

P16.....	2
13.1 Register File.....	2
13.2 Mapa de memória.....	2
13.3 ISA.....	3
13.3.1 Transferência entre registos e memória.....	4
13.3.2 Transferência entre registos.....	7
13.3.3 Data Processing.....	10
13.3.4 Controlo de fluxo de execução.....	16
13.4 Programação <i>assembly</i> .....	19
13.4.1 Operações sobre dados.....	19
13.4.2 Controlo da execução.....	21
13.4.3 Avaliação de condições.....	24
13.4.4 Acesso a variáveis.....	25
13.4.5 Acesso a variáveis em <i>array</i> .....	27
13.4.6 Funções.....	28
Chamada de função sem parâmetros.....	29
Chamada de função com parâmetros.....	30
13.4.7 Stack.....	31

# P16

No sentido de aumentar a eficácia do CPU, relativamente ao processador de 8 bits, passaremos a um processador de 16 bits, que denominaremos P16.

A passagem a 16 bits permite:

- Aumentar o tipo e a dimensão dos parâmetros das instruções;
- Aumentar o número de instruções tornando assim os programas mais eficientes;
- Aumentar a dimensão do *register file*;
- Diminuir o número de acessos à memória;
- Melhorar a decodificação das instruções.

O P16 é um processador de 16 bits com a seguinte especificação:

- Arquitectura LOAD/STORE;
- Banco de registos (*Register File*) com 16 registos de 16 bits;
- Espaço de memória para código e dados 64K\*8 com possibilidade de acesso a 8 ou 16 bits;
- ISA, instruções têm tamanho fixo e ocupam uma única palavra de memória;
- Suporte à implementação de rotinas;
- Suporte à implementação de estrutura de dados *stack*.
- Acesso a memória e periféricos no mesmo espaço de endereçamento.
- Suporte a interrupções externas
- Sincronização na transferência de dados com dispositivos externos;
- Partilha de barramentos com outros dispositivos.

## 13.1 Register File

O banco de registos é constituído por 16 registos de 16 bits de R0 a R15, dividido em dois blocos de 8 registos. Um dos blocos, denominado por bloco baixo é constituído pelos registos de R0 a R7 e o bloco alto, constituído pelos registos de R8 a R15. Os registos do bloco baixo são de uso genérico e acessível por qualquer das instruções do processador que tenha um registo como parâmetro. Os registos do bloco alto só são acessíveis por algumas instruções, não podendo ser utilizados por todas as instruções de forma indiscriminada. No bloco alto, estão incluídos três registos de uso específico: R13/SP (*Stack pointer*), R14/LR (*Link register*) e o R15/PC (*Program counter*).

## 13.2 Mapa de memória

O espaço de endereçamento é de 64K \* 8 para código e dados. Embora o bus de dados seja de 16 bits o P16 pode realizar leituras ou escritas de 8 bits (*byte*) ou 16 bits (*word*). No caso da leitura de oito bits, são lidos sempre 16 bits da memória e internamente o CPU selecciona o *byte* de menor ou maior peso função do endereço par ou ímpar. Para um endereço par é seleccionado o *byte* de menor peso; para o endereço ímpar é seleccionado o *byte* de maior peso. Quanto ao programa, este tem que estar sempre alinhado a 16 bits, ou seja, sempre em endereços par. No caso da escrita de oito bits, é escrito o *byte* de menor peso do registo fonte.

### 13.3 ISA

O P16 mantém a mesma arquitectura do P8, ou seja, arquitectura LOAD/STORE, formada por três grupos de instruções: transferência, processamento e controlo de fluxo.

Todas as instruções têm a mesma dimensão e ocupam uma única palavra de memória (16 bits).

O acesso à memória pode realizar-se com os seguintes modos de endereçamento:

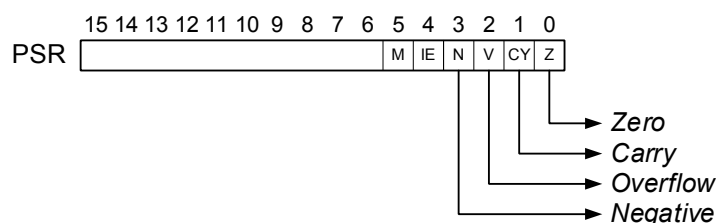
- indirecto;
- baseado e indexado.

No acesso indirecto, o endereço é especificado a 6 bits sem sinal relativo ao PC. Só disponível para leitura e o acesso é sempre realizado a 16 bits (*Word*).

No acesso baseado e indexado, o endereço é calculado pela soma de um registo base e um registo índice ou por um registo base e um índice constante de 4 bits.

As instruções de processamento de dados podem especificar três registos, ou dois registos e uma constante, e incluem instruções de deslocamento.

Estas instruções afectam quatro *flags* que são guardadas no registo denominado por *Processor Status Register (PSR)*. Este registo tem o seguinte formato:



Existem dois registos PSR, o CPSR e o SPSR. O registo CPSR contém o estado actual do processador e o registo SPSR é utilizado para salvar o conteúdo do CPSR quando ocorre um acontecimento específico que veremos adiante.

As instruções de controlo de fluxo condicional, fazem uso das *flags*. A *flag Zero (Z)* toma o valor lógico um, quando ao realizar uma operação lógica ou aritmética o valor resultante seja igual a zero. A *flag Carry (C)* fica a um, quando após uma operação aritmética resulte transporte para o dígito mais significativo, ou nas operações de deslocamento o último *bit* deslocado seja 1. A *flag Overflow (O)* fica ao valor lógico um, quando ao realizar uma subtracção ou uma soma o resultado excede o domínio, entendidos os valores como inteiros com sinal em código dos complementos para dois. A *flag Negative (N)*, após uma operação aritmética ou lógica fica com valor lógico do *bit* de maior peso do resultado.

Nas instruções de salto condicional ou incondicional, o endereço é calculado pela soma do registo PC com um deslocamento especificado em código dos complementos para dois com 10 bits.

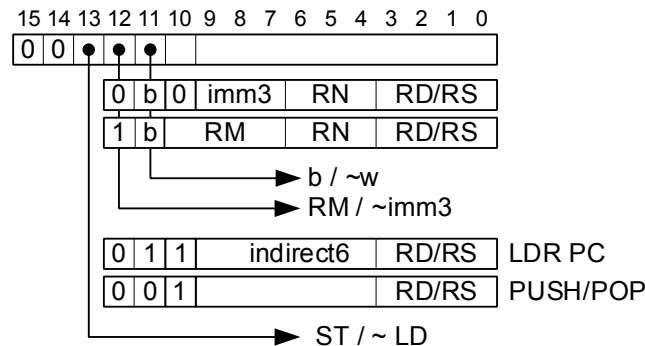
O valor indicado no deslocamento refere o número de instruções do qual se efectua o salto.

Para dar suporte à implementação de rotinas, o P16 põe disponível uma instrução específica – *branch with link (bl)* – que utiliza o registo R14 como registo de ligação (endereço de retorno).

O P16 implementa em memória uma estrutura de dados tipo *stack*. Para suporte a esta estrutura, o P16 utiliza o registo R13 como registo *stack pointer (SP)*, e põe disponíveis duas instruções: uma para empilhar o valor de um dos registos – instrução **push** e outra para desempilhar o valor para um dos registos – instrução **pop**. O *stack* aumenta em memória no sentido dos endereços menores. Os valores empilhados são sempre palavras de 16 *bits*.

### 13.3.1 Transferência entre registos e memória

Nas instruções de transferência, de e para memória, caso se pretenda realizar o acesso ao byte, acrescenta-se a letra **b** à direita da mnemónica. No caso da leitura de um *byte* de memória para registo são adicionados oito zeros na parte alta do *byte* lido. No caso da escrita de um *byte* em memória, o byte a ser escrito corresponde aos 8 bits de menor peso do registo fonte (**RS**) e o conteúdo da memória alterado será o *byte* de menor peso se o endereço for par ou o de maior peso se o endereço for ímpar. Para acesso à memória, o P16 apresenta dois tipos de endereçamento: indirecto e baseado e indexado. O conjunto das instruções de transferência entre registos e memória obedece aos seguintes formatos:



- RD/RS** – registo destino/fonte (R0 a R15)
- RM** – registo índice (R0 a R15)
- RN** – registo base (R0 a R7)
- indirect6** – constante de 6 bits sem sinal.
- imm3** – índice de 3 bits sem sinal.
- b** – indica se a transferência é de um *byte* ou de uma *word*.

## LDR RD, indirect6

Load indirect

Operação:  $RD \Leftarrow \text{mem}[\text{indirect6} * 2 + PC]$  (no flags affected)

Descrição: Escreve no registo **RD** a *word* guardada em memória cujo endereço é estabelecido pela adição do dobro do campo **indirect6** com o conteúdo do registo PC. **indirect6** é um valor positivo na gama 0 - 63.

Código: 

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	indirect6						RD			

Sintaxe: **ldr rd, indirect6**

Exemplo: **ldr r1, addr1** ; r1 recebe a *word* armazenada em memória no endereço addr1.

## LDR RD, [RN, imm3]

Load based indexed

Operação:  $RD \Leftarrow \text{mem}[RN + \text{imm3}]$  (no flags affected)

Descrição: Escreve no registo RD o conteúdo da memória cujo endereço é estabelecido pela soma do registo RN com a constante imm3 de 3 bits. A constante imm3 é multiplicada por 2 se o acesso for a word.

Código: 

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	b	0	imm3			RN			RD			

Sintaxe: **ldr rd, [rn, imm3]**

Exemplo: **ldr r0, [r7, 5]** ; r0 recebe 16 bits de memória, cujo endereço é dado por  $r7 + 5 * 2$ .

**ldrb r1, [r2, 3]** ; r1 recebe o byte cujo endereço é dado por  $r2 + 3$ .

**ldr r14, [r2]** ; r14 recebe 16 bits de memória, cujo endereço é dado apenas por r2.

## LDR RD, [RN, RM]

Load based indexed

Operação:  $RD \leftarrow \text{mem}[RN + RM]$  (no flags affected)

Descrição: Escreve no registo RD o conteúdo da memória cujo endereço é estabelecido pela soma do conteúdo do registo RN com o conteúdo do registo RM.

Código: 

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

Sintaxe: **ldr rd, [rn, rm]**

Exemplo: **ldr r8, [r15, r1]** ; r8 recebe a *word* cujo endereço é dado por  $r15 + r1$

**ldrb r2, [r9, r7]** ; r2 recebe o byte cujo endereço é dada por  $r9 + r7$

## STR RS, [RN, imm3]

Store indexed

Operação:  $RS \rightarrow \text{mem}[RN + \text{imm3}]$  (no flags affected)

Descrição: Escreve o conteúdo do registo RS na memória, cujo endereço é estabelecido pela soma do conteúdo do registo RN com a constante imm3.

Código: 

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	b	0	imm3			RN			RS			

Sintaxe: **str rs, [rn, imm3]**

Exemplo: **str r9, [r7, 5]** ; o conteúdo de r9 é escrito na variável de endereço  $r7 + 5 * 2$ .

**strb r1, [r2, 4]** ; a variável de endereço  $r2 + 4$ , recebe o byte de menor peso de r1.

**strb pc, [r1]** ; a memória de endereço r1, recebe o byte de menor peso do PC.

## STR RS, [RN, RM]

Store based indexed

Operação:  $RS \rightarrow \text{mem}[RN + RM]$  (*no flags affected*)

Descrição: Escreve o conteúdo do registo RS na memória, cujo endereço é estabelecido pela soma do conteúdo do registo RN com o conteúdo do registo RM.

Código: 

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

Sintaxe: **str rs, [rn, rm]**

Exemplo: **str r10, [r8, r2]** ; o registo r10 é escrito na variável de endereço r8 + r2.

**strb r1, [r2, r3]** ; a variável de endereço r2 + r3, recebe o LSB de r1.

## POP RD

Pop register

Operação:  $RD \Leftarrow \text{mem}[SP++]$  (*no flags affected*)

Descrição: Transfere para o registo RD o conteúdo da memória cujo endereço é estabelecido por SP. Posteriormente incrementa o conteúdo do registo SP.

Código: 

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

Sintaxe: **pop rd**

Exemplo: **pop r10** ; transfere para o registo R10 o conteúdo da memória cujo endereço é estabelecido pelo registo SP. O valor do registo SP é incrementado após a transferência.

**pop pc** ; pode corresponder ao retorno de uma função se tiver sido empilhado o registo LR.

## PUSH RS

Push register

Operação:  $\text{mem}[--SP] \Leftarrow RS$  (*no flags affected*)

Descrição: Decrementa de 2 o valor do registo SP e de seguida transfere para a memória cujo endereço é estabelecido por SP o conteúdo do registo RS.

Código: 

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

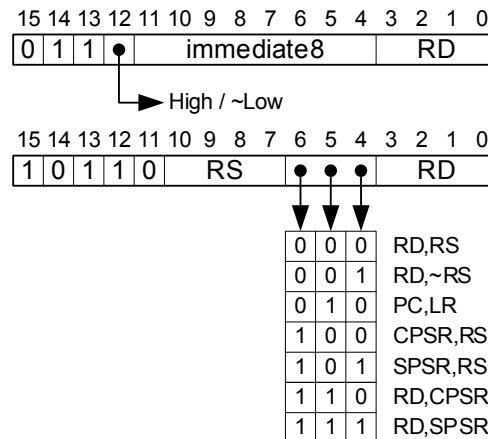
Sintaxe: **push rs**

Exemplo: **push r1** ; admitindo que o registo SP contém o valor 0x0050, o conteúdo do registo R1 é transferido para a memória de endereço 0x004e, ficando o registo SP com o valor 0x004e.

### 13.3.2 Transferência entre registros

O P16\_V1 põe disponíveis instruções para transferência entre registros.

Neste conjunto de instruções estão incluídas, para além das de transferência entre registros do *register file*, instruções com valores imediatos que permitem iniciar um registo com uma constante e instruções que permitem a transferência entre registros do *register file* e o PSR. As instruções de transferência entre registros obedecem ao seguinte formato:



- RD** – Registo destino (R0 a R15)
- RS** – Registo fonte (R0 a R15)
- immediate8** – Constante de oito *bits* a ser carregada no registo destino.

### **MOV RD,RS**

Move register

Operação:  $RD \Leftarrow RS$  (*no flags affected*)

Descrição: Copia o valor de RS para o registo RD.

Código: 

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

Sintaxe: **mov rd, rs**

Exemplo: **mov r1, r15**

### **NOT RD,RS**

NOT register

Operação:  $RD \Leftarrow \sim RS$  (*flags affected*)

Descrição: Escreve no registo RD o complemento bit a bit do conteúdo do registo RS.

Flags: 

N	V	CY	Z
X			X

Código: 

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

Sintaxe: **not rd, rm**

Exemplo: **not r0, r1** ; se r1 = 0101101000001111 então r0 = 1010010111110000

## MOV<sub>S</sub> PC, LR

Move to PC, Saved Link Register

Operação: PC  $\leftarrow$  SLR, CPSR  $\leftarrow$  SPSR (*all flags affected*)

Descrição: Copia para o PC, o valor do registo de link que foi preservado e copia o registo SPSR para o registo CPSR. Esta instrução concretiza o retorno de uma rotina de interrupção (ver capítulo 19).

Código: 

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

Sintaxe: **movs pc, lr**

Exemplo: **movs pc, lr**

## MOV RD, immediate8

Move immediate

Operação: RD  $\leftarrow$  0x00 | immediate8 (*no flags affected*)

Descrição: Escreve o valor do parâmetro immediate8 nos oito bits de menor peso do registo RD e coloca a zero os oito bits de maior peso.

Código: 

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0												
immediate8												RS			

Sintaxe: **mov rd, immediate8**

Exemplo: **mov r1, 0x2f** ; r1 = 0x002f

## MOVT RD, immediate8

Move immediate top

Operação: RD  $\leftarrow$  (immediate8 << 8) | (RD & 0xff) (*no flags affected*)

Descrição: Escreve o valor do parâmetro immediate8 nos oito bits de maior peso do registo RD permanecendo inalterados os 8 bits de menor peso do registo RD.

Código: 

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0												
immediate8												RS			

Sintaxe: **movt rd, immediate8**

Exemplo: **movt r1, 0x2f** ; se o registo r1 tiver inicialmente o valor 0x1234, após a instrução fica com o valor 0x2f34.

## MSR CPSR, RS

Move to CPSR, Register

Operação: CPSR  $\leftarrow$  RS

Descrição: Escreve no registo CPSR o conteúdo do registo RS.

Código: 

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



Sintaxe: **msr cpsr, rs**

Exemplo: **msr cpsr, r12** ; o conteúdo do registo r12 é transferido para o registo CPSR.

## ***MRS SPSR, RS***

Move to SPSR Register

Operação: **SPSR  $\leftarrow$  RS**

Descrição: Escreve no registo SPSR o conteúdo do registo RS.

Código: 

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

Sintaxe: **mrs spsr, rs**

Exemplo: **mrs spsr, r4** ; o conteúdo do registo r4 é transferido para o registo SPSR.

## ***MRS RD, CPSR***

Move to Register CPSR

Operação: **RD  $\leftarrow$  CPSR**

Descrição: Escreve no registo RD o conteúdo do registo CPSR.

Código: 

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

Sintaxe: **mrs rd, cpsr**

Exemplo: **mrs r0, cpsr** ; o conteúdo do registo CPSR é transferido para o registo r0.

## ***MRS RD, SPSR***

Move to Register SPSR

Operação: **RD  $\leftarrow$  SPSR**

Descrição: Escreve no registo RD o conteúdo do registo SPSR.

Código: 

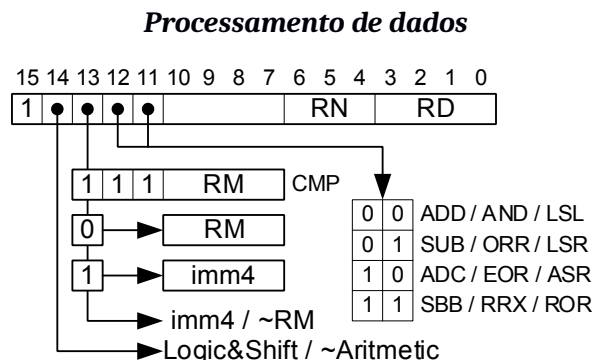
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0						1	1	1			RD

Sintaxe: **mrs rd, spsr**

Exemplo: **mrs r1, spsr** ; o conteúdo do registo SPSR é transferido para o registo r1.

### 13.3.3 Data Processing

O resultado destas instruções são um registo qualquer do *register file*. As instruções de processamento obedecem ao seguinte formato:



- RD** – registo destino (R0 a R15)
- RN** – registo do operando A (R0 a R7)
- RM** – registo do operando B (R0 a R15)
- imm4** – constante de 4 bits sem sinal

#### Adição

A adição pode ser realizada entre registos, ou entre um registo e uma constante, sendo que a constante é de quatro bits estendida com zeros à esquerda. Na adição com registo pode ser ainda adicionado o bit C. As *flags* resultantes da operação N, V, C e Z são armazenadas no registo CPSR.

### **ADD RD, RN, RM**

Add registers

Operação:  $RD \leftarrow RN + RM$  (*all flags affected*)

Descrição: Escreve no registo RD o resultado da adição do conteúdo do registo RM, com o conteúdo do registo RN.

Código: 

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

Sintaxe: **add rd, rn, rm**

Exemplo: **add r0, r2, r10** ; r0 = r2 + r10  
**add r12, r3** ; r12 = r12 + r3

### **ADC RD, RN, RM**

Add registers with carry flag

Operação:  $RD \leftarrow RN + RM + C$  (*all flags affected*)

Descrição: Escreve no registo RD o resultado da adição do conteúdo do registo RM, com o conteúdo do registo RN mais o bit C.

Código: 

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

Sintaxe: **adcs rd, rn, rm**

Exemplo: **adcs r10, r1, r15 ; r10 = r1 + r15 + C**

## **ADDS RD, RN, imm4**

Add constant

Operação:  $RD \leftarrow RN + imm4$  (*all flags affected*)

Descrição: Escreve no registo RD o resultado da adição do conteúdo do registo RN, com a constante imm4.

Código: 

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	imm4				RN			RD			

Sintaxe: **add rd, rn, imm4**

Exemplo: **add r15, r2, 15 ; r15 = r2 + 15**

**add r3, r3, 1 ; r3++**

## **Subtracção**

A subtracção pode ser realizada entre dois registos, ou entre um registo e uma constante, sendo que a constante é de quatro bits estendida com zeros à esquerda. Ao resultado da subtracção pode ser ainda subtraído o bit **C**. O resultado da subtracção é armazenado num qualquer registo, sendo que as *flags* daí resultantes (N, V, C, Z), são armazenadas no registo CPSR. A *flag C* daí resultante, quando a um indica que a subtracção produziu *borrow*.

## **SUB RD, RN, RM**

Subtract registers

Operação:  $RD \leftarrow RN - RM$  (*all flags affected*)

Descrição: Escreve no registo RD o resultado da subtracção do conteúdo do registo RN, pelo conteúdo do registo RM.

Código: 

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

Sintaxe: **sub rd, rn, rm**

Exemplo: **sub r0, r7, r12 ; r0 = r7 - r12**

## **SBB RD, RN, RM**

Subtract registers with carry

Operação:  $RD \leftarrow RN - RM - C$  (*all flags affected*)

Descrição: Escreve no registo RD o resultado da subtracção do conteúdo do registo RN, pelo conteúdo do registo RM, menos o bit C.

Código: 

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

Sintaxe: **sbb rd, rn, rm**

Exemplo: **sbb r10, r7, r12** ; r10 = r7 - r12

## ***SUB RD,RN,imm4***

Subtract constant

Operação:  $RD \leftarrow RN - imm4$  (*all flags affected*)

Descrição: Escreve no registo RD o resultado da subtracção do conteúdo do registo RN, pela constante imm4.

Código: 

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	1	imm4				RM		RD				

Sintaxe: **sub rd, rn, imm4**

Exemplo: **sub r10, r7, 8** ; r10 = r7 - 8

## ***Operações de Comparação***

A operação de comparação é realizada entre dois registos. Esta operação não produz resultado, apenas afecta o registo CPSR com as *flags* resultantes de uma subtracção entre dois registos.

## ***CMP RN,RM***

Compare registers

Operação:  $RN - RM$  (*all flags affected*)

Descrição: Afecta o registo CPSR com as flags resultantes da subtracção do conteúdo do registo RN, pelo conteúdo do registo RM.

Código: 

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

Sintaxe: **cmp rm, rn**

Exemplo: **cmp r7, r12**

## ***Operações Lógicas AND, OR e EOR***

As operações lógicas AND, OR e EOR são realizadas entre cada par de bits da mesma posição dos dois registos operados. O resultado da operação é armazenado num qualquer registo, sendo que as *flags* daí resultantes (N, V, C, Z), são armazenadas no registo CPSR.

## ***AND RD,RN,RM***

AND registers

Operação:  $RD \leftarrow RN \& RM$  (*flags affected*)

Descrição: Escreve no registo RD o resultado da Operação lógica AND bit a bit entre os conteúdos do registo RN e RM.

Flags: 

N	V	C	Z
X			X

Código: 

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

Sintaxe: **and rd, rn, rm**

Exemplo: **and r0, r7, r8**

## OR RD, RN, RM

OR registers

Operação:  $RD \leftarrow RN \mid RM$  (*flags affected*)

Descrição: Escreve no registo RD o resultado da Operação lógica OR bit a bit entre os conteúdos do registo RN e RM.

Flags: 

N	V	C	Y	Z
X				X

Código: 

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

Sintaxe: **or rd, rn, rm**

Exemplo: **or r0, r7, r12**

## EOR RD, RN, RM

XOR registers

Operação:  $RD \leftarrow RN \wedge RM$  (*flags affected*)

Descrição: Escreve no registo RD o resultado da Operação lógica XOR bit a bit entre os conteúdos do registo RM e RN.

Flags: 

N	V	C	Y	Z
X				X

Código: 

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

Sintaxe: **eor rd, rn, rm**

Exemplo: **eor r0, r7, r2**

## Operação de deslocamento (shift)

A operação de deslocamento consiste em deslocar o conteúdo de um registo para a direita ou para a esquerda. O número de posições que o registo é deslocado é um dos parâmetros da instrução.

O resultado da operação é armazenado num registo qualquer, sendo que as *flags* daí resultantes (N, V, C e Z), são armazenadas no registo CPSR.

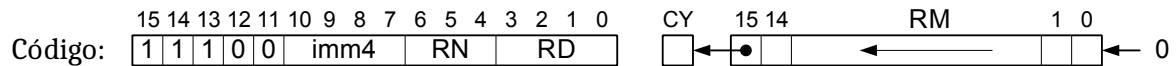
## LSL RD, RN, imm4

Logic Shift Left register

Operação:  $RD \leftarrow RM \ll \text{const4}$  (*all flags affected*)

Descrição: Escreve no registo RD o resultado do deslocamento para a esquerda dos 16 bits do registo RN. Esta Operação corresponde à multiplicação natural do conteúdo do registo RN por uma potência inteira de 2. O número de bits deslocados é

determinado pela constante imm4. Por cada bit deslocado, é inserido no bit de menor peso o valor lógico zero.



Sintaxe: `lsl rd, rn, imm4`

Exemplo: `lsl r0, r2, BIT_POSITION`

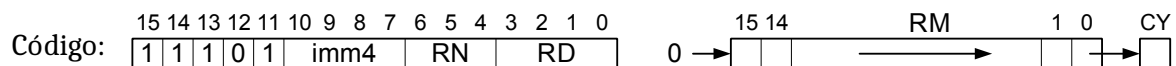
## LSR RD,RN,imm4

Logic Shift Right register

Operação:  $RD \leftarrow RM \gg imm4$  (*all flags affected*)

Descrição: Escreve no registo RD o resultado do deslocamento para a direita dos 16 bits do registo RN.

O número de posições deslocadas é definido pela constante imm4. Por cada posição deslocada, é inserido no bit de maior peso o valor lógico 0. Esta operação corresponde à divisão natural do conteúdo do registo RN por uma potência inteira de 2. O resto da divisão é obtido no bit C. A *flag* V fica com o valor lógico 1, se o bit de sinal do valor original for diferente do sinal do resultado.



Sintaxe: `lsr rd, rn, const4`

Exemplo: `lsr r10, r2, 12` ; r10 = r2 >> 12

## ASR RD,RN,imm4

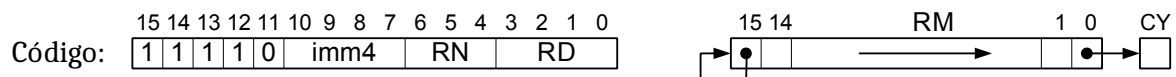
Arithmetic Shift Right register

Operação:  $RD \leftarrow RN \ggg imm4$  (*flags affected*)

Descrição: Escreve no registo RD o resultado do deslocamento para a direita dos 16 bits do registo RN. O número de posições deslocadas é definido pela constante imm4. Por cada *bit* deslocado, é inserido no bit de maior peso o mesmo valor do bit de sinal do conteúdo original de RN. Esta operação corresponde à divisão inteira do conteúdo do registo RN por uma potência inteira de 2, com manutenção do sinal.

Flags: 

N	V	CY	Z
X	0	X	X



Sintaxe: `asr rd, rn, imm4`

Exemplo: `asr r0, r2, 3` ; corresponde a dividir r2 por 8 com manutenção do sinal

## Rotação

A operação de rotação consiste em rodar os 16 bits de um registo.

O resultado da operação é armazenado num qualquer registo, sendo que as *flags* daí resultantes (N, V, C e Z), são armazenadas no registo CPSR.

## ROR RD, RN, imm4

Rotate Right register

Operação:  $RD \leftarrow RM \gg imm4$  (*flags affected*)

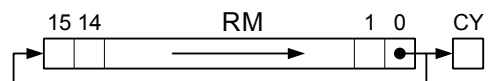
Descrição: Escreve no registo RD o resultado da rotação para a direita dos 16 bits do registo RN. O número de posições deslocadas é determinado pela constante imm4. Os bits de menor peso deslocados vão sendo inseridos nos bits de maior peso.

Flags:

N	V	CY	Z
X		X	X

Código:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	imm4			RN			RD			



Sintaxe: **ror rd, rn, const4**

Exemplo: **ror r0, r2, 5** ; roda o conteúdo de r2 para a direita 5 posições e coloca o resultado no registo r0

**ror r0, r2, 12** ; corresponde a rodar para a esquerda 4 vezes

## RRX RD, RN, imm4

Rotate Right Extended (17 bits rotate, 16 + carry)

Operação:  $RD \leftarrow RM \gg 1$  (*flags affected*)

Descrição: Escreve no registo RD o resultado da rotação de uma posição para a direita dos 16 bits do registo RN mais a *flag* C.

Flags:

N	V	CY	Z
X		X	X

Código:

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



Sintaxe: **rrx rd, rn**

Exemplo: **rrx r10, r2** ; r10 recebe r2 rodado para a direita uma vez incluindo o C.

### 13.3.4 Controlo de fluxo de execução

Nas instruções *branch* o endereço para onde se realiza o salto é sempre dado pela soma do conteúdo do registo **PC**, com uma constante definida a 10 bits – **offset10**. A constante **offset10** é multiplicada por 2, isto porque **offset10** corresponde ao número de instruções a saltar. A constante **offset10** é tomada como um valor inteiro com sinal, permitindo desta forma que o salto se realize para a frente ou para trás relativamente ao conteúdo do registo **PC**, ou seja, +510 ou menos -512 instruções. No cálculo de **offset10**, é necessário tomar em consideração que o valor do registo **PC** no momento da execução corresponde ao endereço imediatamente a seguir ao da instrução *branch*, pois o P16 realiza pré-preparação do **PC**.

As instruções de controlo de fluxo de execução obedecem ao seguinte formato:

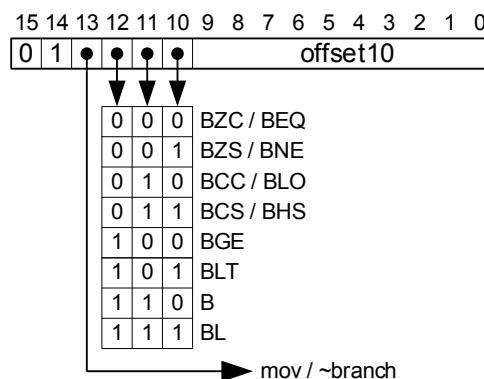


Figura 1: Instruções de controlo de fluxo de execução

### Salto condicional

Os saltos condicionais testam de forma directa, complementar ou combinada as *flags* **N**, **V**, **C**, e **Z**, no sentido de permitir diferentes avaliações sobre as características do resultado de uma operação aritmética ou lógica.

### BZS/BEQ

Branch if zero set or branch if equal

Operação: if (Z)  $PC \leftarrow PC + offset10 * 2$

Descrição: Coloca como conteúdo do registo PC o resultado da soma de PC com  $offset10 \times 2$ , se a *flag* Z se encontrar ao valor lógico 1.

Código: 

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	offset10									

Sintaxe: **bzs/beq offset10**

Exemplo: **bzs LABEL** ; if flag Z is true,  $PC = PC + offset$ ;  $offset = PC - LABEL$ .

**beq LABEL** ; if flag Z is true,  $PC = PC + offset$ ;  $offset = PC - LABEL$ .



## BZC/BNE

Branch if zero clear or not equal

Operação: if (!Z) PC  $\Leftarrow$  PC + offset10\*2

Descrição: Coloca como conteúdo do registo PC o resultado da soma de PC com offset10\*2, se a flag Z se encontrar ao valor lógico 0.

Código: 

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	offset10									

Sintaxe: **bzc offset10**

Exemplo: **bzc LABEL** ; if flags Z is false PC = PC + offset necessário para atingir LABEL

## BCS/BLO

branch if carry set or if below

Operação: if (C) PC  $\Leftarrow$  PC + offset10\*2

Descrição: Coloca como conteúdo do registo PC o resultado da soma de PC com offset10 \* 2, se a flag C se encontrar ao valor lógico 1.

Código: 

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	offset10									

Sintaxe: **bcs offset10**

Exemplo: **bcs LAB1** ; if flag C is true PC = PC + offset necessário para atingir LAB1

## BCC/BHS

branch if carry clear or if higher or same

Operação: if (!C) PC  $\Leftarrow$  PC + offset10\*2

Descrição: Coloca como conteúdo do registo PC o resultado da soma de PC com offset10 \* 2, se a flag C se encontrar ao valor lógico 0.

Código: 

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	offset10									

Sintaxe: **bcc/bhs offset10**

Exemplo: **bcc LAB1** ; if flag C is false PC = PC + offset necessário para atingir LAB1

## BGE

branch if Greater or Equal

Operação: if (N == V) PC  $\Leftarrow$  PC + offset10\*2

Descrição: Esta instrução é utilizada após a subtracção de dois registos A e B no sentido de determinar se A >= B entendidos A e B como dois valores inteiros, representados em código dos complementos para dois.

Código: 

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	0	offset10									

Sintaxe: **bge offset10**

Exemplo: **bge LAB1** ; *if flags N and V are equal* PC = PC + offset. O valor de offset é calculado por forma a atingir LAB1 a partir da posição corrente.

## BLT

branch if Less Than

Operação: if (N != V) PC  $\Leftarrow$  PC + offset10\*2

Descrição: Esta instrução é utilizada após a subtração de dois registos A e B no sentido de determinar se A < B entendidos A e B como dois valores inteiros, representados em código dos complementos para dois.

Código: 

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	1	offset10									

Sintaxe: **blt offset10**

Exemplo: **blt LAB1** ; *if flags N and V are different*, PC = PC + offset. O valor de offset é calculado por forma a atingir LAB1 a partir da posição corrente.

## Salto incondicional

O salto incondicional é relativo ao PC e tem como alcance -512 ou +512 instruções. Um salto incondicional de longo alcance poderá ser realizado utilizando a copia de um registo para o PC (ex: **mov r15, r1**).

## B

branch

Operação: PC  $\Leftarrow$  PC + offset10 \* 2

Descrição: Coloca como conteúdo do registo PC o resultado da soma de PC com offset10 \* 2.

Código: 

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	0	offset10									

Sintaxe: **b offset10**

Exemplo: **b LAB1** ; PC = PC + offset necessário para atingir LAB1

## BL

branch and link

Operação: R14  $\Leftarrow$  PC; PC  $\Leftarrow$  PC + offset10 \* 2

Descrição: Primeiramente transfere o conteúdo do registo PC para LR, seguidamente coloca como conteúdo do registo PC o resultado da soma de PC com offset10 \* 2.

Código: 

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	offset10									

Sintaxe: **bl offset10**

Exemplo: **bl func1** ; PC = PC + offset necessário para atingir o identificador func1

## 13.4 Programação assembly

Nesta secção exemplifica-se a utilização do P16 na implementação de abstrações presentes nas linguagens de alto nível, designadamente: expressões aritméticas e lógicas; estruturas de controlo de execução *if/else*, *for* e *while*; acesso a *arrays* e chamada de funções.

### 13.4.1 Operações sobre dados

#### Operações aritméticas

$a = a + b$ ;

Considerando que a variável “a” está em R0 e variável “b” está em R1:

```
add    r0, r0, r1
```

$a = c + b - d$ ;

Considerando. que a variável “a” está em R0, a variável “b” está em R1, a variável “c” está em R2 e a variável “d” está em R3:

```
add    r0, r2, r1
sub    r0, r0, r3
```

$a = b - 3$ ;

Considerando que a variável “a” está em R0, e a variável “b” está em R1:

```
sub    r0, r1, 3
```

#### Adicionar valores representados a 32 bit

O primeiro operando está em R1:R0, a parte baixa em R0 e a parte alta em R1, o segundo operando está em R3:R2 o resultado é depositado em R5:R4.

```
add    r4, r2, r0
adc    r5, r3, r1
```

#### Afectar com constante

<pre>char a = 3045;</pre>	<pre>; r0 - a mov    r0, 3045 &amp; 0xff movt   r0, 3045 &gt;&gt; 8</pre>
(a)	(b)

#### Programa 1: Afectar com constante

O valor da constante 3045 é igual a 1011 1110 0101 em binário. A instrução **mov** carrega o valor 1110 0101 na parte baixa de R0 e coloca a parte alta a zero. Se o valor da constante fosse inferior a 256 bastaria esta instrução. Sendo superior, é necessária a instrução **movt** para carregar 1011 na parte alta de R0 e assim formar o valor 3045 em R0. A instrução **movt** mantém a parte baixa do registo inalterada.

#### Deslocar um valor representado a 32 bits para a direita

O valor a deslocar encontra-se em R1:R0.

Deslocar uma posição	Deslocar 4 posições
<pre>lsr    r1, r1, 1 rrx    r0, r0</pre>	<pre>lsr    r0, r0, 4 lsl    r2, r1, 16 - 4 add    r0, r0, r2 lsr    r1, r1, 4</pre>
(a)	(b)

### Programa 2: Deslocar 32 bits para a direita

#### Deslocar um valor representado a 32 bits para a esquerda

O valor a deslocar encontra-se em R1:R0.

Deslocar uma posição.	Deslocar quatro posições.
<pre>lsl    r0, r0, 1 adc    r1, r1, r1</pre>	<pre>lsl    r1, r1, 4 lsr    r2, r0, 16 - 4 add    r1, r1, r2 lsl    r0, r0, 4</pre>
(a)	(b)

### Programa 3: Deslocar 32 bits para a esquerda

#### Rodar valores

Rodar uma palavra significa inserir nas posições de maior peso os *bits* que saem das posições de menor peso, se a rotação for para a direita, ou inserir nas posições de menor peso os *bits* que saem das posições de maior peso, se a rotação for para a esquerda.

Rodar o valor de R0 três posições para a direita.	Rodar o valor de R0 cinco posições para a esquerda.
<pre>ror    r0, r0, 3</pre>	<pre>ror    r0, r0, 16 - 5</pre>

#### Afectar um *bit* de um registo com zero

Afectar o *bit* de peso 12 de R0 com o valor zero mantendo o valor dos restantes *bits* de R0.

```
mov    r1, 0b11111111
movt   r1, 0b11101111
and    r0, r0, r1
```

#### Afectar um *bit* de um registo com um *bit* de outro registo

Afectar o *bit* de peso 4 de R0 com o valor do *bit* de peso 13 de R1.

```
shr    r2, r1, 13 - 4
mov    r3, 0b00010000
and    r2, r2, r3
orr    r0, r0, r3
orr    r0, r0, r2
```

## Multiplicar por constante

A multiplicação de uma variável por uma constante pode ser realizada de forma eficiente sem recorrer a instruções de multiplicação ou programa genérico de multiplicação. Veja-se o seguinte exemplo:

$$v * 23 = v * (16 + 4 + 2 + 1) = v * 16 + v * 4 + v * 2 + v * 1$$

A constante é decomposta em parcelas de valor igual a potências de 2. As multiplicações são realizadas por instruções de deslocamento.

<pre>int a; int b = a * 23;</pre>	<pre>;r0 - a, r1 - b mov    r1, r0      ; * 1 lsl    r0, r0, 1 add    r1, r1, r0   ; * 2 lsl    r0, r0, 1 add    r1, r1, r0   ; * 4 lsl    r0, r0, 2 add    r1, r1, r0   ; * 16</pre>
---------------------------------------	---

(a)

(b)

Programa 4: Multiplicar por constante

### 13.4.2 Controlo da execução

#### if; if/else

<pre>if (i != 0)     f += 3; g--;</pre>	<pre>;r5 - i, f - r2, g - r4 add    r5, r5, 0 beq    label add    r2, r2, 3 label: sub    r4, r4, 1</pre>
---	---

(a)

(b)

Programa 5: if

As instruções de salto condicional baseiam-se no valor das *flags* do processador. A avaliação booleana de expressões consistem em realizar operações aritméticas ou lógicas que afectem as *flags* com valores que sejam conclusivos.

No exemplo acima, a instrução **add r5, r5, 0** afecta a *flag* Z com um se **r5** for zero. A instrução **beq** salta sobre a instrução **add r2, r2, 3**, que corresponde a **f += 3**, se a *flag* Z for um, o que significa que **i (r5)** é igual a zero. A condição de salto da instrução *assembly* é contrária à condição definida na linguagem de alto nível.

<pre> if (i == j) {     f &lt;= 1;     i++; } else {     f &gt;= 2;     i -= 2; } j++; </pre>	<pre> 1    ;r5 = i, r3 = j, r2 = f 2        cmp    r5, r3 3        bne    if_else 4        lsl    r2, r2, 1 5        add    r5, r5, 1 6        b      if_end 7    if_else: 8        lsr    r2, r2, 2 9        sub    r5, r5, 2 10   if_end: 11        add    r3, r3, 1 </pre>
(a)	(b)

Programa 6: *if/else*

A instrução **cmp r5, r3** realiza a subtração  $r5 - r3$  e afecta a *flag* Z com um se a diferença for zero, o que significa que **i**(**r5**) e **j**(**r3**) são iguais. No caso de **i** ser diferente de **j**, a *flag* Z é afectada com zero e a instrução **bne** salta por cima do bloco *if* (linhas 4, 5 e 6) directamente para o código do bloco *else* (linhas 8 e 9). No caso de **i** ser igual a **j** a instrução **bne** não realiza o salto e executa o bloco *if*. Este bloco termina com um salto incondicional (linha 6), para não executar o bloco *else* posicionado nos endereços imediatos.

## switch/case

<pre> switch (v) {     case 1:         a = 11;         break;     case 10:         a = 111;         break;     default:         a = 0; } </pre>	<pre> 1    ;r0 = v, r1 = a 2    switch_case1: 3        mov    r2, 1 4        cmp    r0, r2 5        bne    switch_case10 6        mov    r1, 11 7        b      switch_break; 8    switch_case10: 9        mov    r2, 10 10       cmp    r0, r2 11       bne    switch_default 12       mov    r1, 111 13       b      switch_break; 14    switch_default: 15       mov    r1, 0 16    switch_break: </pre>
(a)	(b)

Programa 7: *switch/case*

A implementação do *switch/case* consiste em encadear um conjunto de *ifs*, um para cada caso (*case*).

## do while

do { v >> 1; l += 1; } while (v != 0) {	1     ;r0 - v, r1 - 1 2     do_while: 3         lsr     r0, r0, 1 4         add     r1, r1, 1 5         sub     r0, r0, 0 6         bne     do_while
--	---

(a)

(b)

### Programa 8: do while

A programação *assembly* segue a ordem de escrita e de execução da programação em C – primeiro executa o corpo de instruções e no final avalia a condição.

## while

while (v != 0) { v >> 1; l += 1; }	1     ;r0 - v, r1 - 1 2     while: 3         sub     r0, r0, 0 4         beq     while_end 5         lsr     r0, r0, 1 6         add     r1, r1, 1 7         b       while 8     while_end: 9	1     ;r0 - v, r1 - 1 2     while: 3         b       while_cond 4     while_do: 5         lsr     r0, r0, 1 6         add     r1, r1, 1 7     while_cond: 8         sub     r0, r0, 0 9         bne     while_do
---	---	--

(a)

(b)

(c)

### Programa 9: while

O programa *assembly* da figura 9(b) é escrito e executado pela ordem da linguagem C – primeiro a avaliação da condição e depois o corpo de instruções do *while*. Com esta programação o processador executa 5 instruções durante o ciclo, entre elas duas instruções *branch* – linhas 3 a 7.

Na figura 9(c) o programa é escrito como num *do while*, mas com um salto incondicional (linha 3) para a avaliação da condição, porque esta deve ser executada em primeiro lugar. Esta programação resulta na supressão da execução de uma instrução *branch* durante o ciclo, relativamente à programação apresentada na figura 9(b), o que a torna preferível.

## for

for (i = 0, a = 1; i < n; ++i) { a <= 1; }	1     ;r0 - i, r1 - a, r2 - n 2     mov     r0, 0 3     mov     r1, 1 4     b       for_cond 5     for: 6         lsl     r1, r1, 1 7         add     r0, r0, 1 8     for_cond: 9         cmp     r0, r2 blo     for
--	---

(a)

(b)

### Programa 10: for

A instrução

```
for (exp1; exp2; exp3)
    statement;
```

é equivalente a

```
exp1;
while (exp2) {
    statement;
    exp3;
}
```

A programação *assembly* apresentada na figura10(b) reflete esta equivalência, com o *while* implementado da forma mais eficiente.

### 13.4.3 Avaliação de condições

#### Testar o valor de um *bit*

Testar o valor do *bit* da posição três do registo R0.

```
ror    r0, r0, 3
bcs    label
```

#### Verificar se o valor de um registo é zero

```
add    r5, r5, 0
beq     label
```

#### Comparar números

<pre>if (a &lt; b)     c = a;</pre>	<pre>;r0 ← a, r1 ← b, r2 ← c     cmp    r0, r1     bhs    if_end     mov    r2, r0 if_end:</pre>
(a)	(b)

#### Programa 11: Comparação “menor que”

O código do bloco *if* é colocado imediatamente após o código de avaliação da expressão. A condição da instrução de salto – *high or same* – é contrária à condição da linguagem de alto nível. Nestas instruções a mnemónica da condição aplica-se ao primeiro operando da instrução **cmp** anterior. Quem é maior ou igual é o valor do registo **r0** que corresponde à variável **a**.

<pre>if (a &gt; b)     c = a;</pre>	<pre>;r0 ← a, r1 ← b     cmp    r1, r0     bhs    label     mov    r2, r0 if_end:</pre>
(a)	(b)

#### Programa 12: Comparação “maior que”



Neste caso se se realizasse a mesma instrução de comparação (**cmp r0, r1**) a condição de salto seria *lower or same*. Como não existe no P16 instrução de salto com esta condição, a solução apresentada realiza a subtracção com os operandos trocados (**cmp r1, r0**) e continua a usar-se **bhs**.

	operação	números naturais	números relativos
<b>if (a &lt; b)</b>	<b>cmp r0, r1</b>	<b>bhs</b>	<b>bge</b>
<b>if (a &lt;= b)</b>	<b>cmp r1, r0</b>	<b>blo</b>	<b>blt</b>
<b>if (a &gt; b)</b>	<b>cmp r1, r0</b>	<b>bhs</b>	<b>bge</b>
<b>if (a &gt;= b)</b>	<b>cmp r0, r1</b>	<b>blo</b>	<b>blt</b>

*Tabela 1: Comparação de números*

Na tabela 1 apresentam-se soluções de programação para as quatro relações possíveis.

Na comparação de números relativos, codificados em código de complementos, devem ser utilizadas as instruções **bge** (*branch greater or equal*) ou **blt** (*branch less than*).

#### 13.4.4 Acesso a variáveis

No P16, o acesso a variáveis em memória faz-se utilizando o modo de endereçamento indirecto. Este modo consiste em utilizar o conteúdo de registos do processador como endereço de memória. Antes de realizar a operação de acesso à variável (LDR ou STR) é necessário carregar previamente o endereço da variável num registo do processador.

Transferência do conteúdo de uma variável alojada em memória para os registos do processador.

Variável <b>x</b> , representada a 8 <i>bits</i> , alojada na posição de memória de endereço 5	Variável <b>y</b> , representada a 16 <i>bits</i> , alojada nas posições de memória de endereços 6 e 7
<pre>mov r1, x ldrb r0, [r1]</pre>	<pre>mov r1, y ldr r0, [r1]</pre>
<p>A instrução <b>mov</b> afecta os 8 <i>bits</i> menos significativos de <b>r1</b> com o endereço da variável <b>x</b> (valor 5) e afecta o 8 <i>bits</i> mais significativos com zero.</p> <p>A instrução <b>ldrb</b> copia o conteúdo da posição de memória de endereço 5 (valor 0x23) para os 8 <i>bits</i> menos significativos de <b>r0</b> e afecta os 8 <i>bits</i> mais significativos com zero.</p>	<p>A instrução <b>mov</b> afecta os 8 <i>bits</i> menos significativos de <b>r1</b> com o endereço da variável <b>y</b> (valor 6) e afecta o 8 <i>bits</i> mais significativos com zero.</p> <p>A instrução <b>ldr</b> copia dois <i>bytes</i> da memória para o registo <b>r0</b>. O conteúdo da posição de memória de endereço 6 (valor 0x7a) para os 8 <i>bits</i> menos significativos de <b>r0</b> e o conteúdo da posição de memória de endereço 7 (valor 0x3e) para os 8 <i>bits</i></p>

	mais significativos – <i>little ended</i> .
--	---

Afectação **PUSH** de uma variável alojada em memória com o conteúdo de registos do processador.

Variável <b>x</b> , representada a 8 bit, alojada na posição de memória de endereço 5	Variável <b>y</b> , representada a 16 bit, alojada nas posições de memória de endereços 6 e 7
<pre>mov r1, x strb r0, [r1]</pre>	<pre>mov r1, y str r0, [r1]</pre>
<p>A instrução <b>mov</b> afecta os 8 bits menos significativos de <b>r1</b> com o endereço da variável <b>x</b> (valor 5) e afecta o 8 bits mais significativos com zero.</p> <p>A instrução <b>strb</b> copia o valor dos 8 bits menos significativos de <b>r0</b> para a posição de memória de endereço 5 (valor 0x9b).</p>	<p>A instrução <b>mov</b> afecta os 8 bit menos significativos de <b>r1</b> com o endereço da variável <b>y</b>, valor 6, e afecta o 8 bit mais significativos com zero.</p> <p>A instrução <b>str</b> copia o conteúdo do registo R0 para a memória. O valor dos 8 bits menos significativos de <b>r0</b> para a posição de memória de endereço 6 (valor 0xa4) e o valor dos 8 bits mais significativos de R0 (valor 0x67) para a posição de memória de endereço 7 – <i>little ended</i>.</p>

Esta forma de carregar o endereço da variável no registo do processador com a instrução **mov**, tem a limitação de apenas poder carregar endereços na gama 0 – 255 porque a dimensão do campo destinado à constante, no código desta instrução, é de 8 bit.

A solução geral para carregamento de endereços nos registos do processador passa por utilizar a instrução **ldr rd, label**. Esta instrução copia um valor representado a 16 bits para o registo indicado. Esse valor, que está alojado em memória no endereço definido por **label**, pode ser o endereço de uma variável ou outra constante.

Para aceder à posição de memória definida por **label**, esta instrução usa um método de endereçamento relativo ao PC. O código binário desta instrução tem um campo de 6 bits para codificar, em número de palavras de 16 bit (*words*), a distância, no sentido crescente, a que **label** se encontra do PC.

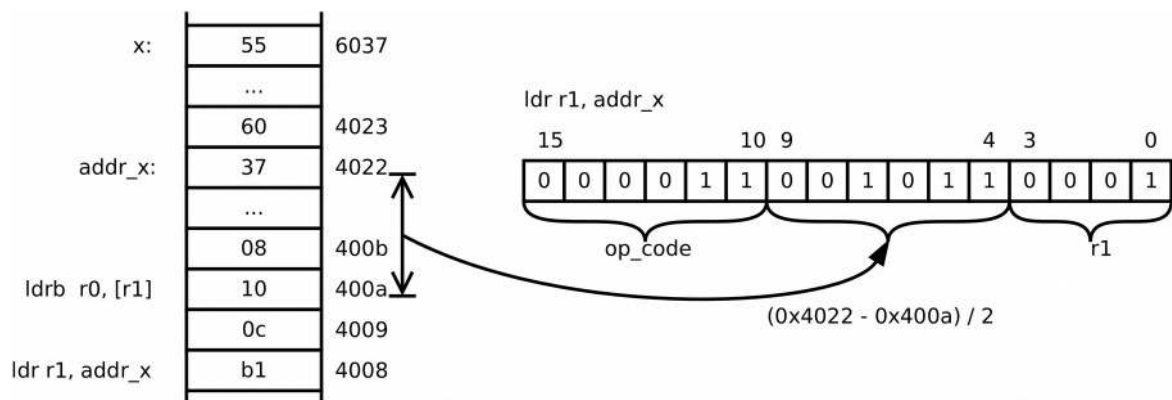


Figura 2: Ilustração da instrução *ldr rd, label*

Na figura 2, ilustra-se a disposição em memória, assim como os códigos binários da sequência de instruções para carregamento do valor da variável **x**, localizada no endereço **0x6037**, no registo **r0**.

```
ldr r1, addr_x
ldrb r0, [r1]
```

A instrução **ldr r1, addr\_x** carrega em **r1** o endereço da variável **x** (**0x6037**) que está armazenado em memória na posição indicada pela *label* **addr\_x** com endereço **0x4022**. Esta instrução determina o endereço de **addr\_x** adicionando ao **PC** (**0x400a**) a distância representada no campo **imm6** (**0x4022 = 0x400a + 0xb \* 2**). (Na fase de execução de uma instrução, o PC tem o endereço da instrução seguinte.)

#### 13.4.5 Acesso a variáveis em *array*

Os *arrays* são conjuntos de variáveis do mesmo tipo alojadas em posições de memória contíguas. As posições do *array* são definidas pelo índice.

Os acessos aos elementos do *array* são realizados pelas seguintes instruções de endereçamento baseado - indexado:

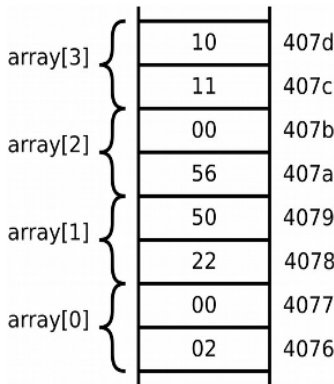
```
ldr rd, [rn, rm]    ldr rd, [rn, imm3]
str rd, [rn, rm]    str rd, [rn, imm3]
```

Estas instruções determinam o endereço de acesso somando o valor de **rn** com **rm** ou com a constante **imm3**. Em **rn** carrega-se o endereço da primeira posição do *array* e através de **rm** ou de **imm3** define-se a posição a que se pretende aceder.

<pre>char array[] = {     2, 0x23, 0x54, 0x10};  for (i = 0; i &lt; 10; ++i)     a += array[i]</pre>	<pre>array[3]  10  407b array[2]  54  407a array[1]  23  4079 array[0]  02  4078           4077</pre>	<pre>; r0 - array, r1 - i, r2 - a mov     r1, 0 b       for_cond for:     ldwb  r3, [r0, r1]     add   r2, r2, r3 for_cond:     cmp   r3, r1, 10     blo   for</pre>
(a)	(b)	(c)

Programa 13: Acesso a array de bytes

No programa 13 assume-se que o endereço inicial do *array* é previamente carregado registo **r0** – valor **0x4078**. Cada posição deste *array* ocupa uma posição de memória. Determinar o endereço duma dada posição do *array* consiste em somar o índice da posição ao endereço inicial, operação realizada pela instrução **ldrb r3, [r0, r1]**.

<pre>int array[] = {     2, 0x5022, 0x56, 0x1011};  for (i = 0; i &lt; 10; ++i)     a += array[i]</pre>		<pre>; r0 - array, r1 - i ; r2 - a      mov    r1, 0     b      for_cond for:     lsl    r3, r1, 1     ldr    r3, [r0, r3]     add    r2, r2, r3 for_cond:     cmp    r3, r1, 10     blo    for</pre>
(a)	(b)	(c)

*Programa 14: Acesso a array de words*

No programa 14, os elementos do *array* são valores representados a 16 *bits* – ocupam duas posições de memória. O acesso ao elemento **array[i]** é realizado pela instrução **ldr r3, [r0, r3]** que acede à posição de memória que resulta da soma de **r0** com **r3**. Assume-se que **r0** tem o endereço da primeira posição do *array* (endereço de **array[0]**) e **r3** a distância, em posições de memória, entre o endereço de **array[i]** e o endereço de **array[0]**. Esta distância é determinada pela instrução **lsl r3, r1, 1** que multiplica o índice (**r1**) pela dimensão de cada elemento (2 *bytes*).

### 13.4.6 Funções

“Função” é o termo que se usa na linguagem C para designar uma sequência de instruções que realizam uma tarefa específica. Também se usam na programação em geral termos como “rotina”, “subrotina”, “método” (em linguagens OO) ou “procedimento” (SQL), com aproximadamente o mesmo significado.

O objectivo da sua utilização é subdividir e organizar os programas, eventualmente extensos e complexos, em operações mais pequenas e mais simples.

Neste texto usa-se o termo “função”, por coerência com a utilização da linguagem C na descrição de algoritmos.

Uma funcionalidade importante num processador é o suporte à implementação de funções, ou seja, a existência de um mecanismo que possibilite invocar um mesmo troço de programa a partir de qualquer ponto do programa e retornar ao ponto de invocação. Esta funcionalidade é concretizada pela acção de chamada que transfere o fluxo de execução para o endereço onde reside a rotina (de modo semelhante a uma instrução de salto) e simultaneamente memoriza o valor corrente do PC. A acção de retorno transfere para o PC o valor anteriormente memorizado.

No P16, durante a execução de uma instrução, o PC contém o endereço da instrução a seguir à que está a ser executada. Assim, se este valor for guardado, fica assegurado o retorno à instrução seguinte.

A solução adoptada no P16 para memorizar o endereço de retorno foi utilizar o registo R14. Devido a esta funcionalidade este registo tem também o nome de registo de ligação (*link register* ou LR). A instrução **bl**, antes de afectar o PC com o endereço da rotina, transfere o valor actual do PC para o LR. O retorno ao ponto de invocação, faz-se copiando o conteúdo de LR para o PC, por exemplo, com a instrução **mov pc, lr**.

## Chamada de função sem parâmetros

Considere-se a sequência de duas chamadas à função **void delay()**

...	1	...
<b>delay();</b>	2	<b>bl delay</b>
...	3	...
<b>delay();</b>	4	<b>bl delay</b>
...	5	...

(a)
(b)

*Programa 15: Chamada a função sem parâmetros*

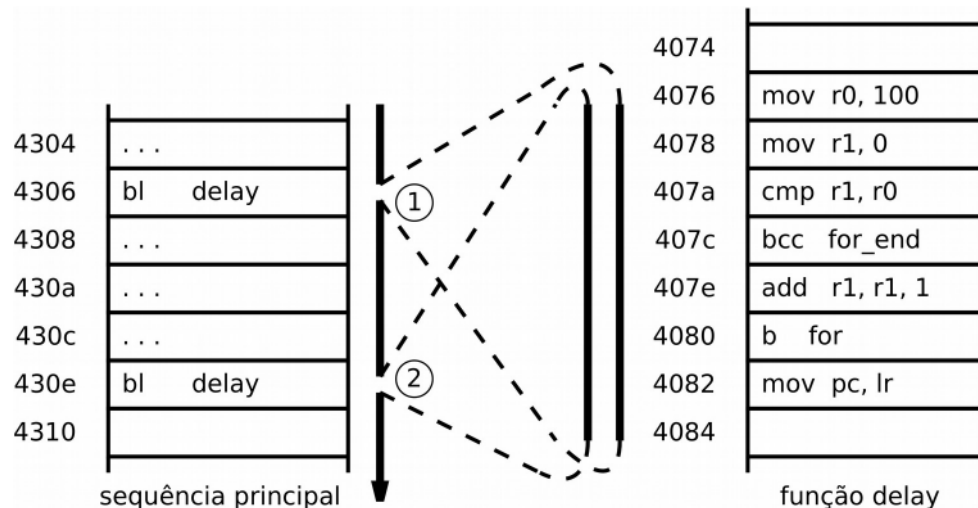
A chamada a funções sem parâmetros e sem valor de retorno corresponde apenas à execução da instrução **bl** para a *label* que define o início da função, neste caso **bl delay** nas linhas 2 e 4.

<b>void delay() {</b>	1	<b>delay:</b>
<b>for (int i = 0; i &lt; 100; ++i)</b>	2	<b>mov r0, 100</b>
<b>;</b>	3	<b>mov r1, 0</b>
<b>}</b>	4	<b>for:</b>
	5	<b>cmp r1, r0</b>
	6	<b>bcc for_end</b>
	7	<b>add r1, r1, 1</b>
	8	<b>b for</b>
	9	<b>for_end:</b>
	10	<b>mov pc, lr</b>

(a)
(b)

*Programa 16: Programação de função*

Na função **delay** a variável local **i** tem apenas o âmbito do *for*, por isso pode ser suportada num registo do processador, neste caso o registo **r1**.



*Figura 3: Ilustração de chamada a função sem parâmetros*

Na primeira chamada, no endereço 0x4306, o processador começa por transferir o conteúdo de PC (0x4308) para LR e em seguida afeta o PC com o endereço de **delay** (0x4076). Na segunda chamada, no endereço 0x430e, o processador realiza ações semelhantes, com a diferença do valor de LR ser 0x4310.

Ao executar a instrução **mov pc, lr** posicionada no final da função, no endereço **0x4082**, o processamento regressa ao endereço 0x4308 no caso da primeira chamada e regressa ao endereço **0x4310** no caso da segunda chamada.

## Chamada de função com parâmetros

Considere-se a sequência de duas chamadas à função **multiply** com a seguinte assinatura:

```
uint16_t multiply(uint8_t multiplying, uint8_t multiplier);
```

<pre>... product[2] = multiply(4, 10); product[3] = multiply(8, 10); ...</pre>	<pre>1  mov    r0, 4 2  mov    r1, 10 3  bl     multiply 4  str     r0, [r4, 4] 5  mov    r0, 8 6  mov    r1, 10 7  bl     multiply 8  str     r0, [r4, 6]</pre>
(a)	(b)

*Programa 17: Chamada a função com parâmetros*

A função **multiply** tem dois parâmetros – **multiplying** e **multiplier** e retorna valor. Na fase de chamada, antes da execução de **bl** é necessário definir os argumentos, o que tem que acontecer antes da função começar a executar. Nesta função usa-se o registo **r0** para passar o primeiro argumento e o registo **r1** para passar o segundo argumento. Na primeira chamada os argumentos são 4 e 10 e são carregados em **r0** e **r1**, respectivamente (linhas 1 e 2), na segunda chamada os argumentos são 6 e 10 e são passados do mesmo modo.

<pre> uint16_t multiply(uint8_t multiplying,                   uint8_t multiplier) {     uint16_t product = 0;     while ( multiplier &gt; 0 ) {         product -= multiplying;         multiplier--;     }     return product; } </pre>	<pre> 1  multiply: 2      mov    r2, 0 3  while: 4      sub    r1, r1, 0 5      beq    while_end 6      add    r2, r2, r0 7      sub    r1, r1, 1 8      b      while 9  while_end: 10     mov    r0, r2 11     mov    pc, lr </pre>
(a)	(b)

Programa 18: Programação de função com parâmetros

Ao programar a função **multiply** em *assembly* assume-se que os registos usados como parâmetros (**r0** e **r1**) já contêm os argumentos. A variável local **product** como não prevalece para além do âmbito desta função é suportada no registo **r2**, entre as linhas 2 e 10. O valor da função – o resultado da multiplicação – é depositado no registo **r0** (linha 10). Em ambas as chamadas o valor retornado pela função é guardado em *array* (linhas 4 e 8 do programa 17).

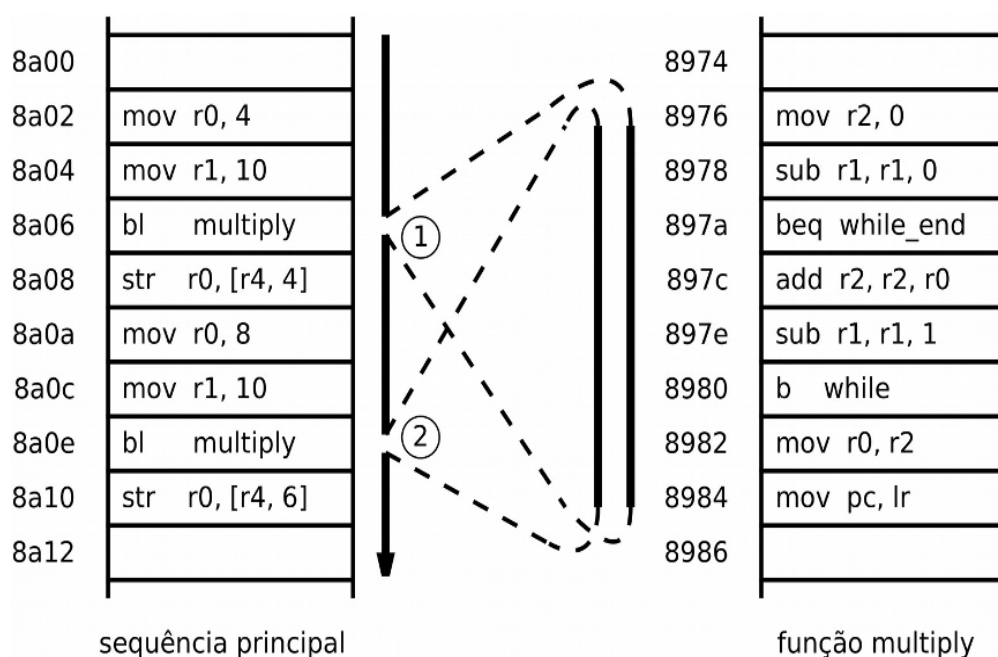


Figura 4: Ilustração de chamada com parâmetros

Na linha 11, a instrução **mov pc, lr** faz o processador retornar ao endereço 0x8a08 na primeira chamada e ao endereço 0x8a10 na segunda chamada. Nestas posições encontram-se instruções para processamento do valor retornado pela função no registo **r0**.

#### 13.4.7 Stack

O *stack* é uma zona de memória para salvaguarda temporária de informação. O seu nome advém do tipo de estrutura de dados que implementa ser do tipo *last-in-first-out* (LIFO), também designada por *stack*.



Esta estrutura de dados tem um funcionamento análogo a uma pilha de objetos: só se consegue retirar da pilha o objecto que se encontra no topo – o que foi lá colocado mais recentemente – e só se consegue colocar um novo objecto sobre o topo da pilha – apenas sobre o objecto anteriormente lá colocado.

O P16 dispõe de um registo específico e duas instruções para manusear o *stack*. O registo R13, neste contexto designado *stack pointer* (SP), destina-se a guardar permanentemente o endereço corrente do topo do *stack*. A instrução PUSH coloca o conteúdo de um registo no topo do *stack* e a instrução POP retira um valor do topo do *stack*, coloca-o num registo.

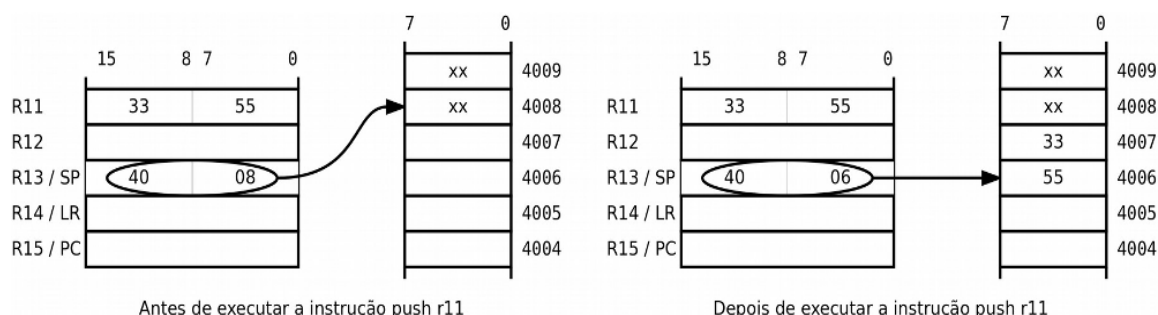
As instruções PUSH e POP transferem o conteúdo completo de um registo (uma *word*), ou seja, não é possível transferir apenas um *byte* como acontece com as instruções LDRB e STRB.

A instrução PUSH começa por decrementar o registo SP de duas unidades e em seguida transfere o conteúdo do registo indicado para a posição do *stack* definida por SP.

**push rs** é equivalente a      **sub sp, sp, 2**  
    **str rs, [sp]**

A instrução POP faz a operação inversa de PUSH. Começa por incrementar o registo SP de duas unidades e em seguida transfere o conteúdo da posição do *stack* definida por SP para o registo indicado.

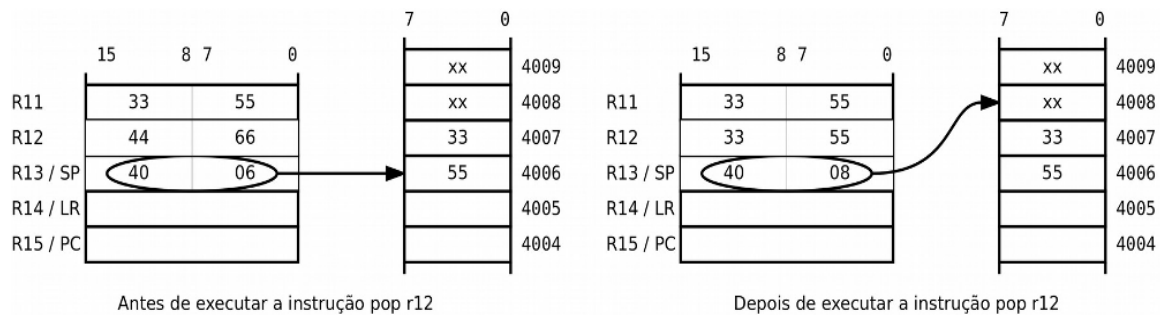
**pop rd** é equivalente a      **ldr rd, [sp]**  
    **add sp, sp, 2**



**Figura 5: Funcionamento da instrução PUSH**

A figura 5 ilustra o efeito da execução da instrução **push r11**. Antes da sua execução o SP contém o endereço 0x4008. Ao executar a instrução **push** o processador começa por decrementar o SP de duas unidades passando para 0x4006. Em seguida escreve o *byte* menos significativo de **r11** (0x55) na posição de endereço 0x4006 e o *byte* mais significativo de **r11** (0x33) na posição de endereço 0x4007. (A ordem segue o critério *little-ended*.)





**Figura 6: Funcionamento da instrução POP**

A figura 6 ilustra o efeito da execução da instrução **pop r12**. Antes da sua execução o SP contém o endereço 0x4006. Ao executar a instrução **pop**, o processador começa por transferir o conteúdo da posição de endereço 0x4006 (0x55) para o byte menos significativo de **r12** e o conteúdo da posição de endereço 0x4007 (0x33) para o byte mais significativo de **r12**. Em seguida incrementa o SP para o endereço 0x4008. O conteúdo das posições de memória 0x4006 e 0x4007 não é alterado, mas estas posições ficam disponíveis para serem reutilizadas na próxima instrução PUSH.