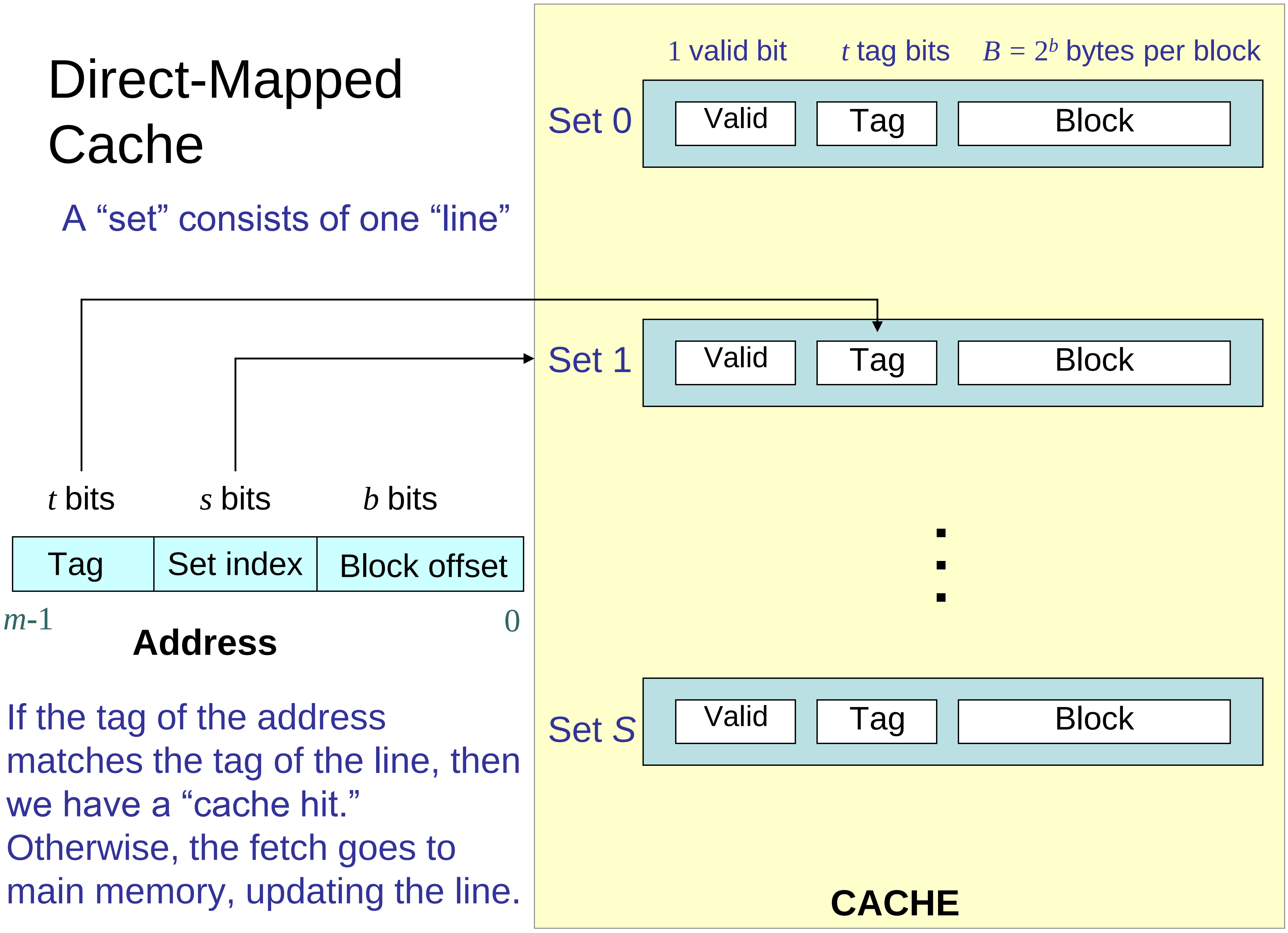
- How do we read or write to a cache?
  - You don't! Caches are automatic.

Personal Computer Example



Direct-Mapped Cache

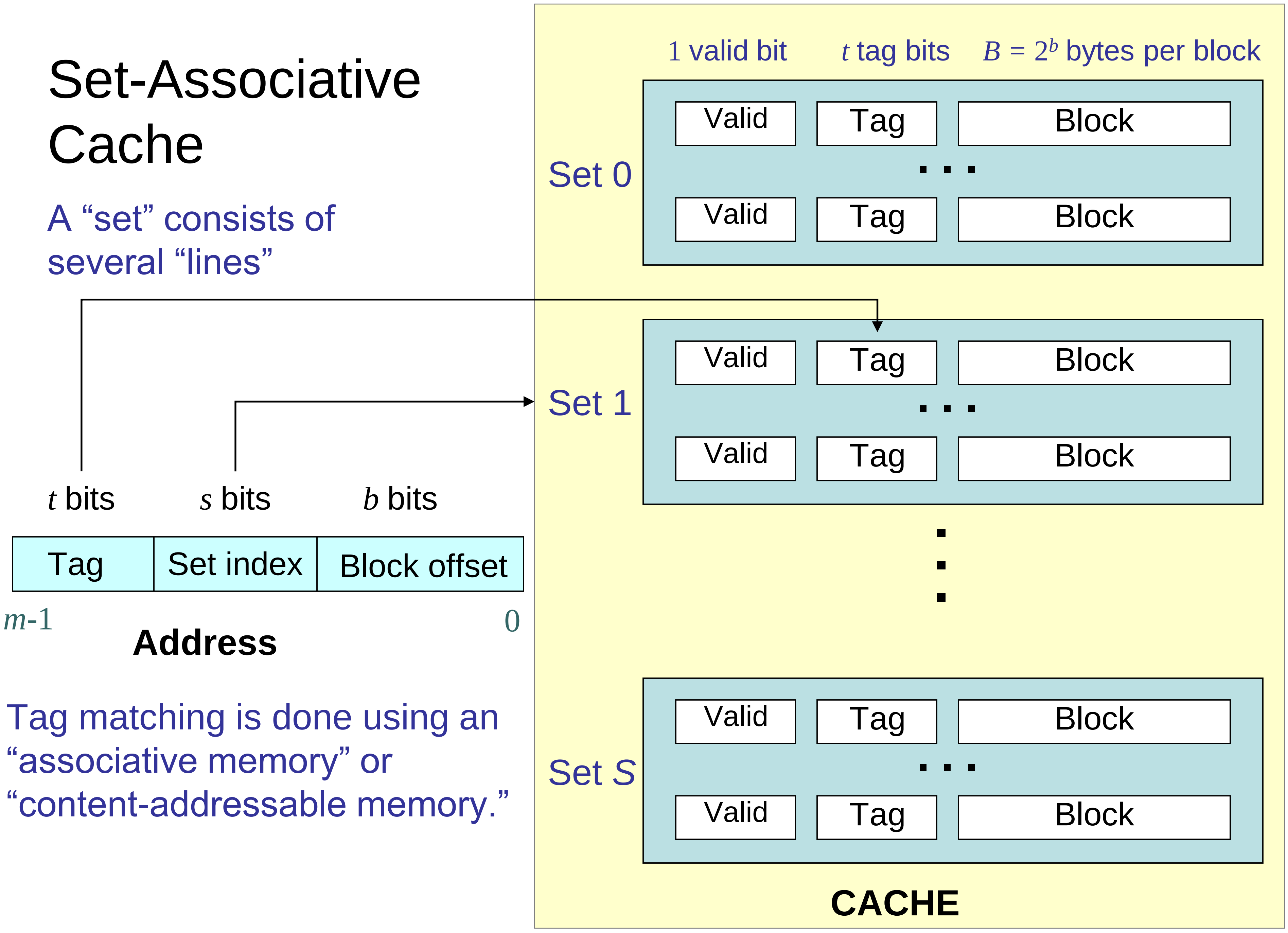
A "set" consists of one "line"



If the tag of the address matches the tag of the line, then we have a "cache hit." Otherwise, the fetch goes to main memory, updating the line.

Set-Associative Cache

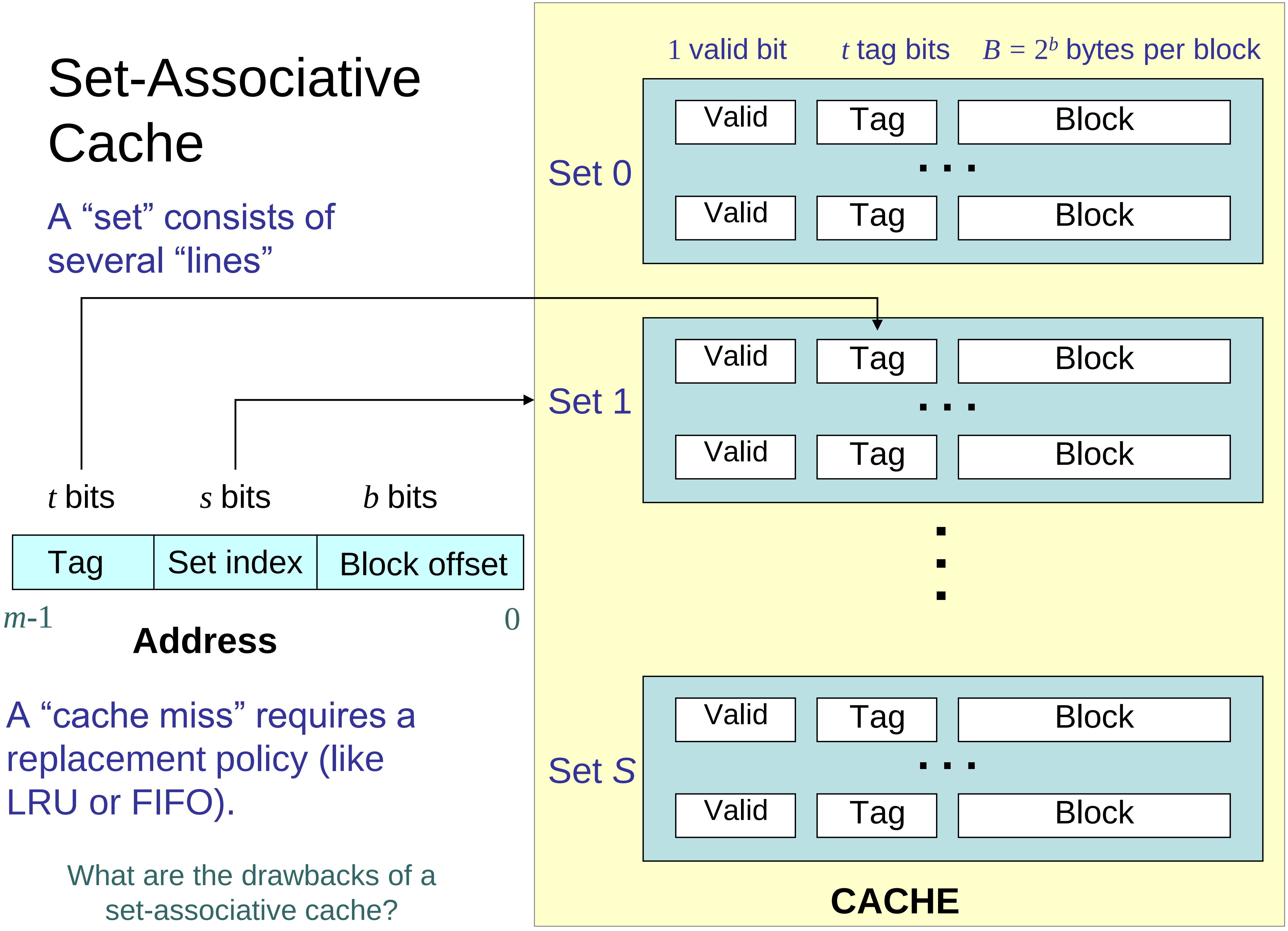
A "set" consists of several "lines"



Tag matching is done using an "associative memory" or "content-addressable memory."

Set-Associative Cache

A “set” consists of several “lines”

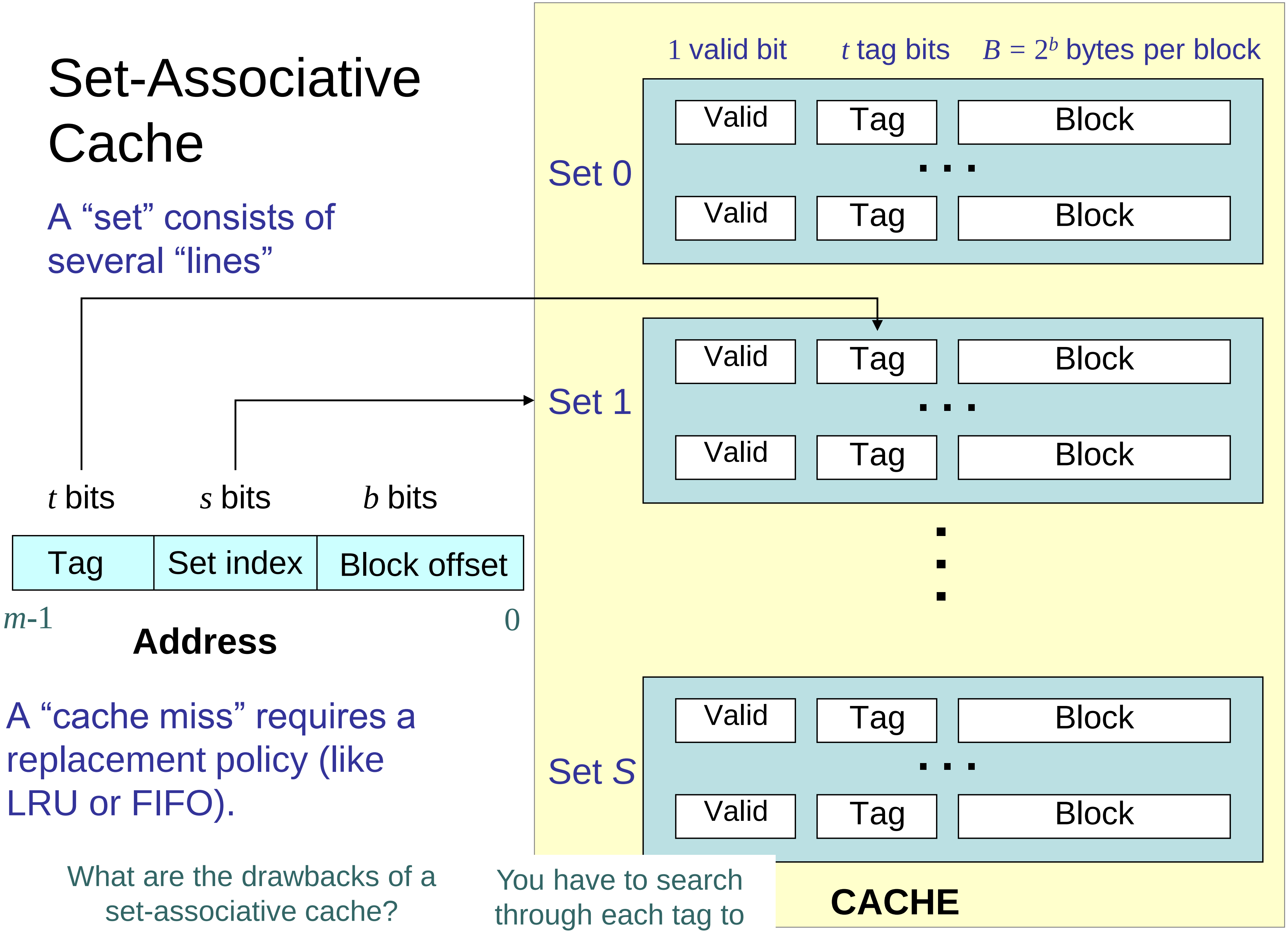


A “cache miss” requires a replacement policy (like LRU or FIFO).

What are the drawbacks of a set-associative cache?

Set-Associative Cache

A "set" consists of several "lines"



A "cache miss" requires a replacement policy (like LRU or FIFO).

What are the drawbacks of a set-associative cache?

You have to search through each tag to check for your data