Manual do PEPE

1 Introdução 2
2 Ligações do módulo PEPE 2
3 Registos 3
3.1 Registo de Estado (RE) 4
3.2 Registo SP (Stack Pointer) 5
3.3 Registo de Ligação (RL) 5
4 Excepções 6
5 Conjunto de instruções 7
5.1 Instruções de dados 7
5.2 Instruções de controlo de fluxo 9
5.3 Instruções 9
6 Aspectos adicionais do assemblador 17
6.1 Literais 17
6.2 Etiquetas 17
6.3 Pseudo-Instruções 18

1 Introdução

O módulo PEPE implementa um processador RISC de 16 bits. Este manual é apenas um resumo das suas características.

2 Ligações do módulo PEPE

O módulo PEPE tem as seguintes ligações:

Designação	Nº de bits	Descrição
Reset	1	Inicialização (activo a 1)
Int0	1	Interrupção 0 (activa a 1)
Int1	1	Interrupção 1 (activa a 1)
Int2	1	Interrupção 2 (activa a 1)
Int3	1	Interrupção 3 (activa a 1)
IntAck	1	Atendimento da Interrupção *(InterruptAcknowledge*) (activo a 1; só para a interrupção 3
Clock	1	Entrada de relógio
DataHigh	8	D(15..8) - Byte de maior peso do barramento de dados
DataLow	8	D(7..0) - Byte de menor peso do barramento de dados
A1-A15	15	A(15..1) - Barramento de endereços, com excepção de A0
A0	1	A0 - Bit de menor peso do barramento de endereços
ByteAdd	1	(*Byte Addressing*) Endereçamento de byte BA=1 – acessos à memória em byte BA=0 – acessos à memória em palavra
nRead	1	Activo a 0 nos ciclos de leitura à memória
nWrite	1	Activo a 0 nos ciclos de escrita na memória
Wait	1	Wait=1 – prolonga o ciclo de acesso à memória Wait=0 – ciclo de acesso à memória com duração mínima
BusRequest	1	Pedido de DMA, activo a 1
BusGrant	1	Autorização para DMA, activo a 1

Os pinos Int0 a Int3 permitem gerar interrupções externas. Estes pinos são activos no flanco de 0 para 1. Os pinos estão ordenados por prioridade, sendo Int0 o mais prioritário e o Int3 o menos prioritário. É possível inibir todas as interrupções externas e cada interrupção em particular de forma independente das restantes.

A interrupção correspondente ao pino Int3 é vectorizada e exige um controlador de interrupções (PIC – *Programmable Interrupt Controller*) adequado (não descrito neste documento). Quando atende esta interrupção, o processador leva o pino IntAck (*Interrupt Acknowledge*) a 1. Esse pino deve ligar ao PIC, que nessa altura é suposto colocar no byte de menor peso do barramento de dados um número (0 a 255) que a rotina de atendimento desta excepção pode usar para identificar a fonte da interrupção.

Os restantes pinos implementam duas interfaces com funcionalidades específicas:

- Interface de memória:
  - DataHigh, Data Low – Barramento de dados, de 16 bits, separado em duas metades de 8 bits;
  - A1-A15, A0 – Barramento de endereços, de 16 bits, separado num barramento de 15 bits e no A0 (a separação destina-se a facilitar o acesso à memória);
  - ByteAdd – *Byte Addressing*. Destina-se a suportar o endereçamento de byte;
  - nRead –Activado a 0 nos ciclos de leitura à memória;
  - nWrite – Activado a 0 nos ciclos de escrita na memória;
  - Wait – Prolonga o ciclo de leitura ou de escrita (para memórias ou periféricos lentos);
- Interface de DMA:
  - BusRequest – Pedido de autorização para operação de DMA;
  - BusGrant –Concessão de autorização para operação de DMA.

O pino ByteAdd destina-se a suportar o endereçamento de byte, juntamente com o bit A0 do barramento de endereços. Os bits A15 a A1 definem qual a palavra de memória endereçada, ficando o acesso a cada um dos bytes dessa palavra dependente dos valores de A0 e ByteAdd, de acordo com a seguinte tabela.

Tipo de acesso	ByteAdd	A0	Byte da palavra acedido
Palavra (16 bits)	0	0	Os dois (acesso a toda a palavra)
		1	Ilegal. Acesso de palavra desalinhado. Gera excepção.
Byte (8 bits)	1	0	Acede só ao byte no endereço par
		1	Acede só ao byte no endereço ímpar

O pinos nRead e nWrite são usados no ciclo de acesso à memória para leitura e escrita, respectivamente, e são activos a 0. Num dado ciclo, apenas um deles estará activo. A transferência de dados está completa quando estes pinos transitam de 0 para 1.

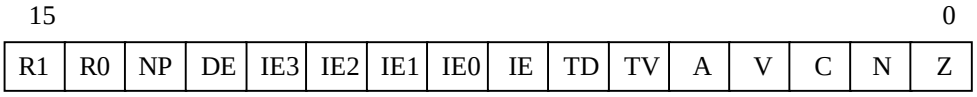
3 Registos

A tabela seguinte indica quais os registos do PEPE.

Número	Sigla	Nome e descrição
---	PC	Contador de Programa (*Program Counter*)
0 a 10	R0 a R10	Registos de uso geral
11	RL ou R11	Registo de Ligação (usado para guardar o PC nas instruções CALLF e RETF, para optimizar as chamadas a rotinas que não chamam outras)
12	SP ou R12	Apontador da Pilha (*Stack Pointer*)
13	RE ou R13	Registo de Estado (*flags*)
14	BTE ou R14	Base da Tabela de Excepções
15	TEMP ou R15	Registo temporário, usado na implementação de algumas instruções (não usar em programação do utilizador)

3.1 Registo de Estado (RE)

O RE (Registo de Estado), contém os bits de estado e de configuração que interessa salvar (na chamada de rotinas e atendimento de excepções) e repôr (no retorno), com a disposição e significado indicados na figura e tabela seguintes. A operação de *reset* do processador coloca todos os bits do Registo de Estado a 0.



Bit	Sigla	Nome e descrição	Tipo
0	Z	Zero. Este bit é colocado a 1 pelas operações da ALU que produzem zero como resultado.	Estado
1	N	Negativo. Este bit é colocado a 1 pelas operações da ALU que produzem um número negativo (bit de maior peso a 1) como resultado.	Estado
2	C	Transporte (*Carry*). Este bit é colocado a 1 pelas operações da ALU que geram transporte.	Estado
3	V	Excesso (*Overflow*). Este bit é colocado a 1 pelas operações da ALU cujo resultado é demasiado grande (em módulo) para ser representado correctamente, seja positivo ou negativo.	Estado
4	A	Bits de estado auxiliar para uso livre pelo utilizador para passar informação entre rotinas, por exemplo. Também pode ser usado na implementação do microcódigo por novas instruções.	Estado
5	TV	Excepção em caso de excesso (*Trap on overflow*). Se este bit estiver a 1, é gerada a excepção EXCESSO na instrução que produzir o excesso. Se estiver a 0, o excesso só actualiza o bit V.	Configuração
6	TD	Excepção em caso de divisão por 0 (*Trap on DIV0*). Se este bit estiver a 1, é gerada a excepção DIV0 numa instrução DIV ou UDIV com quociente 0 (não é gerada a excepção EXCESSO nem o bit V é posto a 1)	Configuração
7	IE	Permissão de Interrupções Externas (*Interrupt Enable*). Só com este bit a 1 as interrupções externas poderão ser atendidas	Configuração
8	IE0	Permissão da Interrupção Externa 0 (*Interrupt Enable*). Só com este bit a 1 os pedidos de interrupção no pino INT0 poderão ser atendidos	Configuração
9	IE1	Idem, para a interrupção INT1	Configuração
10	IE2	Idem, para a interrupção INT2	Configuração
11	IE3	Idem, para a interrupção INT3	Configuração
12	DE	Permissão de accessos directos à memória (*DMA Enable*). Só com este bit a 1 os pedidos de DMA no pino BRQ serão tidos em conta e eventualmente atendidos pelo processador	Configuração
13	NP	Nível de Protecção. 0=Sistema; 1=Utilizador. Define o nível de protecção corrente.	Estado
15, 14	R1, R0	Reservados para utilização futura	A definir

3.2 Registo SP (Stack Pointer)

O registo SP (*Stack Pointer*, ou Apontador da Pilha), contém o índice da última posição ocupada da pilha (topo), que cresce decrementando o SP. As operações de PUSH decrementam o SP de 2 unidades e armazenam um valor na nova posição. As operações de POP fazem a sequência inversa. Por isso, o SP deve ser inicializado com o endereço imediatamente a seguir à zona de memória atribuída à pilha (tem de ser um valor par).

3.3 Registo de Ligação (RL)

O RL (Registo de Ligação) destina-se a guardar o endereço de retorno quando a rotina invocada é terminal, isto é, não invoca outras. No retorno, o PC será actualizado a partir do RL. A vantagem deste esquema é evitar uma operação de escrita em memória, causada pelo guardar do endereço de retorno na pilha. Realmente, muitas rotinas não chamam outras, e uma simples pilha de uma posição (o RL) em registo é muito mais rápida de aceder do que uma pilha verdadeira em memória. As instruções CALL e RET usam a

pilha normalmente. As instruções CALLF e RETF utilizam o RL. Cabe ao compilador (ou ao programador de *assembly*) decidir se usa umas ou outras. Naturalmente, não se pode invocar uma rotina com CALL e retornar com RETF (ou invocar com CALLF e retornar com RET). O RL (ou R11) pode ser usado como um registo de uso geral quando não estiver em uso por um par CALLF-RETF.

4 Excepções

Designam-se por excepções os eventos a que o processador é sensível e que constituem alterações, normalmente pouco frequentes, ao fluxo normal de instruções de um programa.

As excepções podem ter origem externa (correspondentes à activação de pinos externos do processador) ou interna (decorrentes tipicamente de erros na execução das instruções).

Existem alguns pinos do PEPE (INT0 a INT3) que originam excepções explicitamente para interromper o fluxo do programa com o fim de lidar com eventos assíncronos ao programa e associados tipicamente com os periféricos. Essas excepções designam-se por interrupções.

A cada excepção está associada uma rotina de tratamento da excepção (ou rotina de serviço da excepção, ou simplesmente rotina de excepção), cujo endereço consta da Tabela de Excepções, que contém uma palavra (o endereço da rotina de tratamento) para cada uma das excepções suportadas pelo processador.

A Tabela de Excepções começa no endereço indicado pelo registo BTE (Base da Tabela de Excepções), que deverá ser previamente inicializado com um valor adequado.

A tabela seguinte descreve as excepções que o PEPE suporta.

Endereço	Excepção	Causa	Ocorre em	Mascarável	Atendimento	Prioridade
0002H	INT0	O pino INT0 do processador é activado (com IE=1, IE0=1).	Qualquer altura	Sim	Após instrução em que ocorre	2
0004H	INT1	O pino INT1 do processador é activado (com IE=1, IE1=1).	Qualquer altura	Sim	Após instrução em que ocorre	3
0006H	INT2	O pino INT2 do processador é activado (com IE=1, IE2=1).	Qualquer altura	Sim	Após instrução em que ocorre	4
0008H	INT3	O pino INT3 do processador é activado (com IE=1, IE3=1).	Qualquer altura	Sim	Após instrução em que ocorre	5
000AH	EXCESSO	Uma operação aritmética gera excesso (*overflow*) se TV=1 no RE	Execução	Sim	Imediato	1
000CH	DIV0	Uma operação de divisão falha por o quociente ser zero se TD=1 no RE	Execução	Sim	Imediato	1
000EH	SOFTWARE	A instrução SWE (*Software Exception*) é executada. Usada tipicamente como chamada ao sistema operativo	Execução	Não	Incluído na execução da instrução	7
0010H	COD_INV	A Unidade de Controlo encontra uma combinação inválida de *opcode*. Pode ser encarada como uma excepção SWE com opcode próprio e portanto permite extender o conjunto de instruções por meio de software (rotina de excepção que verifica qual o *opcode* que gerou a excepção e invoca uma rotina adequada).	Descodificação	Não	Incluído na descodificação da instrução	7
0012H	D_DESALINHADO	É feito um acesso de 16 bits à memória (dados) especificando um endereço ímpar	Execução	Não	Imediato	1
0014H	I_DESALINHADO	É feita uma busca à memória (*fetch*) tendo o PC um endereço ímpar	Busca	Não	Encadeado	6

5 Conjunto de instruções

5.1 Instruções de dados

A tabela seguinte sumariza os modos de endereçamento, isto é, as formas de obter os operandos.

Modo de endereçamento	Obtenção do operando	Nº de bits na instrução	Exemplos de instruções
Imediato	Constante (dados)	4	ADD R1, 3
		8	MOVL R2, 34H MOVH R2, F3H
Registo	Rs	4	ADD R1, R2
Indirecto	[Rs]	4	MOV R1, [R2]
Baseado	[Rs + constante]	4 + 4	MOV R1, [R2+3]
Indexado	[Rs + Ri]	4 + 4	MOV R1, [R2+R3]
Relativo	Constante (endereços)	8	JZ 100H
		12	CALL 100H
Implícito	[SP]	0	PUSH, POP
	SP, PC	0	RET, CALL

As instruções MOVL e MOVH permitem especificar uma constante de 8 bits para inicializar apenas um dos bytes (o de menor e de maior peso, respectivamente) de um dado registo. Permitem resolver o problema de inicializar os 16 bits de um registo com instruções de apenas 16 bits (usando duas instruções). O assemblador permite especificar apenas uma instrução MOV com uma constante de 16 bits, gerando as instruções necessárias de acordo com o valor da constante

ASSEMBLY	EXEMPLOS	INSTRUÇÕES EQUIVALENTES	RTL	EFEITO

MOV Rd, k	-128 ≤ k ≤ +127
	
	MOV R0,-128
	MOV R3, 0
	MOV R6, +127
		MOVL, Rd, k	Rd(7..0) ← k(7..0)
			Rd(15..8) ← k(7){8 }
				Rd fica com uma cópia do valor da constante (de 8 bits) estendida para 16 bits com o seu bit de sinal, **k**(7). O valor anterior de **Rd** é destruído.

	k ≤ -129 ou k ≥ 128
	
	MOV R7, -32768
	MOV R2, -1000
	MOV R5, +500
	MOV R8, +32767
		MOVL, Rd, k(7..0)
		
		MOVH, Rd, k(15..8)
			Rd(7..0) ← k(7..0)
			Rd(15..8) ← k(7){8 }
			
			Rd(15..8) ← k(15..8)
				Rd fica com uma cópia do valor da constante (de 16 bits). O valor anterior de **Rd** é destruído.

Exemplos:

ASSEMBLY	CONSTANTE (HEXADECIMAL 16 BITS)	INSTRUÇÕES MÁQUINA
MOV R1, 0	00 00H	MOVL, R1, 00H
MOV R1, +1	00 01H	MOVL, R1, 01H
MOV R1, +127	00 7FH	MOVL, R1, 7FH
MOV R1, +32767	7F FFH	MOVL, R1, FFH MOVH, R1, 7FH
MOV R1, -1	FF FFH	MOVL, R1, FFH
MOV R1, -128	FF80H	MOVL, R1, 80H
MOV R1, -32768	80 00H	MOVL, R1, 00H MOVH, R1, 80H

A tabela seguinte descreve as formas de acesso à memória em dados e a sua utilização típica.

Instrução	Modos de endereçamento	Operação com a memória	Utilização típica
MOV Rd, [Rs]	Indirecto	Leitura da memória (16 bits)	Transferência de variáveis (16 bits) entre memória e registos
MOV Rd, [Rs + off]	Baseado		
MOV Rd, [Rs + Ri]	Indexado		
MOV [Rd], Rs	Indirecto	Escrita da memória (16 bits)	
MOV [Rd + off], Rs	Baseado		
MOV [Rd + Ri], Rs	Indexado		
MOVB Rd, [Rs]	Indirecto	Leitura da memória (8 bits)	Processamento de bytes individuais (cadeias de caracteres ASCII, por exemplo)
MOVB [Rd], Rs	Indirecto	Escrita da memória (8 bits)	
MOVP Rd, [Rs]	Indirecto	Leitura da memória (16 bits) sem usar a *cache* nem a memória virtual	Leitura de periféricos
MOVP [Rd], Rs	Indirecto		Escrita de periféricos
SWAP Rd, [Rs] ou [Rs], Rd	Indirecto	Troca atómica de dados (16 bits) entre memória e registo.	Troca de dados, semáforos

		Mesmo com *caches*, o acesso à memória é forçado	
PUSH Rd	Implícito (SP)	Escrita na pilha	Guardar valores para mais tarde recuperar
POP Rd	Implícito (SP)	Leitura da pilha	Recuperar valores guardados na pilha

5.2 Instruções de controlo de fluxo

Os aspectos mais importantes a ter em conta à partida são os seguintes:

- O PEPE suporta endereçamento de byte mas os acessos em busca de instrução têm de ser alinhados, pelo que os endereços têm de ser pares (senão é gerada uma excepção quando o acesso for feito). Para aumentar a gama de endereços que é possível atingir a partir das instruções que aceitam um operando imediato, o valor do operando codificado na instrução é entendido pelo PEPE como designando palavras (instruções) e não bytes, pelo que depois, na implementação das instruções, o PEPE multiplica automaticamente o operando por 2 (seja positivo ou negativo) antes de o utilizar no cálculo do endereço destino do salto;
- Todas as instruções de salto e de chamada de rotinas com operando imediato são relativas, isto é, o operando (em complemento para 2) é multiplicado por 2 e somado ao EIS (Endereço da Instrução Seguinte à instrução de salto). No entanto, para facilitar o utilizador, o assemblador requer não uma constante numérica mas sim um endereço simbólico, ou etiqueta (*label*), e o assemblador faz as contas. O assemblador gera um erro caso a constante (8 ou 12 bits, depende da instrução) não seja suficiente para codificar a diferença entre o valor da etiqueta e EIS. Se for o caso, o utilizador deve usar as instruções JUMP e CALL com endereçamento por registo. Estas últimas já têm endereçamento absoluto, isto é, o valor do registo é o novo endereço da instrução a buscar (e não somado com o anterior). Note-se que

  ```
  L1:   JMP   L1      ; operando imediato ⇒ endereçamento relativo
  ```

  resulta num ciclo infinito e o valor do operando codificado na instrução JMP é –1 (o que corresponde a subtrair -2 a EIS).

5.3 Instruções

As instruções sombreadas são reconhecidas pelo assemblador mas na realidade podem ser sintetizadas com recurso a outras, pelo que não gastam codificações de instruções. São oferecidas apenas como notação alternativa para comodidade do programador de linguagem *assembly* e maior clareza dos programas.

As linhas marcadas com “Livre” correspondem às codificações possíveis e ainda não ocupadas.

Os campos marcados com “XXXX” não são relevantes e podem ter qualquer valor (são ignorados pelo PEPE).

Na coluna “Acções” indica-se o significado de cada instrução numa linguagem de transferência de registos (RTL), cujos aspectos essenciais são indicados pela tabela seguinte.

Se o RE for o destino de uma operação, no RE fica exactamente o resultado dessa operação. Neste caso em particular, os bits de estado não são afectados pelo valor do resultado (Z←1 se o resultado for 0000H, por exemplo) como nas outras operações, mas ficam directamente com os bits correspondentes do resultado.

Simbologia	Significado	Exemplo
Ri	Registo principal *i* (R0 a R15, incluindo RL, SP, RE, BTE e TEMP)	R1
PC	Registo *Program Counter*. Só usado do lado esquerdo da atribuição.	PC ← *expressão*
EIS	Endereço da Instrução Seguinte. Não é um registo, mas apenas uma notação que representa o valor do endereço da instrução seguinte (ou seja, é o endereço da instrução corrente acrescido de 2 unidades).	EIS
RER	Registo do Endereço de Retorno (interno ao processador, não acessível em *assembly*). Contém o endereço de retorno quando se invoca uma rotina ou excepção.	RER
Mw[*end*]	Célula de memória de 16 bits que ocupa os endereços *end* e *end*+1 (*end* tem de ser par, senão gera uma excepção). O PEPE usa o esquema Big-Endian, o que significa que o byte de menor peso de Mw[end] está no endereço *end*+1.	Mw[R1+2] Se R1=1000H, o byte de menor peso está em 1003H e o de maior peso em 1002H
Mb[*end*]	Célula de memória de 8 bits cujo endereço é *end* (que pode ser par ou ímpar)	Mb[R3+R4]
(i)	Bit *i* de um registo ou de uma célula de memória	R2(4) Mw[R1](0)
Ra(i..j)	Bits *i* a *j* (contíguos) do registo Ra (*i* >= *j*)	R2(7..3)
bit{n}	Sequência de *n* bits obtida pela concatenação de n cópias de *bit*, que é uma referência de um bit (pode ser 0, 1 ou Ra(i))	0{4} equivale a 0000 R1(15){2} equivale a R1(15) || R1(15)
dest ← *expr*	Atribuição do valor de uma expressão (*expr*) a uma célula de memória ou registo (*dest*). Um dos operandos da atribuição (expressão ou destino) tem de ser um registo ou um conjunto de bits dentro do processador. O operando da direita é todo calculado primeiro e só depois se destrói o operando da esquerda, colocando lá o resultado de *expr*. *dest* e *expr* têm de ter o mesmo número de bits.	R1 ← M[R2] M[R0] ← R4 + R2 R1(7..0) ← R2(15..8)
Z, N, C, V, IE, IE0 a IE4, DE, NP	Bits de estado no RE – Registo de Estado	V ← 0
Expr : *acção*	Executa a *acção* se *expr* for verdadeira (*expr* tem de ser uma expressão booleana)	((N⊕V)∨Z)=1 : PC ← EIS + 2
∧, ∨, ⊕	E, OU, OU-exclusivo	R1 ← R2 ∧ R3
||	Concatenação de bits (os bits do operando da esquerda ficam à esquerda, ou com maior peso)	R1 ← R2(15..8) || 00H

Classe	Sintaxe em *assembly*		Campos da instrução (16 bits)				Acções	Flags afectadas	Comentários
			1º opcode (4bits)	2º opcode (4bits)	1º operando (4bits)	2º operando (4bits)			
Instruções aritméticas	ADD	Rd, Rs	ARITOP	ADD	Rd	Rs	Rd ← Rd + Rs	Z, N, C, V	
		Rd, k		ADDI	Rd	k	Rd ← Rd + k	Z, N, C, V	k ∈ [-8 .. +7]
	ADDC	Rd, Rs		ADDC	Rd	Rs	Rd ← Rd + Rs + C	Z, N, C, V	
	SUB	Rd, Rs		SUB	Rd	Rs	Rd ← Rd – Rs	Z, N, C, V	
		Rd, k		SUBI	Rd	k	Rd ← Rd - k	Z, N, C, V	k ∈ [-8 .. +7]
	SUBB	Rd, Rs		SUBB	Rd	Rs	Rd ← Rd – Rs - C	Z, N, C, V	
	CMP	Rd, Rs		CMP	Rd	Rs	(Rd – Rs)	Z, N, C, V	Rd não é alterado
		Rd, k		CMPI	Rd	k	(Rd – k)	Z, N, C, V	k ∈ [-8 .. +7] Rd não é alterado
	MUL	Rd, Rs		MUL	Rd	Rs	Rd ← Rd * Rs	Z, N, C, V	O registo Rs é alterado
	DIV	Rd, Rs		DIV	Rd	Rs	Rd ← quociente(Rd / Rs)	Z, N, C, V←0	Divisão inteira
	MOD	Rd, Rs		MOD	Rd	Rs	Rd ← resto(Rd / Rs)	Z, N, C, V←0	Resto da divisão inteira
	NEG	Rd		NEG	Rd	xxxx	Rd ← –Rd	Z, N, C, V	Complemento para 2 V←1 se Rd for 8000H
	Livre								
	Livre								

Classe	Sintaxe em *assembly*		Campos da instrução (16 bits)				Acções	Flags afectadas	Comentários
			1º opcode (4bits)	2º opcode (4bits)	1º operando (4bits)	2º operando (4bits)			
Instruções de bit	AND	Rd, Rs	BITOP	AND	Rd	Rs	Rd ← Rd ∧ Rs	Z, N	
	OR	Rd, Rs		OR	Rd	Rs	Rd ← Rd ∨ Rs	Z, N	
	NOT	Rd		NOT	Rd	xxxx	Rd ← Rd ⊕ FFFFH	Z, N	Complemento para 1
	XOR	Rd, Rs		XOR	Rd	Rs	Rd ← Rd ⊕ Rs	Z, N	
	TEST	Rd, Rs		TEST	Rd	Rs	Rd ∧ Rs	Z, N	Rd não é alterado
	BIT	Rd, n		BIT	Rd	n	Z ← Rd(k) ⊕ 1	Z	Rd não é alterado
	SET	Rd, n		SETBIT	Rd	n	Rd(n) ← 1	Z, N ou outra (se Rd for RE)	n ∈ [0 .. 15] Se Rd=RE, afecta apenas RE(n)
	EI			SETBIT	RE	IE_index	RE(IE_index) ← 1	EI	Enable interrupts
	EI0			SETBIT	RE	IE0_index	RE (IE0_index) ← 1	EI0	Enable interrupt 0
	EI1			SETBIT	RE	IE1_index	RE (IE1_index) ← 1	EI1	Enable interrupt 1
	EI2			SETBIT	RE	IE2_index	RE (IE2_index) ← 1	EI2	Enable interrupt 2
	EI3			SETBIT	RE	IE3_index	RE (IE3_index) ← 1	EI3	Enable interrupt 3
	SETC			SETBIT	RE	C_index	RE (C_index) ← 1	C	Set Carry flag
	EDMA			SETBIT	RE	DE_index	RE (DE_index) ← 1	DE	Enable DMA
	CLR	Rd, n		CLRBIT	Rd	n	Rd(n) ← 0	Z, N ou outra (se Rd for RE)	n ∈ [0 .. 15] Se Rd=RE, afecta apenas RE(n)
	DI			CLRBIT	RE	IE_index	RE (IE_index) ← 0	EI	Disable interrupts
	DI0			CLRBIT	RE	IE0_index	RE (IE0_index) ← 0	EI0	Disable interrupt 0
	DI1			CLRBIT	RE	IE1_index	RE (IE1_index) ← 0	EI1	Disable interrupt 1
	DI2			CLRBIT	RE	IE2_index	RE (IE2_index) ← 0	EI2	Disable interrupt 2
	DI3			CLRBIT	RE	IE3_index	RE (IE3_index) ← 0	EI3	Disable interrupt 3
	CLRC			CLRBIT	RE	C_index	RE (C_index) ← 0	C	Clear Carry flag
	DDMA			CLRBIT	RE	DE_index	RE (DE_index) ← 0	DE	Disable DMA
	CPL	Rd, n		CPLBIT	Rd	n	Rd(n) ← Rd(n) ⊕ 1	Z, N ou outra (se Rd for RE)	n ∈ [0 .. 15] Se Rd=RE, afecta apenas RE(n)
	CPLC			CPLBIT	RE	C_index	RE (C_index) ← RE(C_index) ⊕ 1	C	Complement Carry flag

Classe	Sintaxe em *assembly*		Campos da instrução (16 bits)				Acções	Flags afectadas	Comentários
			1º opcode (4bits)	2º opcode (4bits)	1º operando (4bits)	2º operando (4bits)			
Instruções de bit	SHR	Rd, n	BITOP	SHR	Rd	n	n>0 : C ← Rd(n-1) n>0 : Rd ← 0{n} || Rd(15..n)	Z, N, C	n ∈ [0 .. 15] Se n=0, actualiza Z e N (C não)
	SHL	Rd, n		SHL	Rd	n	n>0 : C ← Rd(15-n+1) n>0 : Rd ← Rd(15-n..0) || 0{n}	Z, N, C	n ∈ [0 .. 15] Se n=0, actualiza Z e N (C não)
	SHRA	Rd, n	ARITOP	SHRA	Rd	n	n>0 : C ← Rd(n-1) n>0 : Rd ← Rd(15){n} || Rd(15..n)	Z, N, C	n ∈ [0 .. 15] Se n=0, actualiza Z e N (C não)
	SHLA	Rd, n		SHLA	Rd	n	n>0 : C ← Rd(15-n+1) n>0 : Rd ← Rd(15-n..0) || 0{n}	Z, N, C, V	n ∈ [0 .. 15] Se n=0, actualiza Z e N (C não) V←1 se algum dos bits que sair for diferente do Rd(15) após execução
	ROR	Rd, n	BITOP	ROR	Rd	n	n>0 : C ← Rd(n-1) n>0 : Rd ← Rd(n-1..0) || Rd(15..n)	Z, N, C	n ∈ [0 .. 15] Se n=0, actualiza Z e N (C não)
	ROL	Rd, n		ROL	Rd	n	n>0 : C ←Rd(15-n+1) n>0 : Rd ← Rd(15-n..0) || Rd(15..15-n+1)	Z, N, C	n ∈ [0 .. 15] Se n=0, actualiza Z e N (C não)
	RORC	Rd, n		RORC	Rd	n	n>0 : Rd || C ← Rd(n-2..0) || C || Rd(15..n-1)	Z, N, C	n ∈ [0 .. 15] Se n=0, actualiza Z e N (C não)
	ROLC	Rd, n		ROLC	Rd	n	n>0 : C || Rd ← Rd(15-n+1..0) || C || Rd(15..15-n+2)	Z, N, C	n ∈ [0 .. 15] Se n=0, actualiza Z e N (C não)
	Livre								
Instruções de transferên-cia de dados	MOV	Rd, [Rs + off]	LDO	Rd	Rs	off/2	Rd ← Mw[Rs + off]	Nenhuma	off ∈ [-16 .. +14]
		Rd, [Rs]		Rd	Rs	0000	Rd ← Mw[Rs + 0000]	Nenhuma	
		Rd, [Rs + Ri]	LDR	Rd	Rs	Ri	Rd ← Mw[Rs + Ri]	Nenhuma	
		[Rd + off], Rs	STO	Rs	Rd	off/2	Mw[Rd + off] ← Rs	Nenhuma	off ∈ [-16 .. +14]
		Rd], Rs		Rs	Rd	0000	Mw[Rd + 0000] ← Rs	Nenhuma	
		[Rd + Ri], Rs	STR	Rs	Rd	Ri	Mw[Rd + Ri] ← Rs	Nenhuma	
	MOVB	Rd, [Rs]	XFER	LDB	Rd	Rs	Rd ← 0{8} || Mb[Rs]	Nenhuma	
		[Rd], Rs		STB	Rd	Rs	Mb[Rd] ← Rs(7..0)	Nenhuma	O byte adjacente a Mb[Rd] não é afectado
	MOVP	Rd, [Rs]		LDP	Rd	Rs	Rd ← Mw[Rs]	Nenhuma	Não usa memória virtual nem *caches* (para acesso aos periféricos)
		[Rd], Rs		STP	Rd	Rs	Mw[Rd] ← Rs	Nenhuma	

Classe	Sintaxe em *assembly*		Campos da instrução (16 bits)				Acções	Flags afectadas	Comentários
			1º opcode (4bits)	2º opcode (4bits)	1º operando (4bits)	2º operando (4bits)			
Instruções de transferência de dados	MOVL	Rd, k	MOVL	Rd	k		Rd ←k(7){8} || k	Nenhuma	k ∈ [-128 .. +127] k é extendido a 16 bits com sinal
	MOVH	Rd, k	MOVH	Rd	k		Rd(15..8) ← k	Nenhuma	k ∈ [0 .. 255] O byte de menor peso não é afectado
	MOV	Rd, k	MOVL	Rd	k		Rd ←k(7){8} || k	Nenhuma	Se k ∈ [-128 .. +127]
	MOV	Rd, k	MOVL MOVH	Rd Rd	k(7..0) k(15..8)		Rd ←k(7){8} || k(7..0) Rd(15..8) ← k(15..8)	Nenhuma	Se k ∈ [-32768 .. -129] ou k ∈ [+128 .. +32767]
	MOV	Rd, Rs	XFER	MOVRR	Rd	Rs	Rd ← Rs	Nenhuma	
	MOV	Ad, Rs		MOVAR	Ad	Rs	Ad ← Rs	Nenhuma	
	MOV	Rd, As		MOVRA	Rd	As	Rd ← As	Nenhuma	
	MOV	Rd, USP		MOVRU	Rd	xxxx	Rd ← USP	Nenhuma	O SP lido é o de nível utilizador, independentemente do bit NP do RE
	MOV	USP, Rs		MOVUR	xxxx	Rs	USP ← Rs	Nenhuma	O SP escrito é o de nível utilizador, independentemente do bit NP do RE
	SWAP	Rd, Rs		SWAPR	Rd	Rs	TEMP ← Rd Rd ← Rs Rs ← TEMP	Nenhuma	
	SWAP	Rd, [Rs] ou [Rs], Rd		SWAPM	Rd	Rs	TEMP ← Mw[Rs] Mw[Rs] ← Rd Rd ← TEMP	Nenhuma	Recomeçável sem reposição de estado mesmo que um dos acessos à memória falhe
	PUSH	Rd		PUSH	Rd	xxxx	Mw[SP-2] ← Rd SP ← SP – 2	Nenhuma	SP só é actualizado no fim para ser re-executável
	POP	Rd		POP	Rd	xxxx	Rd ← Mw[SP] SP ← SP + 2	Nenhuma	
	Livre								
	Livre								
	Livre								

Classe	Sintaxe em *assembly*		Campos da instrução (16 bits)				Acções	Flags afectadas	Comentários
			1º opcode (4bits)	2º opcode (4bits)	1º operando (4bits)	2º operando (4bits)			
Instruções de controlo de fluxo	JZ	etiqueta	COND	JZ	dif =(etiqueta – EIS)/2		Z=1: PC ← EIS + (2*dif)	Nenhuma	etiqueta ∈ [EIS - 256 .. EIS + 254]
	JNZ	etiqueta		JNZ	dif =(etiqueta – EIS)/2		Z=0: PC ← EIS + (2*dif)	Nenhuma	etiqueta ∈ [EIS - 256 .. EIS + 254]
	JN	etiqueta		JN	dif =(etiqueta – EIS)/2		N=1: PC ← EIS + (2*dif)	Nenhuma	etiqueta ∈ [EIS - 256 .. EIS + 254]
	JNN	etiqueta		JNN	dif =(etiqueta – EIS)/2		N=0: PC ← EIS + (2*dif)	Nenhuma	etiqueta ∈ [EIS - 256 .. EIS + 254]
	JP	etiqueta		JP	dif =(etiqueta – EIS)/2		(N∨Z)=0: PC ← EIS + (2*dif)	Nenhuma	etiqueta ∈ [EIS - 256 .. EIS + 254]
	JNP	etiqueta		JNP	dif =(etiqueta – EIS)/2		(N∨Z)=1: PC ← EIS + (2*dif)	Nenhuma	etiqueta ∈ [EIS - 256 .. EIS + 254]
	JC	etiqueta		JC	dif =(etiqueta – EIS)/2		C =1: PC ← EIS + (2*dif)	Nenhuma	etiqueta ∈ [EIS - 256 .. EIS + 254]
	JNC	etiqueta		JNC	dif =(etiqueta – EIS)/2		C =0: PC ← EIS + (2*dif)	Nenhuma	etiqueta ∈ [EIS - 256 .. EIS + 254]
	JV	etiqueta		JV	dif =(etiqueta – EIS)/2		V=1: PC ← EIS + (2*dif)	Nenhuma	etiqueta ∈ [EIS - 256 .. EIS + 254]
	JNV	etiqueta		JNV	dif =(etiqueta – EIS)/2		V=0: PC ← EIS + (2*dif)	Nenhuma	etiqueta ∈ [EIS - 256 .. EIS + 254]
	JA	etiqueta		JA	dif =(etiqueta – EIS)/2		A=1 : PC ← EIS + (2*dif)	Nenhuma	etiqueta ∈ [EIS - 256 .. EIS + 254]
	JNA	etiqueta		JNA	dif =(etiqueta – EIS)/2		A=0 : PC ← EIS + (2*dif)	Nenhuma	etiqueta ∈ [EIS - 256 .. EIS + 254]
	JEQ	etiqueta		JZ	dif =(etiqueta – EIS)/2		Z=1: PC ← EIS + (2*dif)	Nenhuma	etiqueta ∈ [EIS - 256 .. EIS + 254]
	JNE	etiqueta		JNZ	dif =(etiqueta – EIS)/2		Z=0: PC ← EIS + (2*dif)	Nenhuma	etiqueta ∈ [EIS - 256 .. EIS + 254]
	JLT	etiqueta		JLT	dif =(etiqueta – EIS)/2		N⊕V =1 : PC ← EIS + (2*dif)	Nenhuma	etiqueta ∈ [EIS - 256 .. EIS + 254]
	JLE	etiqueta		JLE	dif =(etiqueta – EIS)/2		((N⊕V)∨Z)=1 : PC ← EIS + (2*dif)	Nenhuma	etiqueta ∈ [EIS - 256 .. EIS + 254]
	JGT	etiqueta		JGT	dif =(etiqueta – EIS)/2		((N⊕V)∨Z)=0 : PC ← EIS + (2*dif)	Nenhuma	etiqueta ∈ [EIS - 256 .. EIS + 254]
	JGE	etiqueta		JGE	dif =(etiqueta – EIS)/2		N⊕V =0 : PC ← EIS + (2*dif)	Nenhuma	etiqueta ∈ [EIS - 256 .. EIS + 254]

Classe	Sintaxe em *assembly*	Campos da instrução (16 bits)				Acções	Flags afectadas	Comentários
		1º opcode (4bits)	2º opcode (4bits)	1º operando (4bits)	2º operando (4bits)			
Instruções de controlo de fluxo	JMP etiqueta	JMP	dif =(etiqueta – EIS)/2			PC ← EIS + (2*dif)	Nenhuma	etiqueta ∈ [EIS-4096 .. EIS+4094]
	JMP Rs	CODE	JMPR	xxxx	Rs	PC ← Rs	Nenhuma	
	CALL etiqueta	CALL	dif =(etiqueta – EIS)/2			Mw[SP-2] ← RER PC ← EIS + (2*dif) SP ← SP - 2	Nenhuma	etiqueta ∈ [EIS-4096 .. EIS+4094] SP só é actualizado no fim para ser re-executável
	CALL Rs	CODE	CR	xxxx	Rs	Mw[SP-2] ← RER PC ← Rs SP ← SP - 2	Nenhuma	SP só é actualizado no fim para ser re-executável
	CALLF etiqueta	CALLF	dif =(etiqueta – EIS)/2			RL ← RER PC ← EIS + (2*dif)	Nenhuma	etiqueta ∈ [EIS-4096 .. EIS+4094]
	CALLF Rs	CODE	CRF	xxxx	Rs	RL ← RER PC ← Rs	Nenhuma	
	RET		RET	xxxx	xxxx	PC ← Mw[SP] SP ← SP + 2	Nenhuma	
	RETF		RETF	xxxx	xxxx	PC ← RL	Nenhuma	
	SWE k		SWE	k		TEMP ← RE RE ← 0 Mw[SP-2] ← RER Mw[SP-4] ← TEMP PC ← Mw[BTE+SWE_index] SP ← SP - 4	Todas colocadas a zero	SP só é actualizado no fim para ser re-executável k ∈ [0 .. 255] SWE_index é o endereço da excepção SOFTWARE na tabela de excepções. Tem de se usar a pilha de sistema
	RFE		RFE	xxxx	xxxx	TEMP ← Mw[SP] PC ← Mw[SP+2] SP ← SP + 4 RE ← TEMP	Todas restauradas	SP só é actualizado no fim para ser re-executável Tem de se usar a pilha de sistema
	NOP		NOP	xxxx	xxxx		Nenhuma	Não faz nada
	Livre							
	Livre							
	Livre							
	Livre							
	Livre							
	Livre							
	Livre							
	Livre							

6 Aspectos adicionais do assemblador

6.1 Literais

Os literais são valores constantes (números ou cadeias de caracteres) podem ser especificados de seis formas no código *assembly*:

- **Valor numérico em binário**: para uma constante numérica ser interpretada em binário deve ser terminada com a letra b; são válidos valores entre 0b e 1111111111111111b.
- **Valor numérico em octal**: para uma constante numérica ser interpretada em binário deve ser terminada com a letra o; são válidos valores entre 0o e 177777o.
- **Valor numérico em decimal**: qualquer valor inteiro entre –32768 e 65535. Pode opcionalmente ser terminado com a letra d, embora tal seja assumido quando nenhuma outra base for indicada.
- **Valor numérico em hexadecimal**: para uma constante numérica ser interpretada em hexadecimal deve ser terminada com a letra h; são válidos valores entre 0h e ffffh. As constantes em hexadecimal cujo digito de maior peso é uma letra (a,b,c,d,e ou f) devem ser escritas com um zero antes da letra, de modo a distinguir a constante de uma variável. Assim a constante ffffh deverá ser escrita 0ffffh.
- **Caracter alfanumérico**: um caracter entre plicas, por exemplo, ‘g’, é convertido para o seu código ASCII.
- **Cadeia de caracteres alfanuméricos**: um conjunto de caracteres entre aspas, por exemplo “ola”, é convertido para um conjunto de caracteres ASCII.

É de notar, no entanto, que o uso de literais em código *assembly* (ou qualquer outra linguagem de programação) é desaconselhável. Em vez disso, deve-se usar o comando EQU para definir constantes (ver secção seguinte). Esta prática por um lado torna o código mais legível, pois o símbolo associado à constante dá uma pista sobre a acção que se está a tomar, e por outro lado permite uma actualização mais fácil do código, pois constantes que estão associadas não têm que ser alteradas em vários sítios dentro do código, mas simplesmente na linha do comando EQU.

6.2 Etiquetas

Para referenciar uma dada posição de memória, pode-se colocar uma etiqueta (*label*) antes da instrução que vai ficar nessa posição. A etiqueta consiste num nome (conjunto de caracteres alfanuméricos, mais o caracter ‘_’, em que o primeiro não pode ser um algarismo) seguida de ‘:’. Por exemplo,

```
AQUI:   INC R1
```

Se agora se quiser efectuar um salto para esta instrução, pode-se usar:

```
JMP     AQUI
```

em vez de se calcular o endereço em que a instrução INC R1 ficará depois da assemblagem.

6.3 Pseudo-Instruções

Chamam-se pseudo-instruções ao conjunto de comandos reconhecidos pelo assembler que não são instruções *assembly*, portanto não geram código binário no ficheiro objecto. A função das pseudo-instruções é, por um lado, controlar a forma como o código é gerado (por exemplo, indicando as posições de memória onde colocar o executável ou reservando posições de memória para dados), por outro lado, permitir definir etiquetas (constantes ou posições de memória) que tornam o código mais legível e mais fácil de programar. Nesta secção descrevem-se as pseudo-instruções usadas pelo *assembler* para o processador fornecido com o simulador de circuitos de arquitectura de computadores.

PLACE
Formato: PLACE <endereço>
Função: O assemblador usa um contador de endereços interno, que vai incrementando em cada instrução assemblada (assim, determina em que endereço fica cada instrução). O comando PLACE permite especificar no campo <endereço> um novo valor desse contador. Podem existir várias instruções PLACE no mesmo ficheiro *assembly* correspondentes a vários blocos de memória.

EQU
Formato: <símbolo> EQU <constante>
Função: o comando EQU permite associar um valor constante a um símbolo.

WORD
Formato: <etiqueta> WORD <constante>
Função: o comando WORD permite reservar uma posição de memória para conter uma variável do programa *assembly*, associando a essa posição o nome especificado em <etiqueta>. O campo constante indica o valor a que essa posição de memória deve ser inicializada.

STRING
Formato: <etiqueta> STRING <constante> [,<constante>]
Função: o comando STRING coloca em bytes de memória consecutivos cada uma das constantes nele definidas. Se qualquer dessas constantes for uma cadeia de caracteres o código ASCII de cada um deles é colocado sequencialmente na memória. A etiqueta fica com o endereço do primeiro caracter da primeira constante.

TABLE
Formato: <etiqueta> TABLE <constante>
Função: o comando TABLE reserva o número de posições de memória especificadas no campo <constante>. <etiqueta> fica com o endereço da primeira posição.