Cortex-M Architecture

Computer Engineering 1

Motivation

School of Engineering
InES Institute of Embedded Systems

- **ARM (Cortex-M)**

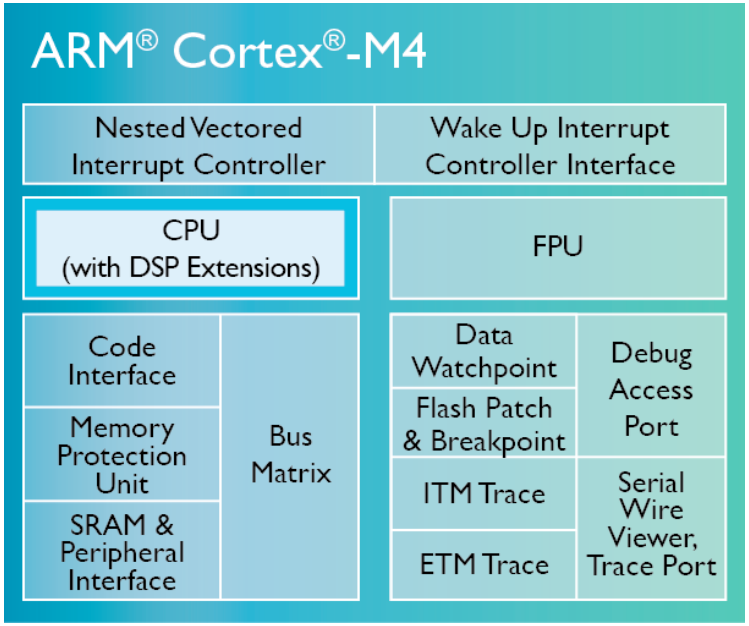
1985

ARM1
25k Transistors
5 MHz

1'400 x transistors

36 x clock speed

2014



ARMv7 - Cortex-M4
35 Mio Transistors
180 MHz
4 mm^2

Agenda

School of Engineering
InES Institute of Embedded Systems

- **Hardware Platform**
  - STM32F4 and evaluation board, ARM Processor Portfolio
- **CPU Model**
  - Register, ALU, Flags, Control Unit
- **Instruction Set**
  - Assembly, Instruction Types, Cortex-M0
- **Program Execution**
  - Fetch and Execute
- **Memory Map**
  - ARM, ST, CT-Board
- **Integer Types**
  - Sizes, Little Endian vs. Big Endian, Alignment
- **Object File Sections**
  - Code, Data, Stack

Learning Objectives

School of Engineering
InES Institute of Embedded Systems

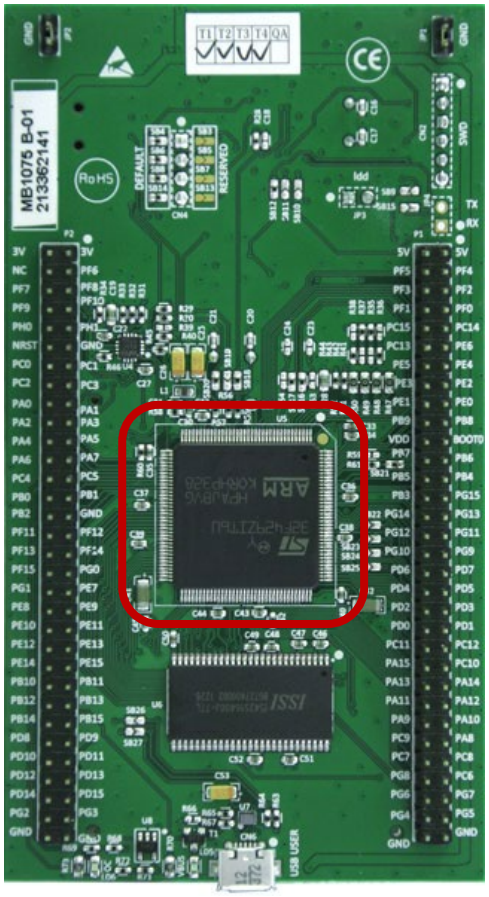
At the end of this lesson you will be able

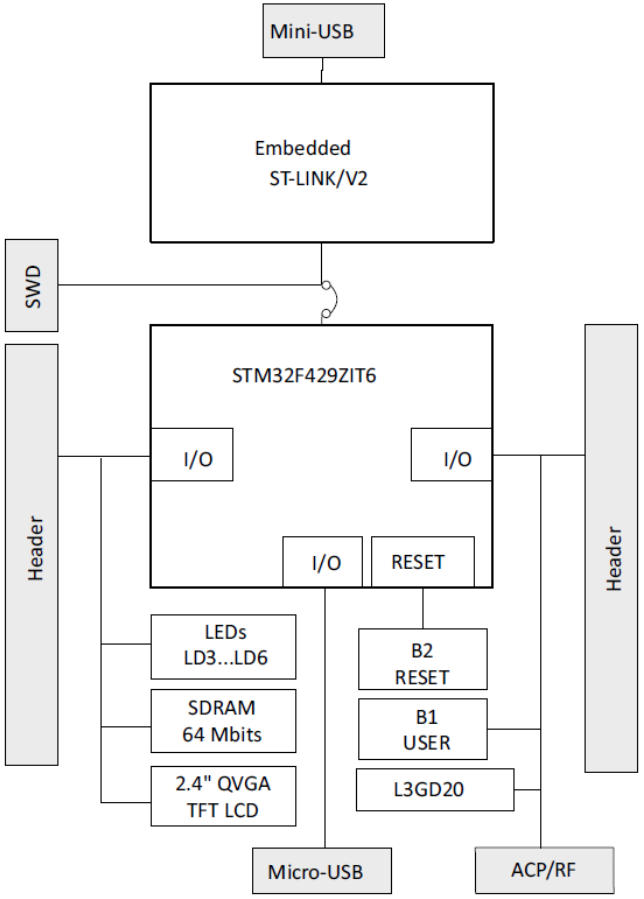
- to describe what an 'Instruction Set Architecture' is
- to outline the Cortex-M architecture and enumerate the main components and their functions
- to enumerate the instruction type categories
- to understand the structure of the Cortex-M instruction set
- to explain how a processor executes a program
- to recall the registers of the Cortex-M, their layout and their functions
- to explain and draw a memory map
- to calculate the size in bytes for a memory block given by its start and end address
- to determine the end address of a memory block given by the start address and the number of bytes
- to understand that sizes of integer types in C depend on the architecture and that portability can be enhanced by using the C99 types in stdint.h
- to explain the difference between 'little endian' and 'big endian' and to show how multi-byte integer values are mapped to individual bytes
- to list the three typical memory sections of an object file and to explain their content

Hardware Platform: STM32F4-Discovery

School of Engineering
InES Institute of Embedded Systems

- **Evaluation board STM32F4-DISCO**





source: STMicroelectronics

Hardware Platform: STM32F429I

School of Engineering
InES Institute of Embedded Systems

System
- Power Supply
- Oscillators 32 kHz + 4 to 26 MHz
- PLL [1]
- Clock Control
- RTC/AWU [2]
- SysTick Timer
- 2 x Watchdog
- 114 I/Os
- CRC [3]

Control
- 2 x PWM Motor Control
- 8 x 16 Bit Timer
- 2 x 32 Bit Timer

[1] Phase Locked Loop
[2] Real Time Clock / Auto-Wakeup
[3] Cyclic Redundancy Check

ARM Cortex-M4 180 MHz
- FPU
- Nested Vector Interrupt Controller (NVIC)
- MPU
- JTAG/SW Debug

- True Random Number Generator (RNG)
- Crypto/Hash Processor
- 16-Chanel DMA

Analog
- 2 x 12-Bit DAC
- 3 x 12 Bit ADC
- Temperature Sensor

[4] One Time Programmable
[5] Secure Digital I/O Interface

- 2 Mbyte Flash
- 256 Kbyte SRAM
- LCD TFT Controller
- External Memory Controller
- Backup SRAM
- 512 OTP Bytes [4]

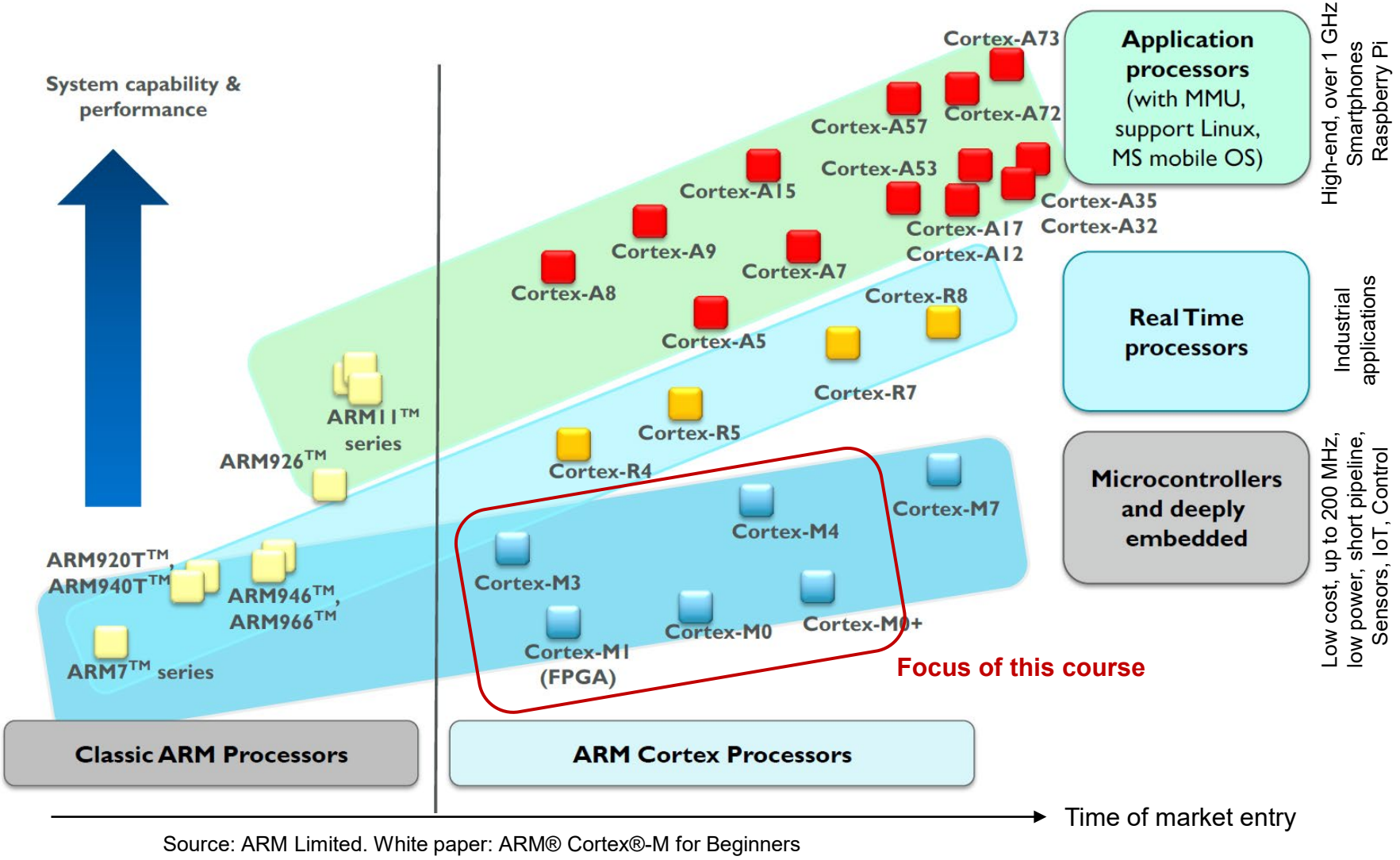
Connectivity
- Camera Interface
- 6 x SPI, 2x I^2C, 2 x I^2S
- Ethernet MAC 10/100
- 2 x CAN 2.0
- 2 x USB 2.0
- 4 x USART, 4 x UART
- 1 x SDIO [5]
- 1 X SAI [6]

[6] Serial Audio Interface

based on source of STMicroelectronics

Hardware Platform: ARM Processor Portfolio

School of Engineering
InES Institute of Embedded Systems

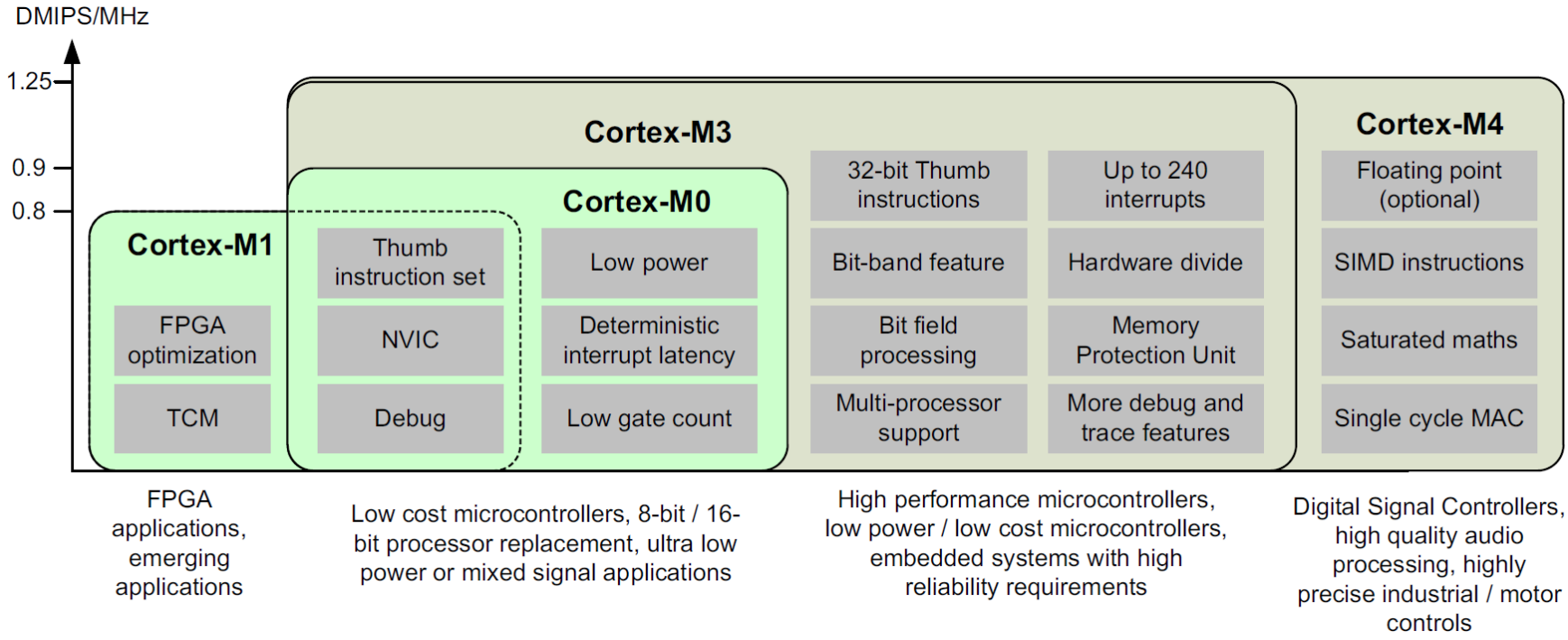


Source: ARM Limited. White paper: ARM® Cortex®-M for Beginners

Hardware Platform

School of Engineering
InES Institute of Embedded Systems

- **Cortex-M Processor Family**
  - Hardware used in this course is a Cortex-M4
  - But most of the time we only use the simpler Cortex-M0 subset



source: Joseph Yiu, The definitive Guide to the Cortex-M0

CPU Model

School of Engineering
InES Institute of Embedded Systems

- **Instruction Set Architecture (ISA)** → CT1

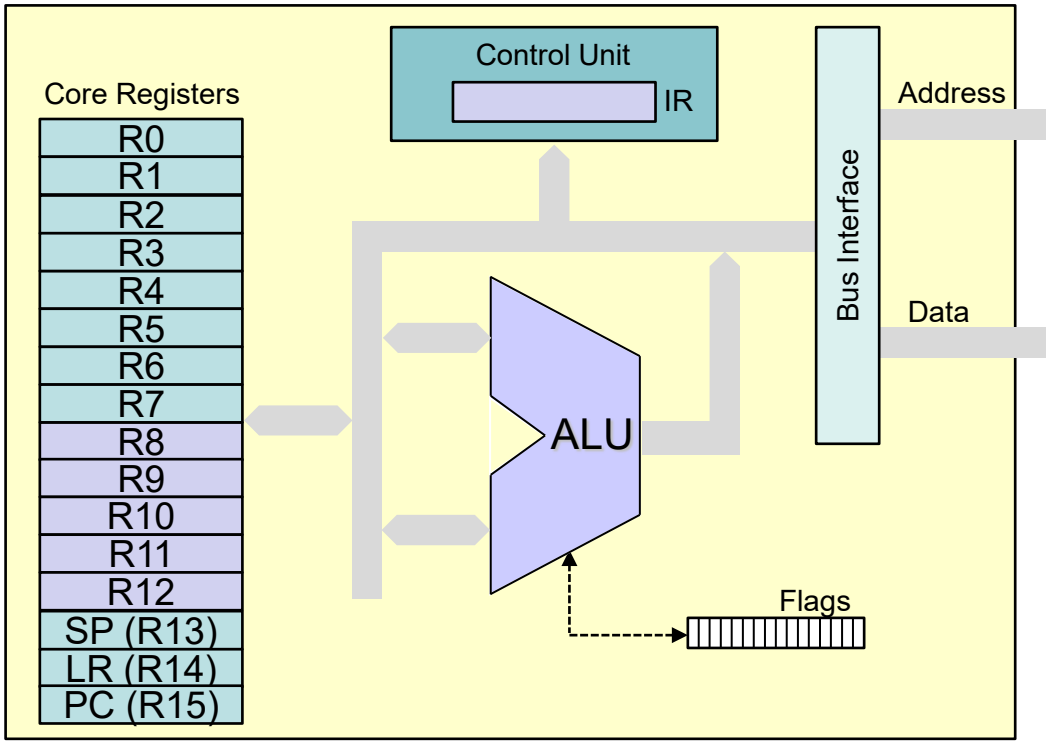
  What the programmer sees of a computer

  - Instruction Set
    - Available instructions?
  - Processing width
    - 8-bit/16-bit/32-bit?
  - Register set
    - How many registers? Which size?
  - Addressing modes
    - How can memory and IO be accessed?
  - ARM Cortex-M
    - ARMv6-M → Cortex-M0
    - ARMv7-M → Cortex-M3/M4 (Superset of ARMv6-M)

CPU Model

School of Engineering
InES Institute of Embedded Systems

- **CPU Components**
  - Core Registers
  - 32-bit ALU
  - Flags (APSR)
  - Control Unit with IR (Instruction Register)
  - Bus Interface

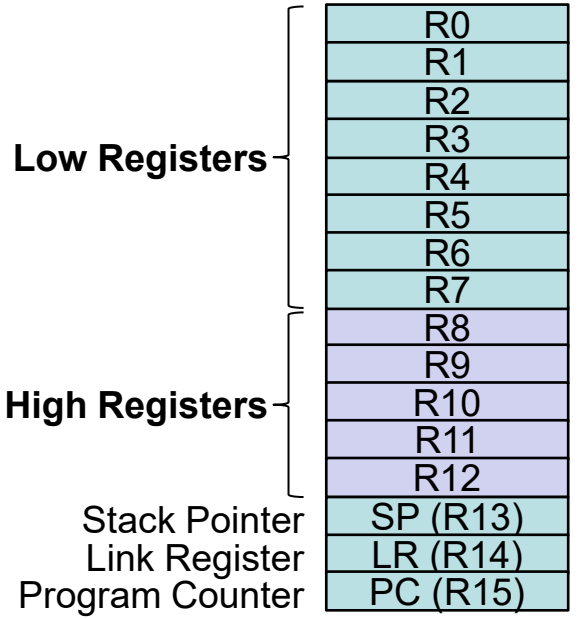


simple CPU model based on the Programmers' Model of the ARM Cortex-Mx CPUs [1)]

[1)] without pipelining

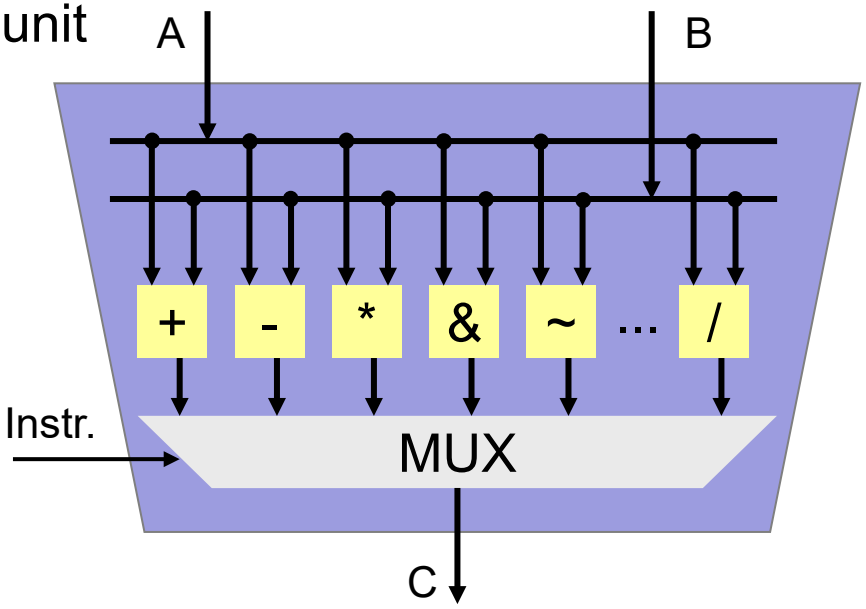
CPU Model

- **16 Core Registers**
  - Each 32-bit wide
  - 13 General-Purpose Registers
    - Low Registers R0 – R7
    - High Registers R8 – R12
    - Used for temporary storage of data and addresses
  - Program Counter (R15)
    - Address of **next** instruction
  - Stack Pointer (R13)
    - Last-In First-Out for temporary data storage
  - Link Register (R14)
    - Return from procedures



CPU Model

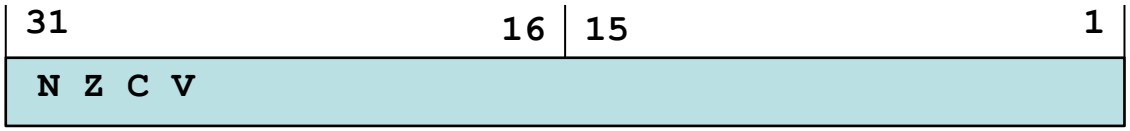
- **ALU – Arithmetic Logic Unit**
  - 32-bit wide data processing unit
    - inputs A and B
    - result C
  - integer arithmetic
    - addition / subtraction
    - multiplication / division
    - sign extension
  - logic operations
    - AND, NOT, OR, XOR
  - shift/rotate
    - left / right



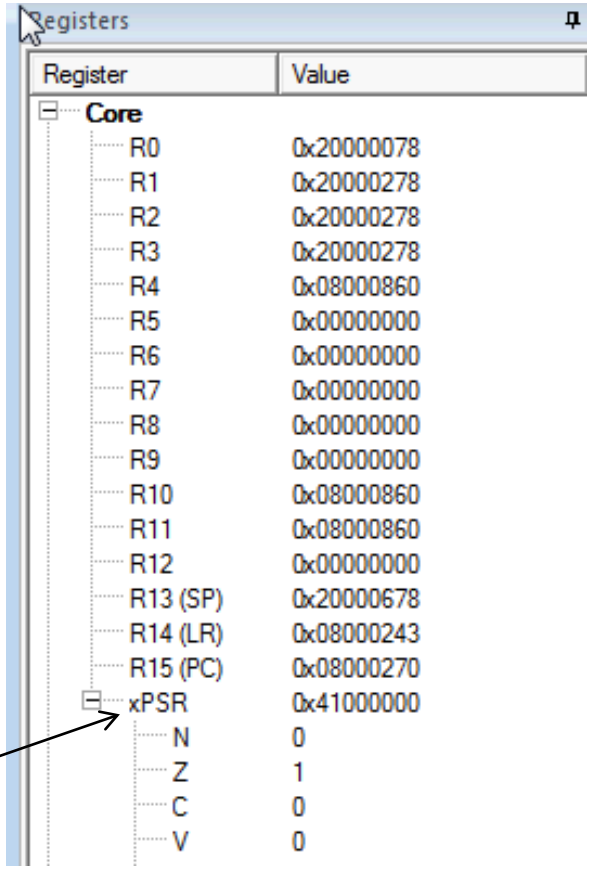
CPU Model

School of Engineering
InES Institute of Embedded Systems

- **APSR[1] or Flag-Register**
  - Bits set by CPU based on results in ALU



N	**Negative**
Z	**Zero**
C	**Carry**
V	**Overflow**



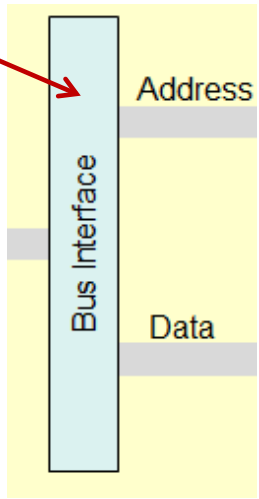
As seen in the IDE

[1] APSR: Application Processor Status Register

CPU Model

School of Engineering
InES Institute of Embedded Systems

- **Control Unit**
  - Instruction Register (IR)
    - Machine code (opcode) of instruction that is currently being executed
  - Controls execution flow based on instruction in IR
  - Generates control signals for all other CPU components
- **Bus Interface**
  - Interface between internal CPU bus and external system-bus
    - contains registers to store addresses

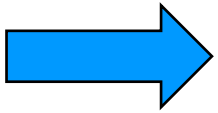


Instruction Set

School of Engineering
InES Institute of Embedded Systems

- **Processors interpret binary coded instructions**
  - But binary is hard for programming
  - Therefore instructions in human readable text form
    - → assembly
  - Assembler (tool) does the translation
    - assembly → binary

```
ADDS   R0,R0,R1
```



```
0001'1000'0100'0000
=
0x1840
```

Instruction Set

School of Engineering
InES Institute of Embedded Systems

- **Assembly Program**
  - Label (optional)
  - Operands
  - Instruction (Mnemonic)
  - Comment (optional)

Label	Instr.	Operands	Comments
demoprg	MOVS	R0,#0xA5	; copy 0xA5 into register R0
	MOVS	R1,#0x11	; copy 0x11 into register R1
	ADDS	R0,R0,R1	; add contents of R0 and R1
			; store result in R0
	LDR	R2,=0x2000	; load 0x2000 into R2
	STR	R0,[R2]	; store content of R0 at
			; the address given by R2

Instruction Set

Instruction Types

- **Data transfer**
  - Copy content of one register to another register
  - Load registers with data from memory
  - Store register contents into memory
- **Data processing**
  - Arithmetic operations → + - * / ...
  - Logic operations → AND, OR, ...
  - Shift / rotate operations
- **Control flow**
  - Branches
  - Function calls
- **Miscellaneous (various)**

Type	Frequency	
Data transfer	43%	
Control flow	23%	
Arithmetic	15%	Data processing
Compare	13%	
Logical	5%	
Miscellaneous	1%	

Instruction Set

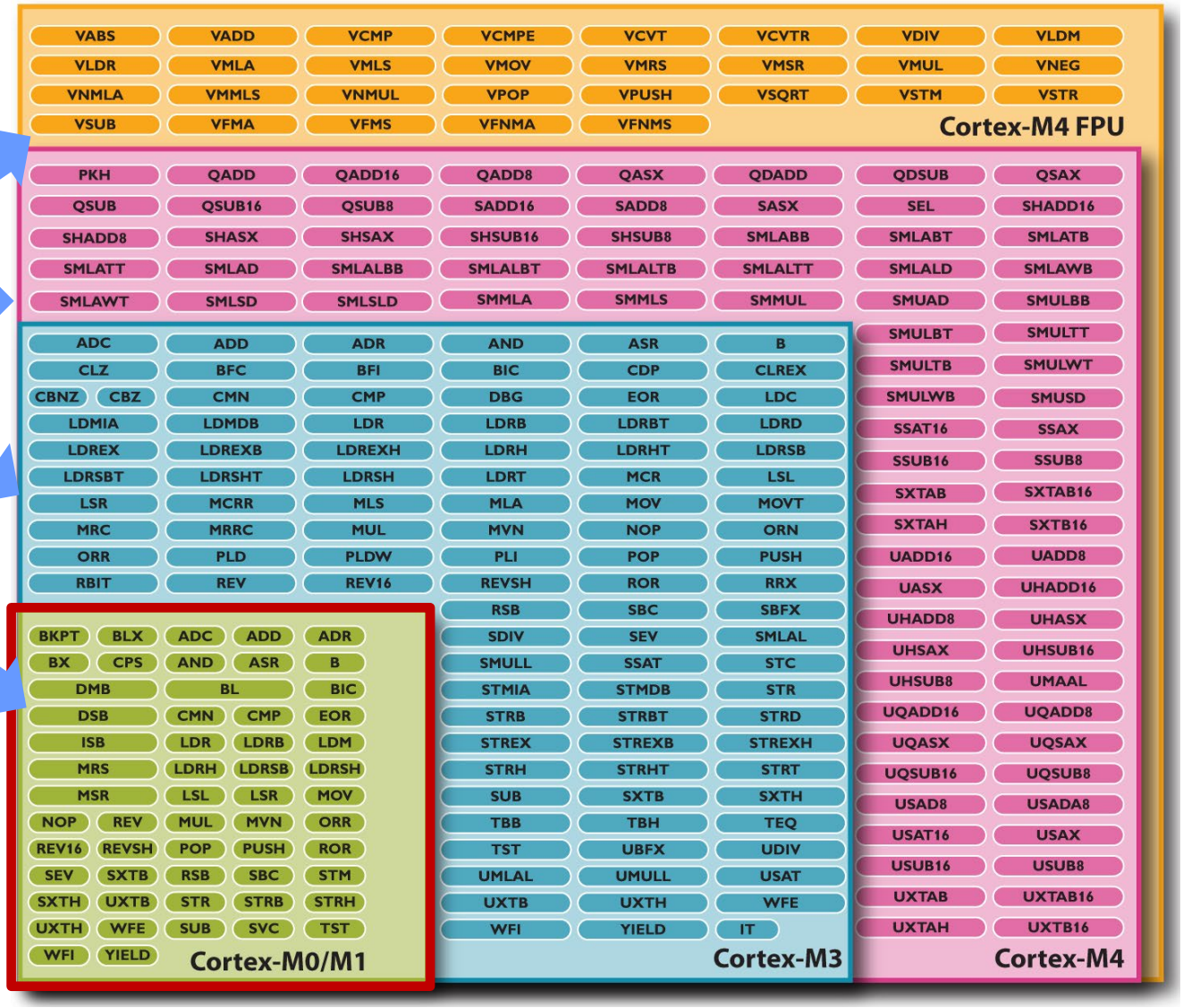
School of Engineering
InES Institute of Embedded Systems

- **Instructions Cortex-M**

ARMv7-M Architecture

ARMv6-M Architecture

CT1
Instruction Set Cortex-M0
=
Subset Cortex-M3/4



source: ARM

Instruction Set

School of Engineering
InES Institute of Embedded Systems

- **Overview Cortex-M0**

Instruction Type	Instructions
Move	MOV, MOVS
Load/Store	LDR, LDRB, LDRH, LDRSB, LDRSH, LDM, STR, STRB, STRH, STM
Add, Subtract, Multiply	ADD, ADDS, ADCS, ADR, SUB, SUBS, SBCS, RSBS, MULS
Compare	CMP, CMN
Logical	ANDS, EORS, ORRS, BICS, MVNS, TST
Shift and Rotate	LSLS, LSRS, ASRS, RORS
Extend	SXTH, SXTB, UXTH, UXTB
Reverse	REV, REV16, REVSH
Branch	B, BL, B{cond}, BX, BLX
Stack	POP, PUSH
Processor State	BKPT, CPS, MRS, MSR, SVC
No Operation	NOP
Hint / Synchronization	DMB, DSB, ISB, SEV, WFE, WFI, YIELD

Instruction Set

School of Engineering
InES Institute of Embedded Systems

Cortex-M0: 16-bit Thumb instruction encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	0	0	opcode			imm5				Rm			Rd			shift by immediate, move register
0	0	0	1	1	0	opc		Rm		Rn			Rd			add/subtract register
0	0	0	1	1	1	opc		imm3		Rn			Rd			add/subtract immediate
0	0	1	opcode			Rdn			imm8							add/sub./comp./move immediate
0	1	0	0	0	0	opcode				Rm			Rd			data-processing register
0	1	0	0	0	1	opcode		DN	Rm				Rd			special data processing
0	1	0	0	0	1	1	1	L	Rm				0	0	0	branch/exchange
0	1	0	0	1	Rt			PC-relative imm8								load from literal pool
0	1	0	1	opcode			Rm			Rn			Rt			load/store register offset
0	1	1	B	L	imm5					Rn			Rt			load/store word/byte imm. offset
1	0	0	0	L	imm5					Rn			Rt			load/store halfword imm. offset
1	0	0	1	L	Rt			SP-relative imm8								load from or store to stack
1	0	1	0	SP	Rd			imm8								add to SP or PC
1	0	1	1	x	x	x	x	x	x	x	x	x	x	x	x	miscellaneous
1	1	0	0	L	Rn			register list								load/store multiple
1	1	0	1	cond				imm8								conditional branch
1	1	0	1	1	1	1	0	x	x	x	x	x	x	x	x	undefined instruction
1	1	0	1	1	1	1	1	imm8								service (system) call
1	1	1	0	0	imm11											unconditional branch
1	1	1	0	1	x	x	x	x	x	x	x	x	x	x	x	32-bit instruction
1	1	1	1	x	x	x	x	x	x	x	x	x	x	x	x	32-bit instruction

Note: There are some inconsistencies in ARMs use of the notation Rd (destination) and Rt (target). The slides use the ARMv6-M Architecture Reference Manual (ARM DDI 0419C (ID092410) as a reference.

Instruction Set

School of Engineering
InES Institute of Embedded Systems

Cortex-M0 Example Program

- Assembler converts each Assembly instruction to 16-bit opcode
- Memory addresses in steps of 2 (Zweierschritte)
  - Reason: opcodes are 16-bit (two bytes) long, e.g. **`0x20A5`**

Memory address *Opcode* *Listfile*

```
00000000 20A5      demoprg MOVS   R0,#0xA5    ; copy 0xA5 into R0
00000002 2111              MOVS   R1,#0x11    ; copy 0x11 into R1
00000004 1840              ADDS   R0,R0,R1    ; add contents of R0 and R1
                                              ; store result in R0
00000006 4A00              LDR    R2,=0x2000  ; load address into R2
00000008 6010              STR    R0,[R2]     ; store content of R0 at
                                              ; the address given by R2
1) 0000000A 00002000
```

Binary ← *generated by Assembler (tool)* — ***Source code***

1) Assembler places constant for the LDR instruction at the end of the code

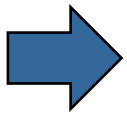
Instruction Set

School of Engineering
InES Institute of Embedded Systems

Load program into memory

- Example assumes code area in memory at address `0x0800'0000` [1)]

```
00000000 20A5     demoprg MOVS   R0,#0xA5
00000002 2111             MOVS   R1,#0x11
00000004 1840             ADDS   R0,R0,R1

00000006 4A00             LDR    R2,=0x2000
00000008 6010             STR    R0,[R2]

0000000A 00002000
```



Memory

address	8 Bit
0x0800'0000	0xA5
0x0800'0001	0x20
0x0800'0002	0x11
0x0800'0003	0x21
0x0800'0004	0x40
0x0800'0005	0x18
0x0800'0006	0x00
0x0800'0007	0x4A
...	
...	

`0x0000'0000` — `linker adds offset of 0x0800'0000 before loading`

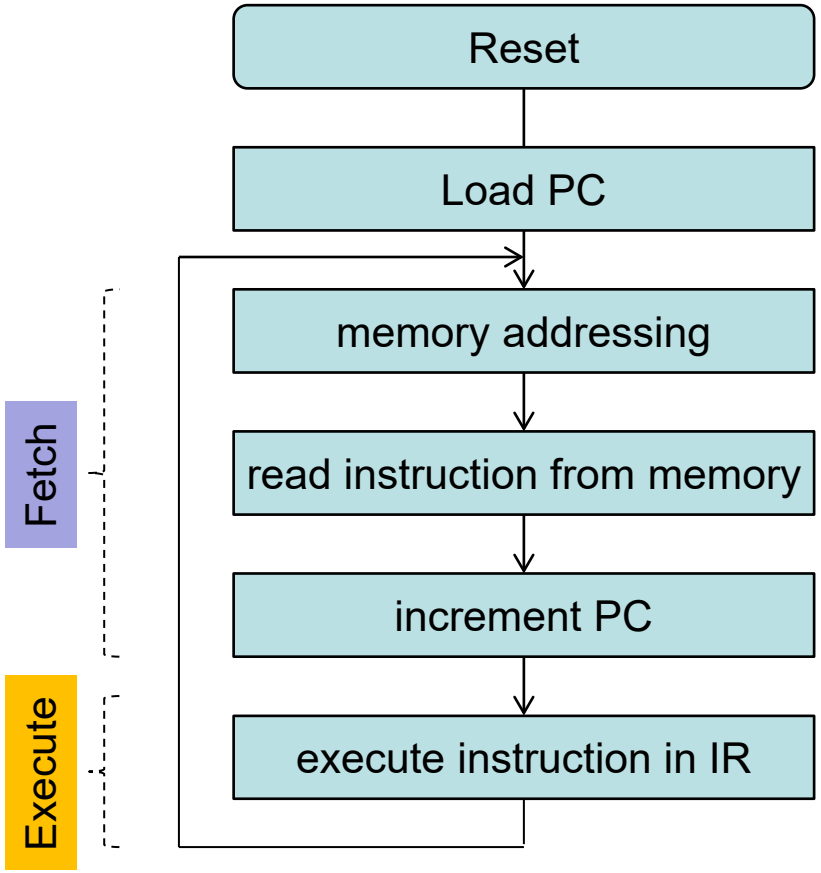
1) address of code area depends on individual system

Program Execution

School of Engineering
InES Institute of Embedded Systems

Precondition
- Loader has copied executable code into memory
- Loader has stored code start address in memory location 0x0000'0004 (ARM convention)

Sequence



Read start address from 0x0000'0004 and load value into Program Counter (PC)

Place content of PC on address bus

Copy content of selected memory address to Instruction Register (IR)

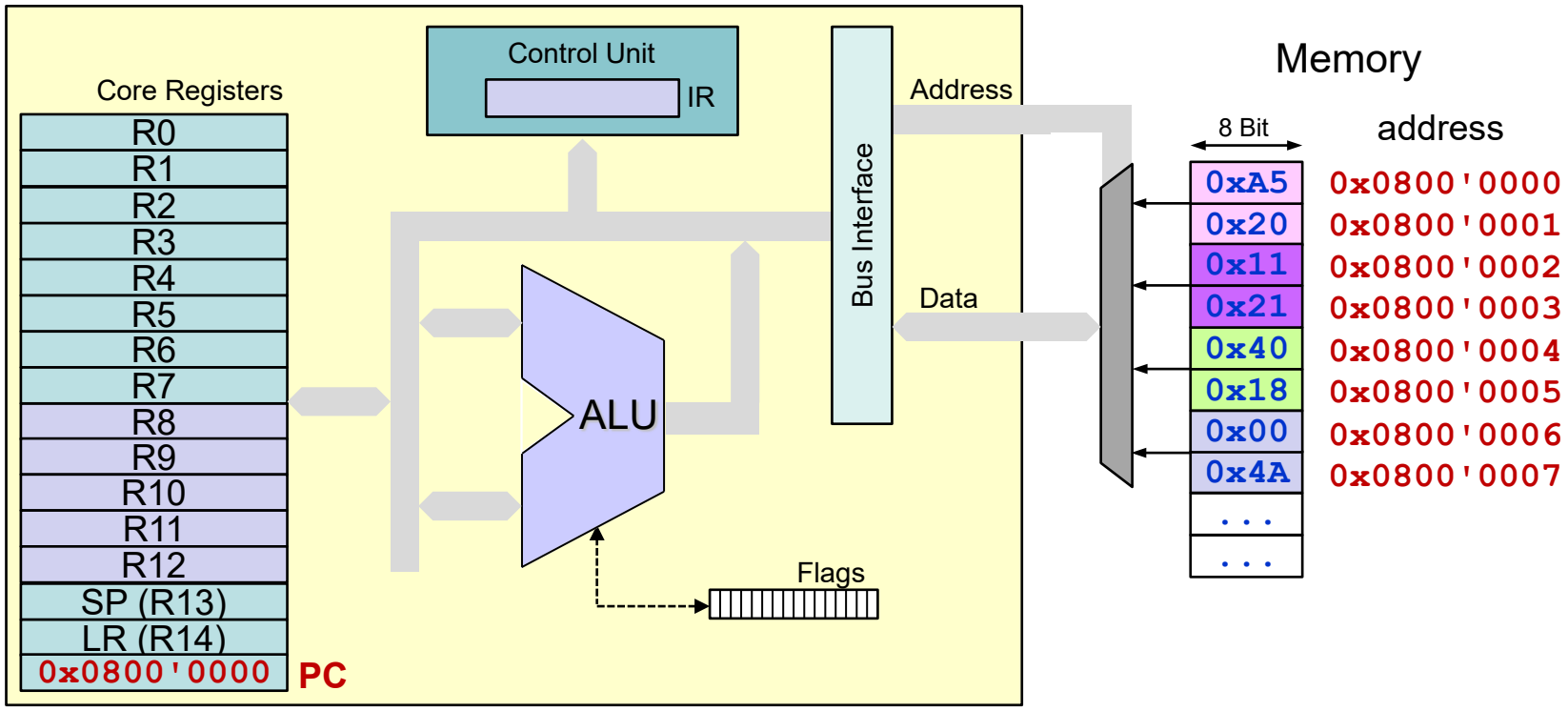
PC = PC + 2 [1)]

Execute instruction in instruction register (IR)

1) '2' is used here for simplicity. In fact the PC is incremented by the length of the executed instruction.

Program Execution

- **"Reset": Read 0x0000'0004 → PC**



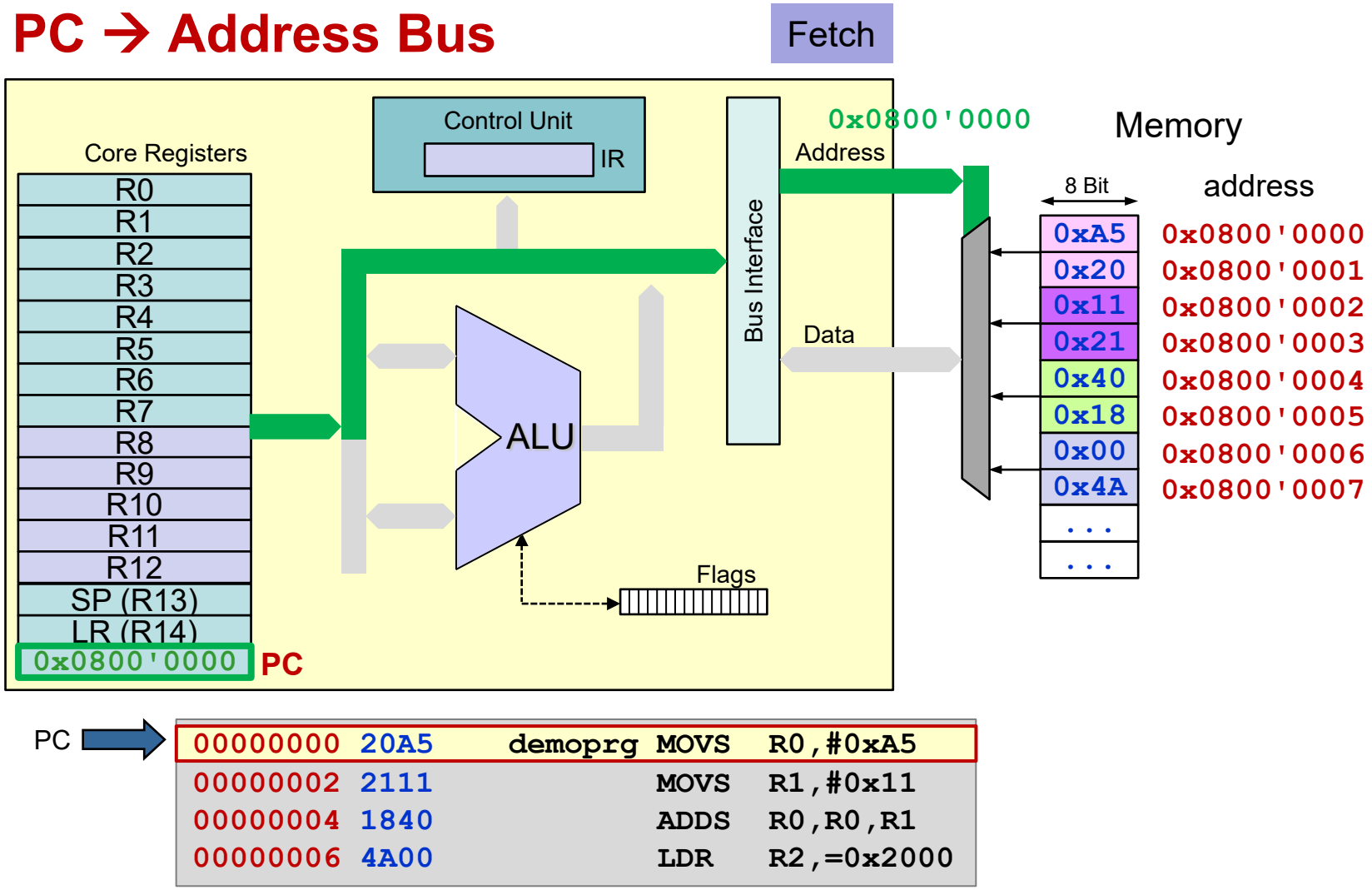
PC

```
00000000 20A5    demoprg MOVS   R0,#0xA5
00000002 2111            MOVS   R1,#0x11
00000004 1840            ADDS   R0,R0,R1
00000006 4A00            LDR    R2,=0x2000
```

Program Execution

School of Engineering
InES Institute of Embedded Systems

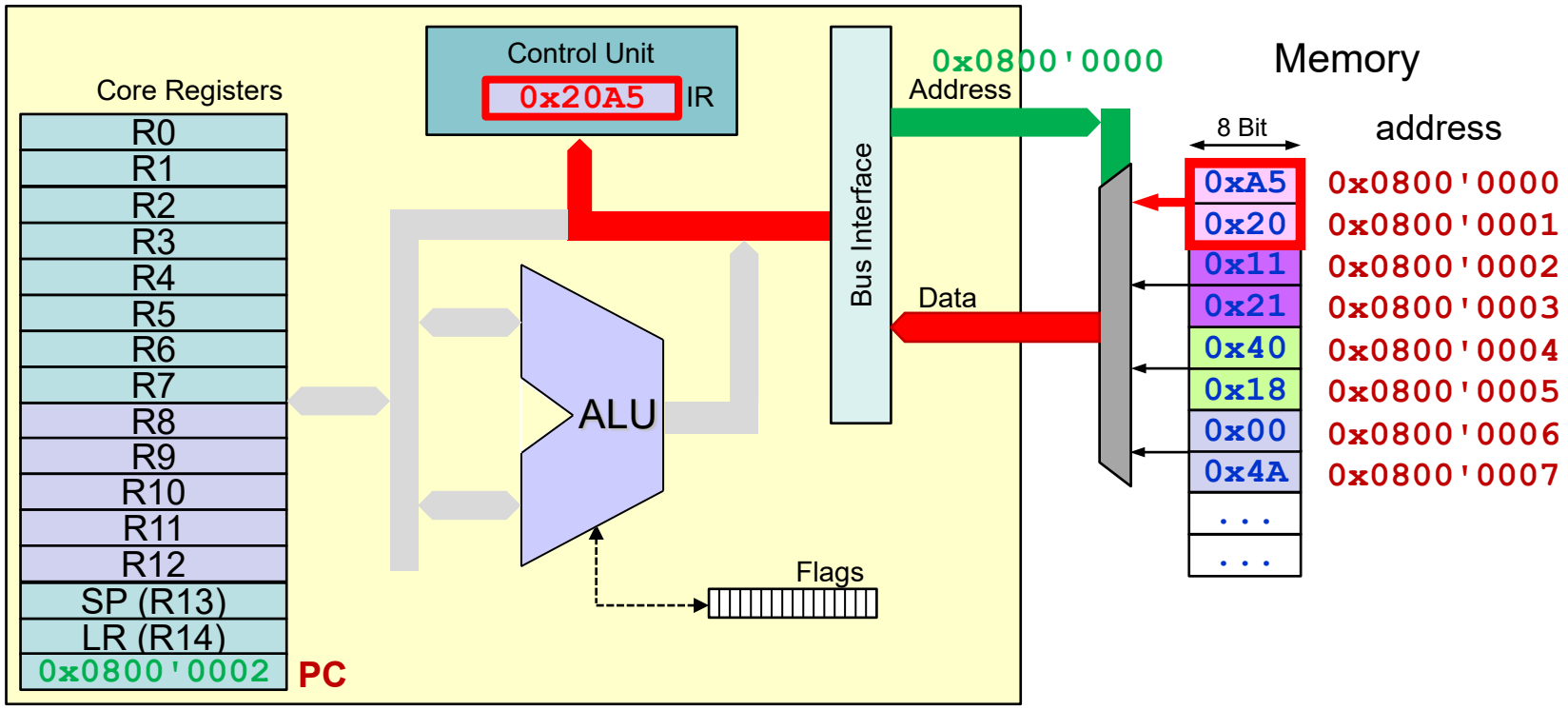
- **PC → Address Bus**



Program Execution

Read Instruction / Increment PC

Fetch

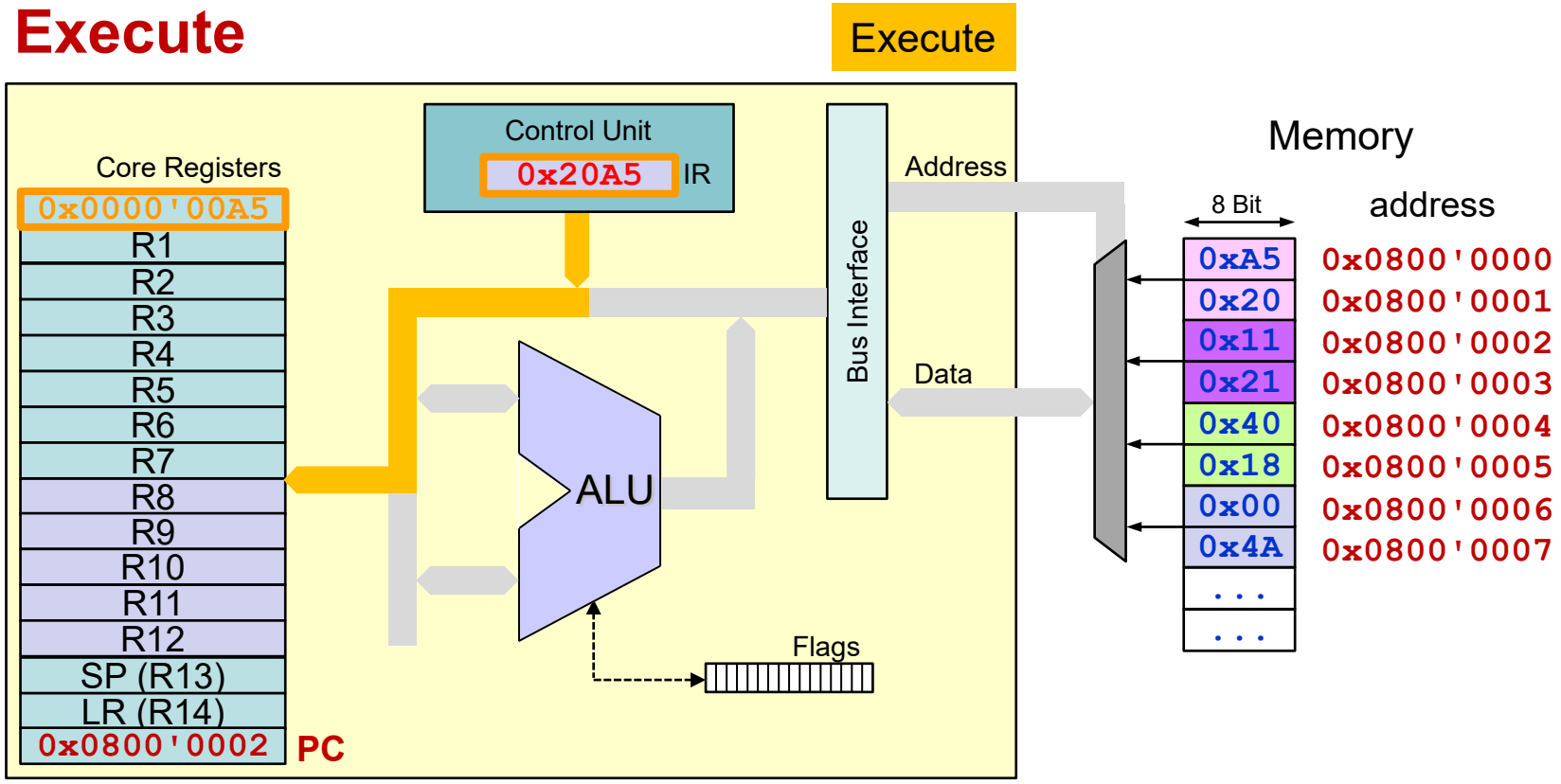


PC

```
00000000 20A5     demoprg MOVS   R0,#0xA5
00000002 2111             MOVS   R1,#0x11
00000004 1840             ADDS   R0,R0,R1
00000006 4A00             LDR    R2,=0x2000
```

Program Execution

- **Execute**

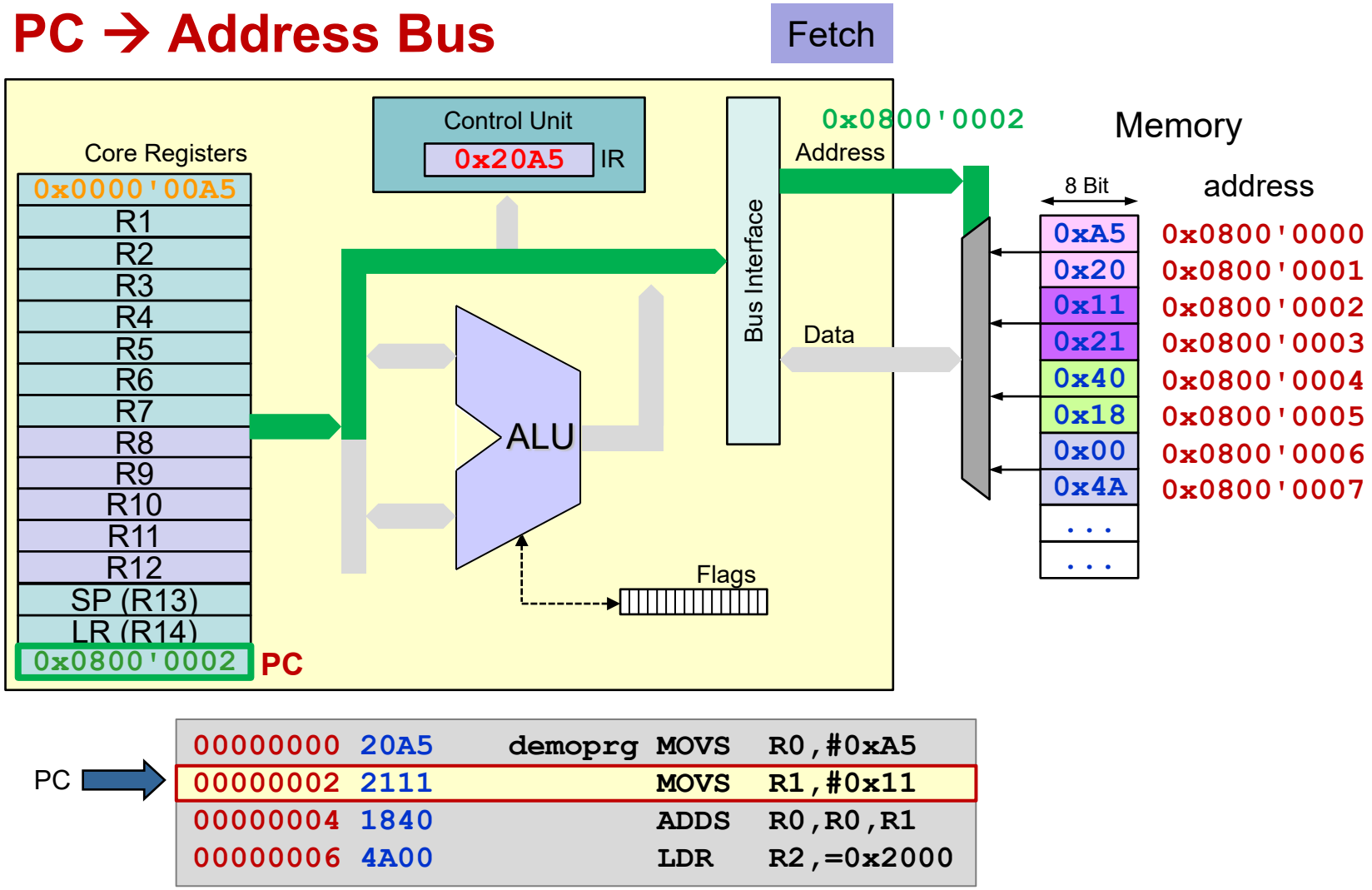


PC

```
00000000 20A5     demoprg MOVS   R0,#0xA5
00000002 2111             MOVS   R1,#0x11
00000004 1840             ADDS   R0,R0,R1
00000006 4A00             LDR    R2,=0x2000
```

Program Execution

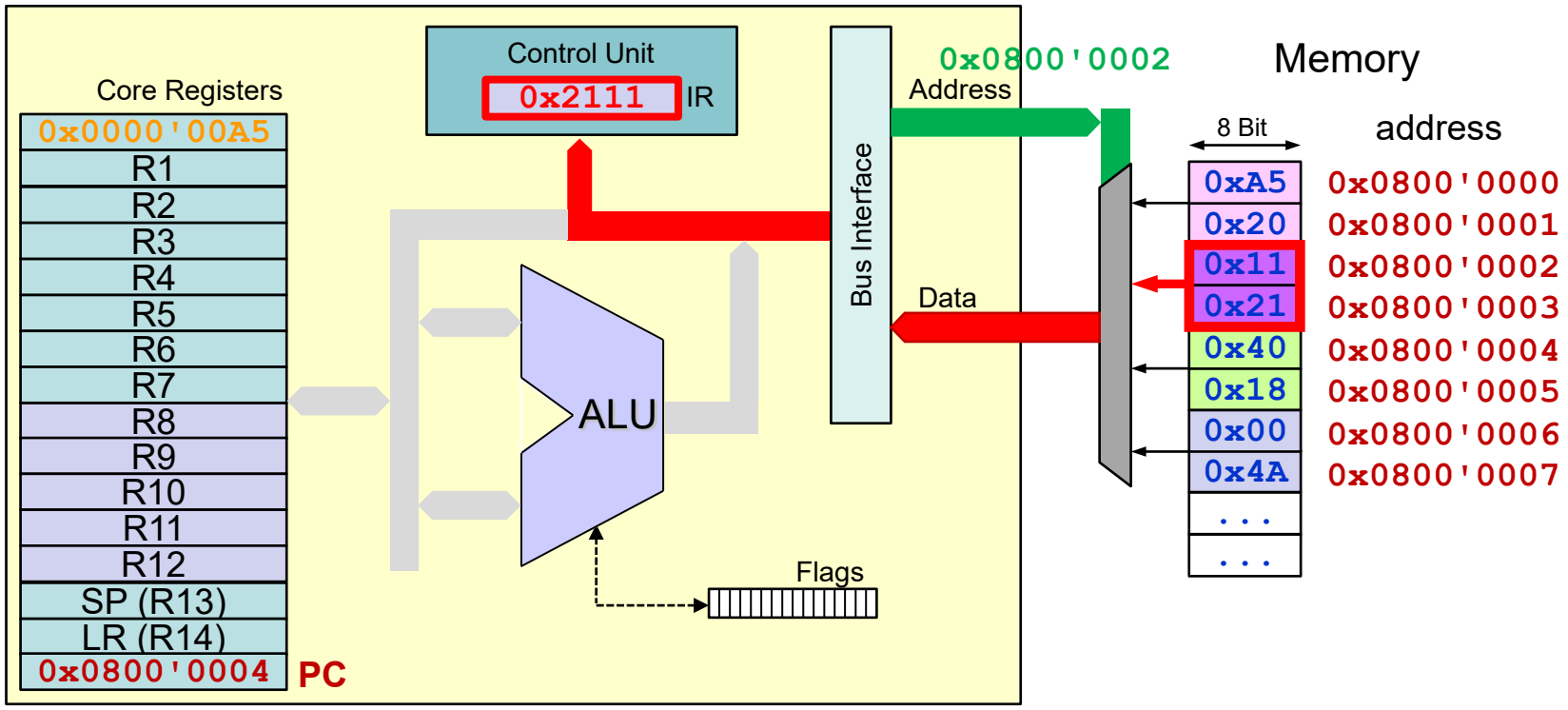
- **PC → Address Bus**



Program Execution

Read Instruction / Increment PC

Fetch

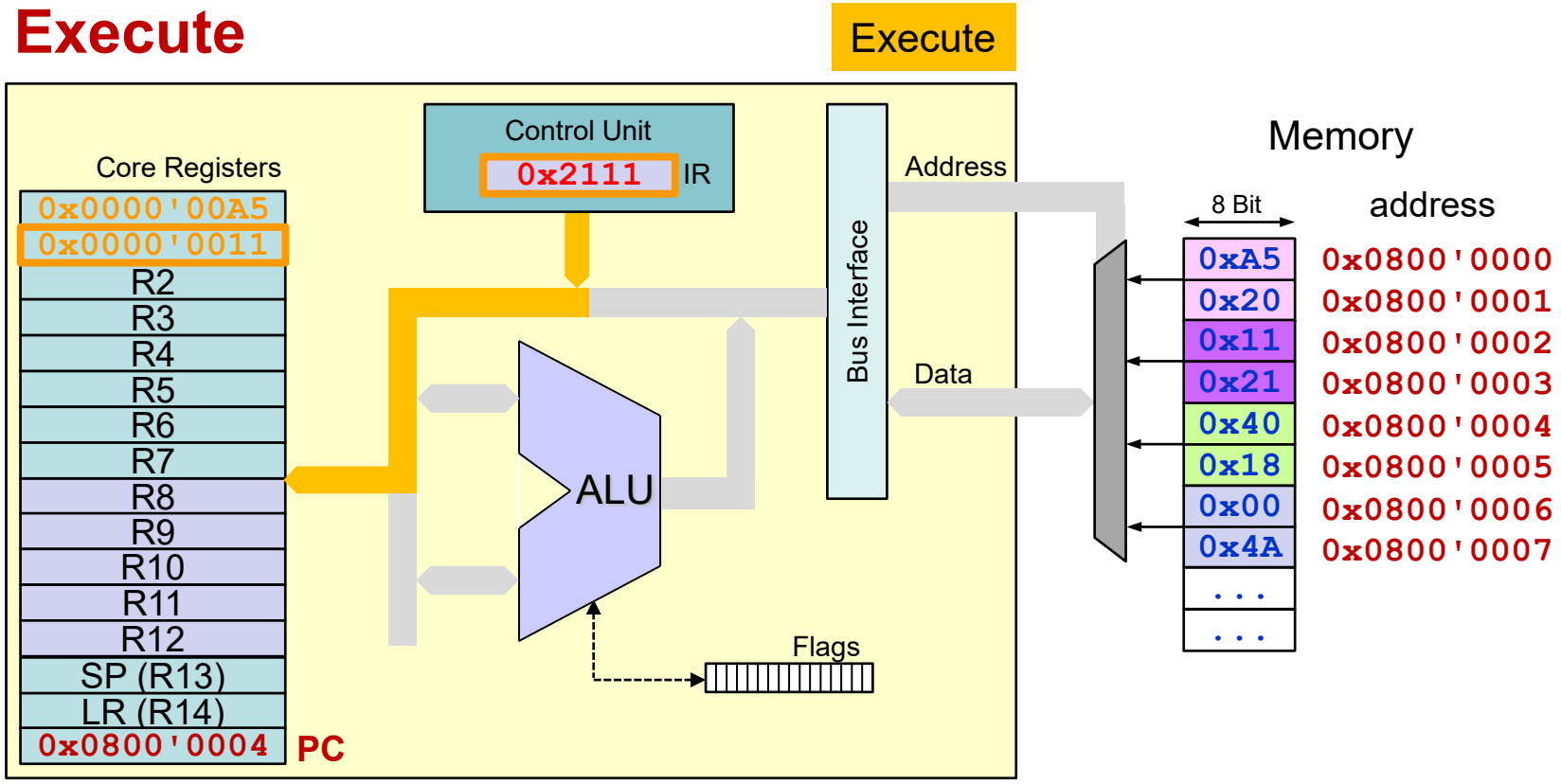



```
00000000 20A5     demoprg MOVS   R0,#0xA5
00000002 2111             MOVS   R1,#0x11
00000004 1840             ADDS   R0,R0,R1
00000006 4A00             LDR    R2,=0x2000
```

PC

Program Execution

- **Execute**

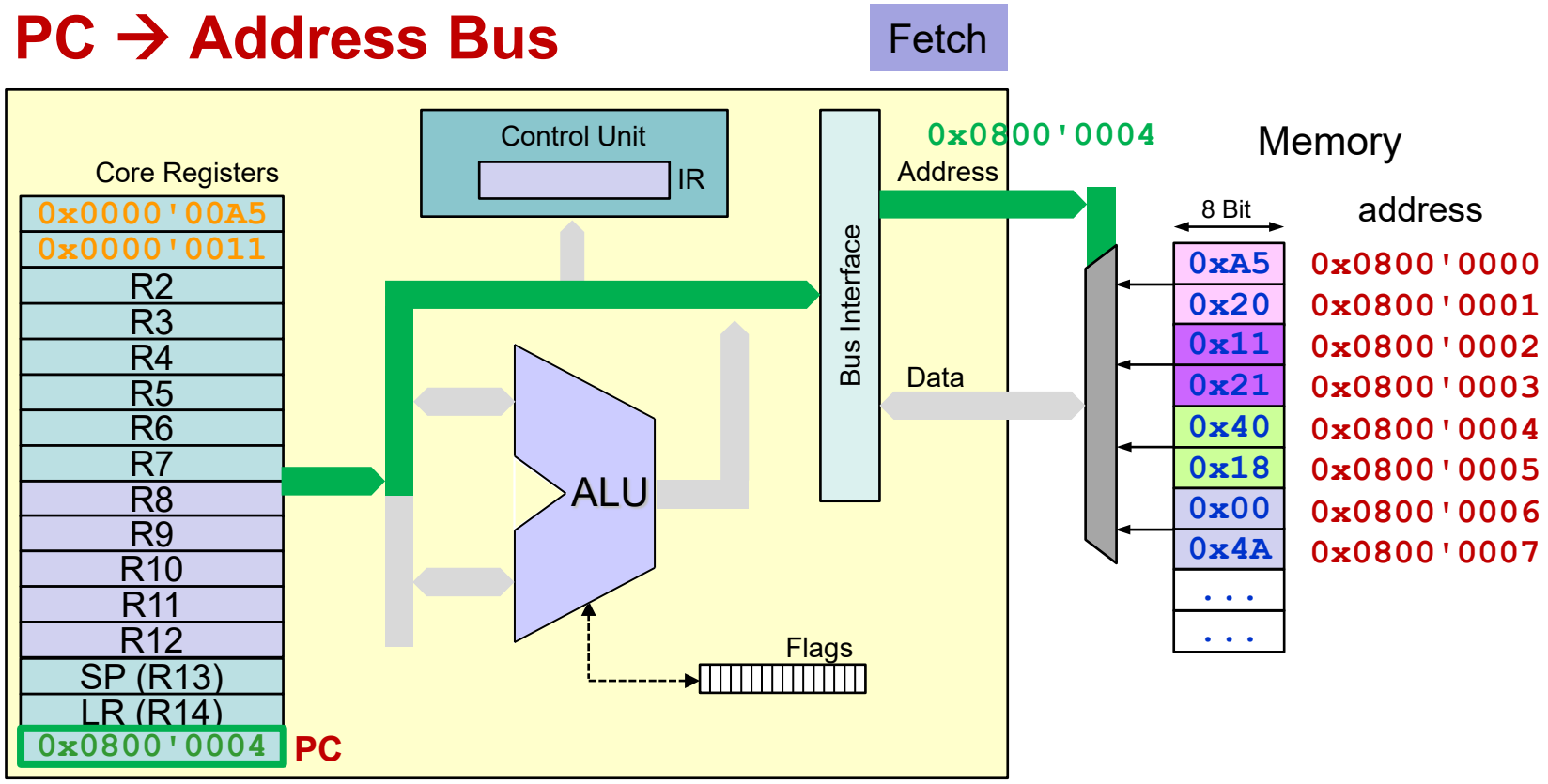



```
00000000 20A5     demoprg MOVS   R0,#0xA5
00000002 2111             MOVS   R1,#0x11
00000004 1840             ADDS   R0,R0,R1
00000006 4A00             LDR    R2,=0x2000
```

PC

Program Execution

- **PC → Address Bus**




```
00000000 20A5    demoprg MOVS   R0,#0xA5
00000002 2111            MOVS   R1,#0x11
00000004 1840            ADDS   R0,R0,R1
00000006 4A00            LDR    R2,=0x2000
```

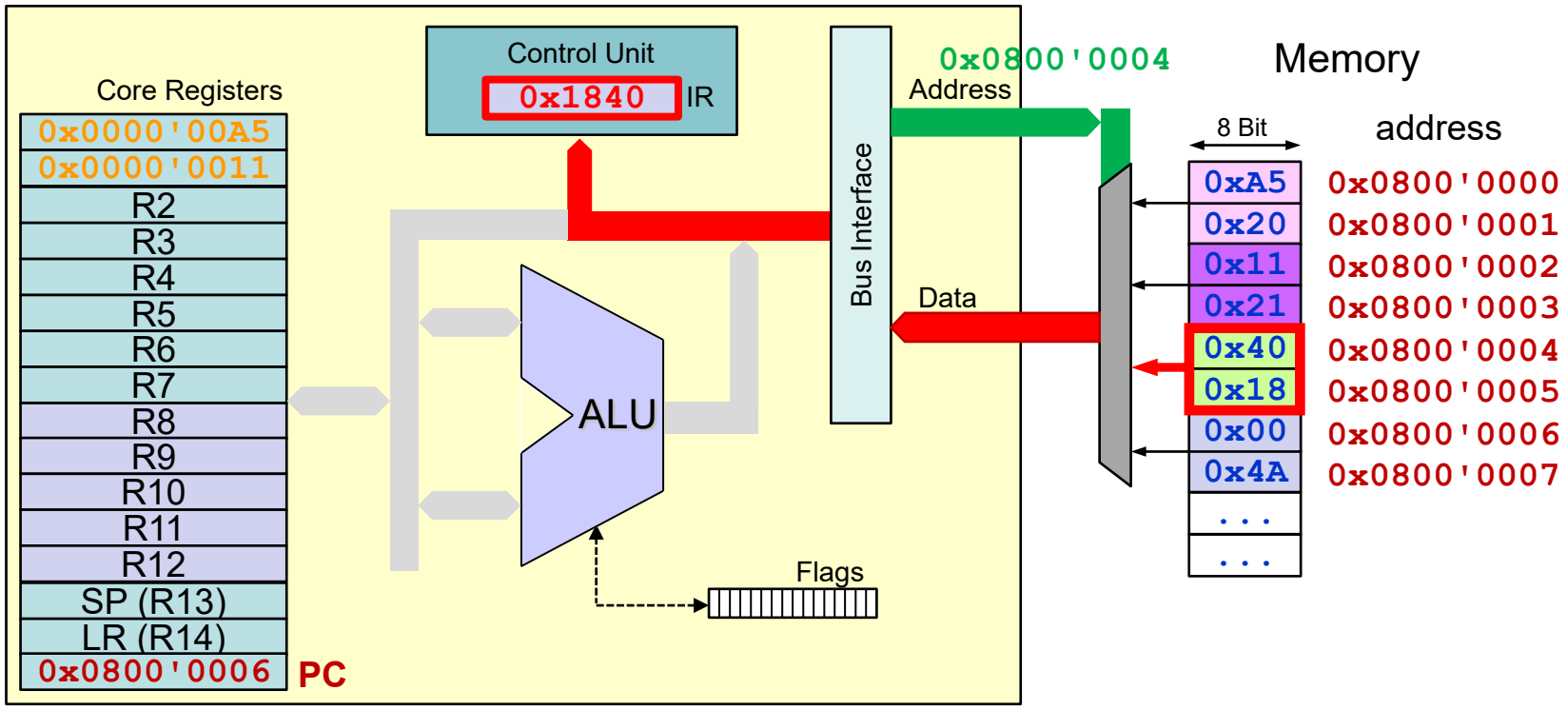
PC

Program Execution

School of Engineering
InES Institute of Embedded Systems

Read Instruction / Increment PC

Fetch

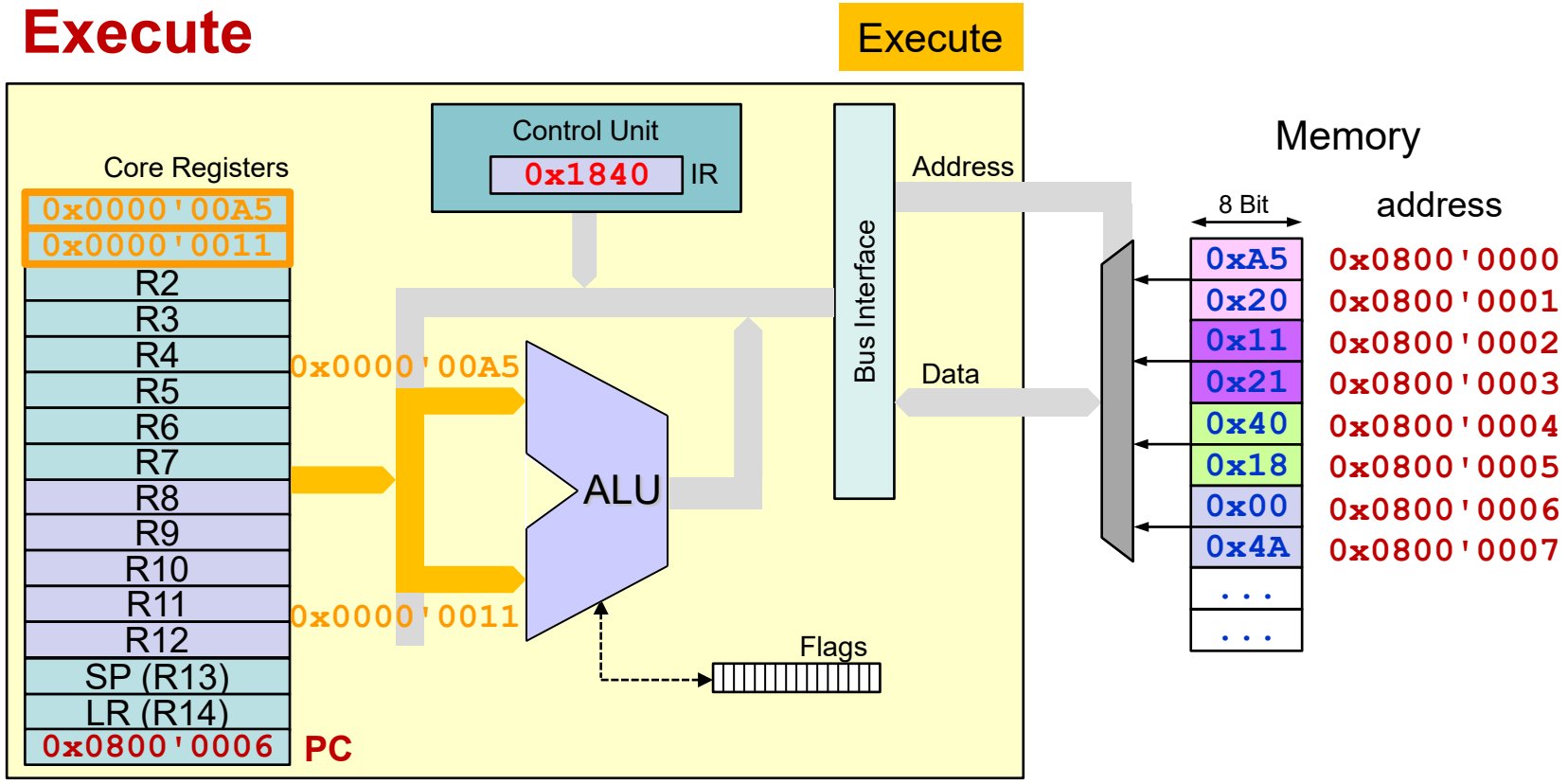



```
00000000 20A5     demoprg MOVS   R0,#0xA5
00000002 2111             MOVS   R1,#0x11
00000004 1840             ADDS   R0,R0,R1
00000006 4A00             LDR    R2,=0x2000
```

PC

Program Execution

- **Execute**

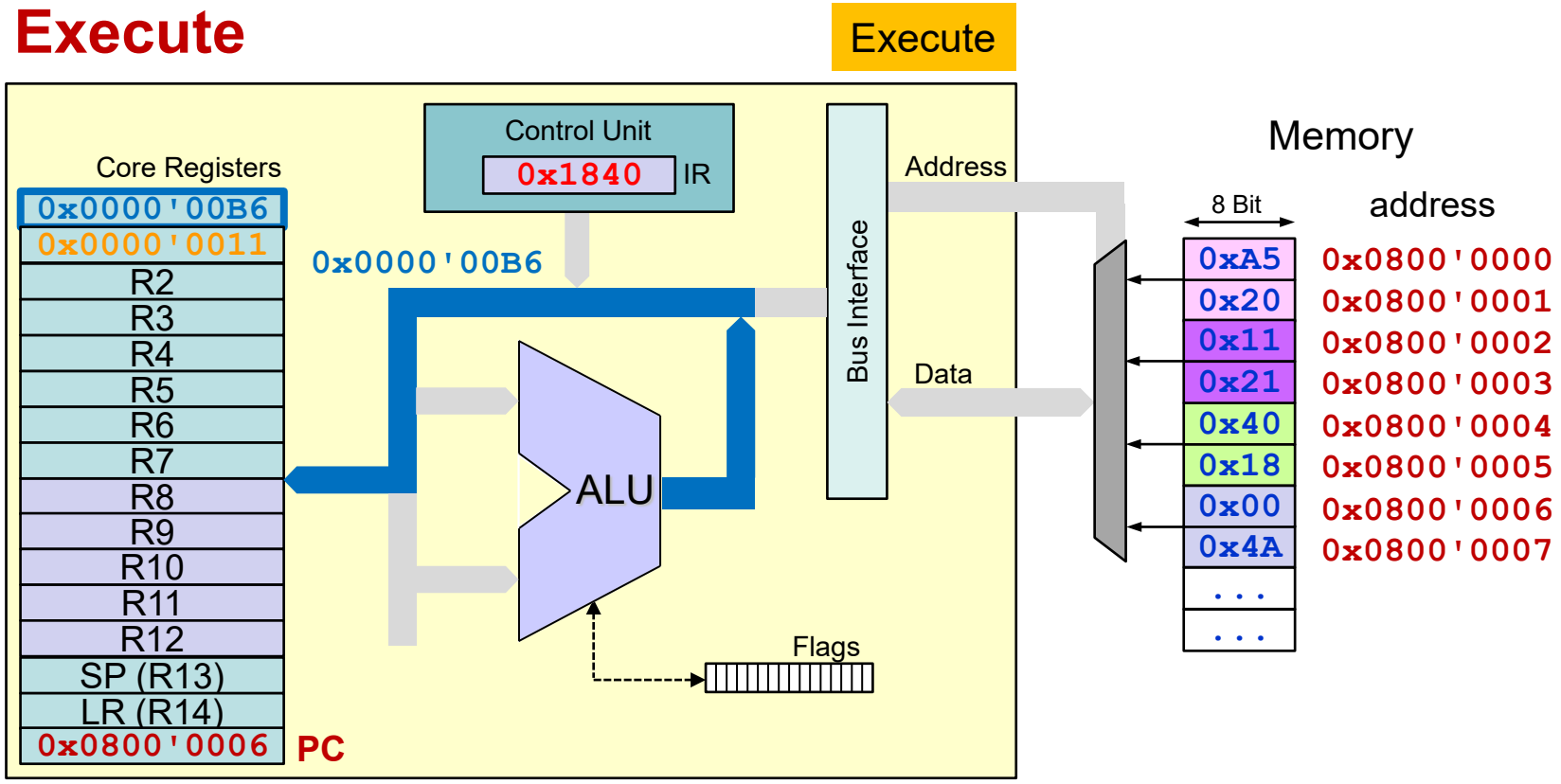



```
00000000 20A5     demoprg MOVS   R0,#0xA5
00000002 2111             MOVS   R1,#0x11
00000004 1840             ADDS   R0,R0,R1
00000006 4A00             LDR    R2,=0x2000
```

PC

Program Execution

- **Execute**



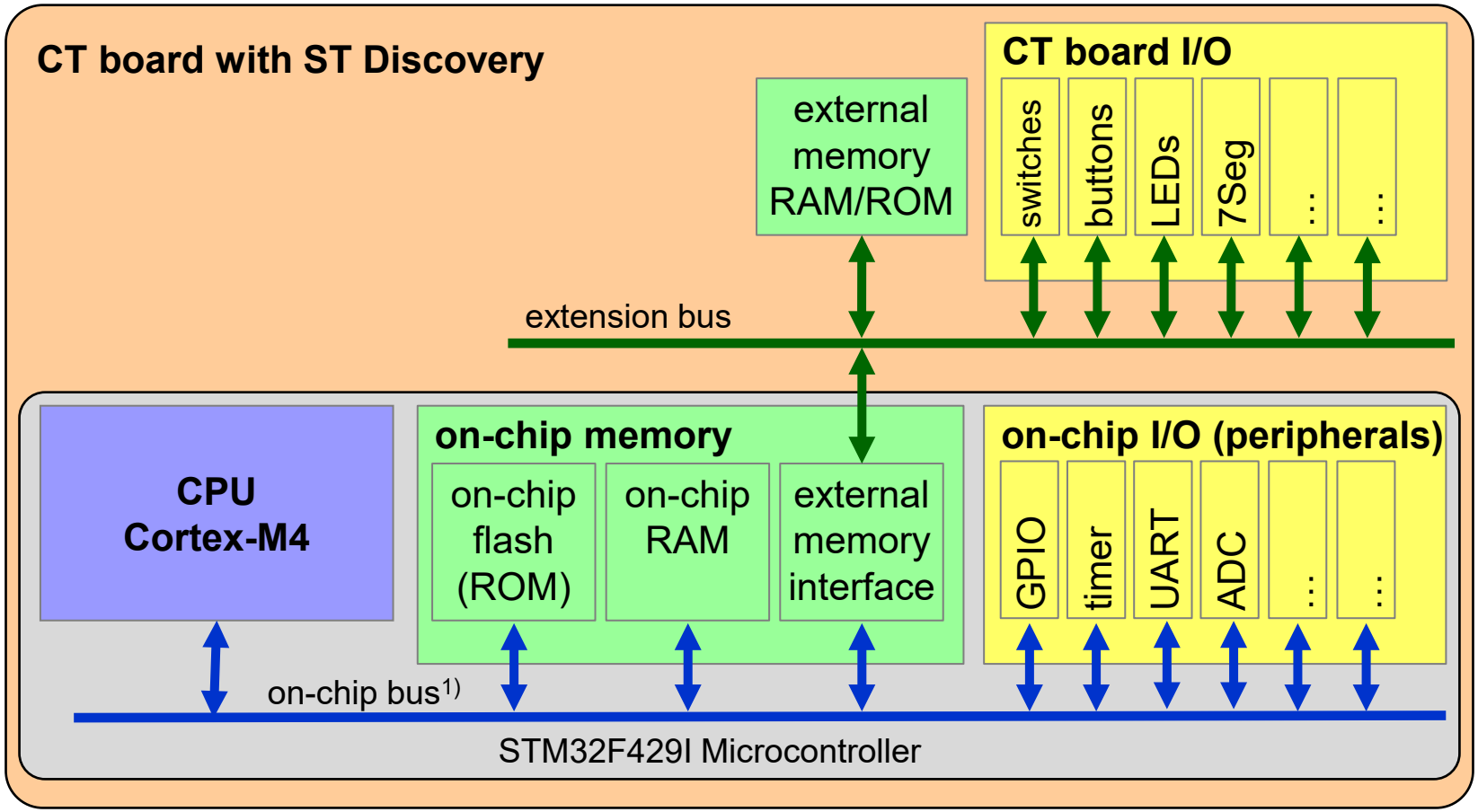

```
00000000 20A5    demoprg MOVS   R0,#0xA5
00000002 2111            MOVS   R1,#0x11
00000004 1840            ADDS   R0,R0,R1
00000006 4A00            LDR    R2,=0x2000
```

PC

Memory Map

School of Engineering
InES Institute of Embedded Systems

Hardware View

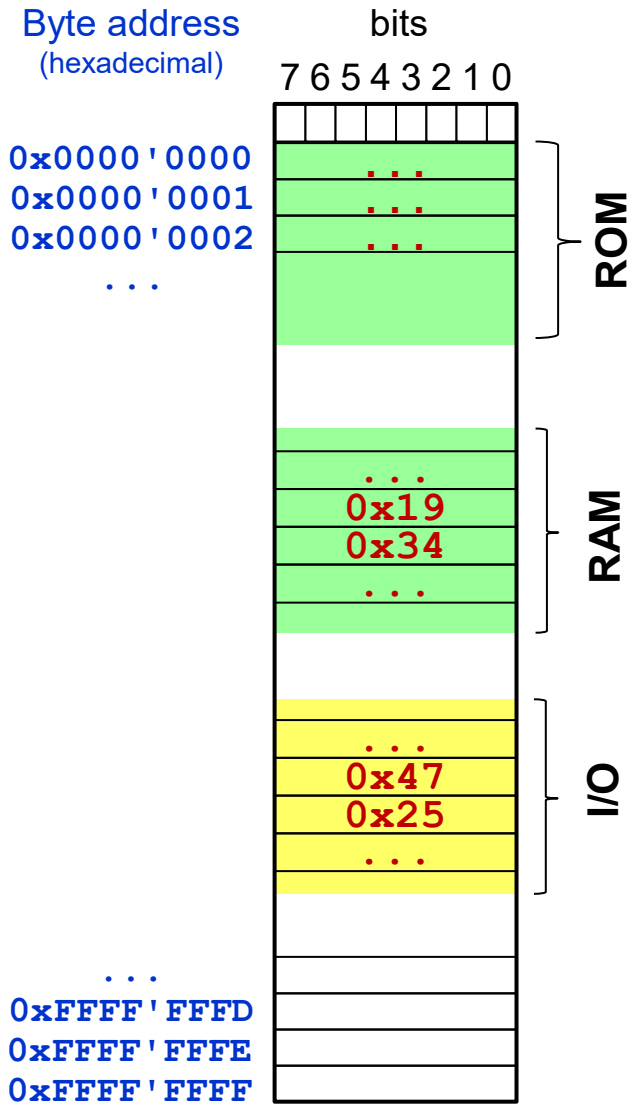


1) The implementation partitions the on-chip bus into several busses called AHBx and APBx

Memory Map

School of Engineering
InES Institute of Embedded Systems

- **Memory Layout**
  - System Address Map
  - Graphical layout of main memory
  - Linear array of bytes
  - What is located where (at which address) in memory?
    - Location of RAM (readable and writable)
    - Location of ROM (only readable)
    - Location of I/O registers



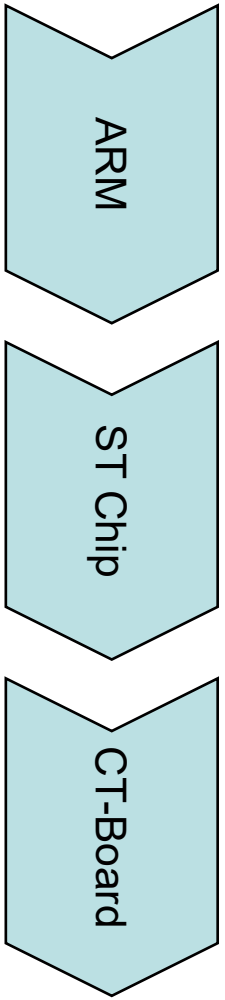
Memory maps in CT1/CT2 will be drawn with lowest address at the top and highest address at the bottom. This simplifies work with assembly listing and tables.

Caution: ARM and ST documentation are the other way round. Lowest address at the bottom and highest address at top.

Memory Map

School of Engineering
InES Institute of Embedded Systems

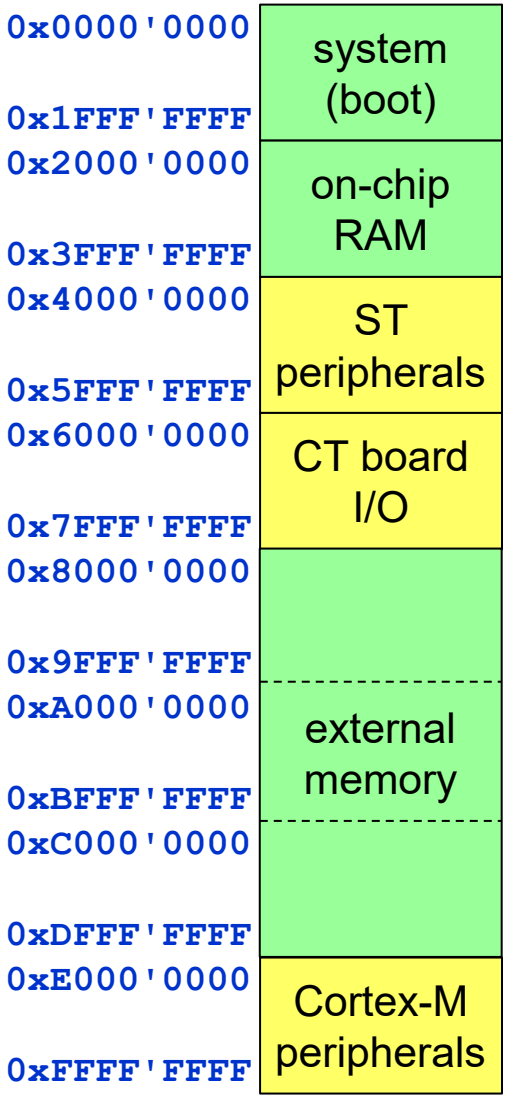
- **Address Allocation**
  - ARM policies
    - Cortex-M specific
    - guide lines for chip manufacturer
  - ST design decisions
    - chip specific
    - number and size of on-chip RAMs
    - size of flash
    - control register for peripherals
  - CT board design decisions
    - board specific
    - LEDs, switches, etc



Memory Map

School of Engineering
InES Institute of Embedded Systems

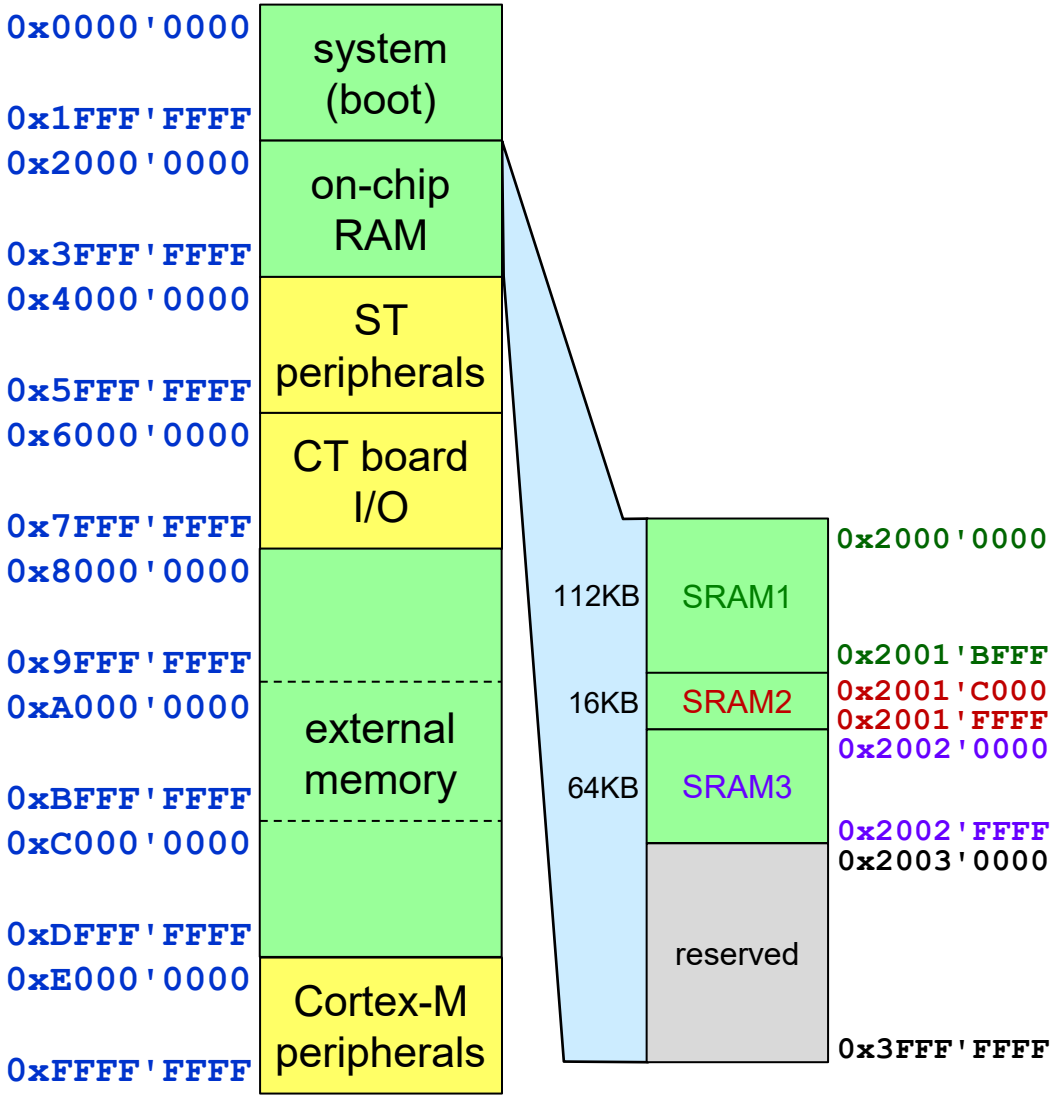
- **CT-Board**
  - Address space
    4 GByte = 2^{32} Bytes
  - From **0x0000'0000**
    to **0xFFFF'FFFF**
  - Partitioned into
    8 blocks of
    512-MByte each



Memory Map

School of Engineering
InES Institute of Embedded Systems

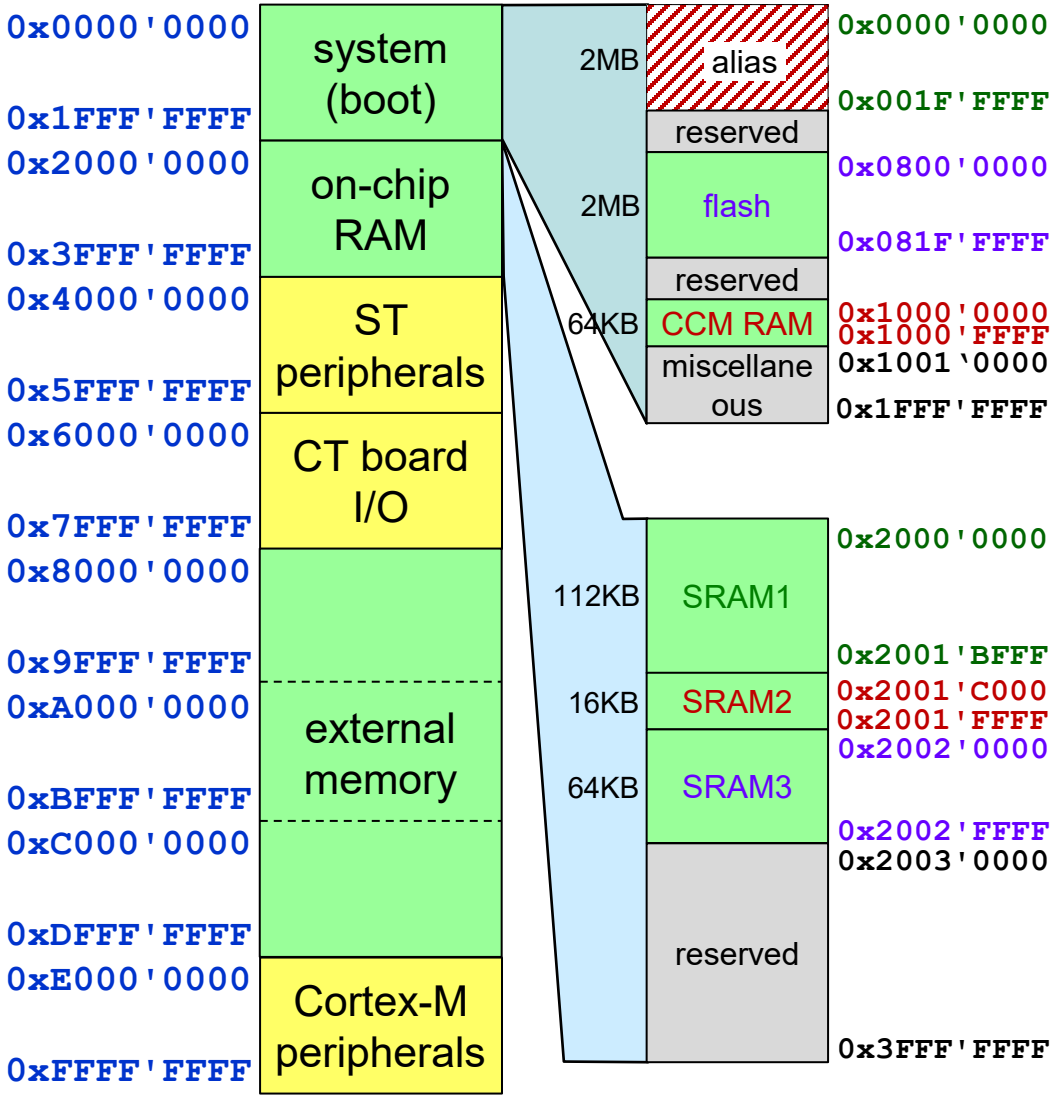
- **ST chip specific**
  - SRAM1
    - 112 KByte
  - SRAM2
    - 16 KByte
  - SRAM3
    - 64 KByte



Memory Map

School of Engineering
InES Institute of Embedded Systems

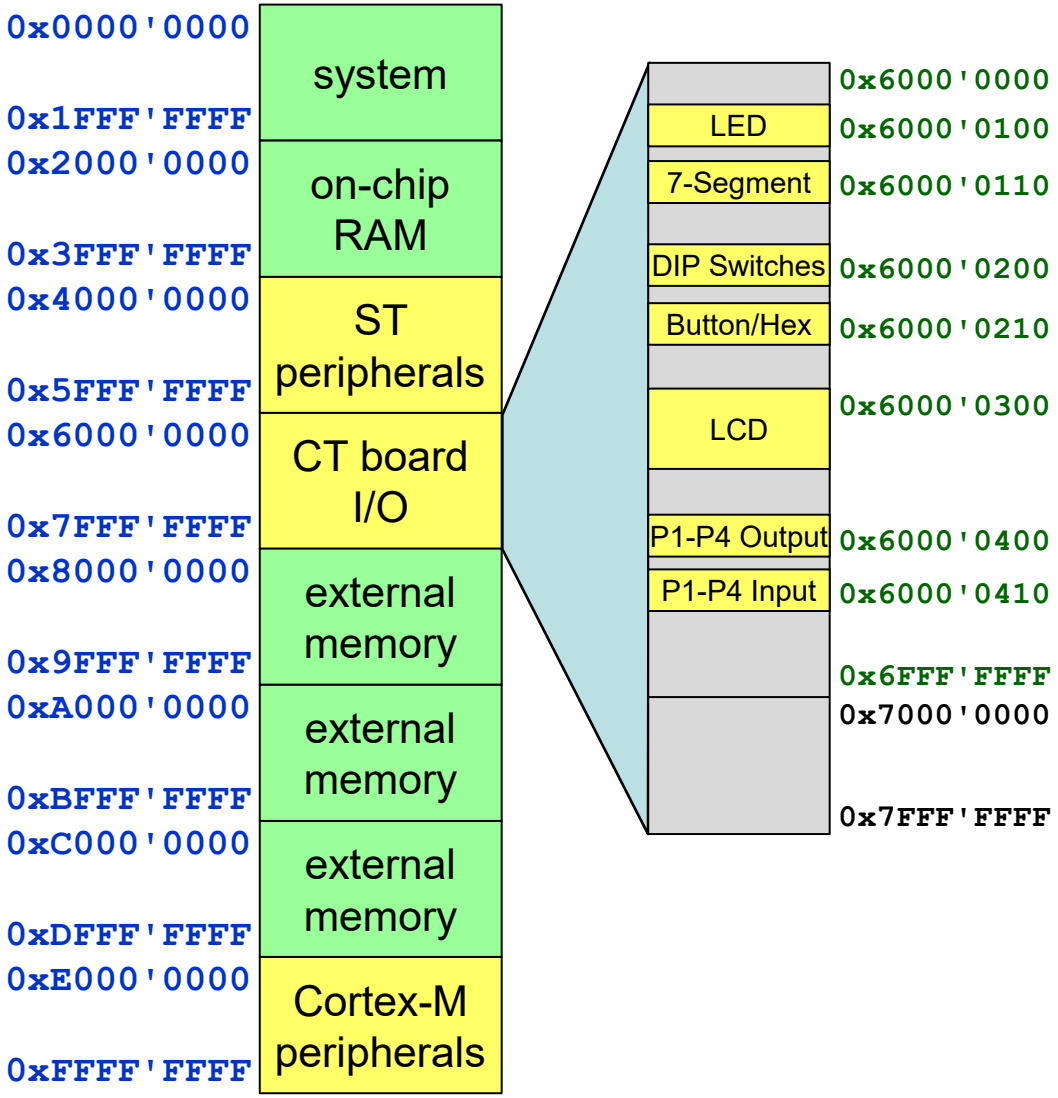
- **ST chip specific**
  - Flash
    - non-volatile memory
  - CCM RAM
    - core coupled memory
    - very fast RAM
  - Alias
    - user configurable mirror
    - physical memory can appear at two locations



Memory Map

School of Engineering
InES Institute of Embedded Systems

- **CT-Board I/O**

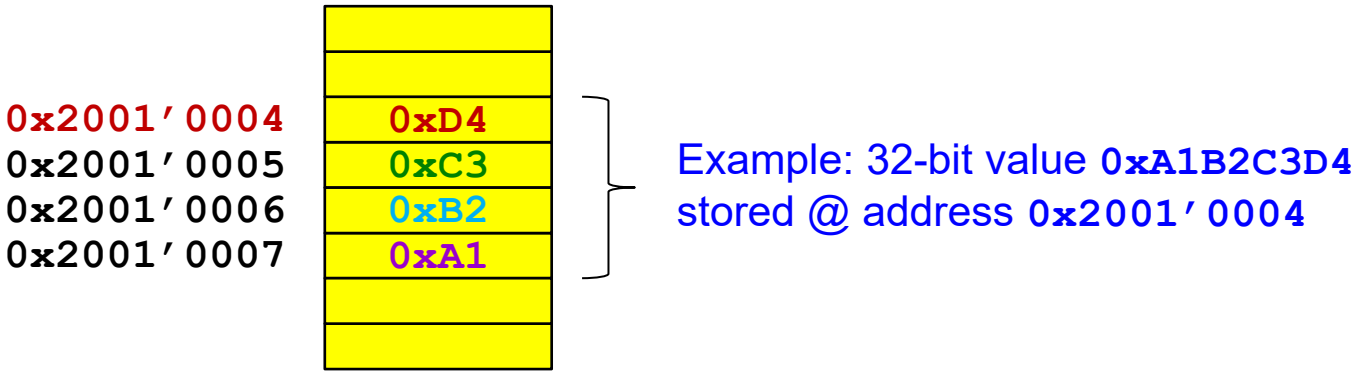


Integer Types

School of Engineering
InES Institute of Embedded Systems

Multi-byte Integer Types

- Usually memory is organized in bytes
  - one address per byte → reasons: space and history
- An integer type often requires several bytes
  - e.g. 4 byte addresses are required to store a 32-bit integer



Integer Types

School of Engineering
InES Institute of Embedded Systems

- **Integer Types in C**
  - Sizes of integer types are platform dependent

Sizes in byte

8051		Cortex-Mx: Keil (ARM)		x86-64 (i7): gcc	
char	1	char	1	char	1
short	2	short	2	short	2
int	2	int	4	int	4
long int	4	long int	4	long int	8
		long long int	8	long long int	8
char *	2	void *	4	void *	8

Integer Types

School of Engineering
InES Institute of Embedded Systems

- **ARM Cortex-M**

C99 / specified width

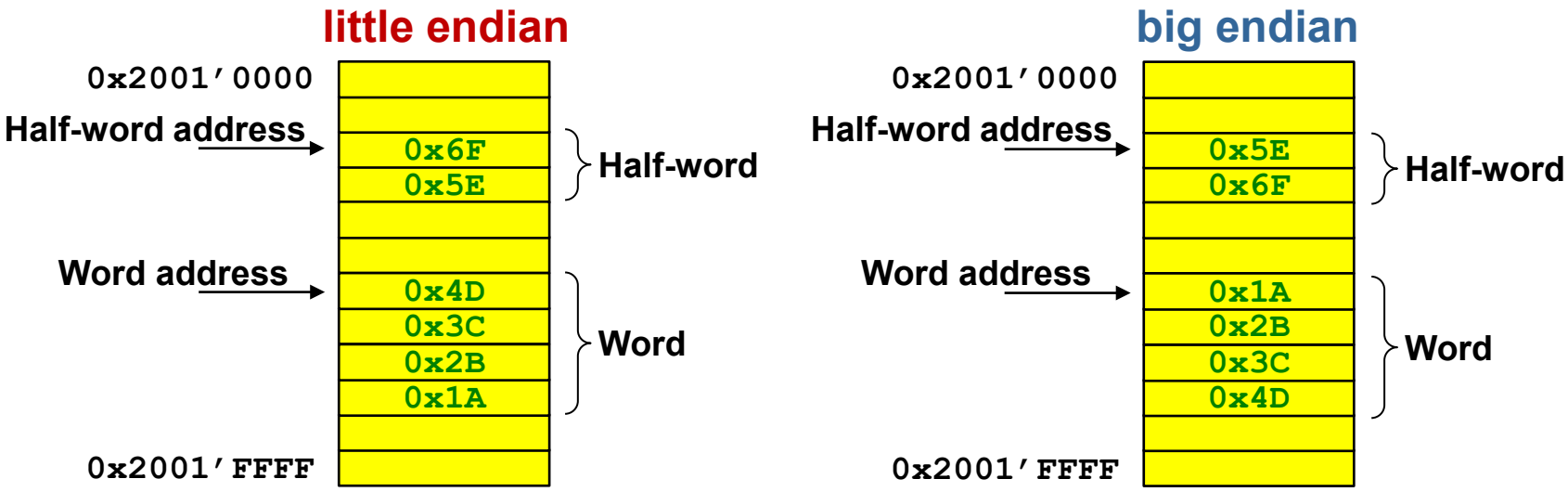
C-Type – unsigned integers	Size	Term	inttypes.h / stdint.h
`unsigned char`	8 Bit	Byte	`uint8_t`
`unsigned short`	16 Bit	Half-word	`uint16_t`
`unsigned int`	32 Bit	Word	`uint32_t`
`unsigned long`	32 Bit	Word	`uint32_t`
`unsigned long long`	64 Bit	Double-word	`uint64_t`

C-Type – signed integers	Size	Term	inttypes.h / stdint.h
`signed char`	8 Bit	Byte	`int8_t`
`short`	16 Bit	Half-word	`int16_t`
`int`	32 Bit	Word	`int32_t`
`long`	32 Bit	Word	`int32_t`
`long long`	64 Bit	Double-word	`int64_t`

Integer Types

- **How are groups of bytes arranged in memory?**
  - little endian → *least significant byte* at lower address
    e.g. Intel x86, Altera Nios, ST ARM (STM32)
  - big endian → *most significant byte* at lower address
    e.g. Freescale (Motorola), PowerPC

Examples: **0x1A2B'3C4D** for Word and **0x5E6F** for Half-word

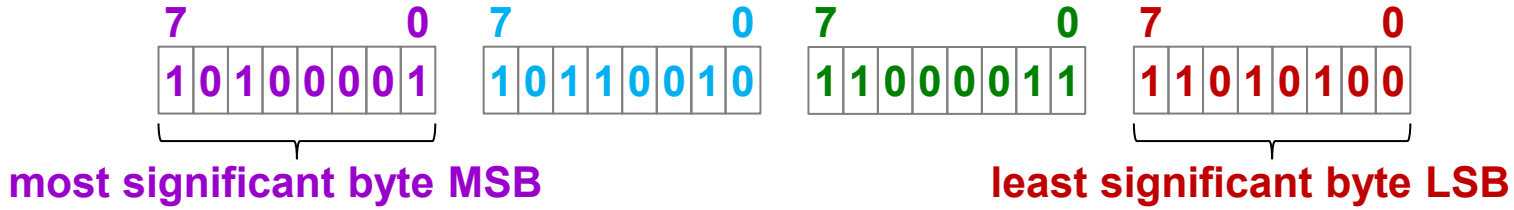


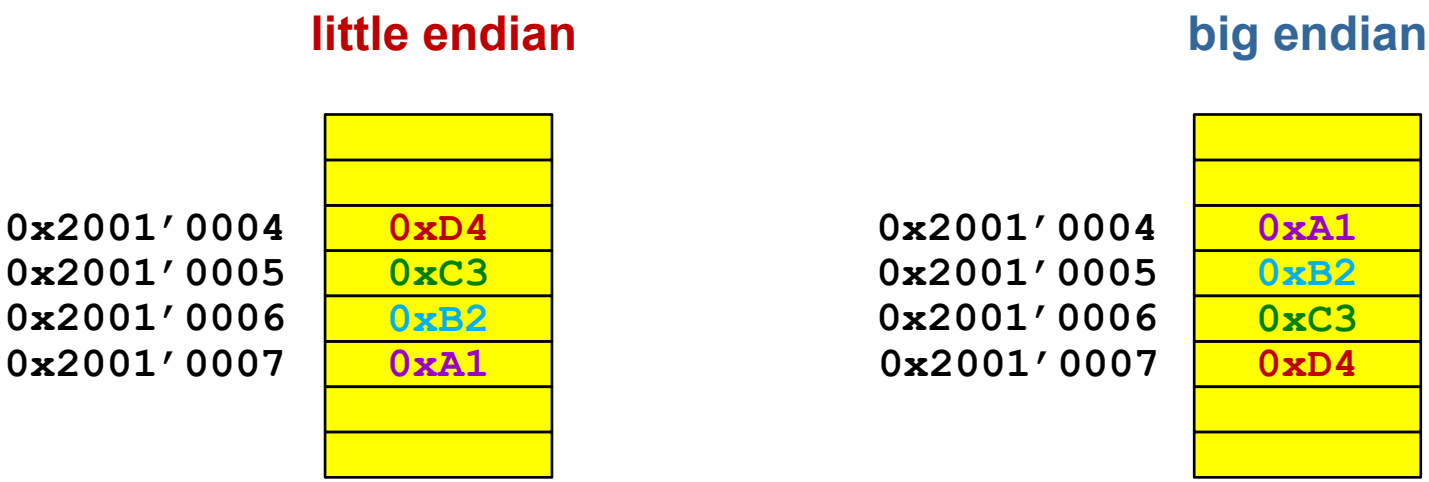
Cortex-M: Chip vendor defines endianness : STM32 (ST Microelectronics) is little endian.

Integer Types

School of Engineering
InES Institute of Embedded Systems

- **Example**
  - Store Word `0xA1B2'C3D4` at Address `0x2001'0004`

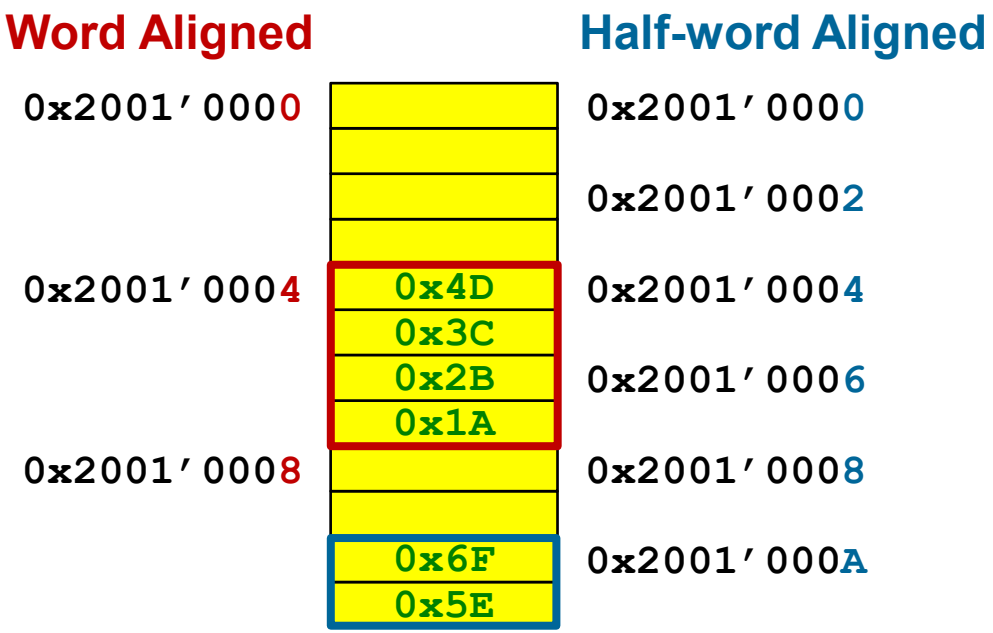




Integer Types

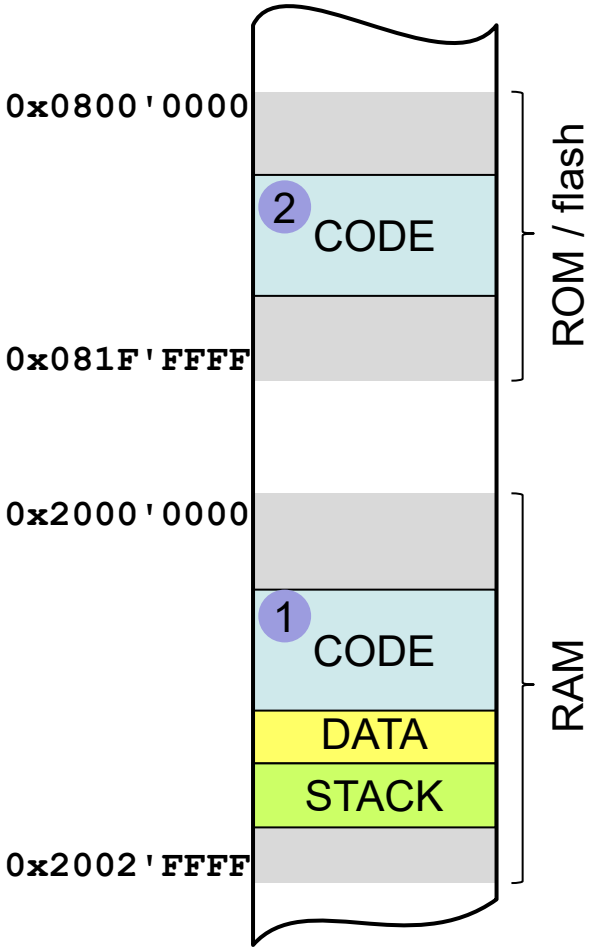
School of Engineering
InES Institute of Embedded Systems

- **Alignment**
  - Half-word aligned — Variables aligned on even addresses
  - Word aligned — Variables aligned on addresses that are divisible by four



Object File Sections

- **CODE** (1) (2)
  - Read-only → RAM or ROM
  - Instructions (opcodes)
  - Literals [1)]
- **DATA** [2)]
  - Read-write → RAM
  - Global variables
  - *static* variables in C
  - Heap in C → `malloc()`
- **STACK**
  - Read-write → RAM
  - Function calls / parameter passing
  - Local variables and local constants



[1)] Literal: a fixed/constant value in source code

[2)] The data section is sometimes further sub-divided into a read-only and a read-write part

Object File Sections

School of Engineering
InES Institute of Embedded Systems

Assembly Program Structure

- AREA directive

```
                AREA    MyCode, CODE, READONLY

                ENTRY
start           MOVS    R4,#12
                ADDS    R3,R4,#5
                B       start

                AREA    MyData, DATA, READWRITE

byte_var        DCB     0x1A,0x00
hw_var          DCW     0x2B3C
word_var        DCD     0x4D5E6F70

                AREA    STACK, NOINIT, READWRITE

stack_mem       SPACE   0x00000400
```

Define code area to include your program

Define data area to store global variables, etc.

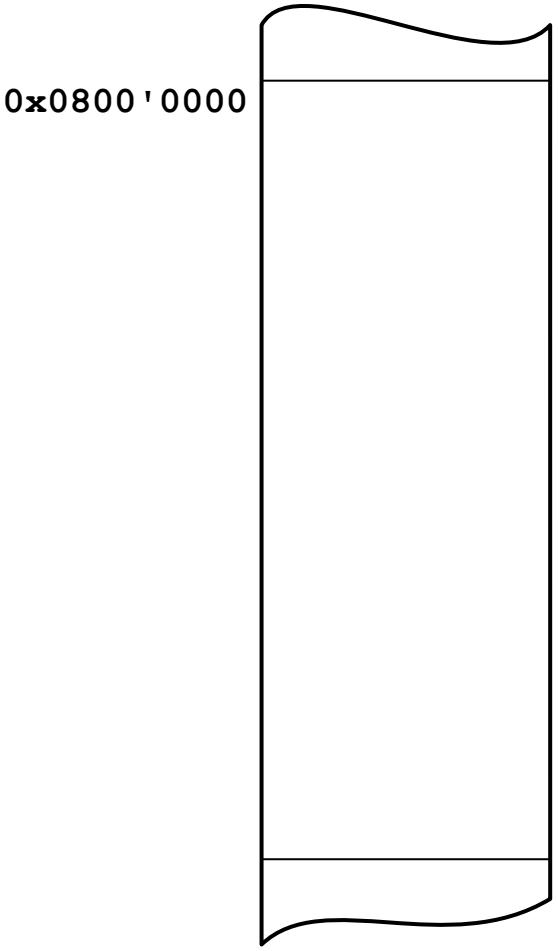
Define stack area to reserve space for stack

Memory Map / Object File Sections

School of Engineering
InES Institute of Embedded Systems

Assume

- **A program uses the following memory segments during execution**
  - Code `0x0800'1000` to `0x0800'17FF`
  - Data `0x2001'0000` to `0x2001'01FF`
  - Stack `0x2001'0200` to `0x2001'05FF`



- **Exercise**
  - Draw the memory map with the three sections
  - For each section mark the first and the last address with its value
  - How many memory cells (bytes) does each section contain?
  - For each section: In which STM32F4 memory is it located?

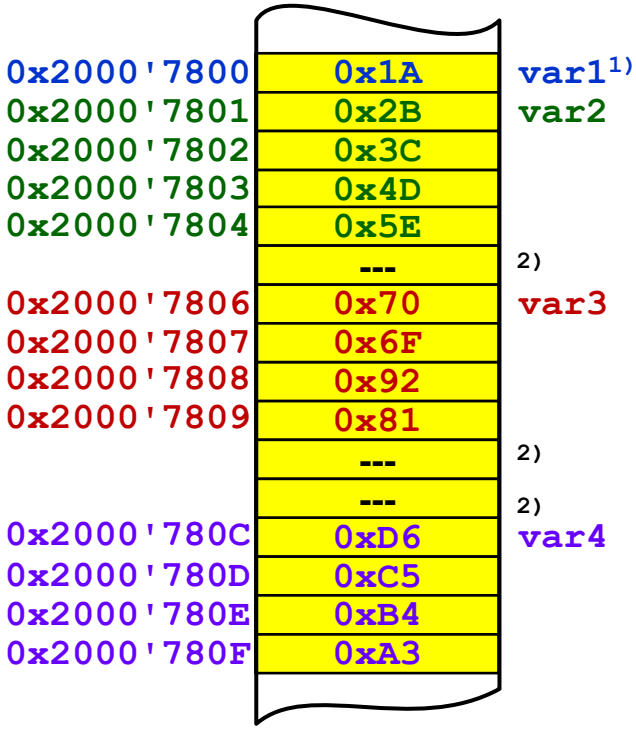
Object File Sections

School of Engineering
InES Institute of Embedded Systems

Memory Allocation in Assembly

- Directives for initialized data
  - **DCB** bytes
  - **DCW** half-words (half-word aligned)
  - **DCD** words (word aligned)
  - Can be located in **DATA** or **CODE** area

```
AREA example1, DATA, READWRITE
var1    DCB   0x1A
var2    DCB   0x2B, 0x3C, 0x4D, 0x5E
var3    DCW   0x6F70, 0x8192
var4    DCD   0xA3B4C5D6
```



1) if we assume that example1 starts at **0x2000'7800**

2) Padding bytes introduced for alignment

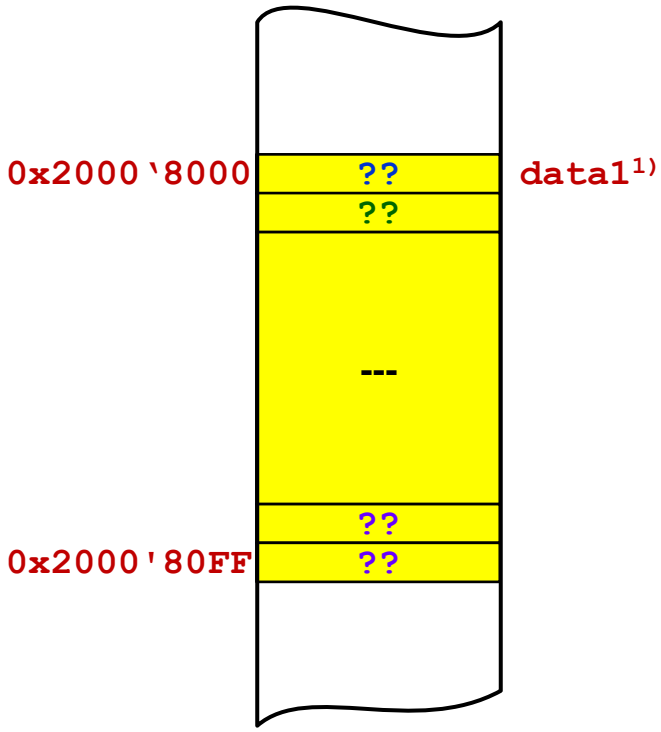
Object File Sections

School of Engineering
InES Institute of Embedded Systems

Memory Allocation in Assembly

- Directives for uninitialized data
  - **SPACE** or **%**
    with number of bytes to be reserved
  - Reserves number of bytes without initialing them

```
AREA example2, DATA, READWRITE
data1   SPACE 256
```



1) if we assume that example2 starts at **0x2000'8000**

Object File Sections

School of Engineering
InES Institute of Embedded Systems

Code Example

show_variables.c

```
uint32_t  g_init_var = 0x4D5E6F70;

uint32_t  g_noinit_var;


const uint32_t g_const = 0xDDEEFF00;

void show_variables(void)
{
    uint32_t local_var;

    const uint32_t local_const =
                            0x7261504F;

    static uint32_t static_local_var;

    local_var = 0xD4E5F607;
    ...
}
```

Into which sections will the colored objects be allocated?

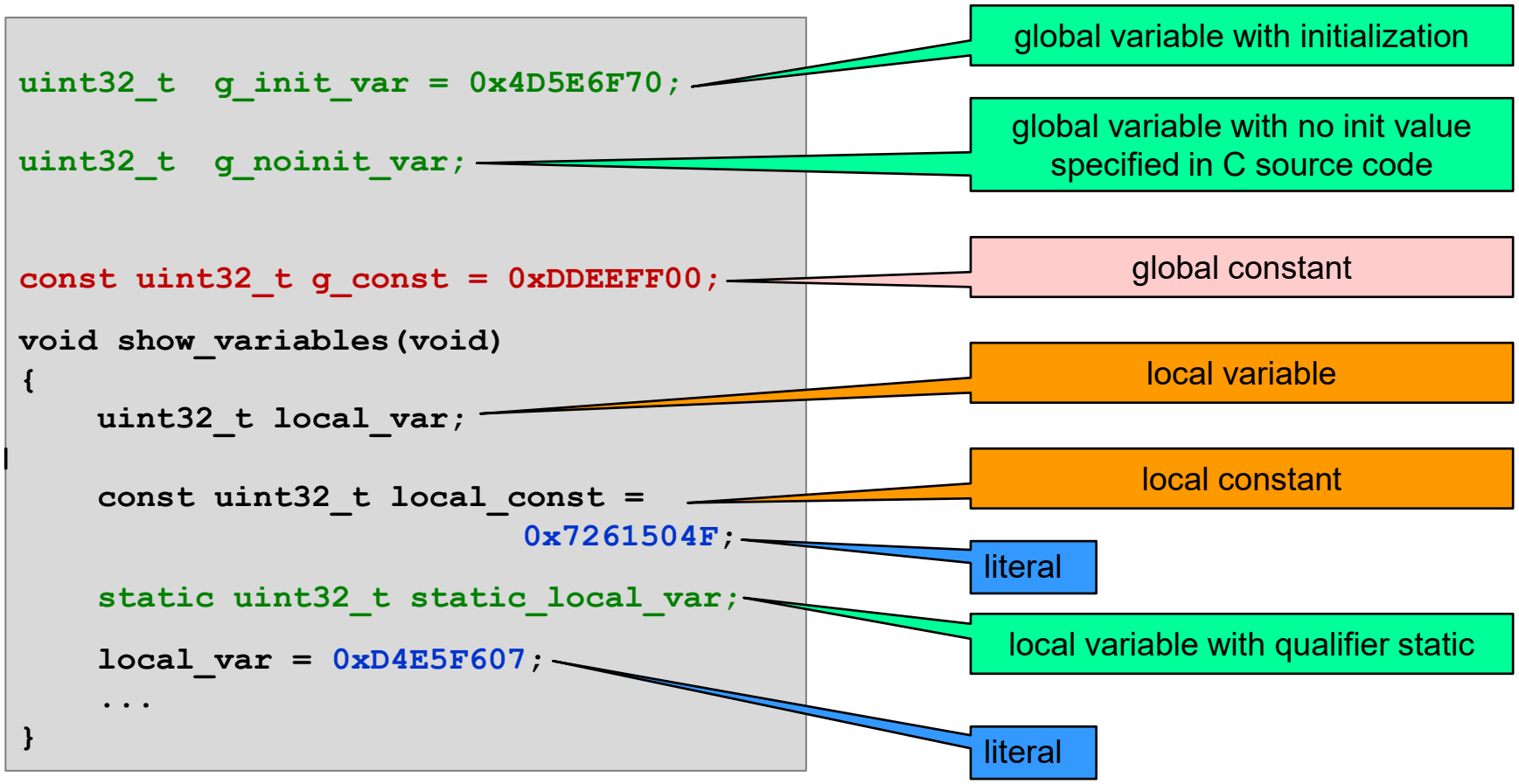
Object File Sections

School of Engineering
InES Institute of Embedded Systems

Code Example

show_variables.c

```
uint32_t  g_init_var = 0x4D5E6F70;

uint32_t  g_noinit_var;


const uint32_t g_const = 0xDDEEFF00;

void show_variables(void)
{
    uint32_t local_var;

    const uint32_t local_const =
                            0x7261504F;

    static uint32_t static_local_var;

    local_var = 0xD4E5F607;
    ...
}
```



Literal Constant	A fixed value in source code **without** an associated symbol name "variable" that cannot be changed after first assignment

Object File Sections

Code Example

show_variables.c

```
uint32_t  g_init_var = 0x4D5E6F70;

uint32_t  g_noinit_var;


const uint32_t g_const = 0xDDEEFF00;

void show_variables(void)
{
    uint32_t local_var;

    const uint32_t local_const =
                              0x7261504F;

    static uint32_t static_local_var;

    local_var = 0xD4E5F607;
    ...
}
```

stored in registers [1)]

literal

assembly

```
AREA GlobalVars, DATA, READWRITE

g_init_var          DCD     0x4d5e6f70
g_noinit_var        DCD     0x00000000
static_local_var    DCD     0x00000000
```

```
AREA MyConsts, DATA, READONLY

g_const             DCD     0xddeeff00
```

```
AREA MyCode, CODE, READONLY,

show_variables      ...
                    ...
                    BX      lr

lit1                DCD     0x7261504f
lit2                DCD     0xd4e5f607
```

[1)] if possible. E.g. if you use the address operator (&) on a local variable it will be allocated on the stack.

Conclusion

School of Engineering
InES Institute of Embedded Systems

- **Components Cortex-M CPU**
  - Core Registers: R0-R12, SP, LR, PC
  - 32-bit ALU
  - Flags (APSR)
  - Control Unit with IR (Instruction Register)
  - Bus Interface
- **Instruction Types**
  - Data transfer, data processing, control flow
- **Program Execution**
  - Fetch – Execute
- **Memory Map**
- **Integer Types**
  - Size depends on architecture → use C99 types for portability
  - 'Little Endian' vs. 'Big Endian', alignment
- **Object File Sections**
  - CODE, DATA, STACK