

Capítulo 3

Linguagem de Montagem

Autora: Wu Shin-Ting

Neste capítulo vamos ver como se escreve um programa utilizando uma linguagem de programação próxima aos códigos binários, de forma que um processador conseguiria decodificar imediatamente sem recorrer a um compilador. Esta linguagem é denominada **linguagem de montagem** (*assembly*).

Embora os códigos em **linguagem de alto nível**, como Fortran, C e Pascal, sejam mais inteligíveis, portáteis e independentes da arquitetura do processador, os **códigos de máquina**, por refletirem diretamente a arquitetura do processador e serem inteligíveis para dispositivos computacionais, permitem os programadores fazerem uma análise mais precisa do desempenho dos códigos e ter um controle maior na sua otimização. Portanto, os códigos de máquina são ainda altamente recomendáveis para aplicações com restrições de desempenho críticas tanto em relação à memória quanto em relação ao tempo de execução.

No entanto, os códigos (binários) de máquina, como os mostrados na aba “*Memory*” da Figura 1, são difíceis de entender e um programa desenvolvido com base nestes códigos são susceptíveis a muitos tipos de erros. Uma solução é renderizar os códigos binários em códigos simbólicos mais inteligíveis, como os mostrados na aba “*Disassembly*” da Figura 1. O conjunto destes códigos simbólicos, ou mnemônicos, constitui a linguagem de montagem. Sendo os códigos de máquina uma linguagem da máquina, eles são altamente dependentes da arquitetura do processador. O processador integrado ao nosso microcontrolador é o processador ARM Cortex-M0+, da arquitetura ARM. Ele suporta códigos de máquina de 16 bits (*Thumb*) e de 32 bits (*Thumb-2*). Nesta disciplina trabalharemos com o repertório de instruções *Thumb* [1]. O montador disponível no IDE CodeWarrior é o montador GNU [2].

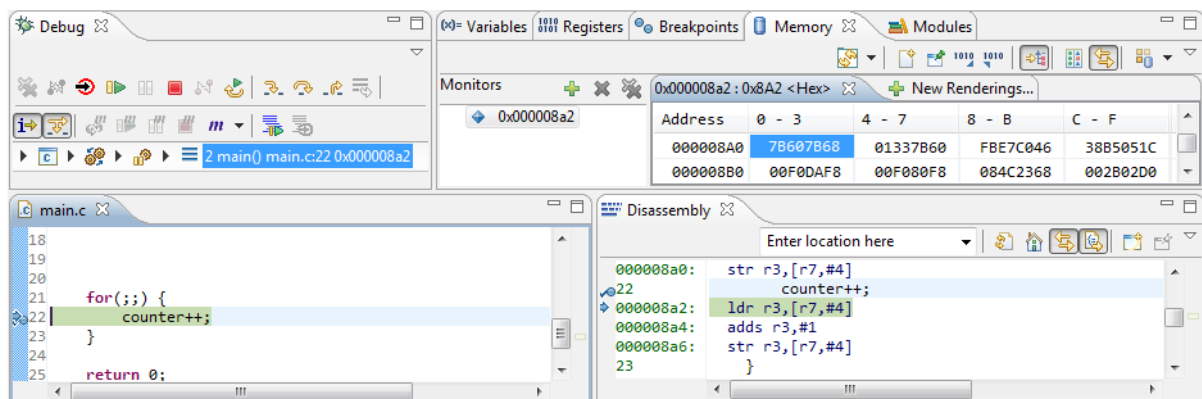


Figura 1: Uma instrução em diferentes níveis de abstração

O IDE CodeWarrior provê mecanismos para visualizar, de forma sincronizada, na perspectiva de depuração a correspondência entre os códigos-fonte (linguagem de alto nível) e códigos em *assembly* compilados (códigos mnemônicos próximos da máquina). Na Figura 1 as instruções correspondentes são destacadas com a cor verde.

3.1 Linguagem de Máquina e de Montagem (*Assembly*)

Vamos fazer o seguinte experimento com o programa-exemplo gerado automaticamente pelo IDE CodeWarrior ao criarmos um novo projeto: setarmos na sua perspectiva de depuração um ponto de parada na linha de instrução “counter++; ” e executarmos passa-a-passo (“*Step Over*”), no modo “*Instruction Stepping Mode*”. Veja na aba “*Disassembly*” que para cada incremento da variável “counter” são executadas 3 instruções de máquina:

```
ldr    r3,[r7,#4]
adds  r3,#1
str   r3,[r7,#4]
```

São chamadas de instruções de máquina, porque cada uma delas corresponde a um código de máquina da arquitetura do processador. Na aba “*Disassembly*” da Figura 1 podemos ver ainda os endereços em que estas instruções são relocadas: 0x000008a2, 0x000008a4 e 0x000008a6. Inserindo o primeiro endereço na aba “*Memory*” temos acesso aos códigos binários armazenados nestes endereços conforme mostra a Figura 1: 0x687B, 0x3301 e 0x607B, respectivamente.

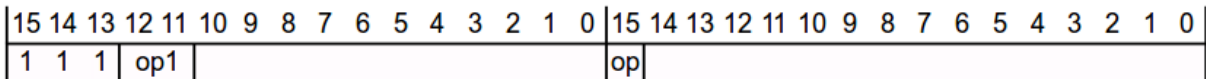
São estes códigos na memória que, de fato, o processador busca, decodifica e executa em cada ciclo de instrução. Se codificarmos o nosso programa em linguagem de alto nível, utilizamos **compiladores** para traduzí-la nestes códigos como vimos no capítulo 2. Quando descrevermos o fluxo de controle do nosso procedimento com uso de mnemônicos, ou *assembly*, usamos **montadores** ou *assemblers* para convertê-los em códigos de máquina. Note que, diferentemente da compilação, a montagem é uma tradução direta de um formato simbólico para um formato binário. Estes dois formatos são, na verdade, duas formas distintas de renderizar um mesmo conjunto de dados. Vamos ver nesta seção a correspondência entre estes dois formatos mais detalhadamente.

3.1.1 Códigos de Operação

Embora seja da arquitetura ARM, o nosso processador suporta somente o repertório de instruções *Thumb* de 16 *bits* e um número bem reduzido de instruções de desvio de 32 *bits* de tecnologia *Thumb-2*. Mesmo suportando somente o modo *Thumb*, o nosso processador segue a convenção de chaveamento entre o repertório de instruções ARM e o de *Thumb* através do *bit* 0 dos endereços. Este *bit* não é usado no endereçamento. Quando ele é 1, o modo de instrução é chaveado para *Thumb*; do contrário, para o modo ARM de 32 *bits*. Por este motivo, o PC é sempre inicializado

com um endereço ímpar.

As instruções *Thumb* de 32 bits são reservadas para operações bem específicas, como chaveamento entre os modos ARM e *Thumb* e o controle de transferência de dados dos registradores especiais, como mostra a Figura 2.



For 32-bit Thumb encoding, $op1 \neq 0b00$. If $op1 == 0b00$, a 16-bit instruction is encoded, see *16-bit Thumb instruction encoding* on page A5-84.

Table A5-9 shows the allocation of ARMv6-M Thumb encodings in this space.

Table A5-9 32-bit Thumb encoding

op1	op	Instruction class
x1	x	UNDEFINED
10	1	See <i>Branch and miscellaneous control</i>
10	0	UNDEFINED

Figura 2: Códigos de operação das instruções *Thumb* de 32 bits.

O repertório de instruções *Thumb* de 16 bits inclui os códigos de processamento de dados numéricos inteiros, os códigos de desvio, os códigos de acesso aos registradores de estado APSR (*Application Program Status Register*), os códigos de acesso à “memória” endereçável através do espaço de endereços de 32 bits, os códigos de acesso múltiplo à “memória”, e os códigos de geração forçada de exceções. Figura 3 apresenta um sumário dos códigos de operação do repertório *Thumb* de 16 bits. No capítulo A.6 em [1] são detalhados a sintaxe de cada instrução *Thumb* e o respectivo código binário. Uma referência rápida do conjunto completo de instruções é encontrada em [3].

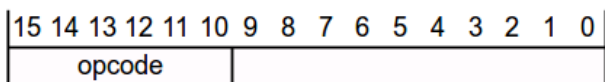


Table A5-1 shows the allocation of 16-bit instruction encodings.

Table A5-1 16-bit Thumb instruction encoding

opcode	Instruction or instruction class
00xxxx	<i>Shift (immediate), add, subtract, move, and compare</i> on page A5-85
010000	<i>Data processing</i> on page A5-86
010001	<i>Special data instructions and branch and exchange</i> on page A5-87
01001x	Load from Literal Pool, see <i>LDR (literal)</i> on page A6-141
0101xx	<i>Load/store single data item</i> on page A5-88
011xxx	
100xxx	
10100x	Generate PC-relative address, see <i>ADR</i> on page A6-115
10101x	Generate SP-relative address, see <i>ADD (SP plus immediate)</i> on page A6-111
1011xx	<i>Miscellaneous 16-bit instructions</i> on page A5-89
11000x	Store multiple registers, see <i>STM, STMIA, STMEA</i> on page A6-175
11001x	Load multiple registers, see <i>LDM, LDMIA, LDMFD</i> on page A6-137
1101xx	<i>Conditional branch, and Supervisor Call</i> on page A5-90
11100x	Unconditional Branch, see <i>B</i> on page A6-119

Figura 3: Códigos de operação do repertório Thumb (Fonte: A5.2 em [1]).

3.1.2 Formato de Instruções

As instruções *Thumb* são constituídas de dois campos: o campo de código de operação e o campo de argumentos. A quantidade de argumentos no segundo campo varia entre 0 a 3 operandos. Por exemplo,

<i>Código de Operação</i>	<i>Argumentos</i>
adds	r3,#1

Opcionalmente, podemos adicionar ainda dois outros campos: o campo de rótulo e o campo de comentários. Todos os rótulos devem ser seguidos de “:”, enquanto o(s) caractere(s) que deve(m) preceder são dependentes do processador. Usualmente é “;”. Porém, para arquitetura i386 e x86_64, ele é “#” e para ARM é “@”. No ambiente IDE CodeWarrior, podemos ainda utilizar a sintaxe de comentários de C /* */. Por exemplo,

Rótulo	Código de Operação	Operandos	Comentários
INC:	adds	r3,#1	@ R3 := [R3] + 1
INC:	adds	r3,#1	/* R3 := [R3] + 1 */

Vale frisar que cada linha do arquivo de um código de montagem só pode conter até um conjunto destes campos, que não necessariamente precisam estar preenchidos.

3.1.3 Modos de Endereçamento

Quando se trata de instruções que manipulam os dados, somente o modo de endereçamento por registrador e o modo de endereçamento imediato são suportados. Observe na coluna “Assembler” da Figura 4 que os argumentos das instruções são registradores (Rd, Rm) ou um valor numérico <imm> precedido por #. Com exceção de algumas instruções que operam sobre o ponteiro da pilha (*stack pointer*), SP, somente os 8 registradores de trabalho R0-R7 podem ser utilizados nas instruções *Thumb*. Os símbolos R e # que aparecem numa instrução em linguagem de montagem (*assembly*) dizem para o montador que tanto o número do registrador quanto o valor <imm> devem ser codificados no próprio código de operação.

Operation		§	Assembler	Updates	Action	Notes
Move	Immediate		MOVS Rd, #<imm>	N Z	Rd := imm	imm range 0-255.
	Lo to Lo		MOVS Rd, Rm	N Z	Rd := Rm	Synonym of LSLS Rd, Rm, #0
	Hi to Lo, Lo to Hi, Hi to Hi		MOV Rd, Rm		Rd := Rm	Not Lo to Lo.
	Any to Any	6	MOV Rd, Rm		Rd := Rm	Any register to any register.
Add	Immediate 3		ADDS Rd, Rn, #<imm>	N Z C V	Rd := Rn + imm	imm range 0-7.
	All registers Lo		ADDS Rd, Rn, Rm	N Z C V	Rd := Rn + Rm	
	Hi to Lo, Lo to Hi, Hi to Hi		ADD Rd, Rd, Rm		Rd := Rd + Rm	Not Lo to Lo.
	Any to Any	T2	ADD Rd, Rd, Rm		Rd := Rd + Rm	Any register to any register.
	Immediate 8		ADDS Rd, Rd, #<imm>	N Z C V	Rd := Rd + imm	imm range 0-255.
	With carry		ADCS Rd, Rd, Rm	N Z C V	Rd := Rd + Rm + C-bit	
	Value to SP		ADD SP, SP, #<imm>		SP := SP + imm	imm range 0-508 (word-aligned).
	Form address from SP		ADD Rd, SP, #<imm>		Rd := SP + imm	imm range 0-1020 (word-aligned).
Form address from PC		ADR Rd, <label>		Rd := label	label range PC to PC+1020 (word-aligned).	
Subtract	Lo and Lo		SUBS Rd, Rn, Rm	N Z C V	Rd := Rn - Rm	
	Immediate 3		SUBS Rd, Rn, #<imm>	N Z C V	Rd := Rn - imm	imm range 0-7.
	Immediate 8		SUBS Rd, Rd, #<imm>	N Z C V	Rd := Rd - imm	imm range 0-255.
	With carry		SBCS Rd, Rd, Rm	N Z C V	Rd := Rd - Rm - NOT C-bit	
	Value from SP		SUB SP, SP, #<imm>		SP := SP - imm	imm range 0-508 (word-aligned).
	Negate		RSBS Rd, Rn, #0	N Z C V	Rd := -Rn	Synonym: NEG Rd, Rn
Multiply	Multiply		MULS Rd, Rd, Rm	N Z * *	Rd := Rm * Rd	* C and V flags unpredictable in §4T, unchanged in §5T and above
Compare	Negative		CMP Rn, Rm	N Z C V	update CPSR flags on Rn - Rm	Can be Lo to Lo, Lo to Hi, Hi to Lo, or Hi to Hi.
	Immediate		CMN Rn, Rm	N Z C V	update CPSR flags on Rn + Rm	
	Immediate		CMP Rn, #<imm>	N Z C V	update CPSR flags on Rn - imm	imm range 0-255.
Logical	AND		ANDS Rd, Rd, Rm	N Z	Rd := Rd AND Rm	
	Exclusive OR		EORS Rd, Rd, Rm	N Z	Rd := Rd EOR Rm	
	OR		ORRS Rd, Rd, Rm	N Z	Rd := Rd OR Rm	
	Bit clear		BICS Rd, Rd, Rm	N Z	Rd := Rd AND NOT Rm	
	Move NOT		MVNS Rd, Rd, Rm	N Z	Rd := NOT Rm	
	Test bits		TST Rn, Rm	N Z	update CPSR flags on Rn AND Rm	
Shift/rotate	Logical shift left		LSLS Rd, Rm, #<shift>	N Z C*	Rd := Rm << shift	Allowed shifts 0-31. * C flag unaffected if shift is 0.
			LSLS Rd, Rd, Rs	N Z C*	Rd := Rd << Rs[7:0]	* C flag unaffected if Rs[7:0] is 0.
	Logical shift right		LSRS Rd, Rm, #<shift>	N Z C	Rd := Rm >> shift	Allowed shifts 1-32.
			LSRS Rd, Rd, Rs	N Z C*	Rd := Rd >> Rs[7:0]	* C flag unaffected if Rs[7:0] is 0.
	Arithmetic shift right		ASRS Rd, Rm, #<shift>	N Z C	Rd := Rm ASR shift	Allowed shifts 1-32.
			ASRS Rd, Rd, Rs	N Z C*	Rd := Rd ASR Rs[7:0]	* C flag unaffected if Rs[7:0] is 0.
	Rotate right		RORS Rd, Rd, Rs	N Z C*	Rd := Rd ROR Rs[7:0]	* C flag unaffected if Rs[7:0] is 0.

Figura 4: Instruções de processamento de dados (Fonte: [3]).

Por exemplo, o código de máquina do código de operação ADDS é o apresentado na Figura 5. O montador traduz a instrução “adds r3,#1” para o código binário 0b001 10 011 00000001, um vez que o valor “1” é codificado no campo <imm8> (*bits* 7-0) e o número do registrador r3 é codificado no campo <Rdn> (*bits* 10-8) do código. Em hexadecimal, o código de máquina da instrução é

0x3301 como vimos na aba “Memory”.

Encoding T2 All versions of the Thumb instruction set.

ADDS <Rdn>, #<imm8>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	Rdn			imm8							

d = UInt(Rdn); n = UInt(Rdn); setflags = !InITBlock(); imm32 = ZeroExtend(imm8, 32);

Figura 5: Código de máquina da instrução ADDS (Fonte: Seção A 6.7.2 em [1]).

Caso seja necessário processar os dados que não estejam carregados nos registradores, deve-se usar instruções de acesso à “memória” para fazer transferência de dados entre a “memória” e os registradores, como mostra a sequência de 3 códigos de operação apresentada na introdução da Seção 3.1: ldr (carrega no registrador), adds (soma) e str (armazena na memória). Para acessos à “memória”, temos mais opções em modos de endereçamento como mostra a coluna “Assembler” da tabela da Figura 6. São suportados o modo de deslocamento imediato e via registrador a partir do endereço no registrador-base Rn (Seção A4.6.2 em [1]).

Operation		§	Assembler	Action	Notes
Load	with immediate offset, word		LDR Rd, [Rn, #<imm>]	Rd = [Rn + imm]	imm range 0-124, multiple of 4.
	halfword		LDRH Rd, [Rn, #<imm>]	Rd = ZeroExtend([Rn + imm][15:0])	Clears bits 31:16, imm range 0-62, even.
	byte		LDRB Rd, [Rn, #<imm>]	Rd = ZeroExtend([Rn + imm][7:0])	Clears bits 31:8, imm range 0-31.
	with register offset, word		LDR Rd, [Rn, Rm]	Rd = [Rn + Rm]	
	halfword		LDRH Rd, [Rn, Rm]	Rd = ZeroExtend([Rn + Rm][15:0])	Clears bits 31:16
	signed halfword		LDRSH Rd, [Rn, Rm]	Rd = SignExtend([Rn + Rm][15:0])	Sets bits 31:16 to bit 15
	byte		LDRB Rd, [Rn, Rm]	Rd = ZeroExtend([Rn + Rm][7:0])	Clears bits 31:8
	signed byte		LDRSB Rd, [Rn, Rm]	Rd = SignExtend([Rn + Rm][7:0])	Sets bits 31:8 to bit 7
	PC-relative		LDR Rd, <label>	Rd = [label]	label range PC to PC+1020 (word-aligned).
	SP-relative		LDR Rd, [SP, #<imm>]	Rd = [SP + imm]	imm range 0-1020, multiple of 4.
Multiple, not including base			LDM Rn!, <loreglist>	Loads list of registers (not including Rn)	Always updates base register, Increment After.
	Multiple, including base		LDM Rn, <loreglist>	Loads list of registers (including Rn)	Never updates base register, Increment After.
Store	with immediate offset, word		STR Rd, [Rn, #<imm>]	[Rn + imm] := Rd	imm range 0-124, multiple of 4.
	halfword		STRH Rd, [Rn, #<imm>]	[Rn + imm][15:0] := Rd[15:0]	Ignores Rd[31:16], imm range 0-62, even.
	byte		STRB Rd, [Rn, #<imm>]	[Rn + imm][7:0] := Rd[7:0]	Ignores Rd[31:8], imm range 0-31.
	with register offset, word		STR Rd, [Rn, Rm]	[Rn + Rm] := Rd	
	halfword		STRH Rd, [Rn, Rm]	[Rn + Rm][15:0] := Rd[15:0]	Ignores Rd[31:16]
	byte		STRB Rd, [Rn, Rm]	[Rn + Rm][7:0] := Rd[7:0]	Ignores Rd[31:8]
	SP-relative, word		STR Rd, [SP, #<imm>]	[SP + imm] := Rd	imm range 0-1020, multiple of 4.
Multiple		STM Rn!, <loreglist>	Stores list of registers	Always updates base register, Increment After.	
Push	Push		PUSH <loreglist>	Push registers onto full descending stack	
	Push with link		PUSH <loreglist>+LR	Push LR and registers onto full descending stack	
Pop	Pop		POP <loreglist>	Pop registers from full descending stack	
	Pop and return	4T	POP <loreglist>+PC	Pop registers, branch to address loaded to PC	
	Pop and return with exchange	5T	POP <loreglist>+PC	Pop, branch, and change to ARM state if address[0] = 0	

Figura 6: Instruções de transferência de dados entre registradores e memória (Fonte: [3]).

Por exemplo, o modo de endereçamento da instrução “ldr r3,[r7,#4]” é o modo de deslocamento imediato (ou modo de endereçamento indireto), pois o endereço acessado é a soma do conteúdo do registrador-base R7 e o valor 4. E o conteúdo deste endereço é carregado no registrador R3. O montador entende que é o conteúdo por causa dos colchetes []. O código de máquina da instrução LDR é mostrado na Figura 7. Vamos ver como o montador traduz a instrução. O valor 4 é colocado no campo <imm5> (bits 10-6), o número do registrador-destino R3 é armazenado no campo <Rt> (bits 2-0) e o do registrador-base no campo <Rn> (bits 5-3). Portanto, o código de máquina correspondente à instrução é 0b011 0 1 00100 111 011. Em hexadecimal, é 0x687B como vimos na

aba “*Memory*”. O argumento [Rn,Rm] nas instruções da Figura 6 representa o modo de deslocamento via registrador. O valor a ser armazenado no registrador destino <Rt> é o conteúdo do endereço definido pela soma do conteúdo dos registradores Rn e Rm.

Vale ressaltar aqui que a versão LDR que usa o contador de programa, PC, como registrador-base é muito utilizada pelo compilador do nosso IDE CodeWarrior. A maioria dos dados residentes na memória tem os seus endereços especificados como um valor deslocado em relação ao PC nas instruções de montagem após a compilação. Conforme mostra a Seção A 6.7.26 em [1], este valor de deslocamento é codificado na própria instrução, em 5 bits ou em 8 bits. Porém, a faixa de valores de deslocamento (*offset*) é 7 e 10 bits, respectivamente. Pois, o processador assume que os endereços sejam sempre múltiplos de 4 e automaticamente complementa o valor binário codificado na instrução com dois dígitos binários menos significativos “00”. Figura 7 ilustra a instrução LDR tendo PC como registrador-base. O valor de deslocamento codificado no campo “imm5” deve ser complementado com “00” antes de somá-lo com o endereço no PC.

Encoding T1 All versions of the Thumb instruction set.

LDR <Rt>, [<Rn>{, #<imm5>}]

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	imm5					Rn		Rt			

```
t = UInt(Rt); n = UInt(Rn); imm32 = ZeroExtend(imm5:'00', 32);
index = TRUE; add = TRUE; wback = FALSE;
```

Figura 7: Código de máquina da instrução LDR (Fonte: Seção A 6.7.26 em [1]).

Quando se deseja transferir um bloco de dados da pilha entre um conjunto de registradores, pode-se utilizar códigos de operação **pop** e **push**. O conjunto de registradores de interesse, separados pela vírgula, deve ser fechado entre as chaves. Só um lembrete: como o endereço do topo da pilha está armazenado no registrador SP (*stack pointer*), são duas instruções de transferência especiais que tem como registrador-base o registrador SP.

Há uma instrução em *assembly* que não faz nada absolutamente e força a execução de um ciclo de instrução que corresponde a um ciclo de relógio. Ela é a instrução **nop**. Essa instrução pode ser encontrada em várias situações em que se precisa “sincronizar” o tempo de execução de um bloco de códigos com outros blocos ou com um intervalo de tempo pré-estabelecido. No entanto, de acordo com a Seção A6.7.47 de [1], que os efeitos da instrução *thumb* NOP são imprevisíveis; portanto, deve-se evitar o seu uso no controle de atrasos.

Encoding T1

ARMv6-M, ARMv7-M

NOP

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0

// No additional decoding required

Figura 8: Código de máquina da instrução NOP (Fonte: Seção A 6.7.47 em [1]).

3.1.4 Desvios

O repertório de instruções *Thumb* inclui as instruções de desvio, conforme mostra a Figura 9.

If-Then	If-Then	T2	IT(pattern) {cond}	Makes up to four following instructions conditional, according to pattern. pattern is a string of up to three letters. Each letter can be T (Then) or E (Else).	The first instruction after IT has condition cond. The following instructions have condition cond if the corresponding letter is T, or the inverse of cond if the corresponding letter is E. See Table Condition Field .
Branch	Conditional branch	T2	B{cond} <label>	If {cond} then PC := label	label must be within -252 to +258 bytes of current instruction. See Table Condition Field .
	Compare, branch if (non) zero	T2	CB{N}Z Rn, <label>	If Rn {== !=} 0 then PC := label	label must be within +4 to +130 bytes of current instruction.
	Unconditional branch		B <label>	PC := label	label must be within ±2KB of current instruction.
	Long branch with link		BL <label>	LR := address of next instruction, PC := label	This is a 32-bit instruction. label must be within ±4MB of current instruction (T2: ±16MB).
	Branch and exchange		BX Rm	PC := Rm AND 0xFFFFFFF	Change to ARM state if Rm[0] = 0.
	Branch with link and exchange	5T	BLX <label>	LR := address of next instruction, PC := label Change to ARM	This is a 32-bit instruction. label must be within ±4MB of current instruction (T2: ±16MB).
	Branch with link and exchange	5T	BLX Rm	LR := address of next instruction. PC := Rm AND 0xFFFFFFF	Change to ARM state if Rm[0] = 0

Figura 9: Instruções de desvio (Fonte: [3]).

Observe que algumas instruções correspondem aos desvios condicionados aos valores dos *bits* de condição *N*(egative), *Z*(ero), *C*(arry) e (*o*)*V*(erflow) do registrador de estado APSR. Estes *bits* são, usualmente, atualizados conforme o resultado de uma instrução que manipula os dados. No entanto, na arquitetura ARM algumas instruções não modificam tais *bits* automaticamente, como ADD (Seção A6.7.3 em [1]) e MOV (Seção A6.7.40 em [1]). Para cada uma destas instruções existe uma versão correspondente que atualiza os *bits* de condição. Ela é diferenciada pelo sufixo S, como ADDS e MOVS. Os mnemônicos utilizados para os desvios condicionados mais utilizados são EQ (igual), NE (diferente), CS (*C*arry em 1), CC (*C*arry em 0), MI (negativo), PL (>=), VS (*O*verflow em 1), VC (*O*verflow em 0), GE (>=), LT (<), GT (>), LE (<=) precedidos de B.

3.1.5 Ciclos de Instrução

A arquitetura ARM foi concebida de forma que a maioria das instruções de processamento dos dados leva apenas um ciclo de relógio para ser executada como mostra a Figura 10. Ou seja, o ciclo de execução dessas instruções corresponde a um ciclo de relógio. O processador Cortex-M0+, integrado no nosso microcontrolador, suporta as instruções de multiplicação em 32-*bits*. Elas

apresentam tempos de execução diferenciados como consta na Figura 10. Informações adicionais podem ser consultadas na Tabela 3.1 em [5].

Table 3-1 Cortex-M0+ instruction summary

Operation	Description	Assembler	Cycles
Move	8-bit immediate	MOVS Rd, #<imm>	1
	Lo to Lo	MOVS Rd, Rm	1
	Any to Any	MOV Rd, Rm	1
	Any to PC	MOV PC, Rm	2
Add	3-bit immediate	ADDS Rd, Rn, #<imm>	1
	All registers Lo	ADDS Rd, Rn, Rm	1
	Any to Any	ADD Rd, Rd, Rm	1
	Any to PC	ADD PC, PC, Rm	2
	8-bit immediate	ADDS Rd, Rd, #<imm>	1
	With carry	ADCS Rd, Rd, Rm	1
	Immediate to SP	ADD SP, SP, #<imm>	1
	Form address from SP	ADD Rd, SP, #<imm>	1
	Form address from PC	ADR Rd, <label>	1
	Subtract	Negate	RSBS Rd, Rn, #0
Multiply	Multiply	MULS Rd, Rm, Rd	1 or 32 ^a
Compare	Compare	CMP Rn, Rm	1
	Negative	CMN Rn, Rm	1
	Immediate	CMP Rn, #<imm>	1

Rotate	Rotate right by register	RORS Rd, Rd, Rs	1
Load	Word, immediate offset	LDR Rd, [Rn, #<imm>]	2 or 1 ^b
	Halfword, immediate offset	LDRH Rd, [Rn, #<imm>]	2 or 1 ^b
	Byte, immediate offset	LDRB Rd, [Rn, #<imm>]	2 or 1 ^b
	Word, register offset	LDR Rd, [Rn, Rm]	2 or 1 ^b
	Halfword, register offset	LDRH Rd, [Rn, Rm]	2 or 1 ^b
	Signed halfword, register offset	LDRSH Rd, [Rn, Rm]	2 or 1 ^b
	Byte, register offset	LDRB Rd, [Rn, Rm]	2 or 1 ^b
Load	Signed byte, register offset	LDRSB Rd, [Rn, Rm]	2 or 1 ^b
	PC-relative	LDR Rd, <label>	2 or 1 ^b
	SP-relative	LDR Rd, [SP, #<imm>]	2 or 1 ^b
	Multiple, excluding base	LDM Rn!, {<loreglist>}	1+N ^c
	Multiple, including base	LDM Rn, {<loreglist>}	1+N ^c

Store	Word, immediate offset	STR Rd, [Rn, #<imm>]	2 or 1 ^b
	Halfword, immediate offset	STRH Rd, [Rn, #<imm>]	2 or 1 ^b
	Byte, immediate offset	STRB Rd, [Rn, #<imm>]	2 or 1 ^b
	Word, register offset	STR Rd, [Rn, Rm]	2 or 1 ^b
	Halfword, register offset	STRH Rd, [Rn, Rm]	2 or 1 ^b
	Byte, register offset	STRB Rd, [Rn, Rm]	2 or 1 ^b
	SP-relative	STR Rd, [SP, #<imm>]	2 or 1 ^b
	Multiple	STM Rn!, {<loreglist>}	1+N ^c
Push	Push	PUSH {<loreglist>}	1+N ^c
	Push with link register	PUSH {<loreglist>, LR}	1+N ^d
Pop	Pop	POP {<loreglist>}	1+N ^c
	Pop and return	POP {<loreglist>, PC}	3+N ^d
Branch	Conditional	B<cc> <label>	1 or 2 ^e
	Unconditional	B <label>	2

Figura 10: Tempos de execução de instruções de processamento de dados (Fonte: Seção 3.3 em [5]).

Diferentemente dos seus irmãos da família Cortex-M, o processador Cortex-M0+ é uma arquitetura com apenas 2 estágios de *pipeline*, conforme ilustra a Figura 11. O primeiro estágio, *FETCH*, compreende a busca da instrução (acesso à memória) e a lógica de pré-codificação da instrução, e o segundo estágio *EXECUTE* inclui o restante do circuito de decodificação da instrução e o circuito de execução da instrução propriamente dita. Com isso, o número mínimo de ciclos de relógio requeridos pelas instruções de transferência de dados entre registradores e memória, LDR e STR, passou de 3 (*FETCH*, *DECODE* e *EXECUTE*) para 2 ciclos (*FETCH* e *EXECUTE*) como se pode constatar na Figura 10. Com isso, pode-se economizar os registradores de um estágio, e portanto reduzir ainda mais o consumo de energia, e reduzir o tempo ocioso na execução de instruções de desvio de dois estágios *DECODE* e *EXECUTE* (2 ciclos de relógio) para um único estágio *EXECUTE* (1 ciclo de relógio).

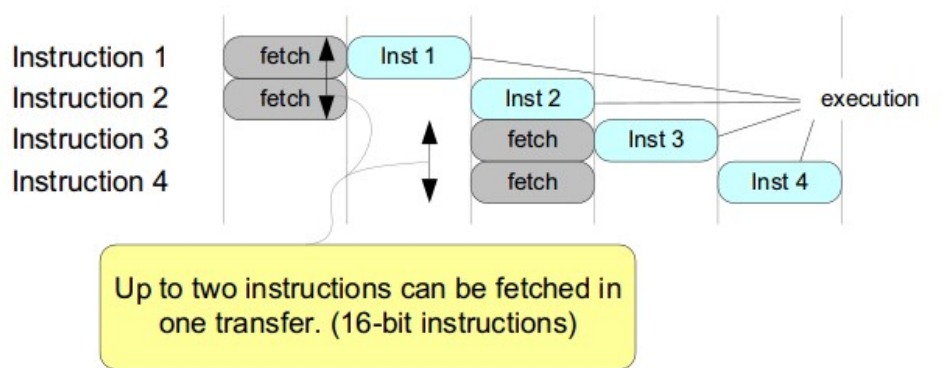


Figura 11: Pipeline de 2 estágios (Fonte: [11]).

Tendo uma ideia da quantidade de ciclos de relógio que uma instrução precisa para ser executada, os códigos de montagem nos permitem estimar com acurácia o tempo de execução de um programa. Seção 3.3 em [5] sintetiza os ciclos de instrução em termos de ciclos de relógio de todas as instruções do processador Cortex-M0+.

Para determinarmos, em unidade de tempo, um ciclo de instrução precisamos saber o período de um ciclo de relógio. Este período vai depender do sinal de relógio configurado para o processador. Conforme a Seção 4.1.3 em [10], o modo de relógio padrão configurado no nosso *kit* FRDM-KL25Z é o modo MCGFLLCLK. A frequência de operação é 20.97MHz, mais precisamente, 20.971520MHz. Isso corresponde a um período de em torno 0.048 microsegundos. Ou seja, o ciclo de instrução de uma operação que manipula dados leva menos de um décimo de um microsegundo para ser executada!

Por exemplo, o seguinte trecho de código

```

mov r2,#100
Loop: sub r2, r2, #1
      bne Loop

```

gasta 3 ciclos de relógio para executar as duas instruções, sub (1 ciclo) e bne (2 ciclos), no laço Loop enquanto o conteúdo do registrador não zere. Temos então $3 \cdot 100$ ciclos de relógio mais um ciclo correspondente à instrução mov. Ao todo, o processador gastará em torno de 301 ciclos de relógio que corresponde a $\sim 0.048\mu\text{s} \cdot 301 = 14.45\mu\text{s}$.

Vale, no entanto, chamar atenção aqui que, em decorrência das otimizações e da estrutura de *instruction pipelining* do processador ARM como mostra a Figura 11, a simples soma dos tempos dos ciclos de instrução de uma sequência de operações que compõem um programa é sempre uma estimativa conservadora do tempo de execução da sequência.

3.2 Diretivas

As diretivas são comandos especiais que mostram ao montador onde se carrega o bloco de códigos binários, como os rótulos são associados aos endereços, quais espaços de memória devem ser reservados para dados, etc. Elas não são traduzidas em códigos binários. Uma referência rápida das diretivas para o montador GNU é [6].

Dentre as diretivas da arquitetura ARM, destacamos algumas mais utilizadas [4]:

- .section:** define uma seção especificada, como `.text`, `.rodata` (*read-only data*) e `.data`. Algumas seções podem ser definidas pelas diretivas próprias
 - .text:** define uma seção de códigos (instruções)
 - .data:** define uma seção de dados armazenados na memória RAM
 - .bss:** define uma seção de dados inicializados com zero
- .space:** reserva um bloco de memória de tamanho especificado
- .byte:** especifica um *byte* de memória para o valor especificado
- .2byte/.hword/.short:** aloca 2 *bytes* de memória para o valor especificado
- .4byte/.word:** aloca 4 *bytes* de memória para o valor especificado
- .8byte/.long:** aloca 8 *bytes* de memória para o valor especificado
- .align:** alinha o endereço num valor que seja múltiplo do valor de *bytes* especificado
- .equ:** cria um símbolo para uma constante
- .set:** cria um símbolo para uma variável
- .global:** especifica que o símbolo é visível por outros arquivos
- .rept:** marca o início de um bloco cuja execução deve ser repetida em número de vezes especificado
- .endr:** marca o fim de um bloco de repetição
- .type:** seta o tipo do símbolo. Dois tipos mais comuns: *function* (nome de uma função) e *object* (nome de um dado)
- .macro :** marca o início da definição de uma macro
- .endm:** marca o fim da definição de uma macro
- .func:** marca o início de uma função
- .endfunc:** marca o fim de uma função
- .include:** inclui um arquivo como a diretiva `#include`

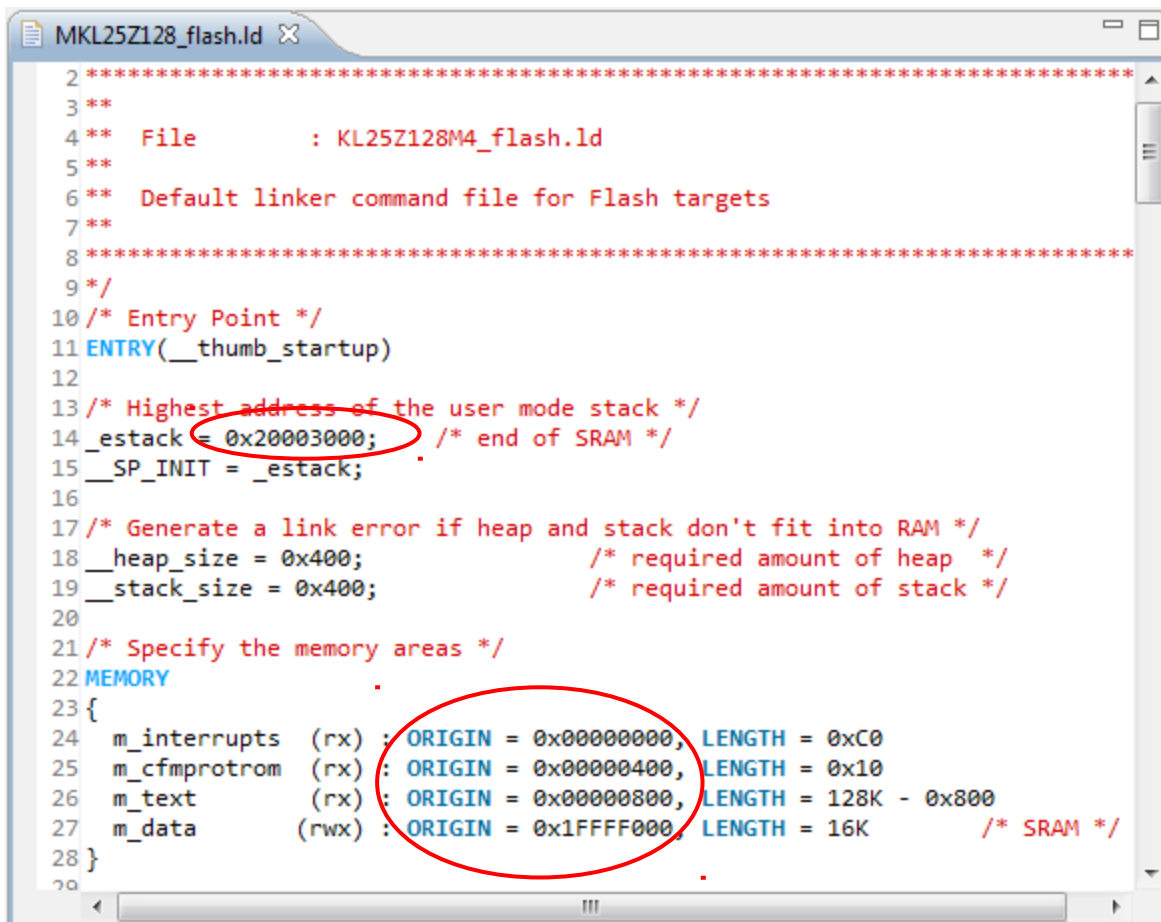
Por exemplo, quando se cria um projeto no ambiente IDE CodeWarrior, configurando *assembly* (ASM) como a linguagem de programação do projeto, é gerado automaticamente um código-fonte `main.s` com as seguintes diretivas

```
.text
.section      .rodata
.align       2
.LC0:
.text
.align       2
.global      main
.type        main function
.align       2
::
```

A primeira linha indica que os códigos de máquina que se seguem devem ser colocados na seção “.text”; na segunda linha, define dentro da seção “.text” uma seção “.rodata”; e a terceira linha especifica que os endereços dos códigos de máquina devem ser alinhados com endereços que sejam múltiplos de 2. O rótulo “.LC0” permite que o endereço do código da máquina correspondente seja referenciado por ele. Neste caso, não há nenhum código de máquina associado ao rótulo. As duas

linhas seguintes orientam o montador a colocar os códigos de máquina na seção “.text” em endereços pares. Em seguida, o símbolo *main* é declarado como global de forma que ele possa ser ligado com o mesmo símbolo nos outros arquivos. Na linha seguinte, a diretiva mostra ao montador que o símbolo *main* é o nome de uma função.

E onde fica localizada a seção de “.text”? Ela é configurada dentro do arquivo MKL25Z128_flash.ld da pasta *Linker_Files*. Observe na Figura 12 que o endereço inicial da seção “.text” é definido como 0x00000800 e o endereço inicial da seção “.data”, 0x1FFFF000. O endereço 0x00000800 pertence a um endereço da memória e o endereço 0x1FFFF000 corresponde a um endereço da memória RAM. É mostrada ainda na Figura 12 a configuração do endereço do topo da pilha (SP). Este endereço é armazenado no endereço 0x00000000 quando se inicializa o microcontrolador (*Reset*).



```
2 *****
3 **
4 ** File      : KL25Z128M4_flash.ld
5 **
6 ** Default linker command file for Flash targets
7 **
8 *****
9 */
10 /* Entry Point */
11 ENTRY(_thumb_startup)
12
13 /* Highest address of the user mode stack */
14 _estack = 0x20003000; /* end of SRAM */
15 _SP_INIT = _estack;
16
17 /* Generate a link error if heap and stack don't fit into RAM */
18 __heap_size = 0x400; /* required amount of heap */
19 __stack_size = 0x400; /* required amount of stack */
20
21 /* Specify the memory areas */
22 MEMORY
23 {
24  m_interrupts (rx) : ORIGIN = 0x00000000, LENGTH = 0xC0
25  m_cfmprom (rx) : ORIGIN = 0x00000400, LENGTH = 0x10
26  m_text (rx) : ORIGIN = 0x00000800, LENGTH = 128K - 0x800
27  m_data (rwx) : ORIGIN = 0x1FFFF000, LENGTH = 16K /* SRAM */
28 }
29
```

Figura 12: Partições da memória

3.3 Chamada de Rotina

Podemos chamar com o código de operação **bl** (Figura 9) uma rotina com o endereço de retorno automaticamente salvo no registrador de *link* (LR, *link register*). Como temos um número bem reduzido de registradores e precisamos retornar ao ponto em que a rotina foi chamada depois do seu

processamento, é imprescindível que se salva o estado do processador antes do processamento da rotina e que se recupera este estado antes de retornar à função que a chamou, ou seja,

1. salve o conteúdo dos registradores, no mínimo o conteúdo do registrador LR na pilha com uso da instrução **push**;
2. processe os dados com uso dos registradores e pilha;
3. recupere o conteúdo dos registradores a partir da pilha com uso da instrução **pop**, carregando o endereço de retorno no PC (contador de programa).

Vale observar que as instruções do repertório *Thumb* só tem 3 bits reservados para especificar os registradores. Como o registrador SP corresponde ao registrador R13, não é possível acessá-lo a não ser por instruções especiais como *ADD (SP plus immediate)*, *ADD (SP plus register)* e *SUB (SP minus immediate)* (Figura 4). Quando é necessário acessar os dados da pilha numa rotina, é comum transferir o conteúdo do SP para um outro registrador de trabalho, por exemplo R7, e usar este novo registrador como registrador-base nos modos de endereçamento de deslocamento.

Por exemplo, a rotina em C

```
void delay (int i)
{
    int j =i;
    while (j) j--;
}
```

pode ser codificada em *assembly* utilizando as seguintes instruções, considerando que o valor do argumento *i* seja passado pelo registrador R0:

delay:

```
push  {r3,r7,lr}
sub   sp,sp,#8
add   r7,sp,#0
str   r0,[r7,#4]
ldr   r3,[r7,#4]
```

iteracao:

```
sub   r3,#1
bne   iteracao
mov   sp,r7
add   sp,sp,#8
pop   {r3,r7,pc}
```

A primeira instrução da rotina é salvar (1) o conteúdo dos registradores R3, R7 que serão utilizados na rotina, e (2) o endereço de retorno no registrador LR. A última instrução da rotina *delay* é recuperar o conteúdo dos registradores e carregar o endereço de retorno no PC. Para preservar o valor no registrador R0, foi utilizado na rotina o registrador R3 para decrementar o valor "i". Só para ilustrar o uso da pilha no armazenamento dos valores das variáveis locais da rotina,

escrevemos ainda o valor do registrador R0 na pilha e lemos da pilha o valor inicial do registrador R3.

3.4 Integração de *Assembly* com C

É consenso que programas em linguagem de alto nível é muito mais inteligível, portátil e de fácil manutenção em relação aos programas em linguagem de montagem. E, particularmente, a linguagem C é uma linguagem de alto nível estruturada com uma tipagem de dados bem flexível, além de suportar operações nativas de manipulação dos *bits* e do uso de memória. Ela é uma linguagem que mantém a filosofia de que o programador tem que saber o que está sendo programado. Porém, como vimos ao longo deste capítulo, ela esconde muitos detalhes críticos de *hardware*, como a pilha, a segmentação do espaço da memória, os sinais de comunicação com os periféricos, o processamento de interrupções. Além disso, programas em C é apenas um meio para chegar aos códigos de máquina. Eles precisam ser compilados, e nenhum compilador superou ainda um exímio programador em *assembly* em termos de tamanho de códigos [7]. Portanto, a prática comum é combinar num mesmo projeto, que apresenta restrições críticas em recursos, as vantagens dos códigos em C e o desempenho dos códigos em *assembly*.

Através da palavra-chave **asm** podemos misturar as instruções em *assembly* com os códigos em C. **Asm** consegue orientar o compilador a incluir as instruções em *assembly* no ponto do programa em C em que elas são definidas (*inline functions*) e o montador é automaticamente chamado para montar os códigos (binários) de máquina. O formato básico do uso de **asm** é [8]:

```
asm [volatile] ( AssemblerTemplate
                : OutputOperands
                [ : InputOperands
                [ : Clobbers ] ])
```

```
asm [volatile] goto ( AssemblerTemplate
                    :
                    : InputOperands
                    : Clobbers
                    : GotoLabels)
```

onde

volatile é um qualificador que orienta o compilador a não otimizar o código da função

goto é um qualificador que orienta o compilador a desviar o fluxo para os símbolos listados na lista GotoLabels.

AssemblerTemplate é o programa constituído de instruções em *assembly*

OutputOperands é uma lista de variáveis em C, separadas pela vírgula, que são processadas pelo *AssemblerTemplate*. A lista pode ser vazia.

InputOperands é uma lista de variáveis em C, separadas pela vírgula, que são lidas pelas instruções do *AssemblerTemplate*. A lista pode ser vazia.

Clobbers é uma lista de registradores ou variáveis, além das especificadas anteriormente, que são

modificadas pelas instruções do *AssemblerTemplate*.

GotoLabels é uma lista de rotinas para as quais as instruções do *AssemblerTemplate* podem chamar/entrar.

Por exemplo, no trecho do código abaixo a variável "counter" é tanto uma variável de entrada quanto a de saída. Portanto, ela consta na lista de de *OutputOperands* (primeira linha) e *InputOperands* (segunda linha). Podemos ter restrições acompanhadas a cada variável. No exemplo abaixo temos duas restrições entre aspas:

= : o conteúdo da variável é sobrescrito, substituído por um novo valor, indicando que se trata de uma variável de saída;

r : qualquer registrador pode ser usado para representar "counter" no bloco de códigos de máquina

```
int main(void)
{
    int counter = 0;

    for (;;) {
        asm (
            "mov r0, %0 \n\t"
            "add r0, r0, #1 \n\t"
            "mov %0, r0 \n\t"
            : "=r" (counter)
            : "r" (counter)
            );
    }
    return 0;
}
```

No entanto, podemos aumentar a inteligibilidade do código, atribuindo a cada variável um nome simbólico. No código abaixo, definimos o nome simbólico [counter] para a variável "counter" dentro do escopo do código de máquina

```
int main(void)
{
    int counter = 0;

    for (;;) {
        asm (
            "mov r0, %[counter] \n\t"
            "add r0, r0, #1 \n\t"
            "mov %[counter], r0 \n\t"
            : [counter] "=r" (counter)
            : "r" (counter)
            );
    }
    return 0;
}
```

Se a função tiver somente variáveis de entrada, a lista de *OutputOperands* ficará vazia como mostra o seguinte trecho de código em que o conteúdo da variável “a” da rotina foo é passada para a função **asm** e não há nenhuma variável de saída. Observe que neste caso, não criamos nenhum nome simbólico para a variável. O seu acesso é pela sua posição na lista de operandos de entrada, %0.

```
void foo (int a)
{
    asm (
        "mov r1, %0 \n\t"
        "add r1, #25 \n\t"
        :
        : "r" (a)
        );
}
```

Vale comentar que, de acordo com a convenção da arquitetura ARM, os registradores R0-R3 são automaticamente utilizados para passar os valores de até 4 argumentos. Quando se trata de um trecho de códigos em *assembly* dentro de uma sub-rotina, podemos assumir que os quatro primeiros argumentos da função têm os seus valores armazenados nos registradores como mostra o seguinte trecho de código em que o valor da variável "a" e o da variável "b" estão nos registradores R0 e R1, respectivamente:

```
int soma (int a, int b)
{
    int k;
    asm (
        "mov r2, r0 \n\t"
        "add r2, r1 \n\t"
        "mov %0, r2 \n\t"
        : "=r" (k)
        );
    return k;
}
```

Vale observar que as instruções representadas como *string* em C, pois está entre duas aspas duplas, tem dois caracteres de controle “\n” (*new line*) e “\t” (*tab*) como terminador. Estes dois caracteres asseguram que os mnemônicos sejam montados no formato requerido por um código em *assembly* conforme mostramos na Seção 3.1.2: cada linha só contém um conjunto de até 4 campos com um mnemônico no máximo. O conjunto “\n\t” pode ser substituído por “;”.

3.5 Desenvolvimento de Software em Sistemas Embarcados

É importante ressaltar a diferença entre programar um computador e programar um microcontrolador em termos de recursos e de aplicações almeçadas. Em termos de recursos, temos

- uma quantidade extremamente limitada de memória para armazenamento de instruções e de

dados; e

- capacidade de processamento limitada.

Isso demanda desenvolvimento de códigos bem compactos com um aproveitamento otimizado da capacidade de processamento disponível. Escolha apropriada de sequências de instruções e de tipos de dados pode ter um impacto direto no tamanho e no desempenho do programa. E esta escolha depende fortemente do domínio da arquitetura do processador/microcontrolador por parte do desenvolvedor. Vale mencionar que na referência [9] há uma série de dicas para otimizar os códigos de montagem no processador ARM Cortex-M0.

E em termos de aplicações, os requisitos são usualmente

- de tempo real, ou seja, com uma restrição temporal crítica; e
- interações diretas com os periféricos físicos, como sensores e atuadores, de forma assíncrona.

Para coordenar o controle destes periféricos de forma eficiente, é imprescindível elaborar uma estratégia de atendimento a eles. Embora existam diversos módulos de circuitos dedicados para coletar e processar os dados específicos dos periféricos, é único o processador para sincronizar todos os dados de entrada e de saída. O projeto de uma boa estratégia requer além do domínio de tecnologia muita **criatividade** da parte do projetista.

Referências

- [1] ARM. ARMv6-M Architecture Reference Manual
<ftp://ftp.dca.fee.unicamp.br/pub/docs/ea871/ARM/ARMv6-M.pdf>
- [2] The GNU Assembler
<http://tigris.org/doc/gnuasm.html>
- [3] Thumb 16-bit Instruction Set – Quick Reference Card
ftp://ftp.dca.fee.unicamp.br/pub/docs/ea871/ARM/ARM_QRC0006_UAL16.pdf
- [4] Jensbauer. Useful assembler directives and macros for the GNU assembler.
<https://community.arm.com/docs/DOC-9652>
- [5] Cortex-M0+ Technical Reference Manual (Revision: r0p0)
<ftp://ftp.dca.fee.unicamp.br/pub/docs/ea871/ARM/Cortex-M0+.pdf>
- [6] GNU AS ARM Reference V2
ftp://ftp.dca.fee.unicamp.br/pub/docs/ea871/ARM/ARM_quickreference.pdf
- [7] Derrick Klotz. C for Embedded Systems Programming
http://www.nxp.com/files/training/doc/dwf/AMF_ENT_T0001.pdf
- [8] Extended Asm – Assembler Instructions with C expression Operands
<https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html>
- [9] Jensbauer. ARM Cortex-M0 assembly programming tips and tricks
<https://community.arm.com/docs/DOC-7869>
- [10] Freescale Semiconductor. Kinetis L Peripheral Module Quick Reference.
<ftp://ftp.dca.fee.unicamp.br/pub/docs/ea871/ARM/KLQRUG.pdf>
- [11] ARM Cortex-M Programming Guide to Memory Barrier Instructions (Application Note 321)
ftp://ftp.dca.fee.unicamp.br/pub/docs/ea871/ARM/DAI0321A_programming_guide_memory_barriers_for_m_profile.pdf