



Introduction to Embedded Systems



Branden Ghena

UC Berkeley

EECS 149/249A

Fall 2018

Content by Prabal Dutta, Edward Lee, and others

© 2008-2017: E. A. Lee, A. L. Sangiovanni-Vincentelli, S. A. Seshia.
All rights reserved.

Chapter 9: Memory Architectures

Homework Announcement

Homework 1 is available on bcourses

It's due in one week on Thursday 9/6 at 11:59 PM

Remember

Homework is 15% of your final grade

And the best 6 out of the 7 homeworks are counted

Outline

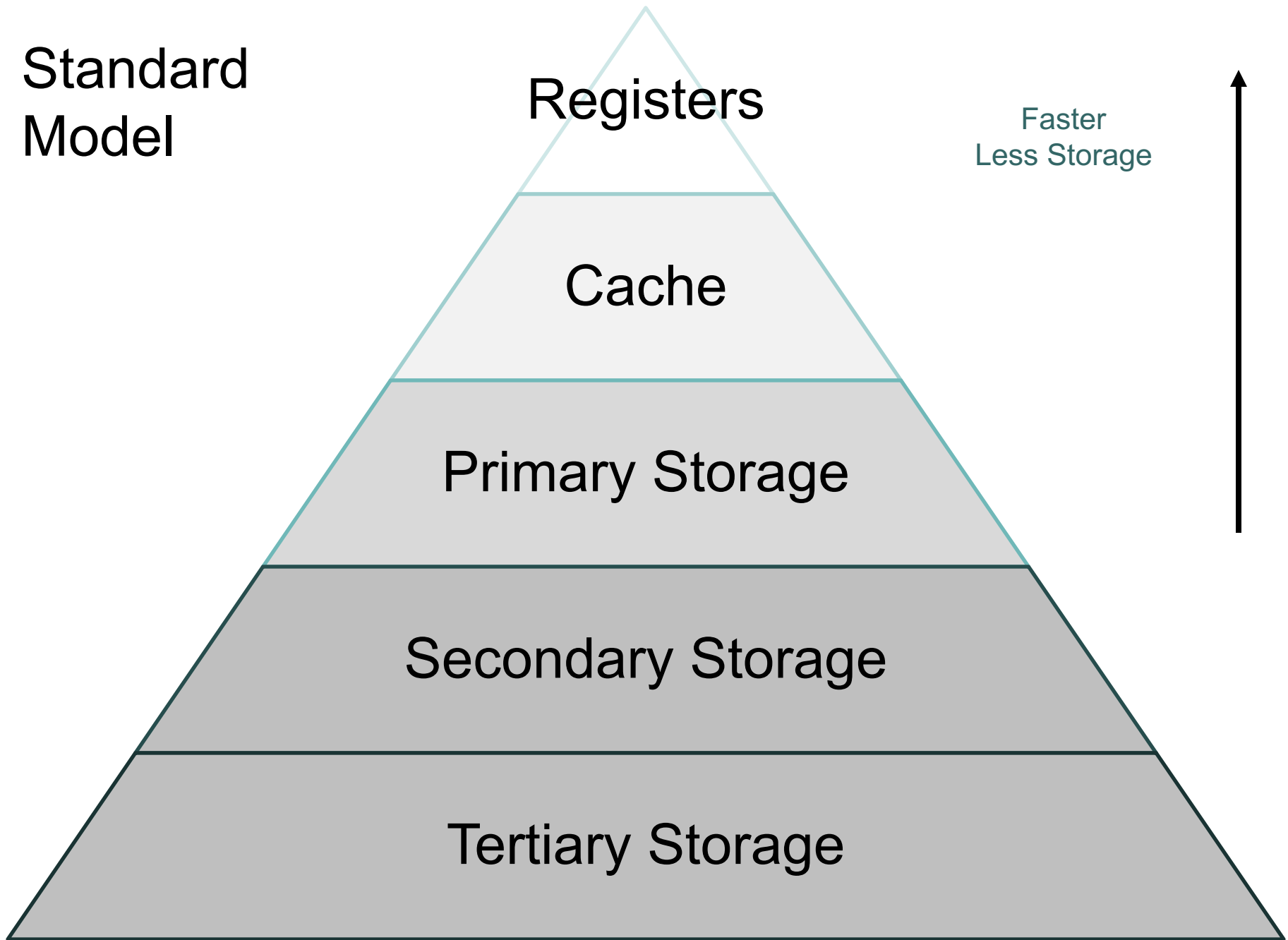
- Memory Hierarchy
 - Types of memory
- Using Memory
 - Caches
 - Memory Maps
 - Memory-mapped I/O
- Lab Hardware
 - nRF52832 example
- Software Organization of Memory
 - Stacks & Heaps
 - Code examples

MEMORY HIERARCHIES

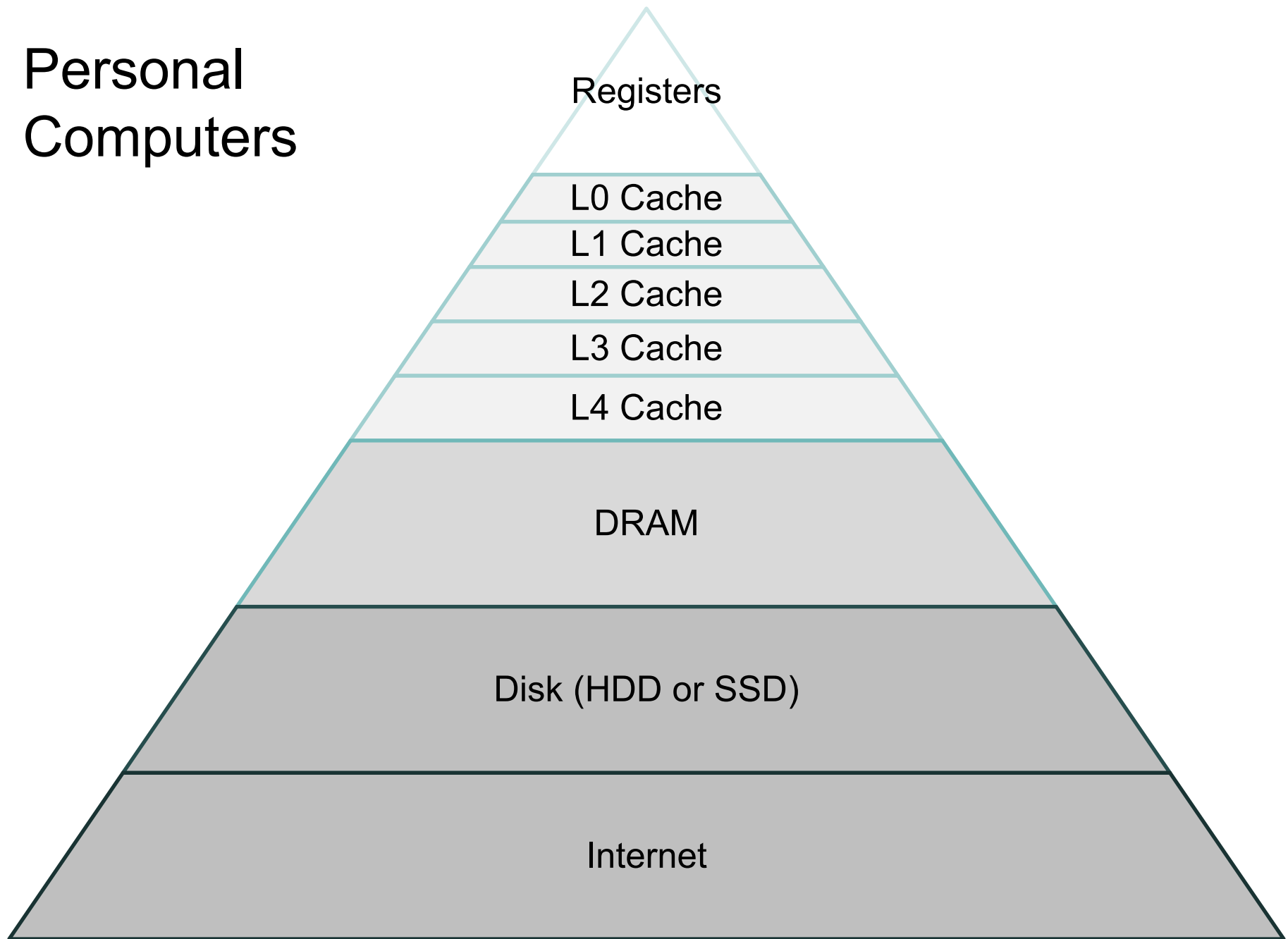
Memory Hierarchy

- 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?
- How do we implement each of these categories?

Standard Model



Personal Computers



Microcontrollers

Registers

~~Cache~~

SRAM

Flash

Memory Classes

Two major types

1. Memory which is temporary
2. Memory which is permanent

Volatile Memory

Loses contents when power is off.



Volatile Memory

Loses contents when power is off.



- Mercury Delay Line
 - 18-bit memory for UNIVAC

Volatile Memory

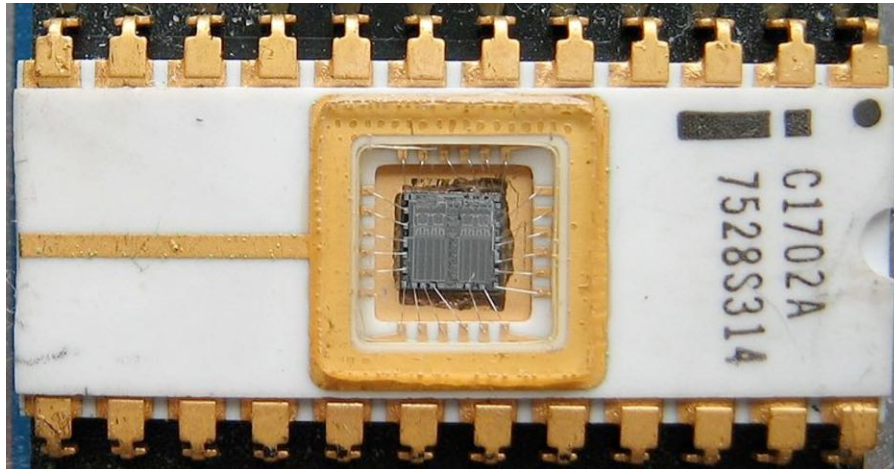
Loses contents when power is off.

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

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



Non-Volatile Memory

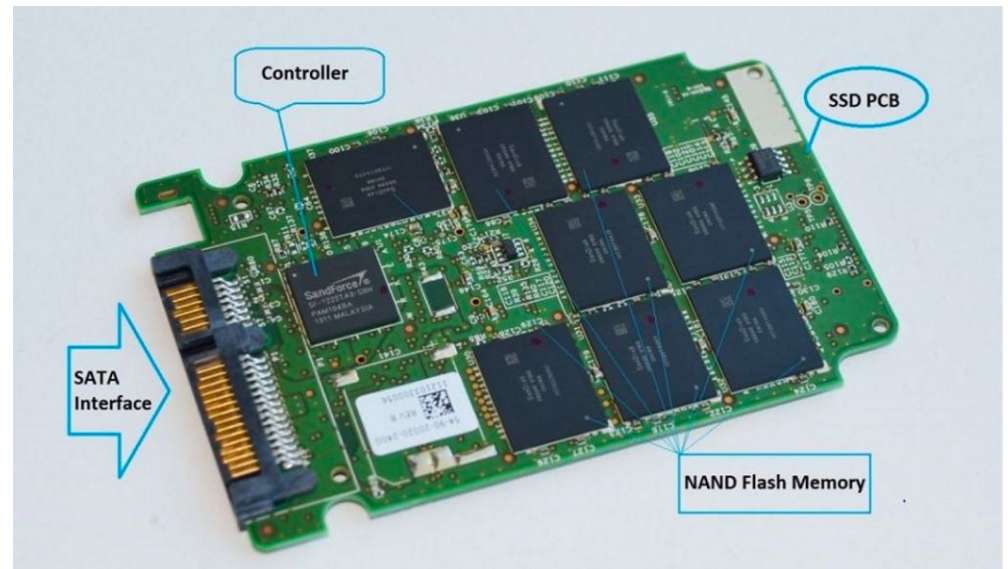
Preserves contents when power is off

- **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

- **Disk drives**

- Not as well suited for embedded systems



Example:

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

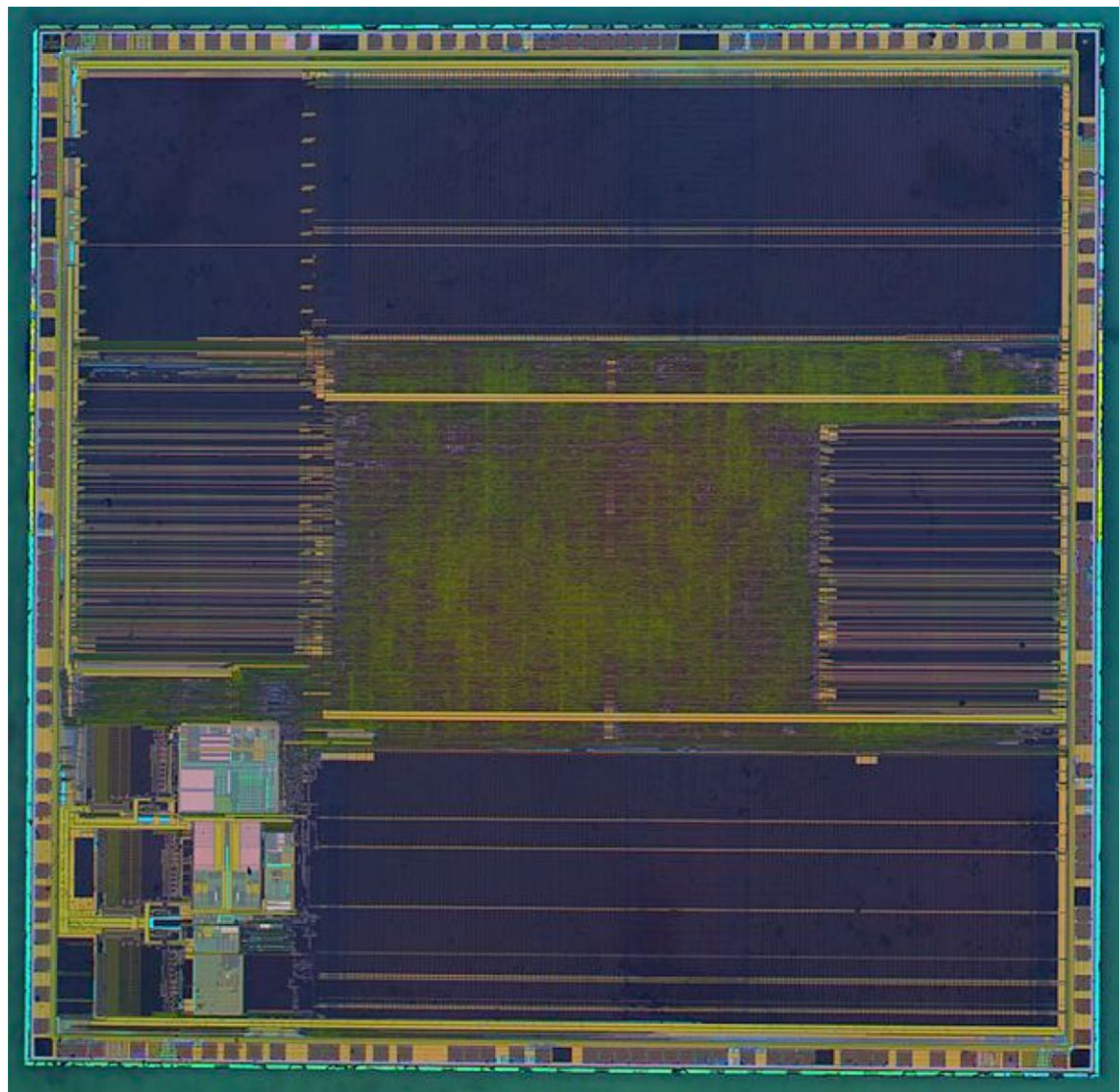


Image from Wikimedia Commons

Which part is the
memory?

Example:

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

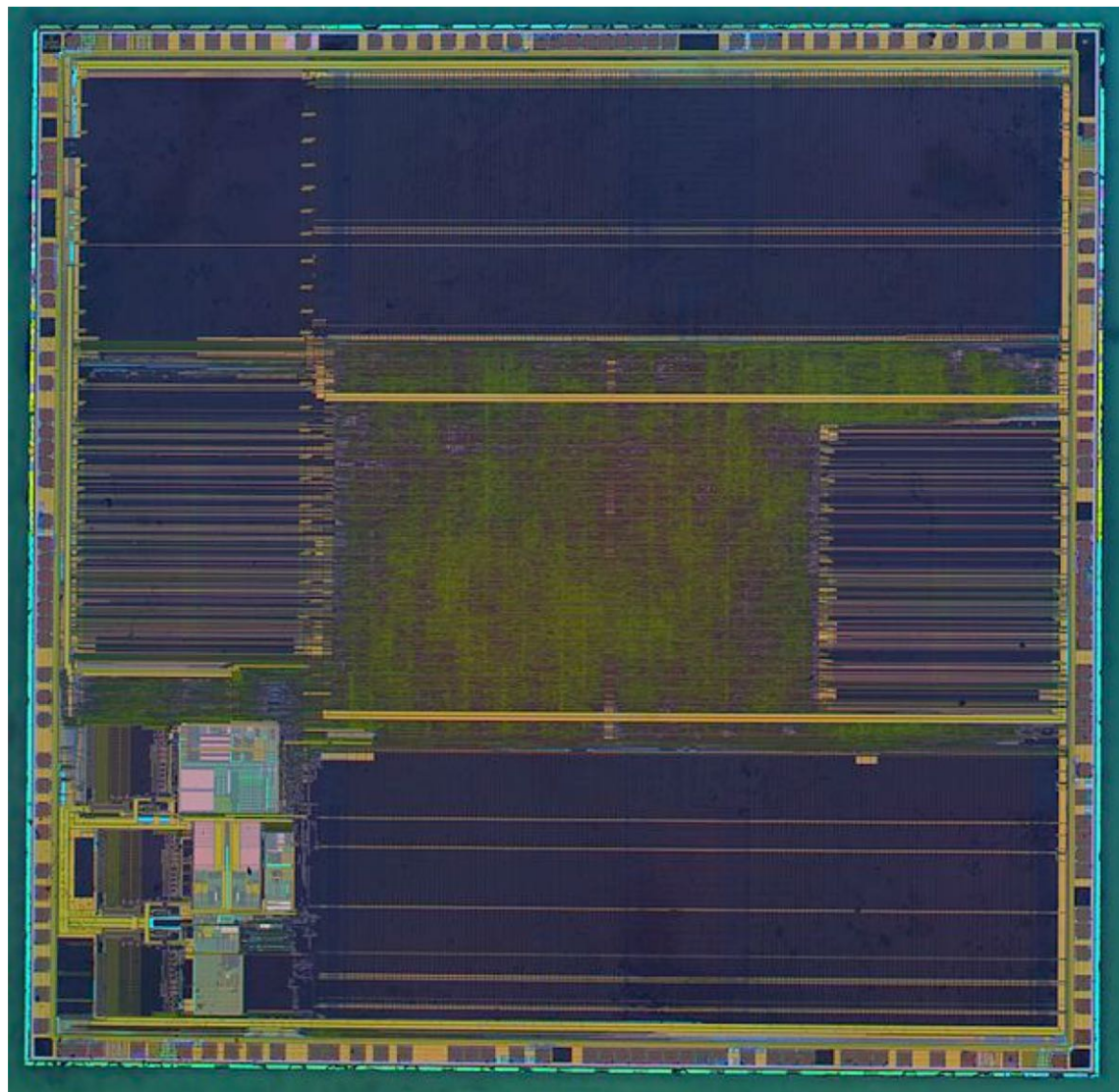


Image from Wikimedia Commons

Which part is the
memory?

Just about everything but
the bottom right corner

USING MEMORY

Registers

- How do we read or write to registers?

Registers

- How do we read or write to registers?
 - Most assembly instructions!

MOV <Destination Register>, <Source Register>

MOV <Destination Register>, #<Literal>

ADD <Destination Register>, <Source 1>, <Source 2>

Registers don't have memory addresses

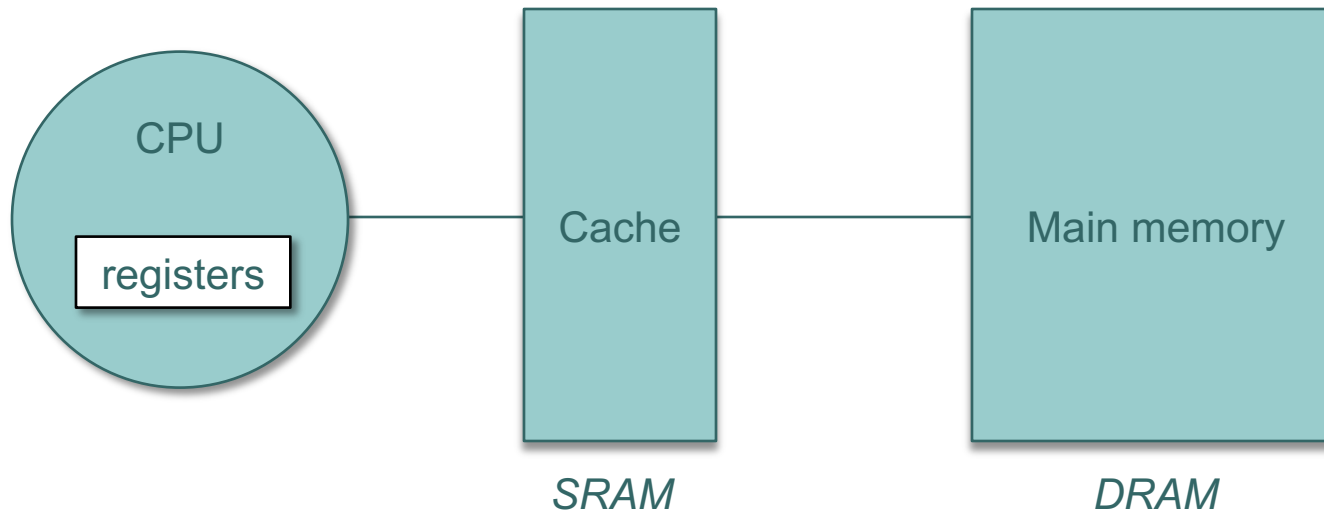
Caches

- How do we read or write to a cache?

Caches

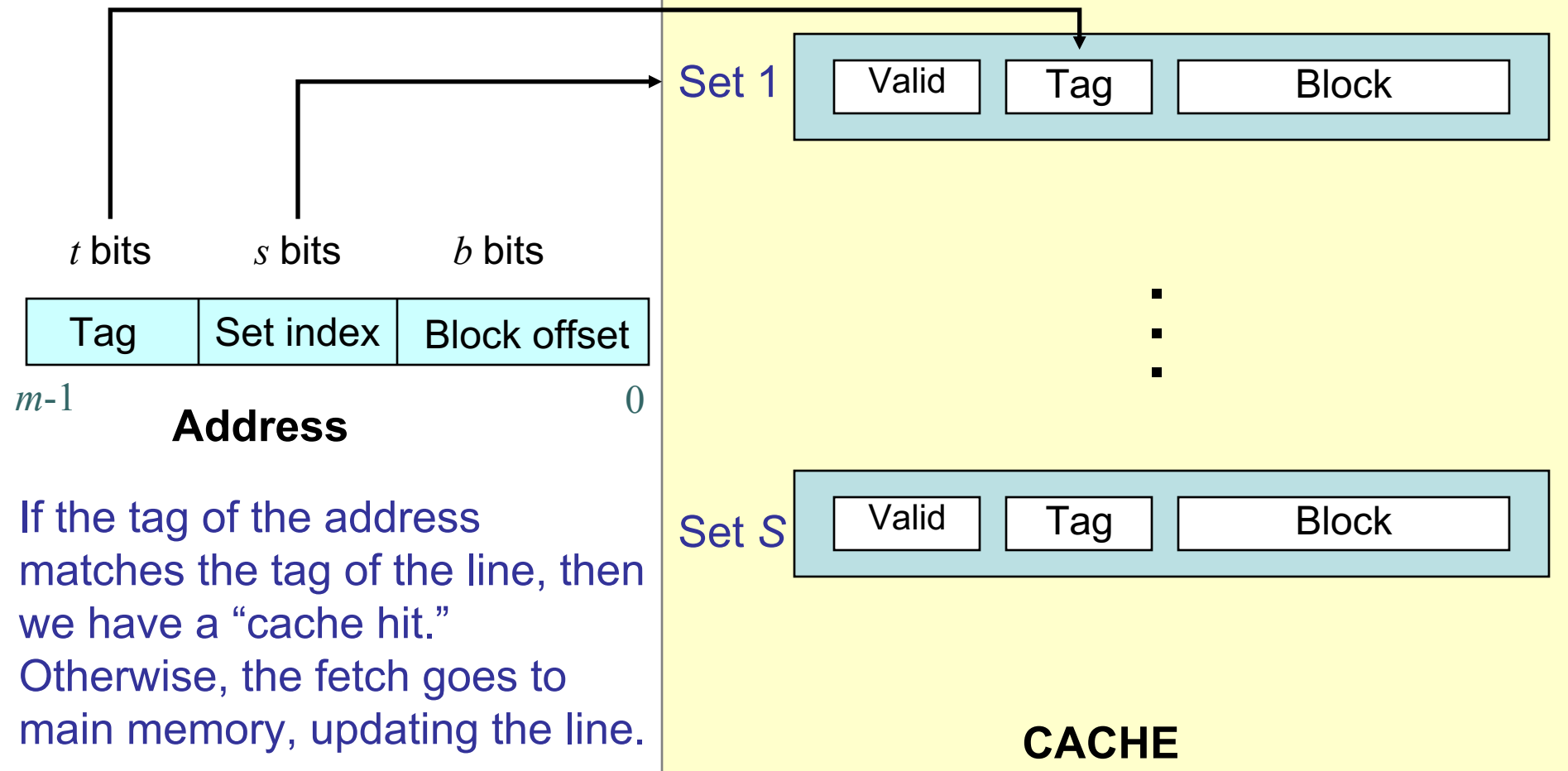
- How do we read or write to a cache?
 - You don't! Caches are automatic.

Personal Computer Example



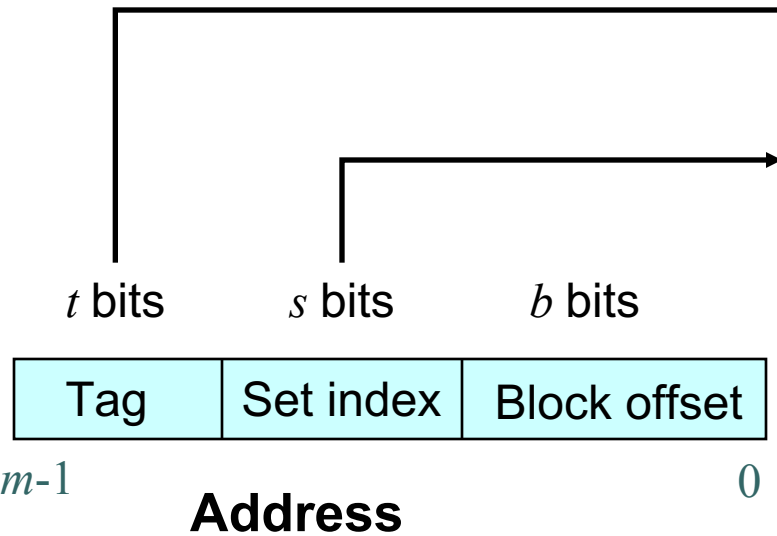
Direct-Mapped Cache

A “set” consists of one “line”

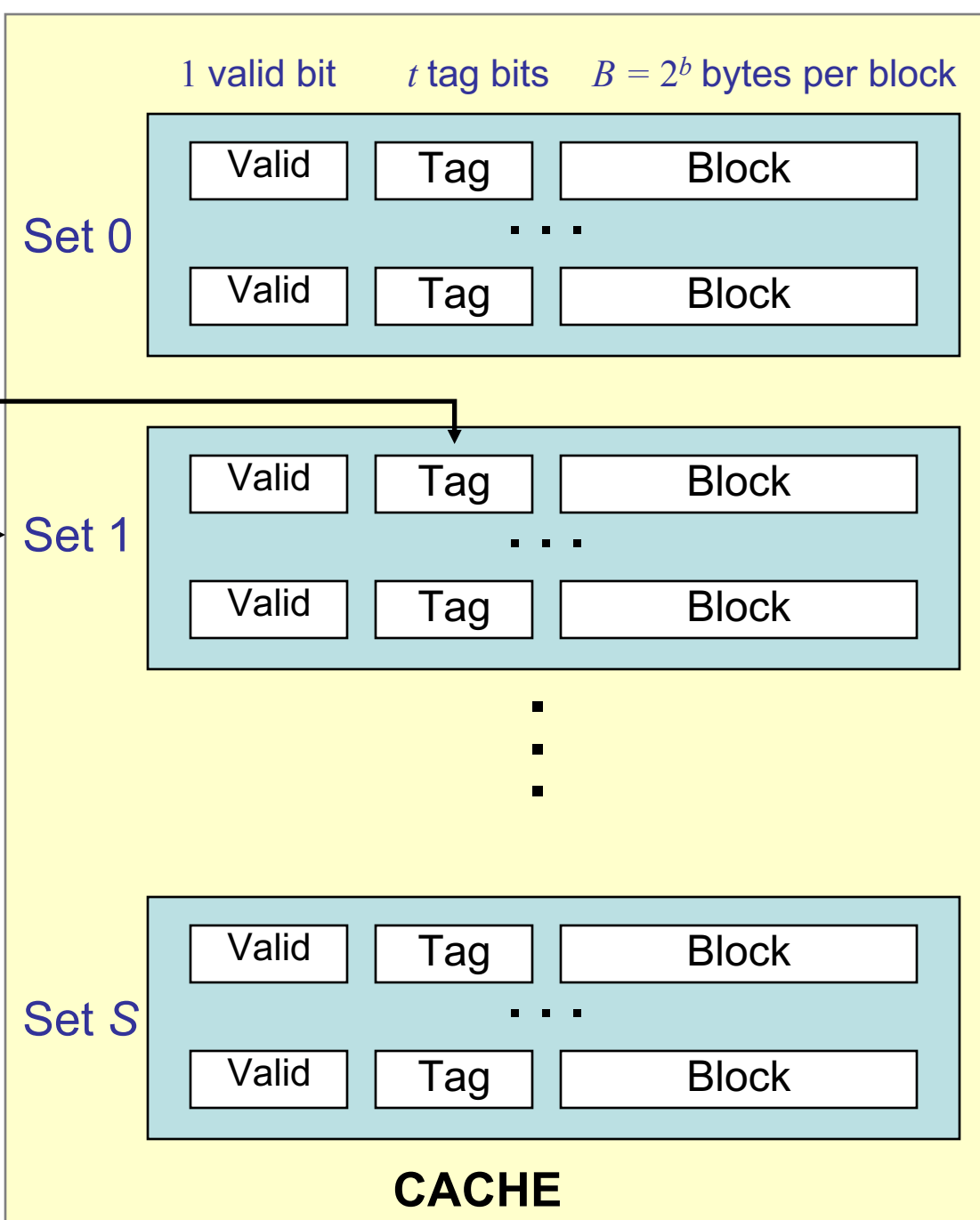


Set-Associative Cache

A “set” consists of several “lines”

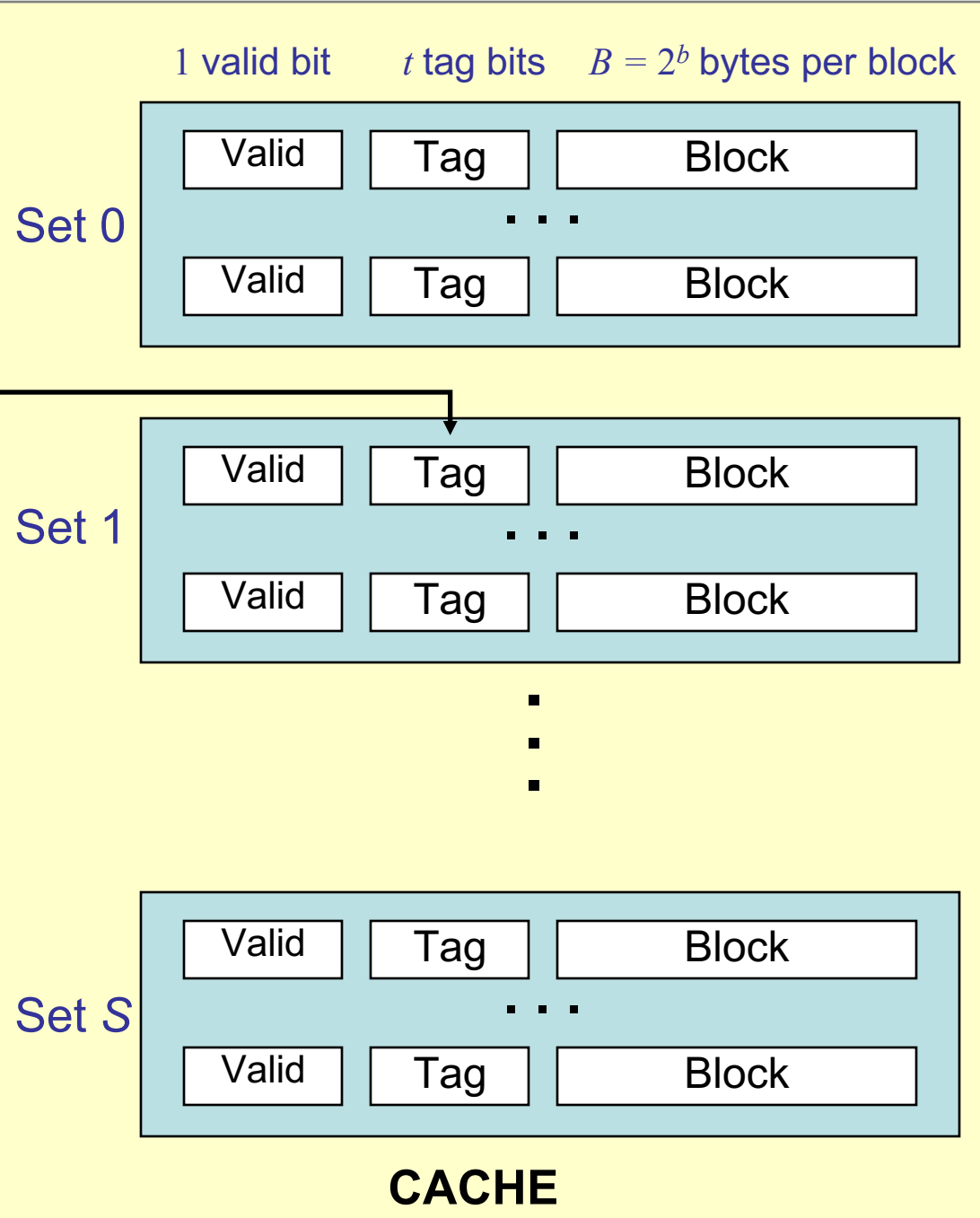
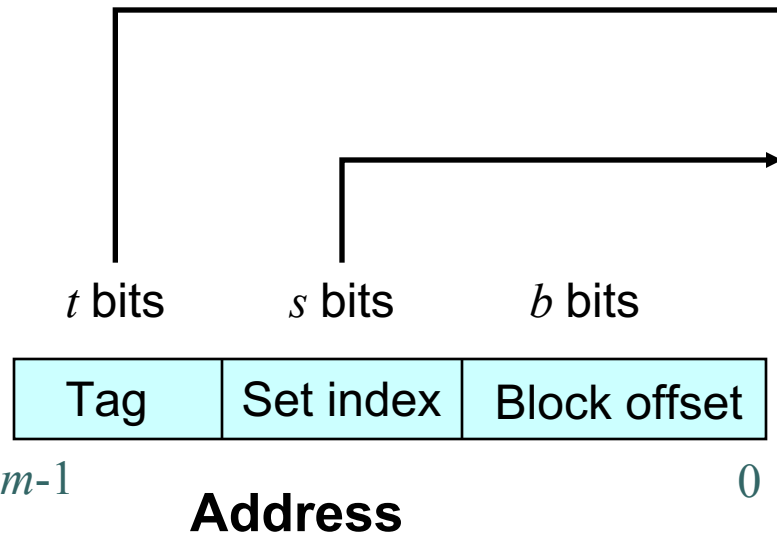


Tag matching is done using an “associative memory” or “content-addressable memory.”



Set-Associative Cache

A “set” consists of several “lines”

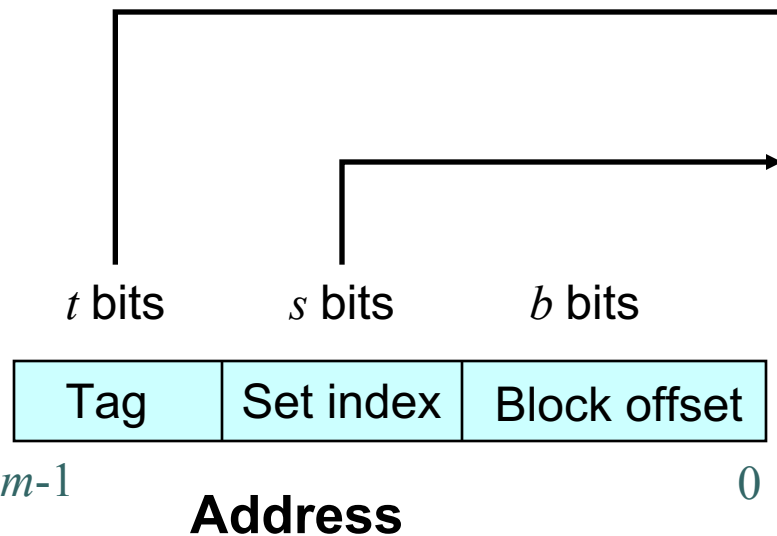


A “cache miss” requires a replacement policy (like LRU or FIFO).

What are the drawbacks of a set-associative cache?

Set-Associative Cache

A “set” consists of several “lines”



Set 0

Set 1

Set S

1 valid bit t tag bits $B = 2^b$ bytes per block

Valid Tag Block

...

Valid Tag Block

...

Valid Tag Block

...

Valid Tag Block

...

Valid Tag Block

CACHE

What are the drawbacks of a set-associative cache?

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

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.

Main Memory (and further)

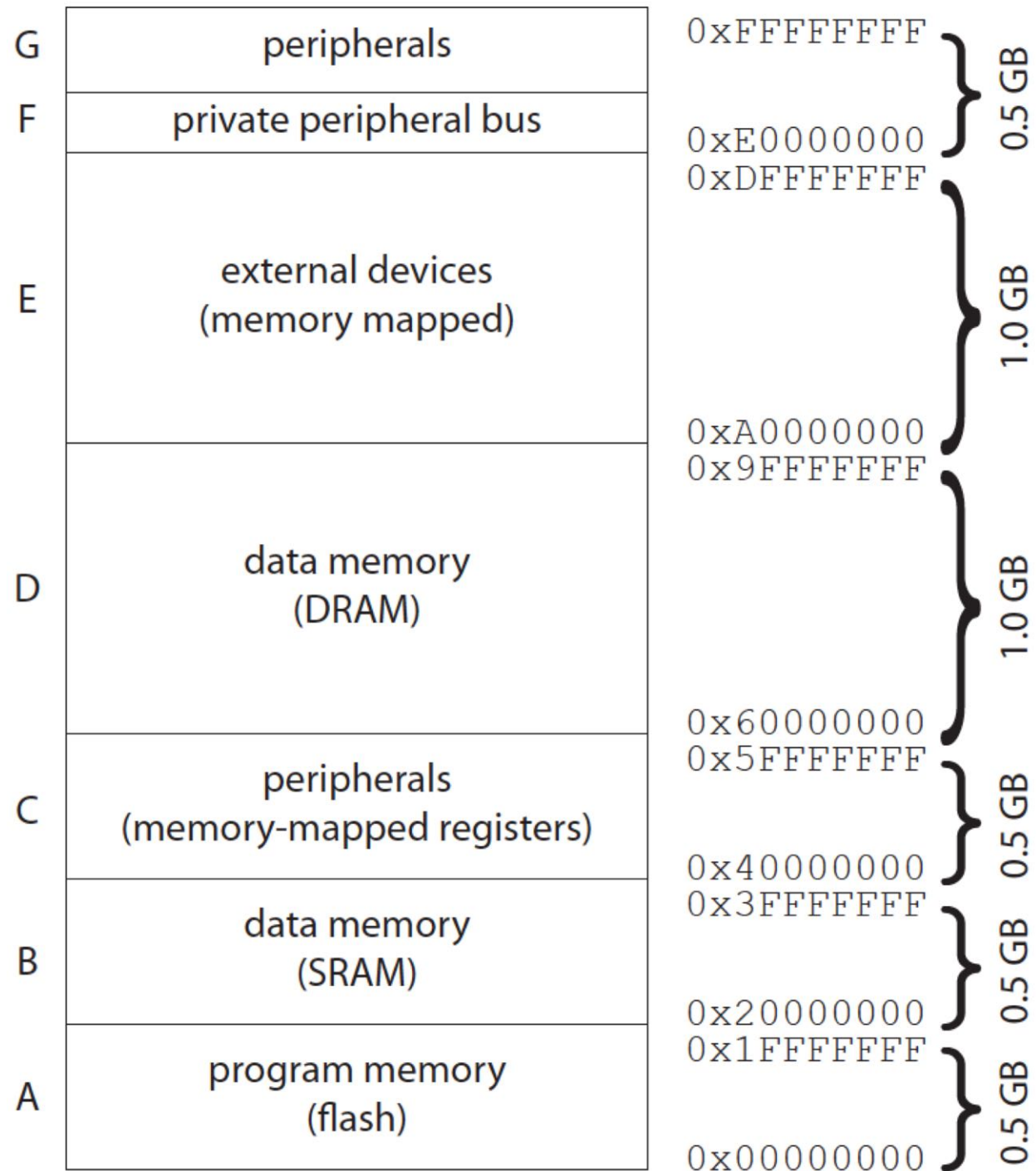
- RAM and Disk are accessed through reads and writes to addresses
- Which addresses are valid and point to which thing depend on the memory “map” of the system

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

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

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, I²C
- How do they access the peripherals?
- Why not create special assembly functions to access them?

Things That Aren't Memory

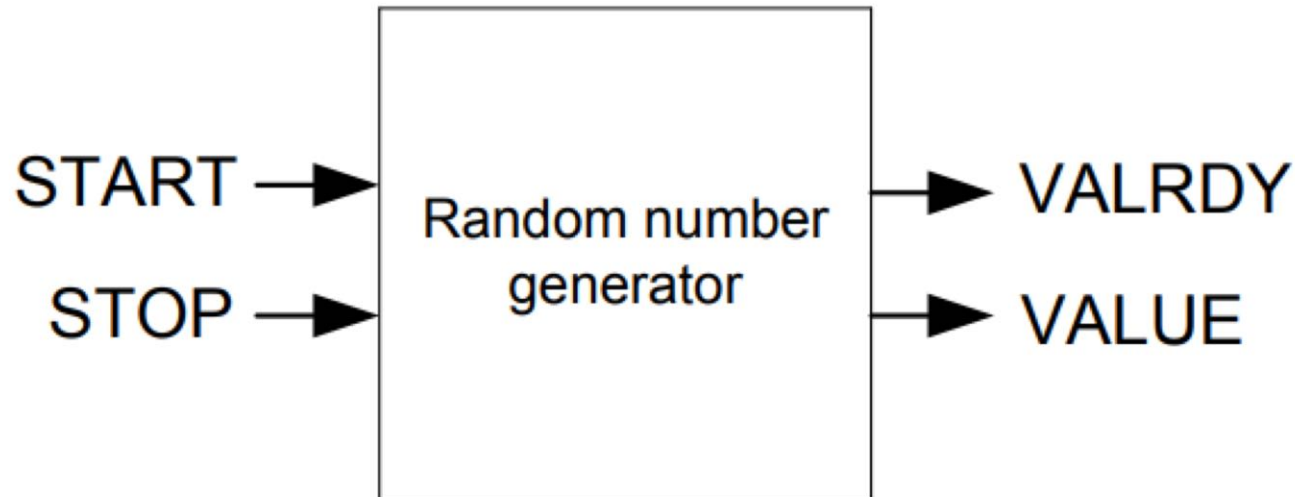
- Microcontrollers have a lot of peripherals
 - General Purpose I/O (GPIO) pins
 - Analog to Digital Converters
 - Digital to Analog Converters
 - Pulse-Width Modulation Generators
 - Timers
 - Various communication buses: UART, SPI, I²C
- How do they access the peripherals?
 - With memory reads and 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
<i>SHORTS</i>	0x200	Shortcut register
<i>INTENSET</i>	0x304	Enable interrupt
<i>INTENCLR</i>	0x308	Disable interrupt
<i>CONFIG</i>	0x504	Configuration register
<i>VALUE</i>	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
<i>SHORTS</i>	0x200	Shortcut register
<i>INTENSET</i>	0x304	Enable interrupt
<i>INTENCLR</i>	0x308	Disable interrupt
<i>CONFIG</i>	0x504	Configuration register
<i>VALUE</i>	0x508	Output random number

Example: Random Number Generator

26.3 Registers

Table 45: Instances

Base address	Peripheral	Instance
0x4000D000	RNG	RNG

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)

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
<i>SHORTS</i>	0x200	Shortcut register
<i>INTENSET</i>	0x304	Enable interrupt
<i>INTENCLR</i>	0x308	Disable interrupt
<i>CONFIG</i>	0x504	Configuration register
<i>VALUE</i>	0x508	Output random number

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
<i>SHORTS</i>	0x200
<i>INTENSET</i>	0x304
<i>INTENCLR</i>	0x308
<i>CONFIG</i>	0x504
<i>VALUE</i>	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	
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
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);
```

LAB HARDWARE

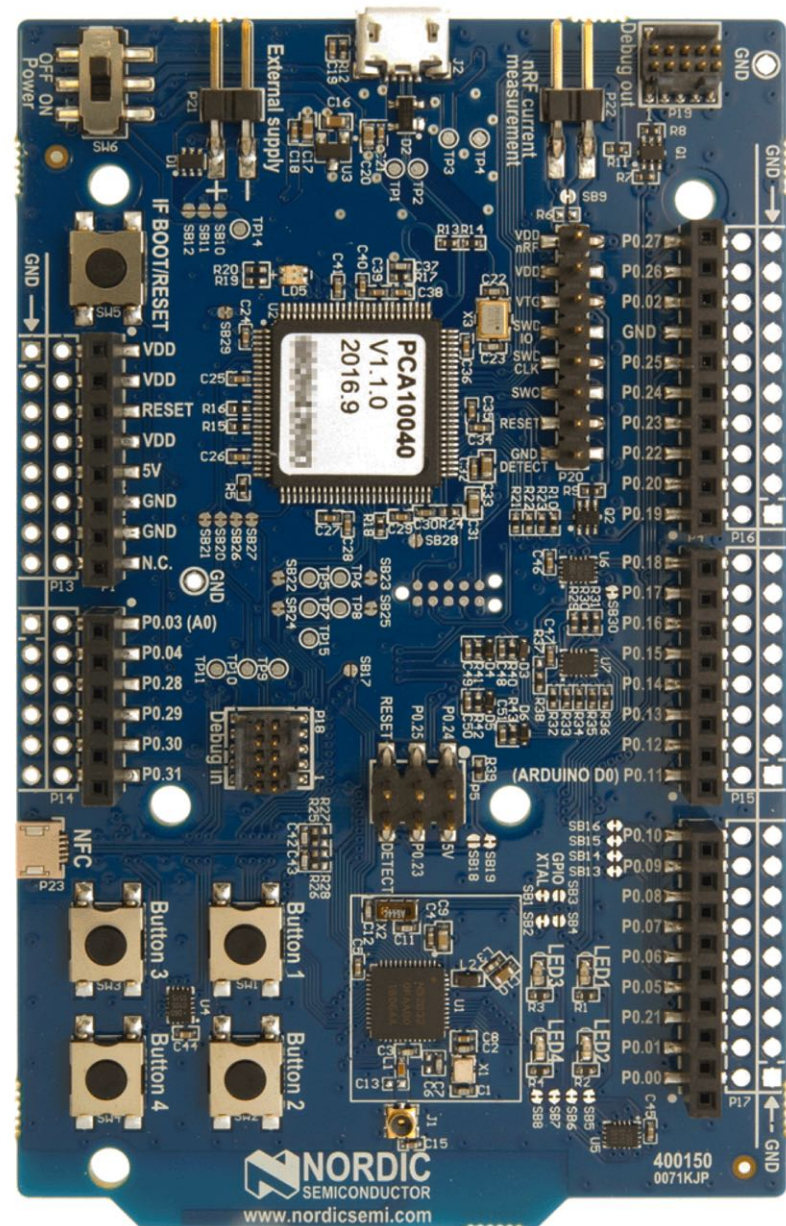
nRF52832 Microcontroller

Processor

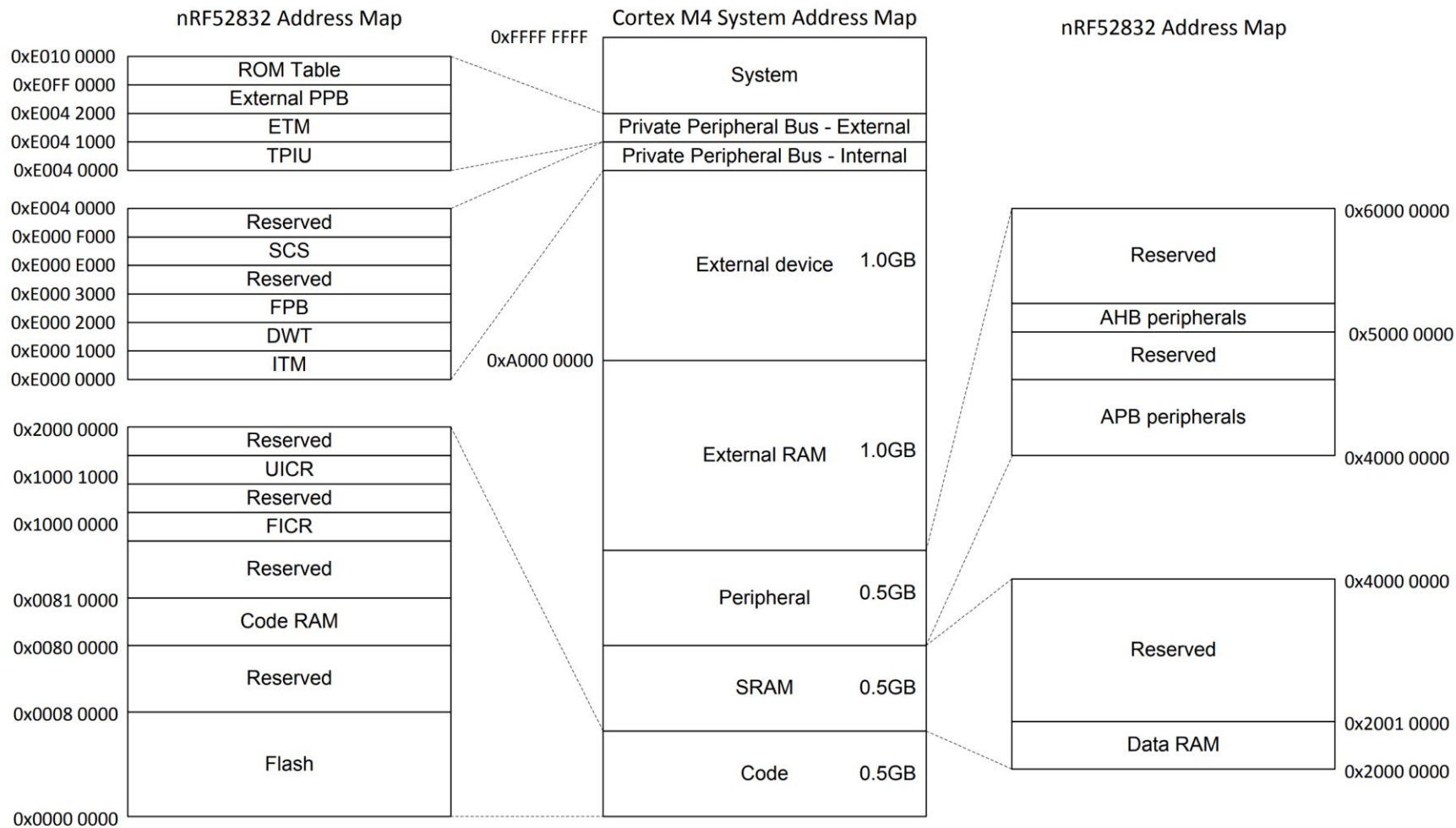
- ARM Cortex-M4F
- 3-stage pipeline!
- Floating point support

Memory

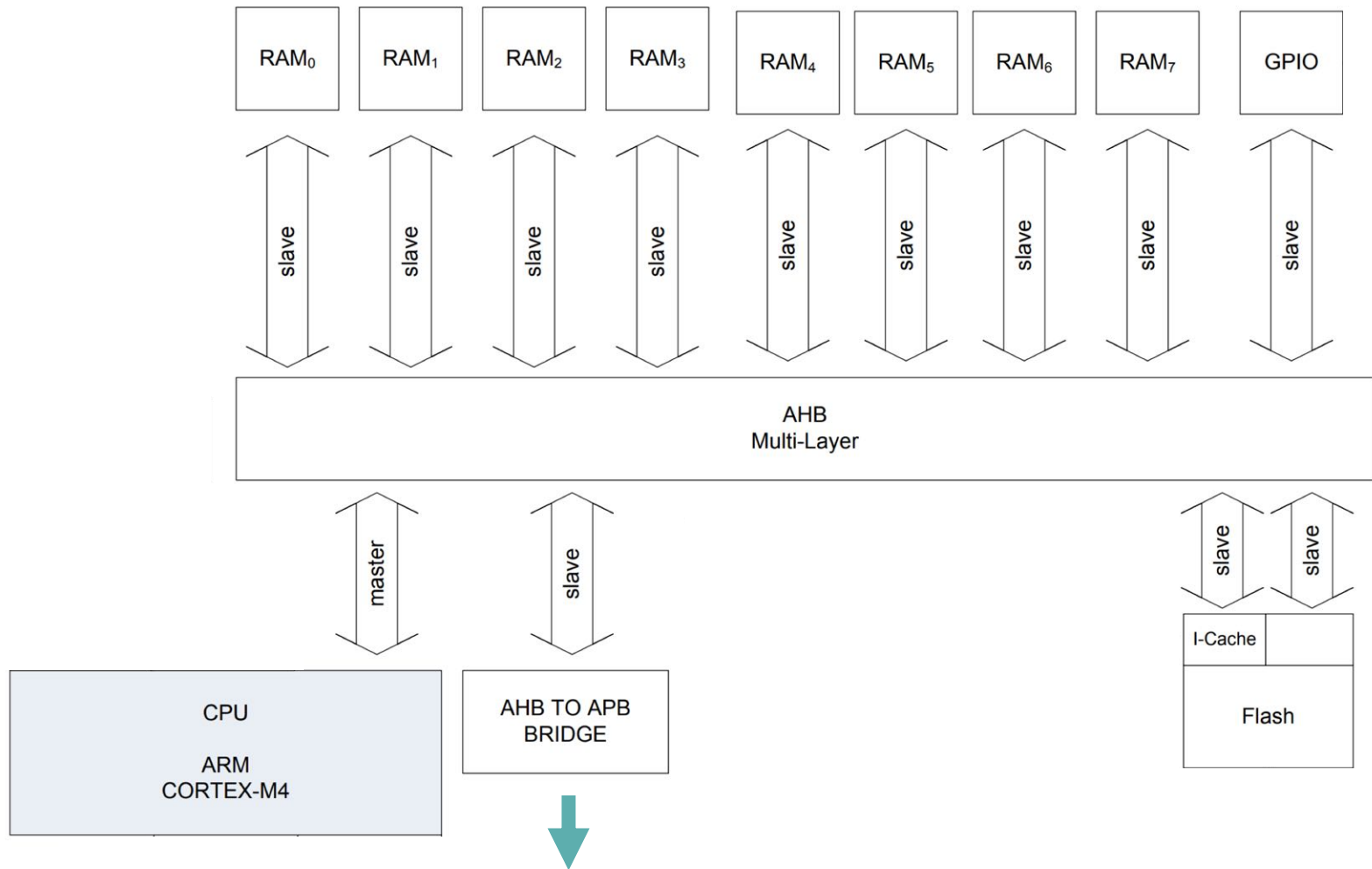
- Instruction Cache
 - Off by default
- 64 kB SRAM
- 512 kB Flash



Memory Map



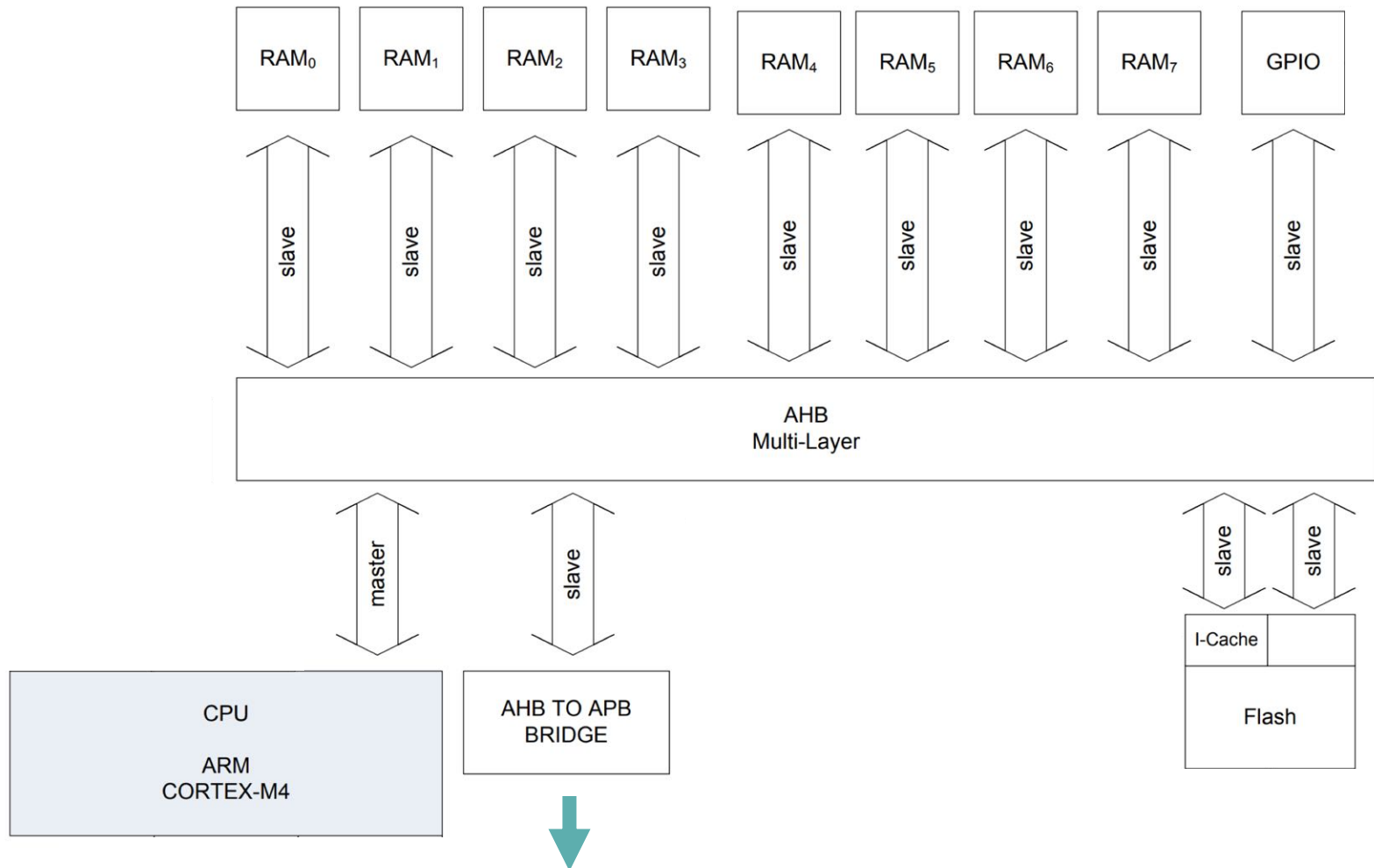
nRF52832 Block Diagram



To Peripherals

nRF52832 Block Diagram

Why have 8 separate
RAM banks?

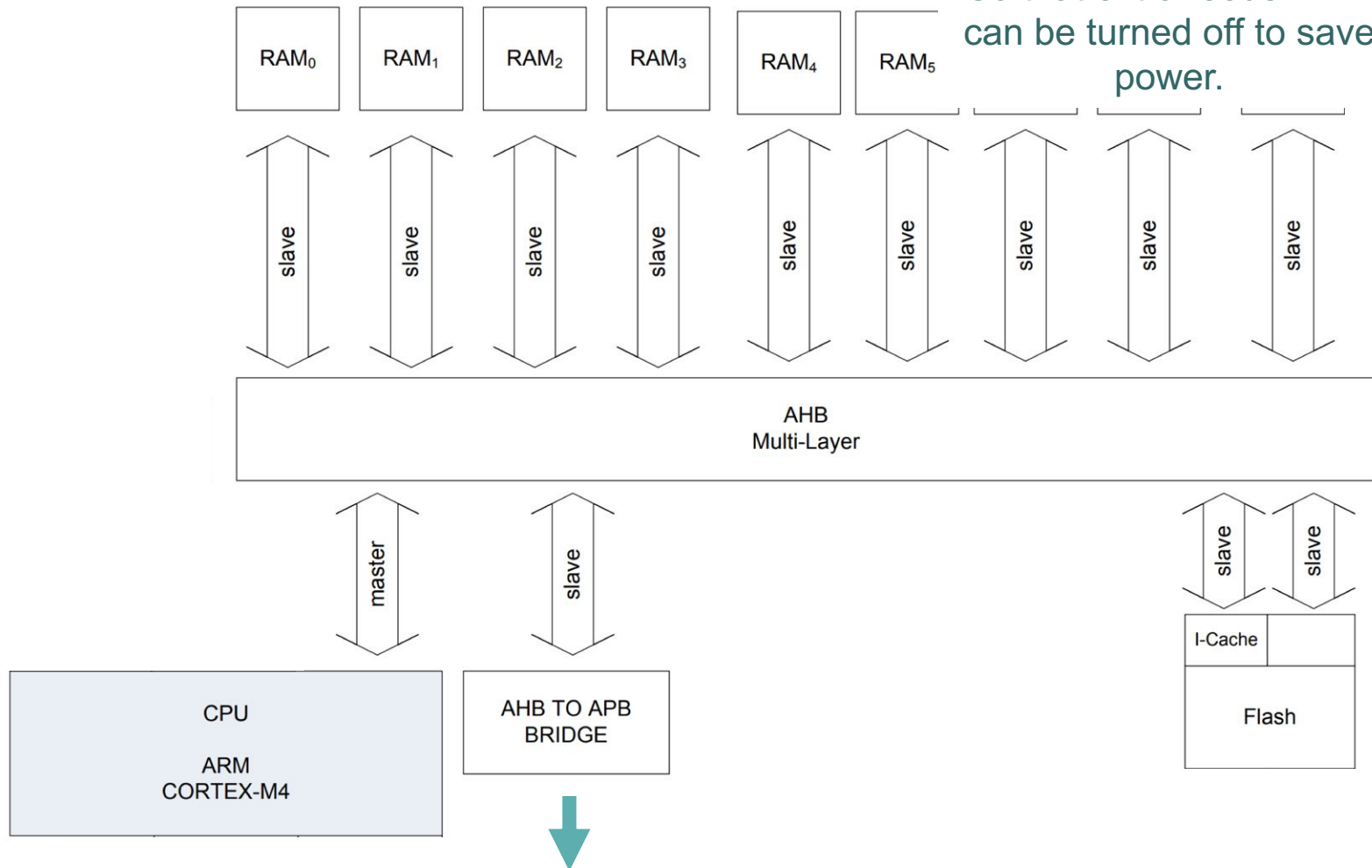


To Peripherals

nRF52832 Block Diagram

Why have 8 separate RAM banks?

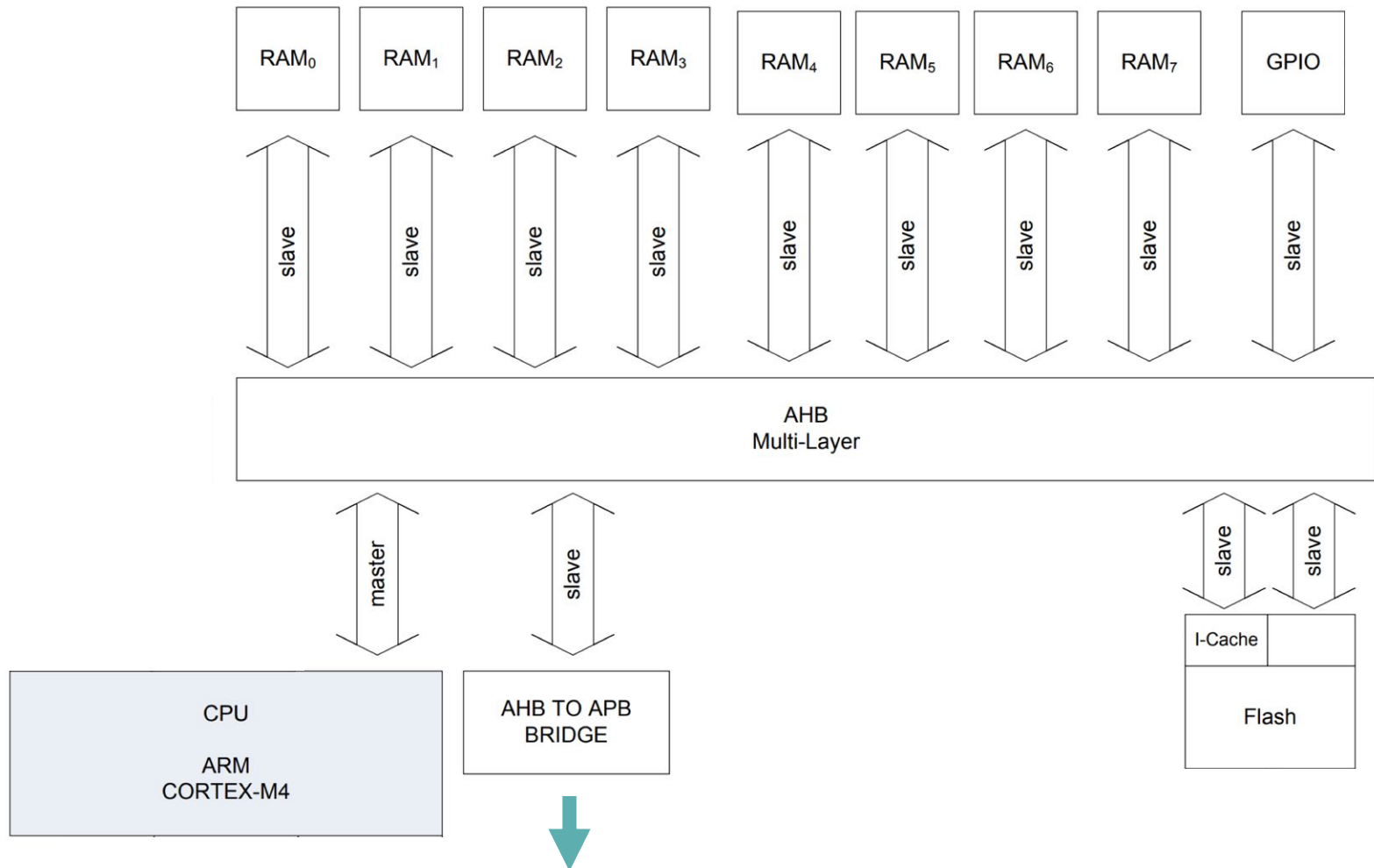
So that extraneous RAM can be turned off to save power.



To Peripherals

nRF52832 Block Diagram

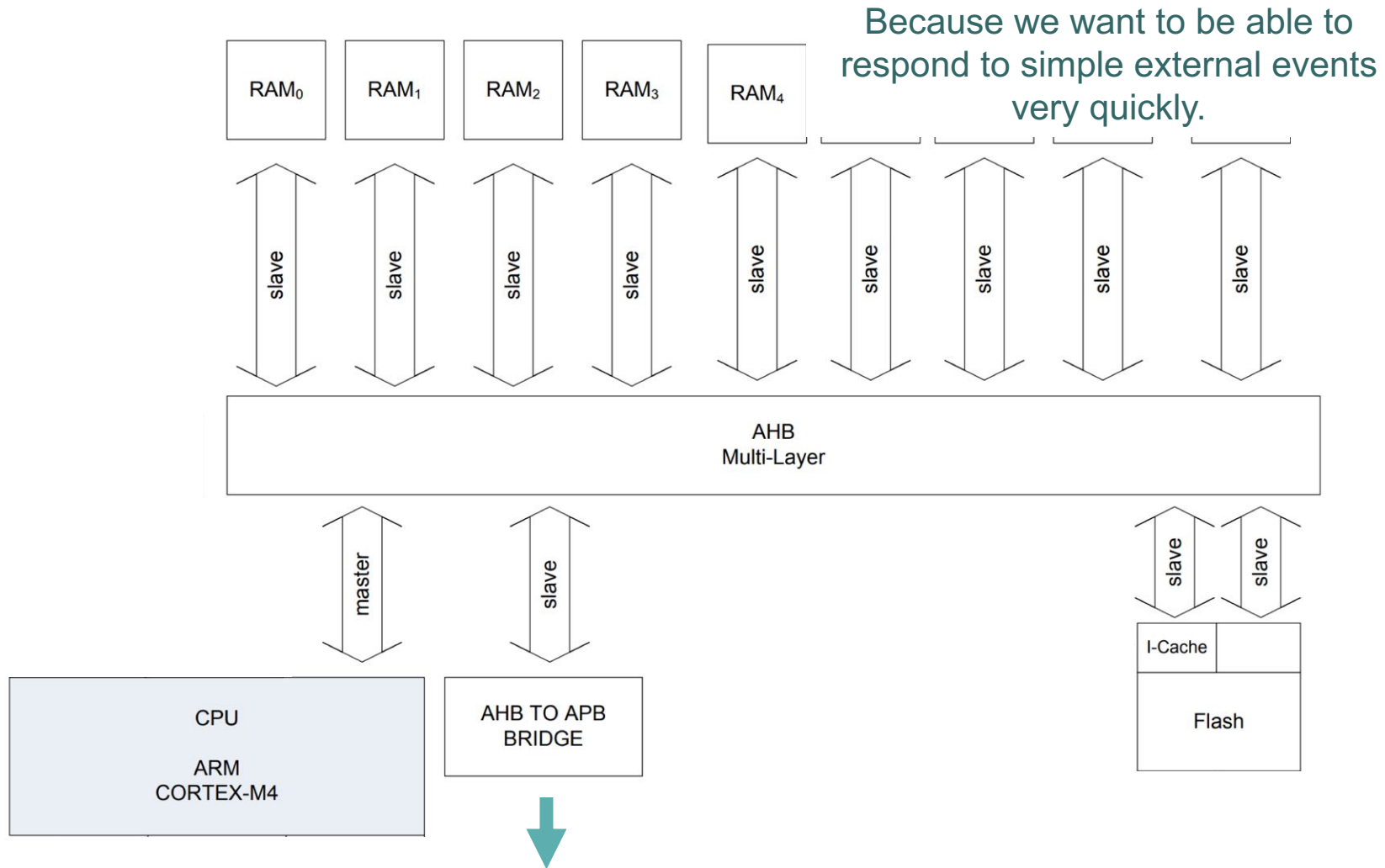
Why is GPIO special?



To Peripherals

nRF52832 Block Diagram

Why is GPIO special?



To Peripherals

SOFTWARE USE OF MEMORY

Memory Organization for Programs

- **Statically-allocated memory**
 - Compiler chooses the address at which to store a variable.
- **Stack**
 - Dynamically allocated memory with a Last-in, First-out (LIFO) strategy
- **Heap**
 - Dynamically allocated memory

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

Memory

```
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?

Memory

Stack

Heap

**Data
(Static)**

1 byte in the data section

Memory Layout Question 2

```
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

Assume a 32-bit
ARM microcontroller

Memory



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



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;
```

statically allocated: compiler assigns a memory location.

```
int* foo(int y) {
```

arguments on the stack

```
    int z;
```

automatic variables on the stack

```
    z = y * x;
```

```
    return &z;
```

```
}
```

```
int main(void) {
```

```
    int* result = foo(10);
```

```
    ...
```

```
}
```

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:

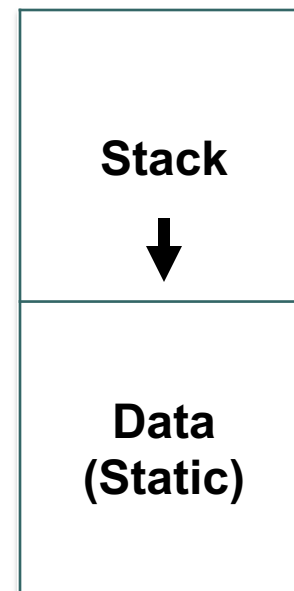
- Memory leaks (allocated memory is never freed)
- Memory fragmentation (allocatable pieces get smaller)

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?

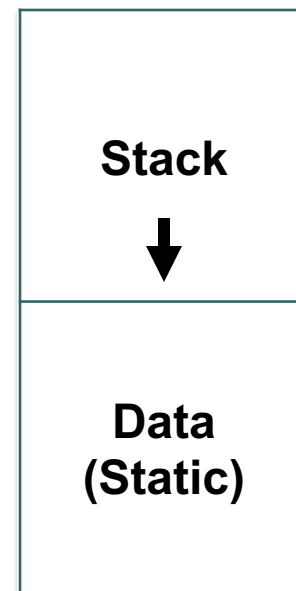
What if the stack grows too much?



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

Understanding memory architectures is essential to programming embedded systems.