13. PDS16

No sentido de aumentar a eficácia do CPU passaremos a um processador de 16 bits, que denominaremos PDS16.
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;
- Diminuir o número de acessos à memória;
- Melhorar a descodificação das instruções.

O PDS16 é um processador de 16 bits com a seguinte especificação:

- Arquitectura LOAD/STORE;
- Banco de registos (*Register File*) com 8 registos de 16 bits;
- Espaço de memória para código e dados 32K*16 com possibilidade de acesso ao byte;
- ISA, instruções têm tamanho fixo e ocupam uma única palavra de memória;
- Suporte à implementação de rotinas;
- Espaço de I/O integrado no mapa de memória;
- Suporte a interrupções externas;
- Bus de dados multiplexado com endereço;
- Sincronização na transferência de dados com dispositivos externos;
- Partilha do bus por outros dispositivos.

13.1 Mapa de memória

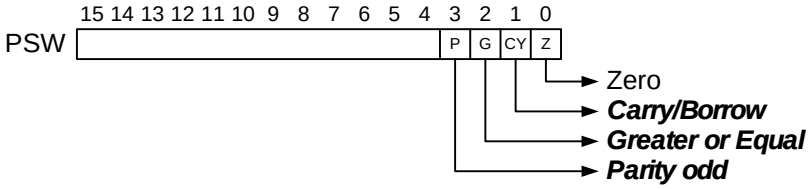
O mapa de memória é de 32K*16 para código e dados. Embora o bus de dados seja de 16 bits (*word*), o PDS16 pode realizar leituras ou escritas de 8 bits (*byte*). No caso da leitura de oito bits, são lidos sempre 16 bits da memória e internamente o CPU selecciona o *byte* de menor ou maior peso função do endereço ímpar ou par. Para um endereço par é seleccionado o byte de maior peso; para o endereço ímpar é seleccionado o *byte* de menor 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.

13.2 ISA

O PDS16 mantém a mesma arquitectura do PDS8, 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:

- Directo
- Indexado
- Baseado indexado

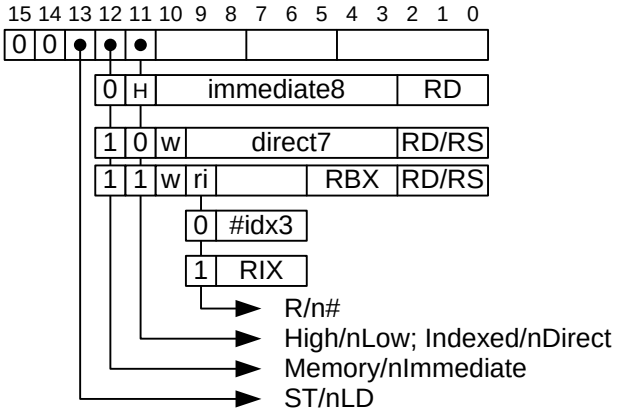
Nas instruções de transferência imediata, as constantes passam a ser de oito bits.
No acesso directo, o endereço é especificado a 7 bits, a leitura e escrita de dados em memória podem ser de 8 ou 16 bits (*Byte* ou *Word*).
As instruções de processamento passam a poder especificar quaisquer três registos do *Register File*, ou dois registos e uma constante, e passam a incluir instruções de deslocamento.
As instruções de processamento produzem quatro *flags* que são guardadas no registo denominado PSW e que corresponde ao registo R6 do *Register File*. O registo PSW tem o seguinte formato:



As instruções de controlo de fluxo condicional, só fazem uso das *flags* Z e CY. As restantes *flags* podem ser testadas realizando operações lógicas sobre o registo R6. A *flag Greater or Equal* fica activa quando ao realizar uma subtracção o diminuendo é maior ou igual ao diminuidor, entendidos os valores como inteiros com sinal em código dos complementos para dois. A *flag Parity odd*, fica activa quando após uma operação lógica ou aritmética o valor daí resultante contenha um número de bits com valor lógico 1 em quantidade ímpar.
Nas instruções de salto condicional ou incondicional, o endereço é calculado pela soma de um registo base com um deslocamento especificado em oito bits em código dos complementos para dois.
O valor indicado no deslocamento refere o número de instruções do qual se efectua o salto.
No PDS16, devido a um maior número de bits disponíveis, o ISA é mais extenso que no PDS8, o formato é mais estruturado e o número de parâmetros e a sua dimensão é maior.

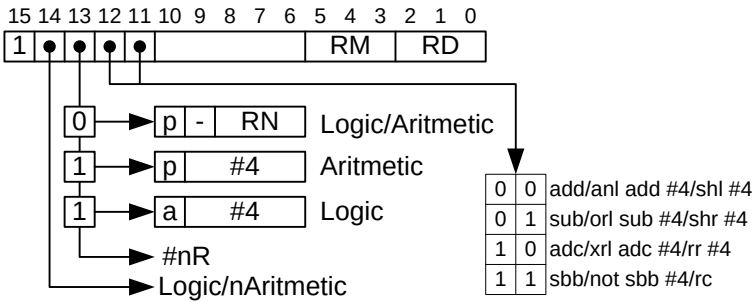
O conjunto das instruções obedece aos seguintes formatos:

Data Transfer



RD – Registo destino
RS – Registo fonte
RBX – Registo base
RIX – Registo índice
immediate8 – Constante de 8 bits sem sinal a ser carregada
direct7 – Constante de 7 bits sem sinal que indica o endereço
idx3 – Constante de 3 bits sem sinal que especifica o índice
H – Indica se a constante **immediate8** se destina à parte ***H****igh* ou *low* do registo.
w – Indica se o acesso à memória transfere uma ***W****ord* ou um *byte*.
ri – Determina se o modo de endereçamento é indexado ou baseado indexado.

Data Processing



RD – Registo destino

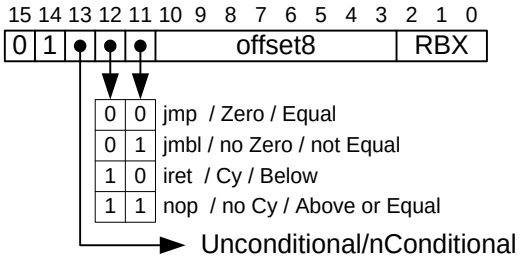
RM/RN – Registos dos operandos

#4 – Constante de 4 bits sem sinal

p – (*psw*) quando a zero indica que a operação não afecta o PSW. Se é indicado que PSW é afectado e o registo destino é o PSW então o PSW é afectado com as *flags* resultantes da operação, perdendo-se assim o resultado.

a – Atributo das instruções de deslocamento e rotação. Para as instruções SHL e SHR indica qual o valor lógico de SIN. Na instrução RC (*Rotate with Carry*) indica se a rotação se realiza para a direita quando a zero ou para esquerda quando a um. Na instrução RR indica qual o bit a ser inserido no bit de maior peso, ou seja, quando a zero é inserido o bit de menor peso existente no registo, quando a um insere o de maior peso.

Flow Control



RBX – Registo base

offset8 – Constante de 8 bits com sinal

13.2.1 *Data Transfer*

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 ímpar ou o de maior peso se o endereço for par. Para acesso à memória, o PDS16_V2 apresenta três tipos de endereçamento: directo, indexado e baseado indexado. O grupo *Data Transfer*, inclui para além das instruções de acesso à memória, duas instruções com modo de endereçamento imediato que permitem iniciar a parte alta ou a parte baixa de um registo com uma constante a oito bits.

LDI

Load immediate into low half word

Operação:	RD ← 0x00 | *immediate8* *(no flags affected)*
Sintaxe assembler:	ldi rd,#immediate8
Exemplo:	ldi r1, #0x2f
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	0	0	0	0	immediate8	RD

LDIH

Load immediate into high word

Operação:	RD ← *immediate8* | LSB(RD) *(no flags affected)*
Sintaxe assembler:	ldih rd,#immediate8
Exemplo:	ldih r1, #0x2f
Descrição:	Escreve o valor do parâmetro **immediate8** nos oito bits de maior peso do registo **RD** mantendo o byte menos significativo (LSB) de **RD** inalterado.

Código:

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

LD RD,direct7

Load direct

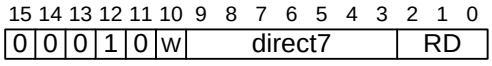
Operação: RD ← memória [0x00 | direct7] *(no flags affected)*

Sintaxe assembler: ld rd,direct7

Exemplo:
```
ldb   r1, var1    ; r1 recebe os 8 bits de memória referenciado por var1.
ld    r3, 0x25    ; r3 recebe os 16 bits da memória de endereço 0x0024.
```

Descrição: Escreve no registo **RD** o conteúdo da memória cujo endereço é estabelecido pelo parâmetro **direct7**.

Código:



LD RD,[RBX,#idx3]

Load indexed

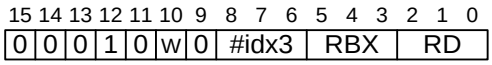
Operação: RD ← memória [RBX + idx3] *(no flags affected)*

Sintaxe assembler: ld rd,[rbx,#idx3]

Exemplo:
```
ld    r0, [r7, #5] ; r0 recebe a quinta word a seguir à referência dada por r7.
ldb   r1, [r2, #3] ; r1 recebe o terceiro byte cujo endereço é dado por r2+3.
```

Descrição: Escreve no registo **RD** o conteúdo da memória cujo endereço é estabelecido pela soma do registo **RBX** com a constante **idx3**. O valor da constante **idx3** é multiplicada por dois quando o acesso é a uma *word*.

Código:



LD RD,[RBX,RIX]

Load based indexed

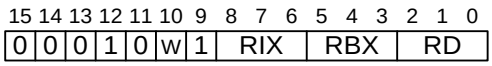
Operação: RD ← memoria[RBX + RIX] *(no flags affected)*

Sintaxe assembler: ld rd,[rbx,rix]

Exemplo:
```
ld    r0, [r1, r2]  ; r0 recebe a word cujo endereço é dado por r1+2*r2
ldb   r2,[r0, r1] ; r2 recebe o byte cujo endereço é dada por r0+r1
```

Descrição: Escreve no registo **RD** o conteúdo da memória cujo endereço é estabelecido pela soma do conteúdo do registo **RBX** com o conteúdo do registo **RIX**. O valor do registo **RIX** é multiplicado por dois quando o acesso é a uma *word*.

Código:



ST RS,direct7

Store direct

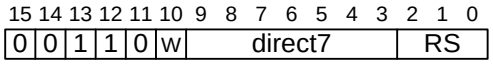
Operação: RS → memória [0x00 : direct7] *(no flags affected)*

Sintaxe assembler: st rs,direct7

Exemplo:
```
stb   r1, var1    ; var1 recebe o byte de menor peso do registo r1.
st    r3, 0x7f    ; a memória de endereço 0x007e recebe o conteúdo r3.
```

Descrição: Escreve na memória cujo endereço é estabelecido pelo parâmetro **dirct7** o conteúdo do registo **RS**.

Código:



ST RS,[RBX,#idx3]

Store indexed

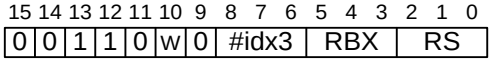
Operação: RS → memoria[RBX + idx3] *(no flags affected)*

Sintaxe assembler: st rs,[rbx,#idx3]

Exemplo:
```
st    r0, [r7, #5] ; a variável de endereço r7+5*2, recebe o conteúdo de r1.
stb   r1, [r2, #4] ; a variável de endereço r2+4, recebe o low byte de r1.
```

Descrição: Escreve na memória, cujo endereço é estabelecido pela soma do conteúdo do registo **RBX** com a constante **idx3**, o conteúdo do registo **RS**. O valor da constante **idx3** é multiplicada por dois quando o acesso é a uma *word*.

Código:



ST RS,[RBX,RIX]

Store based indexed

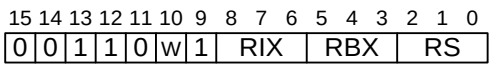
Operação: RS → memoria[RBX + RIX] *(no flags affected)*

Sintaxe assembler: st rs,[rbx,rix]

Exemplo:
```
st    r0, [r7, r2] ; a variável de endereço r7+r2*2, recebe o conteúdo de r2.
stb   r1, [r2, r3]  ; a variável de endereço r2+r3, recebe o low byte de r1.
```

Descrição: Escreve na memória, cujo endereço é estabelecido pela soma do conteúdo do registo **RBX** com o conteúdo do registo **RIX**, o conteúdo do registo **RS**. O valor do registo **RIX** é multiplicado por dois quando o acesso é a uma *word*.

Código:



13.2.2 *Data Processing*

Em algumas das instruções aritméticas ou lógicas pode-se adicionar à direita da mnemónica a letra **F** no sentido de indicar quais os registos que são ou não afectados. A adição da letra **F** indica que a operação não afecta o registo PSW. O sufixo **F** foi inserido porque o **PDS16** não tem nenhuma instrução explícita que permita transferir o conteúdo de um registo para outro. Esta acção pode ser concretizada pela instrução ANL Rx,Ry,Ry, ORL Rx,Ry,Ry ou ADD Rx,Ry,#0. Se colocarmos o sufixo **F** na instrução, o valor do PSW não é actualizado com as características do valor transferido, garantindo assim que o PSW permanece com a informação da última operação lógica ou aritmética, essa sim realizada com esse propósito. Como já foi anteriormente referido, caso o registo destino seja o PSW, este é afectado com as *flags* produzidas.

Adição

A adição 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. Em qualquer um dos casos (registo ou constante) pode ser ainda adicionado o bit CY. O resultado da adição é armazenado em qualquer registo do *register file*, e as *flags* daí resultantes P, CY, Z e GE=(*don´t care*) são armazenadas no PSW.

ADD RD,RM,RN

Add registers

Operação: RD ← RM + RN *(all flags affected)*

Sintaxe assembler: add rd, rm, rn

Exemplo: add r0, r7, r2

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	f	-	RN	RM	RD

ADDC RD,RM,RN

Add registers with CY flag

Operação: RD ← RM + RN + CY *(all flags affected)*

Sintaxe assembler: addc rd, rm, rn

Exemplo: addc r0, r7, r2

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

Código:

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

ADD RD,RM,#const4

Add constant

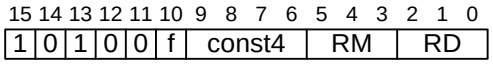
Operação: RD ← RM + const4 *(all flags affected)*

Sintaxe assembler: add rd, rm, #const4

Exemplo: add r0, r7, #15
addf r1, r2, #0 ; copia o conteúdo de r2 para r1 sem afectar *flags*

Descrição: Escreve no registo **RD** o resultado da adição do conteúdo do registo **RM**, com a constante **const4**.

Código:



ADDC RD,RM,#const4

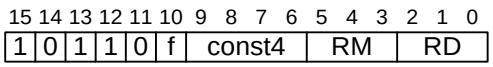
Add constant with CY flag

Operação: RD ← RM + const4 + CY *(all flags affected)*

Sintaxe assembler: add rd, rm, #const4

Exemplo: add r0, r7, #12

Código:



Descrição: Escreve no registo **RD** o resultado da adição do conteúdo do registo **RM**, com a constante **const4** mais o bit **CY**.

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 **CY**. O resultado da subtracção é armazenado num qualquer registo, sendo que as *flags* daí resultantes (P, GE, CY, Z), são armazenadas no registo PSW. A *flag* **CY** produzida, representa neste caso o *borrow* da subtracção.

SUB RD,RM,RN

Subtract registers

Operação: RD ← RM - RN *(all flags affected)*

Sintaxe assembler: sub rd, rm, rn

Exemplo:
```
sub   r0, r7,r2
subr  r2, r2, r3    ; compara o conteúdo de r2 com o de r3
```

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

Código:

SBB RD,RM,RN

Subtract registers with borrow

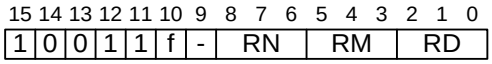
Operação: RD ← RM – RN - CY *(all flags affected)*

Sintaxe assembler: sbb rd, rm, rn

Exemplo: sbb r0, r7,r2

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

Código:



SUB RD,RM,#const4

Subtract constant

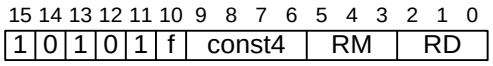
Operação: RD ← RM – const4 *(all flags affected)*

Sintaxe assembler: sub rd, rm, #const4

Exemplo: sub r0, r7, #5

Descrição: Escreve no registo **RD** o resultado da subtracção do conteúdo do registo **RM**, pelo constante **const4**.

Código:



SBB RD,RM,#const4

Subtract constant with CY

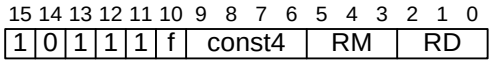
Operação: RD ← RM – const4 - CY *(all flags affected)*

Sintaxe assembler: sbb rd, rm, #const4

Exemplo: sbb r0, r7, #3

Descrição: Escreve no registo **RD** o resultado da subtracção do conteúdo do registo **RM**, pela constante de 4 bits **const4** menos o bit **CY**.

Código:



Operações Lógicas AND, OR, XOR e NOT

As operações lógicas AND, OR e XOR são realizadas exclusivamente entre dois registos. A operação NOT, por ser unária, é sobre um único registo. A operação lógica é realizada entre cada um dos 16 bits dos registos. O resultado da operação é armazenado num qualquer registo, sendo que as *flags* daí resultantes (P, GE=0, CY=0, Z), são armazenadas no registo PSW.

ANL

AND registers

Operação: RD ← RM & RN (*flags affected: Z, P, GE=0)*

Sintaxe assembler: anl rd, rm, rn

Exemplo:
```
anl   r0, r7,r2
anlr  r2, r2, r3   ; testa o conteúdo de r2 com a mascara em r3
```

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

Código:

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

ORL

OR registers

Operação: RD ← RM | RN (*flags affected: Z, P, GE=0)*

Sintaxe assembler: orl rd, rm, rn

Exemplo:
```
orl   r0, r7,r2
orlf  r2, r3, r3   ; copia o conteúdo de r3 para o registo r2
```

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

Código:

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

XRL

XOR registers

Operação: RD ← RM ^ RN (*flags affected: Z, P, GE=0)*

Sintaxe assembler: xrl rd, rm, rn

Exemplo: xrl r0, r7,r2

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

Código:

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

NOT

NOT register

Operação: RD ← ~RM (*flags affected: Z, P, GE=0)*

Sintaxe assembler: not rd, rm

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

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

Código:

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

Deslocamento (Shift)

A operação de deslocamento consiste em deslocar para a direita ou para a esquerda os 16 bits de um registo. O número de bits que o registo é deslocado é um dos parâmetros da instrução. Um outro parâmetro da instrução é o valor lógico a inserir no extremo do registo de cada vez que um bit é deslocado.

O resultado da operação é armazenado num qualquer registo, sendo que as *flags* daí resultantes (P, GE=0, CY, Z), são armazenadas no registo PSW.

SHL

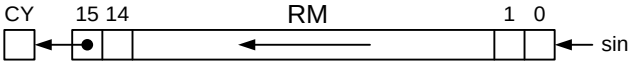
Shift left register

Operação: RD ← RM << const4 (*flags affected: Z, P, CY, GE=0)*

Sintaxe assembler: shl rd, rm, #const4,sin

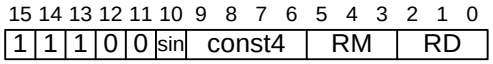
Exemplo: shl r0, r2, #BIT_POSITION, 0

Descrição: Escreve no registo **RD** o resultado do deslocamento para a esquerda dos 16 bits do registo **RM**.



O número de deslocamentos é determinado pela constante **const4**. Por cada bit deslocado, é inserido no bit de menor peso o valor lógico de **sin**. Esta operação, caso **sin** tenha o valor zero, corresponde a multiplicar o conteúdo do registo **RM** pela potência inteira de dois (2^{const4}).

Código:



SHR

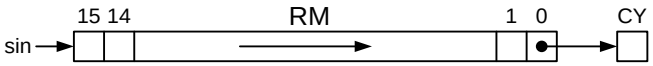
Shift right register

Operação: RD ← RM >> const4 (*flags affected: Z, P, CY, GE=0)*

Sintaxe assembler: shr rd, rm, #const4,sin

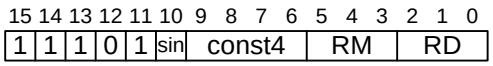
Exemplo: shr r0, r2, #BIT_POSITION, 0

Descrição: Escreve no registo **RD** o resultado do deslocamento para a direita os 16 bits do registo **RM**.



O número de deslocamentos é determinado pela constante **const4**. Por cada bit deslocado, é inserido no bit de maior peso o valor lógico de **sin**. Esta operação corresponde à divisão inteira do conteúdo do registo **RM** por uma potência inteira de 2. O resto da divisão é obtido no bit **CY**.

Código:



Rotação

A operação de rotação consiste em rodar os 16 bits de um registo. Em algumas das instruções o número de bits que o registo é rodado é um dos parâmetros da instrução.

O resultado da operação é armazenado num qualquer registo, sendo que as *flags* daí resultantes (P, GE=0, CY, Z), são armazenadas no registo PSW.

RRL

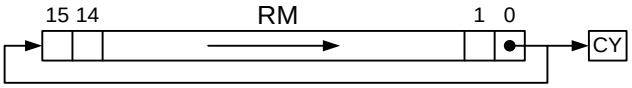
Rotate right least significant bit

Operação: RD ← RM >> const4 (*flags affected: Z, P, CY, GE=0)*

Sintaxe assembler: rrl rd, rm, #const4

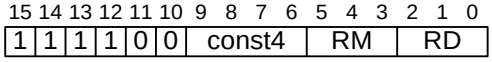
Exemplo: rrl r0, r2, #12 ; corresponde a rodar para a esquerda 4 vezes

Descrição: Escreve no registo **RD** o resultado da rotação para a direita dos 16 bits do registo **RM**.



O número de deslocamentos é determinado pela constante **const4**. Por cada bit deslocado, é inserido no bit de maior peso o bit de menor peso.

Código:



RRM

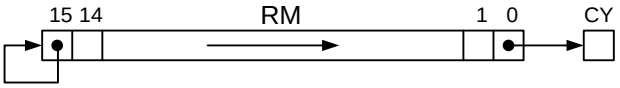
Rotate right must significant bit

Operação: RD ← RM << const4 (*flags affected: Z, P, CY, GE=0)*

Sintaxe assembler: rrm rd, rm, #const4

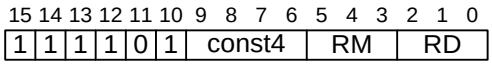
Exemplo: rrm r0, r2, #3 ; corresponde a dividir r2 por 8 com extensão de sinal

Descrição: Escreve no registo **RD** o resultado da rotação para a direita dos 16 bits do registo **RM**.



O número de deslocamentos é determinado pela constante **const4**. Por cada bit deslocado, é inserido no bit de maior peso o bit de sinal do conteúdo original de **RM**. Esta operação corresponde à divisão inteira do conteúdo do registo **RM** por uma potência inteira de 2, com extensão do sinal.

Código:



RCR

Rotate with carry right

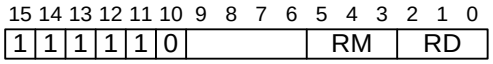
Operação: RD ← RM >> 1 (*flags affected: Z, P, CY, GE=0)*

Sintaxe assembler: rcr rd, rm

Exemplo: rcr r0, r2 ; corresponde a rodar para a direita uma vez o registo r2

Descrição: Escreve no registo **RD** o resultado da rotação para a direita dos 16 bits do registo **RM** mais o bit de *carry*.

Código:



RCL

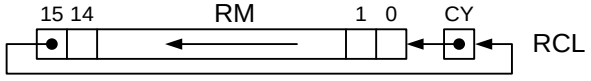
Rotate with carry left

Operação: RD ← RM << 1 (*flags affected: Z, P, CY, GE=0)*

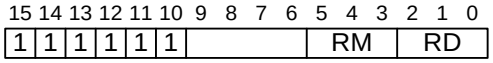
Sintaxe assembler: rcl rd, rm

Exemplo: rcl r0, r3 ; corresponde a rodar para a esquerda uma vez o registo r3

Descrição: Escreve no registo **RD** o resultado da rotação para a esquerda dos 16 bits do registo **RM** mais o bit de *carry*.



Código:



13.2.3 *Flow Control*

No caso dos *jumps* o endereço para onde se realiza o salto é sempre dado pela soma do conteúdo de um registo base **RBX**, com uma constante de 8 bits **offset8**. A constante **offset8** é multiplicada por 2, isto porque **offset8** corresponde ao número de instruções a saltar. A constante **offset8 é** tomada como inteiro com sinal, permitindo desta forma que o salto se realize para a frente ou para traz relativamente ao conteúdo do registo **RBX**. No cálculo de **offset8**, quando se utiliza o **PC** como registo **RBX,** é necessário tomar em consideração que o valor deste no momento da execução corresponde ao endereço imediatamente a seguir ao da instrução de *jump,* pois o PDS16 realiza pré preparação do PC. Quando na instrução, em vez de um registo, é indicada uma referência (*Label*) o registo base usado é o registo r7 (PC).

Salto condicional

Os saltos condicionais testam de forma directa ou complementar as *flags* **Z** e **CY**.

JZ

Jump if zero

Operação: *if* (Z) PC ← RBX + offset8*2

Sintaxe assembler: jz rbx,#offset8

Exemplo:
```
jz    LABEL    ; if flag Z true, PC = PC+offset necessário para atingir LABEL
je    r0,#CASE3 ; if equal, PC = r0+CASE3*2
```

Descrição: Coloca como conteúdo do registo **PC** o resultado da soma de **RBX** com **offset8***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	offset8	RBX

JNZ

Jump if not Zero

Operação: *if* (!Z) PC ← RBX + offset8*2

Sintaxe assembler: jnz rbx,#offset8

Exemplo:
```
jnz   LABEL    ; if flag Z false, PC = PC+offset8 necessário para atingir LABEL
jne   r1,#CASE  ; if not equal, PC = r1+CASE*2
```

Descrição: Coloca como conteúdo do registo **PC** o resultado da soma de **RBX** com **offset8***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	1	offset8	RBX

JC

Jump if carry

Operação: *if* (CY) PC ← RBX + offset8*2

Sintaxe assembler: jc rbx,#offset8

Exemplo:
```
jc    LAB1     ; if flag CY true, PC = PC+offset necessário para atingir LAB1
jb    r7,#CASE ; if below, PC = PC+CASE*2
```

Descrição: Coloca como conteúdo do registo **PC** o resultado da soma de **RBX** com **offset8***2, se a *flag* **CY** 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	1	0	offset8								RBX		

JNC

Jump if not carry

Operação: *if* (!CY) PC ← RBX + offset8*2

Sintaxe assembler: jnc rbx,#offset8

Exemplo:
```
jnc   LAB1     ; if flag CY false, PC = PC+offset necessário para atingir LAB1
jae   r7,#CASE ; if above or equal, PC = PC+CASE*2
```

Descrição: Coloca como conteúdo do registo **PC** o resultado da soma de **RBX** com **offset8***2, se a *flag* **CY** 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	1	1	offset8								RBX		

Salto incondicional

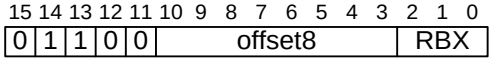
O salto incondicional tem o mesmo limite de alcance [-128, +127] que os saltos condicionados. Um salto de longo alcance terá que ser realizado utilizando a adição de um registo com o PC (add r7,r7,rx), ou utilizando para registo **RBX**, um qualquer registo contendo o endereço pretendido especificando deslocamento zero (jmp rx,#0).

JMP

Jump unconditional

Operação:	PC ← RBX + offset8*2
Sintaxe assembler:	jmp rbx,#offset8
Exemplo:	jmp LAB1 ; PC = PC+offset necessário para atingir LAB1 jmp r7,#CASE ; PC = PC+CASE*2
Descrição:	Coloca como conteúdo do registo **PC** o resultado da soma de **RBX** com **offset8***2.

Código:



Salto incondicional com ligação

Uma funcionalidade de extrema importância num processador é o suporte à implementação do conceito de rotina ou função, ou seja, a existência de um mecanismo no CPU que possibilite evocar um mesmo troço de programa a partir de qualquer ponto do programa que estejamos a executar e o retorno ao ponto de evocação. Esta funcionalidade é concretizada pelo par de instruções chamada (*call*) e retorno (*return*). A acção de *call* transfere o fluxo do programa para o endereço onde reside a rotina e simultaneamente memoriza o valor corrente do registo PC. A acção de *return* transfere para o PC o valor anteriormente memorizado.

Existem normalmente nos CPUs reais dois mecanismos: um em que o CPU (CISC) implementa na memória de dados uma estrutura de dados tipo pilha; e outro, onde o CPU (RISC) dispõe de um único registo específico para armazenar o valor do PC corrente. O mecanismo de pilha, permite o aninhamento de funções, enquanto o mecanismo de um único registo de ligação, o aninhamento tem que ser assegurado por uma estrutura de dados desenhada pelo programador. Como já vimos anteriormente, durante a execução de uma instrução o PC contém o endereço da instrução exactamente a seguir à que está a ser executada. Assim, se o valor corrente do PC for guardado, fica assegurado o retorno ao endereço exactamente a seguir ao da evocação da rotina.

No PDS16 a implementação deste mecanismo não necessitaria de nenhum mecanismo específico uma vez que o registo PC (R7) faz parte do *Register File* e assim a chamada da rotina poderia ser realizada pela seguinte sequência:

```
      addf r5,r7,#2    ; r5=L1
      jmp rotina
L1:
```

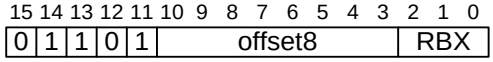
O retorno ao ponto de evocação seria realizado pela instrução `addf r7,r5,#0`. Por razões que adiante estudaremos, a chamada tem que ser feita de forma indivisível, e assim sendo, a solução adoptada no PDS16 foi a de eleger o registo R5 do *Register File* como registo LINK e foi adicionada uma instrução de salto `jmpl`. A instrução `jmpl`, antes de afectar o PC com o endereço da rotina, transfere o valor actual do PC para o registo LINK. O retorno ao ponto de evocação, faz-se copiando o conteúdo do registo LINK para o PC.

JMPL

Jump unconditional and link

Operação:	R5 ← PC; PC ← RBX + offset8*2
Sintaxe assembler:	jmpl rbx,#offset8
Exemplo:	jmpl func1 ;PC=PC+offset necessário para atingir o identificador func1 jmpl r3,#CASE ; PC = r3+CASE*2
Descrição:	primeiramente coloca como conteúdo do registo **LINK** o conteúdo do registo **PC**, seguidamente coloca como conteúdo do registo **PC** o resultado da soma de **RBX** com **offset8***2.

Código:



Retorno de interrupção (ver capitulo 18)

No retorno de uma ISR (*Interrupt Service Routine*) ao programa interrompido, é necessário realizar de forma indivisível as seguintes acções: cópia do conteúdo do registo **LINK** de interrupção para o registo **PC** e o conteúdo do registo **R0** de interrupção para o registo **PSW**.

IRET

Interrupt return

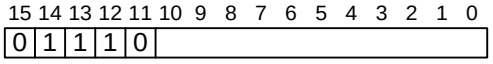
Operação: PC ← R5i ; PSW ← R0i

sintaxe assembler: iret

exemplo: iret

Descrição: Copia simultaneamente o conteúdo do registo R5i para o PC e o de R0i para o PSW. O banco de registos e a permissão de interrupções é reposta função do conteúdo do registo R0i.

Código:



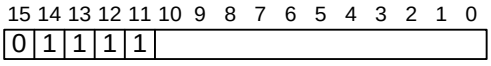
NOP

No operation

Operação: não realiza nenhuma acção

Sintaxe assembler: nop

Código:



13.2.4 Pseudo instruções

Baseado no conjunto das instruções a que o PDS16 obedece, o *assembler* põe disponível um conjunto de outras instruções, no sentido de facilitar a escrita e a compreensão dos programas.

`movf rd,rs` mover o conteúdo do registo **rs** para o registo **rd**. Esta instrução poderá ser traduzida por **`addf rd,rs,#0`**. Se em vez de **movf** utilizar **mov**, a *flag* Z indica se o valor copiado é igual a zero.

`incf rn` incrementar o conteúdo do registo **rn**. É traduzida por **`addf rn,rn,#1`**.

`inc rn` incrementa o conteúdo do registo **rn**, afectando as *flags* de Z e CY.

`decf rn` decrementar o conteúdo do registo **rn**. É traduzida por **`subf rn,rn,#1`**.

`dec rn` decrementa o conteúdo do registo **rn**, afectando as *flags* de Z e CY.

`ret` retornar de uma função. É traduzida por **`addf r7,r5,#0`**.

13.3 Exercícios

13.3.1 Ex1 (*switch case*)

Implemente a função `int cmd(char a)` que executa o comando referente ao carácter de entrada **a** e devolve -1 se comando inválido. Este exercício tem como único objectivo mostrar a implementação da instrução ***switch case*** utilizando uma tabela de saltos (*Jump Table*).

```
        .EQU    CASE_TAB_DIM,6

        .section startup
        .org    0
        jmp     main

        .data
lk:     .space  2
jumpTab:
        .word   default,case_k,case_w,case_x,case_y,case_z
caseTab:
        .byte   0,'k','w','x','y','z'

        .text
main:   ldi     r0,# 'w'
        jmpl    cmd
        jmp     $
/*
int cmd(char a) {
  switch (a) {
    case 'k' : return cmd_k();
    case 'x' : return cmd_w();
    case 'y' : return cmd_y();
    case 'z' : return cmd_z();
    default : return -1;
  }
}
*/
cmd:    ldi     r1,#low(caseTab)
        ldih    r1,#high(caseTab) ;r1=endereço da tabela de cases
        ldi     r2,#CASE_TAB_DIM-1
cmd_2:  ldb     r3,[r1,r2]
        subr    r0,r0,r3
        jz      cmd_1
        dec     r2
        jnz     cmd_2
cmd_1:  ldi     r1,#low(jumpTab)
        ldih    r1,#high(jumpTab) ;r1=endereço da tabela de jumps
        ld      r7,[r1,r2]
case_k: ...
        ldi     r0,#0
        ret
case_x: ...
        ldi     r0,#0
        ret
case_y: ...
        ldi     r0,#0
        ret
case_z: ...
        ldi     r0,#0
        ret
default:
        sub     r0,r2,#1        ;return -1 (0xffff)
        ret
```

13.3.2 Ex2 (*string length*)

Desenhe a função `int length(char str[])` que determina a dimensão da *string C* `str` terminada por zero. Escreva um programa `main` para teste da função.

```
        .EQU    FALSE,0
        .EQU    TRUE,1
        .EQU    GE,3

        .section startup
        .org    0
        jmp     main

        .data
stringC:
        .asciiz         "qual o comprimento desta stringC"

        .text
main:   ldi     r0,#low(stringC)
        ldih    r0,#high(stringC) ;r0=endereço da string
        jmpl    length
        jmp     $
/*
int length(char str[])  {
  for (int i=0; str[i] != '\0'; i++);
  return i;
}
*/
;R0=endereço de str[]
length:
        ldi     r1,#0           ;i=0
sortInt_for:
        ldb     r2,[r0,r1]      ;r2=str[i]
        orl     r2,r2,r2
        jz      sortIntEnd      ;if (str[i] == 0)
        inc     r1              ;i++
        jmp     sortInt_for
sortIntEnd:
        mov     r0,r1           ;devolve i em R0
        ret

        .end
```

13.3.3 Ex3 (*ascending sort*)

Desenhe a função `sortInt(int a[], int dim)` que ordena por ordem crescente um *array* de inteiros. Escreva um programa `main` para teste da função.

```
        .EQU    FALSE,0
        .EQU    TRUE,1
        .EQU    GE,3

        .section startup
        .org    0
        jmp     main

        .data
a_base: .space 2
f:      .space 1

        .text
main:   ldi     r0,#low(array)
        ldih    r0,#high(array) ;r0=endereço de array
        ldi     r1,#DIM
        jmpl    sortInt
        jmp     $
```

```
/*
sortInt(int a[], int dim)  { //algoritmo bubble sort
  boolean f;
  int aux;
  do {
    f=false;
    for (int i=0; i < dim-1; i++)
      if (a[i] > a[i+1]) {
        aux=a[i];
        a[i]=a[i+1];
        a[i+1]=aux;
        f=true;
      };
    dim--;
  } while(f);
}
*/
sortInt:
;R0=endereço de a[], R1=dim
        st      r0,a_base       ;preserva endereço base do array
        sub     r1,r1,#1        ;dim-1
sortInt_3:
;do
        ldi     r2,#FALSE
        stb     r2,f
        ldi     r2,#0           ;i=0
sortInt_for:
        subr    r1,r2,r1        ;i - (dim-1)
        shr     r6,r6,#GE,0
        jc      sortInt_1       ;if (i < dim-1)
        ld      r3,[r0,#0]      ;r3=a[i]
        ld      r4,[r0,#1]      ;r4=a[i+1]
        subr    r3,r4,r3        ;a[i+1]-a[i]
        shr     r6,r6,#GE,0     ;CY=GE
        jc      sortInt_2       ; if (a[i] < a[i+1])
        st      r4,[r0,#0]      ; troca a[i] com a[i+1]
        st      r3,[r0,#1]
        ldi     r4,#TRUE        ;f=TRUE
        stb     r4,f
sortInt_2:
        add     r0,r0,#2        ;word ptr++
        inc     r2              ;i++
        jmp     sortInt_for
sortInt_1:
        dec     r1              ;dim--
        ld      r0,a_base
        ldb     r4,f
        orl     r4,r4,r4
        jnz     sortInt_3       ;while (f)
        ret

        .bss
        .equ    DIM,20
array:  .space  DIM

        .end
```

13.3.4 Ex4 (multiplicação)

Desenhe a função `unsigned int mul(unsigned char op1,unsigned char op2)` que realiza a multiplicação entre os operandos op1 e op2 utilizando o algoritmo *add and shift* (ver capítulo 3.8). Os parâmetros op1 e op2 são números de 8 bits pelo que o resultado é expresso em 16 bits. O produto de **M** por **m** pode ser dado pela seguinte expressão:

$$P = M\,.\,m\ =\ M.(m_0 + 2.m_1 +\ 4.m_2 + \cdots + 2^{n-1}.m_{n-1})$$

$$P = M.m = M.m_0 + 2.M.m_1 + \cdots + 2^7.M.m_{n-1}$$

O produto de M pelas potências inteiras de 2, é obtido deslocando (*shift*) para a esquerda o valor de M.

```
        .EQU    op1,0
        .EQU    op2,1

        .section startup
        .org    0
        jmp     main

        .data
a_base: .space 2
f:      .space 1

        .text
main:   ldi     r0,#op1
        ldi     r1,#op2
        jmpl    mul
        jmp     $
/*
unsigned int mul(unsigned char op1, unsigned char op2)  { //algoritmo add shift
  unsigned int res=0;
    for (int i=8; i > 0; i--) {
      if ((op2 & 1) != 0)
        res=res+op1;
      op2=op2 >> 1;
      op1=op1 << 1;
    }
    return res;
}
*/


mul:    // r0=Op1 r1=Op2
        ldi     r2,#8   ;i=8
        ldi     r3,#0   ; res=0
        anl     r2,r2,r2
        jz      mul_end
        shr     r1,r1,#1,0
        jnc     mul_1
        add     r3,r3,r0
mul_1:
        shl     r0,r0,#1,0
        dec     r2
        jmp     mul_2
mul_end:
        mov     r0,r3
        ret

        .end
```

Na solução apresentada, a soma realiza-se a 16 bits, ou seja, com o número de bits igual aos do resultado, pelo que, se os operandos forem de 16 bits produzirão um resultado a 32 bits, implicando somas a 32 bits que, no caso do processador PDS16, implicava duas somas de 16 com arrasto. No sentido de diminuir o número de somas podemos realizar o mesmo algoritmo da seguinte forma:

$$P = M.m = M.m_0 + 2.M.m_1 + 2^2.M.m_2 + \cdots + 2^{15}.M.m_{15}$$

$$P = M.m = 2^{15}.M.(((((m_0/2 + m_1)/2 + m_2)/2 + m_3)/2 + \cdots + m_{14})/2 + m_{15})$$

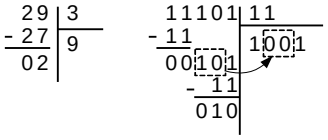
```
mul:    // r0=Op1 r1=Op2
        ldi     r2,#16  ; i=16
        ldi     r3,#0   ; res_high=0
        ldi     r4,#0   ; res_low=0
        anl     r2,r2,r2
        jz      mul_end
        shr     r1,r1,#1,0
        jnc     mul_1           ; if ((Op2 & 1) == 0)
        add     r3,r3,r0        ;res=res+op2*(2^16)
mul_1:
        rcr     r3,r3
        rcr     r4,r4   ;res=res/2
        dec     r2
        jmp     mul_2
mul_end:
        mov     r0,r3
        ret
```

13.3.5 Ex5 (divisão)

Desenhe a função

```
unsigned int div(unsigned int dividendo,unsigned int divisor,
                 unsigned int * resto);
```

que realiza a divisão não inteira entre dois operandos de 16 bits, utilizando o algoritmo *sub and shift*. A função `div` retorna o valor do quociente sendo o resto actualizado por referência. Os parâmetros são números de 16 bits pelo que o quociente e o resto também são expressos a 16 bits. O algoritmo que vamos implementar é o utilizado correntemente na divisão de números decimais. Na divisão de números decimais é necessário descobrir o algarismo a colocar no quociente que multiplicado pelo divisor resulte no valor mais próximo mas inferior aos dígitos do dividendo. Este processo em binário é simplificado pois o produto é do divisor por zero ou por um, produzindo respectivamente zero ou o divisor. Como exemplo, considere a divisão de 29 (11101) por 3 (11).




```
/*
unsigned int div(unsigned int dividendo, unsigned int divisor, int * resto){
  unsigned int resto = 0;
  unsigned int quociente = 0;
  for (i= 16 ; i != 0 ; i--) {
    rcl(&resto,rcl(&dividendo,false));
    if (resto >= divisor) {
      resto= resto - divisor;
      rcl(&quotient,true);
    }
    else rcl(&quotient,false);
  }
}

boolean rcl(int * op, boolean cy) { /* rotat carry left */
  boolean aux=(op&0x8000)!=0;
  *op=(*op <<1)+(cy)?1:0
  return aux;
}
*/

main:   ldi     r0,#29
        ldi     r1,#3
        ldi     r2,#rest_ref
        jmpl    div
        jmp     $


// r0=dividendo r1=divisor r2=&resto
div:    st      r3,r3t
        st      r4,r4t
        st      r5,r5t  ;preserva registos utilizados

        ldi     r3,#0           ; resto=0
        ldi     r4,#0           ; quociente=0
        ldi     r5,#16          ; i=16
div_3:  shl     r0,r0,#1,0      ; dividendo << 1
        adc     r3,r3,r3        ; (resto << 1) + cy
        subr    r6,r3,r1
        jc      div_1           ; if (divisor > resto)
        sub     r3,r3,r1        ; resto = resto - divisor
        shl     r4,r4,#1,1      ; (quociente << 1) + 1
        jmp     div_2
div_1:  shl     r4,r4,#1,0      ; (quociente << 1)
div_2:  dec     r5 ; i--
        jnz     div_3           ; if (i > 0)
        st      r3,[r2,#0]      ; rest_ref=resto
        mov     r0,r4           ; retorna quociente
        ld      r3,r3t
        ld      r4,r4t
        ld      r5,r5t
        ret
```