Embedded Systems

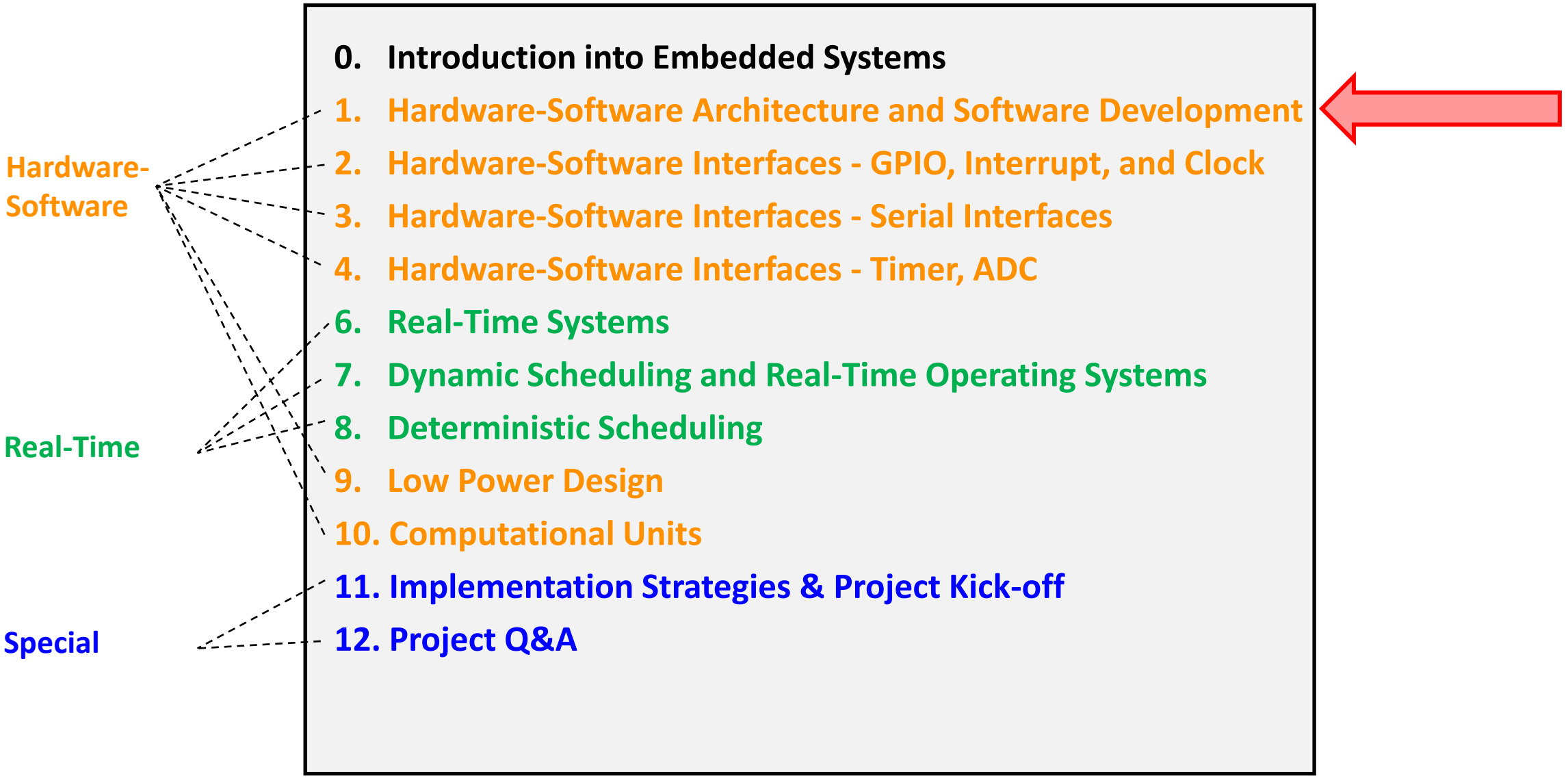
Lecture 1
Hardware-Software Architecture and Software Development

Michele Magno
D-ITET Center for Project-Based Learning

ETH zürich

Slides partially adopted from Prof. Thiele

Where We are


Hardware-Software
Real-Time
Special
0. Introduction into Embedded Systems
1. Hardware-Software Architecture and Software Development
2. Hardware-Software Interfaces - GPIO, Interrupt, and Clock
3. Hardware-Software Interfaces - Serial Interfaces
4. Hardware-Software Interfaces - Timer, ADC
6. Real-Time Systems
7. Dynamic Scheduling and Real-Time Operating Systems
8. Deterministic Scheduling
9. Low Power Design
10. Computational Units
11. Implementation Strategies & Project Kick-off
12. Project Q&A

Exam Information

There will be a written 2-hour exam in English. All content discussed in the lecture, as well as the exercises, is relevant to the examination.

Written Aids

You are allowed to bring **20 single-sided** or **10 double-sided A4 pages** of personal notes or printouts, and a calculator without communication capabilities.

Language of examination	English
Repetition	The performance assessment is offered every session. Repetition possible without re-enrolling for the course unit.
Mode of examination	written 120 minutes
Additional information on mode of examination	Toward the end of the lecture, the student can participate in an optional mini project where a 0.25 bonus to the final grade can be earned. During the semester practical programming sessions, will demonstrate and introduce all the concepts needed for completing the project successfully. Constantly following and solving the practical sessions is the best way to acquire the knowledge for completing the bonus project.
Written aids	20 single-sided or 10 double-sided A4 pages of personal notes or printouts, and a calculator without communication capabilities.

Microcontroller & Architecture

Recap: What is a Microcontroller ?

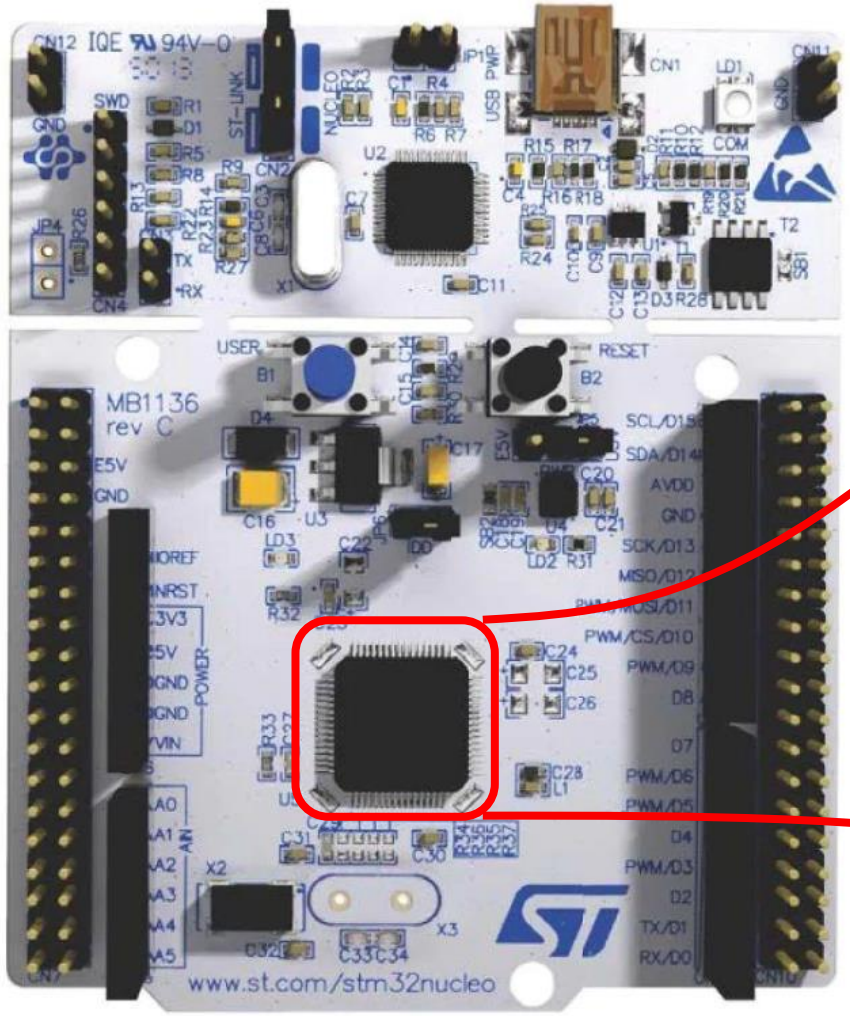
A microcontroller is a small processor with a core, memory and integrated I/O peripherals such as timers, analog-to-digital converters, and serial communications [1].

Features of a microcontroller (MCU):

- Self Contained (CPU, Memory, I/O)
- Application or Task Specific (Not a general-purpose computer)
- Appropriately scaled for the job
- Small power consumption
- Small size
- Low costs ($0.50 to $5.00.)

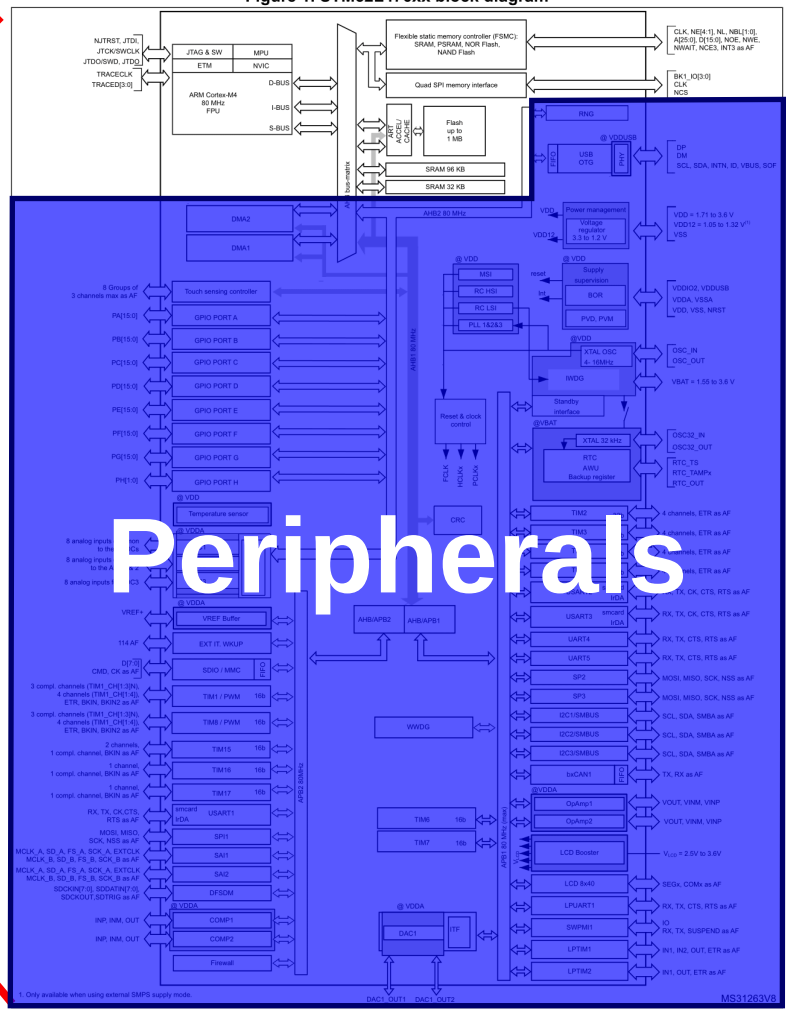
[1] Embedded Systems with ARM Cortex-M Microcontrollers in Assembly Language and C, page 56

Microcontroller – Basic components



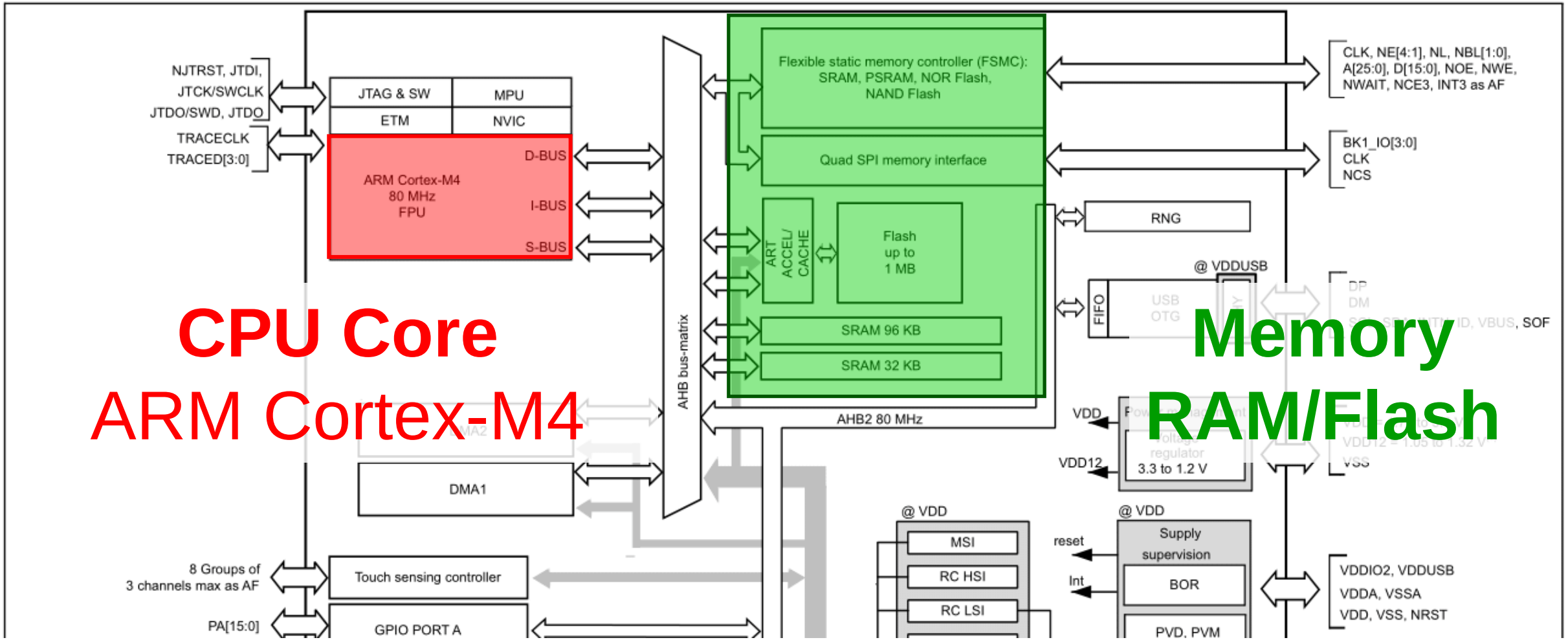
NUCLEO-L476RG Board with STM32L476RG MCU

Figure 1. STM32L476xx block diagram



Microcontroller – Basic components

Figure 1. STM32L476xx block diagram



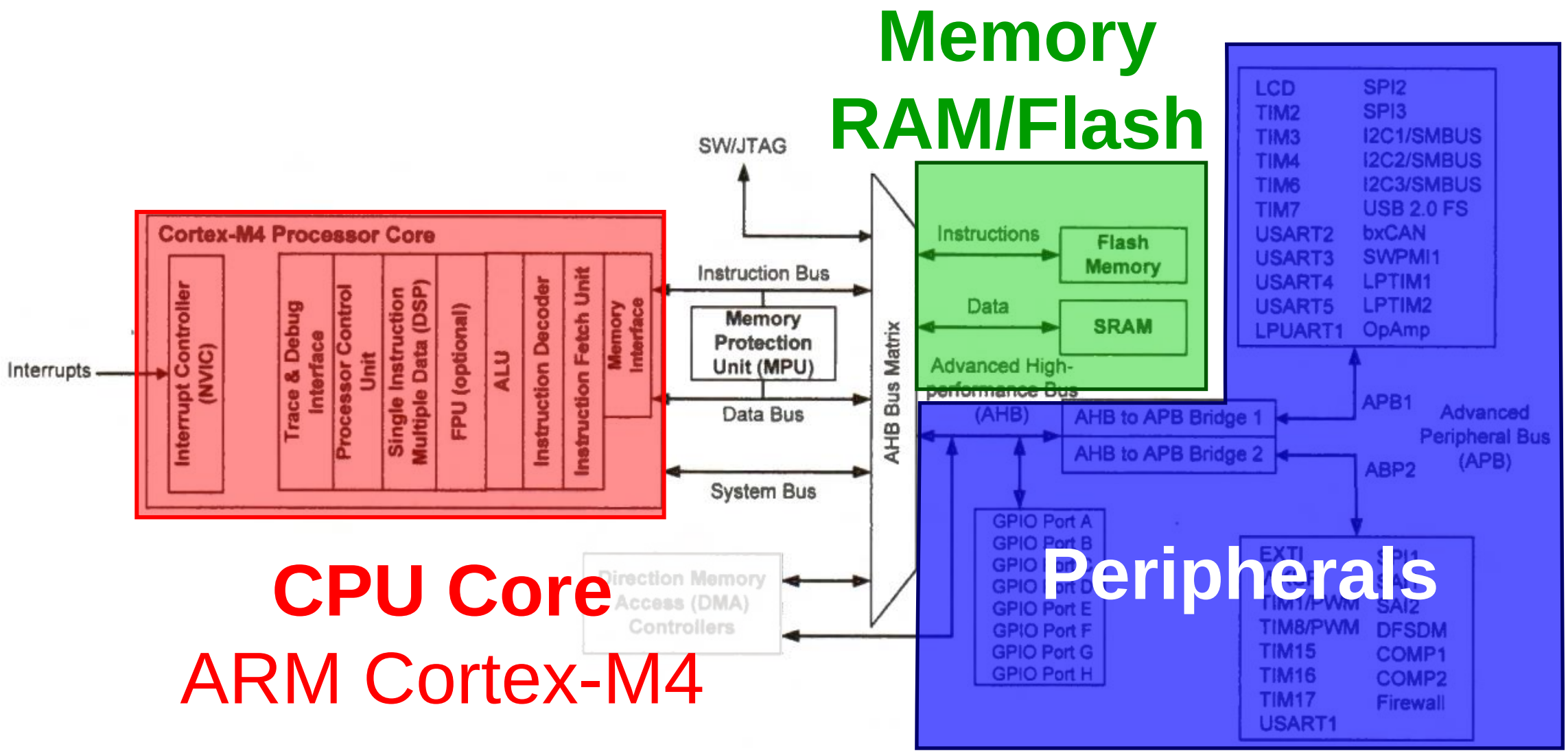
CPU Core

ARM Cortex-M4

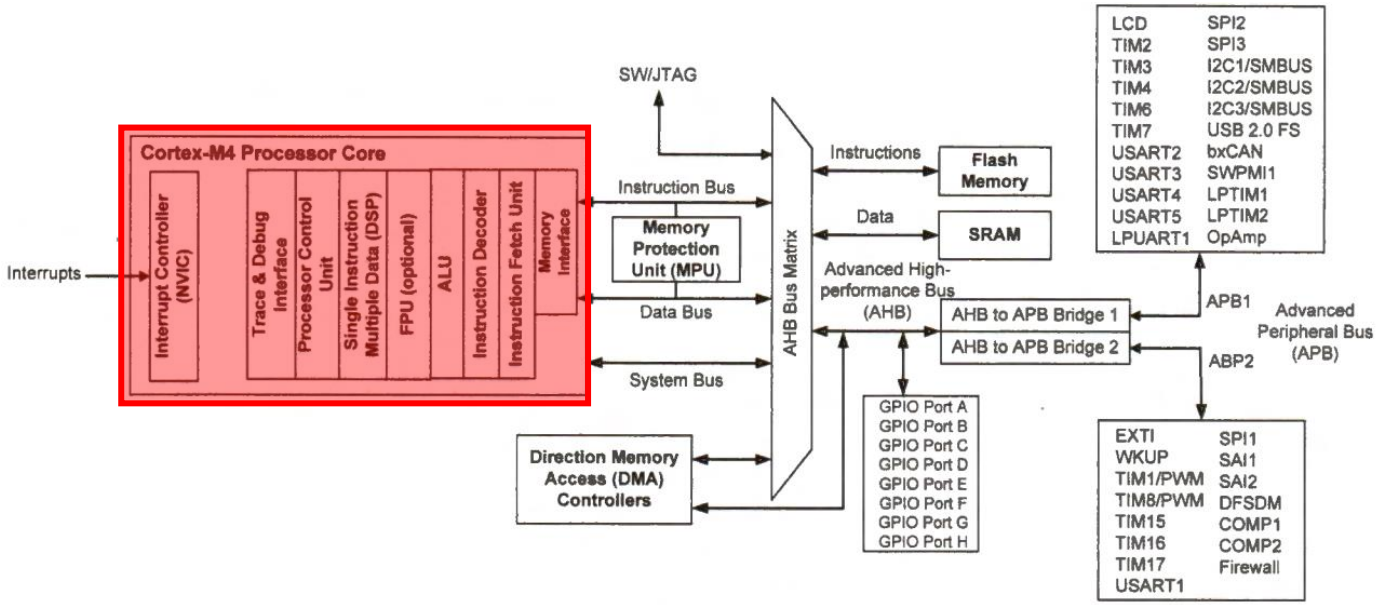
Memory

RAM/Flash

Microcontroller – Basic components

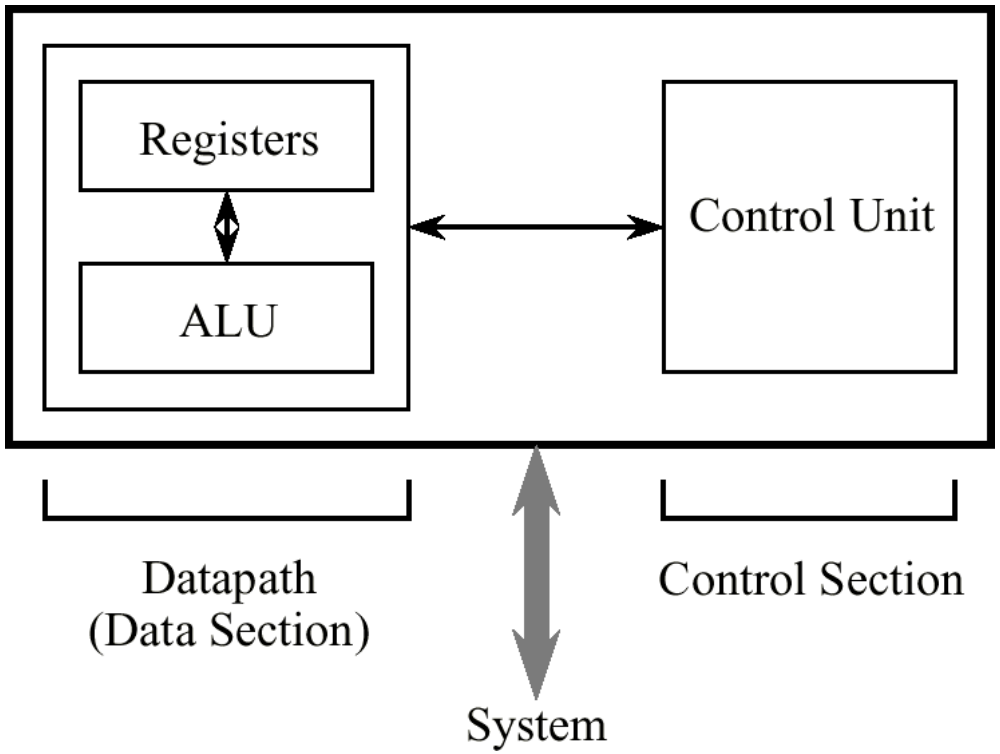


CPU Core


Cortex-M4 Processor Core
Interrupt Controller (NVIC)
Trace & Debug Interface
Processor Control Unit
Single Instruction Multiple Data (DSP)
FPU (optional)
ALU
Instruction Decoder
Instruction Fetch Unit
Memory Interface
Interrupts
SW/JTAG
Instruction Bus
Memory Protection Unit (MPU)
Data Bus
System Bus
Direction Memory Access (DMA) Controllers
AHB Bus Matrix
Instructions
Flash Memory
Data
SRAM
Advanced High-performance Bus (AHB)
AHB to APB Bridge 1
AHB to APB Bridge 2
APB1
ABP2
Advanced Peripheral Bus (APB)
GPIO Port A
GPIO Port B
GPIO Port C
GPIO Port D
GPIO Port E
GPIO Port F
GPIO Port G
GPIO Port H
LCD
TIM2
TIM3
TIM4
TIM6
TIM7
USART2
USART3
USART4
USART5
LPUART1
SPI2
SPI3
I2C1/SMBUS
I2C2/SMBUS
I2C3/SMBUS
USB 2.0 FS
bxCAN
SWPMI1
LPTIM1
LPTIM2
OpAmp
EXTI
WKUP
TIM1/PWM
TIM8/PWM
TIM15
TIM16
TIM17
USART1
SPI1
SAI1
SAI2
DFSDM
COMP1
COMP2
Firewall

Abstract View of a CPU Core

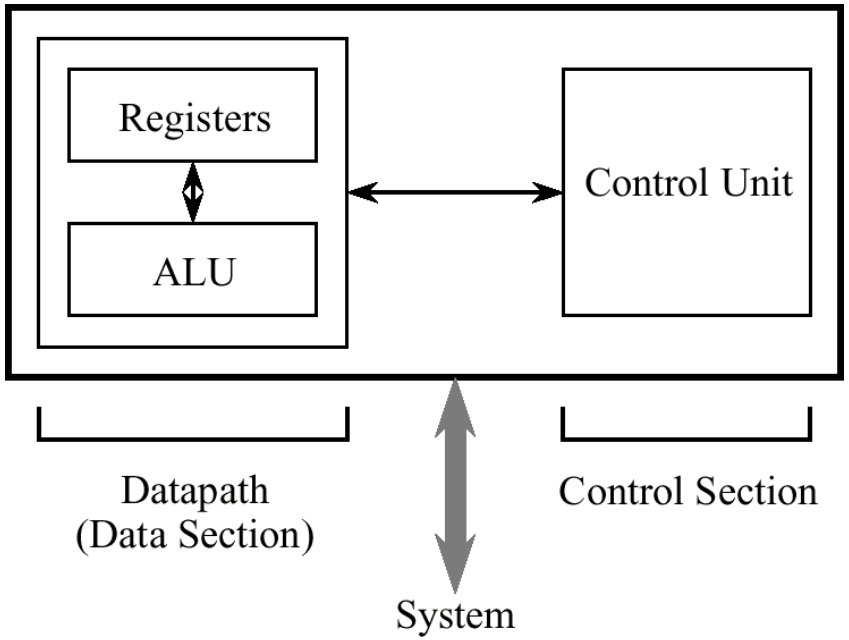
The *central processing unit (CPU)* consists of a data section containing registers and an ALU (Arithmetic and Logic Unit), and a control section, which *interprets instructions* and effects register transfers. The data section is also known as the datapath.



Abstract View of a CPU Core

The CPU can basically only do the following operations:

- Read instructions
- Move data in and out of memory
- Perform arithmetic and logic operations on the data
- Change the instruction flow based on certain conditions



Abstract View of a CPU Core

CPU-Registers

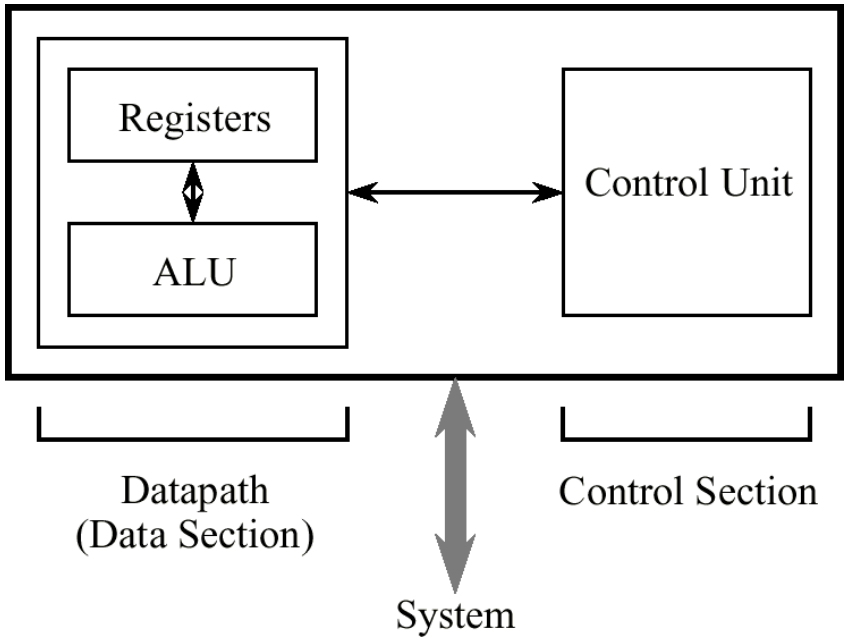
- Small fast storage locations for inputs, outputs and intermediate values of operations
- In our MCU – 16x 32bit (`r0, r1, r2,... r15`)

ALU

- Arithmetic Logic Unit
- Can perform
  - Arithmetic operations on registers (`add, mul, sub` etc)
  - Logic operations on registers (`not, and, or`)

Control Unit

- Directs the operation of the processor based on the instructions



How do we instruct the CPU?

An *Instruction* is a binary-encoded command that directs the CPU to perform a specific operation such as *data processing*, *memory access* or *control flow*.

The *Compiler* translates the code you write into these instructions.

```
void add(int *a, int *b, int *c){
        *c = *a + *b;
}
```


```
add:
        ldr     r3, [r0]
        ldr     r1, [r1]
        add     r3, r3, r1
        str     r3, [r2]
        bx      lr
```

Anatomy of an instruction

The binary instructions are often represented using human readable *assembly language:*

Square brackets around registers are like the C pointer dereference operator `*r0` - address stored in `r0`

Operation
Represented using a shorthand op-code

Output
Usually, the first operand is the destination register

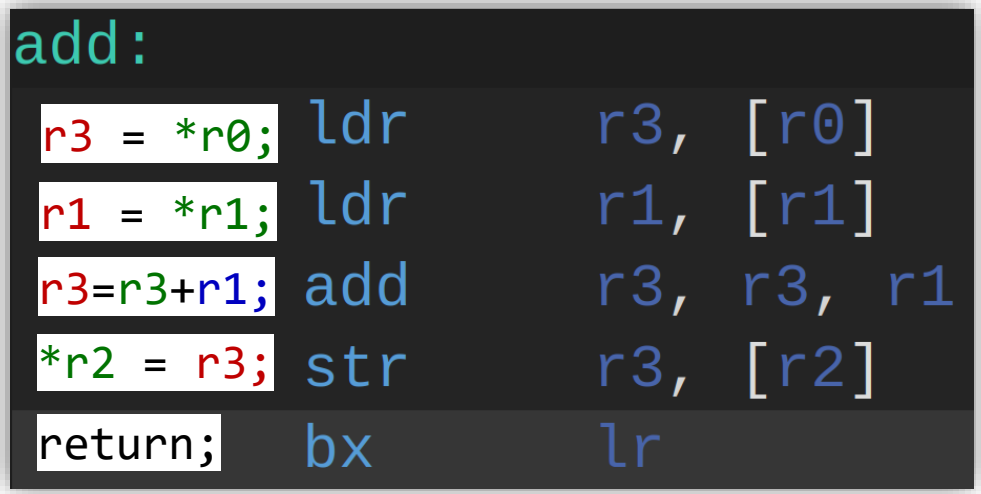
Inputs
The remaining operands are generally the inputs to the instruction

The *Instruction Set* is the complete collection of available instructions supported by the CPU hardware.

Anatomy of an instruction

Examples of instructions in ARM Instruction Set:

- `ldr r3, [r0]`
  - Load register `r3` with value at address in `[r0]`
- `ldr r1, [r1]`
  - Load register `r1` with value at address in `[r1]`
- `add r3, r3, r1`
  - add values in register `r3` and `r1`, storing the result in `r3`
- `str r3, [r2]`
  - Store register `r3` in memory at the address in `[r2]`
- `bx lr`
  - Return from function back to the caller



How do we instruct the CPU?

We will now examine the step-by-step execution of this function:

```c
void add(int *a, int *b, int *c){
        *c = *a + *b;
}
```

```
add:
        ldr     r3, [r0]
        ldr     r1, [r1]
        add     r3, r3, r1
        str     r3, [r2]
        bx      lr
```

How do we instruct the CPU?

```c
void add(int *a, int *b, int *c){
    *c = *a + *b;
}
```

Function `void add(int *a, int *b, int *c);`

- Receives the address of three variables as arguments
- It loads the first two `*a` and `*b` from memory
- Computes their sum
- Stores the result back in memory in `*c`

```
add:
        ldr     r3, [r0]
        ldr     r1, [r1]
        add     r3, r3, r1
        str     r3, [r2]
        bx      lr
```

How do we instruct the CPU?

```
int a = 1; // Address 0x2000 AAAA
int b = 2; // Address 0x2000 BBBB
int c = 0; // Address 0x2000 CCCC
add(&a, &b, &c);
```

Variable	Address	Value
a	0x2000 AAAA	0x0000 0001
b	0x2000 BBBB	0x0000 0002
c	0x2000 CCCC	0x0000 0000

Register	Value
r0	0x2000 AAAA
r1	0x2000 BBBB
r2	0x2000 CCCC
r3	0x0000 0000

- We initialize 3 variables
- When we call `add(&a, &b, &c)` the caller first preloads the following registers:
  - `r0 = &a` (address of variable a)
  - `r1 = &b`
  - `r2 = &c`
- We now enter the `add` function

How do we instruct the CPU?

Variable	Address	Value
a	0x2000 AAAA	0x0000 0001
b	0x2000 BBBB	0x0000 0002
c	0x2000 CCCC	0x0000 0000

Memory values

Register	Value
r0	0x2000 AAAA
r1	0x2000 BBBB
r2	0x2000 CCCC
r3	0x0000 0001

Register values

ldr

```
void add(int *a, int *b, int *c){
        *c = *a + *b;
}
```

```
add:
        ldr     r3, [r0]
        ldr     r1, [r1]
        add     r3, r3, r1
        str     r3, [r2]
        bx      lr
```

`ldr r3, [r0]` – load register (`r3`) with value from memory at address in `[r0]`

How do we instruct the CPU?

Variable	Address	Value
a	0x2000 AAAA	0x0000 0001
b	0x2000 BBBB	0x0000 0002
c	0x2000 CCCC	0x0000 0000

Memory values

ldr

Register	Value
r0	0x2000 AAAA
r1	0x0000 0002
r2	0x2000 CCCC
r3	0x0000 0001

Register values

```
void add(int *a, int *b, int *c){
        *c = *a + *b;
}
```

```
add:
        ldr     r3, [r0]
        ldr     r1, [r1]
        add     r3, r3, r1
        str     r3, [r2]
        bx      lr
```

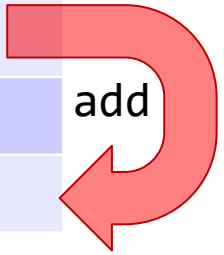
`ldr r1, [r1]` – load register (`r1`) with value from memory at address in `[r1]`

How do we instruct the CPU?

Variable	Address	Value
a	0x2000 AAAA	0x0000 0001
b	0x2000 BBBB	0x0000 0002
c	0x2000 CCCC	0x0000 0000

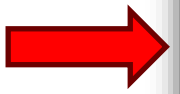
Memory values

Register	Value
r0	0x2000 AAAA
r1	0x0000 0002
r2	0x2000 CCCC
r3	0x0000 0003



Register values

```
void add(int *a, int *b, int *c){
        *c = *a + *b;
}
```



```
add:
        ldr     r3, [r0]
        ldr     r1, [r1]
        add     r3, r3, r1
        str     r3, [r2]
        bx      lr
```

add r3, r3, r1 – add registers r3 and r1 and store the result in r3

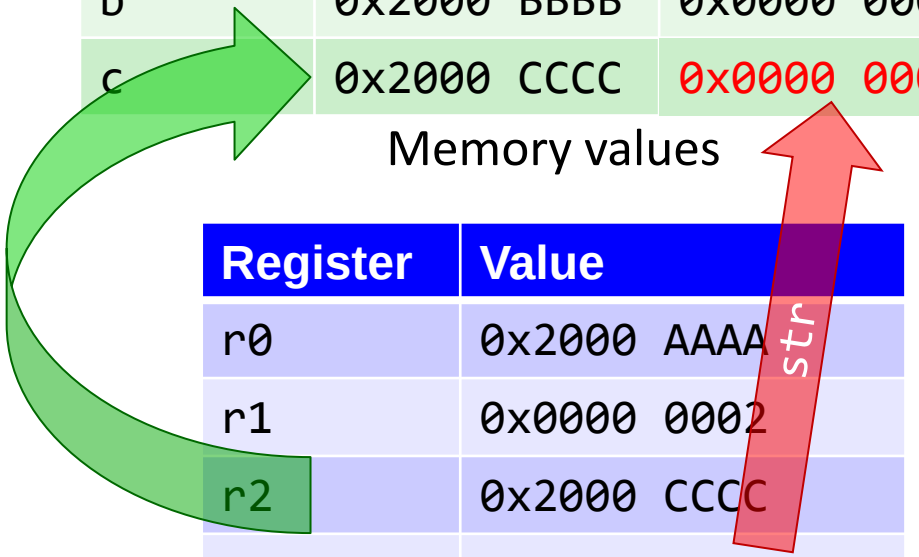
How do we instruct the CPU?

Variable	Address	Value
a	0x2000 AAAA	0x0000 0001
b	0x2000 BBBB	0x0000 0002
c	0x2000 CCCC	0x0000 0003

Memory values

Register	Value
r0	0x2000 AAAA
r1	0x0000 0002
r2	0x2000 CCCC
r3	0x0000 0003

Register values




```
void add(int *a, int *b, int *c){
        *c = *a + *b;
}
```

```
add:
        ldr     r3, [r0]
        ldr     r1, [r1]
        add     r3, r3, r1
        str     r3, [r2]
        bx      lr
```

`str r3, [r2]` – store register (`r3`) in memory at the addres in `[r2]`

How do we instruct the CPU?

Variable	Address	Value
a	0x2000 AAAA	0x0000 0001
b	0x2000 BBBB	0x0000 0002
c	0x2000 CCCC	0x0000 0003

Memory values

Register	Value
r0	0x2000 AAAA
r1	0x0000 0002
r2	0x2000 CCCC
r3	0x0000 0003

Register values

```
void add(int *a, int *b, int *c){
        *c = *a + *b;
}
```

Results in variable c

```
add:
        ldr     r3, [r0]
        ldr     r1, [r1]
        add     r3, r3, r1
        str     r3, [r2]
        bx      lr
```

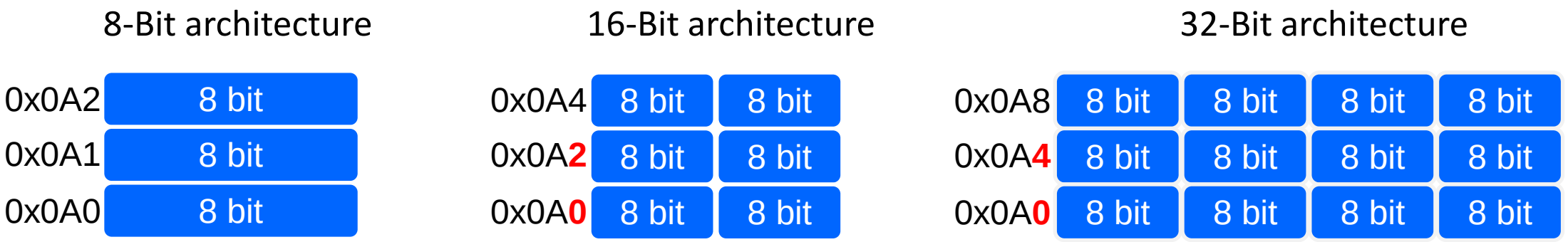
bx `lr` – bx retruns to the function caller, we will not go into further details

Instruction Set Properties

Processor Size

Processor size is described in terms of 'bits' (e.g. an 8-bit or 32-bit processor)

- Corresponds to the data size that can be manipulated at a time by the processor
- Typically reflected in the size of the processor internal data path and memory bank. The addresses of the "data segments" are referred to the first byte address
- An 8-bit processor can only manipulate one byte of data at a time, while a 32-bit processor can handle one 32-bit double word sized data at a time even though the data content may only be of single byte size



Instruction Sets - CISC

Complex Instruction Set Computing (CISC) is a computer architecture that emphasizes a large and complex instruction set CISC processors have many instructions that can perform multiple operations in a single instruction.

The goal of CISC architecture is to *reduce the number of instructions a program* needs to execute, which can lead to faster program execution

- *Intel/AMD x86*

Advantages of CISC:

- Ability to perform complex instructions
- Programs require fewer instructions to execute
- Greater hardware support for performing complex instructions

Disadvantages of CISC:

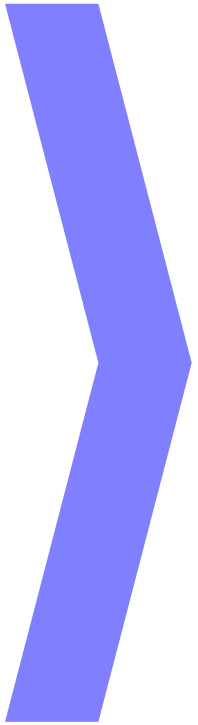
- Increased complexity can lead to slower processing times
- Larger chip size can lead to increased costs

Do we need Complex instructions?

Typical frequency distribution of executed instructions:

- *Memory/Register Moves*
  - `ldr,str`
  - ➢ **35-50%**
- *Conditional branching*:
  - Loops, branches
  - ➢ 15-25%
- Arithmetic/Logic:
  - `and, or, add, mul…`
  - ➢ 10-25%

- Others: 5%-10%



RISC:
Make these fast and efficient!

Instruction Sets - RISC

Reduced Instruction Set Computing (RISC) is a computer architecture that emphasizes a simple and efficient instruction set where each instruction performing a single operation.

The goal of RISC architecture is to *reduce the amount of work the processor needs to do for each instruction*, which leads to faster and more efficient processing.

- *ARM, RISC-V*

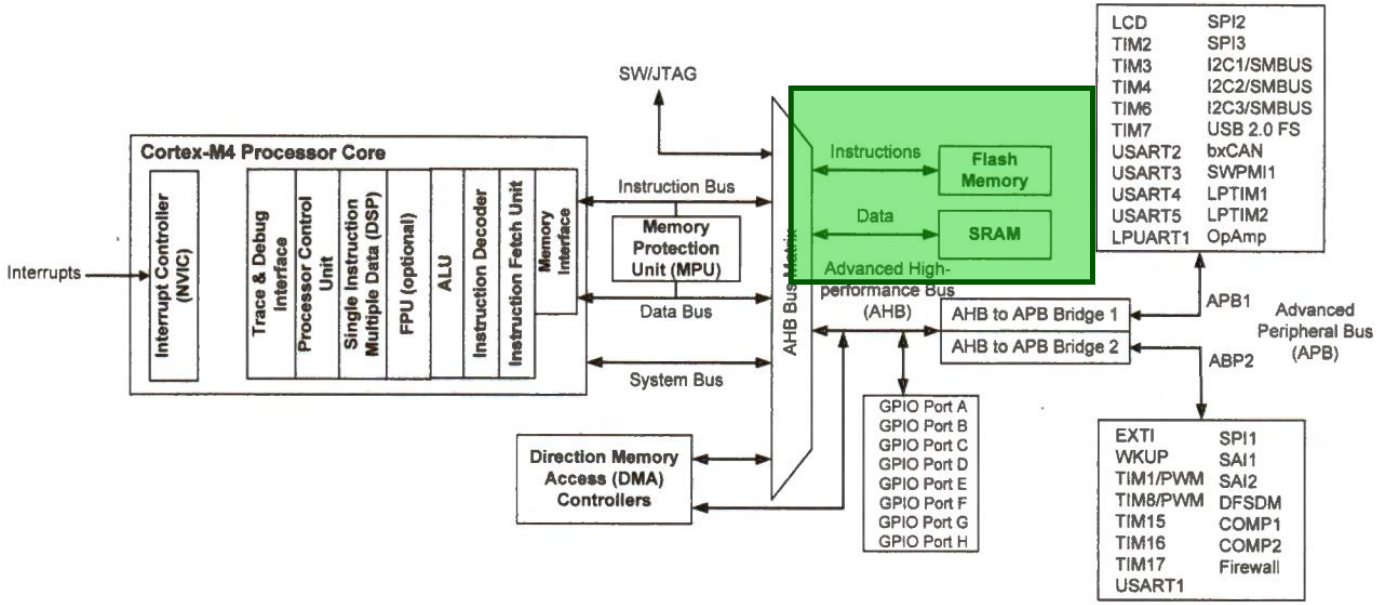
Advantages of RISC:

- Simplified instruction set leads to faster processing
- Pipelining can increase performance
- Lower power consumption
- Smaller chip size, which can lead to cost savings

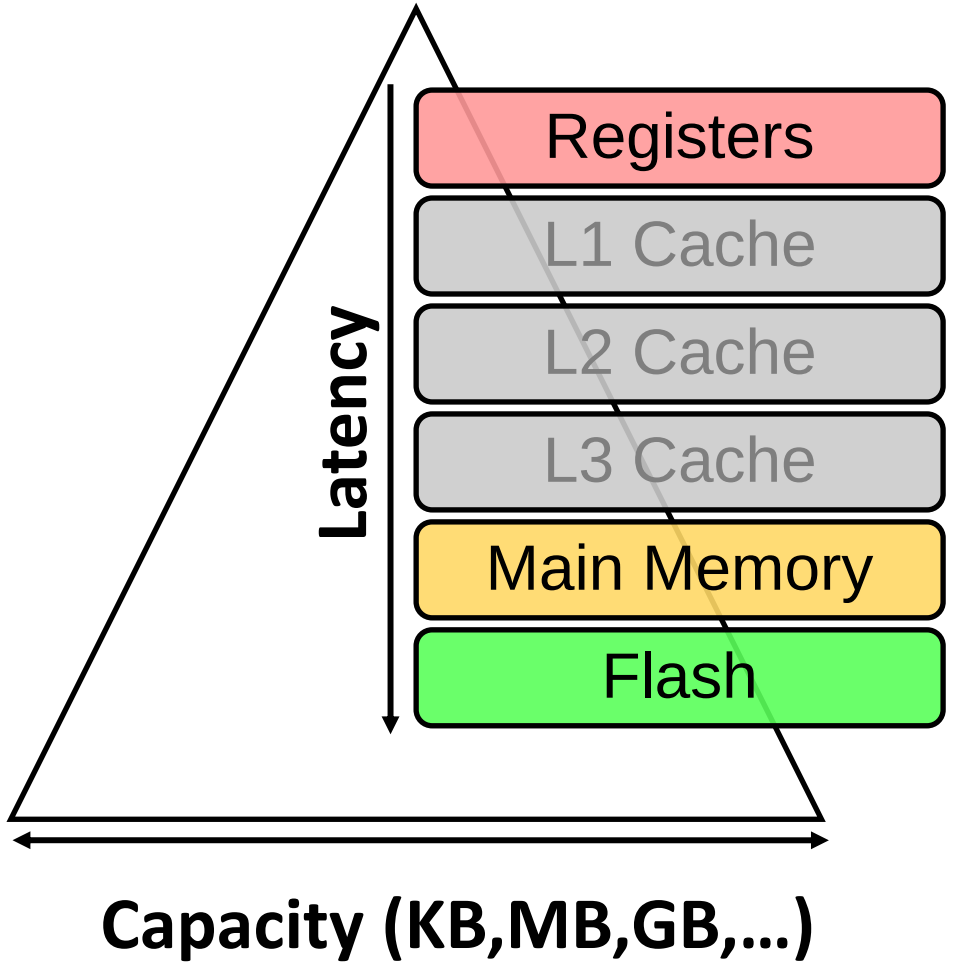
Disadvantages of RISC:

- Programs may require more instructions to complete a task
- Limited ability to perform complex instructions

Storage: Registers / SRAM / DRAM / Flash

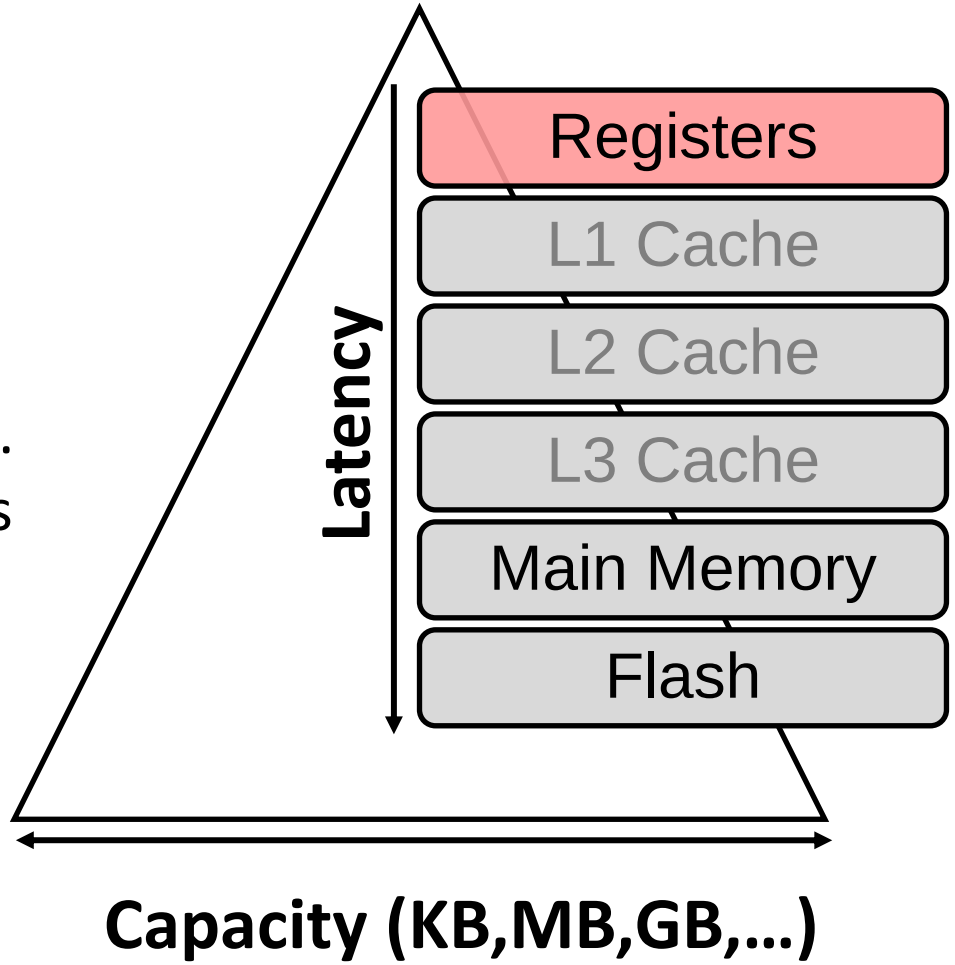


Memory Hierarchy, Persistent and Volatile Memory



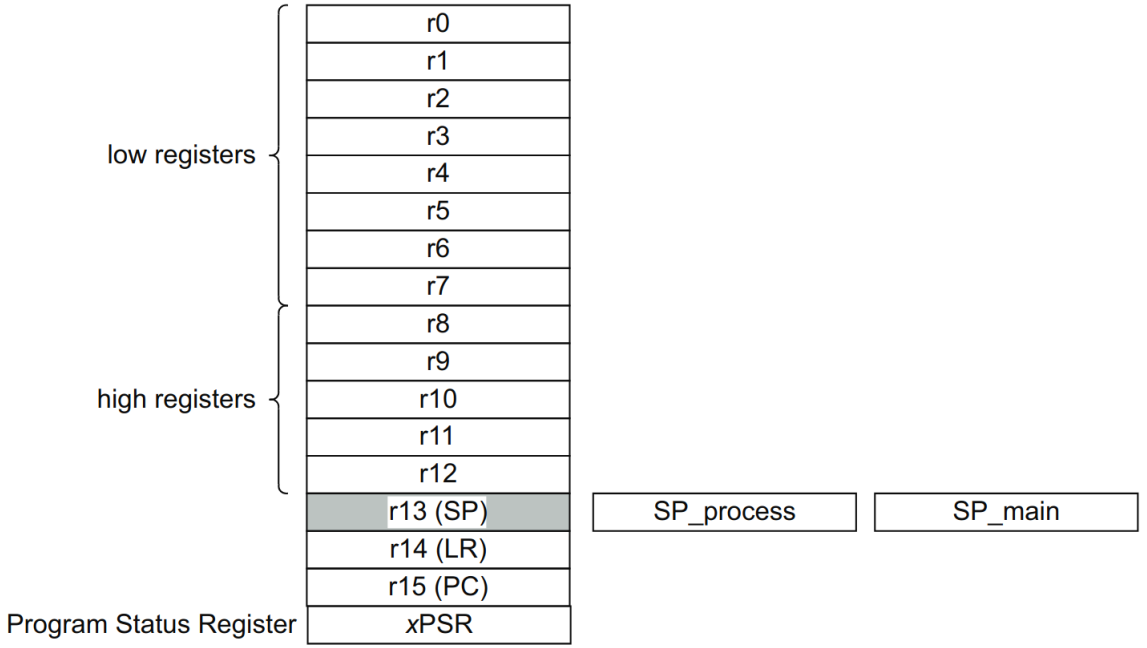
Memory Hierarchy, Persistent and Volatile Memory

- Processor *General Purpose register* are small fast storage locations in the CPU used to hold data temporarily during instruction execution. Registers store operands, intermediate results or memory addresses, enabling quick access and manipulation by the processor



Memory Hierarchy, Persistent and Volatile Memory

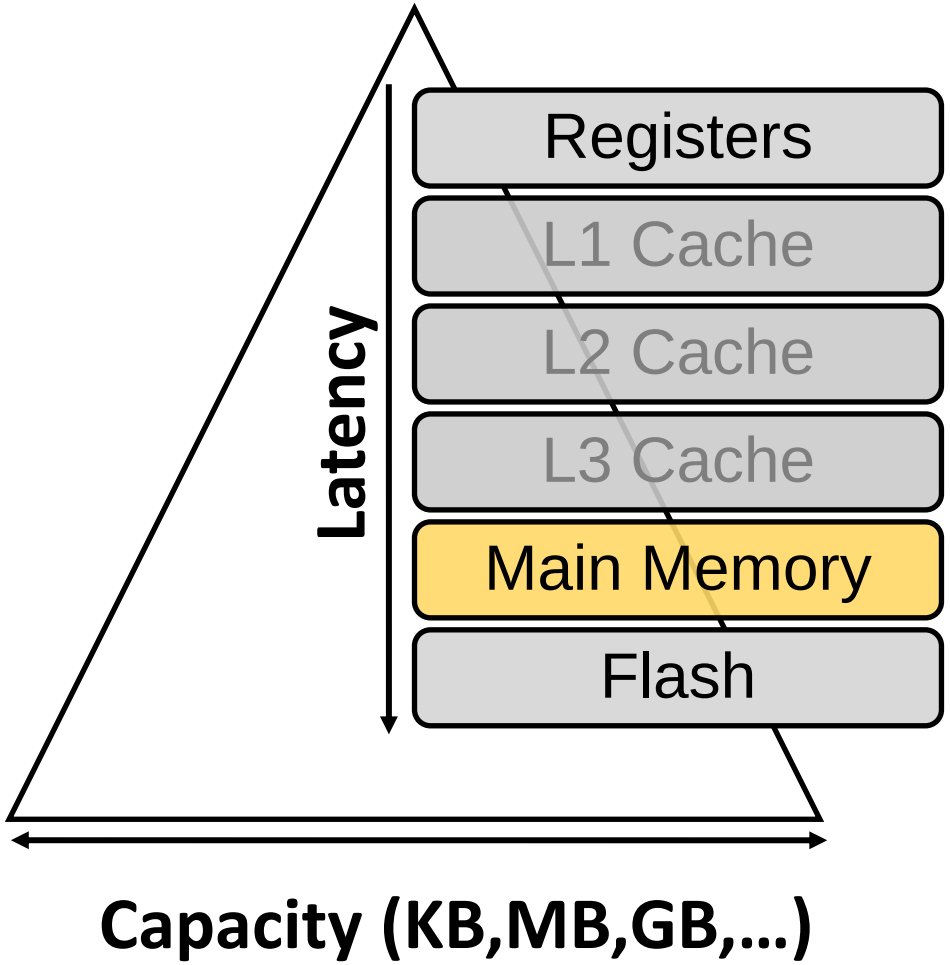
- ARM Cortex M4
  - 16x 32bit registers
  - r0-r12 accessible by most instructions
  - r13 – Stack Pointer
  - r14 – Link Register
  - r15 – Program Counter



[1] Cortex-M4 Technical Reference Manual

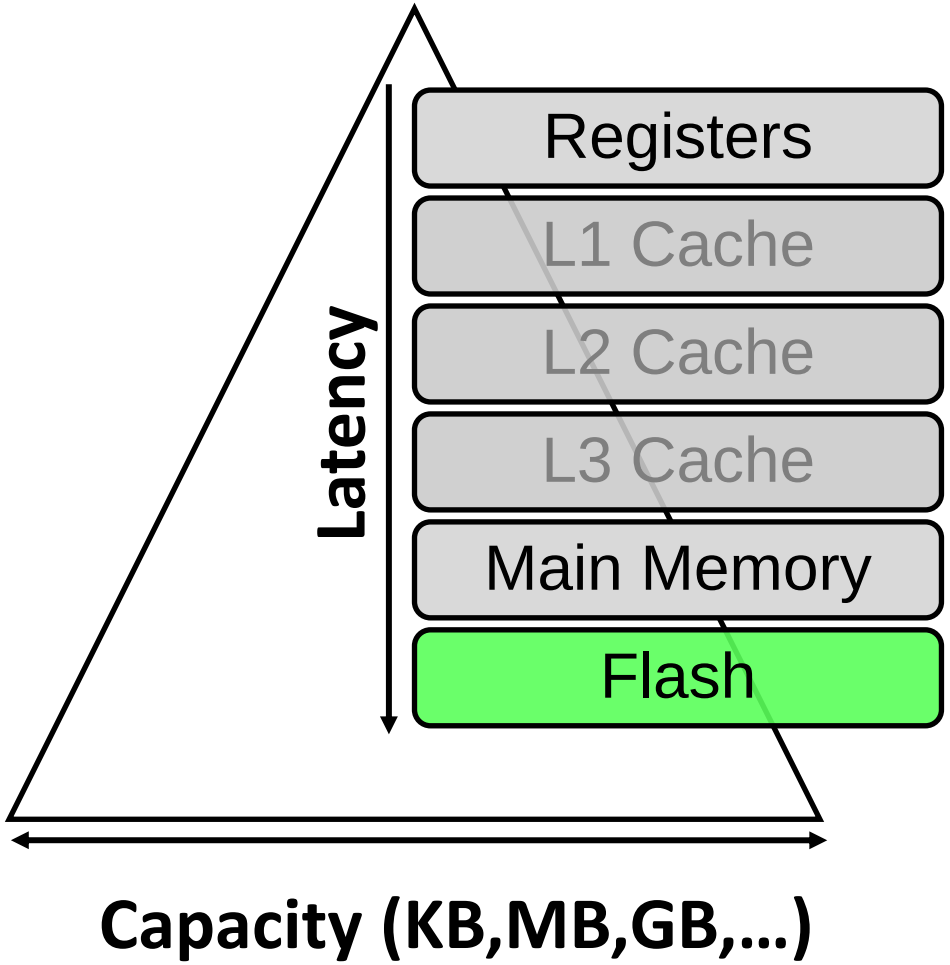
Memory Hierarchy, Persistent and Volatile Memory

- The *working memory* (or **main memory**) of computer systems implements the storage implied by processor memory addresses. Usually, it has a capacity between a few kilobytes and some gigabytes and is **volatile**.



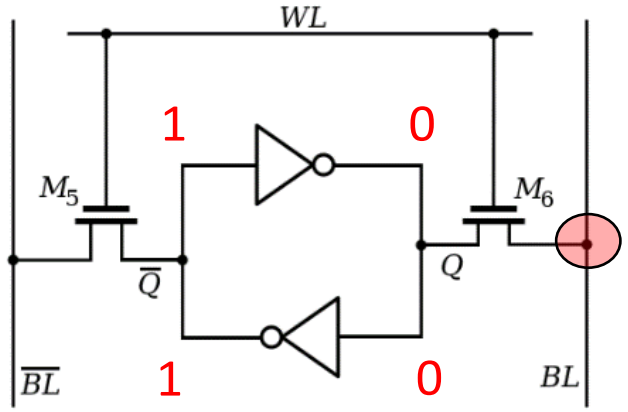
Memory Hierarchy, Persistent and Volatile Memory

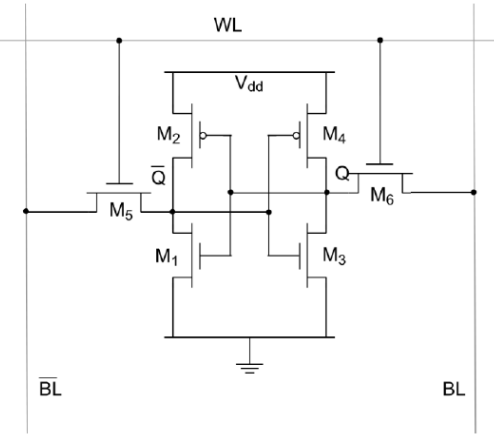
- Flash or Hard drive
  - Persistent across power loss
  - Storage of code
  - Larger capacity
  - Slower



Static Random Access Memory (SRAM)

Single bit is stored in a bi-stable circuit





Static Random Access Memory is used for:

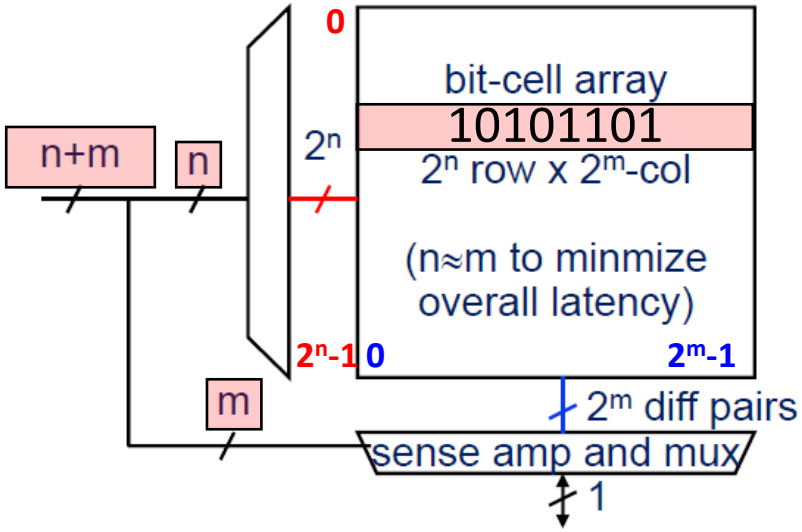
- Caches
- register file within the processor core
- Small capacity but fast memories

Read Cycle:

1. Pre-charge all bit-lines to average voltage
2. decode address (n+m bits)
3. select row of cells using 2^n single-bit word lines (WL)
4. selected bit-cells drive all bit-lines BL (2^m pairs)
5. sense difference between bit-line pairs and read out

Write:

- select row and overwrite bit-lines using strong signals



Dynamic Random Access (DRAM)

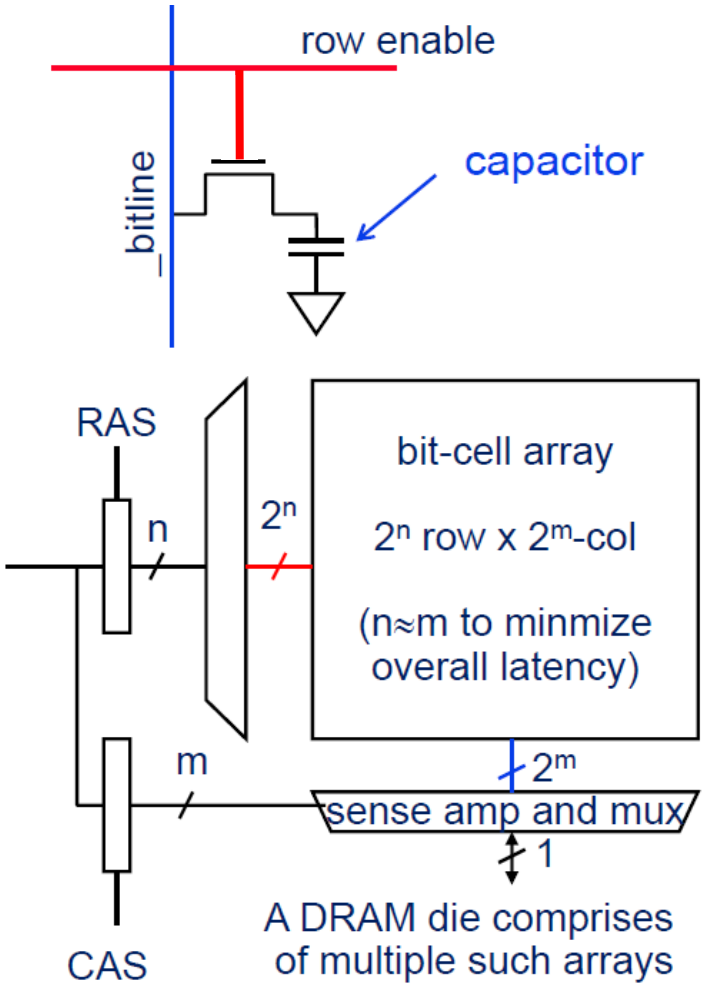
Single bit is stored as a charge in a capacitor

- Bit cell loses charge when read, bit cell drains over time
- Slower access than with SRAM due to small storage capacity in comparison to capacity of bit-line.
- Higher density than SRAM (1 vs. 6 transistors per bit)

DRAMs require periodic refresh of charge

- Performed by the memory controller
- Refresh interval is tens of ms
- DRAM is unavailable during refresh

For reading and writing



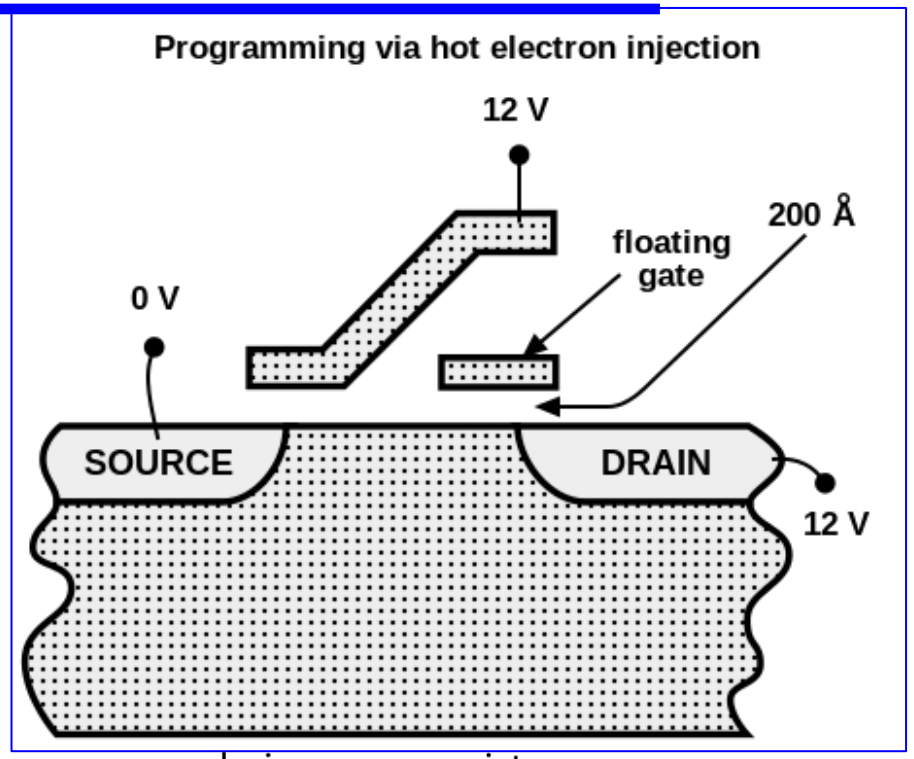
(RAS/CAS = row/column address select)

Flash Memory

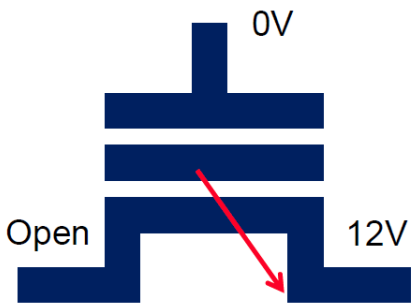
Electrically modifiable, non-volatile storage

Principle of operation:

- Transistor with a second *floating gate*
- Floating gate can trap electrons
- This results in a detectable change in threshold voltage

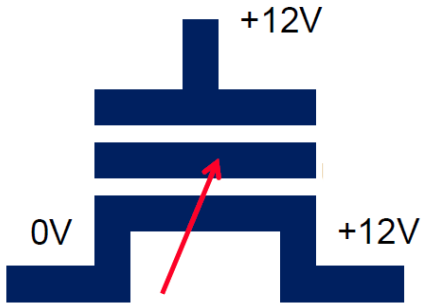


Erasing to logical "1"



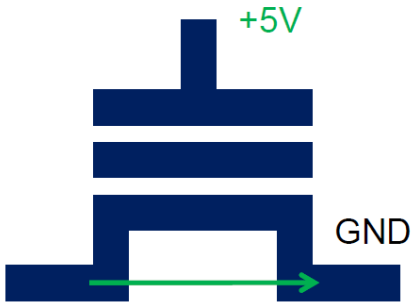
"Quantum tunneling"
Drains charge from FG

Programming (=writing) to logical "0"



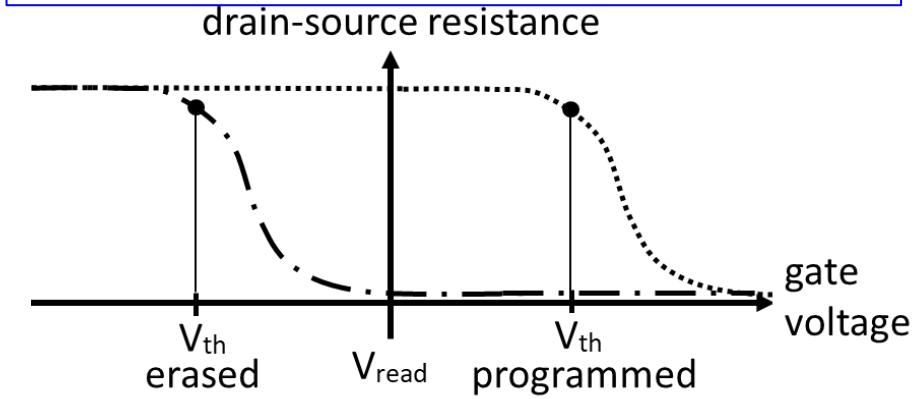
"Hot-electron injection"
traps charge in FG

Reading



Turn on low Vt or High Vt?

Detect I_{on} to read 0 or 1



Memory Summary

SRAM – Static Random Access Memory

- Speed: Very Fast
- Pros: High Speed, low standby power
- Cons: Expensive, lower density

DRAM – Dynamic Random Access Memory

- Speed: Fast
- Pros: Cost-effective, higher storage density
- Cons: Needs refreshing, higher power usage

Flash Memory

- Speed: Slow
- Pros: Non-Volatile, high density
- Cons: Limited write/erase cycles, slow

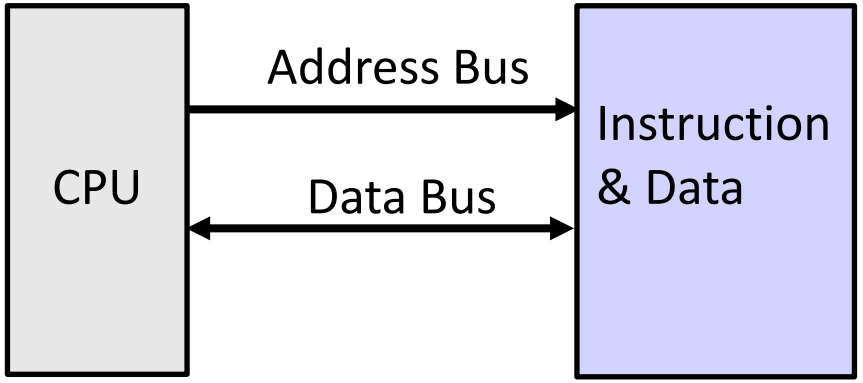
Volatile (Non-persistent)

Non-Volatile (Persistent)

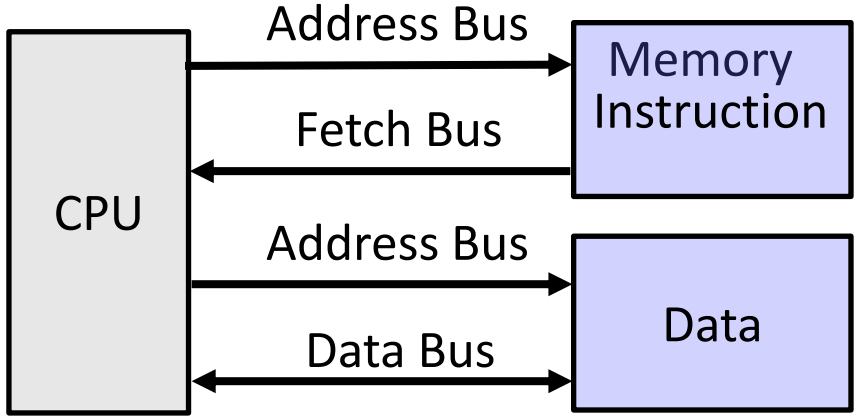
Memory System Architectures

- Two types of information are found in a typical program code:
  - Instruction codes for execution
  - Data that is used by the instruction codes

- Two classes of memory systems designed to store this information:



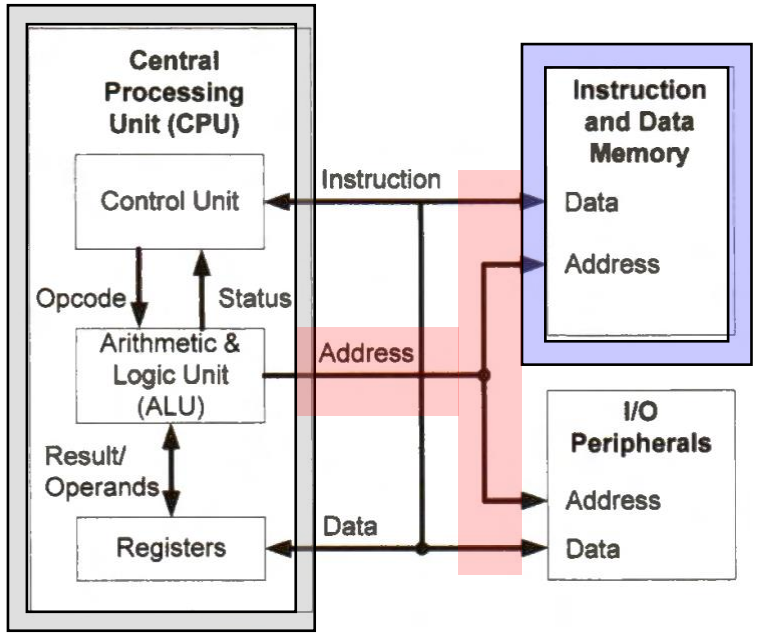
Von Neumann Architecture



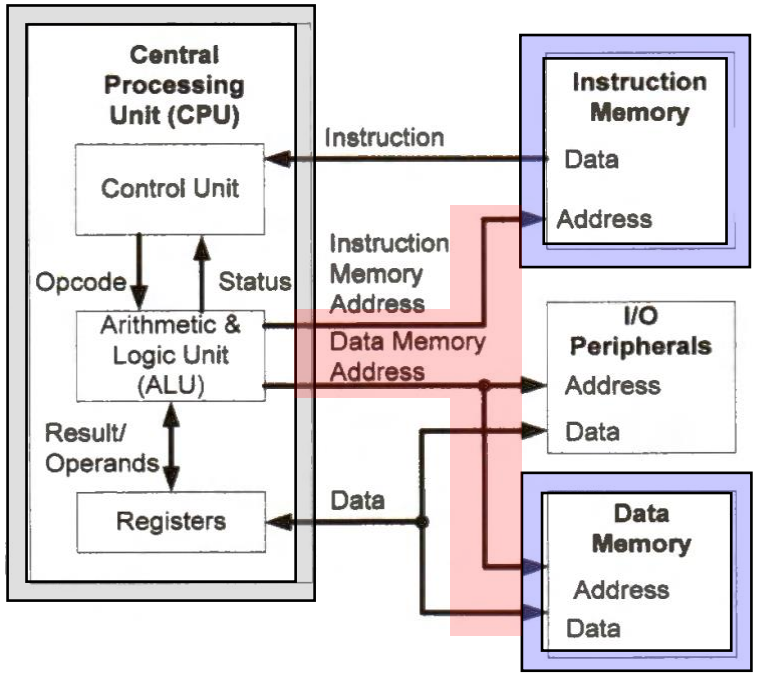
Harvard Architecture

Von Neumann vs. Harvard

- Harvard allows two simultaneous memory fetches.
- Most DSPs use Harvard architecture for streaming data: greater memory bandwidth
- Harvard cannot deal with self-modified code



Von Neumann Architecture



Harvard Architecture [1]

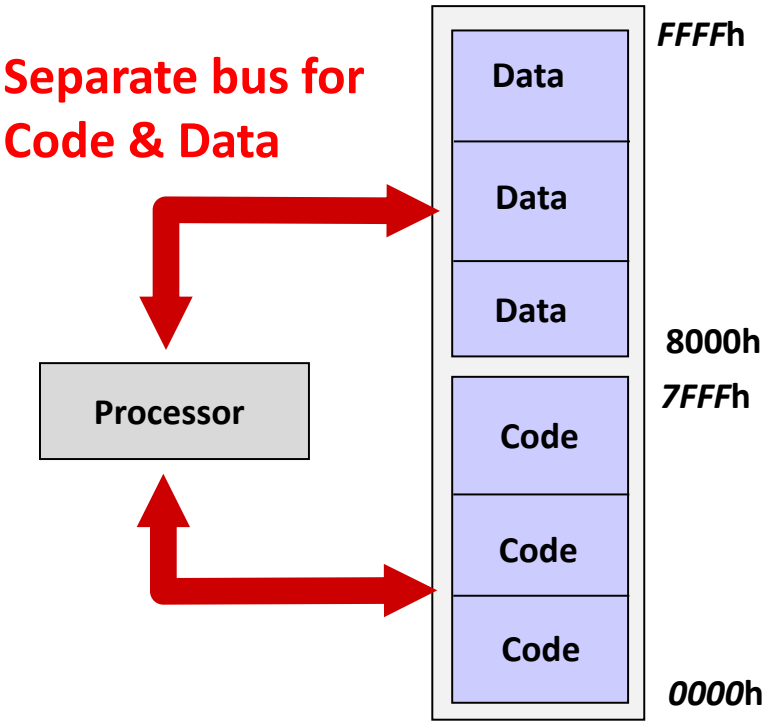
[1] Embedded Systems with ARM Cortex-M Microcontrollers in Assembly Language and C, page 5

Harvard Architecture

The Harvard architecture utilizes separate instruction bus and data bus.
Code and data may still share the same memory space.

Consequence and tradeoff:

- Allow code and data access at the same time
- Provide better support for instruction pipeline operations and shorter instruction execution time
- Allow different sizes of data and instructions to be used which results in more flexibility
- Do not incur any code corruption by data which makes the operations more robust
- More sophisticated hardware glue logic is required to support multiple interface buses

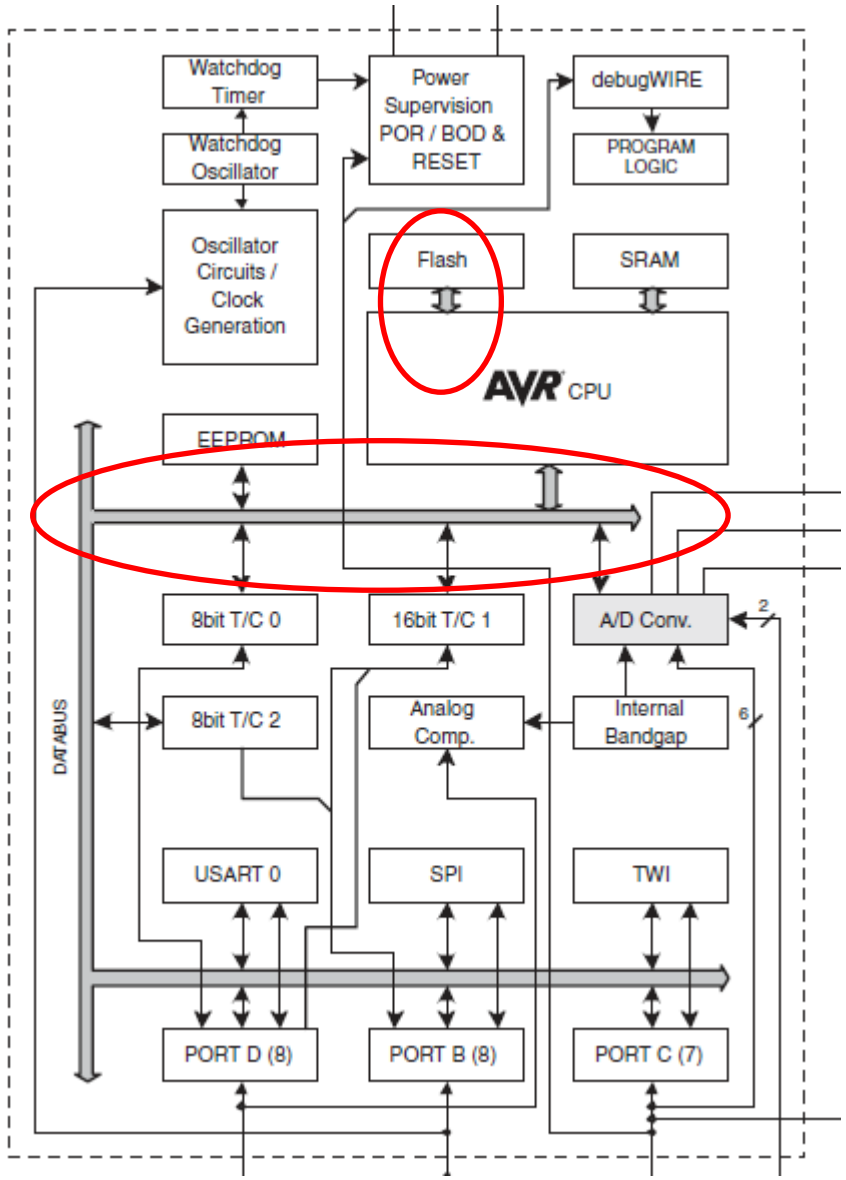


Harvard Architecture, an Example

AVR microcontroller (Microchip)

- Separate memories and buses for program and data.
- Instructions in the program memory are executed with a single level pipelining.

- Harvard architecture

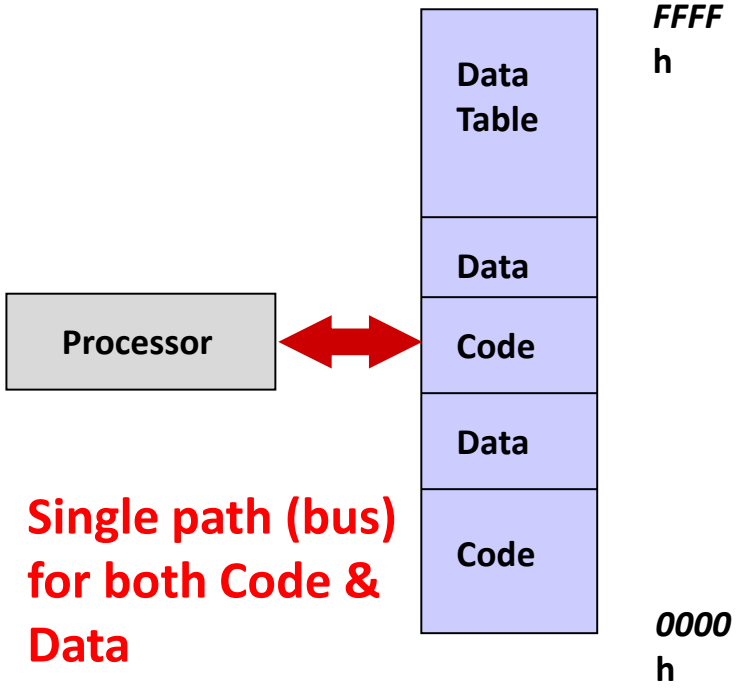


Von Neumann Architecture

The von Neumann architecture utilizes only one memory bus for both instruction fetching and data access

Consequence and tradeoff :

- Simplifies the hardware and glue logic design
- More efficient use of memory
- More flexible programming style (can include self-modified code)
- Instructions and data can reside on the same physical memory chip
- Data may overwrite instructions (e.g. due to program bug)
- Need memory protection (e.g. hardware-based MPU)
- Bottleneck in code and data transfer

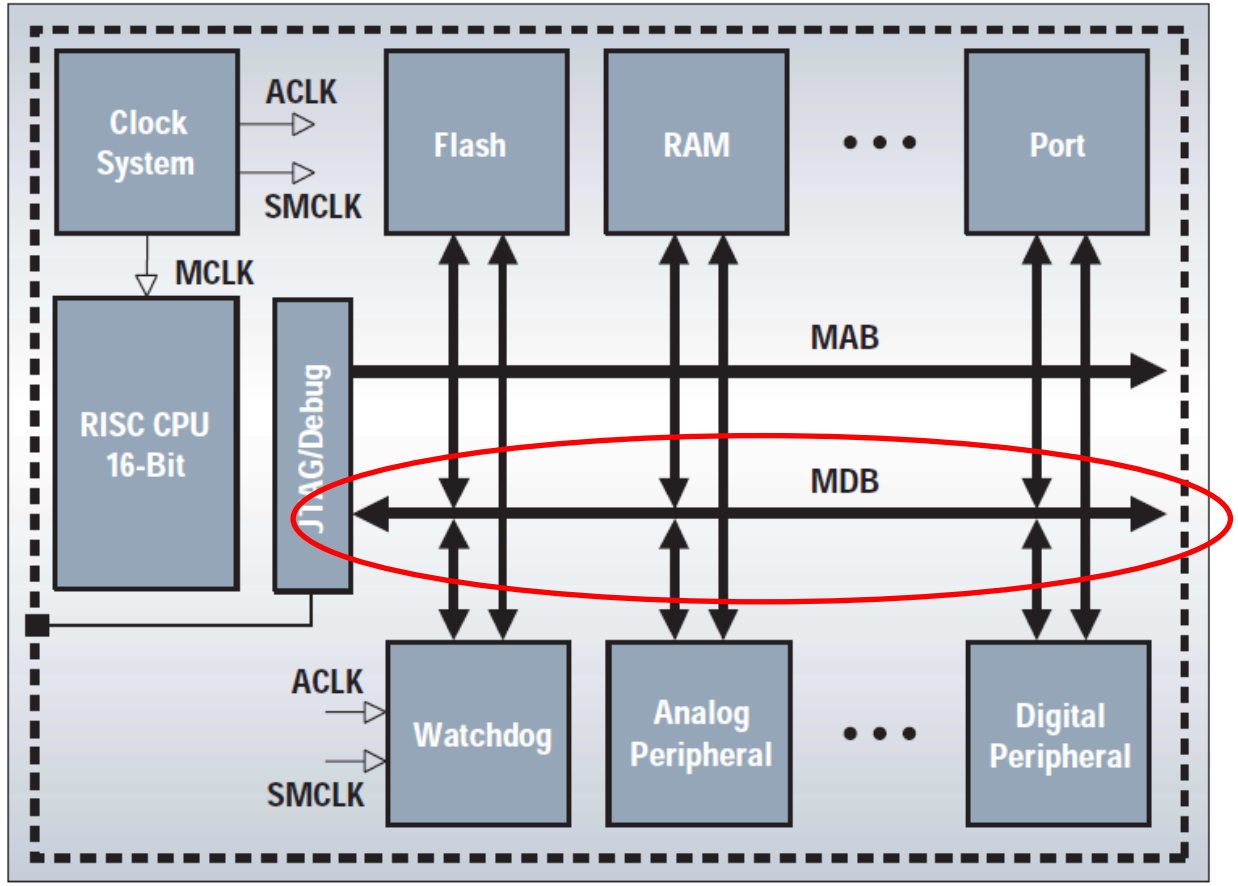


Von Neumann Architecture, an Example

MSP430, Texas Instruments

- All program, data memory and peripherals share a common bus structure.
- Consistent CPU instructions and addressing modes are used.

- Von Neumann Architecture



Performance Metrics

How we compare and classify CPU and microcontrollers?

Performance metrics are NOT easy to define and mostly application depended.

Electrical:

- Power Consumptions
- Voltage Supply
- Noise Immunity
- Sensitivity

Computation:

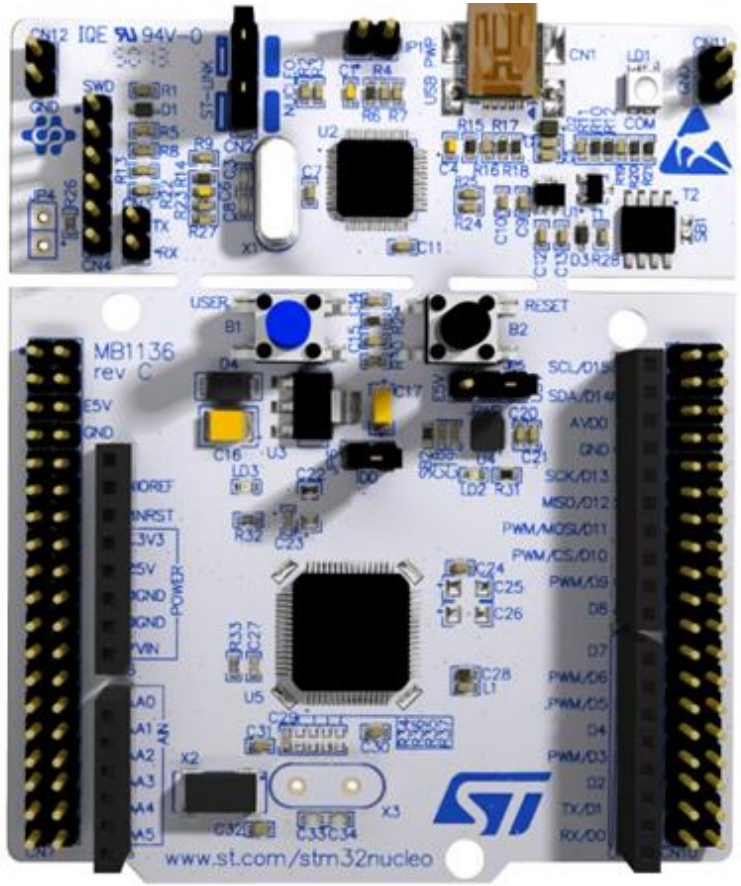
- Clock Speed
- MIPS (Millions of Instructions Per Sec)
- Latency
  - Lateness of the response
  - Lag between the begin and the end of the computation
- Throughput
  - Tasks per second
  - Byte per second

Goal: find the best tradeoff

power consumptions vs. performances

Microcontroller Platform for This Course

NUCLEO-L476RG Board



Microcontroller Platform for This Course

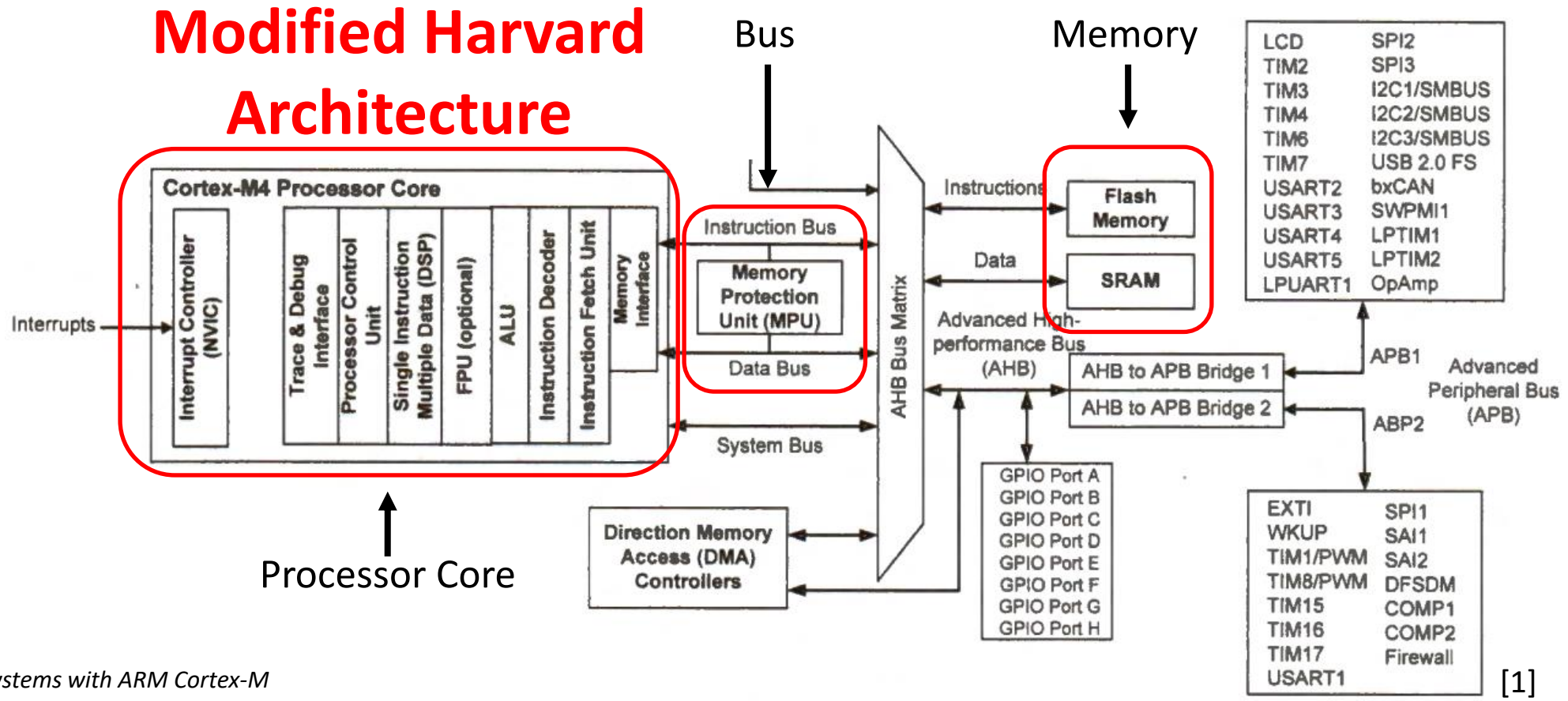
NUCLEO-L476RG Board

Features:

- STM32L476RG MCU – Arm® Cortex®-M4 32-bit RISC
  - 80 MHz with 1 Mbyte of Flash memory
  - Up to 128KB of SRAM
  - Floating point unit
- Peripherals
  - 3x 12-bit ADC 5 Msps
  - 2x 12-bit DAC output channels
  - 3x I2C
  - 5x USARTs
  - 3x SPIs
  - 14-channel DMA controller

Interaction: Memory System Architecture

Which architecture as been implemented on the Cortex-M4 Processor? Explain!



[1]

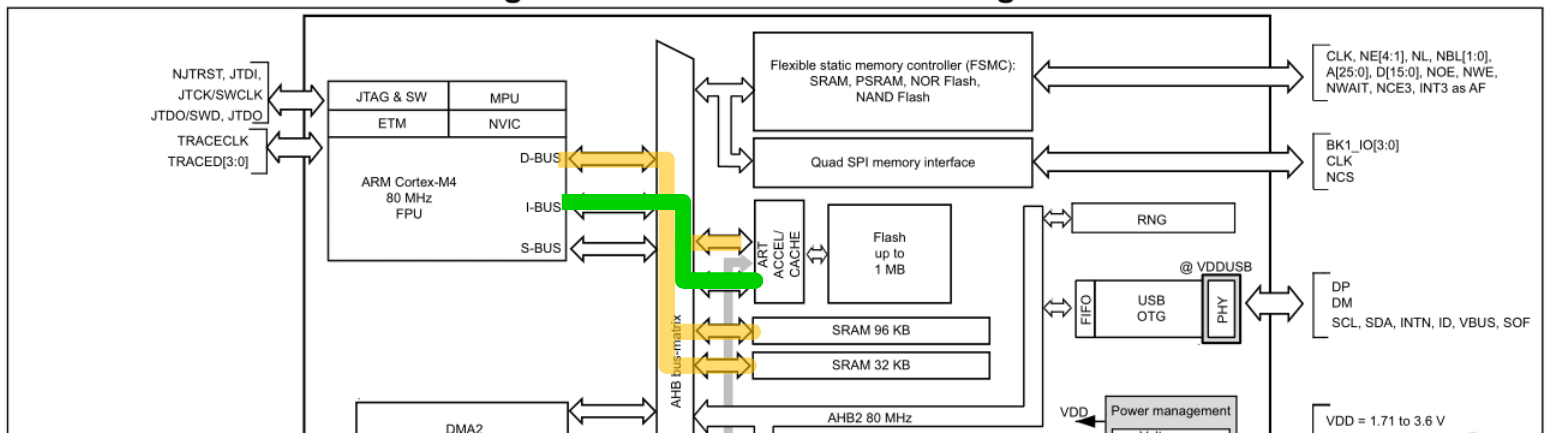
[1] Embedded Systems with ARM Cortex-M Microcontrollers in Assembly Language and C, page 55

STM32L476, ARM Cortex-M4

D-Bus: Data Bus

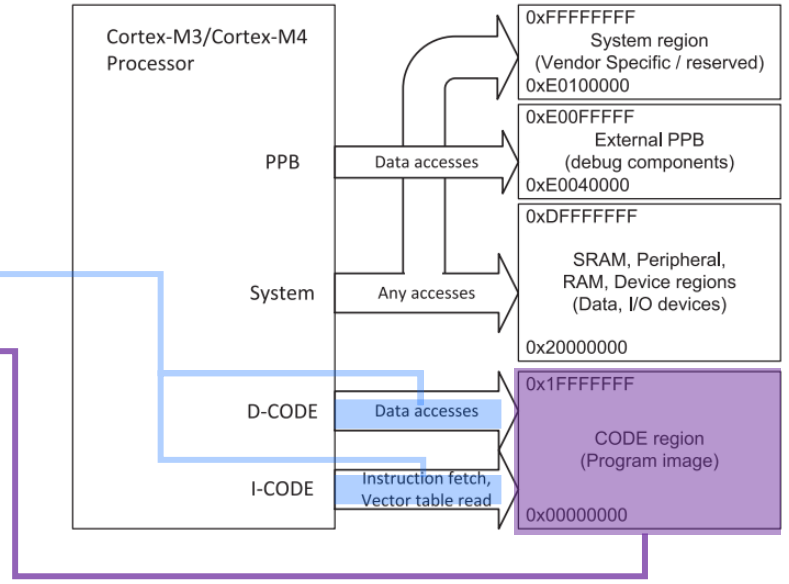
I- Bus: Instruction Bus

Figure 1. STM32L476xx block diagram



Modified Harvard:

- Two separate buses
- Unified memory space
- S-bus for Peripherals
- …



Storage: Memory Map

Memory in STM32L476

Available memory:

- The processor used in this lecture has up to 1MB flash memory, 128kB SRAM and 32kB ROM (Read Only Memory).

Memory Purposes:

- The memory in a microcontroller is used to store data (instructions and data), and to control and configure the hardware (circuit control at CMOS level). **Controlling peripherals by manipulating register values, which in turn control hardware at a low level.**

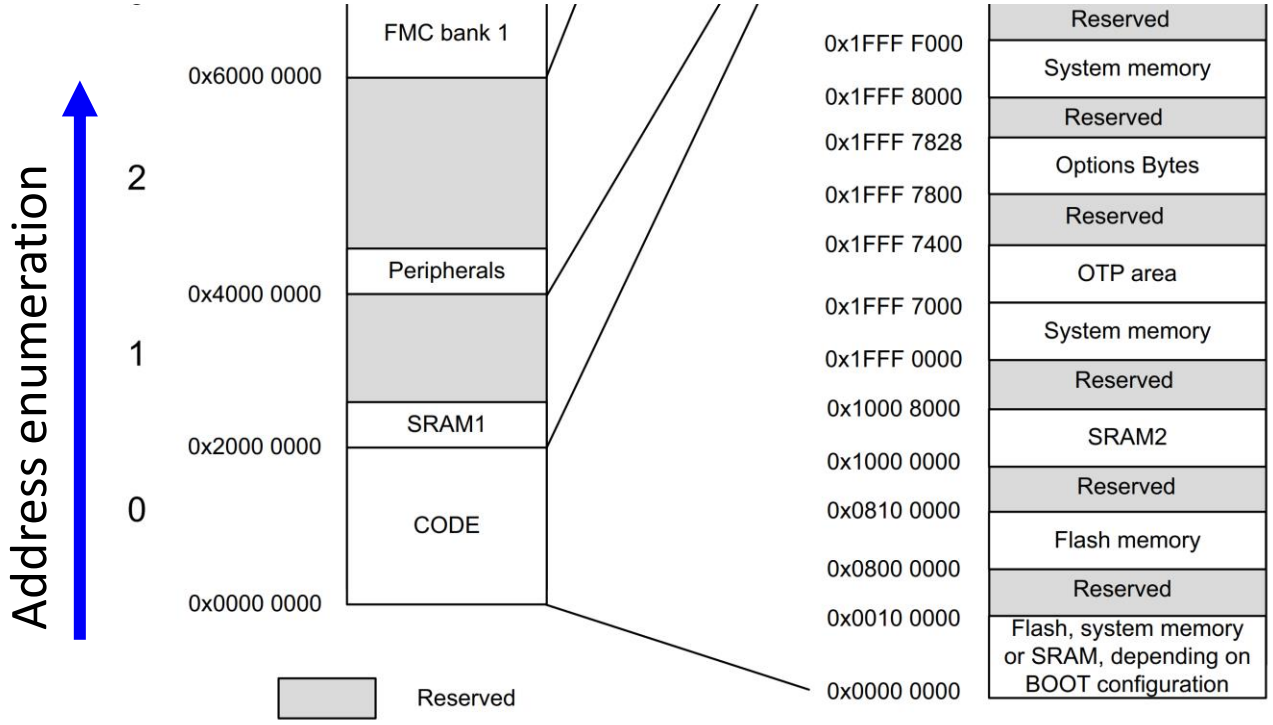
Address space:

- The processor **uses 32 bit addresses** as all the ARM Cortex-M Microcontrollers. Therefore, the addressable memory space is 4 GByte (= 2^{32} Byte) as each memory location corresponds to 1 Byte.
- The address space is used to address the memories (reading and writing)
- The address space is partitioned into zones, each one with a dedicated use

Memory Map in STM32L476

Memory Map:

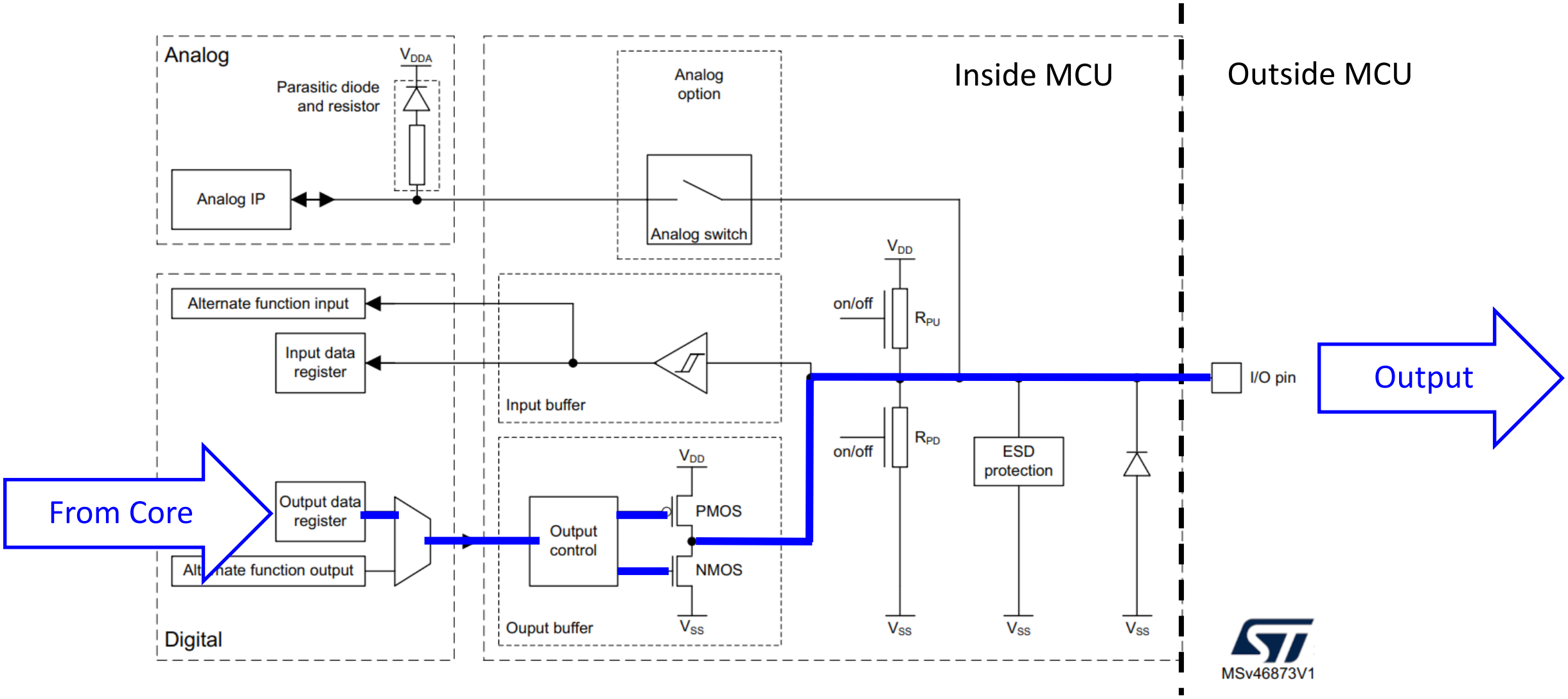
- The memory map defines which register can be found on which address.
- The map is realized in hardware and cannot be changed.



Part of the memory map of the STM32L476

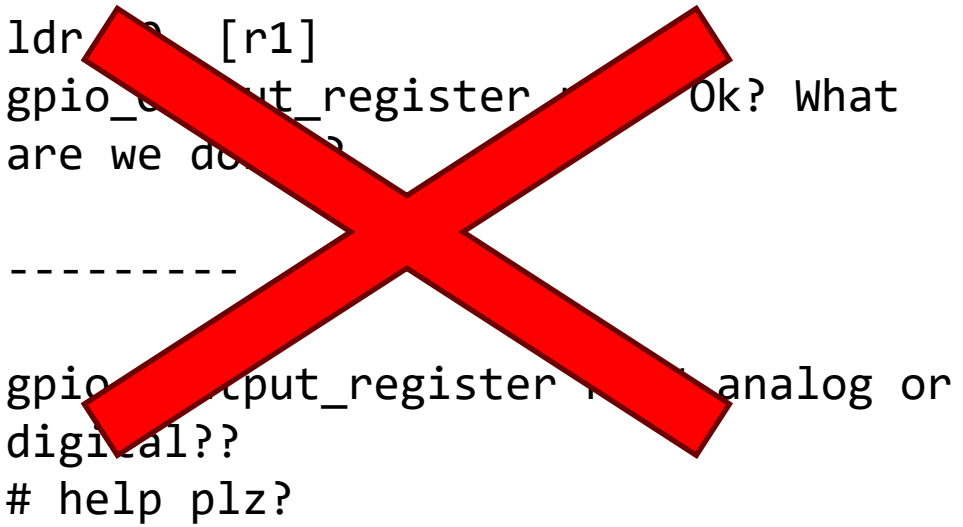
Memory Mapped I/O

GPIO STM32L476 – Remember?



How can we talk to the hardware?

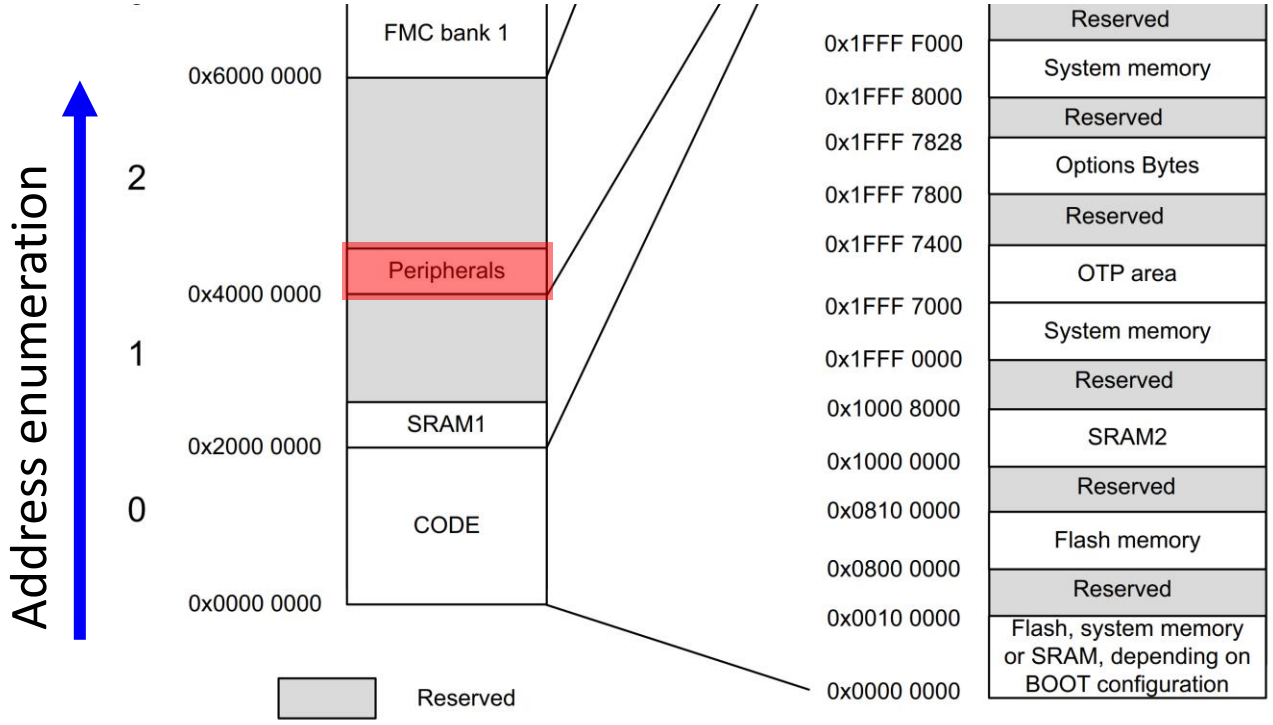
- Naïve Idea: *Add custom instructions?*
  - Would make CPU architecture more complex.
  - Not very flexible – What if we add new hardware in the future?
  - Less portable



Can we use existing instructions?

Memory Mapped I/O

- *Memory-mapped I/O* is a technique where I/O devices are assigned memory addresses, allowing the CPU to interact with them using standard memory read and write instructions (`ldr`,`str`,`...` etc.)

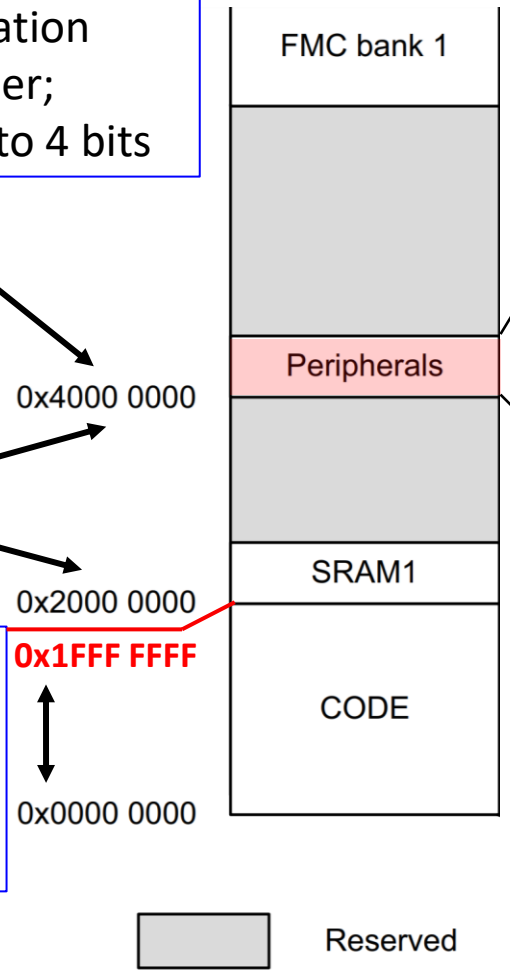


Part of the memory map of the STM32L476

Memory Map in STM32L476 – Peripherals

Memory map:

Hexadecimal representation of a 32-bit binary number; each digit corresponds to 4 bits



diff. = 0x1FFF FFF → 2^{29} different addresses

capacity = 2^{29} Byte = 512 MByte

Bus	Boundary address	Size (bytes)	Peripheral
AHB3	0xA000 1000 - 0xA000 13FF	1 KB	QUADSPI
	0xA000 0000 - 0xA000 0FFF	4 KB	FMC
AHB2	0x5006 0800 - 0x5006 0BFF	1 KB	RNG
	0x5004 0400 - 0x5006 07FF	129 KB	Reserved
	0x5004 0000 - 0x5004 03FF	1 KB	ADC
	0x5000 0000 - 0x5003 FFFF	16 KB	OTG_FS
	0x4800 2000 - 0x4FFF FFFF	~127 MB	Reserved
	0x4800 1C00 - 0x4800 1FFF	1 KB	GPIOH
	0x4800 1800 - 0x4800 1BFF	1 KB	GPIOG
	0x4800 1400 - 0x4800 17FF	1 KB	GPIOF
	0x4800 1000 - 0x4800 13FF	1 KB	GPIOE
	0x4800 0C00 - 0x4800 0FFF	1 KB	GPIOD
	0x4800 0800 - 0x4800 0BFF	1 KB	GPIOC
	0x4800 0400 - 0x4800 07FF	1 KB	GPIOB
	0x4800 0000 - 0x4800 03FF	1 KB	GPIOA

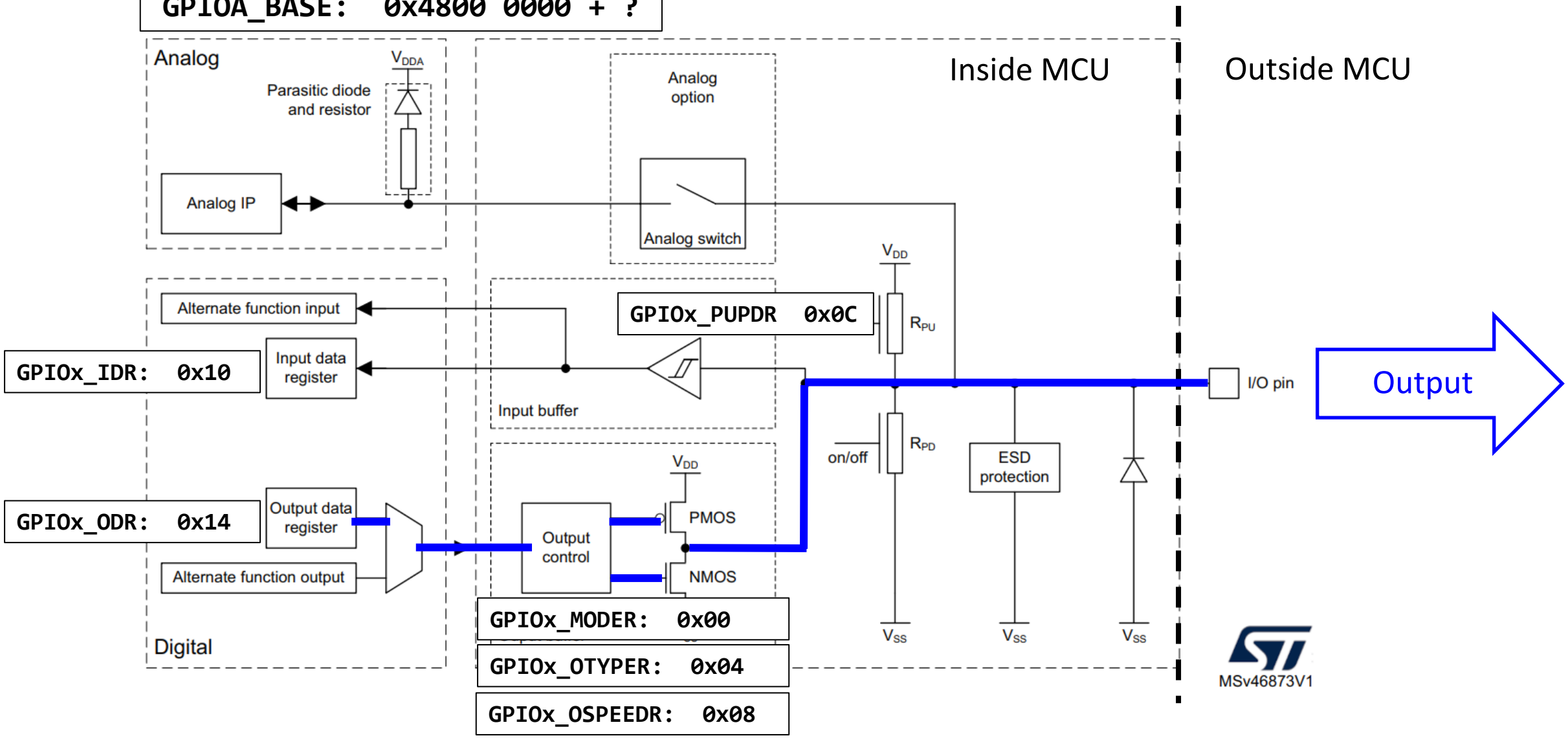
Table 40. GPIO register map and reset values (continued)

Offset	Register name	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
0x14	GPIOx_ODR (where x = A..I)	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	OD15	OD14	OD13	OD12	OD11	OD10	OD9	OD8	OD7	OD6	OD5	OD4	OD3	OD2	OD1	OD0
	Reset value																	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	GPIOx_BSRR	BR15	BR14	BR13	BR12	BR11	BR10	BR9	BR8	BR7	BR6	BR5	BR4	BR3	BR2	BR1	BR0	BS15	BS14	BS13	BS12	BS11	BS10	BS9	BS8	BS7	BS6	BS5	BS4	BS3	BS2	BS1	BS0

ODR = Output Data Register

GPIOA STM32L476 – Memory Map

GPIOA_BASE: 0x4800 0000 + ?



GPIOx_PUPDR 0x0C

GPIOx_IDR: 0x10

GPIOx_ODR: 0x14

GPIOx_MODER: 0x00

GPIOx_OTYPER: 0x04

GPIOx_OSPEEDR: 0x08

Memory Map in STM32L476 (more in the lab!)

How do we toggle LD2?

- Check Schematic of the NUCLEO-L476RG Board
- LD2 connected to PA5 = Port A GPIO 5
- Configure GPIO for output:
  - Set mode to Output - Bit 5 of GPIOA_MODER
- Write value to Output Register – Bit 5 of GPIOA_ODR

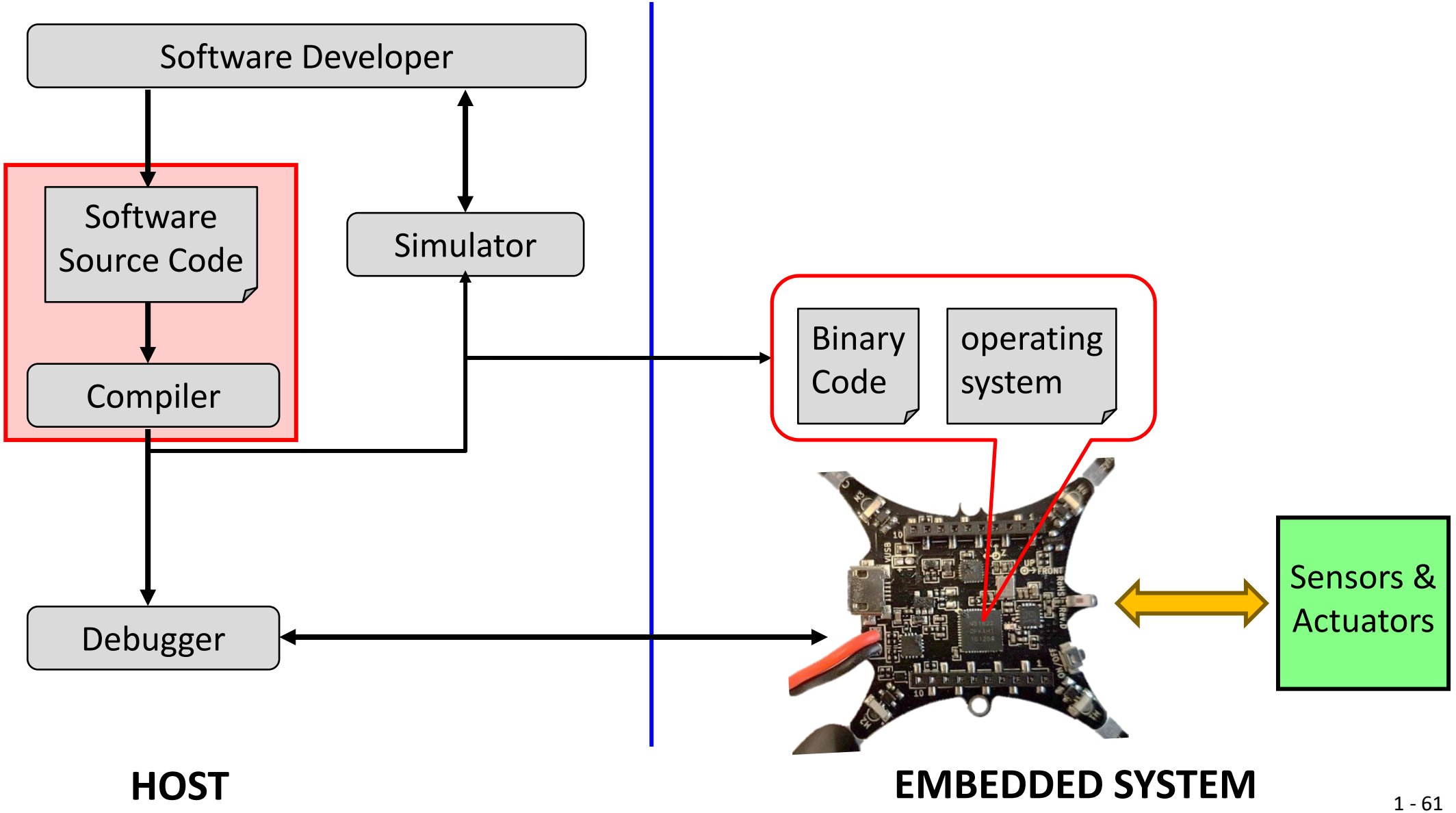
Table 40. GPIO register map and reset values (continued)

Offset	Register name	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
0x14	GPIOx_ODR (where x = A..I)	res.	res.	res.	res.	res.	res.	res.	res.	res.	res.	res.	res.	res.	res.	res.	res.	OD15	OD14	OD13	OD12	OD11	OD10	OD9	OD8	OD7	OD6	OD5	OD4	OD3	OD2	OD1	OD0
	Reset value																	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	GPIOx_BSRR																																

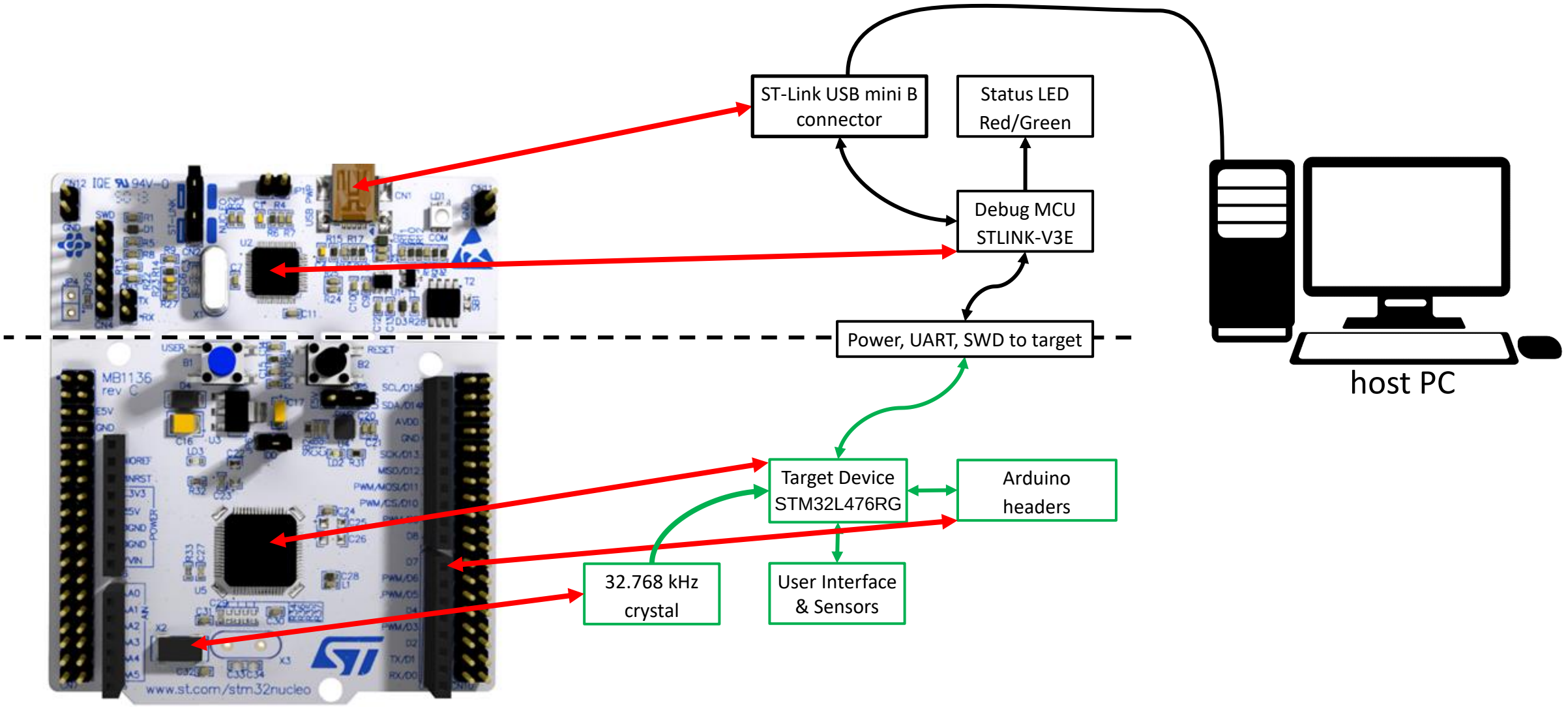
```
volatile uint32_t* GPIOA_ODR =(volatile uint32_t*) 0x48000014; // GPIOA_ODR register

// set 5th bit in ODR register, this turns the pin “on"
*GPIOA_ODR |= (1<<5);
// clear 5th bit in ODR register, this turns the pin “off"
*GPIOA_ODR &= ~(1<<5);
```

Embedded Software Development

Embedded Software Development



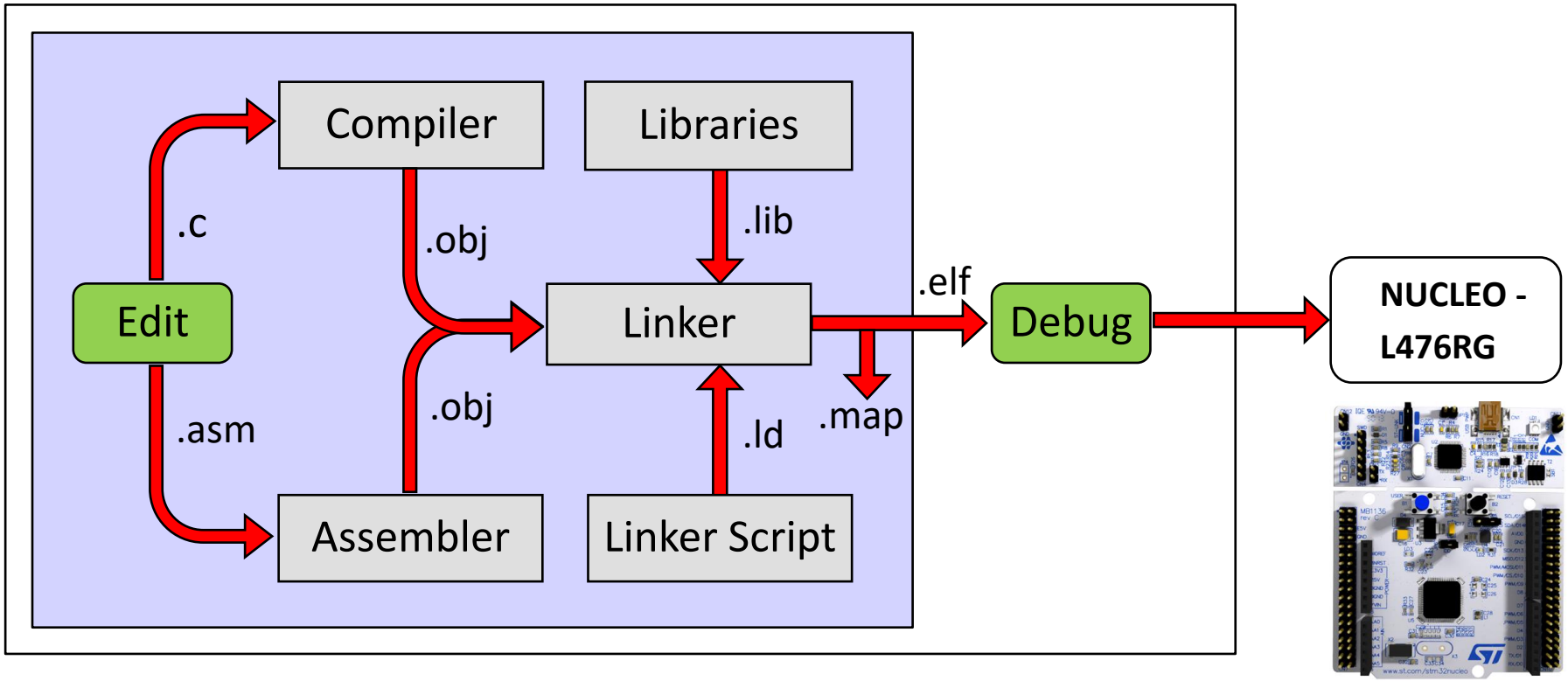
Software Development with NUCLEO-L476RG



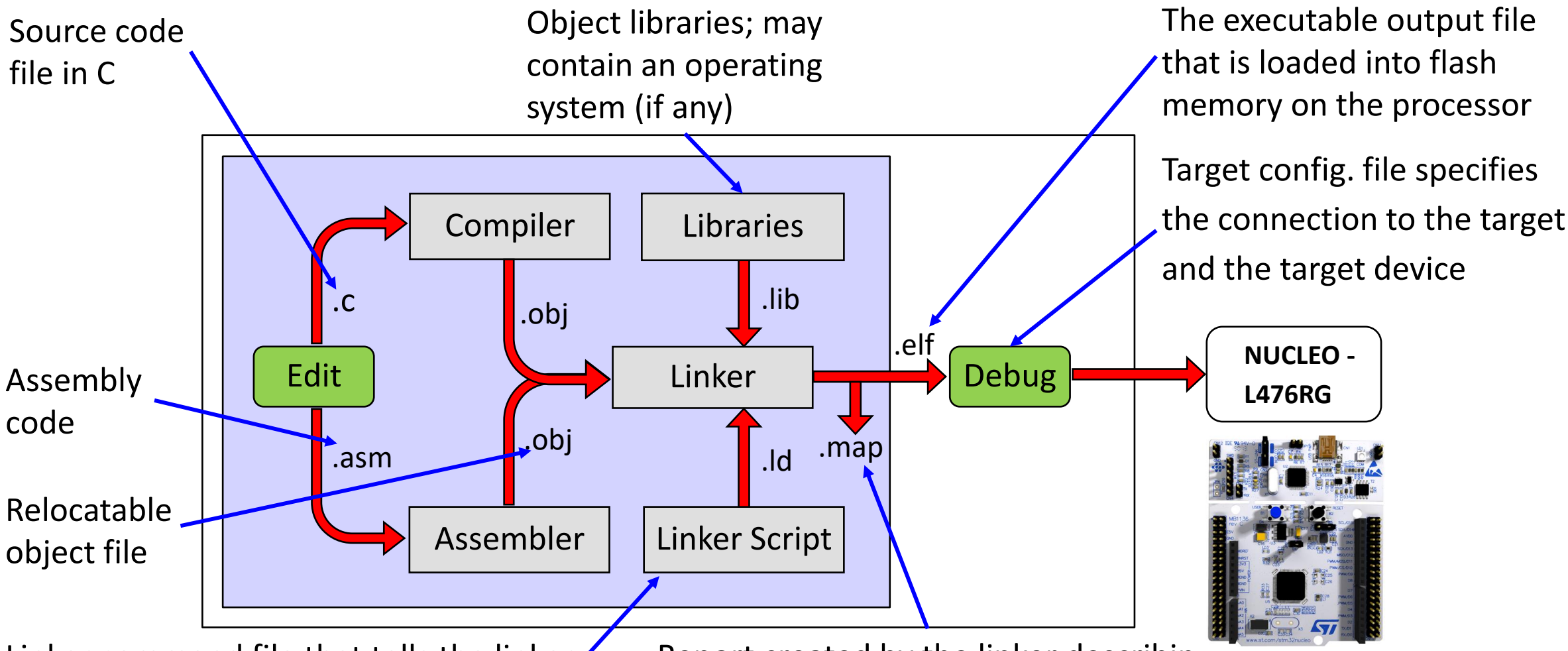
Software Development

Software development is nowadays usually done with the support of an IDE (Integrated Debugger and Editor / Integrated Development Environment)

- edit and build the code
- debug and validate



Software Development

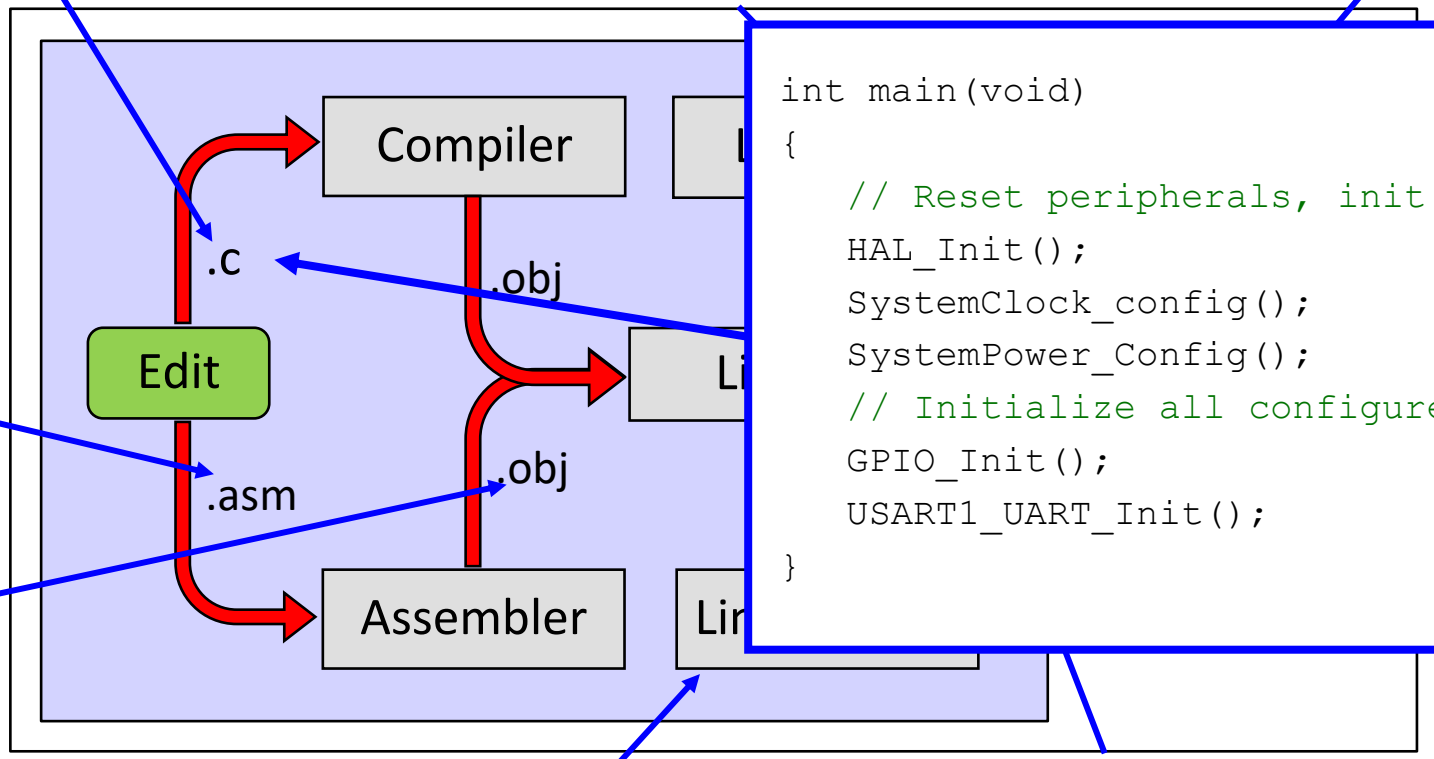


Software Development

Source code file in C

Object libraries; may contain an operating system (if any)

The executable output file that is loaded into flash memory on the processor




```
int main(void)
{
    // Reset peripherals, init flash and Systick
    HAL_Init();
    SystemClock_config();
    SystemPower_Config();
    // Initialize all configured peripherals
    GPIO_Init();
    USART1_UART_Init();
}
```

Assembly code

Relocatable object file

Linker command file that tells the linker how to allocate memory and to stitch the object files and libraries together.

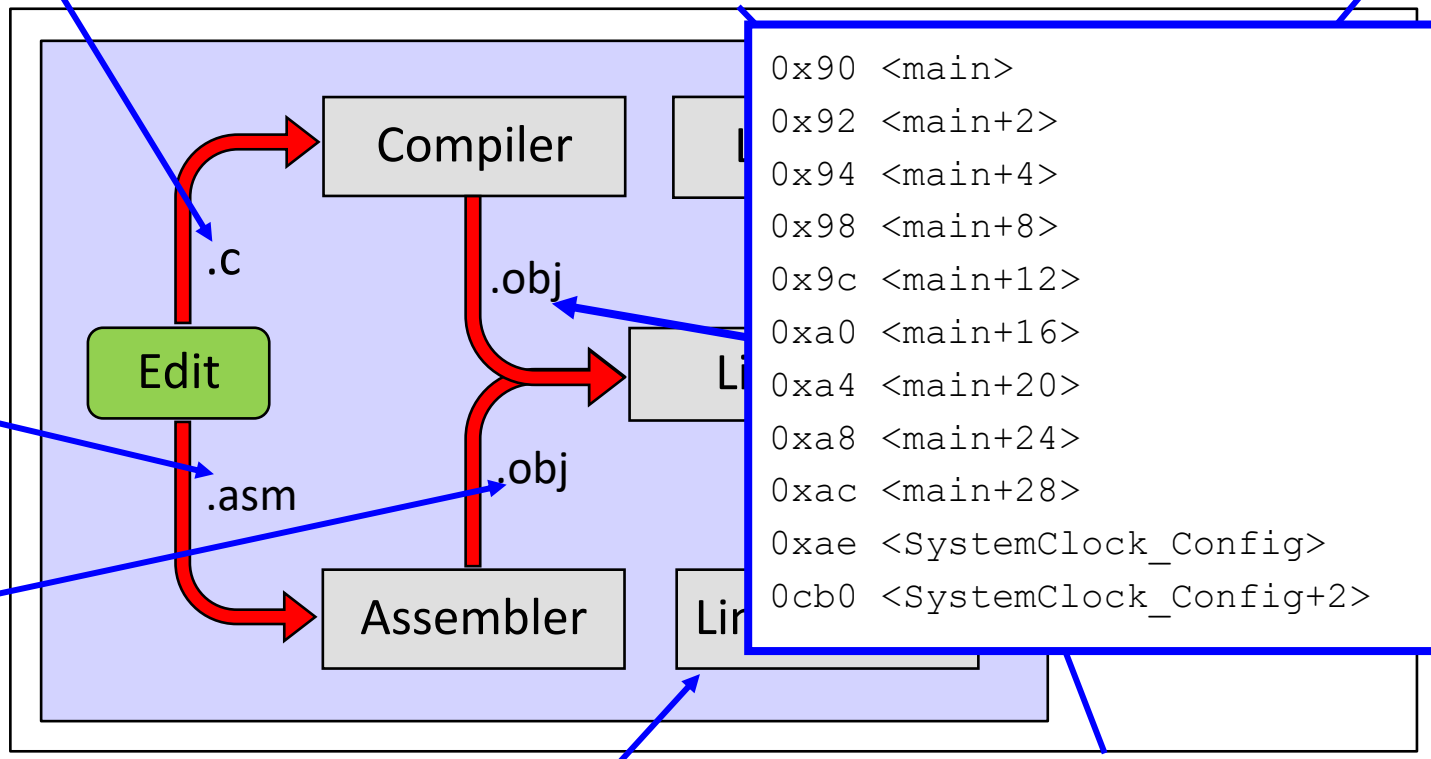
Report created by the linker describing where the program and data sections are located in memory.

Software Development

Source code file in C

Object libraries; may contain an operating system (if any)

The executable output file that is loaded into flash memory on the processor




```
0x90 <main>                          push    {r7,lr}
0x92 <main+2>                        add     r7, sp, #0
0x94 <main+4>                        bl      0x94 <main+4>
0x98 <main+8>                        bl      0xa8 <main+8>
0x9c <main+12>                       bl      0x9c <main+12>
0xa0 <main+16>                       bl      0xa0 <main+16>
0xa4 <main+20>                       bl      0xa4 <main+20>
0xa8 <main+24>                       bl      0xa8 <main+24>
0xac <main+28>                       b.n     0xac <main+28>
0xae <SystemClock_Config>            push    {r7,lr}
0cb0 <SystemClock_Config+2>          sub     sp,#120
```

Linker command file that tells the linker how to allocate memory and to stitch the object files and libraries together.

Report created by the linker describing where the program and data sections are located in memory.

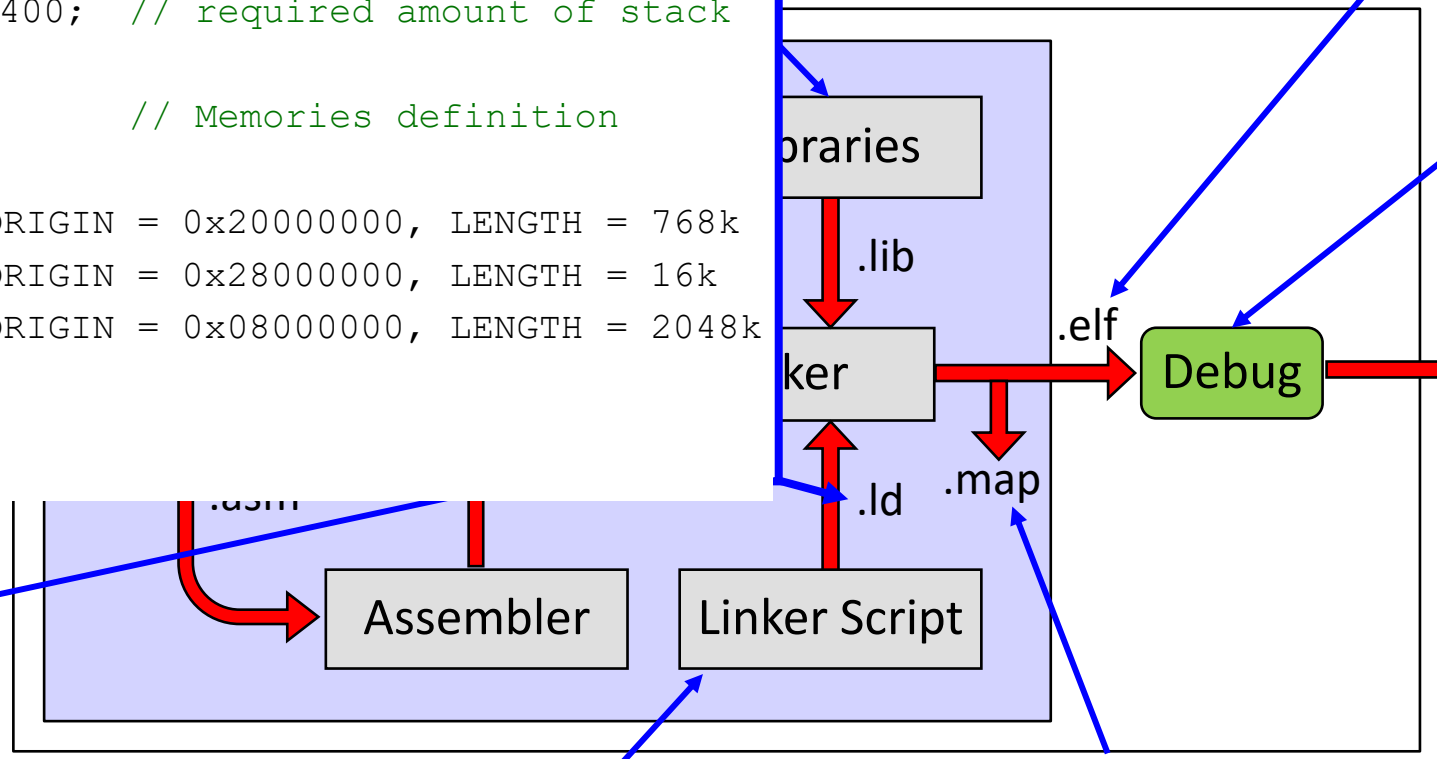
Software Development

```
ENTRY(Reset_Handler)        // Entry Point
// Highest address of the user mode stack
_estack = ORIGIN(RAM)+ LENGTH(RAM);
_Min_Heap_Size = 0x200;     // required amount of heap
_Min_Stack_Size = 0x400;    // required amount of stack

MEMORY                      // Memories definition
{
    RAM      (xrw)  : ORIGIN = 0x20000000, LENGTH = 768k
    SRAM4    (xrw)  : ORIGIN = 0x28000000, LENGTH = 16k
    FLASH    (rx)   : ORIGIN = 0x08000000, LENGTH = 2048k
}
…
```

es; may
perating
y)

The executable output file that is loaded into flash memory on the processor

Target config. file specifies the connection to the target and the target device



Relocatable object file

Linker command file that tells the linker how to allocate memory and to stitch the object files and libraries together.

Report created by the linker describing where the program and data sections are located in memory.

What did You Learn?

- ✓ Microcontroller vs. Microprocessors
  - ✓ What is an Instruction RISC and CISC, 8-16-32 bits.
- ✓ Architecture of a Microcontroller
  - CPU, ALU, Peripherals
- ✓ Memory Architecture
  - Von Neumann architecture, Harvard architecture
- ✓ Different Types of Storage
  - Registers, SRAM, DRAM, Flash
- ✓ Memory map
  - Addresses, Registers
- ✓ Embedded Software Development
  - Compiler, Assembler, Linker