

Cortex-M Architecture

Computer Engineering 1

■ ARM (Cortex-M)

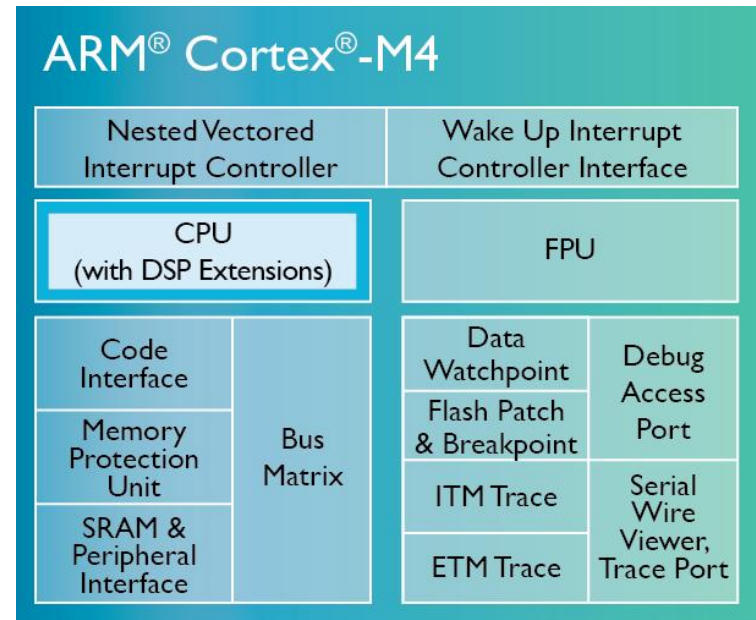
1985



ARM1
25k Transistors
5 MHz

2014

1'400 x
transistors
→
36 x
clock speed



ARMv7 - Cortex-M4
35 Mio Transistors
180 MHz
4 mm²

■ 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

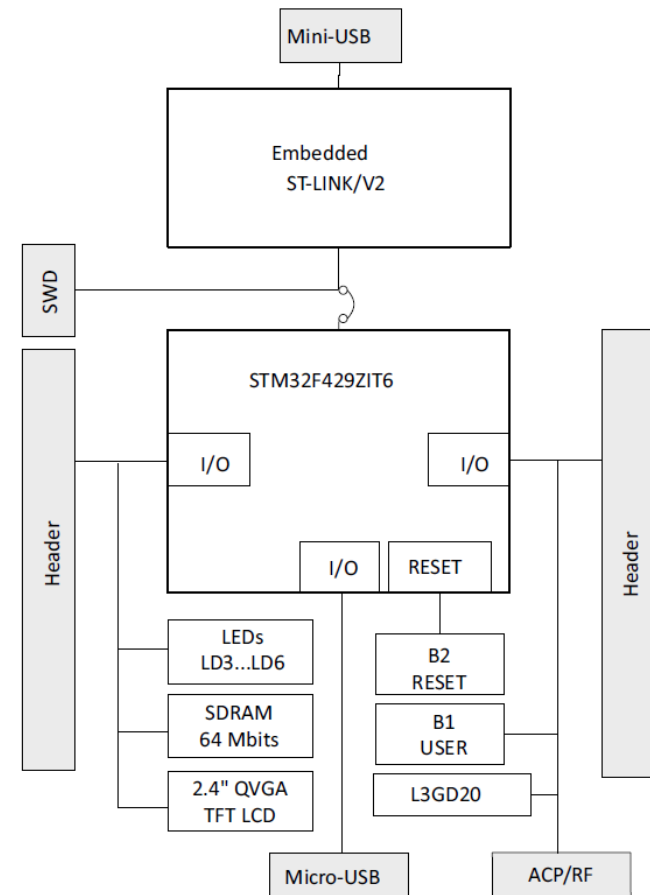
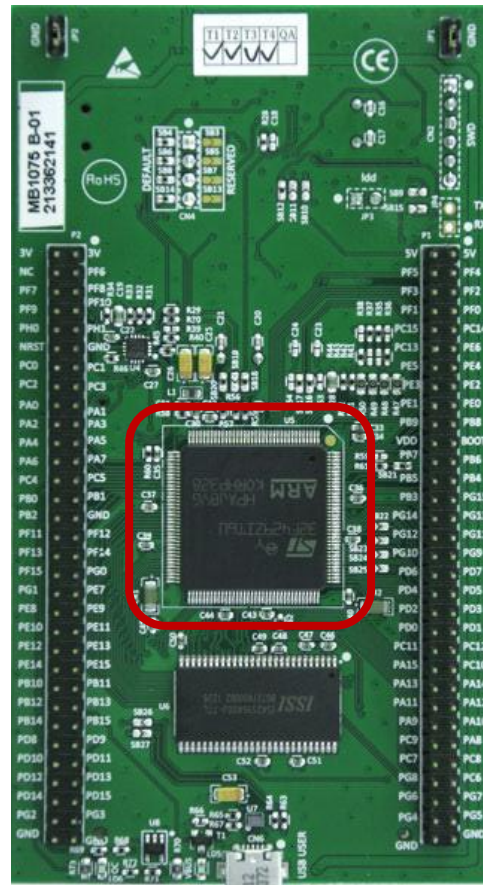
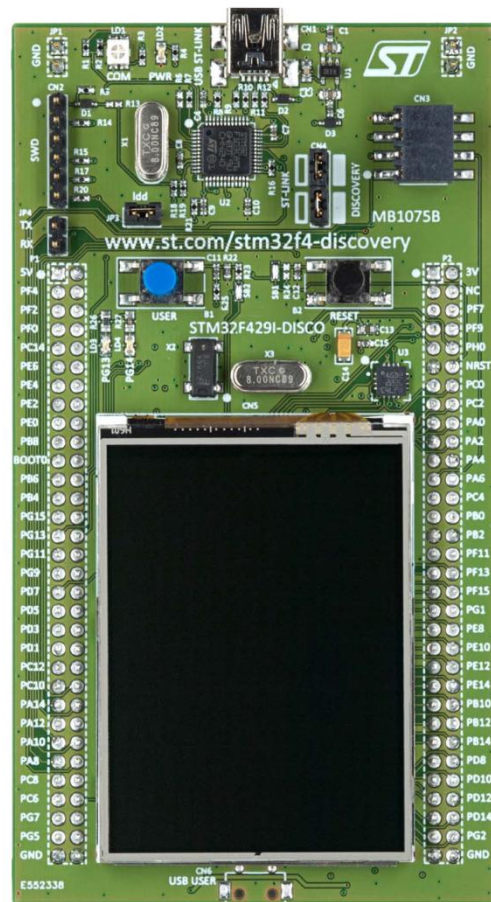
Learning Objectives

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

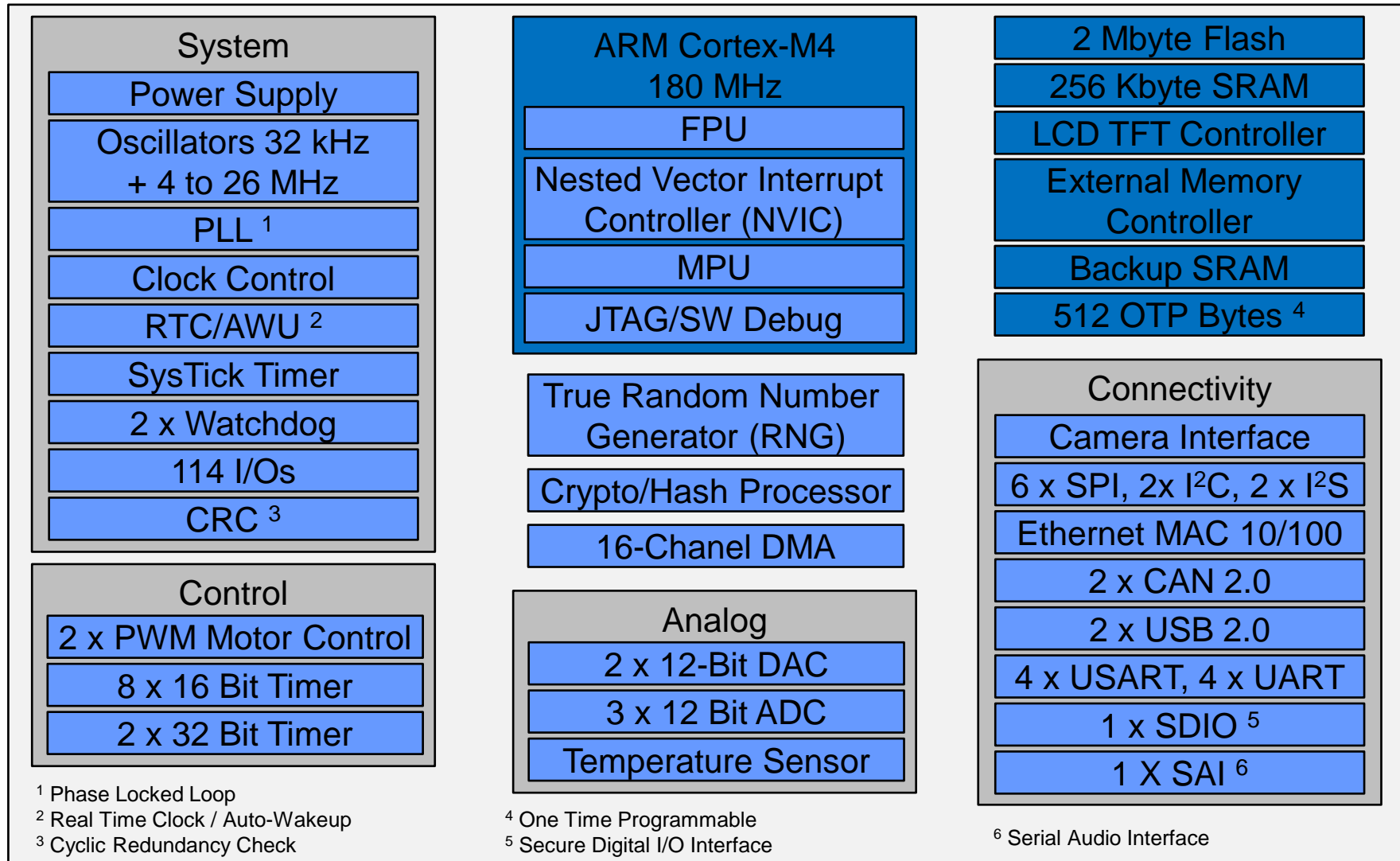
Hardware Platform: STM32F4-Discovery

■ Evaluation board STM32F4-DISCO



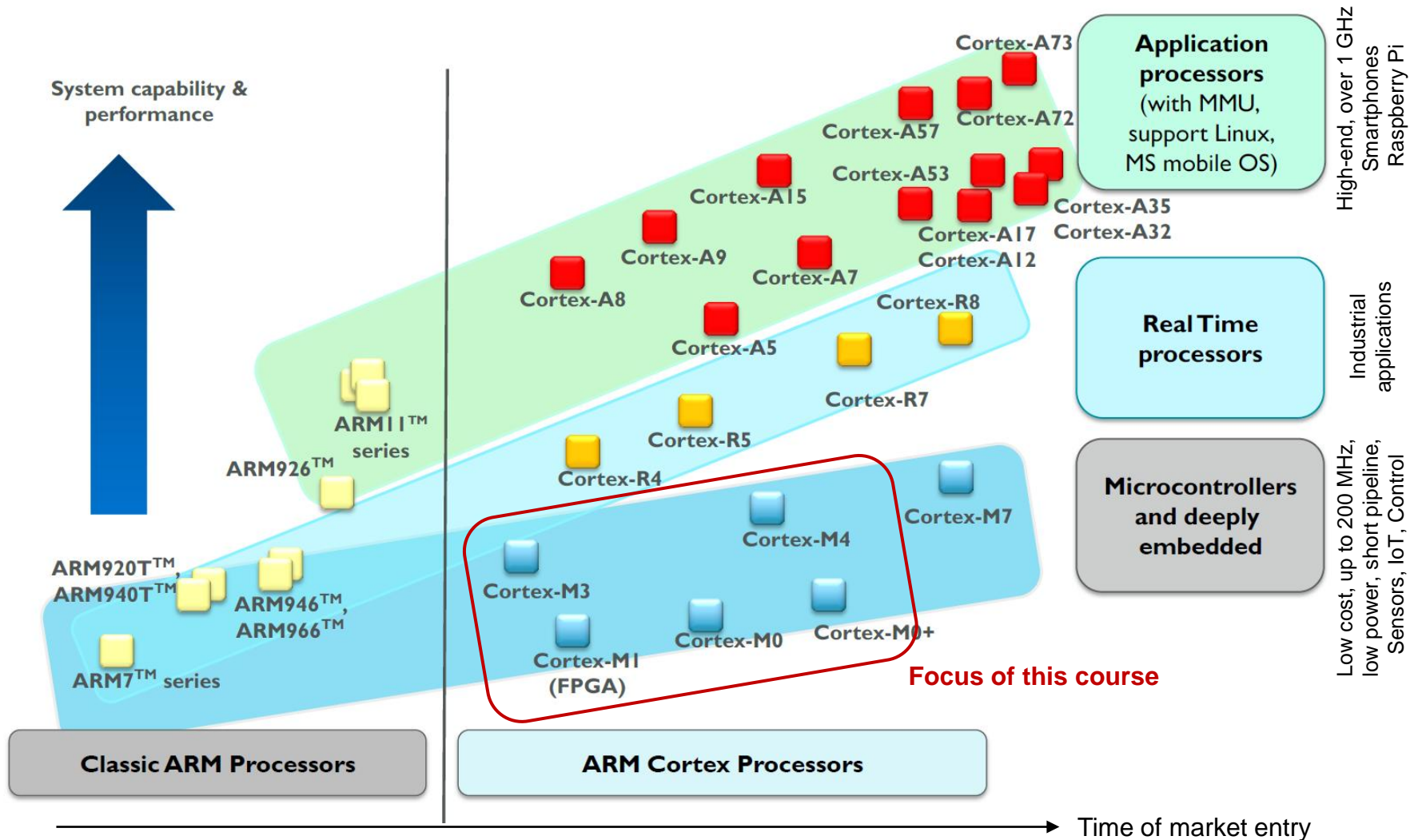
source: STMicroelectronics

Hardware Platform: STM32F429I



based on source of STMicroelectronics

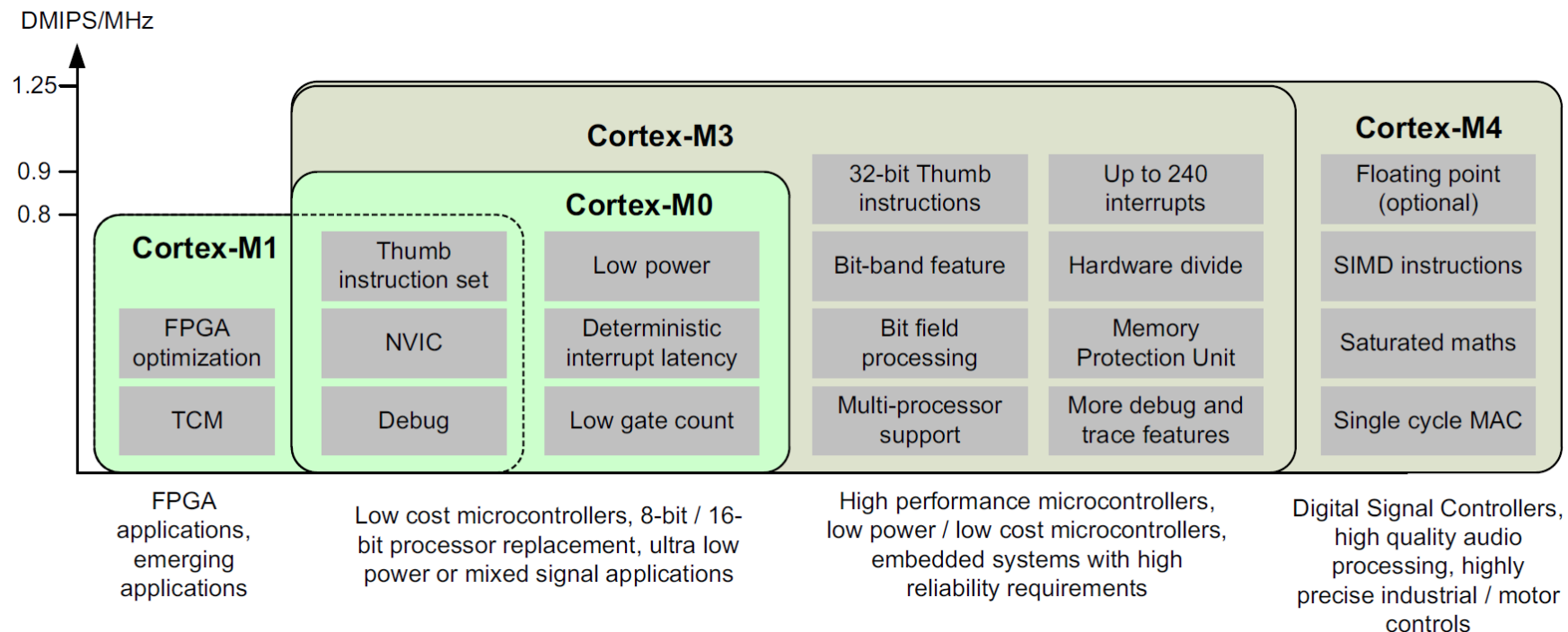
Hardware Platform: ARM Processor Portfolio



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

■ 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*

■ Instruction Set Architecture (ISA)

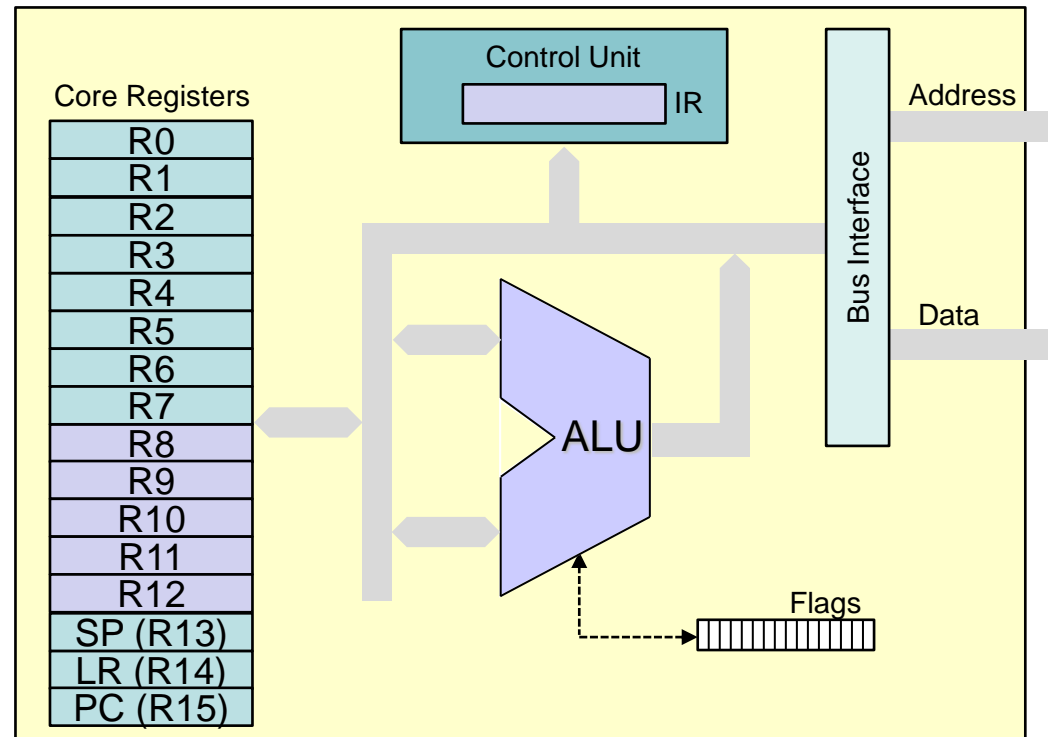
→ CT1

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 Components

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

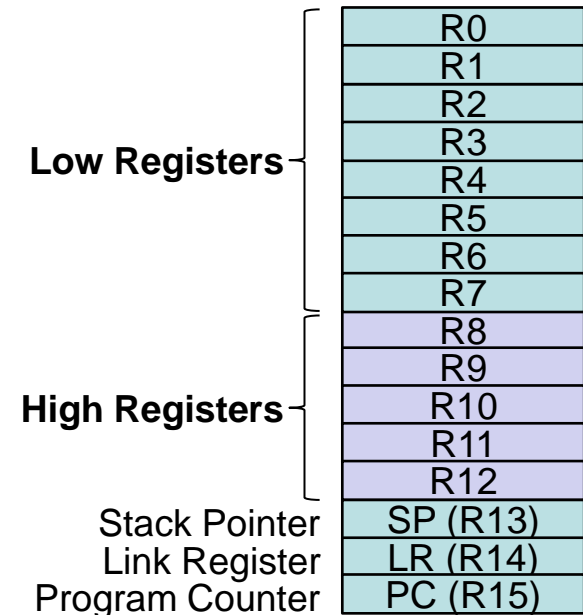


simple CPU model based on the Programmers' Model of the ARM Cortex-Mx CPUs ¹⁾

¹⁾ without pipelining

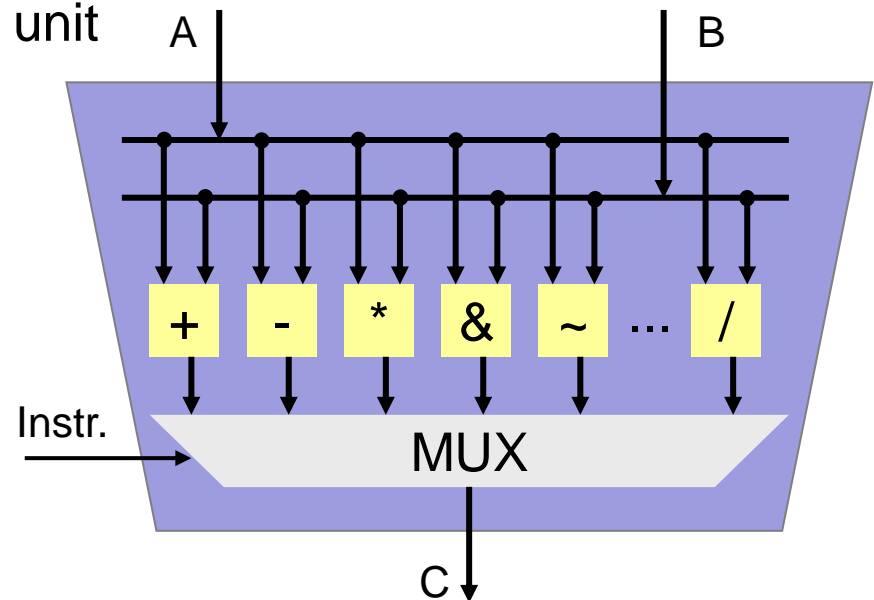
■ 16 Core Registers

- 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



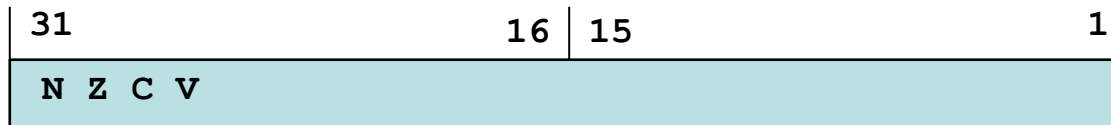
■ ALU – Arithmetic Logic Unit

- 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



■ APSR¹⁾ 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

Registers	
Register	Value
Core	
R0	0x20000078
R1	0x20000278
R2	0x20000278
R3	0x20000278
R4	0x08000860
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x08000860
R11	0x08000860
R12	0x00000000
R13 (SP)	0x20000678
R14 (LR)	0x08000243
R15 (PC)	0x08000270
xPSR	0x41000000
N	0
Z	1
C	0
V	0

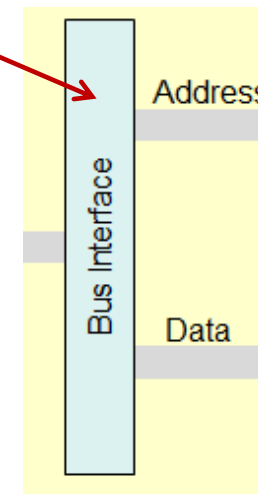
¹⁾ APSR: Application Processor Status Register

■ 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



■ 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

■ 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 Types

- **Data transfer**

- Copy content of one register to another register
- Load registers with data from memory
- Store register contents into memory

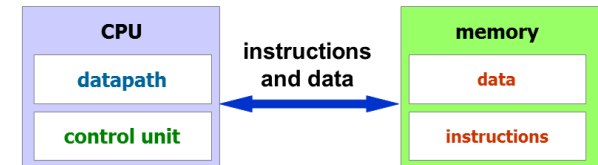
- **Data processing**

- 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

■ Instructions Cortex-M

ARMv7-M Architecture

ARMv6-M Architecture

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

VABS	VADD	VCMP	VCMPPE	VCVT	VCVTR	VDIV	VLDM
VLDR	VMLA	VMLS	VMOV	VMRS	VMSR	VMUL	VNEG
VNMLA	VMMLS	VNMUL	VPOP	VPUSH	VSQRT	VSTM	VSTR
VSUB	VFMA	VFMS	VFNMA	VFNMS			
Cortex-M4 FPU							
PKH	QADD	QADD16	QADD8	QASX	QDADD	QDSUB	QSAX
QSUB	QSUB16	QSUB8	SADD16	SADD8	SASX	SEL	SHADD16
SHADD8	SHASX	SHSAX	SHSUB16	SHSUB8	SMLABB	SMLABT	SMLATB
SMLATT	SMLAD	SMLALBB	SMLALBT	SMLALTB	SMLALTT	SMLALD	SMLAWB
SMLAWT	SMLSD	SMLSLD	SMMLA	SMMLS	SMMUL	SMUAD	SMULBB
						SMULBT	SMULTT
						SMULTB	SMULWT
						SMULWB	SMUSD
ADC	ADD	ADR	AND	ASR	B	SSAT16	SSAX
CLZ	BFC	BFI	BIC	CDP	CLREX	SSUB16	SSUB8
CBNZ	CBZ	CMN	CMP	DBG	EOR	SXTAB	SXTAB16
LDMIA	LDMDB	LDR	LDRB	LDRBT	LDRD	SXTAH	SXTB16
LDREX	LDREXB	LDREXH	LDRH	LDRHT	LDRSB	UADD16	UADD8
LDRSBT	LDRSHT	LDRSH	LDRT	MCR	LSL	UASX	UHADD16
LSR	MCRR	MLS	MLA	MOV	MOVT	UHADD8	UHASX
MRC	MRRC	MUL	MVN	NOP	ORN	UHSAX	UHSUB16
ORR	PLD	PLDW	PLI	POP	PUSH	UHSUB8	UMAAL
RBIT	REV	REV16	REVSH	ROR	RRX	UQADD16	UQADD8
			RSB	SBC	SBFX	UQASX	UQSAX
			SDIV	SEV	SMLAL	UQSUB16	UQSUB8
			SMULL	SSAT	STC	USAD8	USADA8
			STMIA	STMDB	STR	USAT16	USAX
			STRB	STRBT	STRD	USUB16	USUB8
			STREX	STREXB	STREXH	UXTAB	UXTAB16
			STRH	STRHT	STRT	UXTAH	UXTB16
			SUB	SXTB	SXTH		
			TBB	TBH	TEQ		
			TST	UBFX	UDIV		
			UMLAL	UMULL	USAT		
			UXTB	UXTH	WFE		
			WFI	YIELD	IT		
Cortex-M3							
BKPT	BLX	ADC	ADD	ADR			
BX	CPS	AND	ASR	B			
DMB	BL	BIC					
DSB	CMN	CMP	EOR				
ISB	LDR	LDRB	LDM				
MRS	LDRH	LDRSB	LDRSH				
MSR	LSL	LSR	MOV				
NOP	REV	MUL	MVN	ORR			
REV16	REVSH	POP	PUSH	ROR			
SEV	SXTB	RSB	SBC	STM			
SXTH	UXTB	STR	STRB	STRH			
UXTH	WFE	SUB	SVC	TST			
WFI	YIELD						
Cortex-M0/M1							

■ 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

■ 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.

■ 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

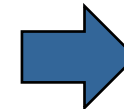
■ Load program into memory

- 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
...	...
...	...

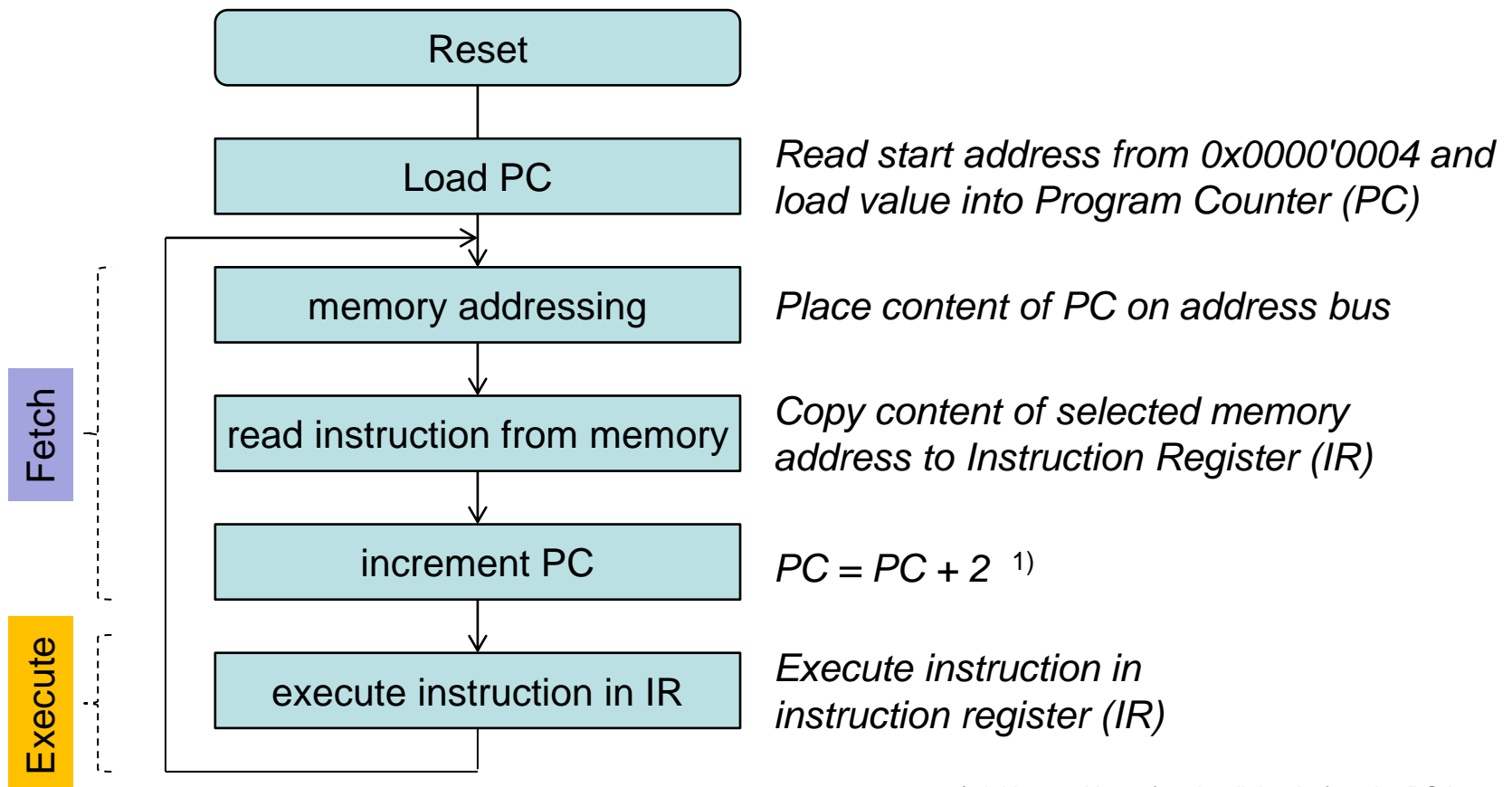
0x0000'0000

linker adds offset of 0x0800'0000 before loading

■ Sequence

Precondition

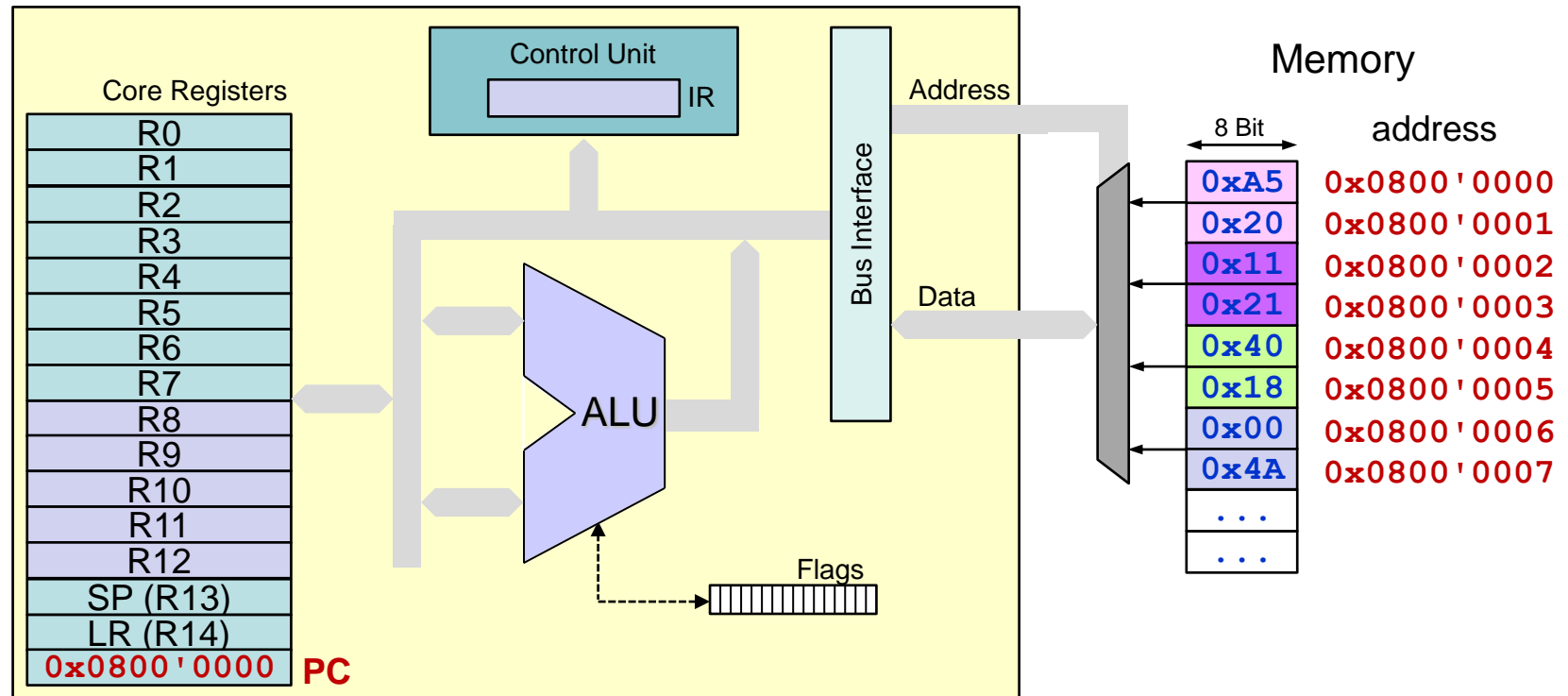
- Loader has copied executable code into memory
- Loader has stored code start address in memory location 0x0000'0004 (ARM convention)



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

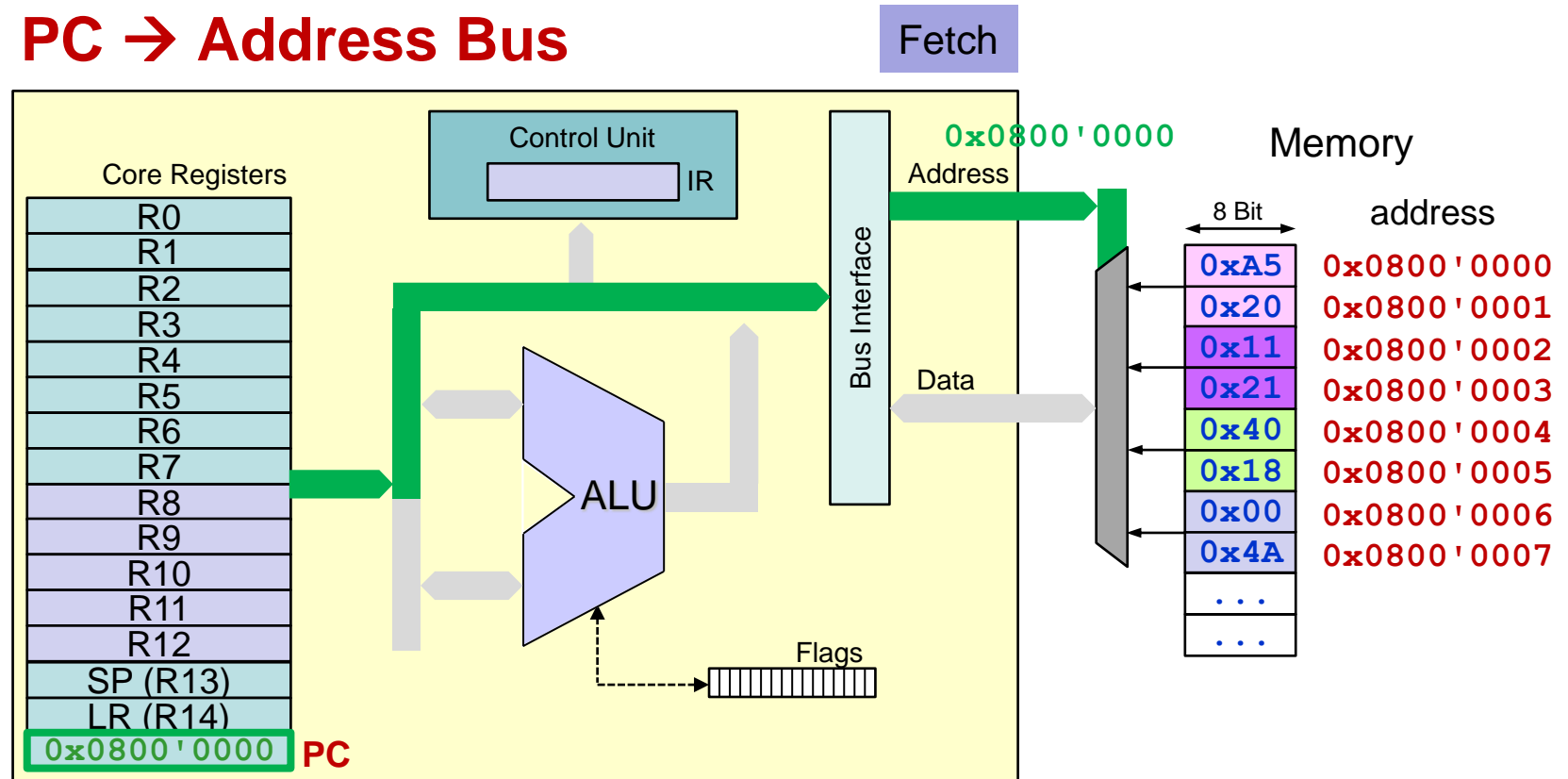


PC →

Address	Value	Instruction
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

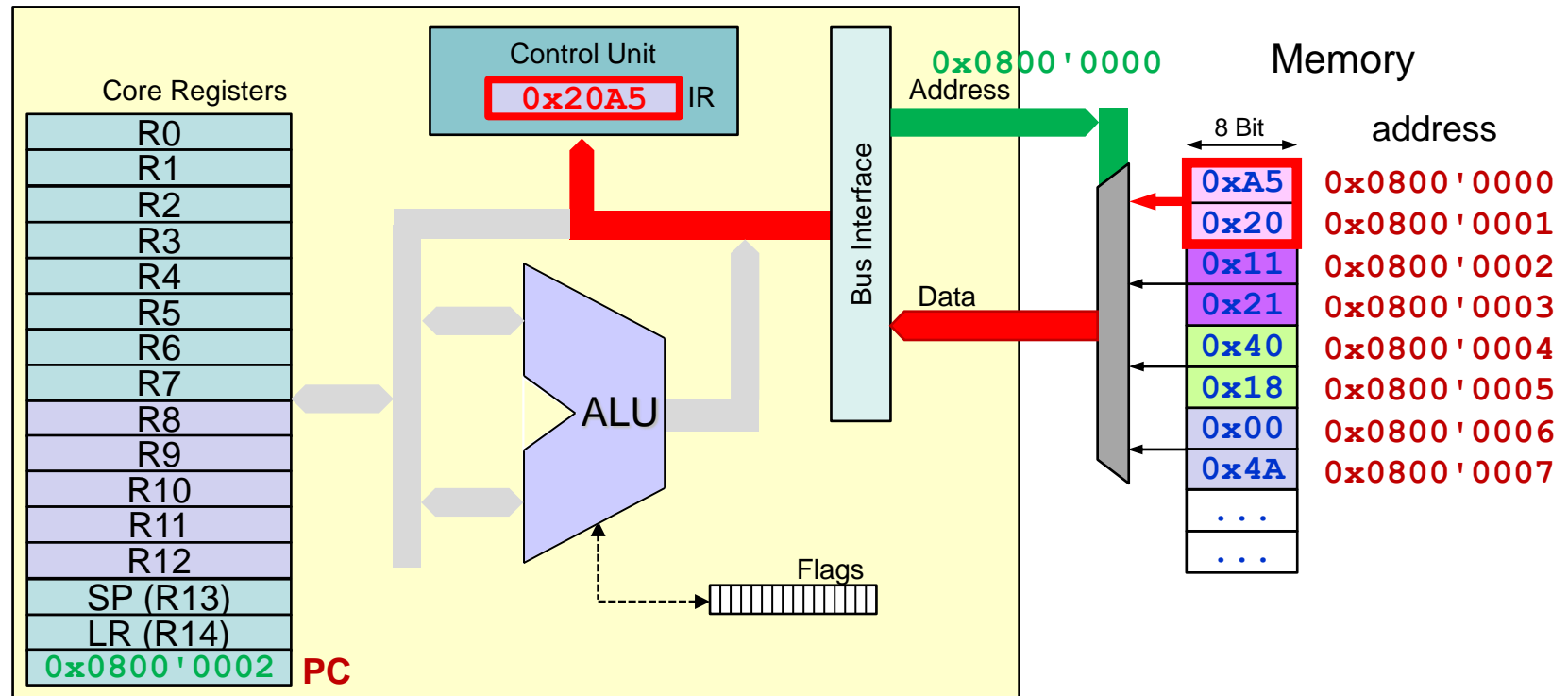


PC	00000000	20A5	demoprg	MOVS	R0, #0xA5
	00000002	2111		MOVS	R1, #0x11
	00000004	1840		ADDS	R0, R0, R1
	00000006	4A00		LDR	R2, =0x2000

Program Execution

■ Read Instruction / Increment PC

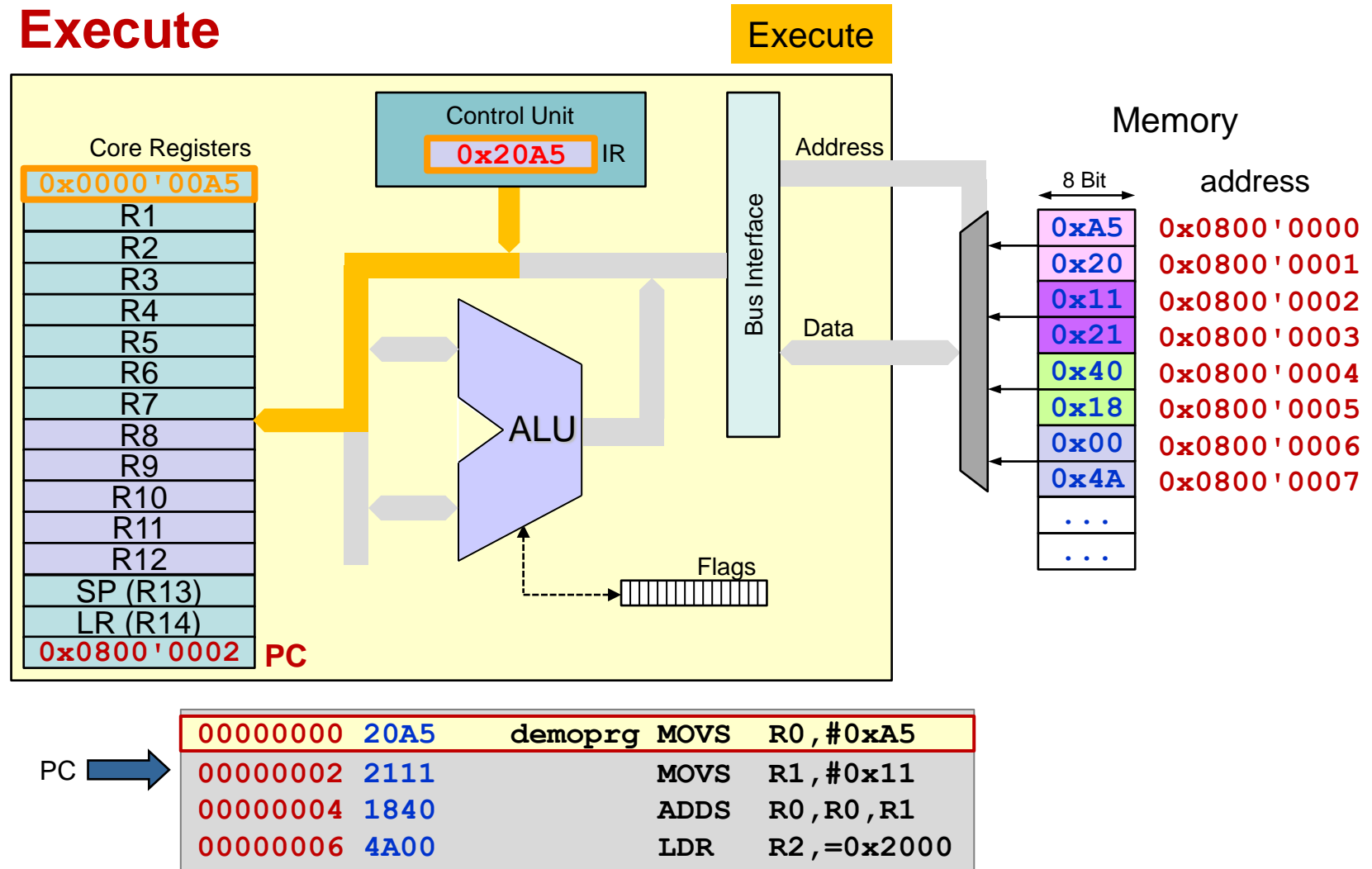
Fetch



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

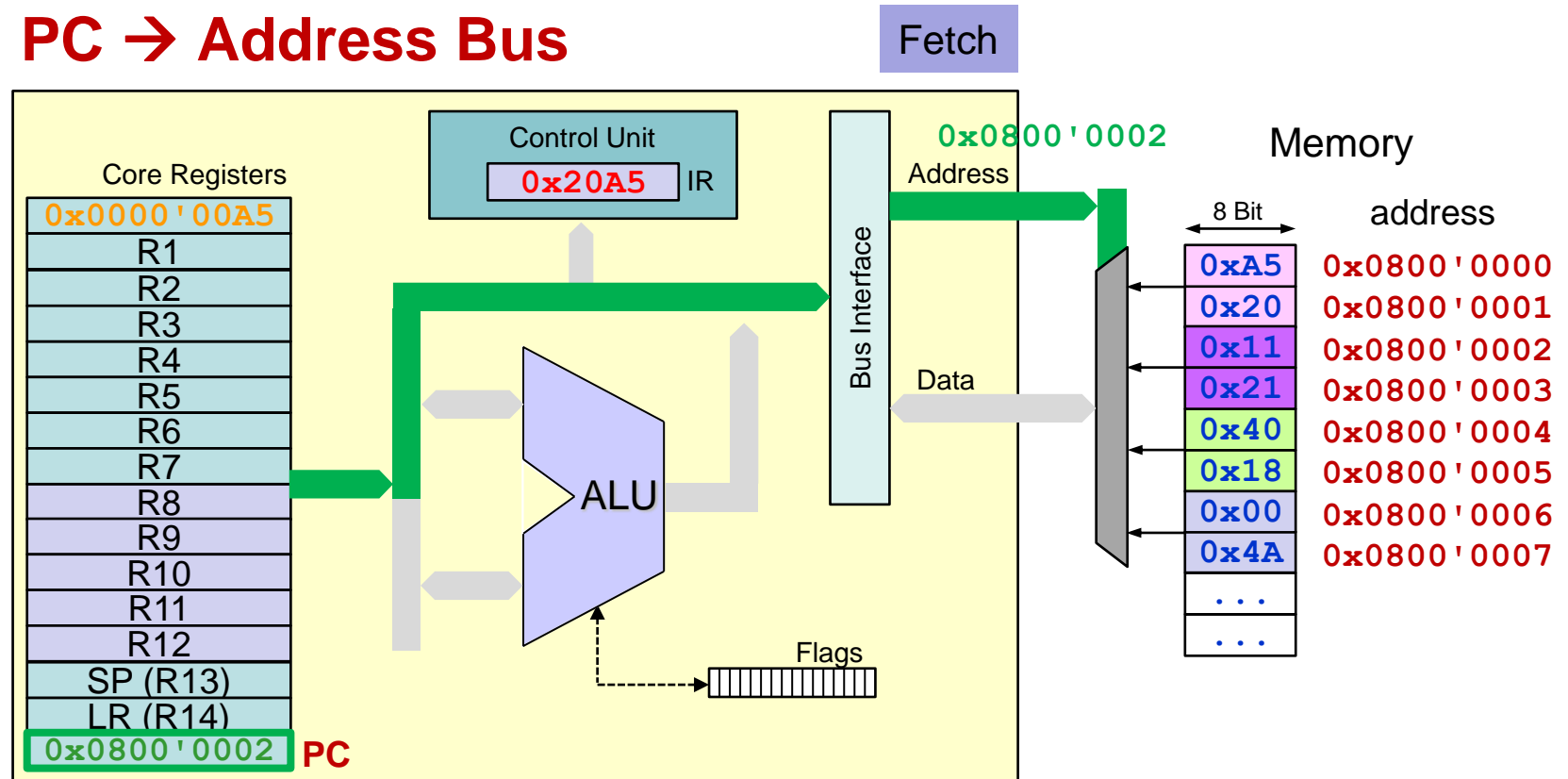
Program Execution

■ Execute



Program Execution

■ PC → Address Bus

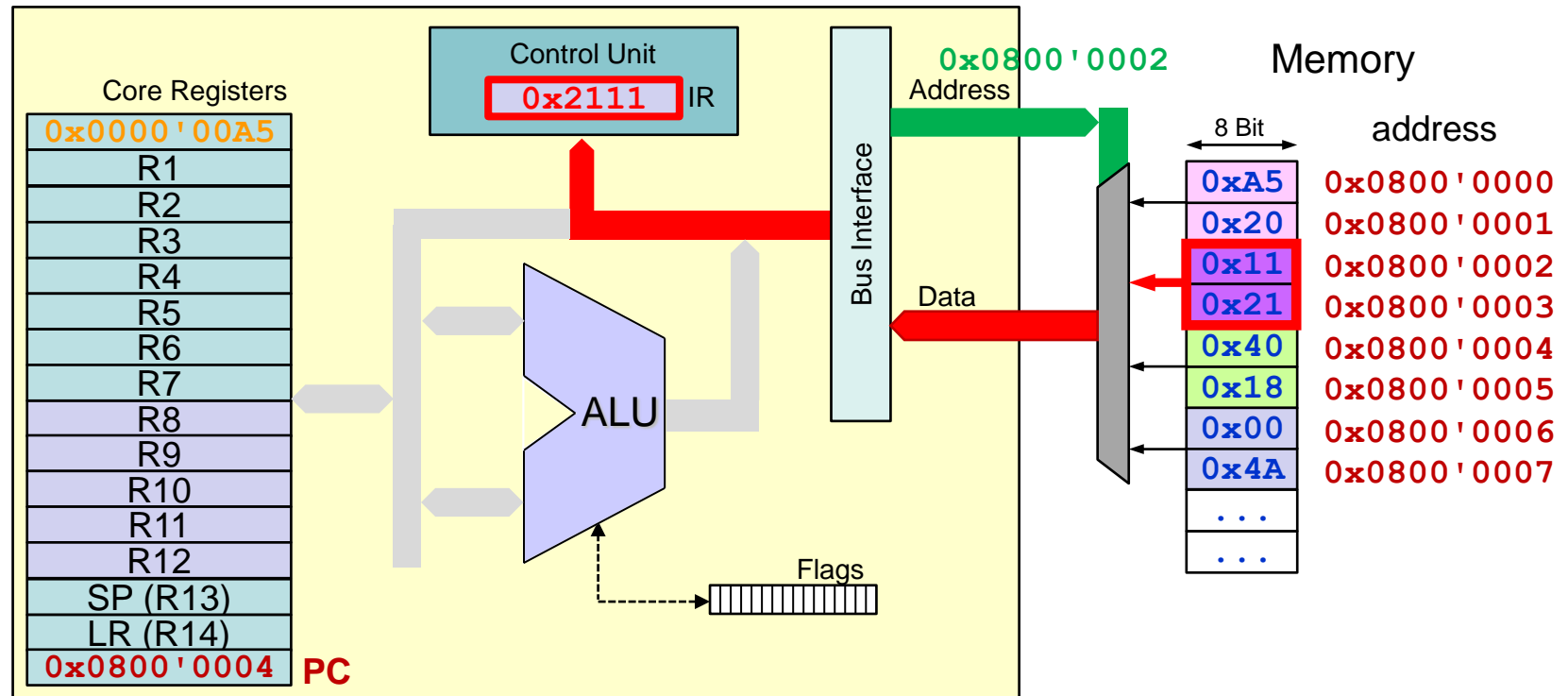


PC →	00000000	20A5	demoprg	MOVS	R0, #0xA5
	00000002	2111		MOVS	R1, #0x11
	00000004	1840		ADDS	R0, R0, R1
	00000006	4A00		LDR	R2, =0x2000

Program Execution

■ Read Instruction / Increment PC

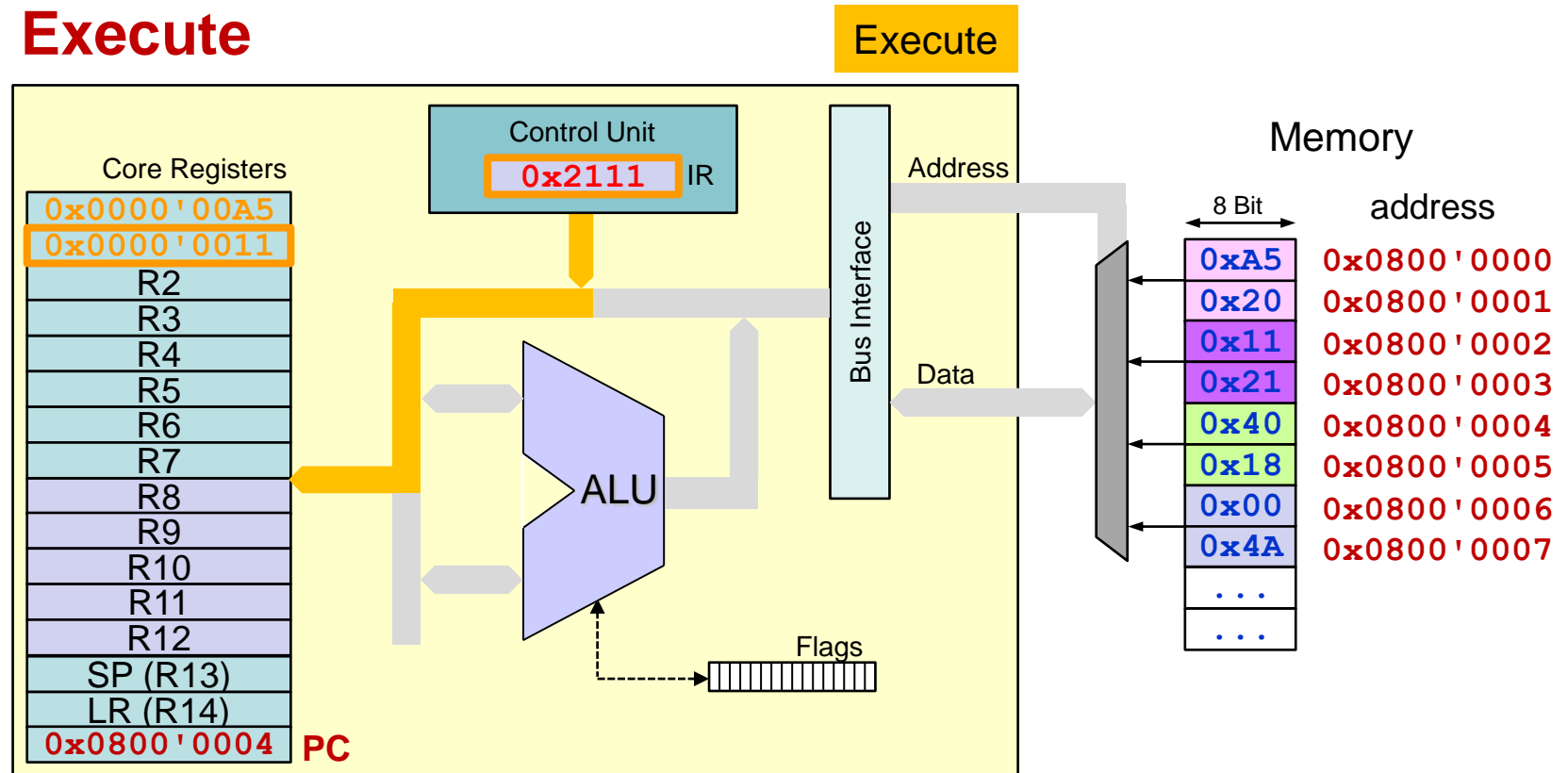
Fetch



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

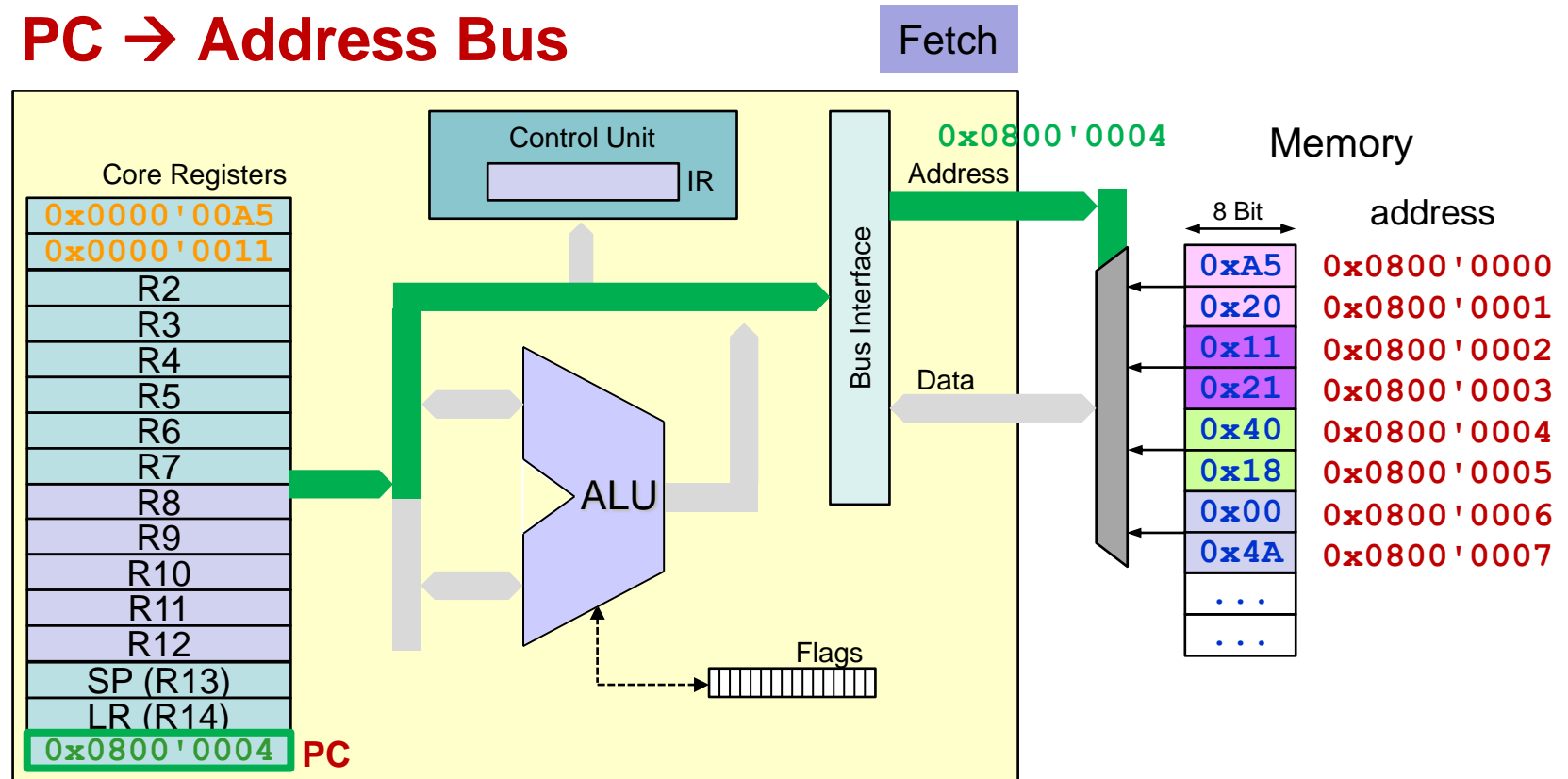
■ Execute



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

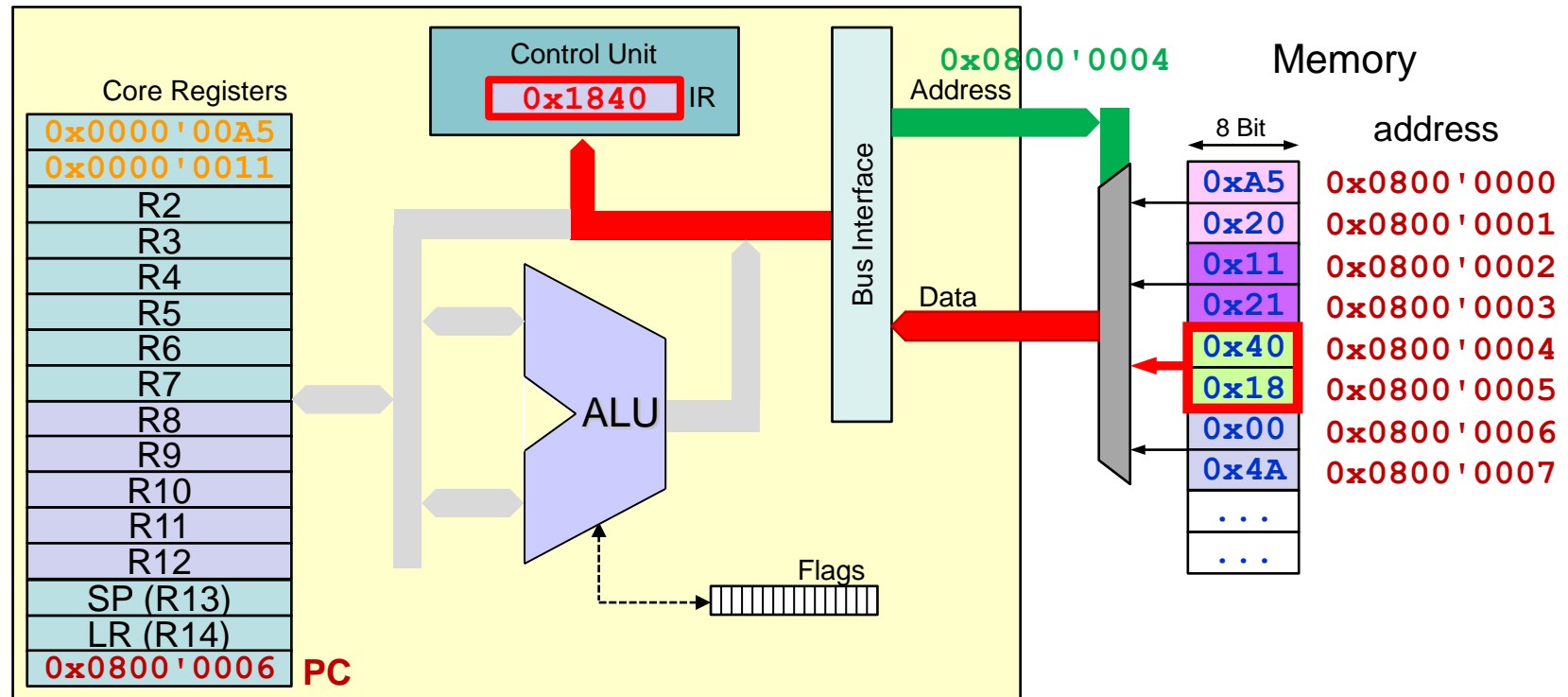


PC →	00000000	20A5	demoprg	MOVS	R0,#0xA5
	00000002	2111		MOVS	R1,#0x11
	00000004	1840		ADDS	R0,R0,R1
	00000006	4A00		LDR	R2,=0x2000

Program Execution

■ Read Instruction / Increment PC

Fetch

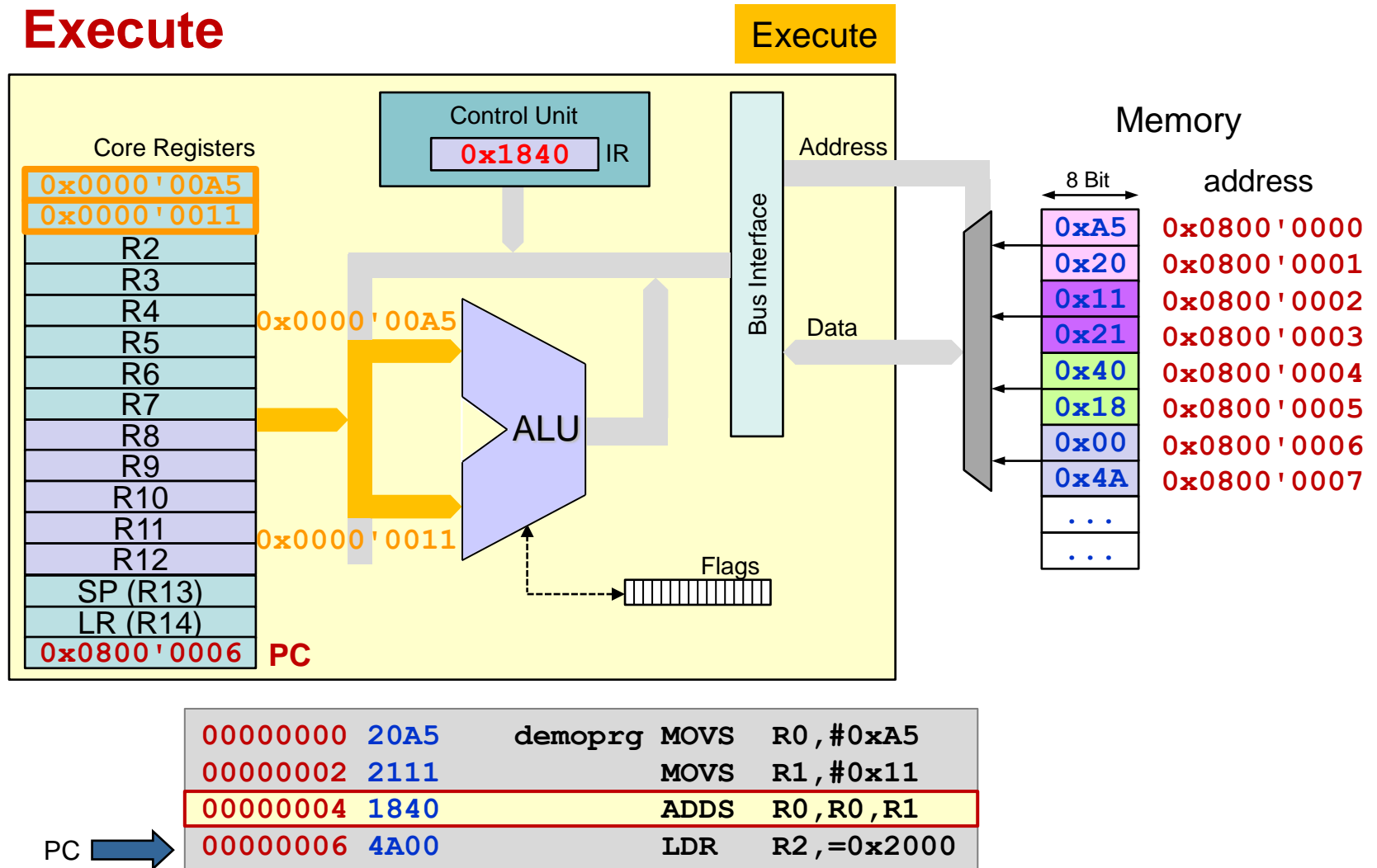


00000000	20A5	demoprg	MOVS	R0,#0xA5
00000002	2111		MOVS	R1,#0x11
00000004	1840		ADDS	R0,R0,R1
00000006	4A00		LDR	R2,=0x2000

PC →

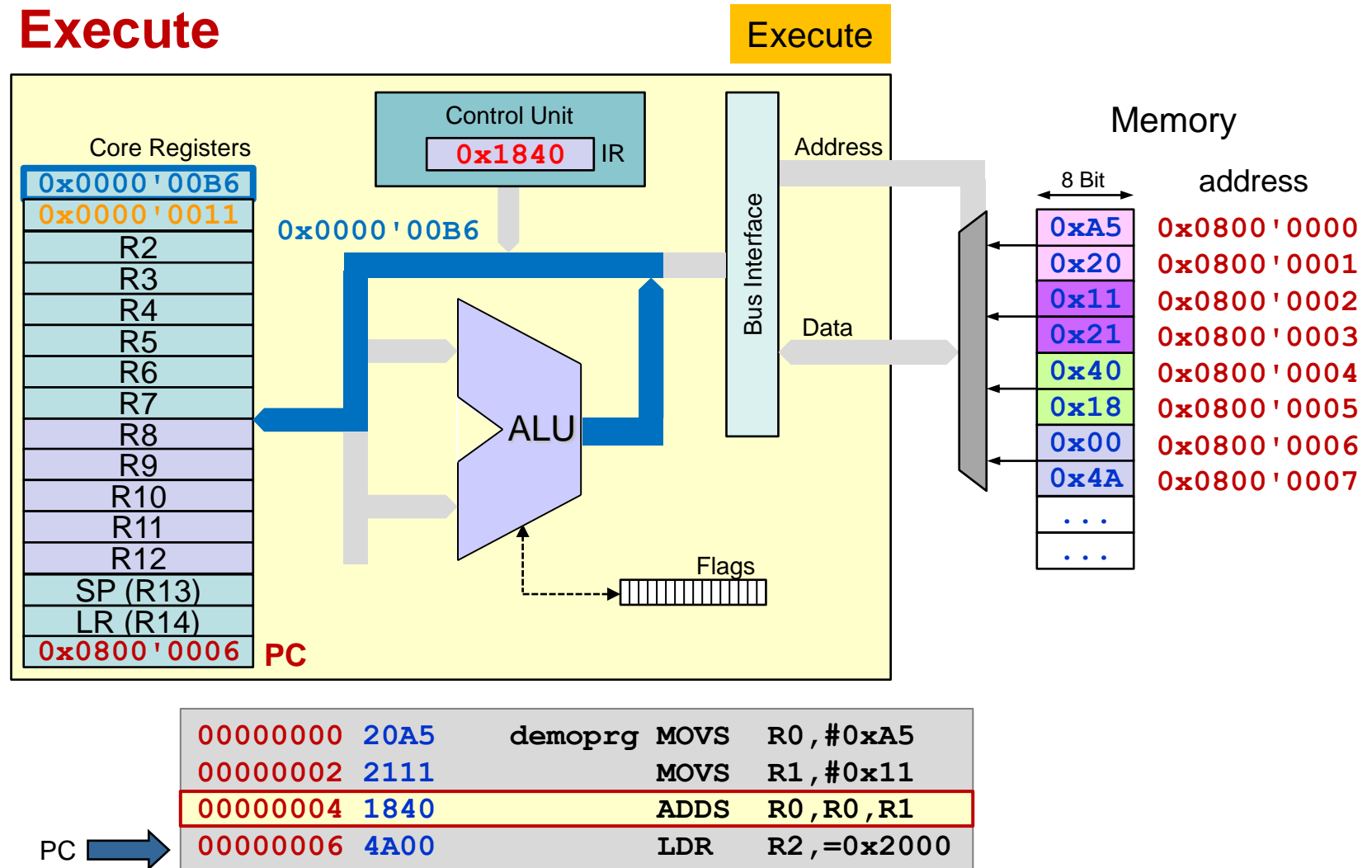
Program Execution

■ Execute

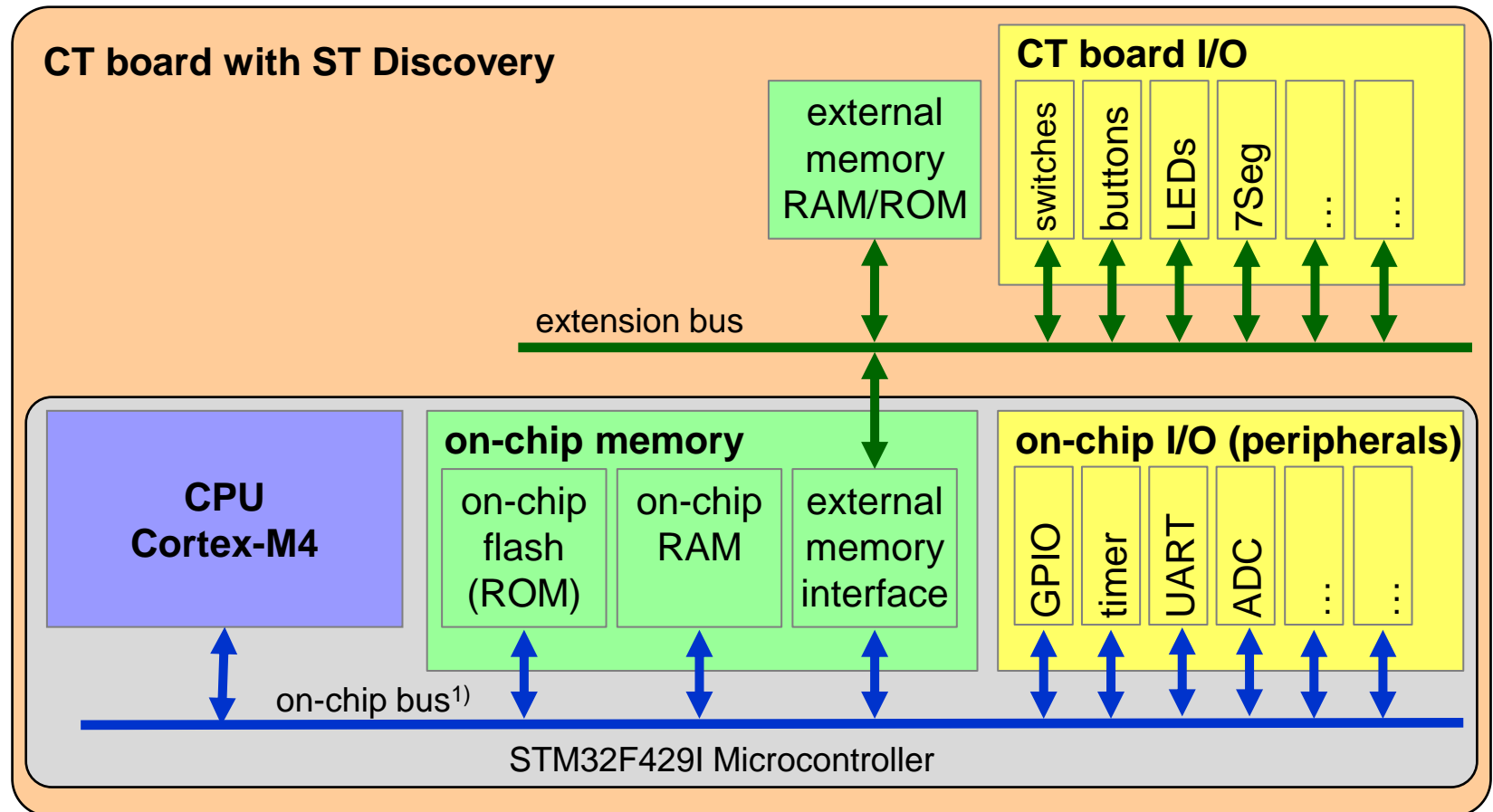


Program Execution

Execute



■ Hardware View



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

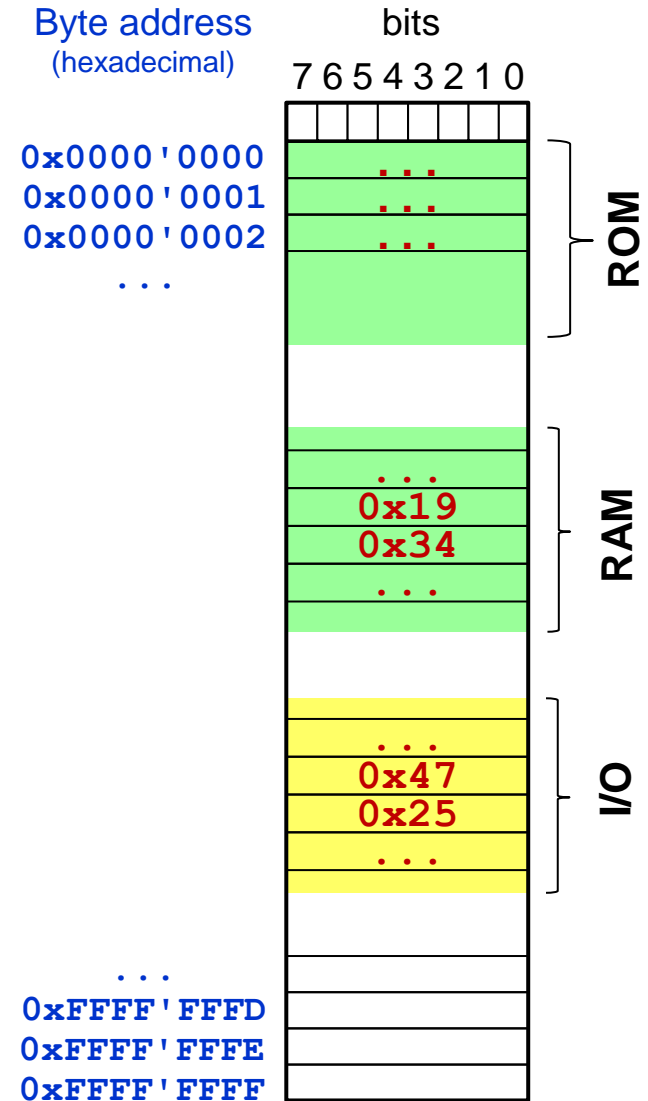
Memory Map

■ 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

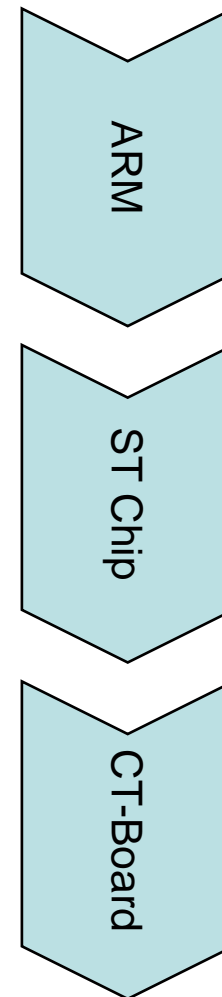
Memory maps in CT1/CT2 will be drawn with lowest address at the top and highest address at the bottom. This simplifies work with assembly listing and tables.

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



■ 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

■ CT-Board

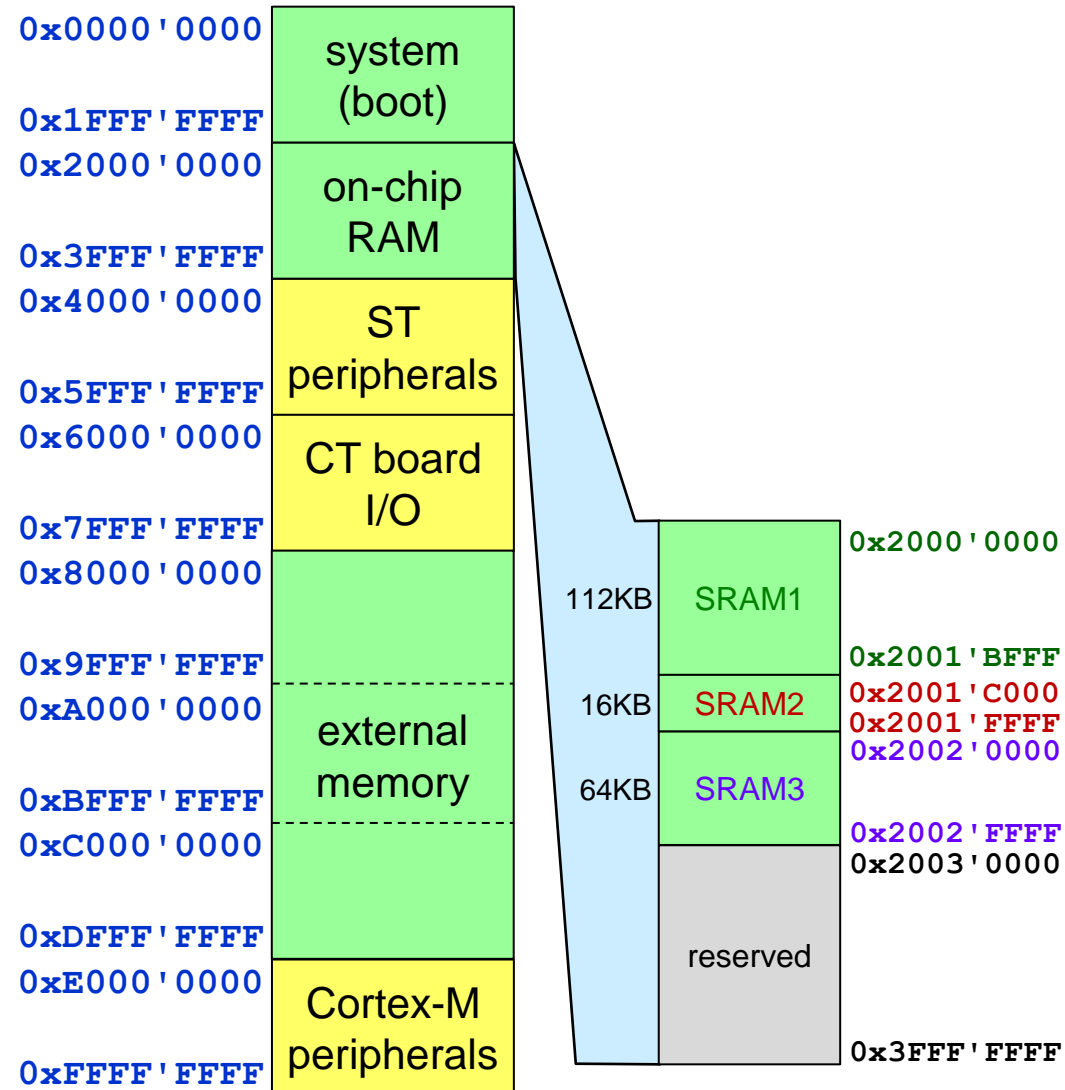
- Address space
4 GByte = 2^{32} Bytes
- From **0x0000'0000**
to **0xFFFF'FFFF**
- Partitioned into
8 blocks of
512-MByte each

0x0000'0000	system (boot)
0x1FFF'FFFF 0x2000'0000	on-chip RAM
0x3FFF'FFFF 0x4000'0000	ST peripherals
0x5FFF'FFFF 0x6000'0000	CT board I/O
0x7FFF'FFFF 0x8000'0000	
0x9FFF'FFFF 0xA000'0000	external memory
0xBFFF'FFFF 0xC000'0000	
0xDFFF'FFFF 0xE000'0000	
0xFFFF'FFFF	Cortex-M peripherals

Memory Map

■ ST chip specific

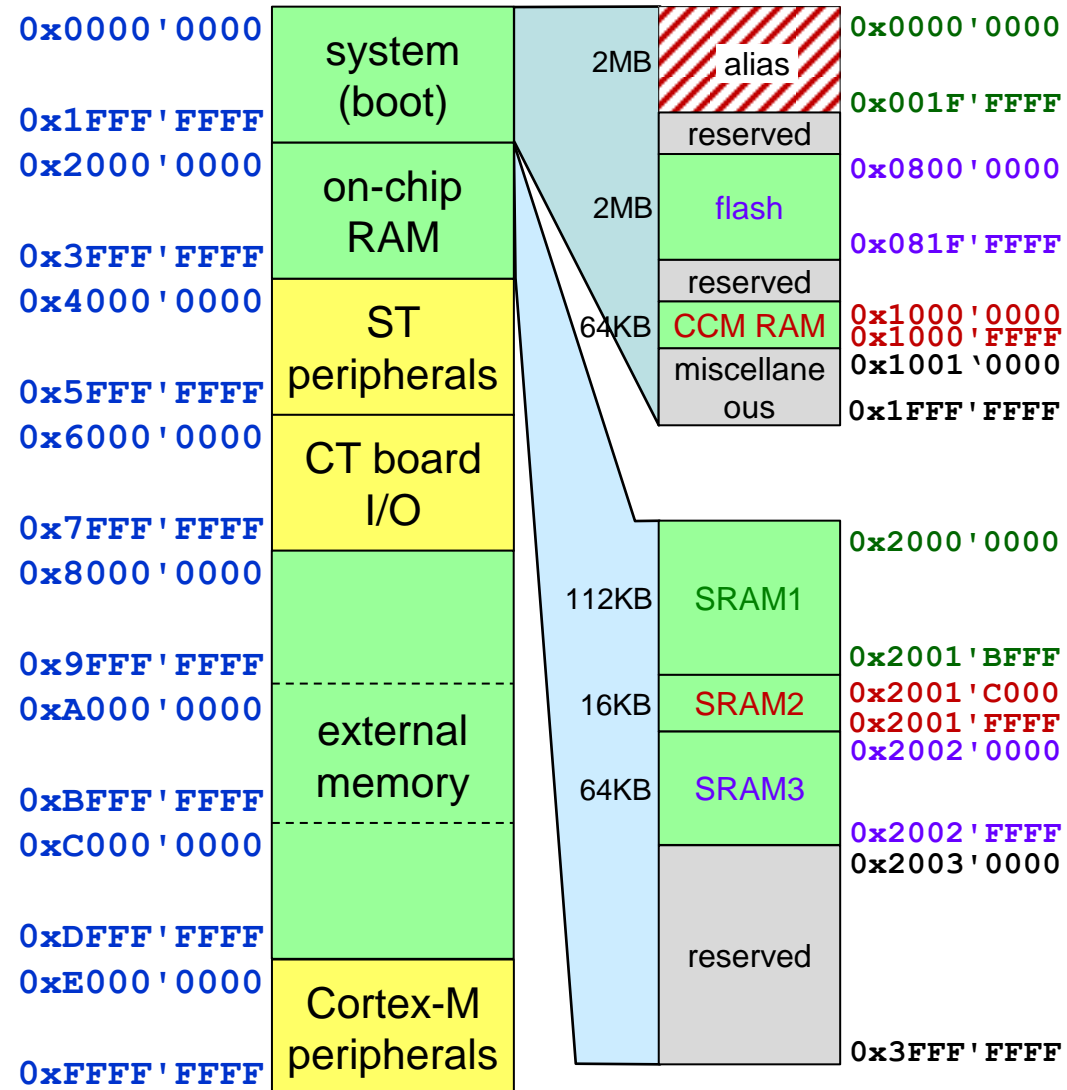
- SRAM1
 - 112 KByte
- SRAM2
 - 16 KByte
- SRAM3
 - 64 KByte



Memory Map

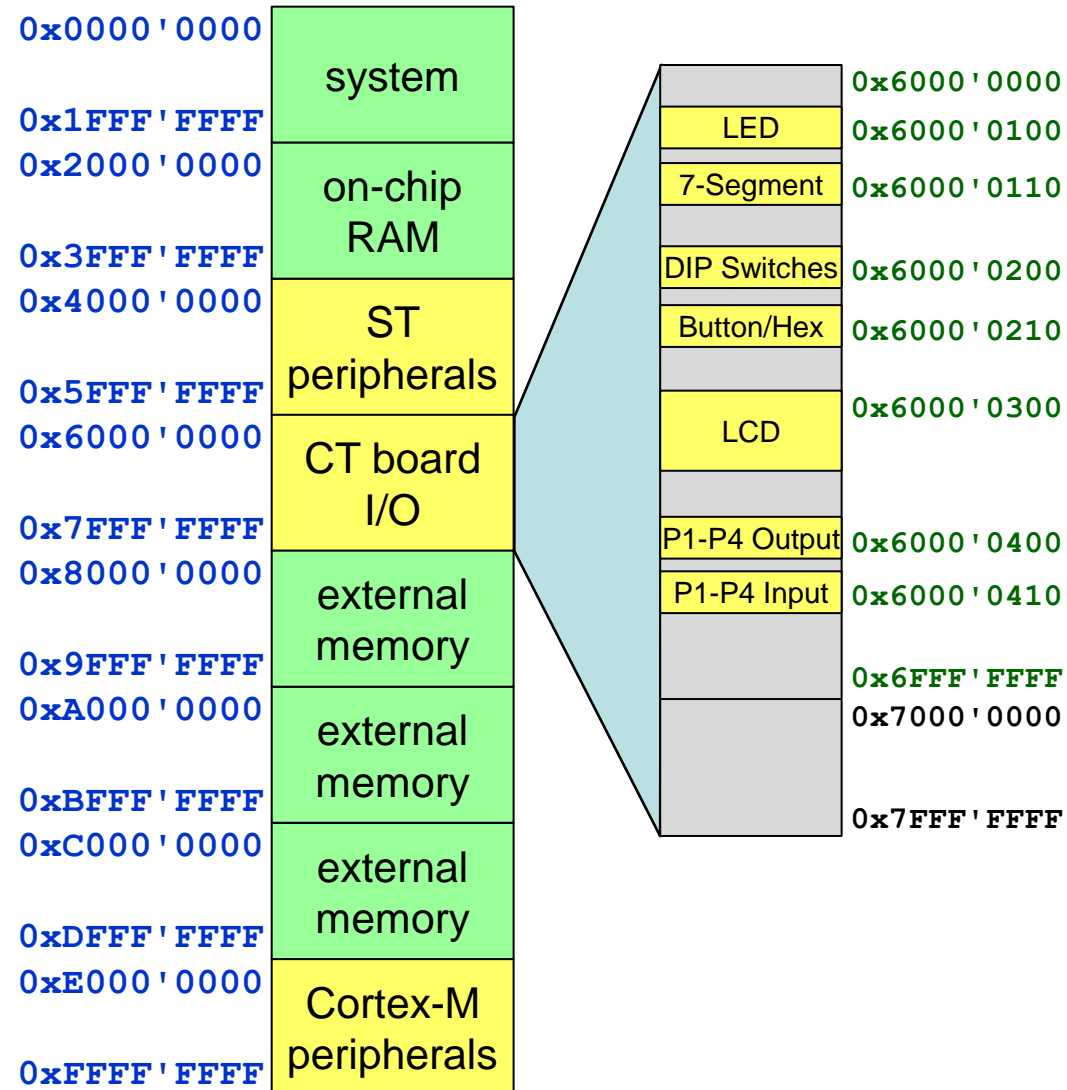
■ ST chip specific

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



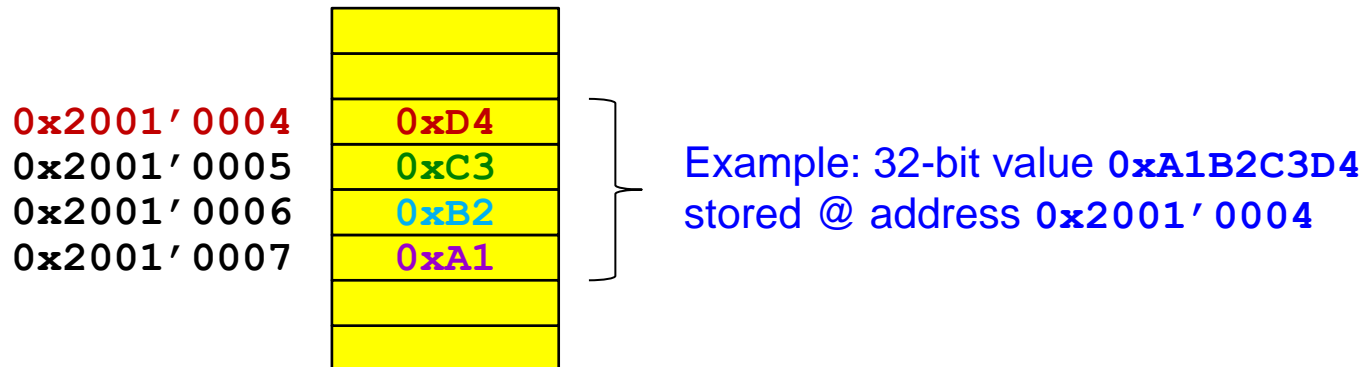
Memory Map

■ CT-Board I/O



■ 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 in C

- Sizes of integer types are platform dependent

Sizes in byte

8051

char	1
short	2
int	2
long int	4
char *	2

Cortex-Mx: Keil (ARM)

char	1
short	2
int	4
long int	4
long long int	8
void *	4

x86-64 (i7): gcc

char	1
short	2
int	4
long int	8
long long int	8
void *	8

C99 / specified width



■ ARM Cortex-M

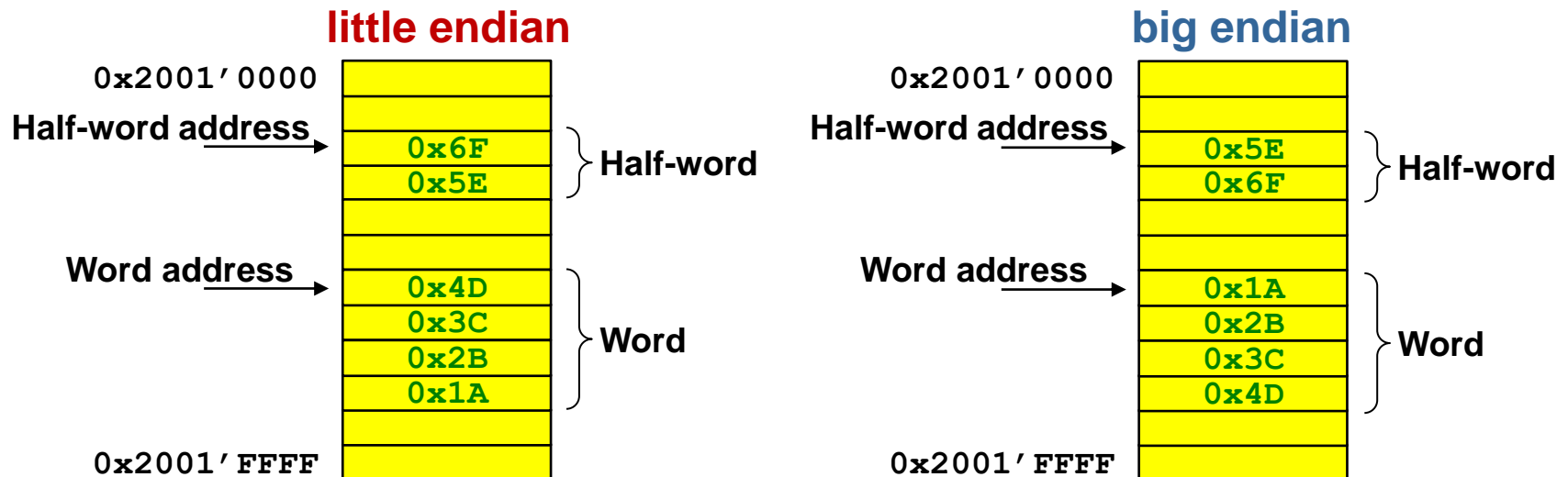
C-Type – unsigned integers	Size	Term	inttypes.h / stdint.h
<code>unsigned char</code>	8 Bit	Byte	<code>uint8_t</code>
<code>unsigned short</code>	16 Bit	Half-word	<code>uint16_t</code>
<code>unsigned int</code>	32 Bit	Word	<code>uint32_t</code>
<code>unsigned long</code>	32 Bit	Word	<code>uint32_t</code>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

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

■ How are groups of bytes arranged in memory?

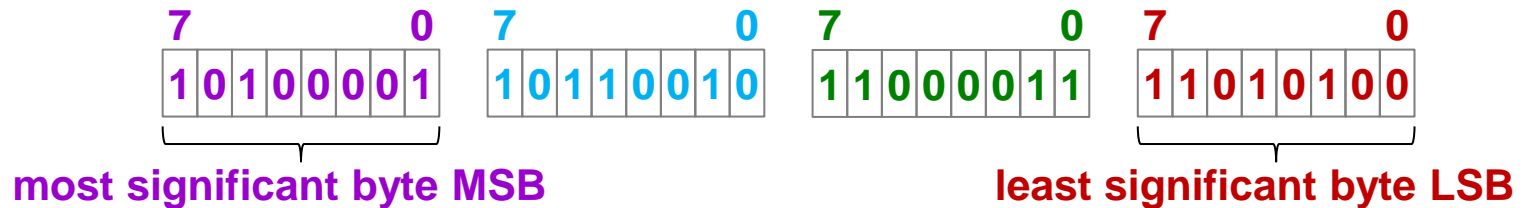
- little endian → *least significant byte* at lower address
e.g. Intel x86, Altera Nios, ST ARM (STM32)
- big endian → *most significant byte* at lower address
e.g. Freescale (Motorola), PowerPC

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



■ Example

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



little endian

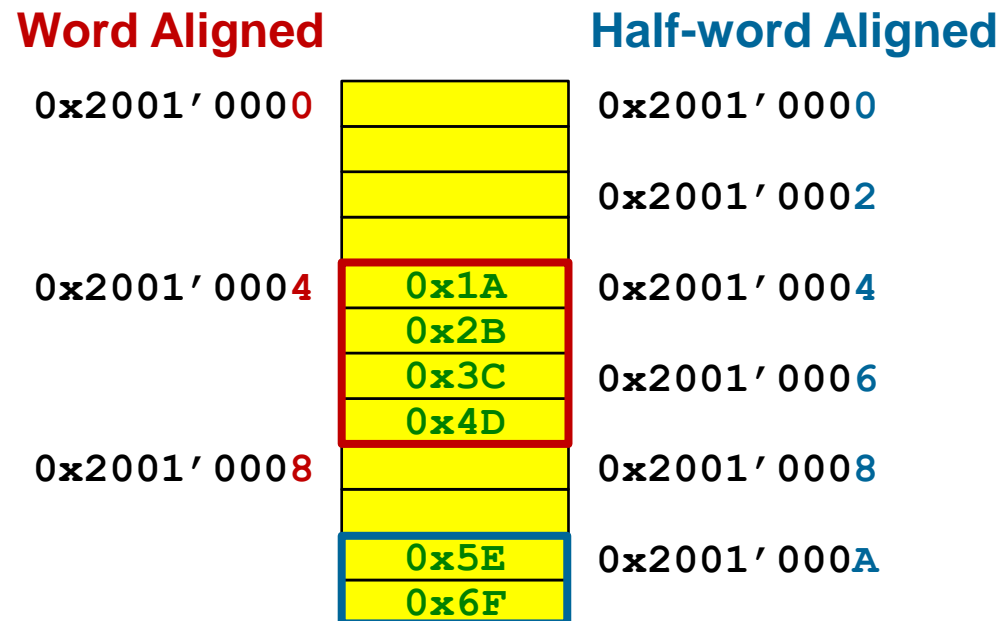
0x2001' 0004	0xD4
0x2001' 0005	0xC3
0x2001' 0006	0xB2
0x2001' 0007	0xA1

big endian

0x2001' 0004	0xA1
0x2001' 0005	0xB2
0x2001' 0006	0xC3
0x2001' 0007	0xD4

■ Alignment

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



Object File Sections

■ CODE

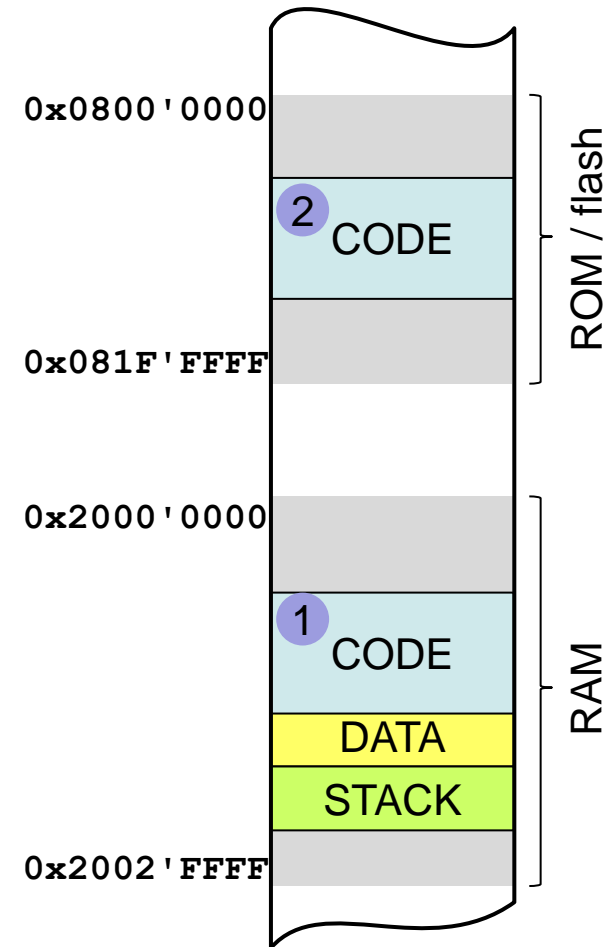
- Read-only → RAM or ROM
- Instructions (opcodes)
- Literals ¹⁾

■ 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

■ Assembly Program Structure

- AREA directive

	AREA	MyCode, CODE, READONLY	 Define code area to include your program
start	ENTRY		
	MOVS	R4, #12	
	ADDS	R3, R4, #5	
	B	start	
	AREA	MyData, DATA, READWRITE	 Define data area to store global variables, etc.
byte_var	DCB	0x1A, 0x00	
hw_var	DCW	0x2B3C	
word_var	DCD	0x4D5E6F70	
	AREA	STACK, NOINIT, READWRITE	 Define stack area to reserve space for stack
stack_mem	SPACE	0x00000400	

Memory Map / Object File Sections

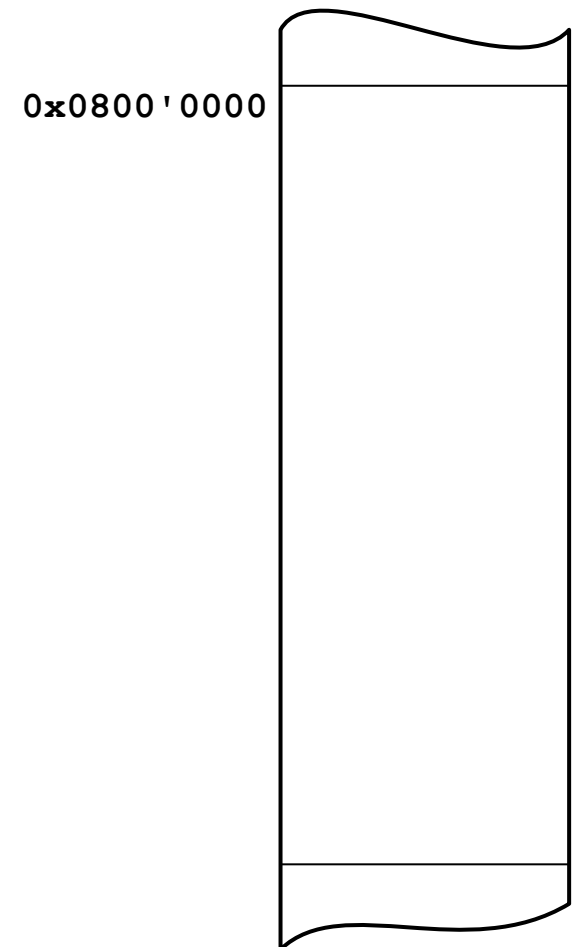
Assume

- 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

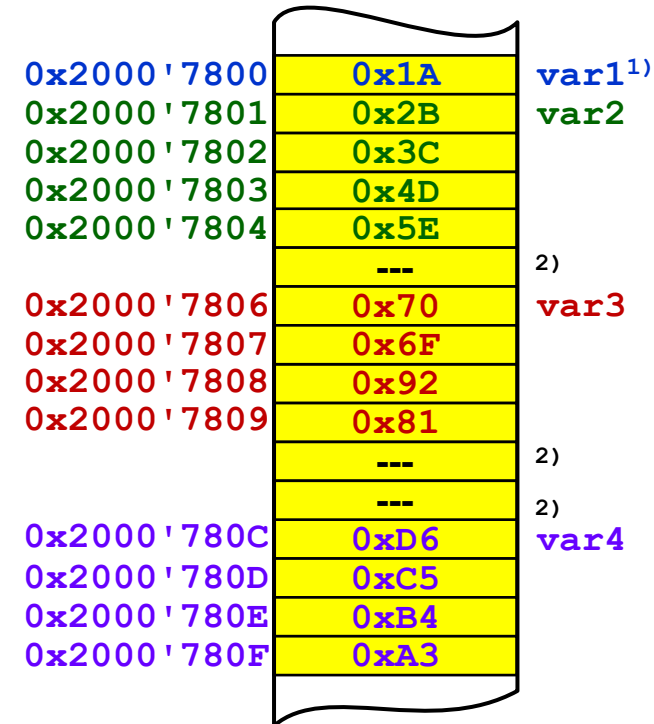
- Draw the memory map with the three sections
- For each section mark the first and the last address with its value
- How many memory cells (bytes) does each section contain?
- For each section: In which STM32F4 memory is it located?



■ 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
```



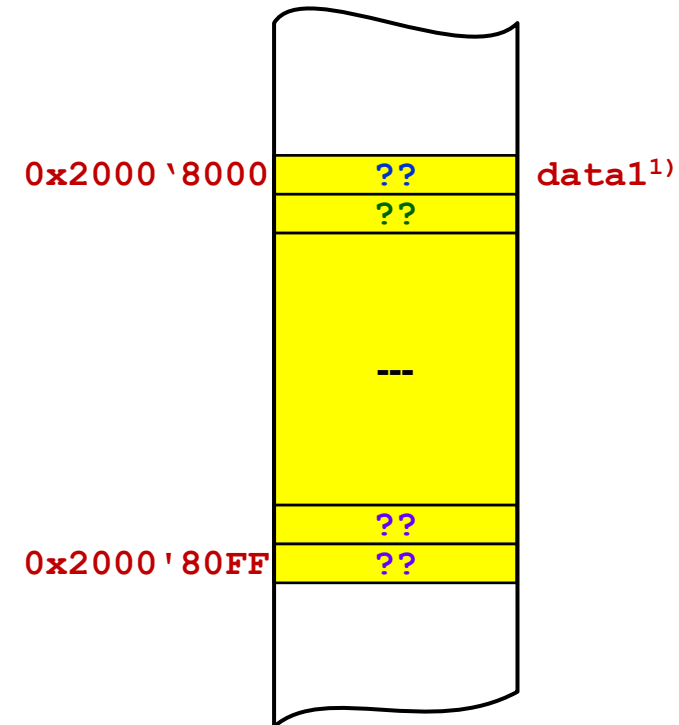
¹⁾ if we assume that example1 starts at 0x2000'7800

²⁾ Padding bytes introduced for alignment

■ Memory Allocation in Assembly

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

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



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

■ 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?

■ 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;
    ...
}
```

global variable with initialization

global variable with no init value specified in C source code

global constant

local variable

local constant

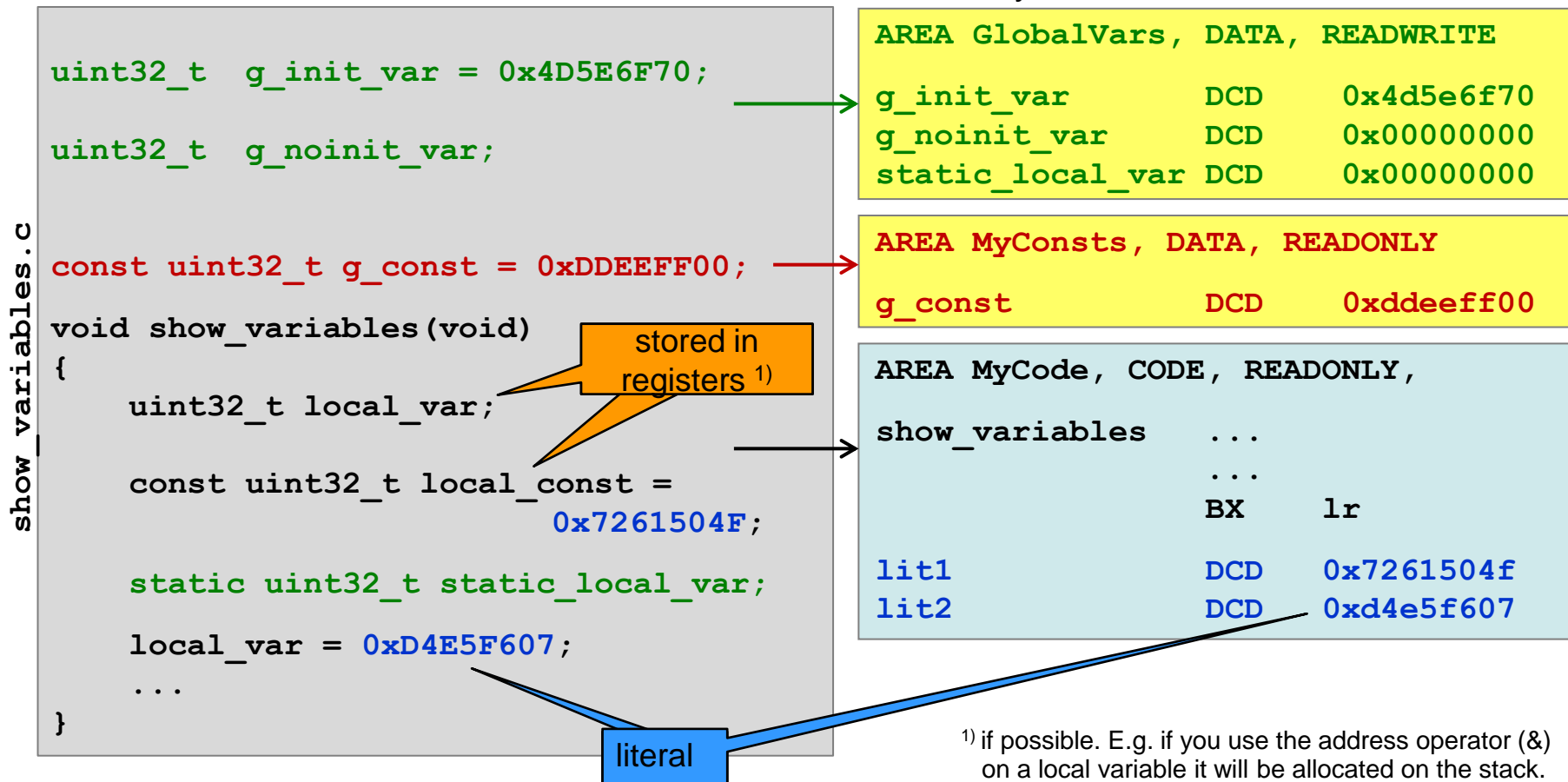
literal

local variable with qualifier static

literal

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

■ Code Example



■ 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