School of Engineering
InES Institute of Embedded Systems

Cortex-M Architecture

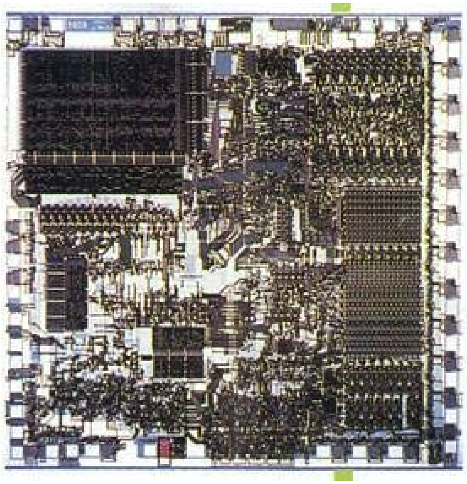
Computer Engineering 1

CT Team: A. Gieriet, J. Gruber, R. Gübeli, M. Meli, M. Rosenthal,
A. Rüst, J. Scheier, M. Thaler

Motivation

- **Intel**

1978

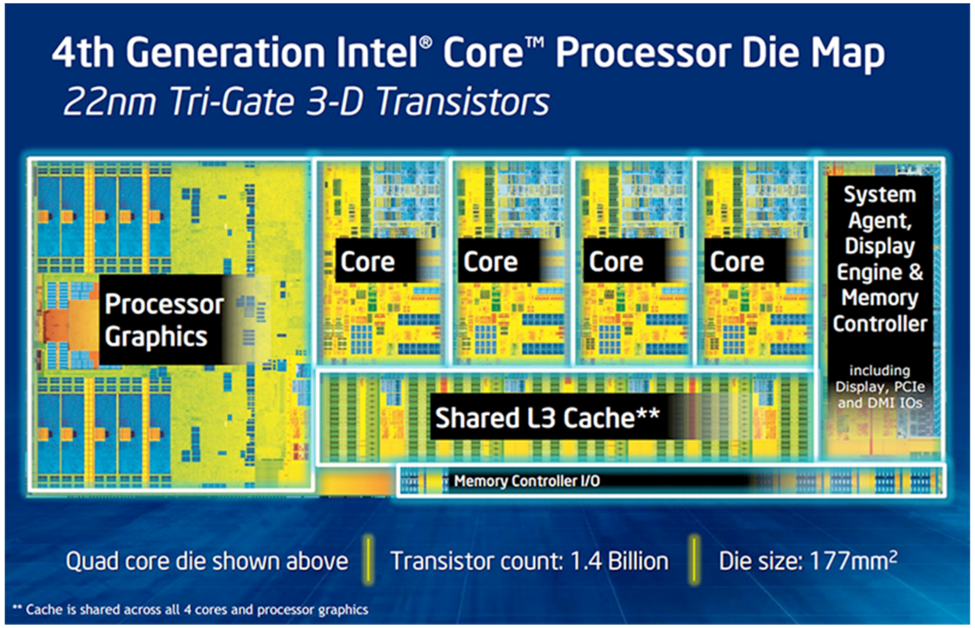


8086
33 mm^2
29k transistors
5 MHz

50'000 x
transistors

720 x
clock speed

2013


4th Generation Intel® Core™ Processor Die Map
22nm Tri-Gate 3-D Transistors
Processor Graphics
Core
Core
Core
Core
System Agent, Display Engine & Memory Controller
including Display, PCIe and DMI IOs
Shared L3 Cache**
Memory Controller I/O
Quad core die shown above
Transistor count: 1.4 Billion
Die size: 177mm²
** Cache is shared across all 4 cores and processor graphics

core i7 Haswell
177 mm^2
1400M transistors
3.6 GHz

Motivation

- **ARM (Cortex-M)**

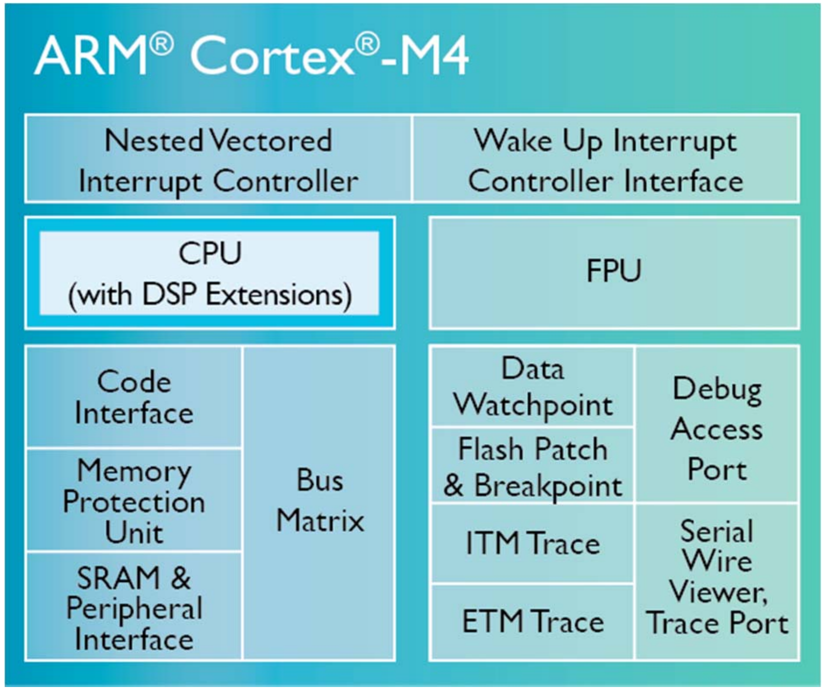
1985

ARM1
25k Transistors
5 MHz

1'400 x
transistors

36 x
clock speed

2014



ARM® Cortex®-M4

Nested Vectored Interrupt Controller

Wake Up Interrupt Controller Interface

CPU (with DSP Extensions)

FPU

Code Interface

Memory Protection Unit

SRAM & Peripheral Interface

Bus Matrix

Data Watchpoint

Flash Patch & Breakpoint

ITM Trace

ETM Trace

Debug Access Port

Serial Wire Viewer, Trace Port

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

Agenda

- **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

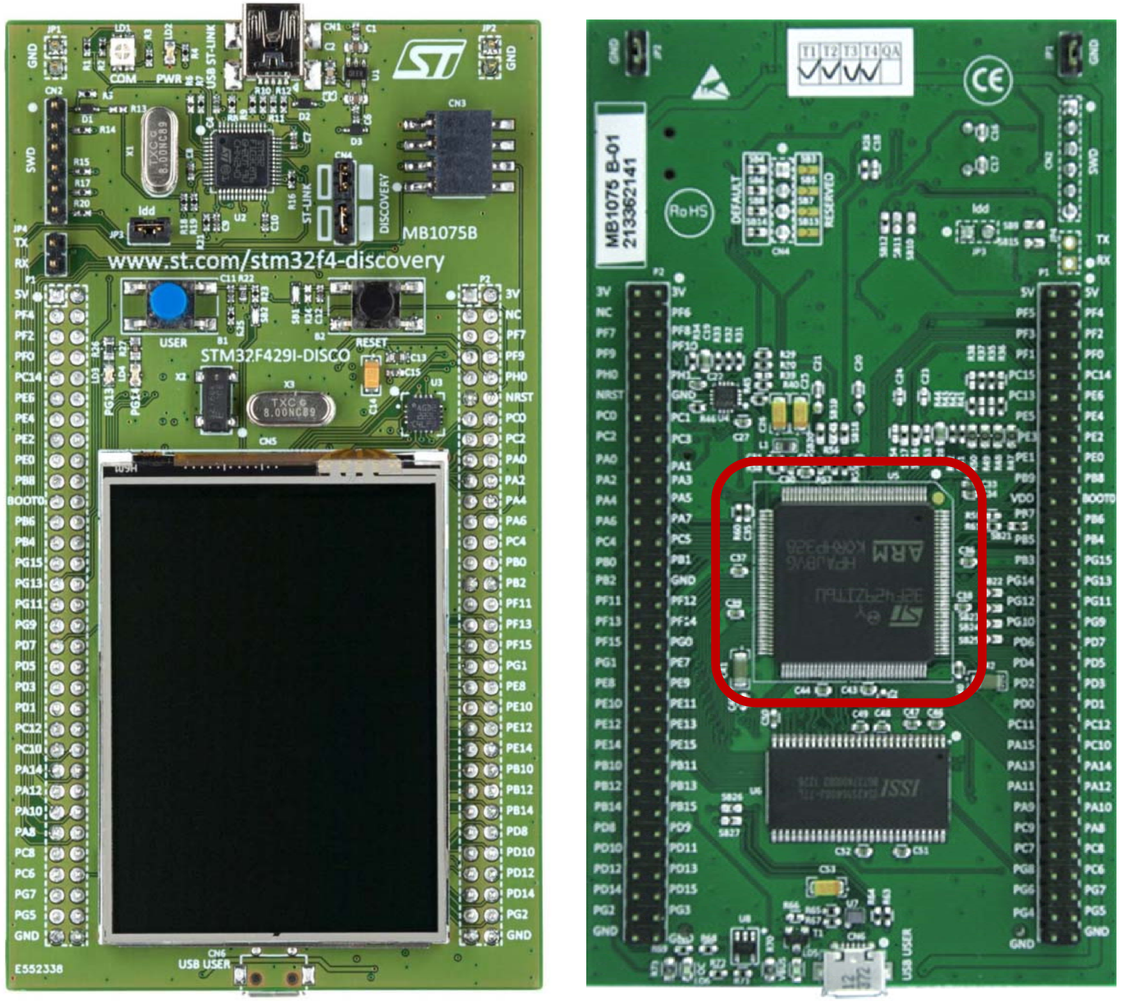
Learning Objectives

At the end of this lesson you will be able

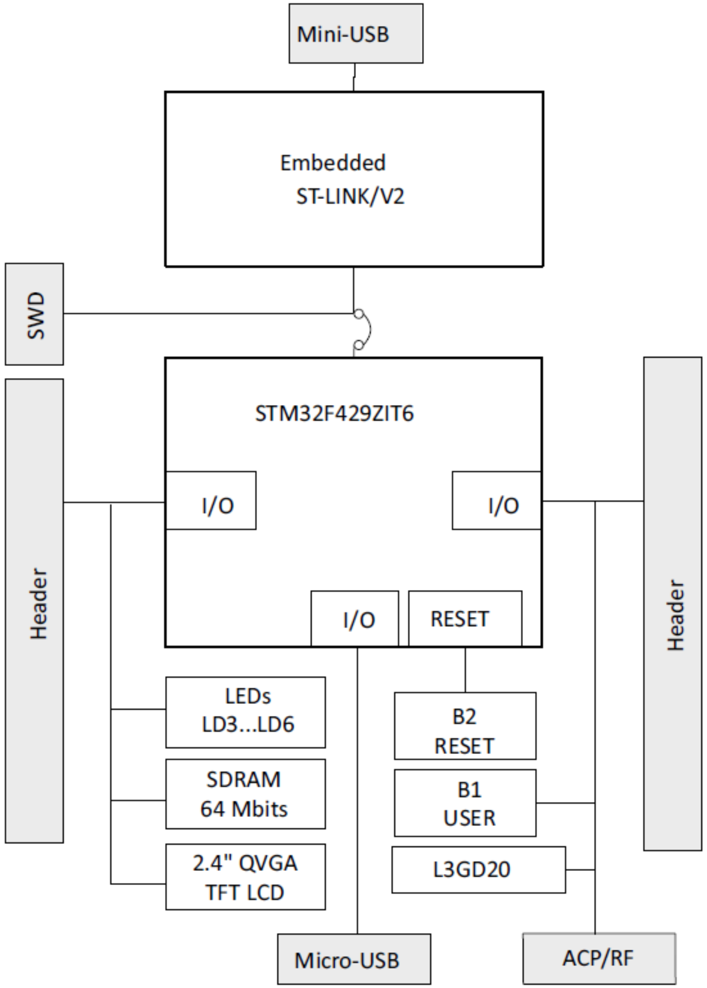
- to describe what an 'Instruction Set Architecture' is
- to outline the Cortex-M architecture and enumerate the main components and their functions
- to enumerate the instruction type categories
- to understand the structure of the Cortex-M instruction set
- to explain how a processor executes a program
- 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



Mini-USB

Embedded ST-LINK/V2

SWD

STM32F429ZIT6

I/O

I/O

I/O

RESET

Header

Header

LEDs LD3...LD6

SDRAM 64 Mbits

2.4" QVGA TFT LCD

B2 RESET

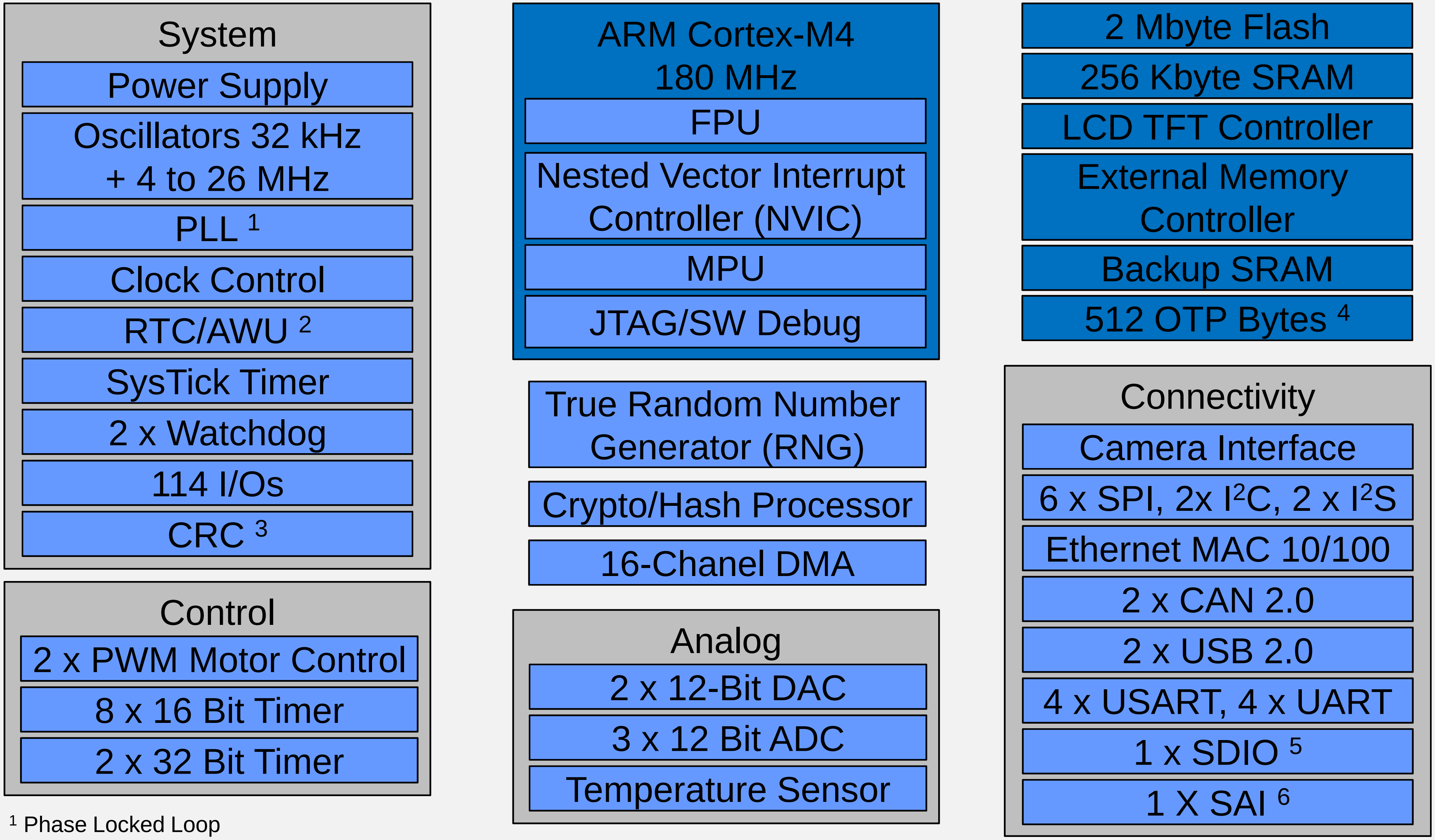
B1 USER

L3GD20

Micro-USB

ACP/RF

Hardware Platform: STM32F429I



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

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

2 Mbyte Flash

256 Kbyte SRAM

LCD TFT Controller

External Memory Controller

Backup SRAM

512 OTP Bytes [4]

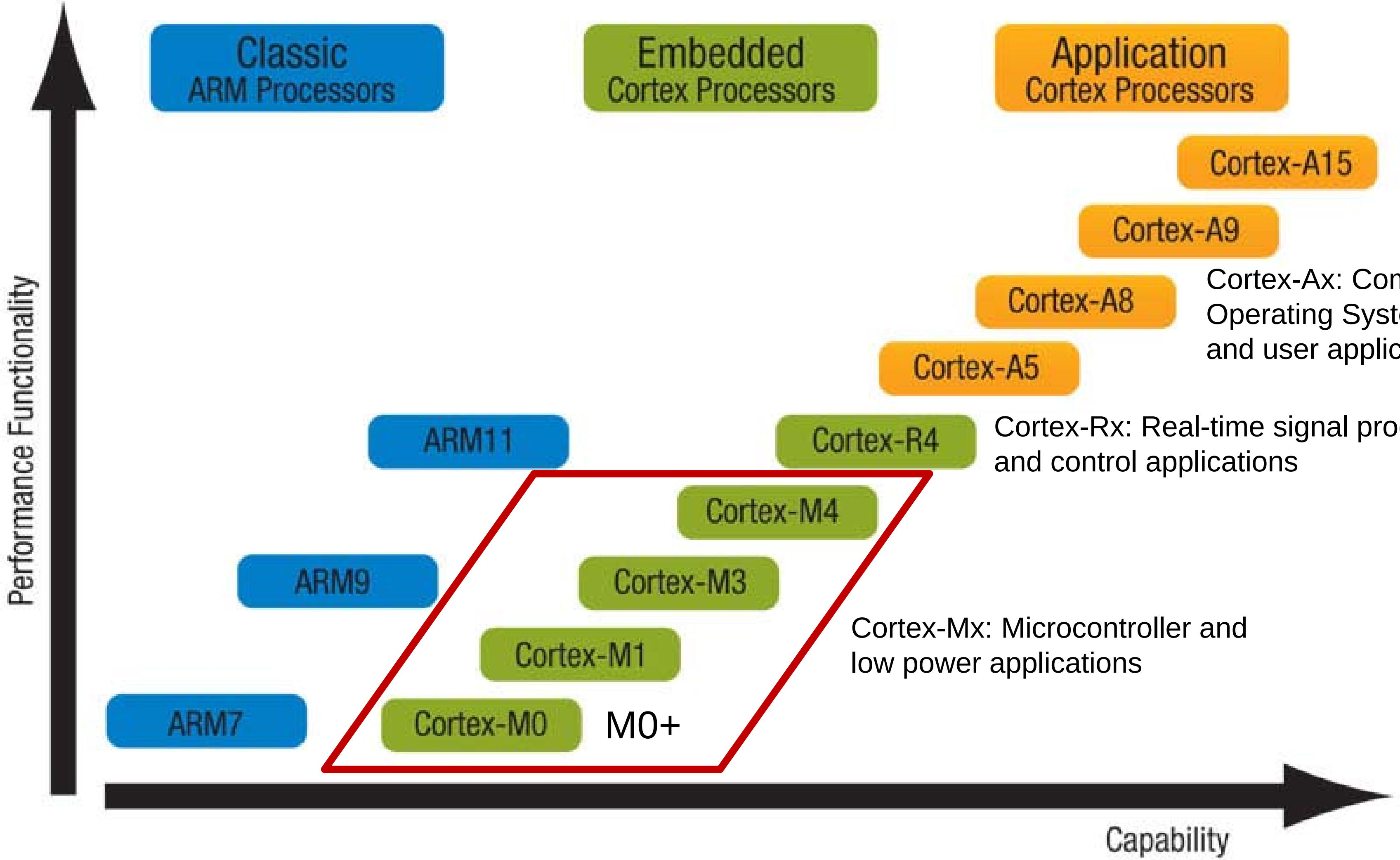
Connectivity
- Camera Interface
- 6 x SPI, 2x I^2C, 2 x I^2S
- Ethernet MAC 10/100
- 2 x CAN 2.0
- 2 x USB 2.0
- 4 x USART, 4 x UART
- 1 x SDIO [5]
- 1 X SAI [6]

[1] Phase Locked Loop
[2] Real Time Clock / Auto-Wakeup
[3] Cyclic Redundancy Check
[4] One Time Programmable
[5] Secure Digital I/O Interface
[6] Serial Audio Interface

based on source of STMicroelectronics

Hardware Platform

ARM Processor Portfolio



Cortex-Ax: Complex Operating Systems and user applications

Cortex-Rx: Real-time signal processing and control applications

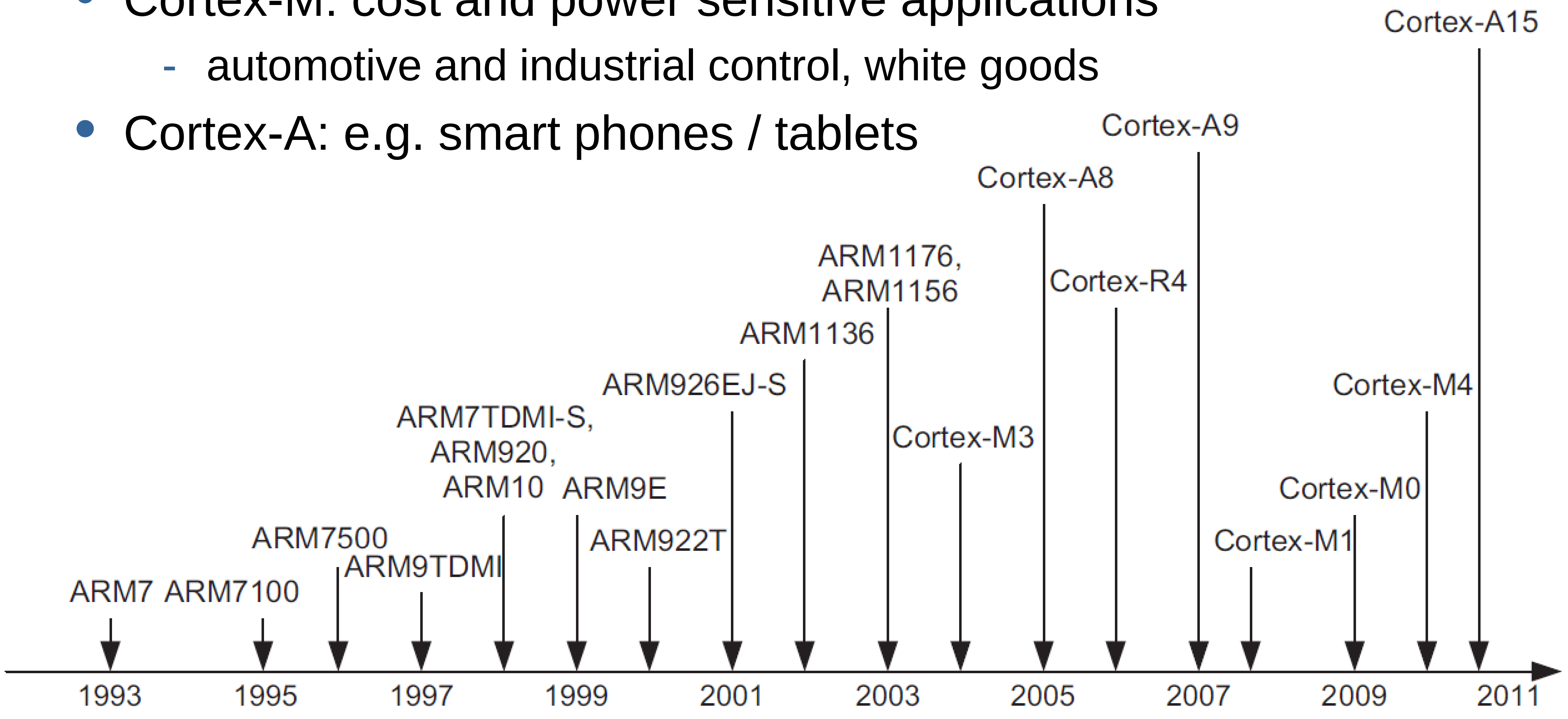
Cortex-Mx: Microcontroller and low power applications

source: www.rtcmagazine.com

Hardware Platform

Timeline ARM Processors

- Cortex-M: cost and power sensitive applications
  - automotive and industrial control, white goods
- Cortex-A: e.g. smart phones / tablets

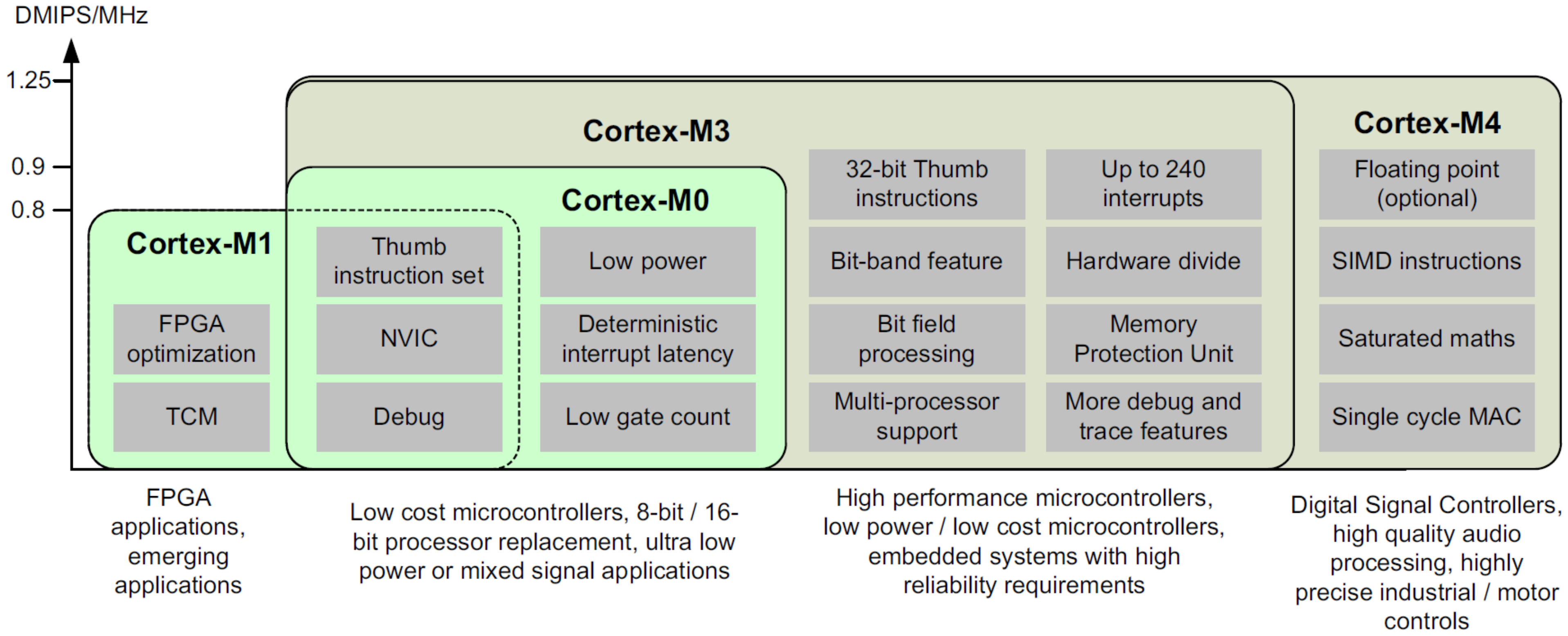


source: Joseph Yiu, The definitive Guide to the Cortex-M0

Hardware Platform

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



DMIPS/MHz
1.25
0.9
0.8
Cortex-M3
Cortex-M4
Cortex-M0
Cortex-M1
32-bit Thumb instructions
Up to 240 interrupts
Floating point (optional)
Thumb instruction set
Low power
Bit-band feature
Hardware divide
SIMD instructions
FPGA optimization
NVIC
Deterministic interrupt latency
Bit field processing
Memory Protection Unit
Saturated maths
TCM
Debug
Low gate count
Multi-processor support
More debug and trace features
Single cycle MAC
FPGA applications, emerging applications
Low cost microcontrollers, 8-bit / 16-bit processor replacement, ultra low power or mixed signal applications
High performance microcontrollers, low power / low cost microcontrollers, embedded systems with high reliability requirements
Digital Signal Controllers, high quality audio processing, highly precise industrial / motor controls

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

CPU Model

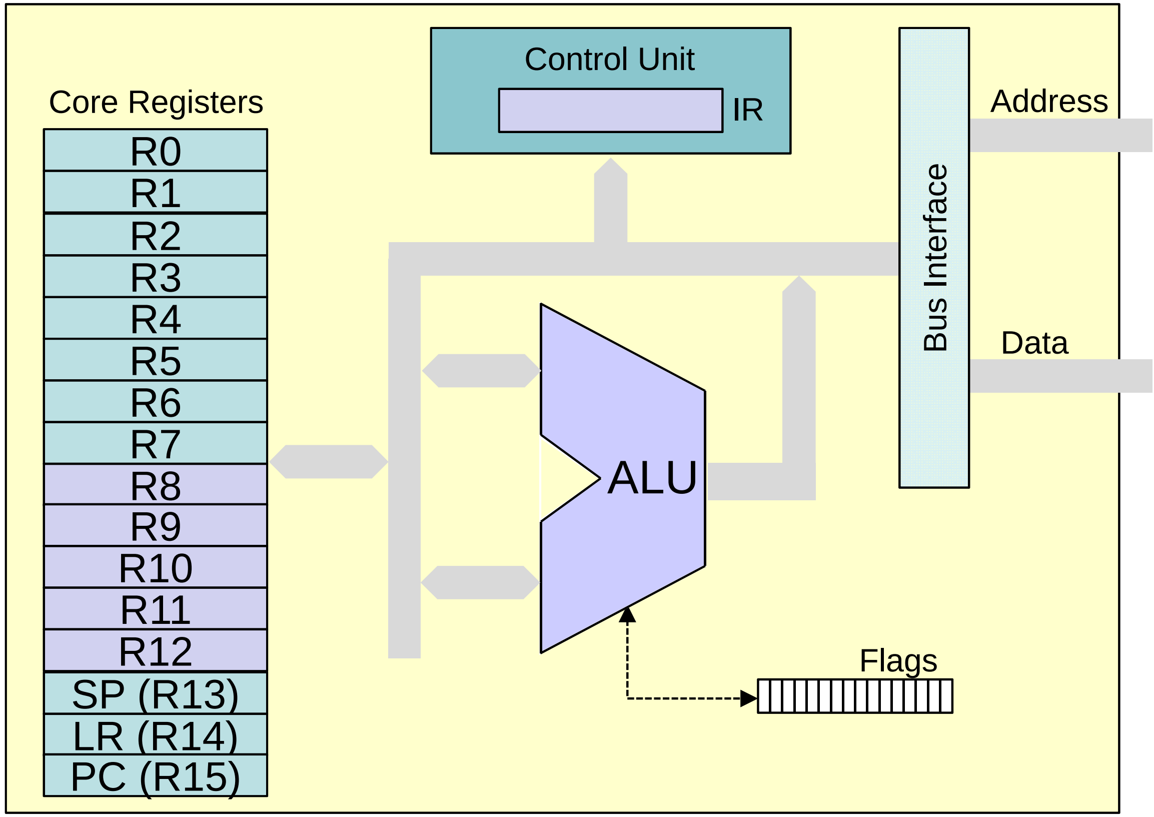
- **Instruction Set Architecture (ISA)** → **CT1**

  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

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

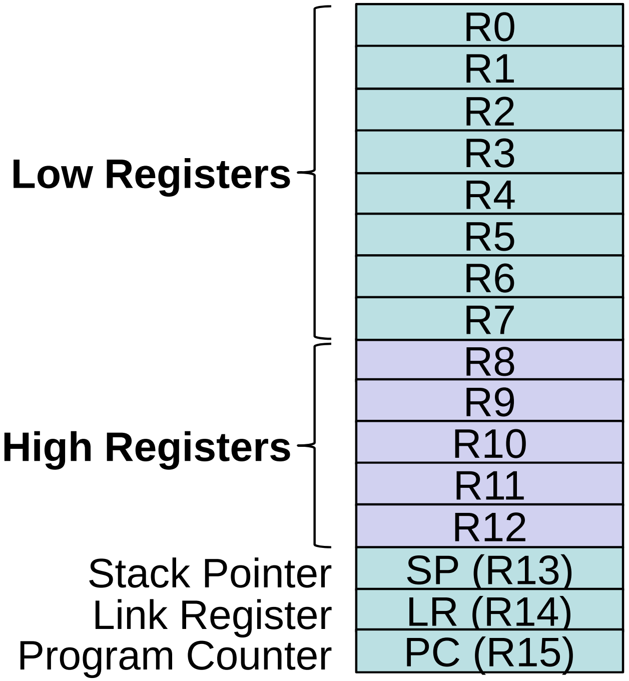


simple CPU model based on the Programmers' Model of the ARM Cortex-Mx CPUs [1)]

[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 **<u>next</u>** instruction
  - Stack Pointer (R13)
    - Last-In First-Out for temporary data storage
  - Link Register (R14)
    - Return from procedures

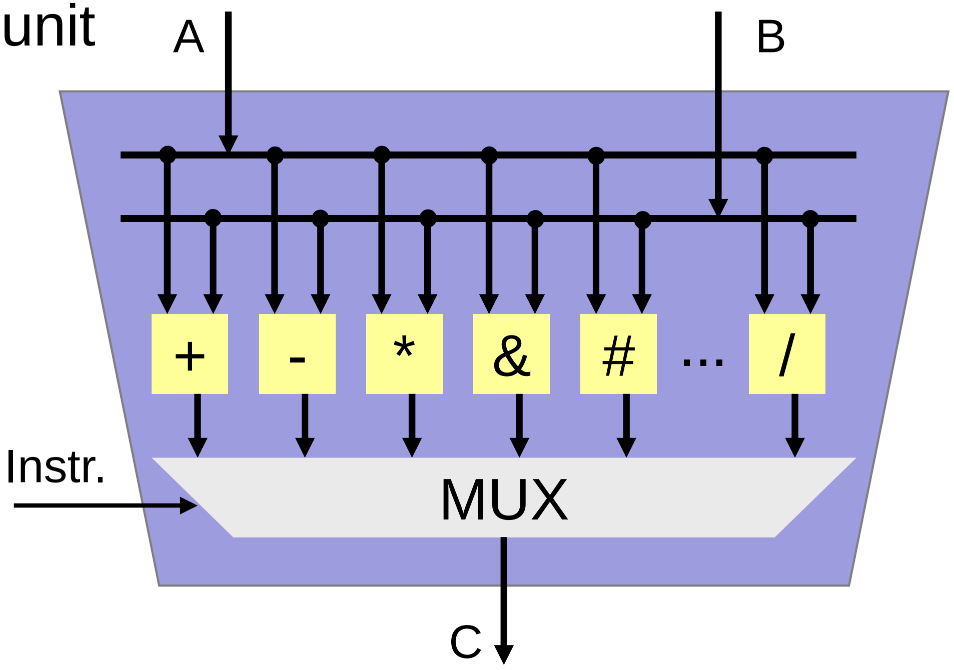


Low Registers	R0 R1 R2 R3 R4 R5 R6 R7
High Registers	R8 R9 R10 R11 R12
Stack Pointer	SP (R13)
Link Register	LR (R14)
Program Counter	PC (R15)

CPU Model

ALU – Arithmetic Logic Unit

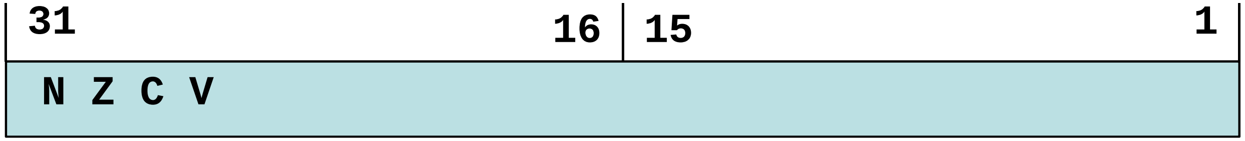
- 32-bit wide data processing unit
  - inputs A and B
  - result C
- integer arithmetic
  - addition / subtraction
  - multiplication / division
  - sign extension
- logic operations
  - AND, NOT, OR, XOR
- shift/rotate
  - left / right



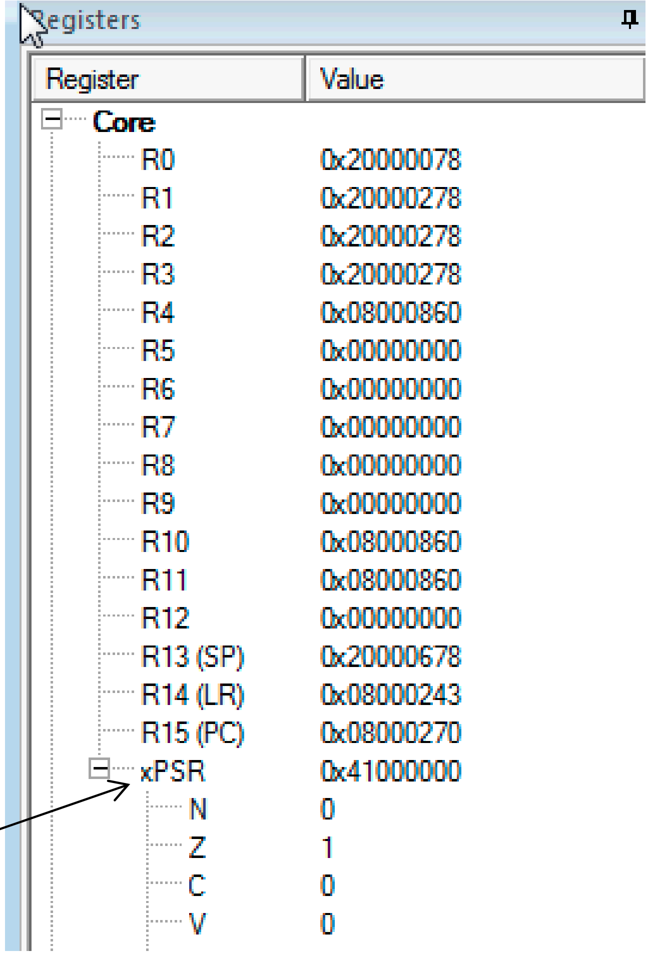
CPU Model

APSR[1] or Flag-Register

- Bits set by CPU based on results in ALU


31
16
15
1
N Z C V

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


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

As seen in the IDE

[1] APSR: Application Processor Status Register

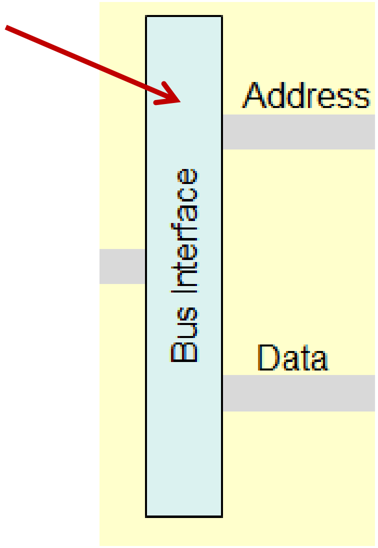
CPU Model

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

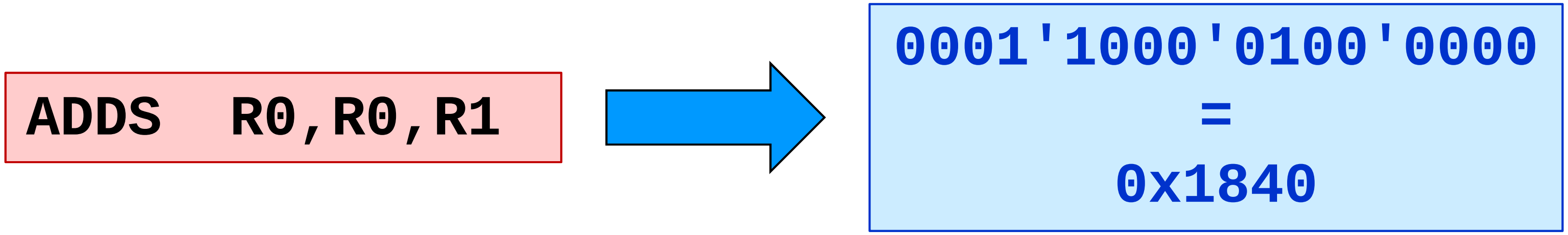
Bus Interface

- Interface between internal CPU bus and external system-bus
  - contains registers to store addresses



Instruction Set

- **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

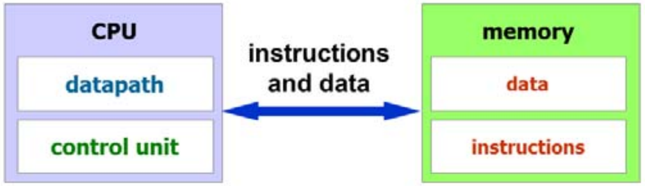
Assembly Program

- 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**

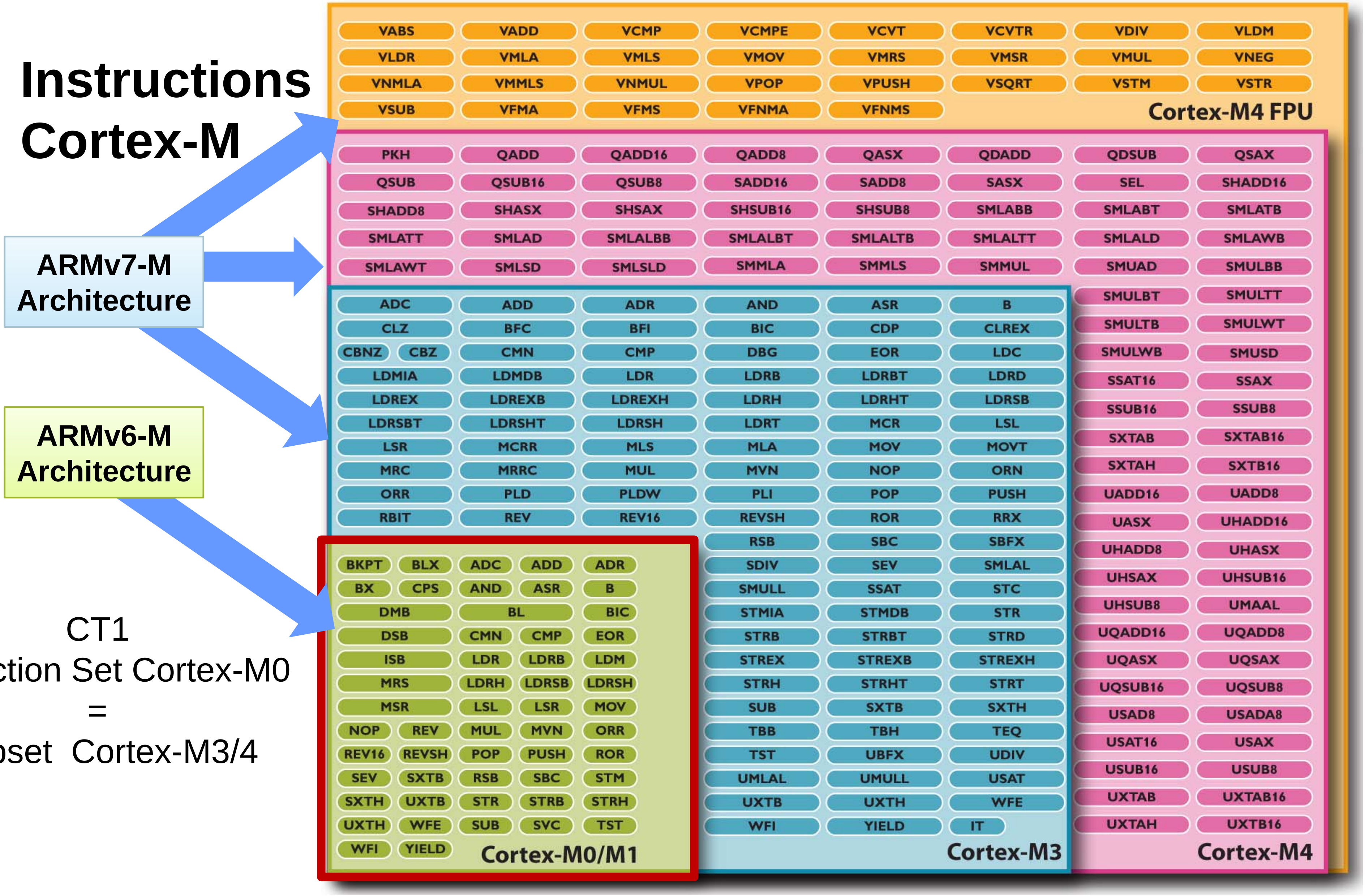


  - **Data transfer**
    - Copy content of one register to another register
    - Load registers with data from memory
    - Store register contents into memory
  - **Data processing**
    - Arithmetic operations → + - * / ...
    - Logic operations → AND, OR, ...
    - Shift / rotate operations
  - **Control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

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

Instruction Set

- **Instructions Cortex-M**



Cortex-M4 FPU

VABS	VADD	VCMP	VCMPE	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

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
						SSAT16	SSAX
						SSUB16	SSUB8
						SXTAB	SXTAB16
						SXTAH	SXTB16
						UADD16	UADD8
						UASX	UHADD16
						UHADD8	UHASX
						UHSAX	UHSUB16
						UHSUB8	UMAAL
						UQADD16	UQADD8
						UQASX	UQSAX
						UQSUB16	UQSUB8
						USAD8	USADA8
						USAT16	USAX
						USUB16	USUB8
						UXTAB	UXTAB16
						UXTAH	UXTB16

Cortex-M3

ADC	ADD	ADR	AND	ASR	B
CLZ	BFC	BFI	BIC	CDP	CLREX
CBNZ CBZ	CMN	CMP	DBG	EOR	LDC
LDMIA	LDMDB	LDR	LDRB	LDRBT	LDRD
LDREX	LDREXB	LDREXH	LDRH	LDRHT	LDRSB
LDRSBT	LDRSHT	LDRSH	LDRT	MCR	LSL
LSR	MCRR	MLS	MLA	MOV	MOVT
MRC	MRRC	MUL	MVN	NOP	ORN
ORR	PLD	PLDW	PLI	POP	PUSH
RBIT	REV	REV16	REVSH	ROR	RRX
			RSB	SBC	SBFX
			SDIV	SEV	SMLAL
			SMULL	SSAT	STC
			STMIA	STMDB	STR
			STRB	STRBT	STRD
			STREX	STREXB	STREXH
			STRH	STRHT	STRT
			SUB	SXTB	SXTH
			TBB	TBH	TEQ
			TST	UBFX	UDIV
			UMLAL	UMULL	USAT
			UXTB	UXTH	WFE
			WFI	YIELD	IT

Cortex-M0/M1

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			

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

source: ARM

Instruction Set

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

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			Rdn			data-processing register
0	1	0	0	0	1	opcode		DN	Rm				Rdn			special data processing
0	1	0	0	0	1	1	1	L	Rm				0	0	0	branch/exchange
0	1	0	0	1	Rd			PC-relative imm8								load from literal pool
0	1	0	1	opcode			Rm			Rn			Rd			load/store register offset
0	1	1	B	L	imm5					Rn			Rd			load/store word/byte imm. offset
1	0	0	0	L	imm5					Rn			Rd			load/store halfword imm. offset
1	0	0	1	L	Rd			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

Instruction Set

Cortex-M0 Example Program

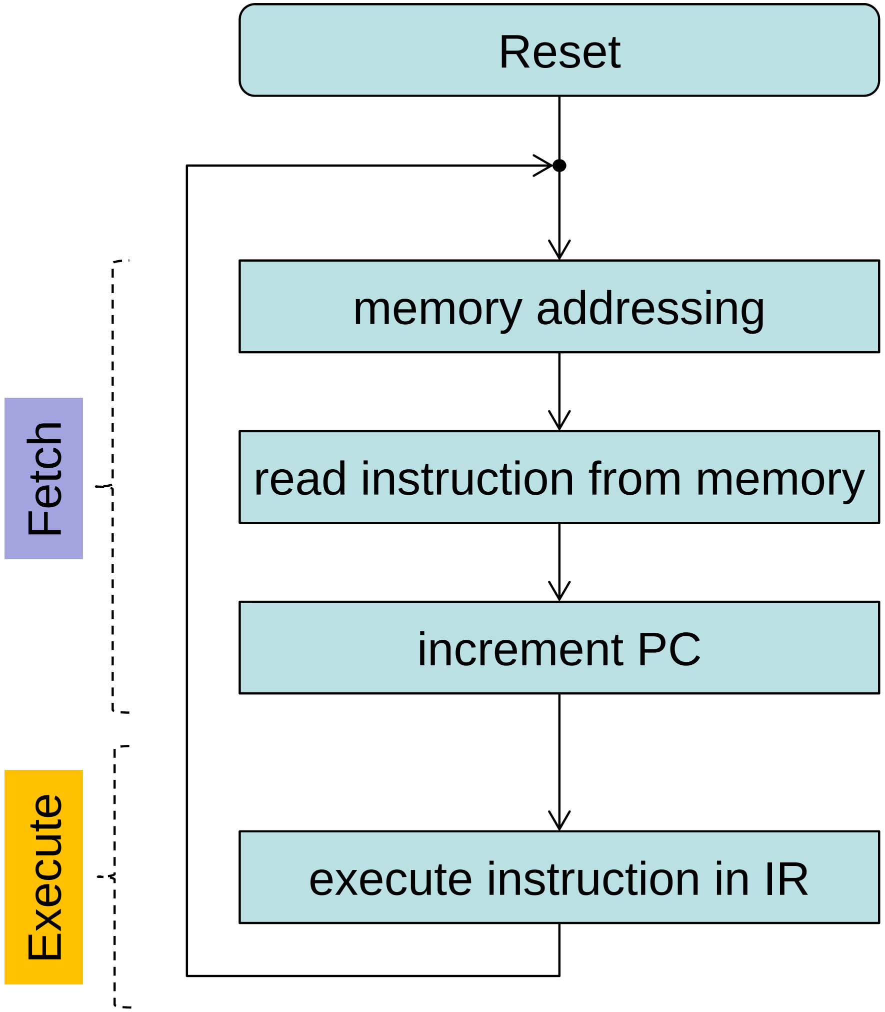
- Assembler converts each Assembly instruction to 16-bit opcode
- Addresses in steps of 2
  - Reason: 16-bit opcode uses two bytes
- Constant **0x00002000** at the end

Memory address *Opcode*

```
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
0000000A 00002000
```

Program Execution

- **Sequence**



Set Program Counter (PC) to 0x0000'0000

Place content of PC on address bus

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

PC = PC + 2

Execute instruction in instruction register (IR)

Program Execution

- **Load program into memory**

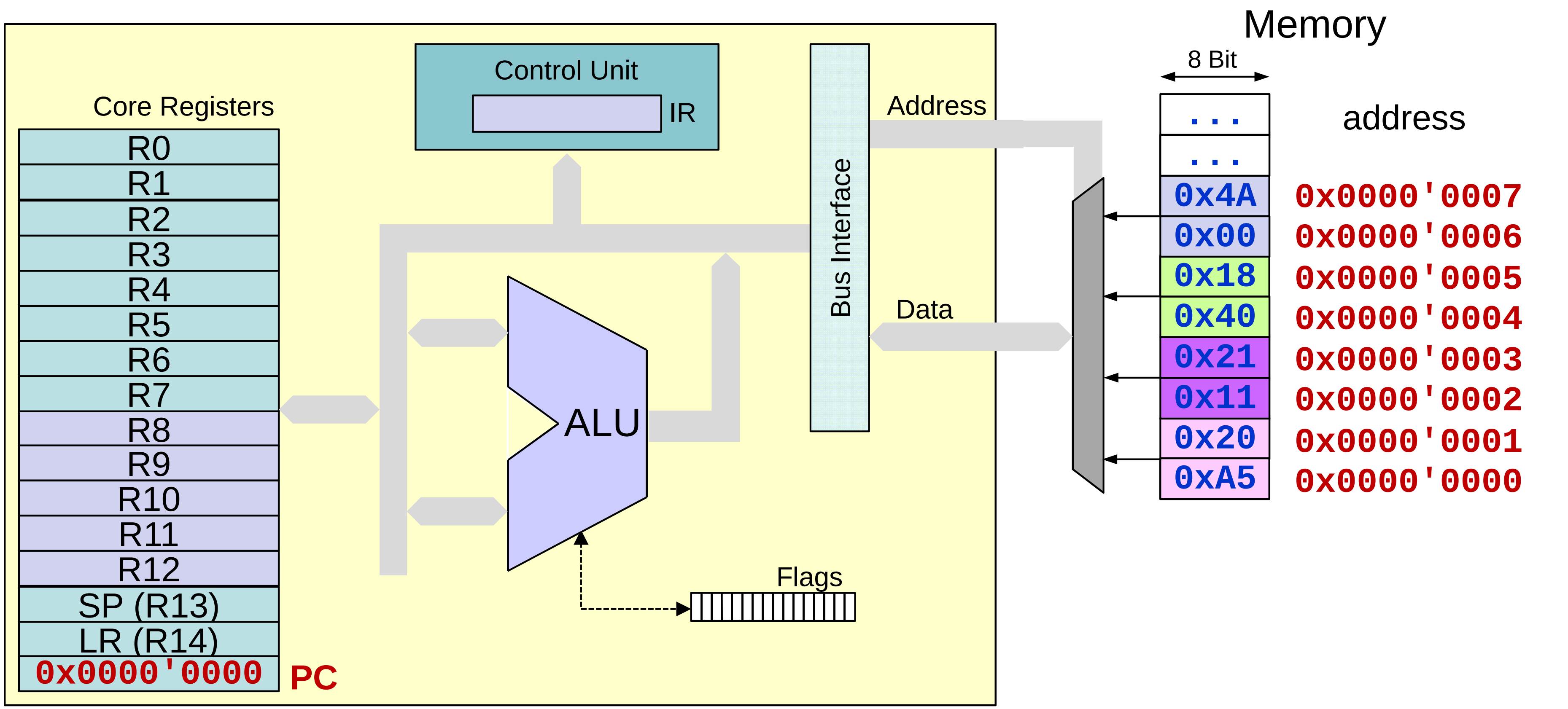
```
00000000 20A5    demoprg MOVS   R0,#0xA5
00000002 2111            MOVS   R1,#0x11
00000004 1840            ADDS   R0,R0,R1

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

0000000A 00002000
```

Memory

address	8 Bit
	...
	...
0x0000'0007	0x4A
0x0000'0006	0x00
0x0000'0005	0x18
0x0000'0004	0x40
0x0000'0003	0x21
0x0000'0002	0x11
0x0000'0001	0x20
0x0000'0000	0xA5

Program Execution

- **"Reset": 0x0000'0000 → PC**



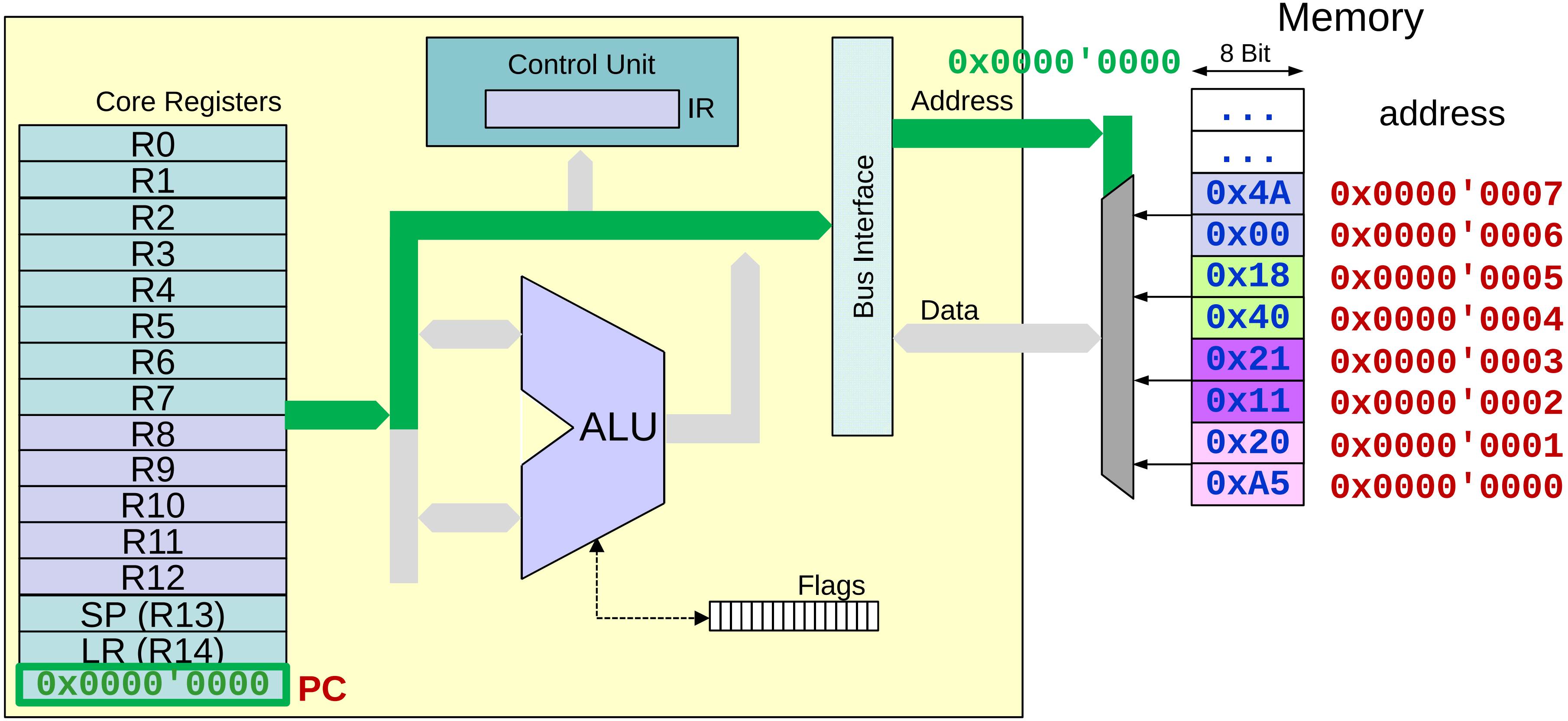
PC →

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

Program Execution

- **PC → Address Bus**

Fetch



...	
...	
0x4A	0x0000'0007
0x00	0x0000'0006
0x18	0x0000'0005
0x40	0x0000'0004
0x21	0x0000'0003
0x11	0x0000'0002
0x20	0x0000'0001
0xA5	0x0000'0000

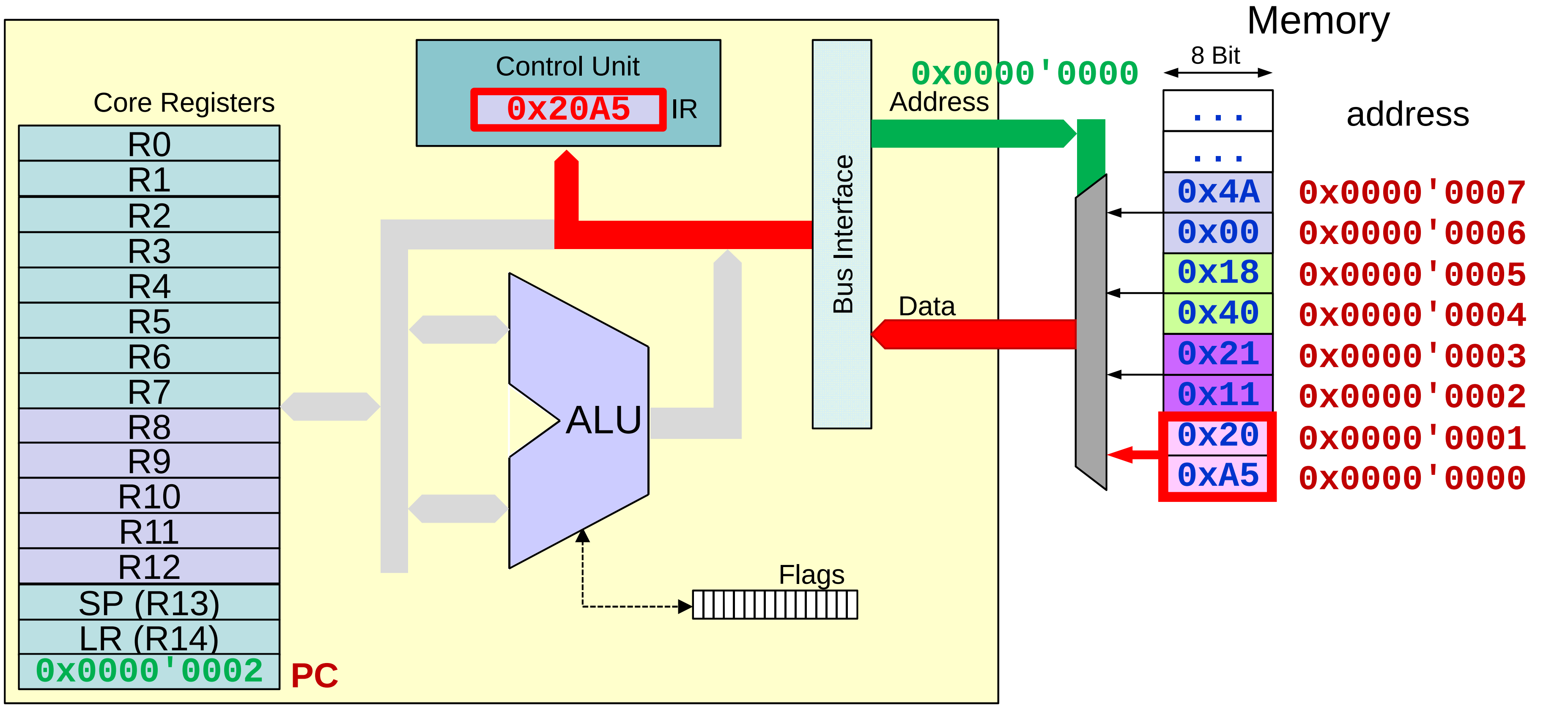
PC →

```
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

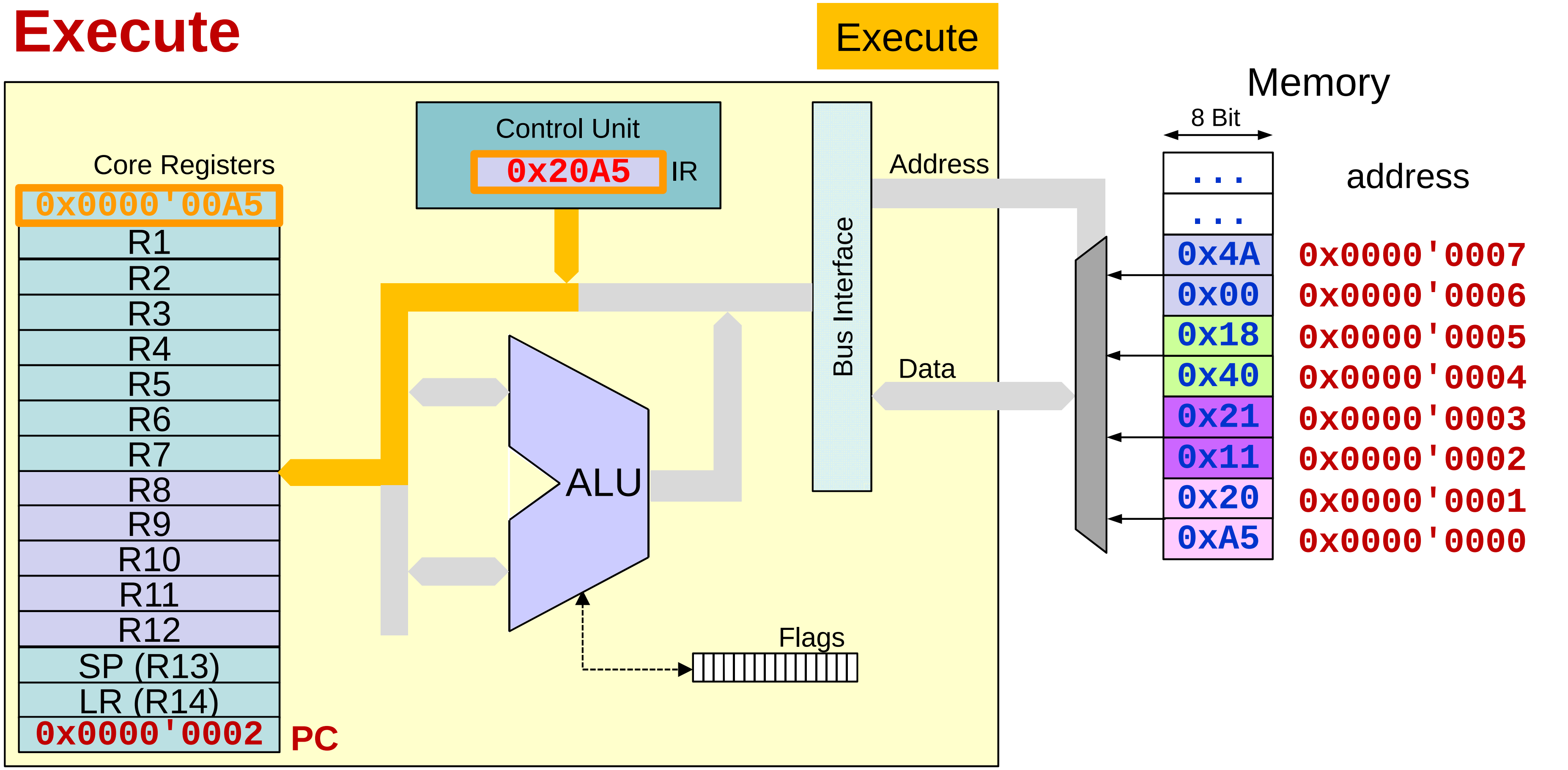


PC

```
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



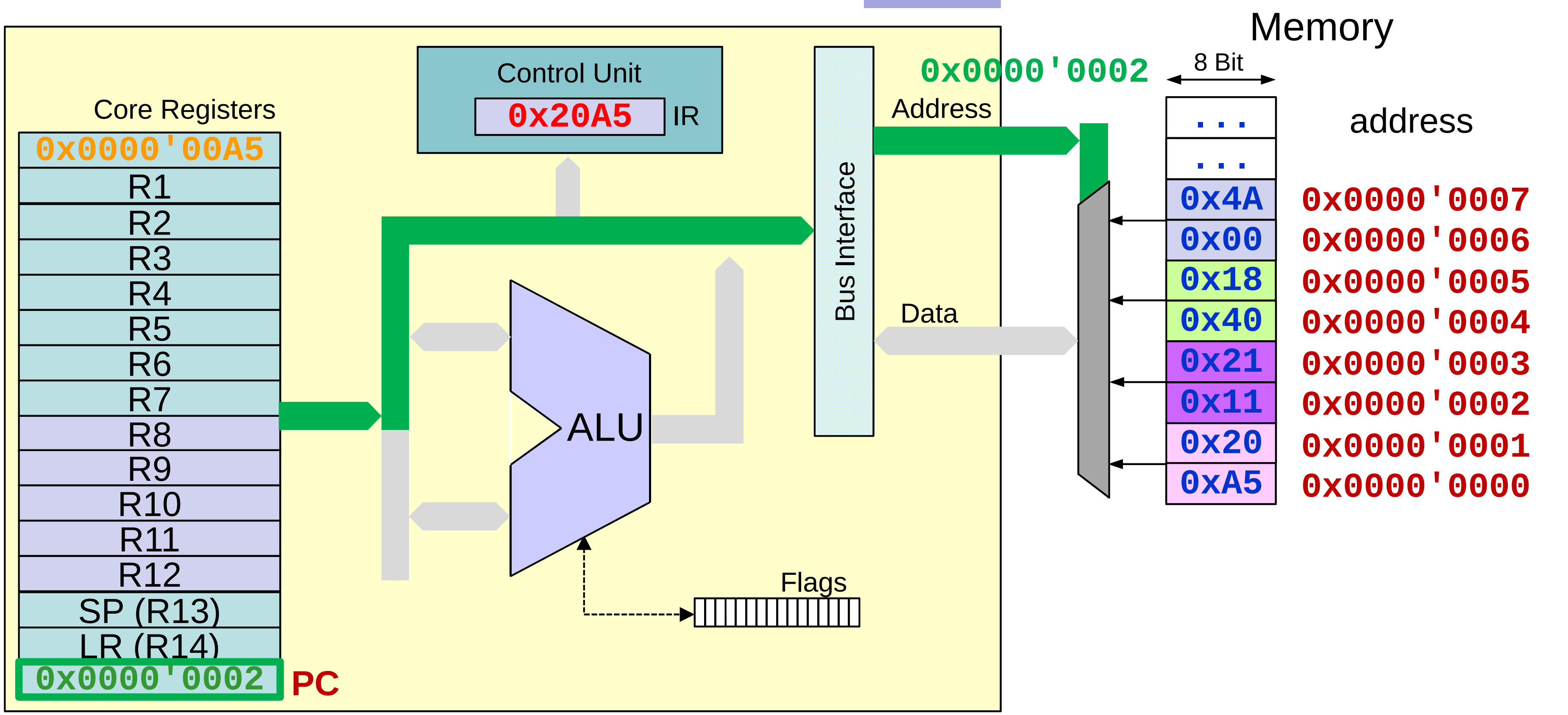
PC

```
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**

Fetch



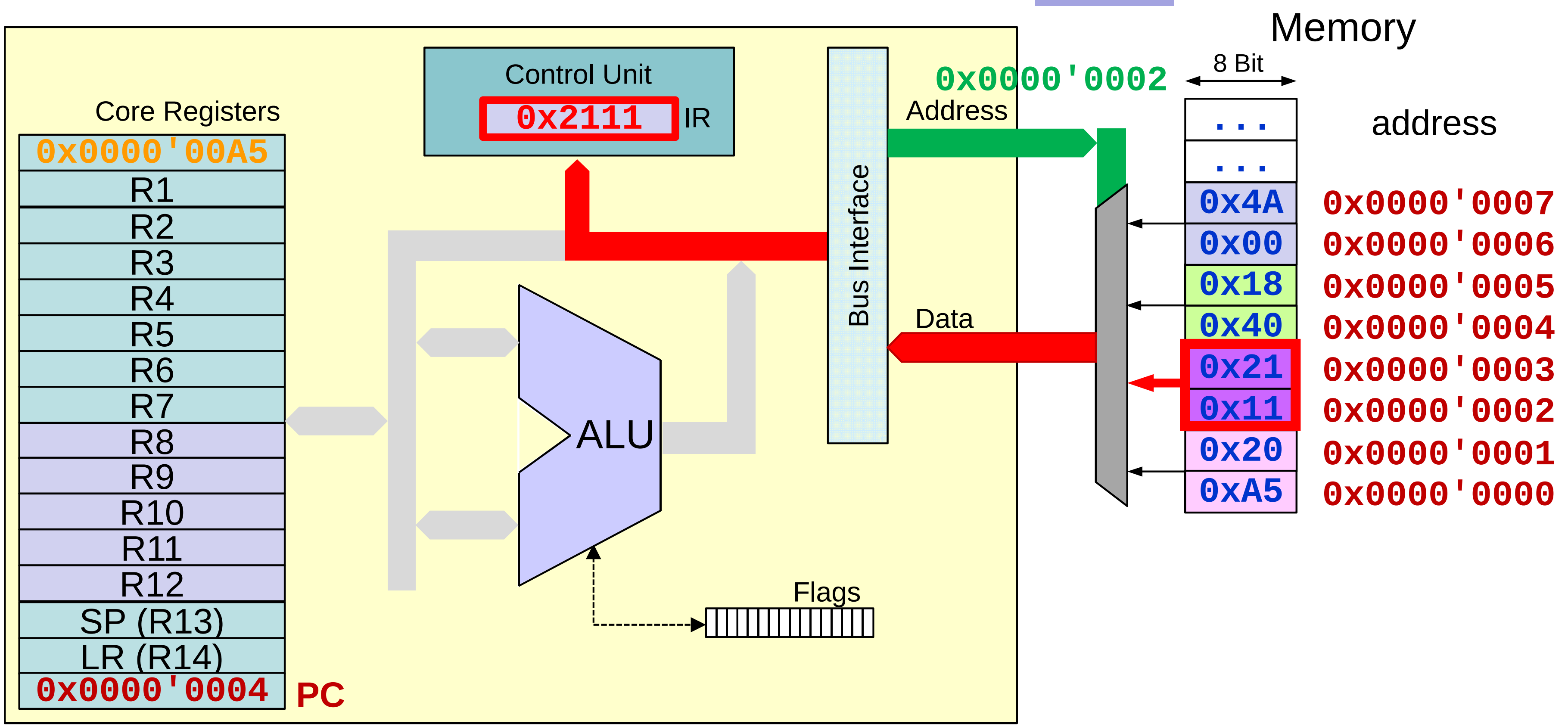
PC →

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

Program Execution

Read Instruction / Increment PC

Fetch



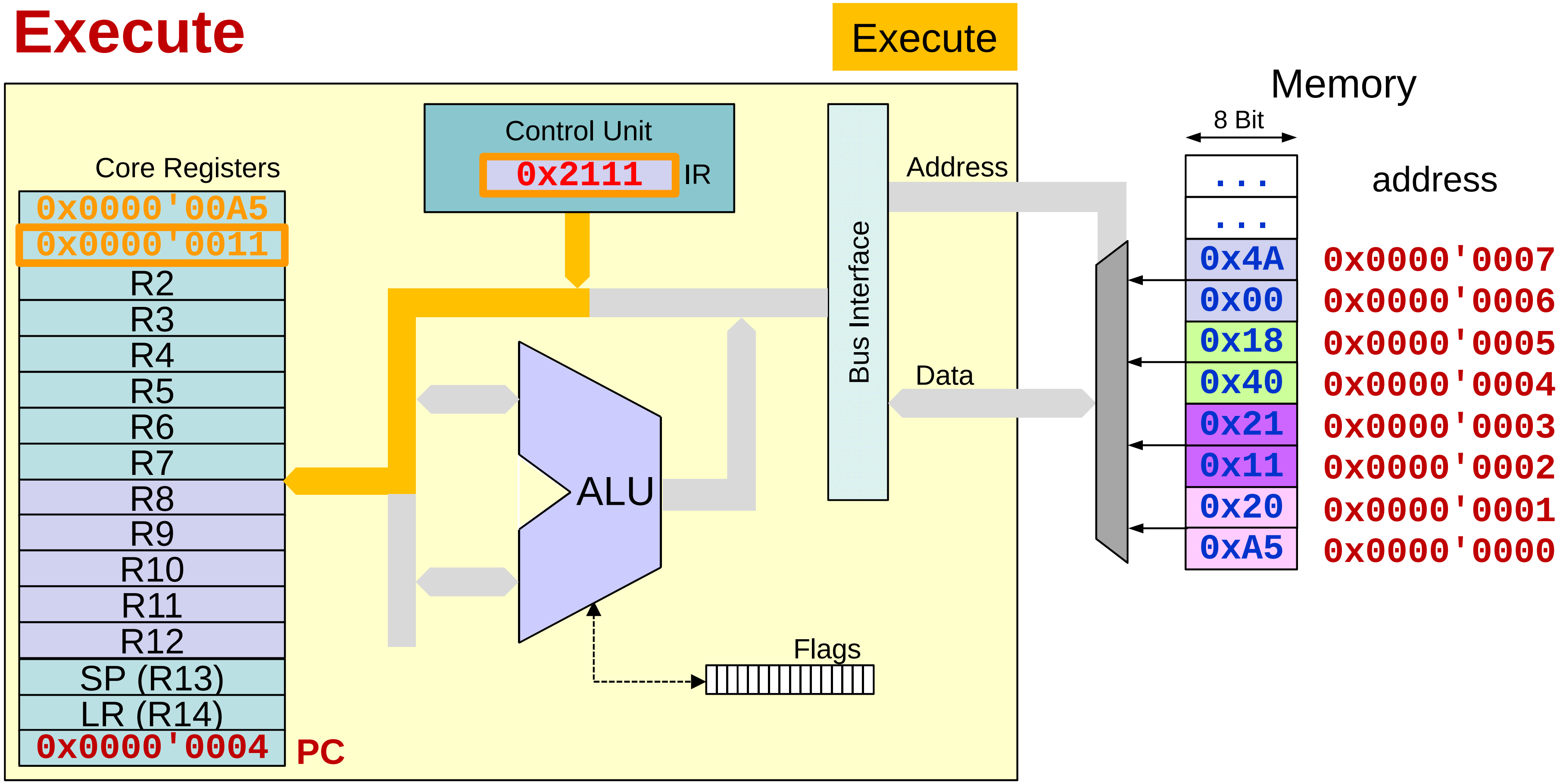
8 Bit	address
. . .	
. . .	
0x4A	0x0000'0007
0x00	0x0000'0006
0x18	0x0000'0005
0x40	0x0000'0004
0x21	0x0000'0003
0x11	0x0000'0002
0x20	0x0000'0001
0xA5	0x0000'0000

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

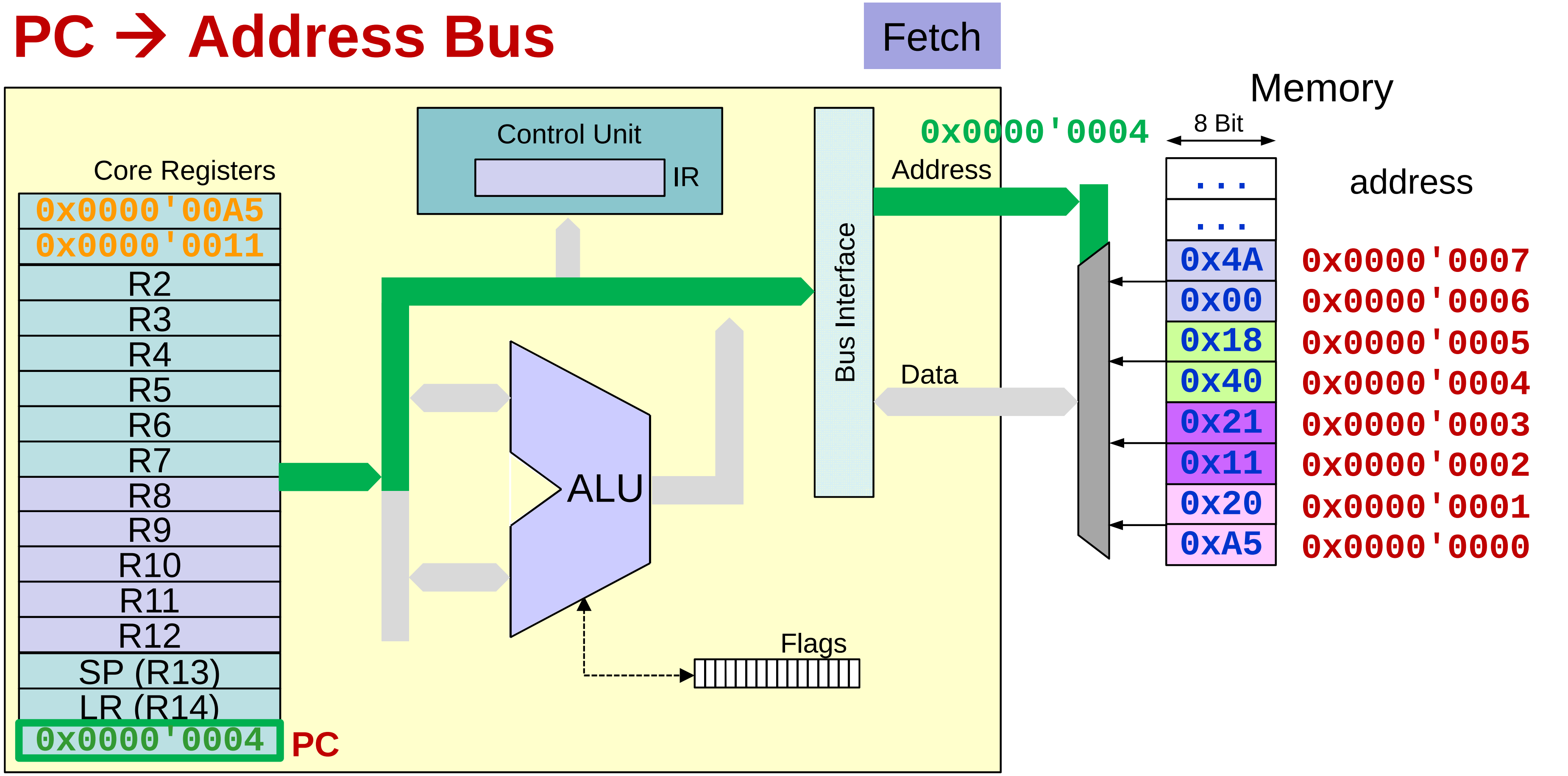
- **Execute**



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

Program Execution

PC → Address Bus



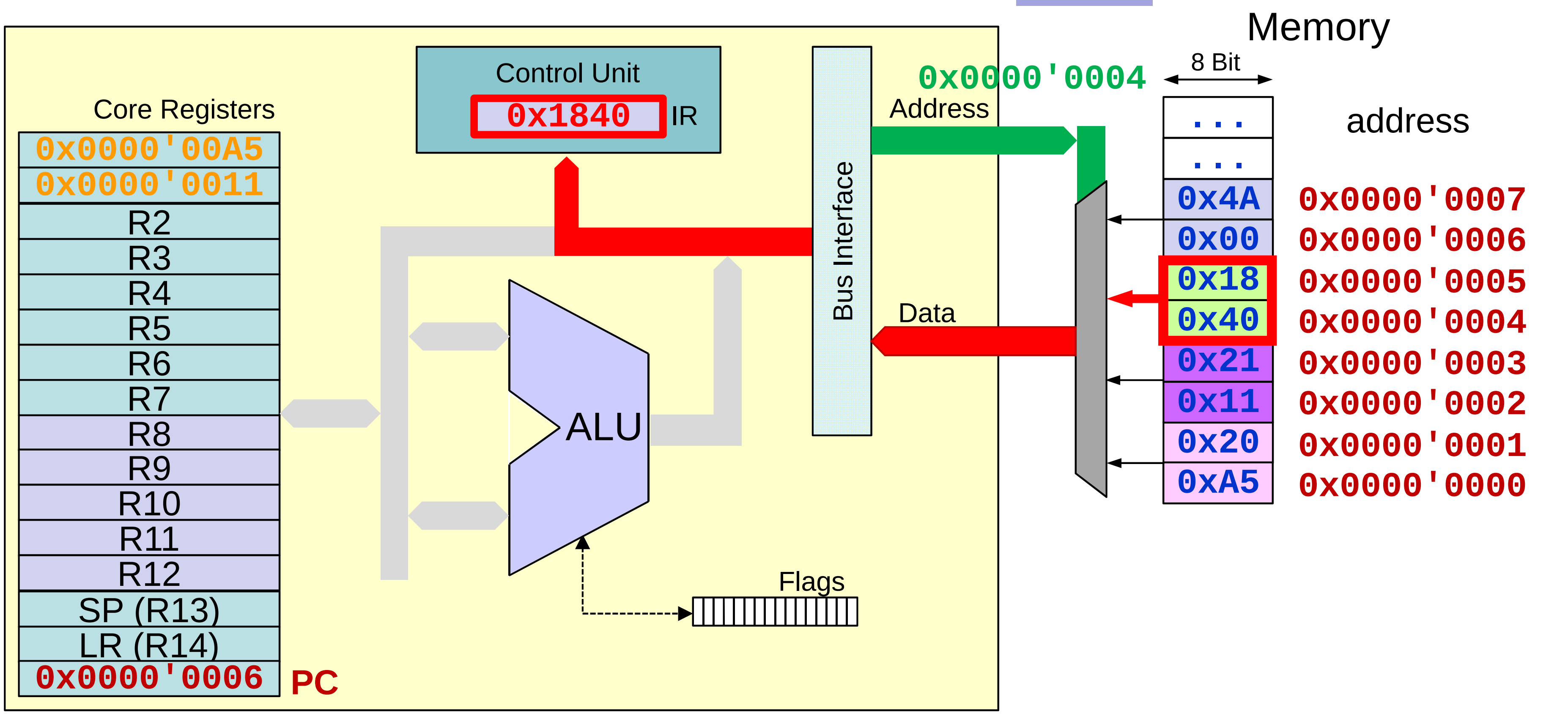
PC →

```
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

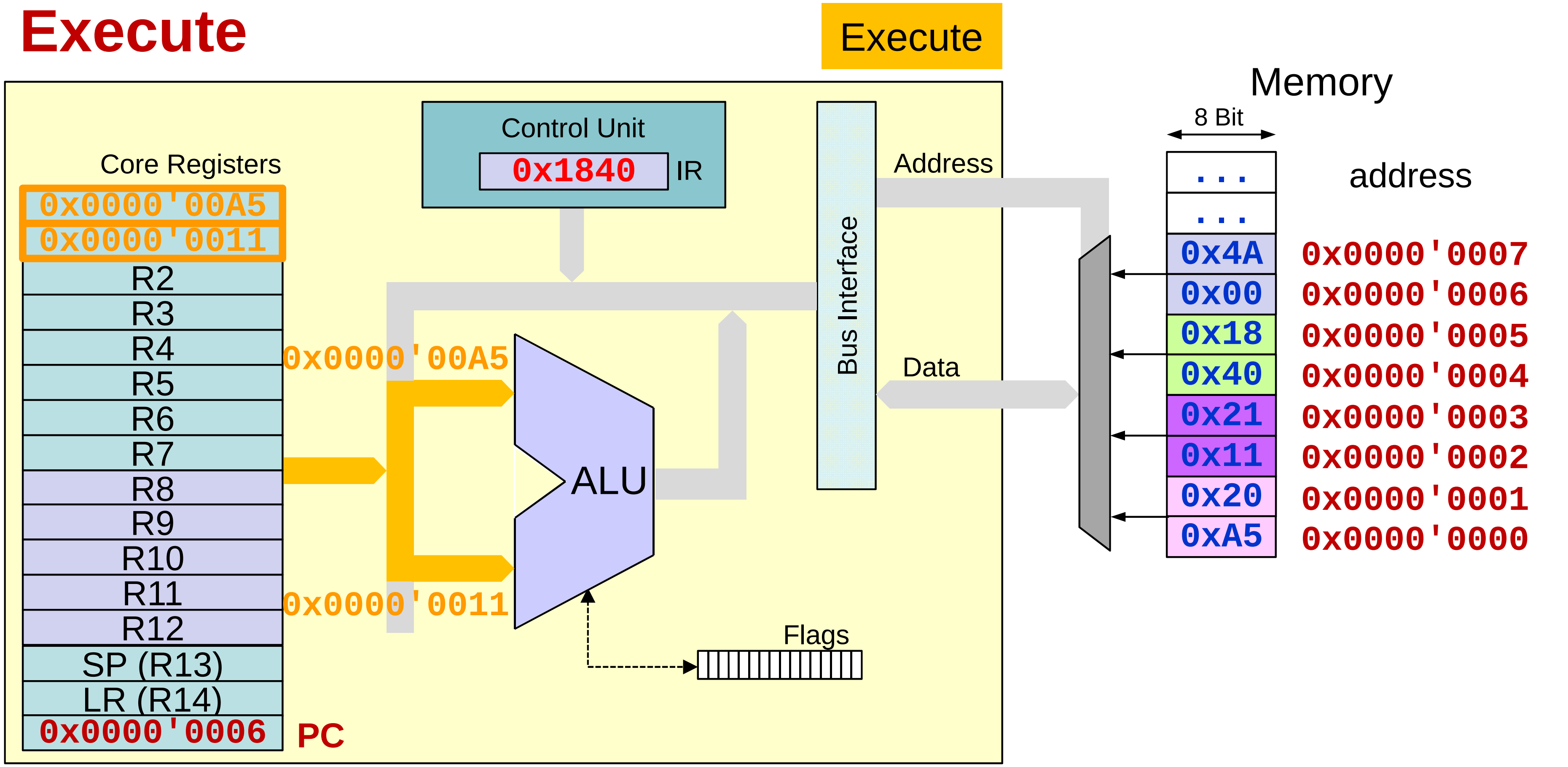


PC →

```
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

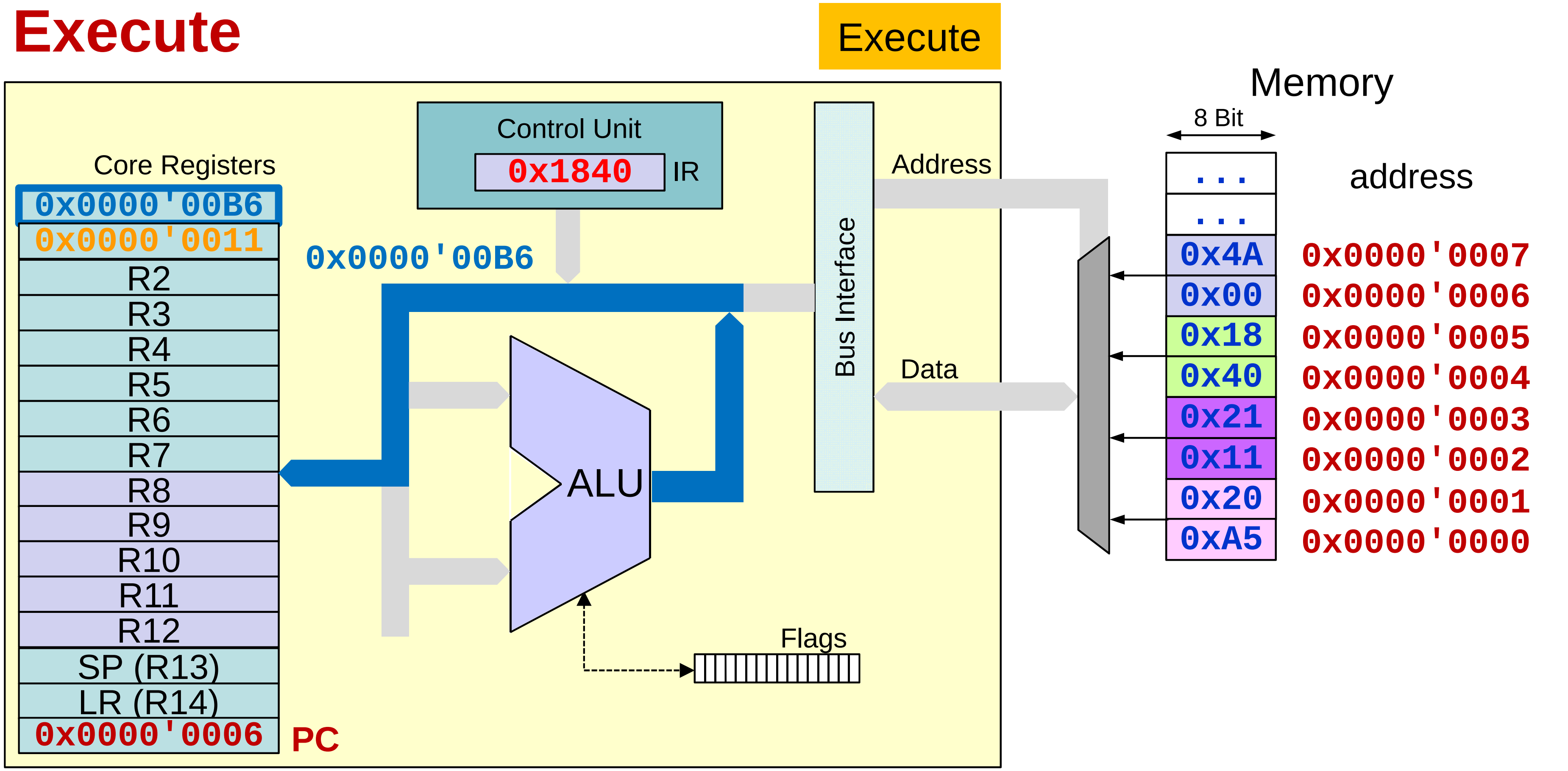


```
00000000 20A5     demoprg MOVS   R0,#0xA5
00000002 2111             MOVS   R1,#0x11
00000004 1840             ADDS   R0,R0,R1
00000006 4A00             LDR    R2,=0x2000
```

PC →

Program Execution

- **Execute**

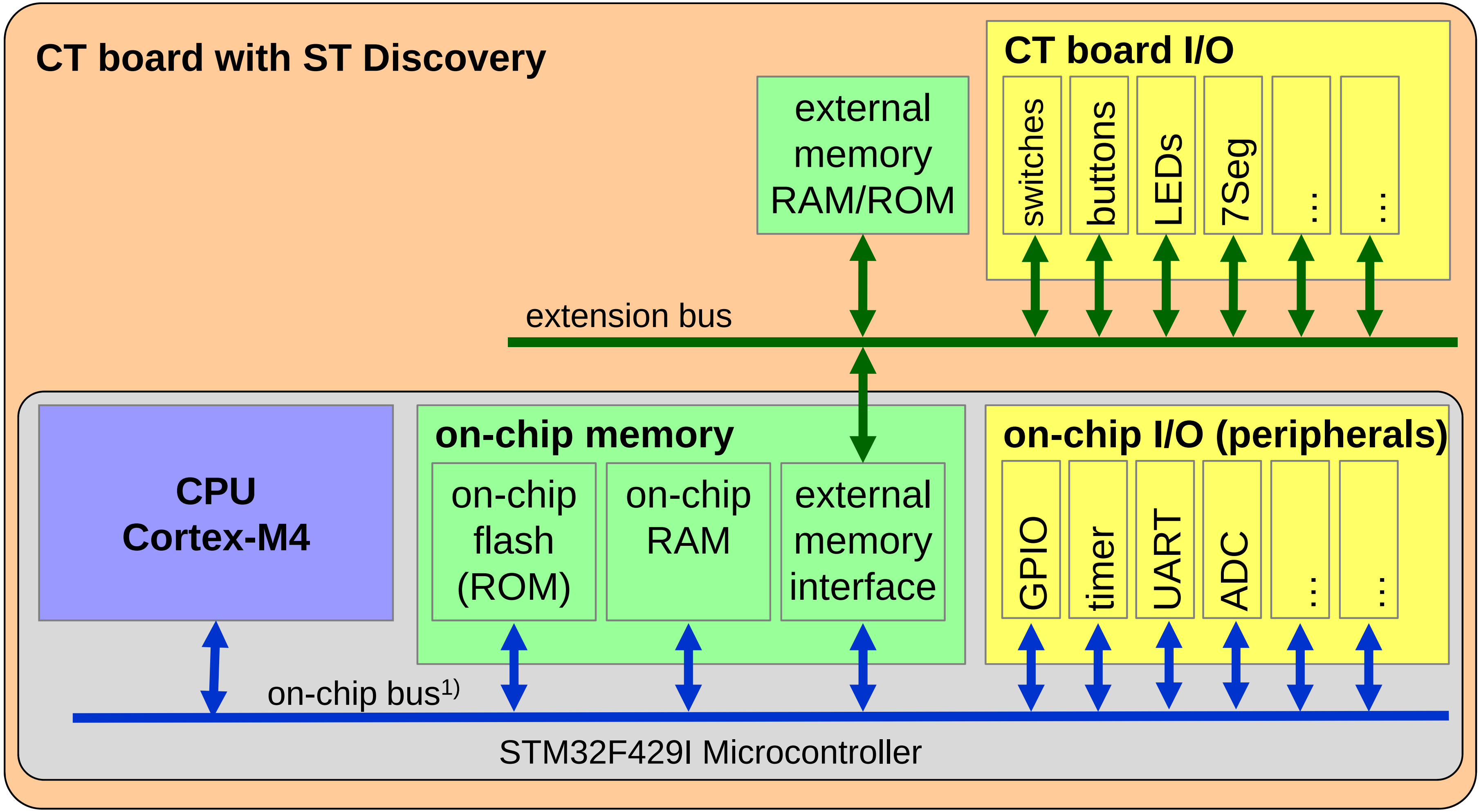


```
00000000 20A5    demoprg MOVS   R0,#0xA5
00000002 2111            MOVS   R1,#0x11
00000004 1840            ADDS   R0,R0,R1
00000006 4A00            LDR    R2,=0x2000
```

PC

Memory Map

Hardware View

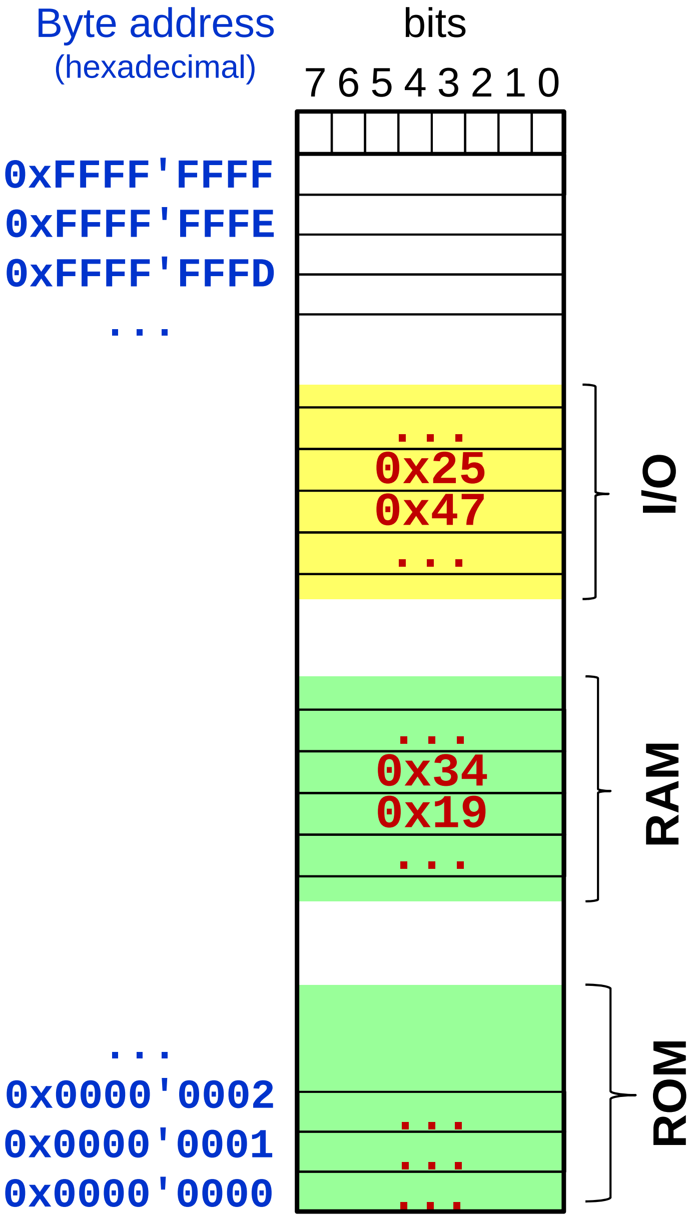


[1] The implementation partitions the on-chip bus into several busses called AHBx and APBx

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 bottom and highest address at the top.

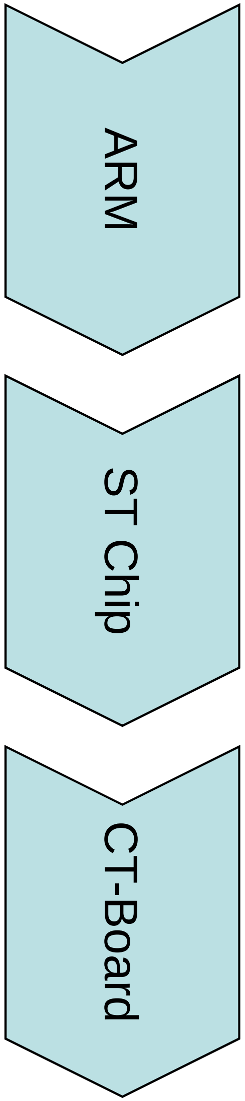
Memory Map

Address Allocation

- ARM policies
  - Cortex-M specific
  - guide lines for chip manufacturer

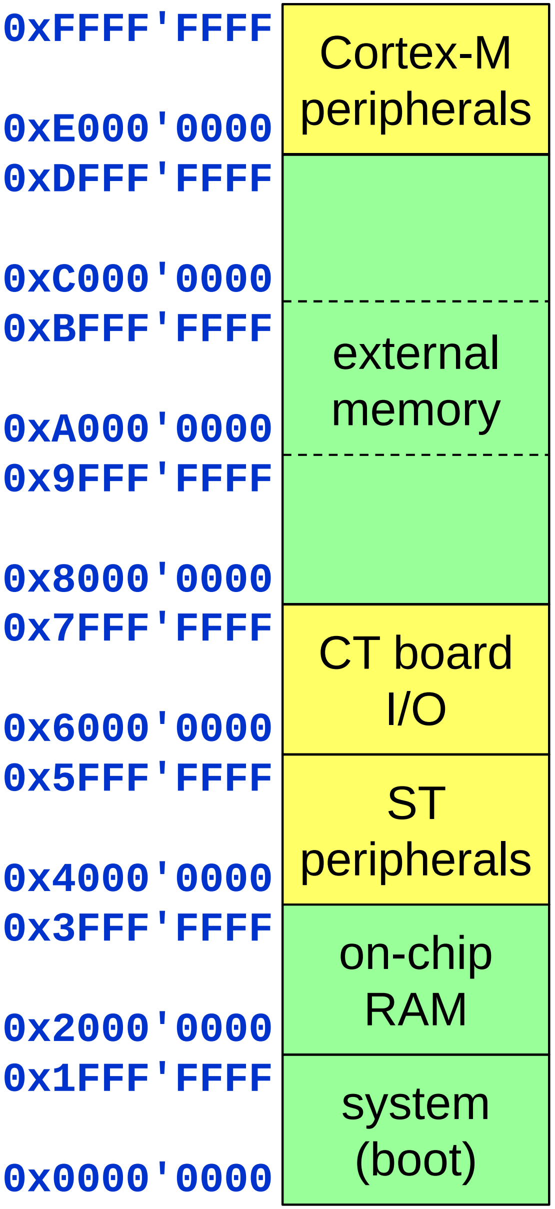
- ST design decisions
  - chip specific
  - number and size of on-chip RAMs
  - size of flash
  - control register for peripherals

- CT board design decisions
  - board specific
  - LEDs, switches, etc



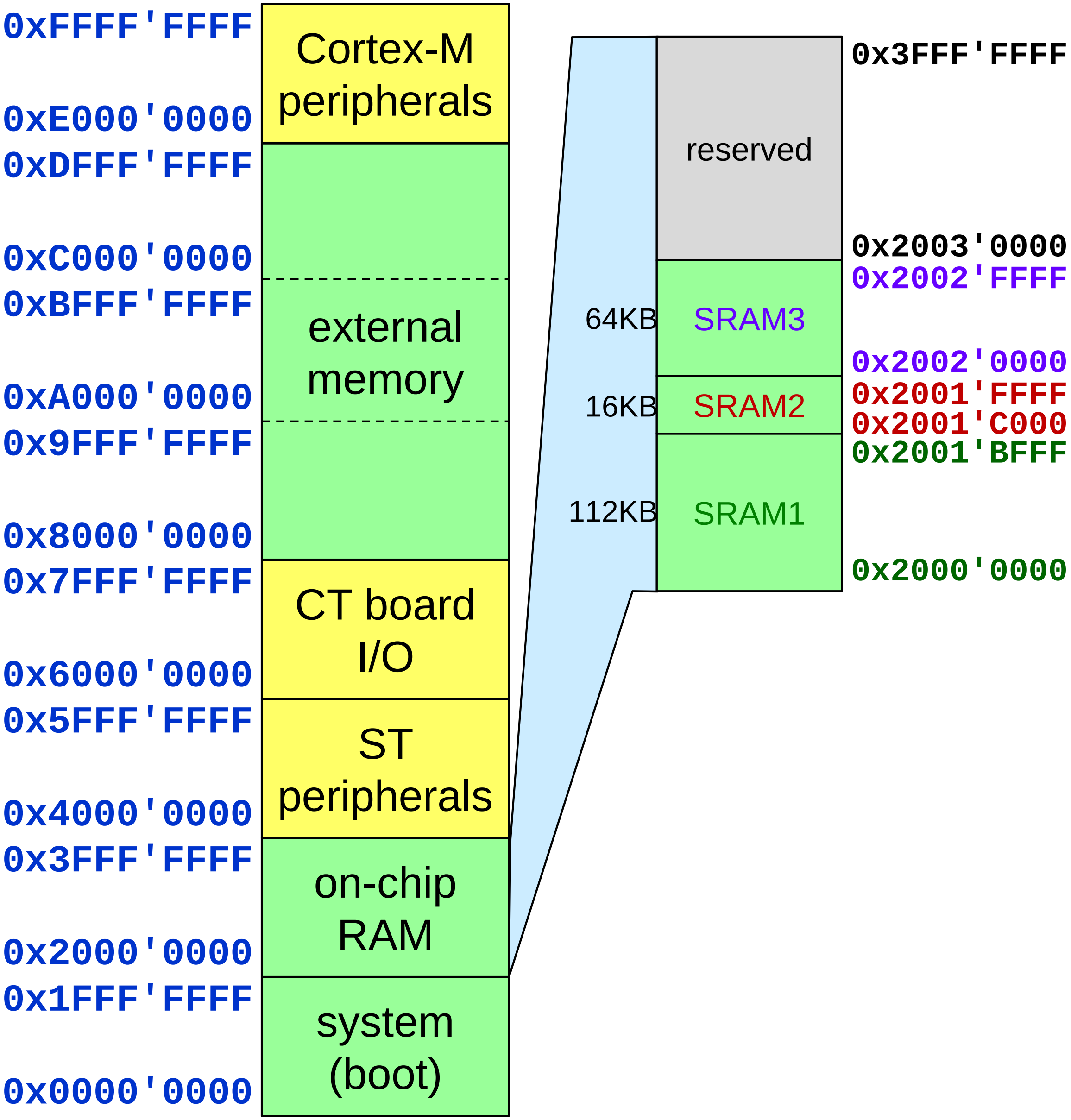
Memory Map

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



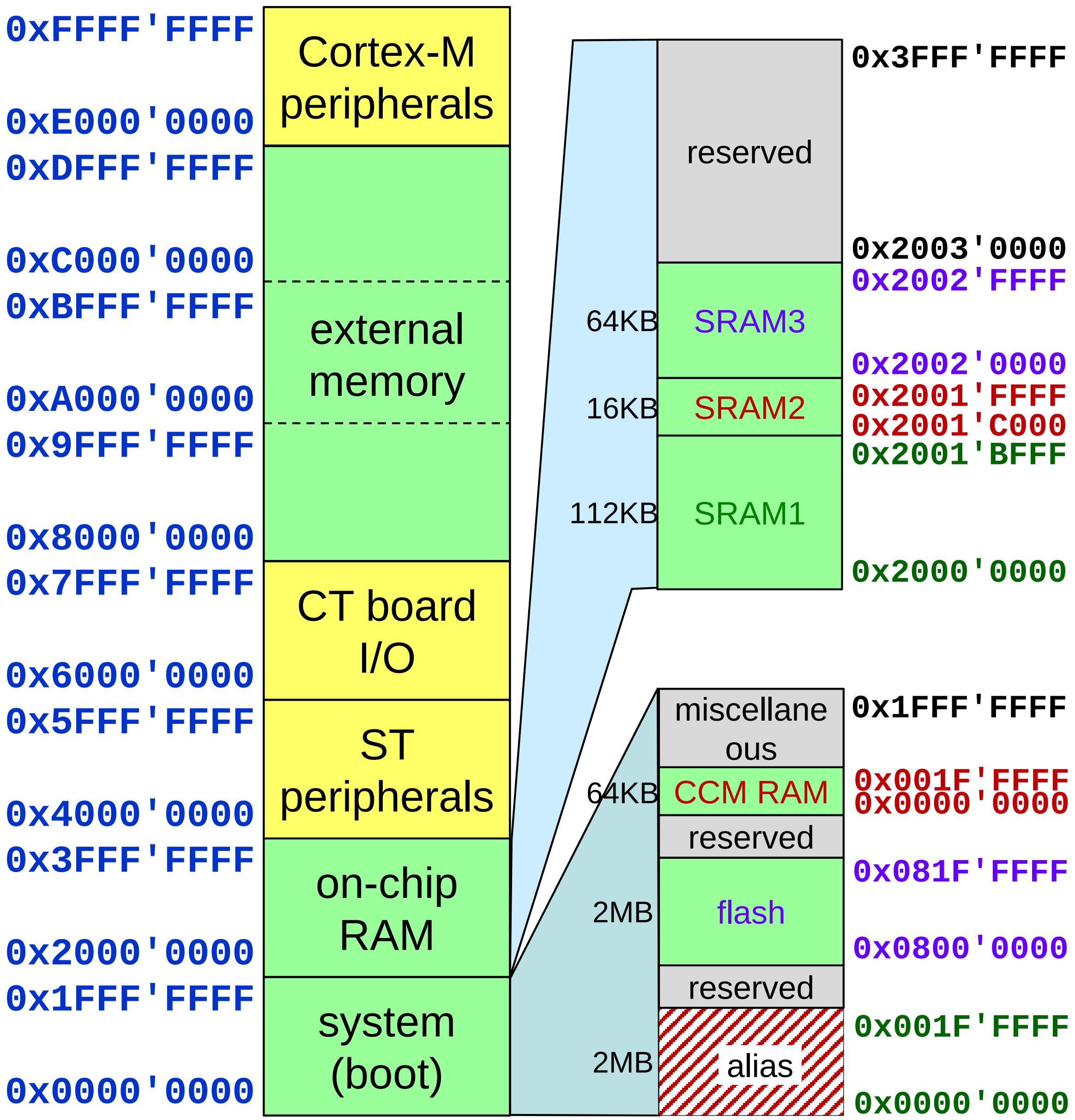
Memory Map

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



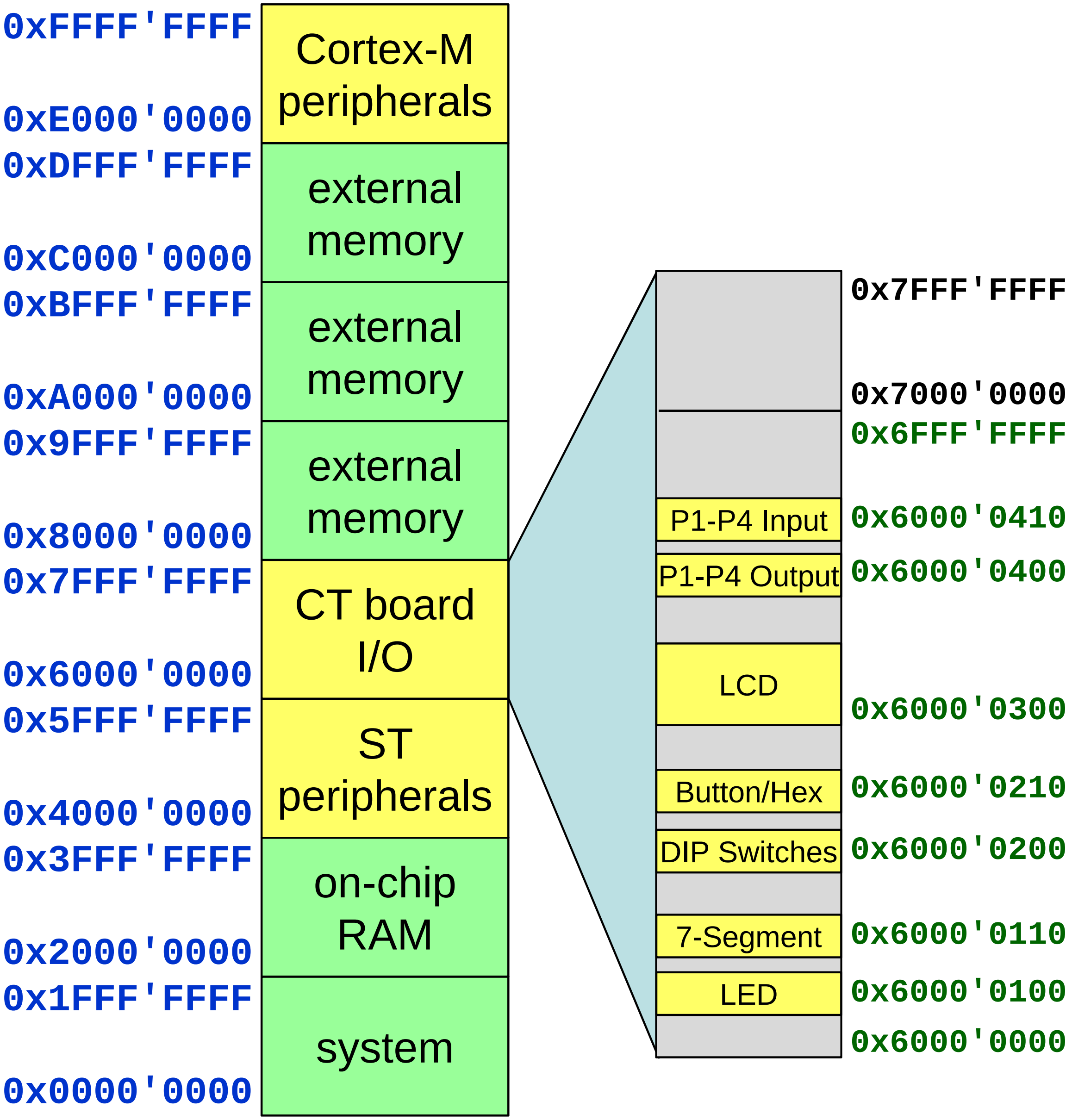
Memory Map

- **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**



Integer Types

Integer Types in C

- Usually memory is organized in bytes
  - one address per byte → reasons: space and history
- An integer type often requires several bytes
- 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

Integer Types

- **ARM Cortex-M**

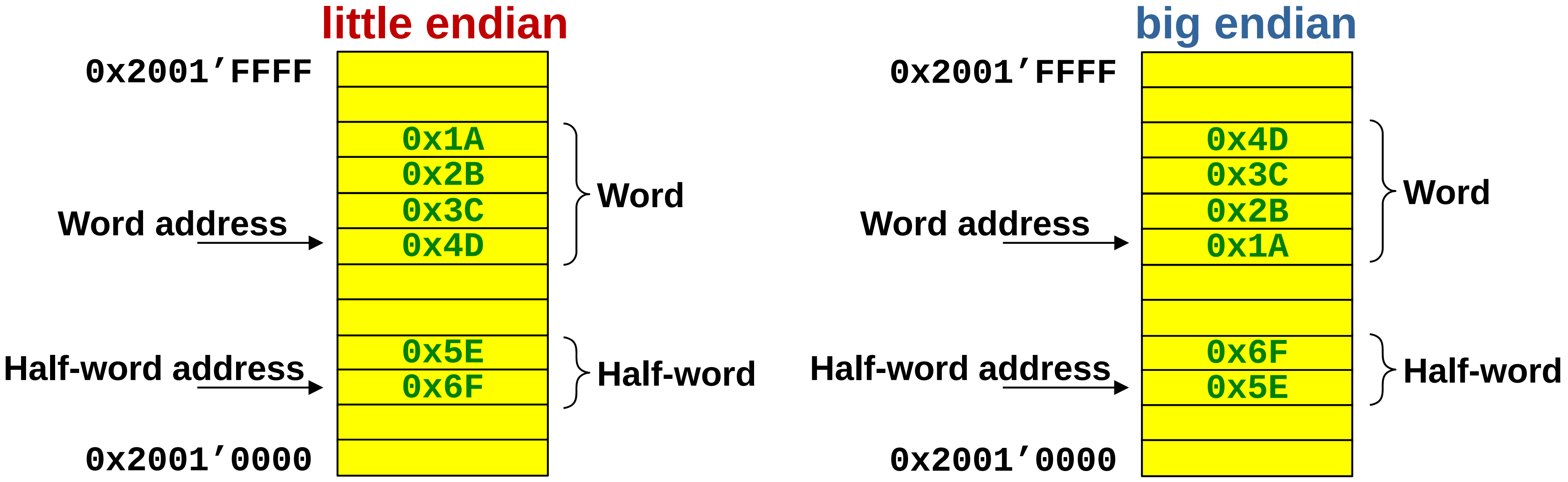
C-Type – unsigned integers	Size	Term	inttypes.h / stdint.h
`unsigned char`	8 Bit	Byte	`uint8_t`
`unsigned short`	16 Bit	Half-word	`uint16_t`
`unsigned int`	32 Bit	Word	`uint32_t`
`unsigned long`	32 Bit	Word	`uint32_t`
`unsigned long long`	64 Bit	Double-word	`uint64_t`

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

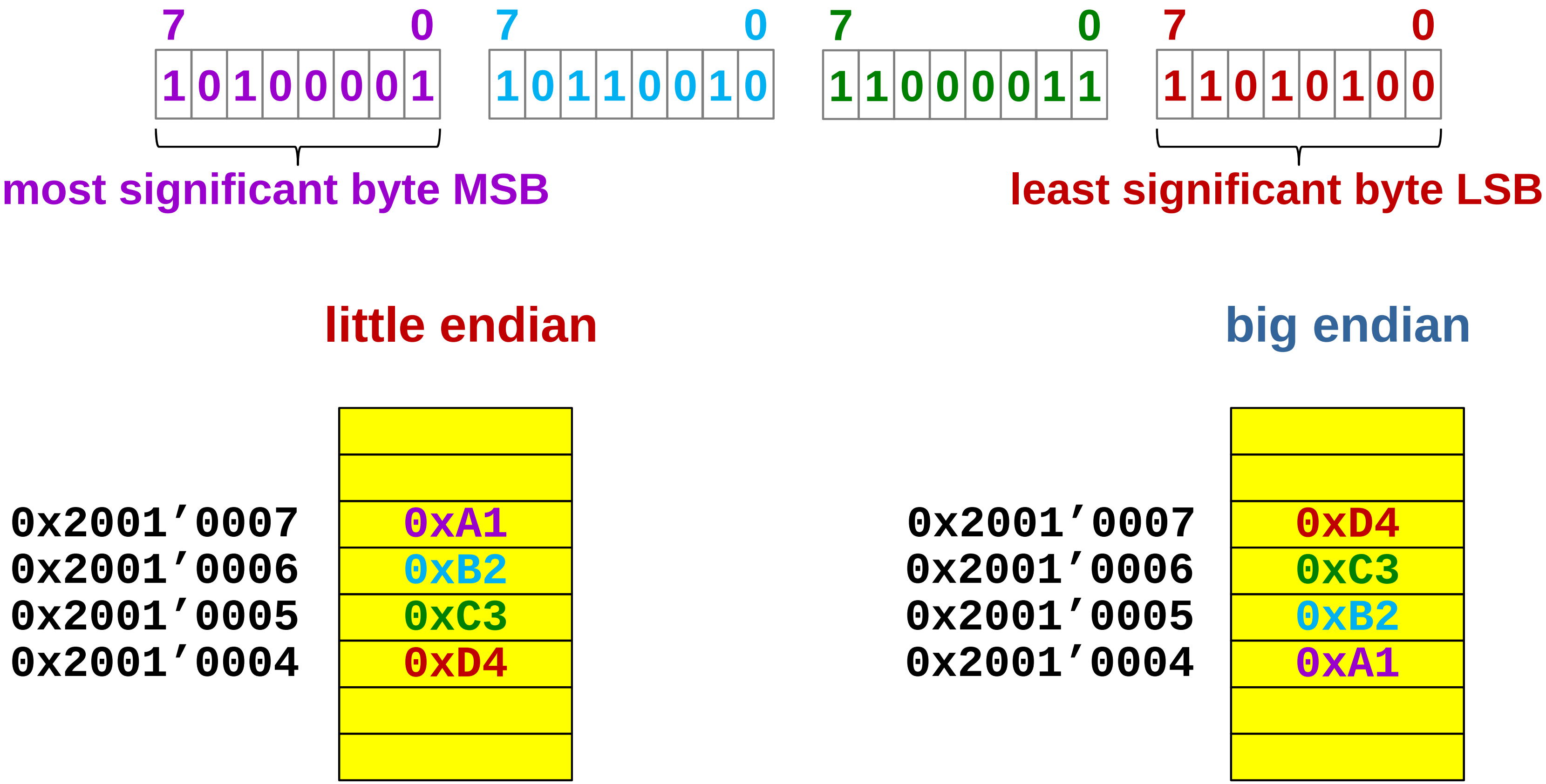


Cortex-M: Chip vendor defines endianness : STM32 (ST Microelectronics) is little endian.

Integer Types

Example

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



7							0	7							0	7							0	7							0
1	0	1	0	0	0	0	1	1	0	1	1	0	0	1	0	1	1	0	0	0	0	1	1	1	1	0	1	0	1	0	0

little endian

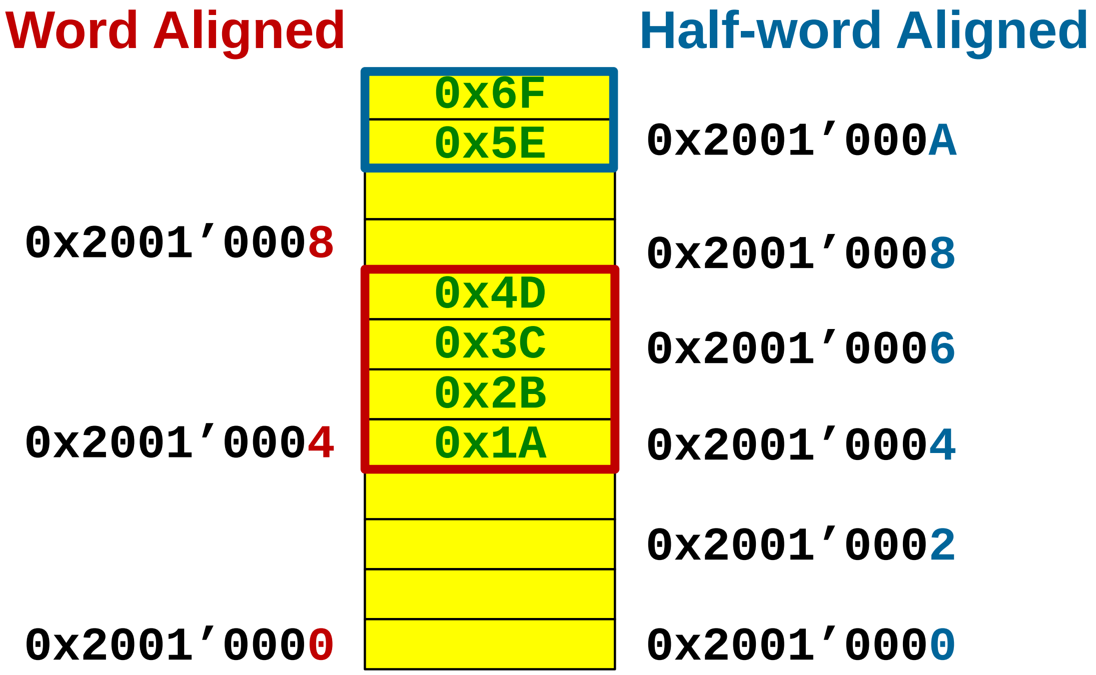
0x2001’0007	0xA1
0x2001’0006	0xB2
0x2001’0005	0xC3
0x2001’0004	0xD4

big endian

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

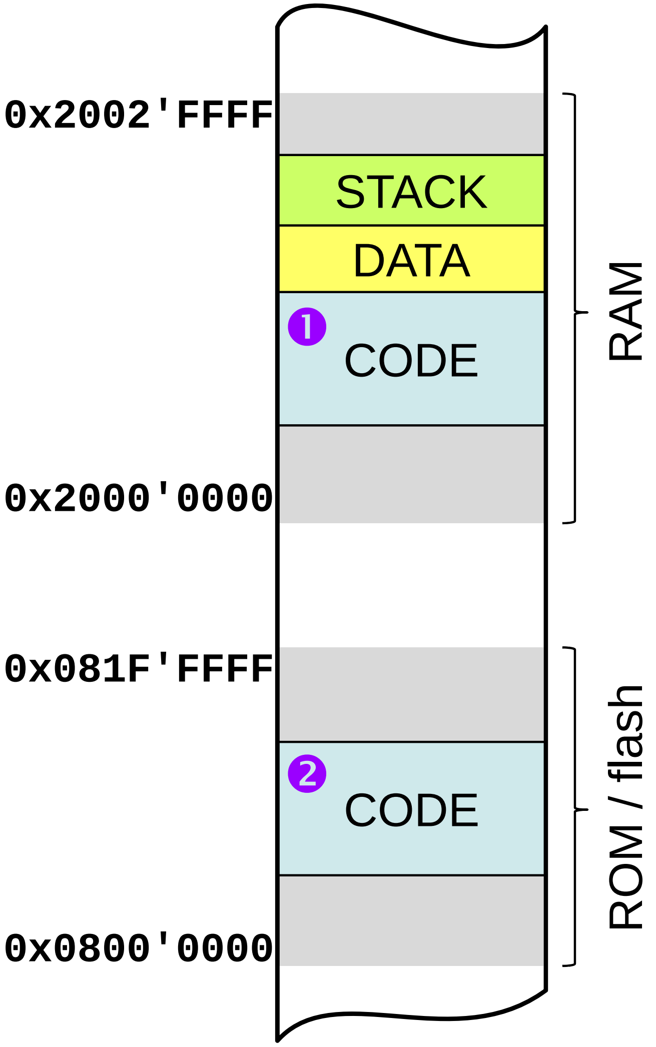
Integer Types

- **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 ❶ or ROM ❷
  - 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

Assembly Program Structure

- AREA directive

```
            AREA    |.text|, CODE, READONLY, ALIGN=2

            ENTRY
start       MOVS     R4,#12
            ADDS     R3,R4,#5
            B        start
            AREA    |.data|, DATA, READWRITE, ALIGN=3

byte_var    DCB      0x1A,0x00
hw_var      DCW      0x2B3C
word_var    DCD      0x4D5E6F70
            AREA     STACK, NOINIT, READWRITE, ALIGN=3

stack_mem   SPACE    0x00000400

END
```

Define code area to include your program

Define data area to store global variables, etc.

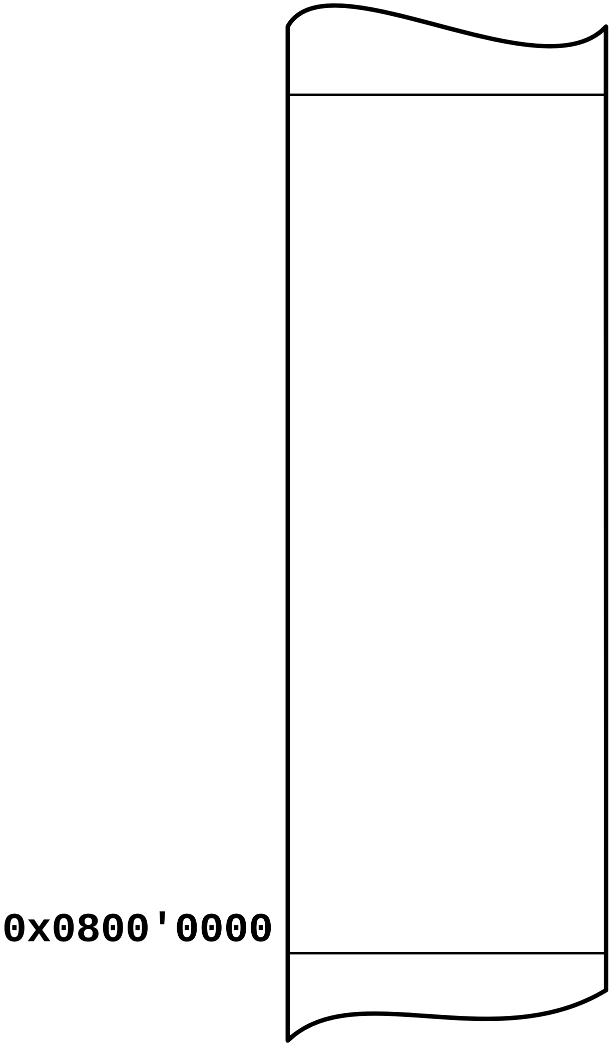
Define stack area to reserve space for stack

Memory Map / Object File Sections

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?

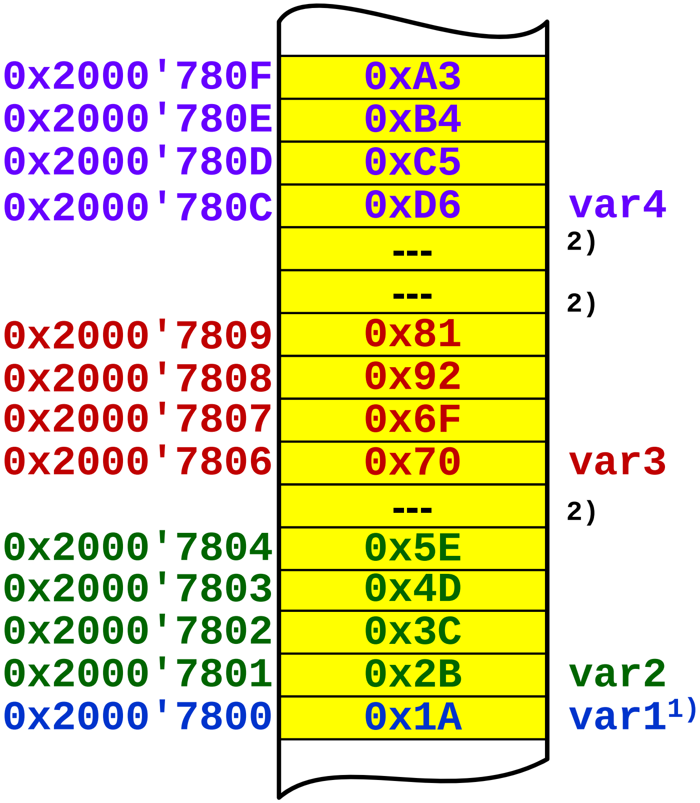


Object File Sections

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

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



0x2000'780F	0xA3	
0x2000'780E	0xB4	
0x2000'780D	0xC5	
0x2000'780C	0xD6	var4
	---	[2]
	---	[2]
0x2000'7809	0x81	
0x2000'7808	0x92	
0x2000'7807	0x6F	
0x2000'7806	0x70	var3
	---	[2]
0x2000'7804	0x5E	
0x2000'7803	0x4D	
0x2000'7802	0x3C	
0x2000'7801	0x2B	var2
0x2000'7800	0x1A	var1[1]

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

[2] Padding bytes introduced for alignment

- Directives for uninitialized data
  - **SPACE** or **%** with number of bytes to be reserved

```
AREA example2, DATA, READWRITE

data1  SPACE 256
```

Object File Sections

Code Example

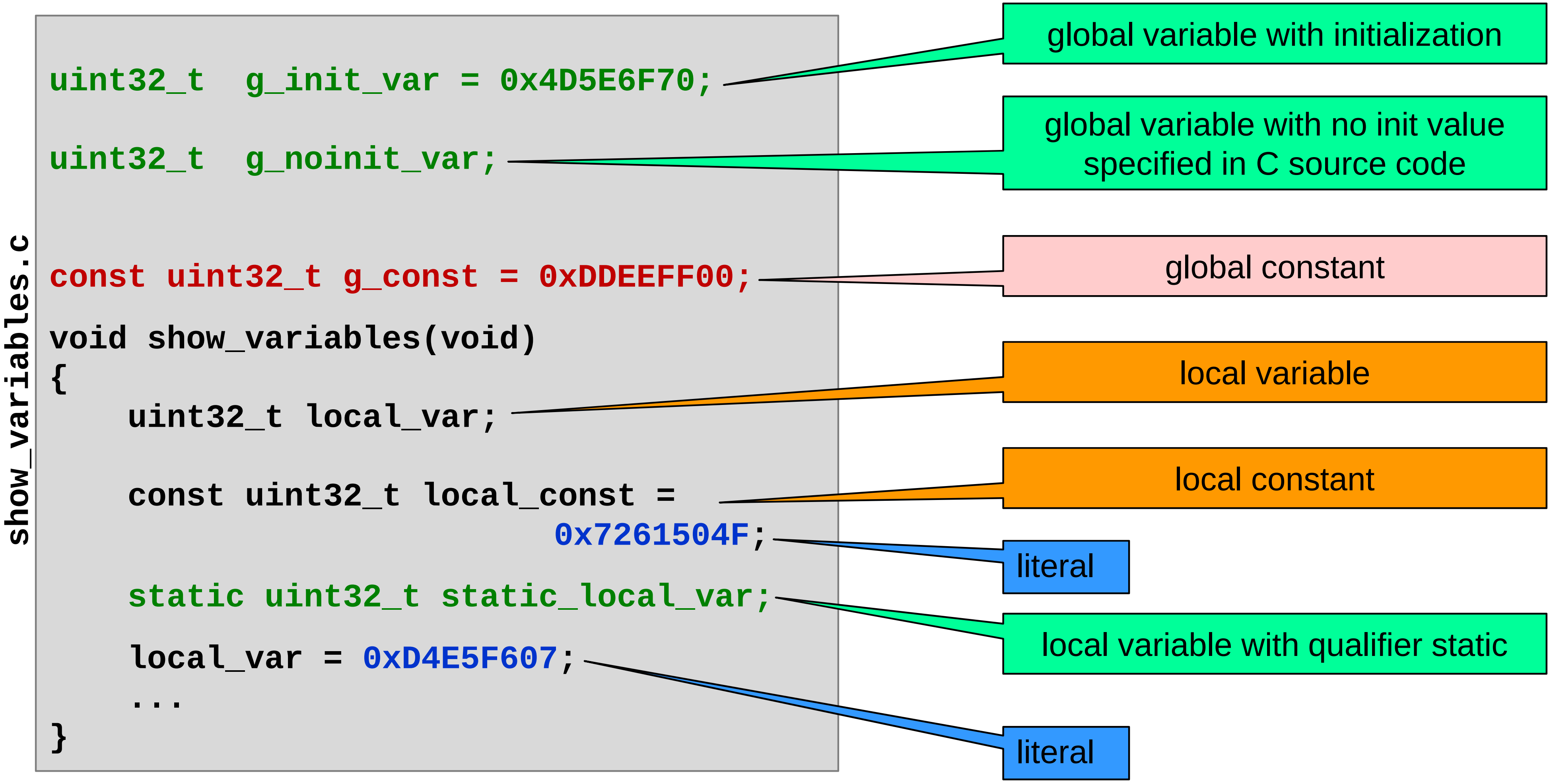
show_variables.c

```
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;
    ...
}
```

Into which sections will the colored objects be allocated?

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;
    ...
}
```

global variable with initialization

global variable with no init value specified in C source code

global constant

local variable

local constant

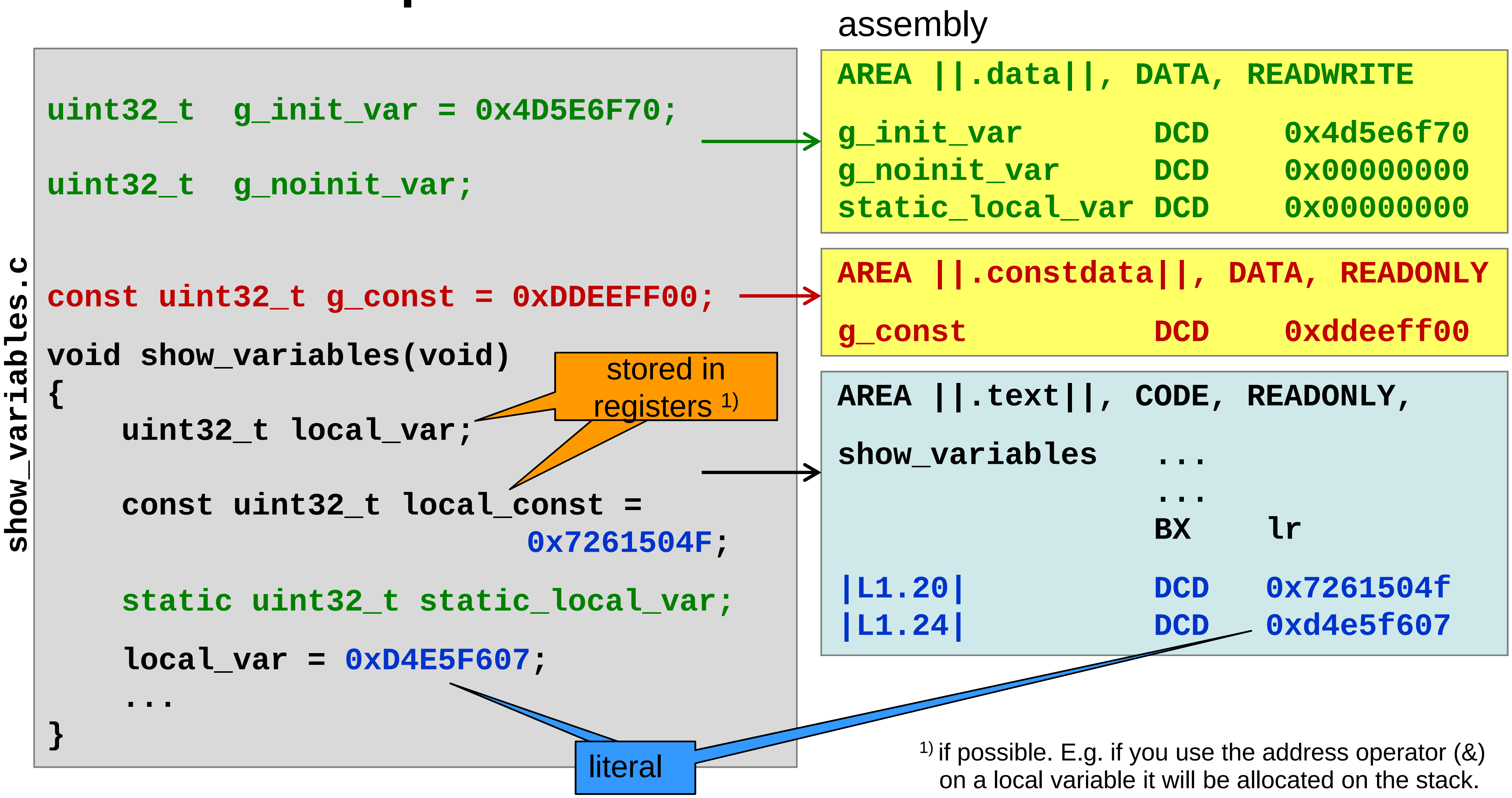
literal

local variable with qualifier static

literal

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;
    ...
}
```

assembly

```
AREA ||.data||, DATA, READWRITE

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

```
AREA ||.constdata||, DATA, READONLY

g_const          DCD    0xddeeff00
```

```
AREA ||.text||, CODE, READONLY,

show_variables   ...
                 ...
                 BX     lr

|L1.20|          DCD   0x7261504f
|L1.24|          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

- **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