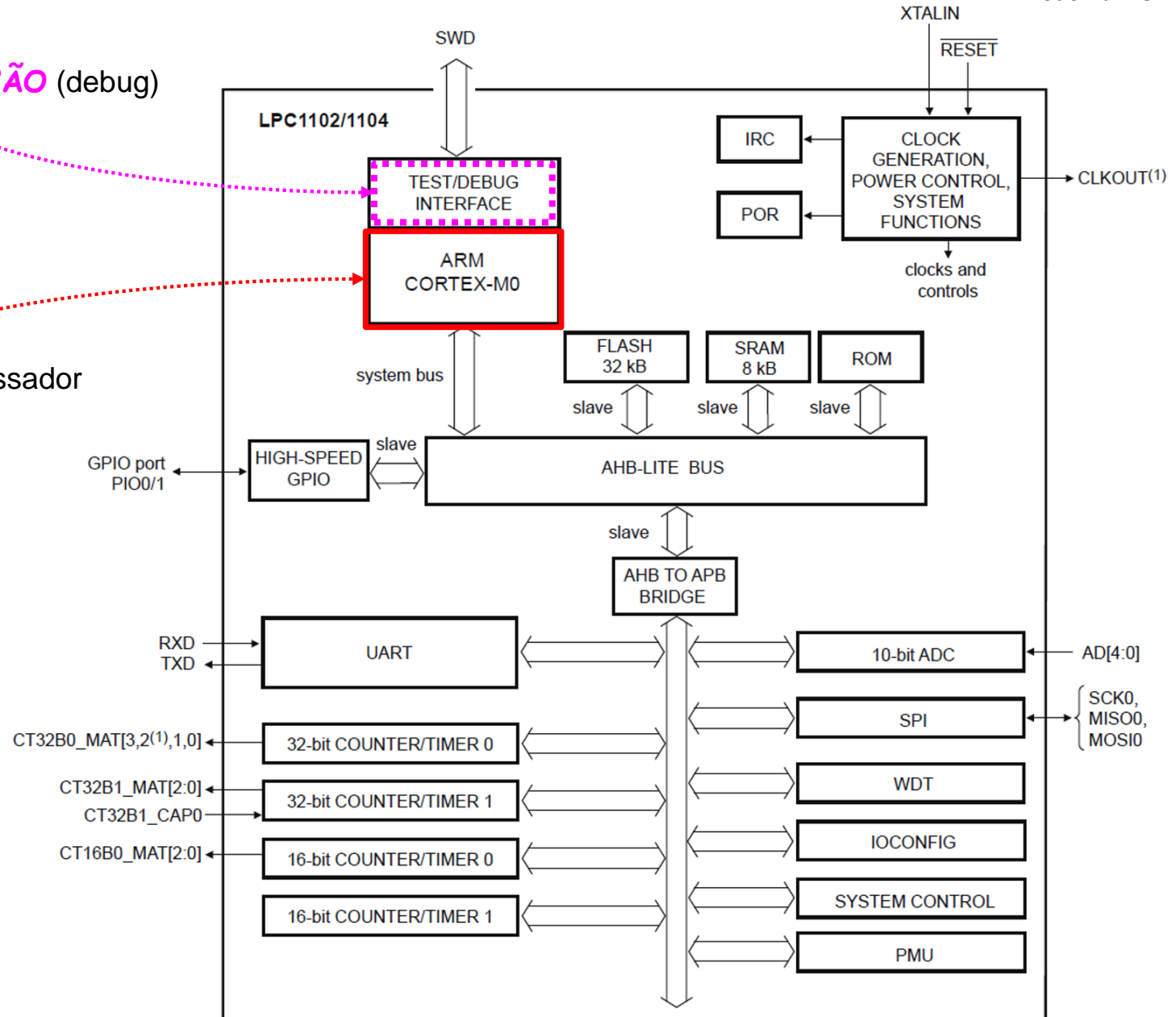


núcleo (CORE) processador

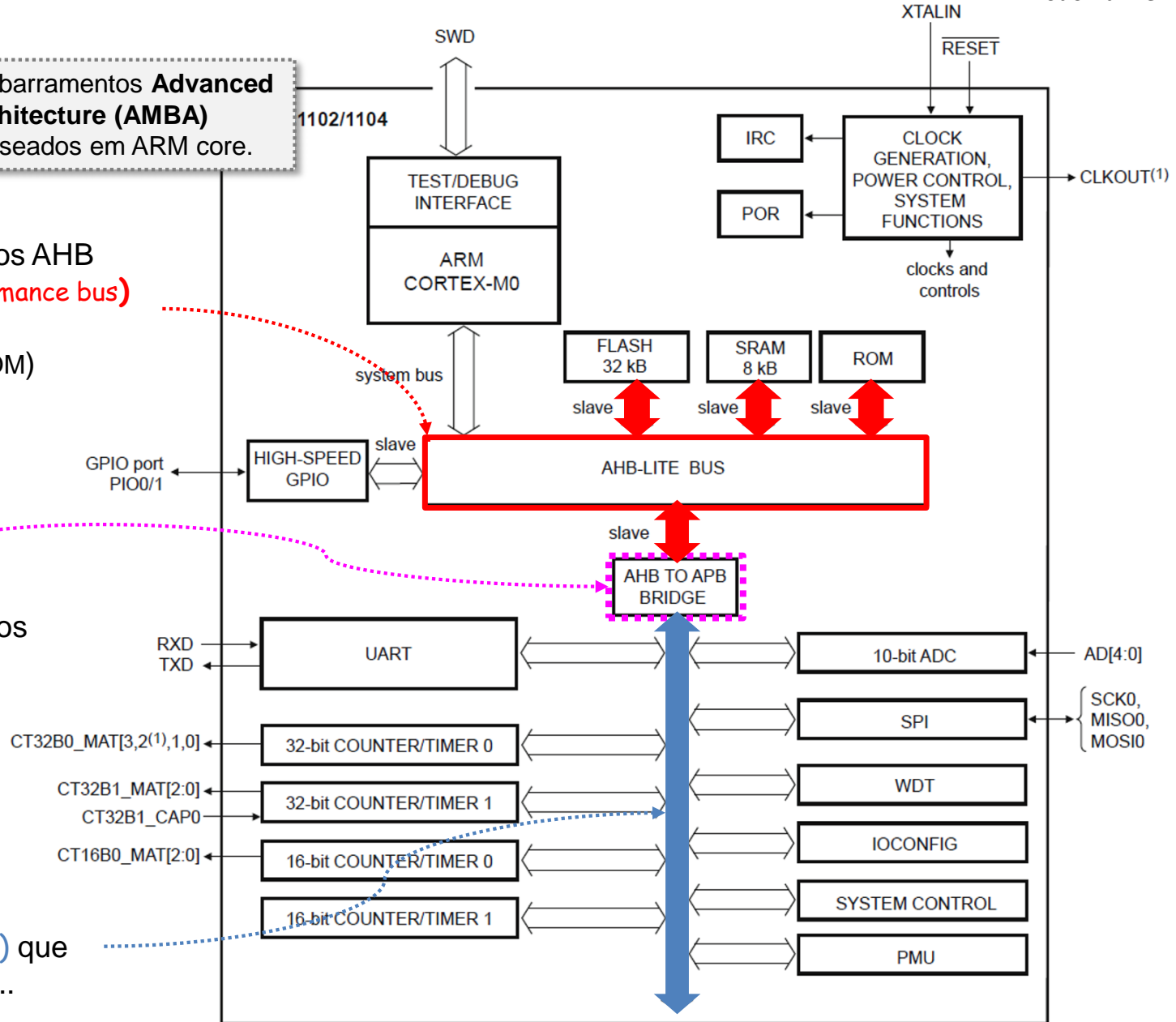


A ARM criou o padrão de barramentos **Advanced Microcontroller Bus Architecture (AMBA)** usado em todos os μC baseados em ARM core.

Matriz de barramentos AHB
(*advanced high-performance bus*)
de alta velocidade...
(liga FLASH, RAM e ROM)

Ponte (*bus bridge*)
com bus de periféricos
de baixa velocidade

Barramento APB
(*advanced peripheral bus*) que
interliga os periféricos...

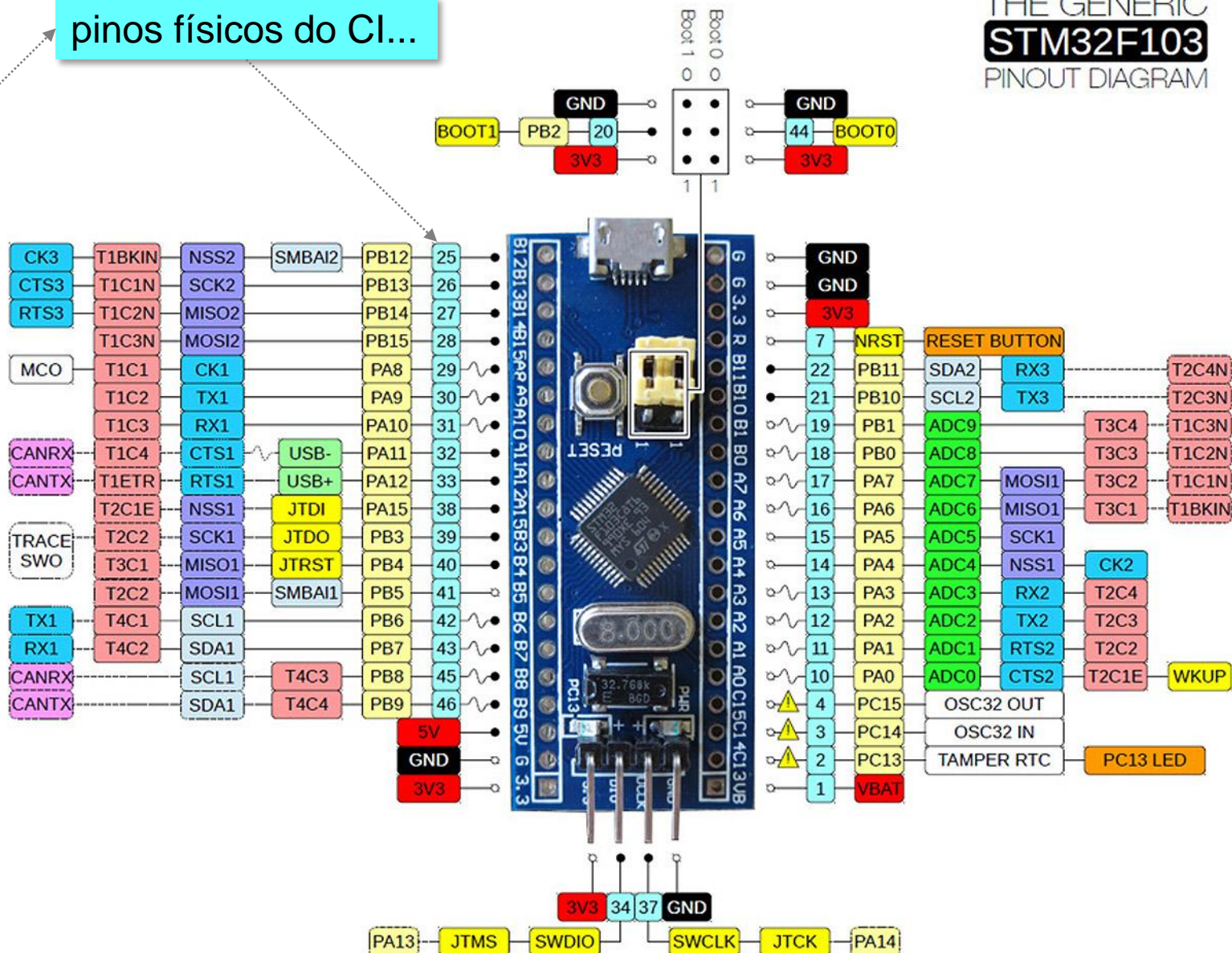


LEGEND

POWER
GROUND
PHYSICAL PIN
PIN NAME
CONTROL
ANALOG
TIMER & CHANNEL
USART
SPI
I2C
CAN BUS
USB
MISC
BOARD HARDWARE
5V tolerant
Not 5V tolerant
PWM pin
Alternate function
PC13, PC14, PC15: Sink max 3mA, source 0mA, max 2MHz, max 30pF
Absolute MAX 150mA total source/sink for entire CPU
Max ± 20 mA per pin, ± 8 mA recommended

pinos físicos do Cl...

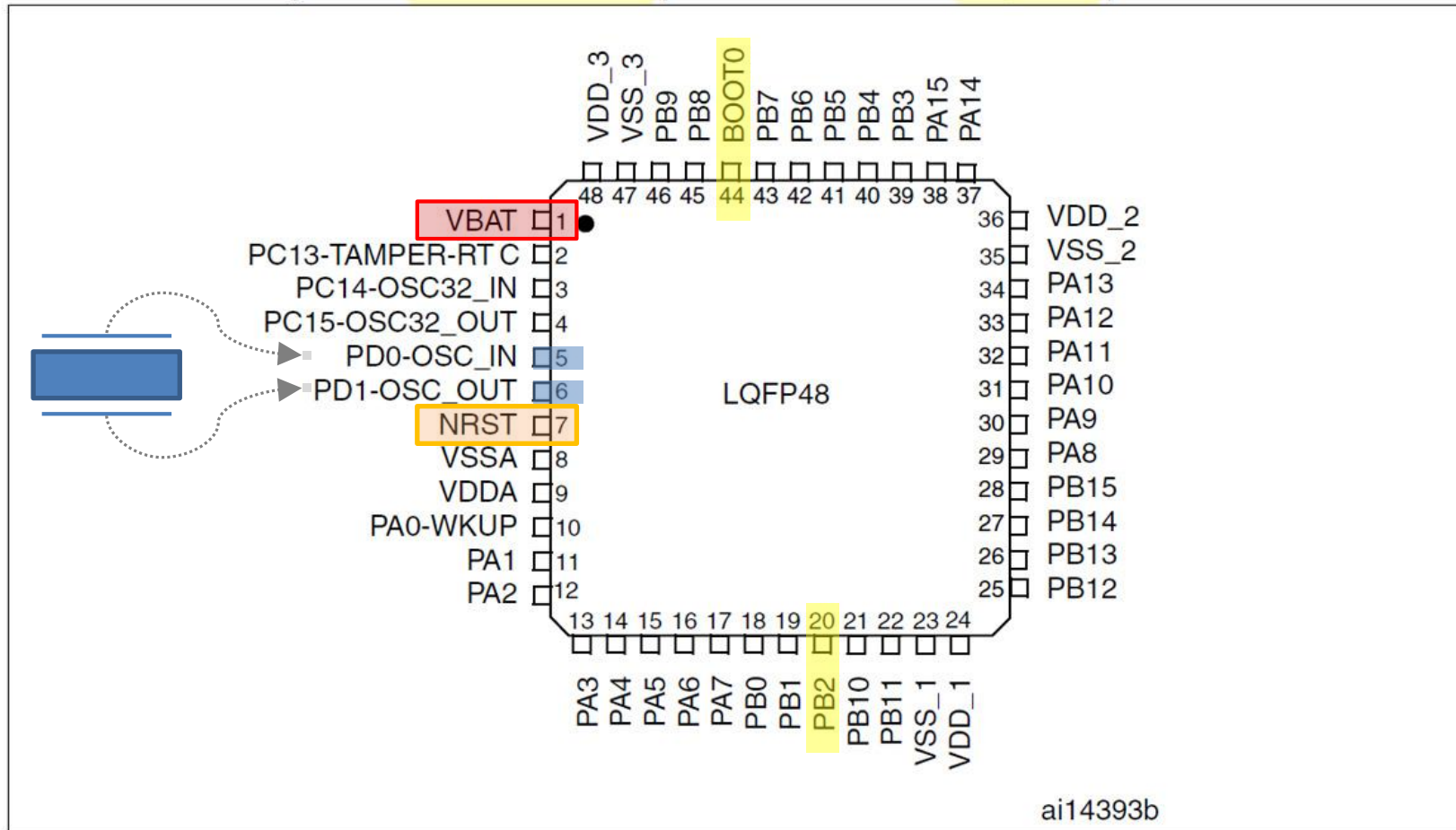
THE GENERIC
STM32F103
PINOUT DIAGRAM



Pinouts and pin description

STM32F103x8, STM32F103xB

Figure 8. STM32F103xx performance line LQFP48 pinout

*apenas conferindo alguns pinos !!!*

Formato típico de uma instrução:
(instruções podem ter um, dois ou três operandos...)

```
operation code      reg destino , operando 2
    MOV             R0      ,      R1
resultado...        ( R0 ← R1 )
```

```
operation code      reg destino , reg fonte , operando 2
    ADD             R2      ,      R3      ,      #0xA
resultado...        ( R2 ← R3 + 10 )
```

Instruções no ARM:

- ✓ 32 bits – full instructions (complexas – com todos os operandos)
- ✓ 16 bits (thumb) – mais simples (economizam memória – programas menores)

Instruções ARM são 32 bits (*normais*) ou 16 bits (*Thumb*) como abaixo:

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
Data processing and FSR transfer	Cond				0	0	1	Opcode				S	Rn				Rd				Operand 2													
Multiply	Cond				0	0	0	0	0	0	A	S	Rd				Rn				Rs				1	0	0	1	Rm					
Multiply long	Cond				0	0	0	0	1	U	A	S	RdHi				RdLo				Rn				1	0	0	1	Rm					
Single data swap	Cond				0	0	0	1	0	B	0	0	Rn				Rd				0	0	0	0	1	0	0	1	Rm					
Branch and exchange	Cond				0	0	0	1	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	1	Rn					
Halfword data transfer, register offset	Cond				0	0	0	P	U	0	W	L	Rn				Rd				0	0	0	0	1	S	H	1	Rm					
Halfword data transfer, immediate offset	Cond				0	0	0	P	U	1	W	L	Rn				Rd				Offset				1	S	H	1	Offset					
Single data transfer	Cond				0	1	1	P	U	B	W	L	Rn				Rd				Offset													
Undefined	Cond				0	1	1																			1								
Block data transfer	Cond				1	0	0	P	U	S	W	L	Rn				Register list																	
Branch	Cond				1	0	1	L											Offset															
Coprocessor data transfer	Cond				1	1	0	P	U	N	W	L	Rn				CRd				CP#				Offset									
Coprocessor data operation	Cond				1	1	1	0	CP Opc				CRn				CRd				CP#				CP		0	CRm						
Coprocessor register transfer	Cond				1	1	1	0	CP Opc				L	CRn				Rd				CP#				CP		1	CRm					
Software interrupt	Cond				1	1	1	1											Ignored by processor															
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Data processing and FSR transfer	Cond		0	0	1	Opcode				S	Rn			Rd			Operand 2															
Multiply	Cond		0	0	0	0	0	0	0	0	A	S	Rd			Rn			Rs			1	0	0	1	Rm						
Multiply long	Cond		0	0	0	0	0	1	U	A	S	RdHi			RdLo			Rn			1	0	0	1	Rm							
Single data swap	Cond		0	0	0	0	0	0	0	0	0	0	Rn			Rd			Rs			1	0	0	1	Rm						
Branch and exchange	Cond		0	0	0	0	0	0	0	0	0	0	Rn			Rd			Rs			1	0	0	1	Rm						
Halfword data transfer, register offset	Cond		0	0	0	0	0	0	0	0	0	0	Rn			Rd			Rs			1	0	0	1	Rm						
Halfword data transfer, immediate offset	Cond		0	0	0	0	0	0	0	0	0	0	Rn			Rd			Rs			1	0	0	1	Rm						
Single data transfer	Cond		0	1	0	1	0	0	0	0	0	0	Rn			Rd			Rs			1	0	0	1	Rm						
Undefined	Cond		0	1	0	1	1	0	0	0	0	0	Rn			Rd			Rs			1	0	0	1	Rm						
Block data transfer	Cond		1	0	0	1	1	1	0	0	0	0	Rn			Rd			Rs			1	0	0	1	Rm						
Branch	Cond		1	0	1	0	0	1	0	0	0	0	Rn			Rd			Rs			1	0	0	1	Rm						
Coprocessor data transfer	Cond		1	1	0	1	0	1	0	0	0	0	Rn			Rd			Rs			1	0	0	1	Rm						
Coprocessor data operation	Cond		1	1	1	0	0	1	0	0	0	0	Rn			Rd			Rs			1	0	0	1	Rm						
Coprocessor register transfer	Cond		1	1	1	0	1	0	0	0	0	0	Rn			Rd			Rs			1	0	0	1	Rm						
Software interrupt	Cond		1	1	1	1	1	1	1	1	1	1	Rn			Rd			Rs			1	0	0	1	Rm						

Cond: Condition field

0000	EQ (EQual)
0001	NE (NEver)
0010	CS (Carry Set)
0011	CC (Carry Clear)
0100	MI (MInus)
0101	PL (PLus)
0110	VS (oVerflow Set)
0111	VC (oVerflow Clear)
1000	HI (Hlgher)
1001	LS (Lower or Same)
1010	GE (Greater or Equal)
1011	LT (Less Than)
1100	GT (Greater Than)
1101	LE (Less than or Equal)
1110	AL (ALways)
1111	NV (NeVer)

OpCode: Operation code

0000	AND
0001	EOR
0010	SUB
0011	RSB
0100	ADD
0101	ADC
0110	SBC
0111	RSC
1000	TST
1001	TEQ
1010	CMP
1011	CMN
1100	ORR
1101	MOV
1110	BIC
1111	MVN

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Data processing and FSR transfer	Cond		0	0	1	Opcode				S	Rn				Rd				Operand 2													
Multiply	Cond	0	0	0	0	0	0	0	0	A	S	Rd				Rn				Rs				1	0	0	1	Rm				
Multiply long	Cond	0	0	0	0	0	1	U	A	S	RdHi				RdLo				Rn				1	0	0	1	Rm					
Single data swap	Cond	0	0																													
Branch and exchange	Cond	0	0																													
Halfword data transfer, register offset	Cond	0	0																													
Halfword data transfer, immediate offset	Cond	0	0																													
Single data transfer	Cond	0	1																													
Undefined	Cond	0	1																													
Block data transfer	Cond	1	0																													
Branch	Cond	1	0																													
Coprocessor data transfer	Cond	1	1																													
Coprocessor data operation	Cond	1	1																													
Coprocessor register transfer	Cond	1	1																													
Software interrupt	Cond	1	1																													

Cond: Condition field

0000	EQ (EQual)
0001	NE (NEver)
0010	CS (Carry Set)
0011	CC (Carry Clear)
0100	MI (MInus)
0101	PL (PLus)
0110	VS (oVerflow Set)
0111	VC (oVerflow Clear)
1000	HI (Hlgher)
1001	LS (Lower or Same)
1010	GE (Greater or Equal)
1011	LT (Less Than)
1100	GT (Greater Than)
1101	LE (Less than or Equal)
1110	AL (ALways)
1111	NV (NeVer)

OpCode: Operation code

0000	AND
0001	EOR
0010	SUB
0011	RSB
0100	ADD
0101	ADC
0110	SBC
0111	RSC
1000	TST
1001	TEQ
1010	CMP
1011	CMN
1100	ORR
1101	MOV
1110	BIC
1111	MVN

Exemplo:

SUBNE r0, r1, r2

se (NEQ) r0 = r1 – r2

instrução	Operando	Descrição	Ação executada
ADC, ADCS	{Rd,} Rn, Op2	Add with Carry	$Rd \leftarrow Rn + Op2 + \text{Carry}$, ADCS updates N,Z,C,V
ADD, ADDS	{Rd,} Rn, Op2	Add	$Rd \leftarrow Rn + Op2$, ADDS updates N,Z,C,V
ADD, ADDS	{Rd,} Rn, #imm12	Add Immediate	$Rd \leftarrow Rn + \text{imm12}$, ADDS updates N,Z,C,V
ADR	Rd, label	Load PC-relative Address	$Rd \leftarrow \langle \text{label} \rangle$
AND, ANDS	{Rd,} Rn, Op2	Logical AND	$Rd \leftarrow Rn \text{ AND } Op2$, ANDS updates N,Z,C
ASR, ASRS	Rd, Rm, <Rs n>	Arithmetic Shift Right	$Rd \leftarrow Rm \gg (Rs n)$, ASRS updates N,Z,C
B	label	Branch	$PC \leftarrow \text{label}$
BFC	Rd, #lsb, #width	Bit Field Clear	$Rd[(\text{width} + \text{lsb} - 1) : \text{lsb}] \leftarrow 0$
BFI	Rd, Rn, #lsb, #width	Bit Field Insert	$Rd[(\text{width} + \text{lsb} - 1) : \text{lsb}] \leftarrow Rn[(\text{width} - 1) : 0]$
BIC, BICS	{Rd,} Rn, Op2	Bit Clear	$Rd \leftarrow Rn \text{ AND NOT } Op2$, BICS updates N,Z,C
BKPT	#imm	Breakpoint	<i>Prefetch abort or enter debug state</i>
BL	label	Branch with Link	$LR \leftarrow \text{next instruction}$, $PC \leftarrow \text{label}$
BLX	Rm	Branch reg with link	$LR \leftarrow \text{next instr addr}$, $PC \leftarrow Rm[31:1]$
BX	Rm	Branch register	$PC \leftarrow Rm$
CBNZ	Rn, label	Comp & Branch if Non-zero	$PC \leftarrow \text{label}$ if $Rn \neq 0$
CBZ	Rn, label	Compare & Branch if Zero	$PC \leftarrow \text{label}$ if $Rn == 0$
CLREX	-	Clear	Clear local processor exclusive tag
CLZ	Rd, Rm	Count Leading Zeros	$Rd \leftarrow \text{number of leading zeros in } Rm$
CMN	Rn, Op2	Compare Negative	Update N,Z,C,V flags on $Rn + Op2$
CMP	Rn, Op2	Compare	Update N,Z,C,V flags on $Rn - Op2$
CPSID	i	Disable specified (i) interrupts	optional change mode
CPSIE	i	Enable specified (i) interrupts	optional change mode

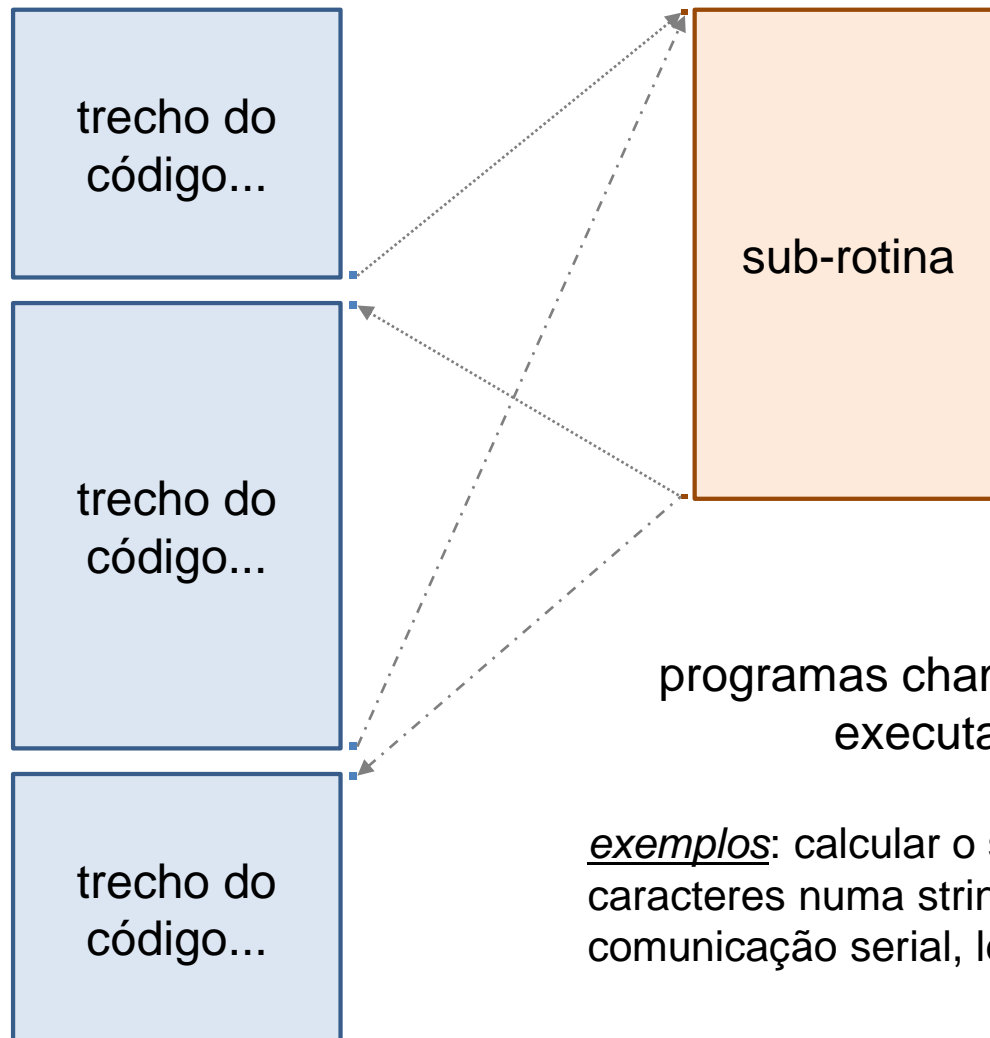
instrução	operando	Descrição	Ação executada
DMB	-	Data Memory Barrier,	ensure memory access order
DSB	-	Data Synchronization Barrier,	ensure completion of access
EOR, EORS	{Rd,} Rn, Op2	Exclusive OR,	$Rd \leftarrow Rn \text{ XOR } Op2$, EORS updates N,Z,C
ISB	-	Instruction Synchronization Barrier	
IT	-	If-Then Condition Block	
LDM	Rn{!}, reglist	Load Multiple Regs, increment after	$\langle reglist \rangle = mem[Rn]$, Rn ++ after each mem access
LDMDB, LDMEA	Rn{!}, reglist	Load Multiple Regs, decrement before	$\langle reglist \rangle = mem[Rn]$, Rn - - before each mem access
LDMFD, LDMIA	Rn{!}, reglist		$\langle reglist \rangle = mem[Rn]$, Rn ++ after each mem access
LDR	Rt, [Rn, #offset]	Load Register with Word,	$Rt \leftarrow mem[Rn + offset]$
LDRB, LDRBT	Rt, [Rn, #offset]	Load Register with Byte,	$Rt \leftarrow mem[Rn + offset]$
LDRD	Rt, Rt2, [Rn, #offset]	Load Register with two words	$Rt \leftarrow mem[Rn + offset]$, $Rt2 \leftarrow mem[Rn + offset + 4]$
LDREX	Rt, [Rn, #offset]	Load Register Exclusive	$Rt \leftarrow mem[Rn + offset]$
LDREXB	Rt, [Rn]	Load Reg Exclusive with Byte	$Rt \leftarrow mem[Rn]$
LDREXH	Rt, [Rn]	Load Reg Exclusive with Half-word	$Rt \leftarrow mem[Rn]$
LDRH, LDRHT	Rt, [Rn, #offset]	Load Reg with Half-word	$Rt \leftarrow mem[Rn + offset]$
LDRSB, LDRSBT	Rt, [Rn, #offset]	Load Reg with Signed Byte	$Rt \leftarrow mem[Rn + offset]$
LDRSH, LDRSHT	Rt, [Rn, #offset]	Load Reg with Signed Half-word	$Rt \leftarrow mem[Rn + offset]$
LDRT	Rt, [Rn, #offset]	Load Register with Word	$Rt \leftarrow mem[Rn + offset]$
LSL, LSLs	Rd, Rm, <Rs #n>	Logic Shift Left	$Rd \leftarrow Rm \ll Rs n$, LSLs update N,Z,C
LSR, LSRS	Rd, Rm, <Rs #n>	Logic Shift Right	$Rd \leftarrow Rm \gg Rs n$, LSRS update N,Z,C

instrução	operando	Descrição	Ação executada
MLA	Rd, Rn, Rm, Ra	Multiply with Accumulate	$Rd \leftarrow (Ra + (Rn * Rm))[31:0]$
MLS	Rd, Rn, Rm, Ra	Multiply with Subtract	$Rd \leftarrow (Ra - (Rn * Rm))[31:0]$
MOV, MOVS	Rd, Op2	Move	$Rd \leftarrow Op2$, MOVS updates N,Z,C
MOVT	Rd, #imm16	Move Top	$Rd[31:16] \leftarrow imm16$, $Rd[15:0]$ unaffected
MOVW, MOVWS	Rd, #imm16	Move 16-bit Constant	$Rd \leftarrow imm16$, MOVWS updates N,Z,C
MRS	Rd, spec_reg	Move from Special Register	$Rd \leftarrow spec_reg$
MSR	spec_reg, Rm	Move to Special Register	$spec_reg \leftarrow Rm$, Updates N,Z,C,V
MUL, MULS	{Rd,} Rn, Rm	Multiply	$Rd \leftarrow (Rn * Rm)[31:0]$, MULS updates N,Z
MVN, MVNS	Rd, Op2	Move NOT	$Rd \leftarrow 0xFFFFFFFF \text{ EOR } Op2$, MVNS updates N,Z,C
NOP	-	No Operation	
ORN, ORNS	{Rd,} Rn, Op2	Logical OR NOT,	$Rd \leftarrow Rn \text{ OR NOT } Op2$, ORNS updates N,Z,C
ORR, ORRS	{Rd,} Rn, Op2	Logical OR,	$Rd \leftarrow Rn \text{ OR } Op2$, ORRS updates N,Z,C
POP	reglist	Canonical form of LDM SP!	<reglist>
PUSH	reglist	Canonical form of STMDB SP!	<reglist>
RBIT	Rd, Rn	Reverse Bits,	for (i = 0; i < 32; i++): $Rd[i] = Rn[31-i]$
REV	Rd, Rn	Reverse Byte Order in a Word	$Rd[31:24] \leftarrow Rn[7:0]$, $Rd[23:16] \leftarrow Rn[15:8]$, $Rd[15:8] \leftarrow Rn[23:16]$, $Rd[7:0] \leftarrow Rn[31:24]$
REV16	Rd, Rn	Reverse Byte Order in a Half-word,	$Rd[15:8] \leftarrow Rn[7:0]$, $Rd[7:0] \leftarrow Rn[15:8]$, $Rd[31:24] \leftarrow Rn[23:16]$, $Rd[23:16] \leftarrow Rn[31:24]$
REVSH	Rd, Rn	Reverse Byte order in Low Half-word and sign extend,	$Rd[15:8] \leftarrow Rn[7:0]$, $Rd[7:0] \leftarrow Rn[15:8]$, $Rd[31:16] \leftarrow Rn[7] * \&0xFFFF$
ROR, RORS	Rd, Rm, <Rs #n>	Rotate Right,	$Rd \leftarrow ROR(Rm, Rs n)$, RORS updates N,Z,C
RRX, RRXS	Rd, Rm	Rotate Right with Extend,	$Rd \leftarrow RRX(Rm)$, RRXS updates N,Z,C
RSB, RSBS	{Rd,} Rn, Op2	Reverse Subtract,	$Rd \leftarrow Op2 - Rn$, RSBS updates N,Z,C,V

instrução	operando	Descrição	Ação executada
SBC, SBCS	{Rd,} Rn, Op2	Subtract with Carry,	$Rd \leftarrow Rn - Op2 - NOT(Carry)$, updates NZCV
SBFX	Rd, Rn, #lsb, #width	Signed Bit Field Extract,	$Rd[(width-1):0] = Rn[(width+lsb-1):lsb]$, $Rd[31:width] = Replicate(Rn[width+lsb-1])$
SDIV	{Rd,} Rn, Rm	Signed Divide,	$Rd \leftarrow Rn / Rm$
SEV	-	Send Event	
SMLAL	RdLo, RdHi, Rn, Rm	Signed Multiply with Accumulate,	$RdHi, RdLo \leftarrow signed(RdHi, RdLo + Rn * Rm)$
SMULL	RdLo, RdHi, Rn, Rm	Signed Multiply,	$RdHi, RdLo \leftarrow signed(Rn * Rm)$
SSAT	Rd, #n, Rm{,shift #s}	Signed Saturate,	$Rd \leftarrow SignedSat((Rm \text{ shift } s), n)$. Update Q
STM	Rn{!}, reglist	Store Multiple Registers	
STMDB, STMEA	Rn{!}, reglist	Store Multiple Regs Decrement Before	
STMFD, STMIA	Rn{!}, reglist	Store Multiple Regs Increment After	
STR	Rt, [Rn, #offset]	Store Register with Word,	$mem[Rn+offset] = Rt$
STRD	Rt, Rt2, [Rn, #offset]	Store Register with two Words,	$mem[Rn+offset] = Rt$, $mem[Rn+offset+4] = Rt2$
STREX	Rd, Rt, [Rn, #offset]	Store Register Exclusive if allowed, .	$mem[Rn + offset] \leftarrow Rt$, clear exclusive tag, $Rd \leftarrow 0$. Else $Rd \leftarrow 1$
STREXB	Rd, Rt, [Rn]	Store Register Exclusive Byte,	$mem[Rn] \leftarrow Rt[15:0]$ or $mem[Rn] \leftarrow Rt[7:0]$, clear exclusive tag, $Rd \leftarrow 0$. Else $Rd \leftarrow 1$
STREXH	Rd, Rt, [Rn]	Store Register Exclusive Half-word,	$mem[Rn] \leftarrow Rt[15:0]$ or $mem[Rn] \leftarrow Rt[7:0]$, clear exclusive tag, $Rd \leftarrow 0$. Else $Rd \leftarrow 1$
STRH, STRHT	Rt, [Rn, #offset]	Store Half-word,	$mem[Rn + offset] \leftarrow Rt[15:0]$
STRT	Rt, [Rn, #offset]	Store Register with Translation,	$mem[Rn + offset] = Rt$
SUB, SUBS	{Rd,} Rn, Op2	Subtraction,	$Rd \leftarrow Rn - Op2$, SUBS updates N,Z,C,V
SUB, SUBS	{Rd,} Rn, #imm12	Subtraction,	$Rd \leftarrow Rn - imm12$, SUBS updates N,Z,C,V
SVC	#imm	Supervisor Call	
SXTB	{Rd,} Rm {,ROR #n}	Sign Extend Byte,	$Rd \leftarrow SignExtend((Rm \text{ ROR } (8*n))[7:0])$
SXTH	{Rd,} Rm {,ROR #n}	Sign Extend Half-word,	$Rd \leftarrow SignExtend((Rm \text{ ROR } (8*n))[15:0])$

instrução	operando	Descrição	Ação executada
TBB	[Rn, Rm]	Table Branch Byte	$PC \leftarrow PC + \text{ZeroExtend}(\text{Memory}(\text{Rn} + \text{Rm}, 1) \ll 1)$
TBH	[Rn, Rm, LSL #1]	Table Branch Halfword	$PC \leftarrow PC + \text{ZeroExtend}(\text{Memory}(\text{Rn} + \text{Rm} \ll 1, 2) \ll 1)$
TEQ	Rn, Op2	Test Equivalence	Update N,Z,C,V on Rn EOR Operand2
TST	Rn, Op2	Test	Update N,Z,C,V on Rn AND Op2
UBFX	Rd, Rn, #lsb, #width	Unsigned Bit Field Extract	$Rd[(\text{width}-1):0] = Rn[(\text{width} + \text{lsb} - 1):\text{lsb}]$, $Rd[31:\text{width}] = \text{Replicate}(0)$
UDIV	{Rd,} Rn, Rm	Unsigned Divide,	$Rd \leftarrow Rn / Rm$
UMLAL	RdLo, RdHi, Rn, Rm	Unsigned Multiply with Accumulate,	$RdHi, RdLo \leftarrow \text{unsigned}(RdHi, RdLo + Rn * Rm)$
UMULL	RdLo, RdHi, Rn, Rm	Unsigned Multiply,	$RdHi, RdLo \leftarrow \text{unsigned}(Rn * Rm)$
USAT	Rd, #n, Rm{,shift #s}	Unsigned Saturate,	$Rd \leftarrow \text{UnsignedSat}((Rm \text{ shift } s), n)$, Update Q
UXTB	{Rd,} Rm {,ROR #n}	Unsigned Extend Byte,	$Rd \leftarrow \text{ZeroExtend}((Rm \text{ ROR } (8 * n))[7:0])$
UXTH	{Rd,} Rm {,ROR #n}	Unsigned Extend Halfword,	$Rd \leftarrow \text{ZeroExtend}((Rm \text{ ROR } (8 * n))[15:0])$
WFE	-	Wait For Event and Enter Sleep Mode	
WFI	-	Wait for Interrupt and Enter Sleep Mode	

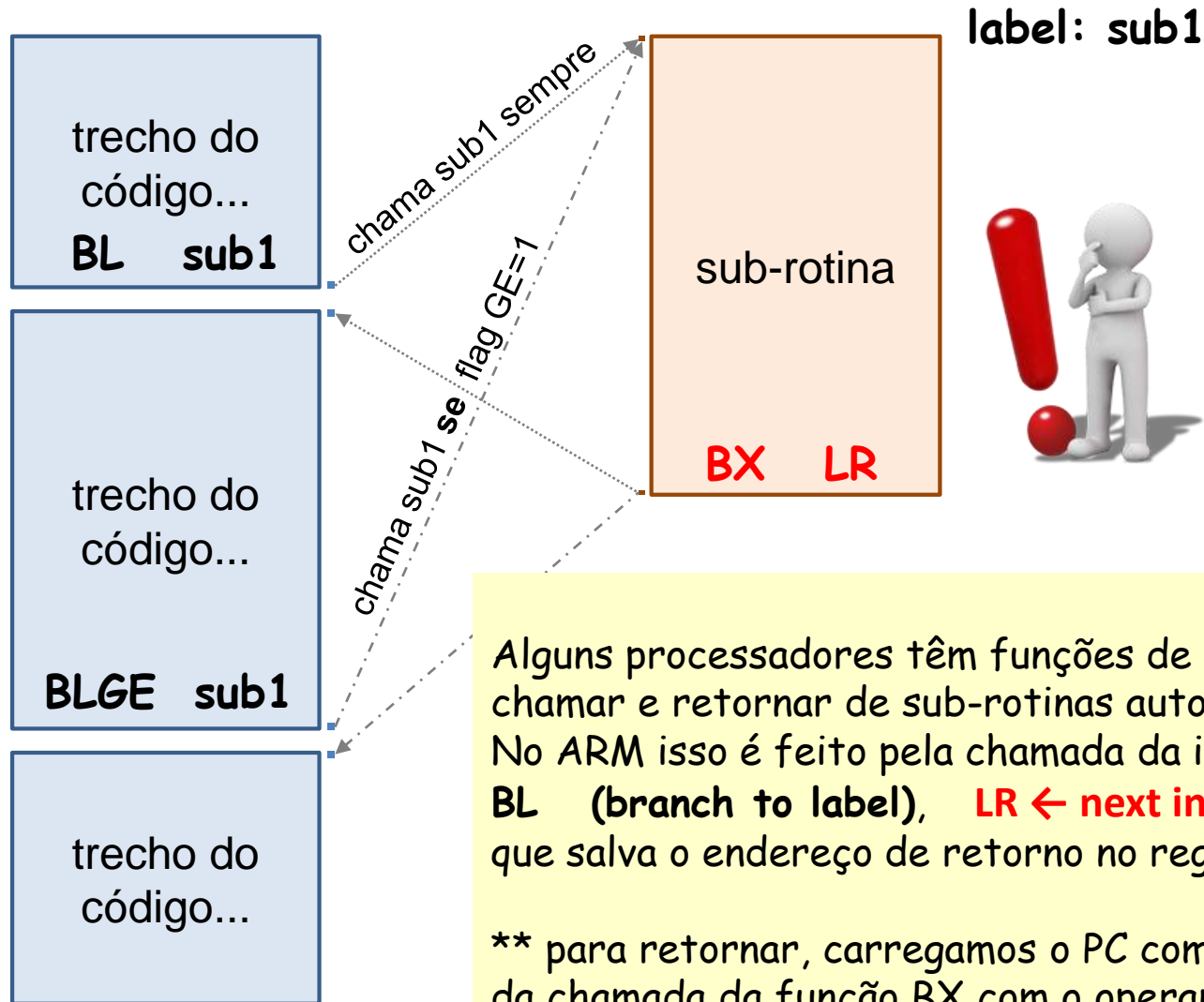
- Recordando o conceito de sub-rotina (ou chamada de função)



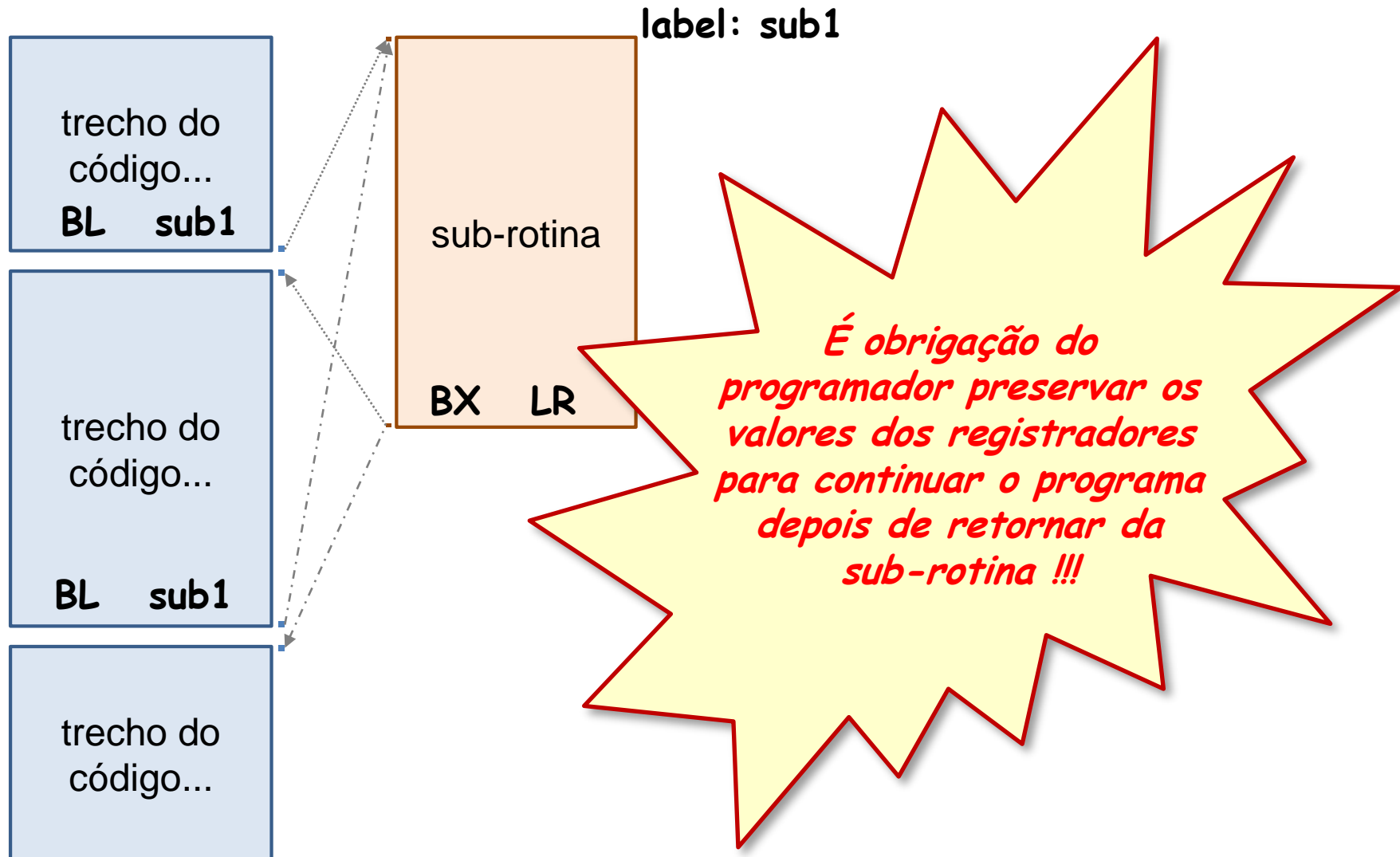
programas chamam funções (sub-rotinas) para executar algoritmos repetitivos

exemplos: calcular o seno de uma função, ordenar caracteres numa string, enviar um byte por uma porta de comunicação serial, ler um dado do conversor ADC, etc.

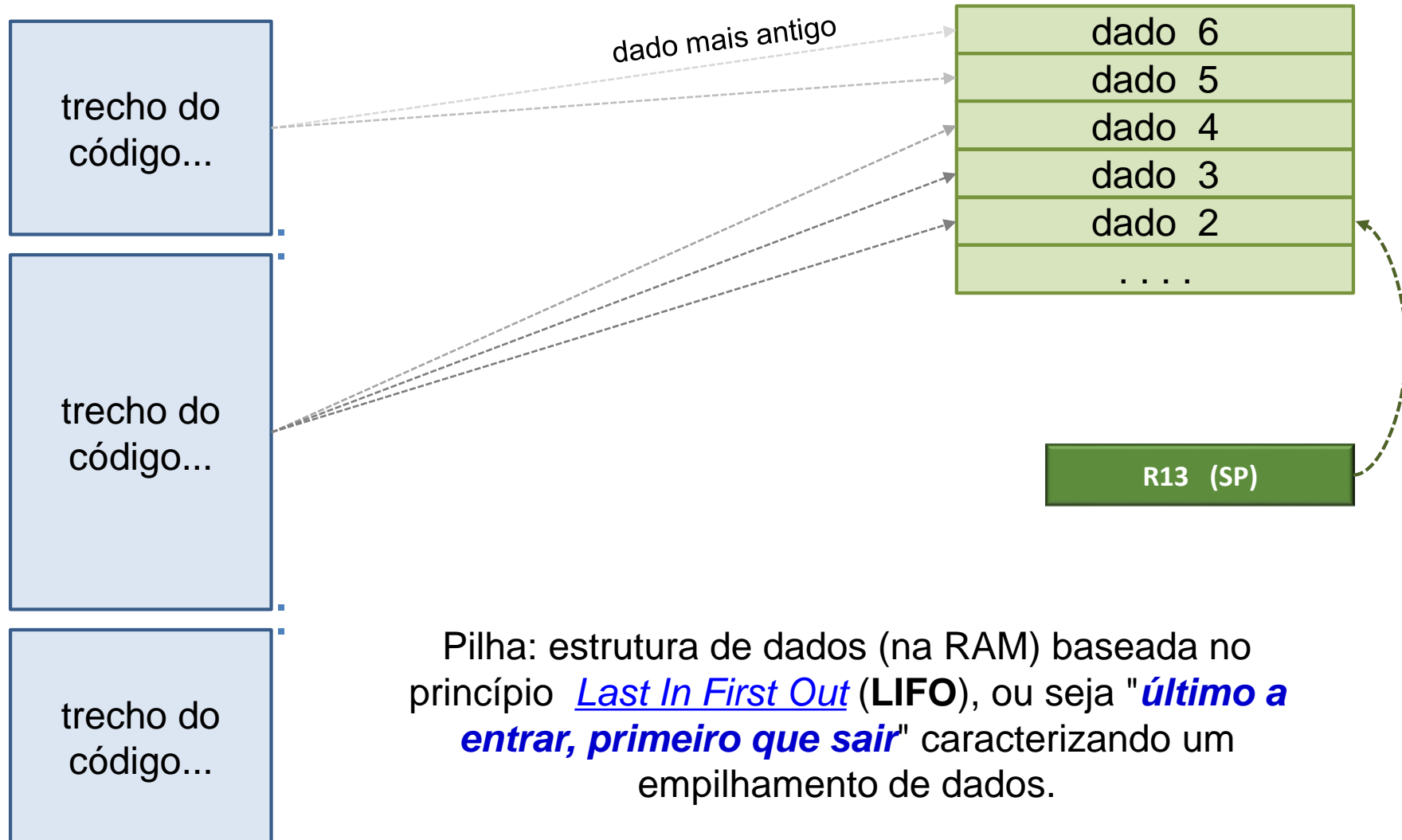
- Recordando o conceito de sub-rotina (ou chamada de função)



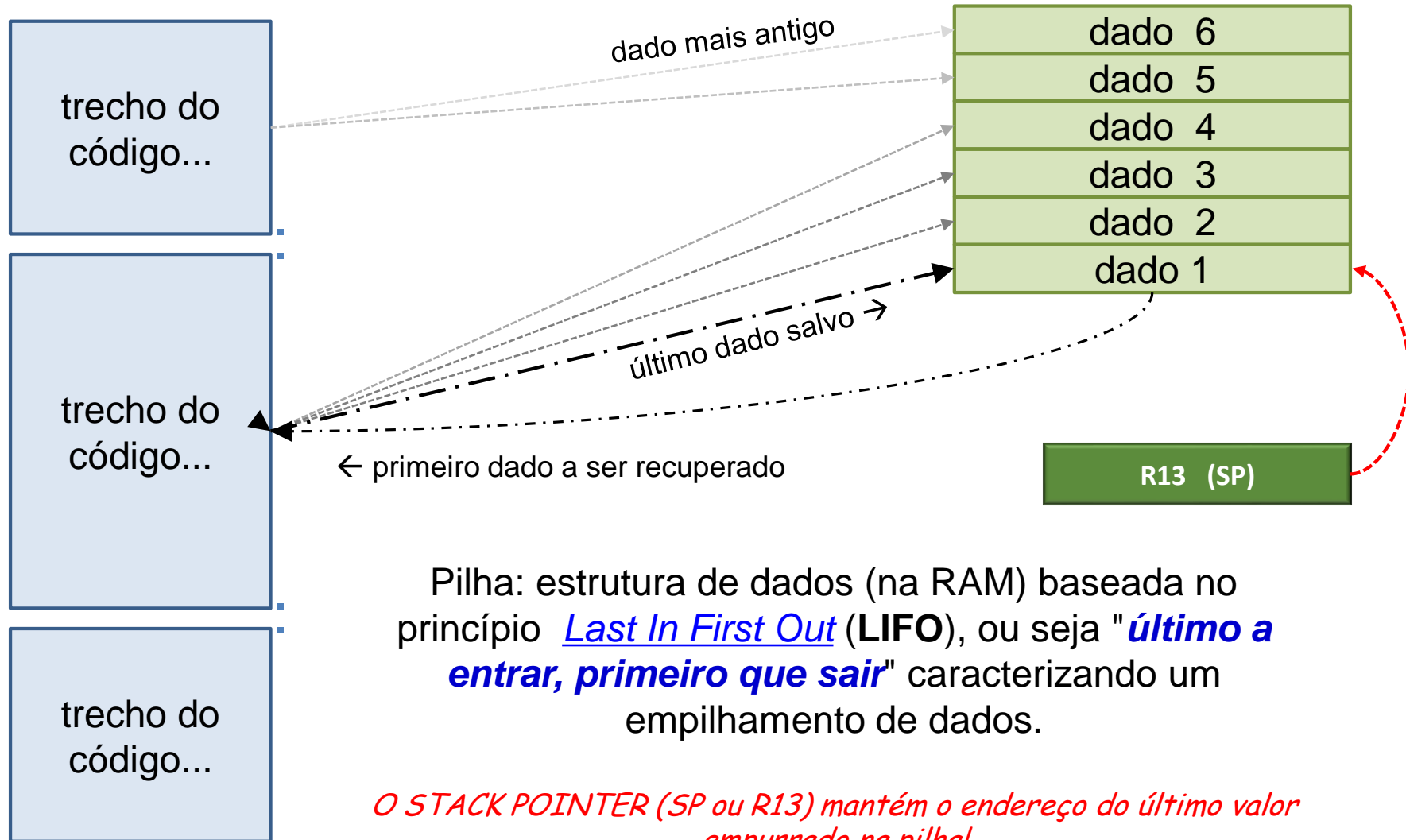
- Recordando o conceito de sub-rotina (ou chamada de função)



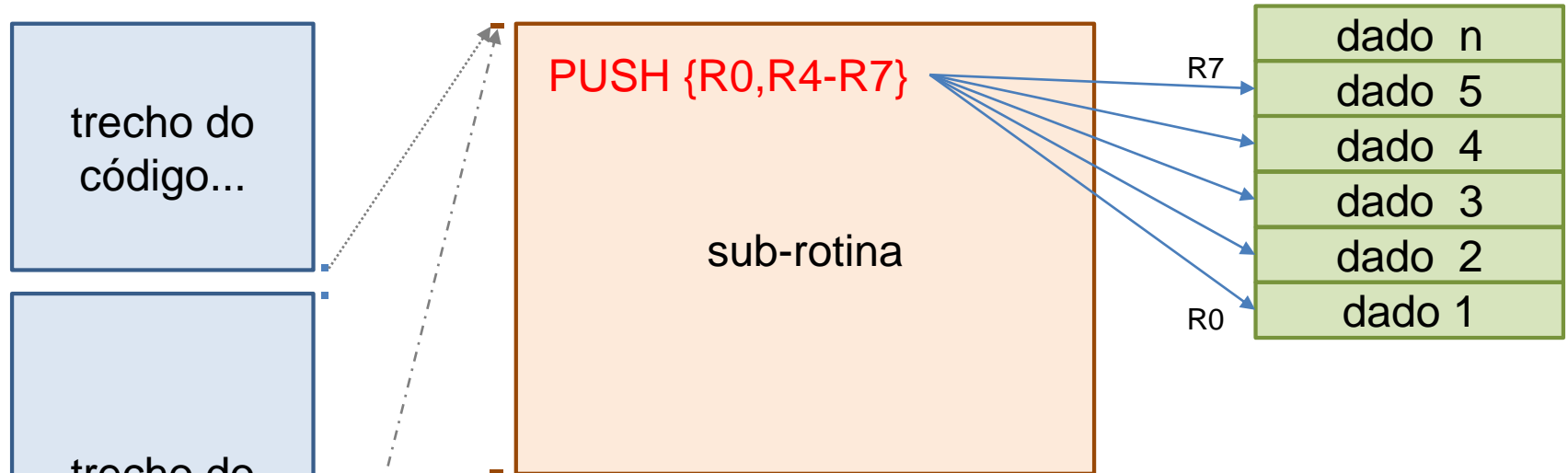
- Recordando o conceito de **PILHA** ... (como preservar conteúdo dos registradores)



- Recordando o conceito de **PILHA** ... (como preservar conteúdo dos registradores)



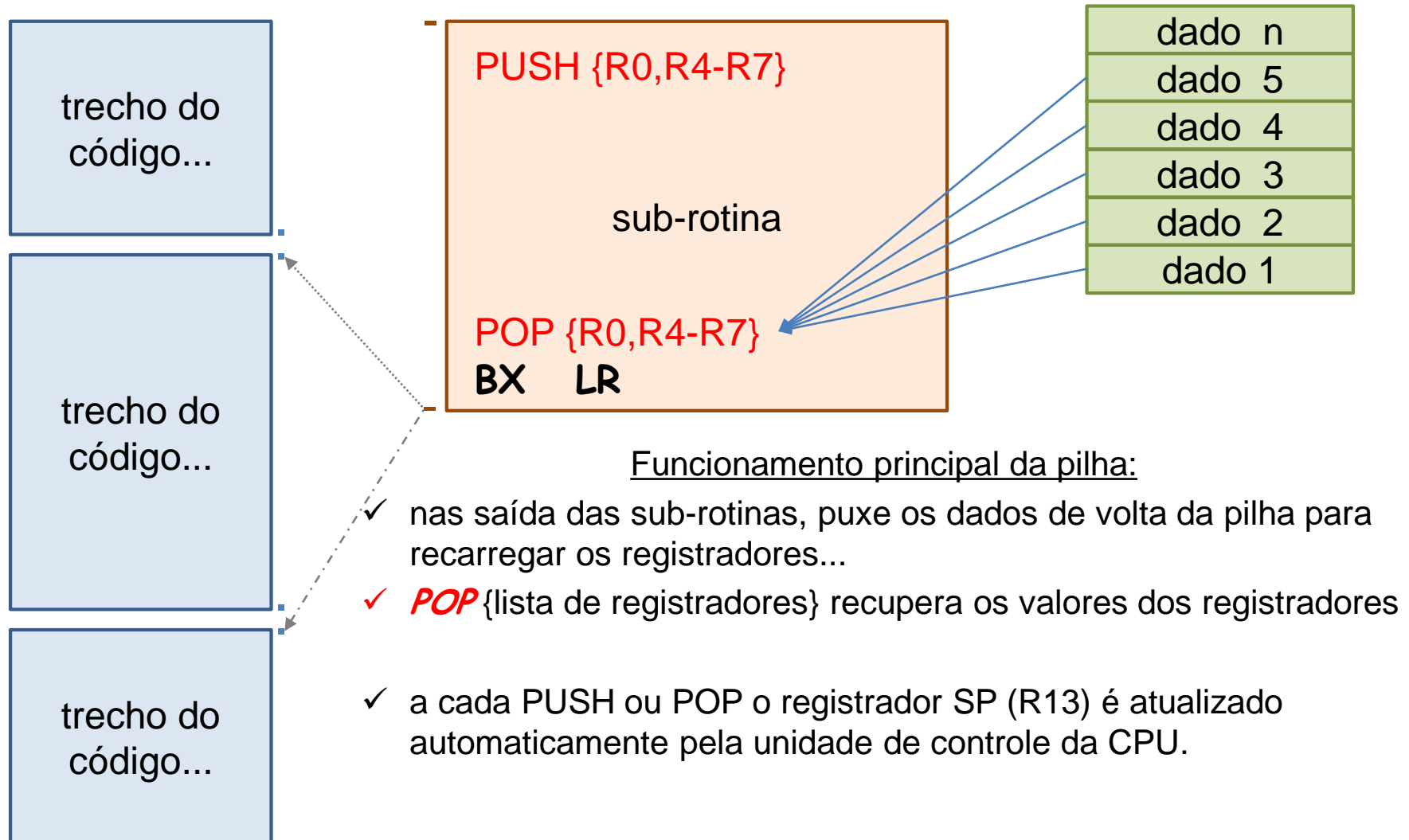
- Recordando o conceito de **PILHA** ... (como preservar conteúdo dos registradores)



Funcionamento principal da pilha:

- ✓ nas entradas das sub-rotinas, empurre para a pilha os valores dos registradores que serão modificados
- ✓ **PUSH**{lista de registradores} empurra dados na pilha...
- ✓ na saída (no retorno) da sub-rotina, recupere os valores dos registradores salvos na pilha.
- ✓ *a instrução segue a ordem e salva o registrador de menor índice na lista na menor posição de memória da pilha dentro dessa instrução...*

- Recordando o conceito de **PILHA** ... (como preservar conteúdo dos registradores)



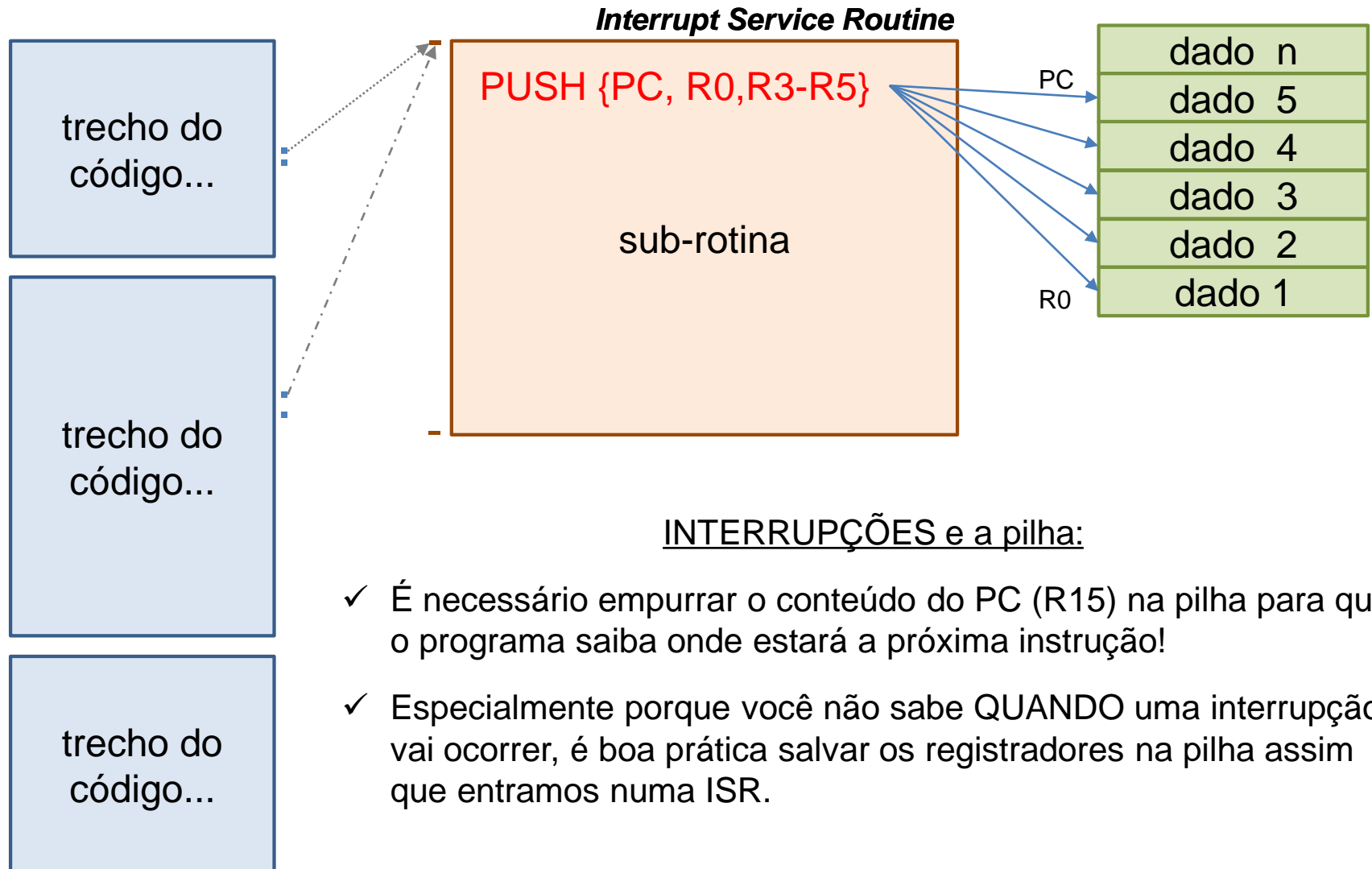
➤ Conceito de **INTERRUPÇÃO** ...

trecho do
código...

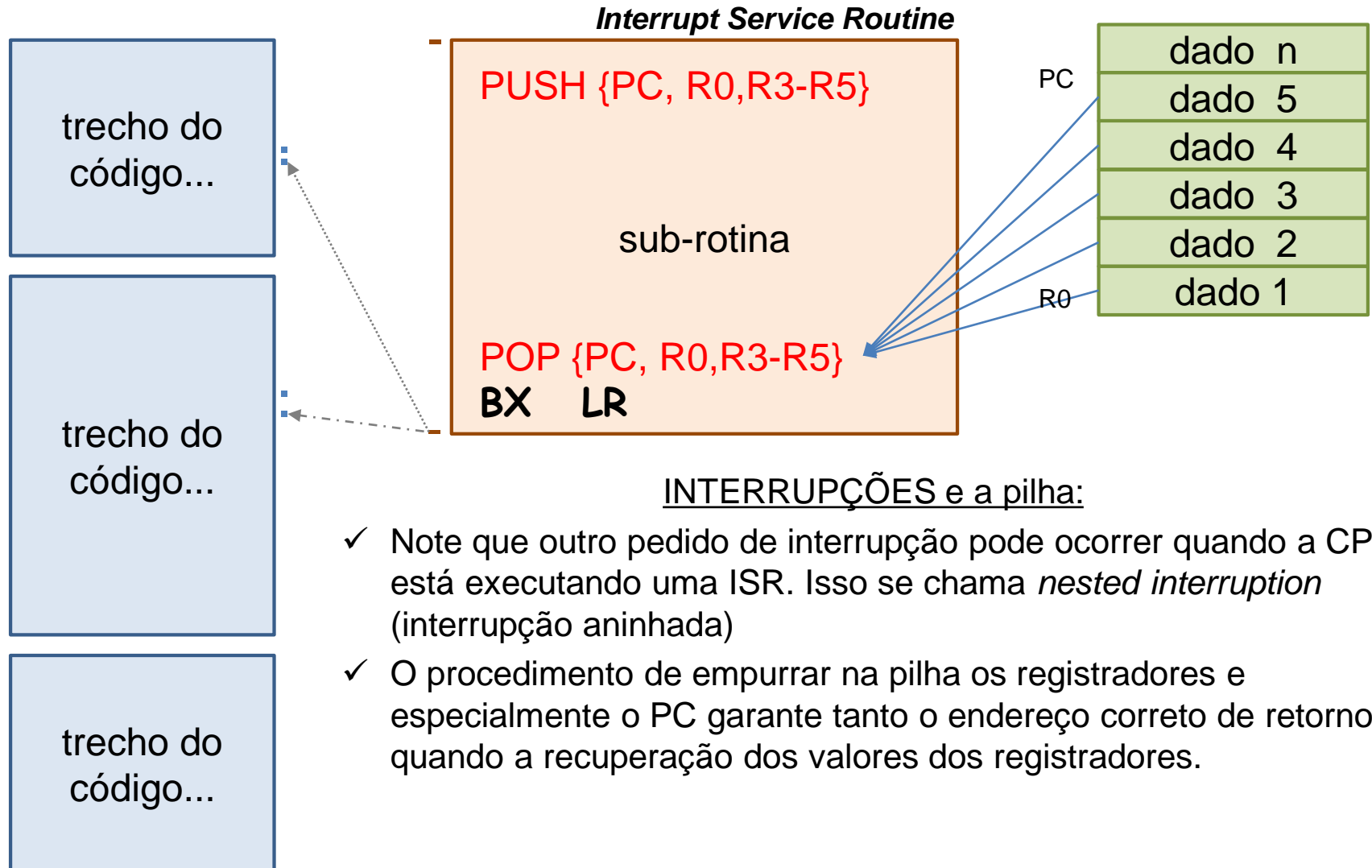
trecho do
código...

trecho do
código...

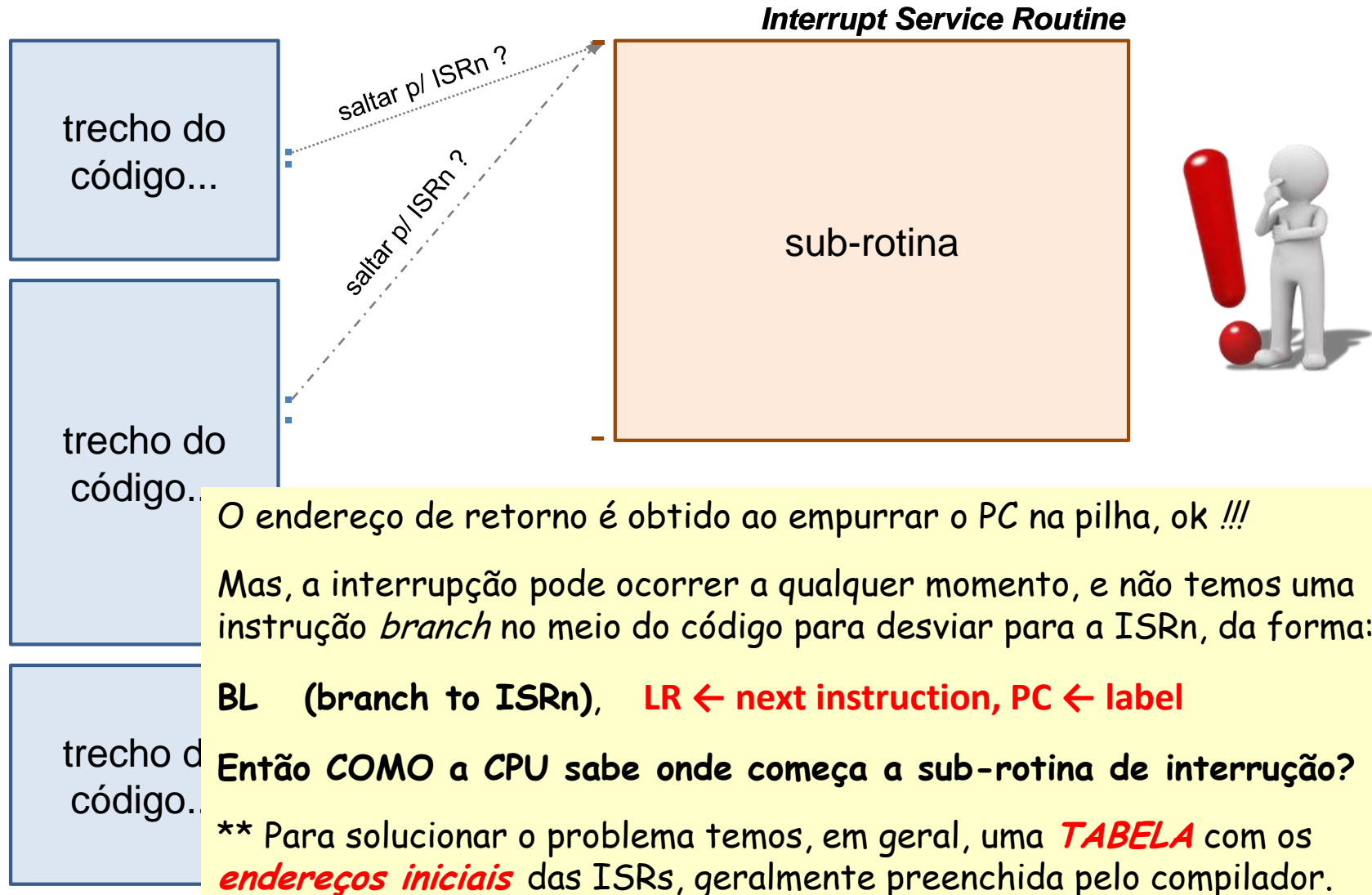
- Eventos no mundo real são imprevisíveis : você não sabe quando eles podem ocorrer...
- Fazer seu código verificar cada um dos eventos aos quais seu sistema vai responder não é boa tática!
- Melhor é que o mundo te avise que alguns eventos ocorreram e “desperte” certa parte do seu código para responder a esse evento
- Uma forma de o mundo avisar que um evento merece atenção do seu código é (entre outras táticas) por meio de **interrupções!**
- Para atender a uma interrupção você deve escrever um pedaço de código chamado “**INTERRUPT SERVICE ROUTINE**” ou ISR
- ISR funcionam como as sub-rotinas que vimos antes e podem ser disparadas por hardware ou por software.

➤ Conceito de **INTERRUPÇÃO** ...

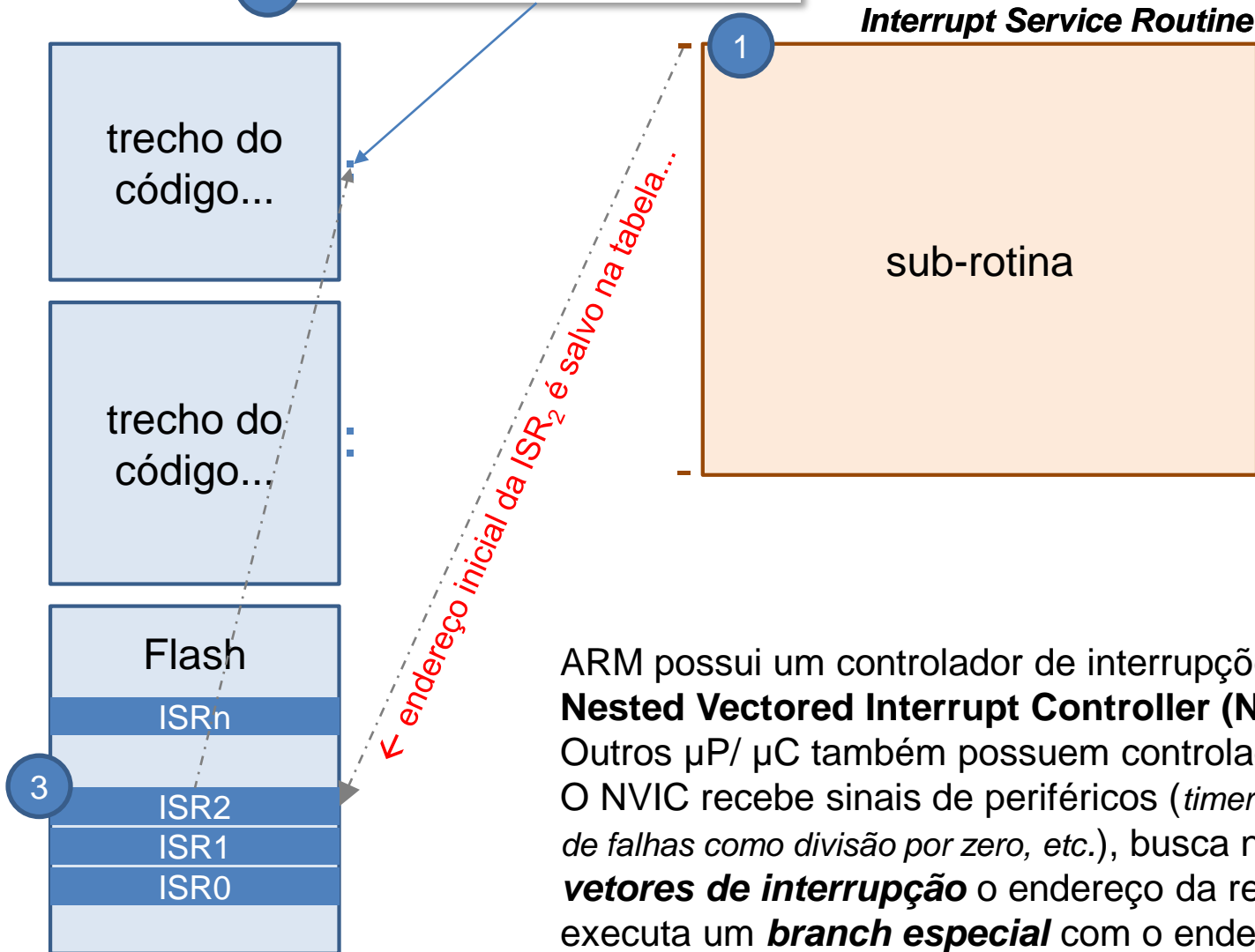
➤ Conceito de **INTERRUPÇÃO** ...



➤ Conceito de **INTERRUPÇÃO** ...



2 Aqui chega a requisição de Interrupção.
O NVIC busca endereço da ISR na tabela,
para saber como fazer o branching.



ARM possui um controlador de interrupções, chamado **Nested Vectored Interrupt Controller (NVIC)** .
Outros μP/ μC também possuem controlador similar...
O NVIC recebe sinais de periféricos (*timers, pinos GPIO, avisos de falhas como divisão por zero, etc.*), busca na **tabela de vetores de interrupção** o endereço da respectiva ISR_n, e executa um **branch especial** com o endereço inicial da ISR.

➤ Conceito de **INTERRUPÇÃO** ...

trecho do código...

trecho do código...

Flash
ISRn
ISR2
ISR1
ISR0

Interrupt Service Routine

sub-rotina

- Propriedades das INTERRUPÇÕES:
- Podem ser geradas por software ou hardware
- ISR são como sub-rotinas e funcionam ≈ da mesma forma
- Algumas interrupções podem ser mascaradas (ou seja, podemos impedir temporariamente que alguns pedidos de interrupções sejam feitos)
- Há ainda interrupções que **NÃO** podem ser mascaradas por serem cruciais para o funcionamento do sistema (ex: reset, DZ e outras)

➤ Exemplos da instrução MOV

0x0000001A	R0
0x00000A08	R1
0xFFFFFFFF7	R2
.....	R3
.....	R4
.....	R5
.....	R6
.....	R7
.....	R8
.....	R9
.....	R10
.....	R11
.....	R12
.....	R13
.....	R14
.....	R15

MOV R0, #0x1A

MOV R1, #2568

MVN R2, #8

faz o complemento-1 de 8 e carrega em R2

OBS: atualiza carry... uma soma com carry em seguida equivale à subtração em complemento-2

➤ Exemplos da instrução MOV

0x0000001A	R0
0x00000A08	R1
0xFFFFFFFF7	R2
0xCADEFABC	R3
.....	R4
.....	R5
.....	R6
.....	R7
.....	R8
.....	R9
.....	R10
.....	R11
.....	R12
.....	R13
.....	R14
.....	R15

```
MOV    R0, #0x1A
```

```
MOV    R1, #2568
```

```
MVN    R2, #8
```

```
MVW    R3, #0xFABC
```

```
MVT    R3, #0xCADE
```

Para carregar 32 bits num registrador é preciso usar 2 instruções: MVW e MVT !
(não existem nos μC Cortex M0, M0+ e M1)

➤ Exemplos da instrução MOV

```
MOV    R0, #0x1A
```

```
MOV    R1, #2568
```

```
MVN    R2, #8
```

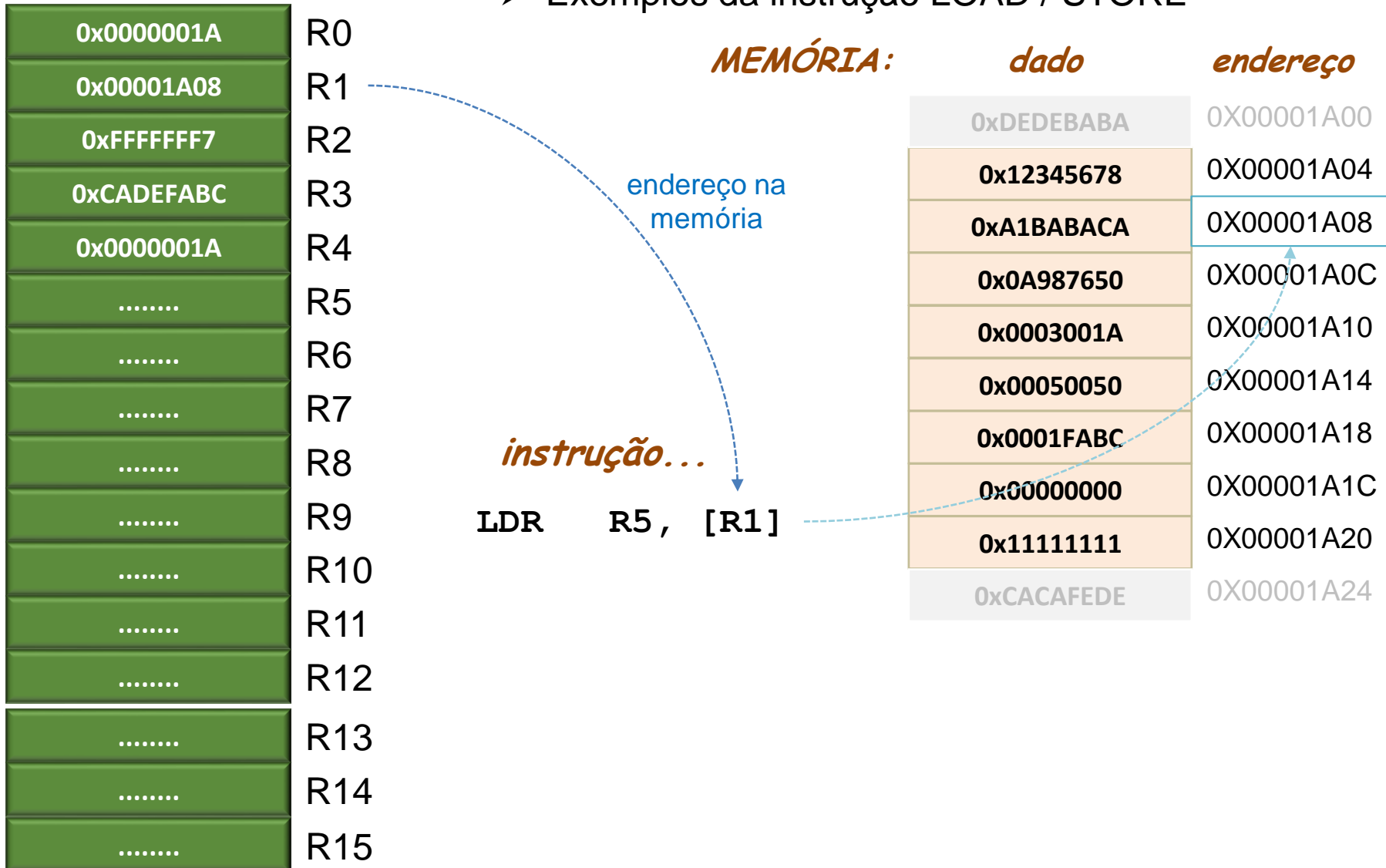
```
MVW    R3, #0xFABC
```

```
MVT    R3, #0xCADE
```

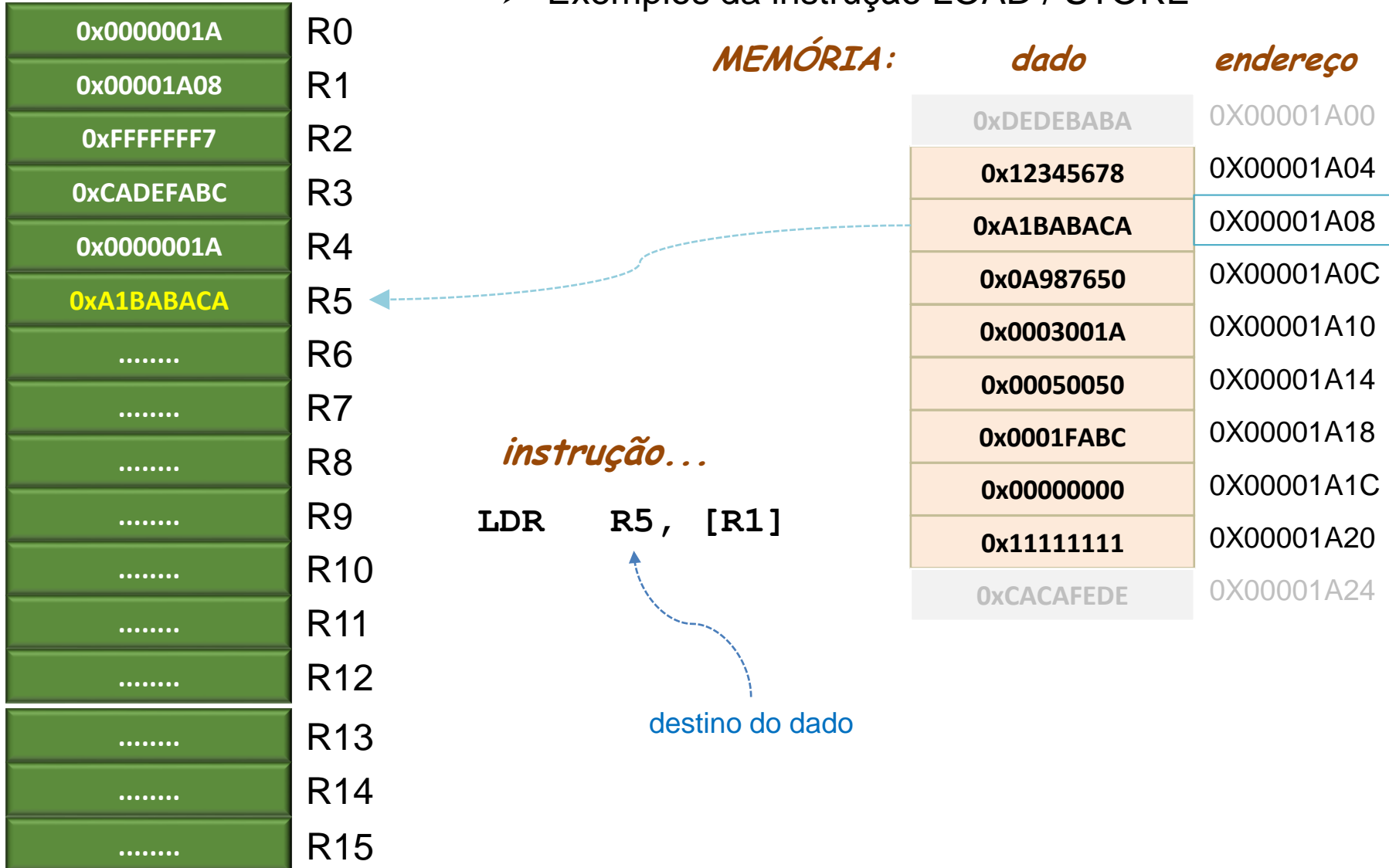
```
MVT    R4, R0
```

0x0000001A	R0
0x00000A08	R1
0xFFFFFFFF7	R2
0xCADEFABC	R3
0x0000001A	R4
.....	R5
.....	R6
.....	R7
.....	R8
.....	R9
.....	R10
.....	R11
.....	R12
.....	R13
.....	R14
.....	R15

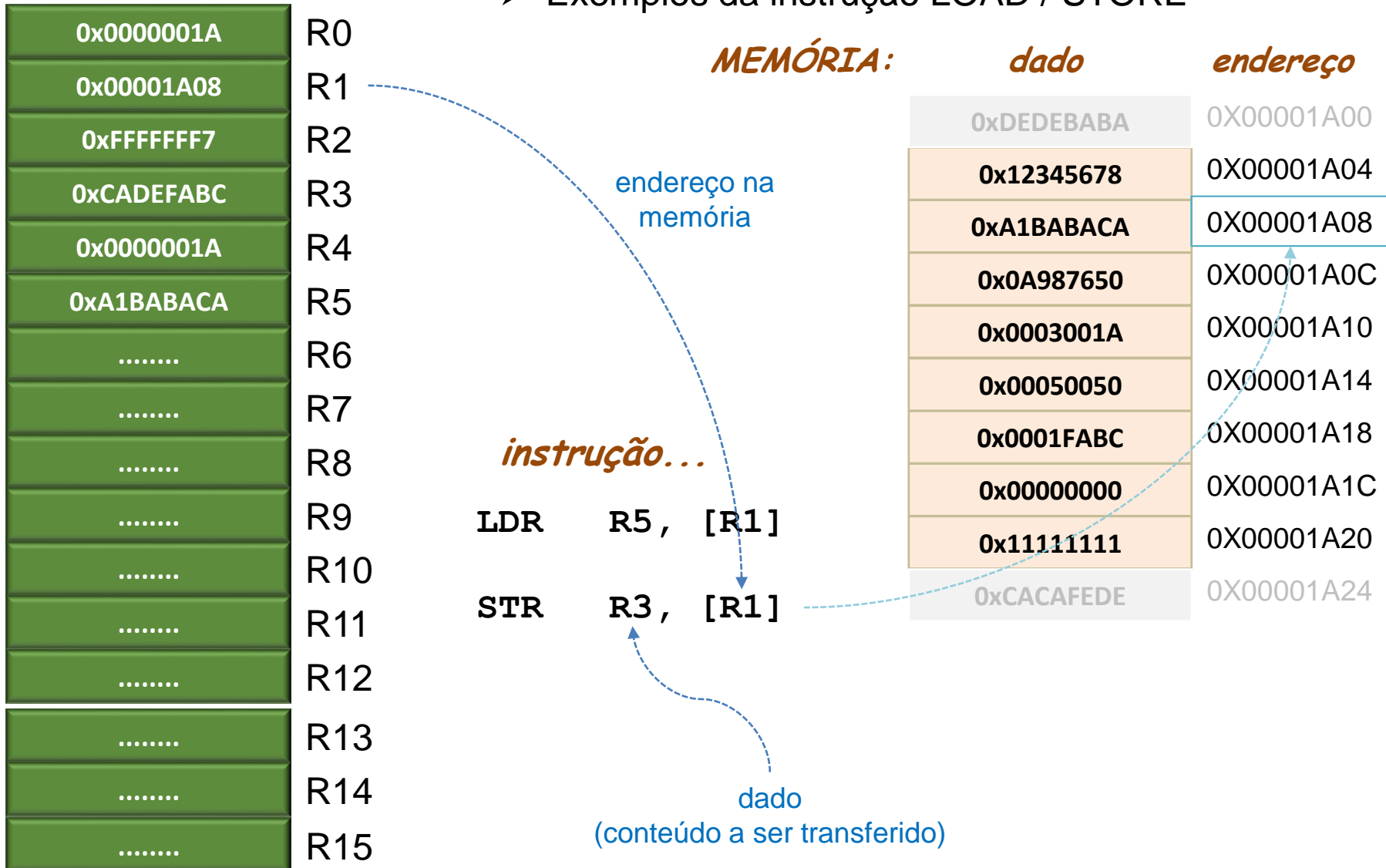
➤ Exemplos da instrução LOAD / STORE



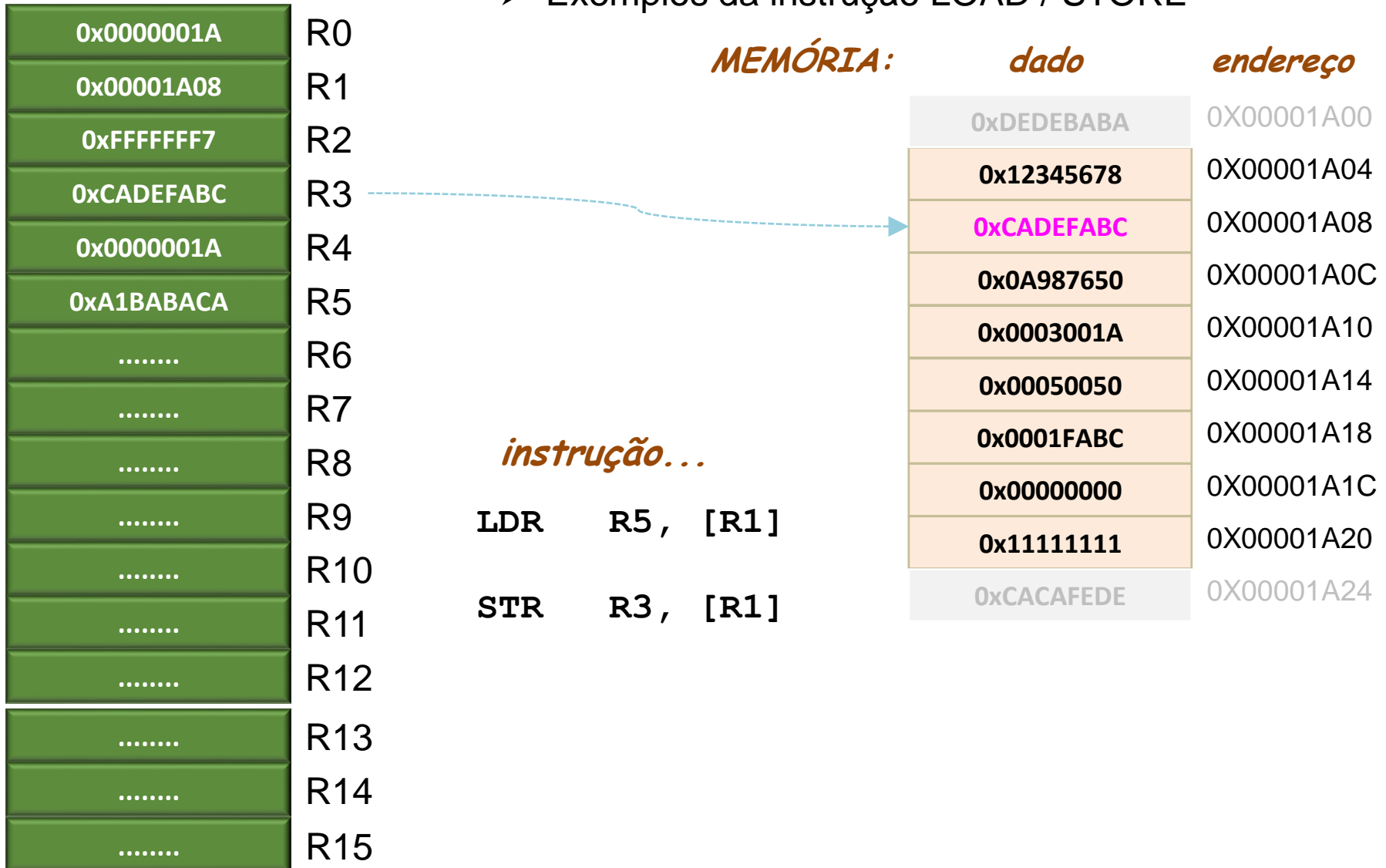
➤ Exemplos da instrução LOAD / STORE



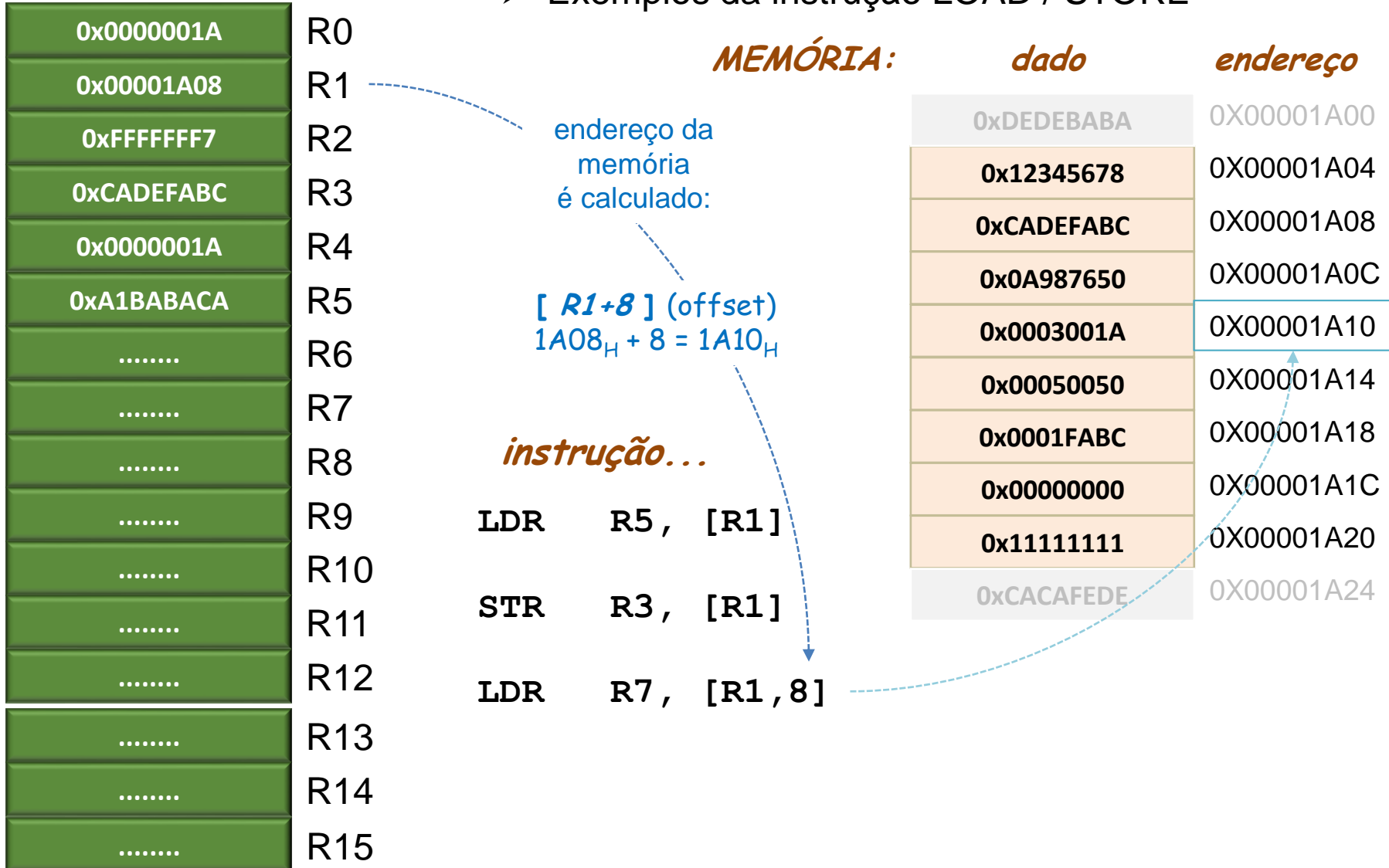
➤ Exemplos da instrução LOAD / STORE



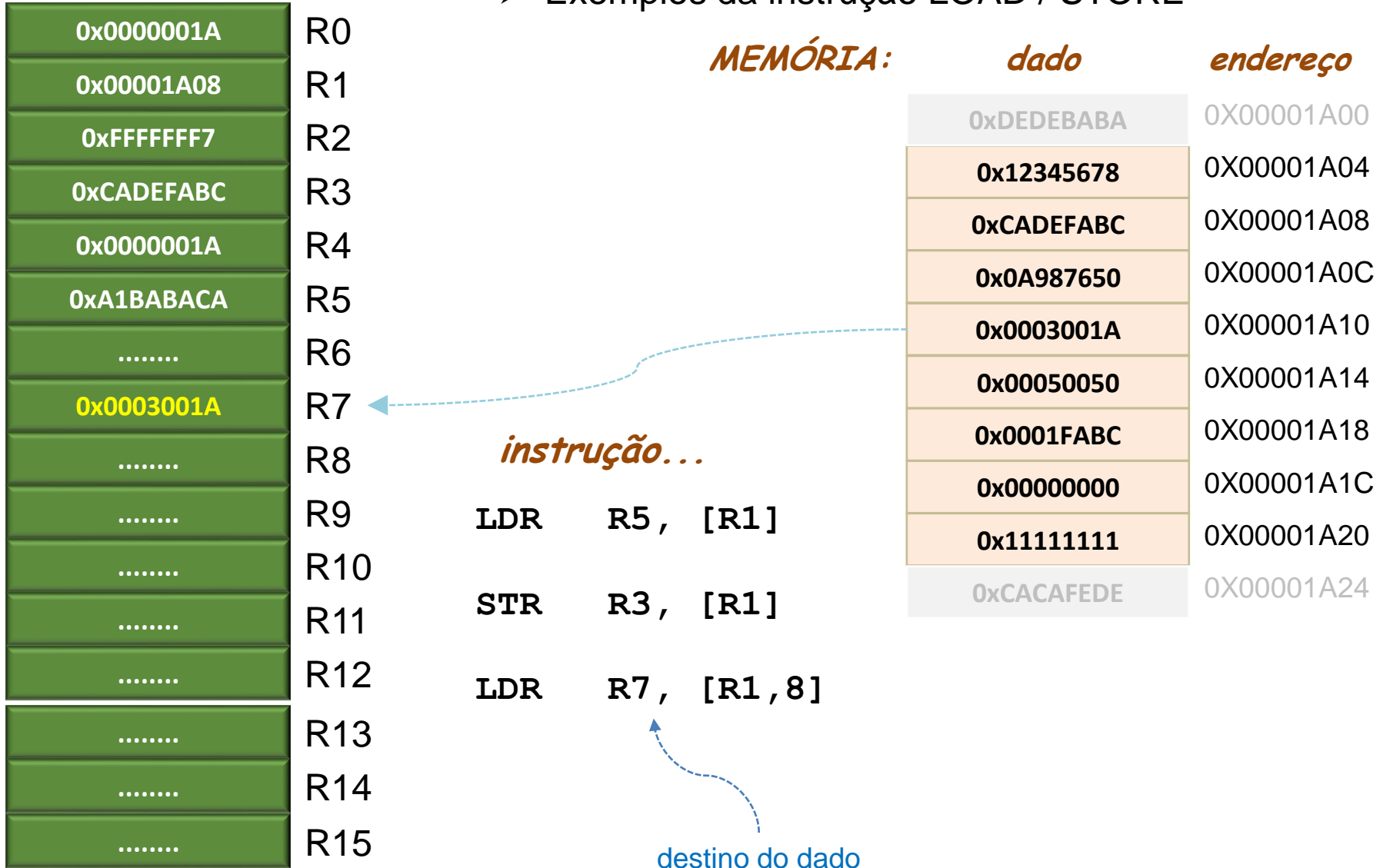
➤ Exemplos da instrução LOAD / STORE



➤ Exemplos da instrução LOAD / STORE



➤ Exemplos da instrução LOAD / STORE



➤ Exemplos da instrução LOAD / STORE

0x0000001A
0x00001A08
0xFFFFFFFF7
0xCADEFABC
0x0000001A
0xA1BABACA
.....
0x0003001A
.....
.....
.....
.....
.....
.....
.....
.....
.....

R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13
R14
R15

Note que nessas instruções **R1** funciona como um apontador (pointer) e não foi modificado; embora exista variações das instruções LDR e STR que alteram o valor dos apontadores!



instrução...

LDR R5 , [R1]

STR R3 , [R1]

LDR R7 , [R1 , 8]

dado

endereço

0xDEDEBABA

0X00001A00

0x12345678

0X00001A04

0xCADEFABC

0X00001A08

0x0A987650

0X00001A0C

0x0003001A

0X00001A10

0x00050050

0X00001A14

0x0001FABC

0X00001A18

0x00000000

0X00001A1C

0x11111111

0X00001A20

0xCACAFEDE

0X00001A24

➤ Diretivas para compilador ASSEMBLY

Em um arquivo definimos a diretiva (uma constante)

```
.equ KTE1, 0x87654321
```

Noutro arquivo, usamos a kte numa função...

@ uma certa sub-rotina começa aqui...

...

```
LDR    R0, =KTE1
```

@ outras linhas do código...

...

@ final da sub-rotina...

```
.word 0x87654321
```

0x0000001A	R0
0x00001A08	R1
0xFFFFFFFF7	R2
0xCADEFABC	R3
0x0000001A	R4
0xA1BABACA	R5
.....	R6
0x0003001A	R7
.....	R8
.....	R9
.....	R10
.....	R11
.....	R12
.....	R13
.....	R14
.....	R15

➤ Diretivas para compilador ASSEMBLY

.equ KTE1, 0x87654321

sub-rotina (função)

@ uma certa sub-rotina começa aqui...

...
LDR R0, =KTE1

@ outras linhas do código...

...

@ depois do final da sub-rotina...

.word 0x87654321

0x	0X000F1000
0x4806	0X000F1004
0x	0X000F1008
0x	0X000F100C
0x	0X000F1010
0x	0X000F1014
0x	0X000F1018
0x	0X000F101C
0x87654321	0X000F1020
0xCACAFEDE	0X000F1024

Valor da constante

final da sub-rotina...

0x0000001A	R0
0x00001A08	R1
0xFFFFFFFF7	R2
0xCADEFABC	R3
0x0000001A	R4
0xA1BABACA	R5
.....	R6
0x0003001A	R7
.....	R8
.....	R9
.....	R10
.....	R11
.....	R12
.....	R13
.....	R14
0X000F1004	R15

➤ Diretivas para compilador ASSEMBLY

```
.equ KTE1, 0x87654321
```

sub-rotina (função)

distância ao final da
sub-rotina não muda...

@ uma certa sub-rotina começa aqui...

```
...  
LDR R0, [PC, 24]
```

@ outras linhas do código...

...

@ depois do final da sub-rotina...

```
.word 0x87654321
```

0x	0X000F1000
0x4806	0X000F1004
0x	0X000F1008
0x	0X000F100C
0x	0X000F1010
0x	0X000F1014
0x	0X000F1018
0x	0X000F101C
0x87654321	0X000F1020
0xCACAFEDE	0X000F1024

Valor da constante

O compilador usa um offset (**24**) relativo ao PC [**R15**], porque se a sub-rotina for recompilada em outro endereço da memória, o acesso à KTE1 permanece.

➤ Diferenças entre MOV e LDR/STR

0x0000001A	R0
0x00001A08	R1
0xFFFFFFFF7	R2
0xCADEFABC	R3
0x0000001A	R4
0xA1BABACA	R5
.....	R6
0x0003001A	R7
.....	R8
.....	R9
.....	R10
.....	R11
.....	R12
.....	R13
.....	R14
.....	R15

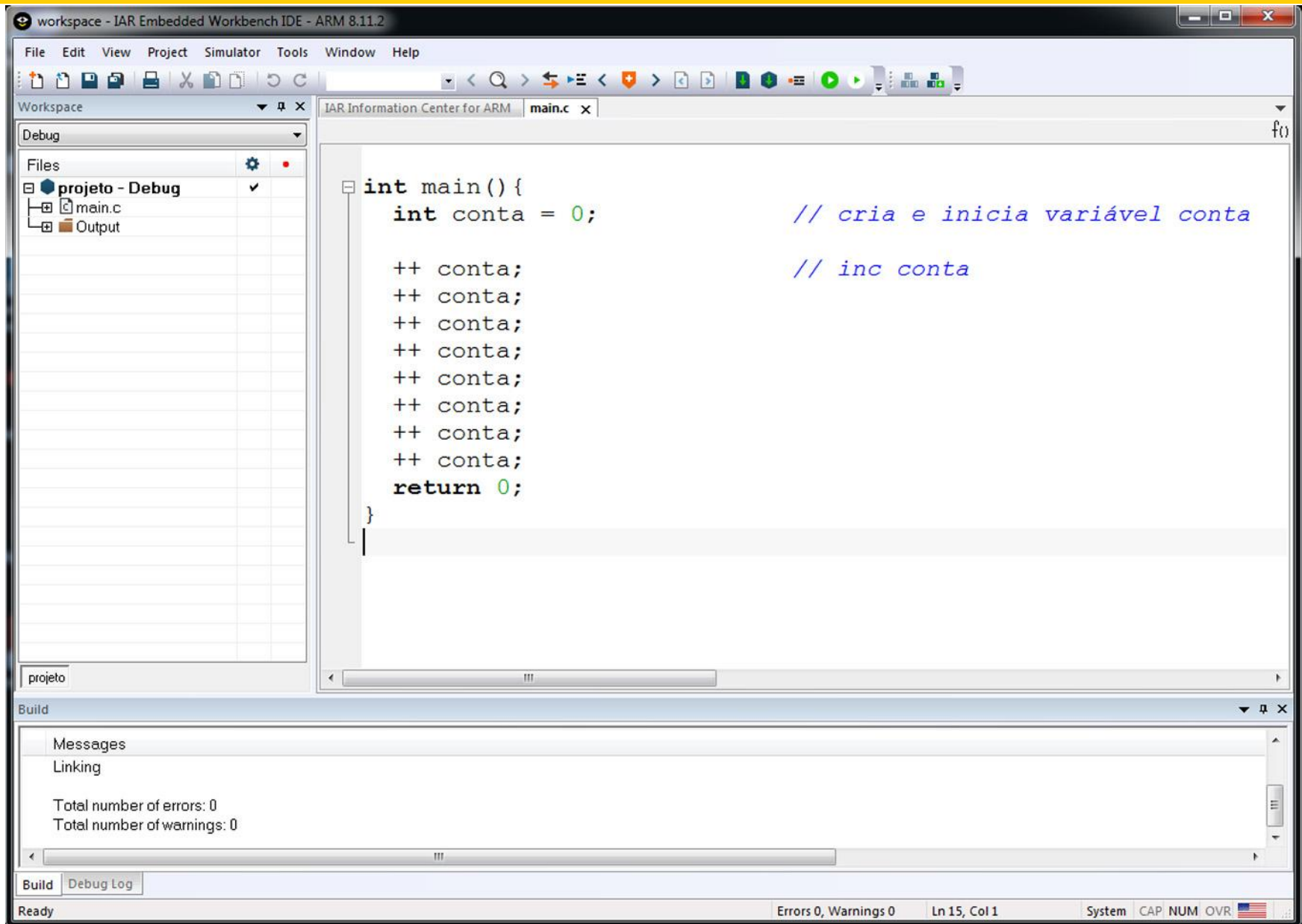
R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13
R14
R15

- Move é usado geralmente para deslocar conteúdos entre registradores (interno à CPU) ou valores imediatos (kts)
- LDR e STR são usadas para mover conteúdo entre a CPU e a memória (RAM ou ROM)

0x	0X000F1000
0x4806	0X000F1004
0x	0X000F1008
0x	0X000F100C
0x	0X000F1010
0x	0X000F1014
0x	0X000F1018
0x	0X000F101C
0x87654321	0X000F1020
0xCACAFEDE	0X000F1024

- Ao acessar a memória deve-se definir um endereço, que estará em um **registrador** (direto), ou será calculado a partir do conteúdo de um **registrador + offset** !

0x0000001A	R0	➤ Instruções ARITMÉTICAS e LÓGICAS ...	
0x00001A08	R1	➤ Nenhuma instrução aritmética e lógica ocorre diretamente na memória – sempre os operadores são registradores	
0xFFFFFFFF7	R2		
0xCADEFABC	R3		
0x0000001A	R4		
0xA1BABACA	R5		
.....	R6	SUB R1, R2, #5	// lê-se: r1 = r2-5
0x0003001A	R7	ADD R2, R3, R3 LSL #2	// lê-se: r2 = r3+(r3*4)
.....	R8		
.....	R9	ADDEQ R5, R5, R6	// se (flag EQ) r5 = r5+r6
.....	R10	AND R4, R3, R7	// lê-se: r4 = r3(i) AND r7(i)
.....	R11		
.....	R12		
.....	R13		
.....	R14		
.....	R15		



workspace - IAR Embedded Workbench IDE - ARM 8.11.2

File Edit View Project Simulator Tools Window Help

Workspace

Debug

Files

- projeto - Debug
 - main.c
 - Output

```
int main(){  
    int conta = 0;           // cria e inicia variável conta  
  
    ++ conta;                // inc conta  
    ++ conta;  
    ++ conta;  
    ++ conta;  
    ++ conta;  
    ++ conta;  
    ++ conta;  
    ++ conta;  
    return 0;  
}
```

projeto

Build

Messages

Linking

Total number of errors: 0
Total number of warnings: 0

Build Debug Log

Ready

Errors 0, Warnings 0 Ln 15, Col 1 System CAP. NUM. OVR

- Instruções do ARM podem ser 32 bits ou 16 bits (*thumb*)
- Instruções possuem 1, 2 ou 3 operandos
- As instruções MOV são geralmente usadas para mover dados entre registradores ou com valores imediatos
- Operações com a memória usam as instruções LOAD ou STORE e os registradores funcionam como apontadores de endereços de memória
- Compiladores podem usar offset relativo ao PC para armazenar constantes ao final de sub-rotinas para facilitar
- Todas as operações aritméticas/lógicas usam registradores como operandos
- Não há operações aritméticas e lógicas direto na memória
- Chamadas de sub-rotinas são feitas com *branching* (BL e BX), que guardam automaticamente o endereço de retorno em LR e depois retornam ao PC
- É dever do programador preservar conteúdo dos registradores nas chamadas de sub-rotinas, quando for necessário
- ARM busca instruções (*fetch*) ao mesmo tempo em que decodifica e executa as instruções em andamento – processo chamado pipeline.

- As *instruções* da arquitetura ARM são executadas em *pipeline*.
- Toda instrução tem um código (que define a ação), e operandos (usados na operação)

Opcode DestReg, Operand2

Opcode DestReg, SrcReg, Operand2

SUB R1, R2, #5

// lê-se: $r1 = r2 - 5$

ADD R2, R3, R3 LSL #2

// lê-se: $r2 = r3 + (r3 * 4)$

ADDEQ R5, R5, R6

// se (flag EQ) $r5 = r5 + r6$

LDR R0, [R1]

// $r0 = \text{val mem cujo end está em } r1$ (pointer *r1)

MOV R0, #0x101

// $r0 = \text{valor imediato (101 hexa) = 257 decimal}$

STRNEB R2, [R3, R4]

// se (flag NEQ) o endereço $*(r3+r4) = r2$

- Não vamos nos fixar em desenvolver programas em linguagem assembly ARM...
 - arquiteturas modernas, como ARM, focam em programação em “C”
 - isso permite que o desenvolvedor “migre” de um processador para outro
 - Isso melhora a manutenção e escalabilidade dos projetos de eletrônica embarcada...

- Vamos focar (nas aulas teóricas) em compreender os periféricos agregados ao ARM...
 - coprocessadores e circuitos de arbitragem de barramento (DMA, INTERRUPÇÃO, etc)
 - periféricos de comunicação serial (SPI, I2C, USART, CAN, etc...)
 - periféricos de interface geral (GPIO)
 - periféricos de conversão (ADC e DAC) e de saída analógica (PWM)
 - periféricos temporizadores (Watchdog, RTC, timers de uso geral, etc)