11. PROCESSADOR DIDÁCTICO SIMPLES (PDS8_V1)

A introdução de uma arquitectura didáctica simples tem como único objectivo a fácil compreensão das várias componentes de um processador enquanto sistema programável.
Embora abordemos as várias alternativas, optaremos sempre por uma arquitectura que esteja mais de acordo com as actuais arquitecturas.

11.1 Definição da arquitectura

Como já foi referido anteriormente, existem essencialmente dois tipos de arquitectura: CISC (computador com um conjunto de instruções complexo) e RISC (computador com um conjunto de instruções reduzido), também denominada arquitectura LOAD/STORE. Esta última denominação será talvez mais adequada, por existirem hoje CPUs com arquitectura RISC com um conjunto de instruções muito vasto. As arquitecturas CISC desde os anos 70 até aos anos 90 dominaram o mercado de computadores, pois apresentavam um melhor desempenho que as arquitecturas RISC. Ao que se deve então a inversão desta tendência? A razão, é que as arquitecturas CISC, têm um conjunto de instruções muito irregular sob vários pontos de vista, o formato, a dimensão, o acesso a memória, etc. As arquitecturas RISC são, muito regulares no formato, na dimensão e no acesso à memória. Contrariamente ao CISC, nos processadores RISC o acesso à memória de dados só é realizado pelas instruções LOAD e STORE. São essencialmente estes factores que têm levado a baptizar as novas arquitecturas como sendo ou não arquitecturas RISC. Com o aumento da capacidade de integração, esta característica veio trazer-lhe uma grande vantagem pois, por ser regular permitiu construir arquitecturas *pipeline*, ou seja, arquitecturas que permitem executar várias instruções em cadeia, levando a que o CPU possa em cada *clock* dar início a uma instrução e finalizar outra, podendo quase dizer-se que as instruções são executadas num único ciclo de relógio. Este tipo de implementação *pipeline* só será abordada na parte final deste documento.
Ao longo deste documento optaremos por uma arquitectura dita LOAD/STORE que, como já foi referido anteriormente têm as seguintes características:

- As instruções realizam operações elementares;
- Acesso à memória de dados é realizado exclusivamente pelas instruções LOAD e STORE;
- Número reduzido de modos de endereçamento;
- Instruções têm tamanho fixo, constituídas por uma única palavra de memória.

A maioria dos processadores disponíveis no mercado, utilizam o mesmo espaço de memória para código e dados, e são denominadas arquitecturas Von Neumann. Para simplicidade de compreensão, usaremos numa primeira fase uma arquitectura denominada Harvard, que dispõe de uma memória para armazenar o código (memória de código), e outra para armazenar os dados (memória de dados), ambas compostas por 256 elementos.
Para que a arquitectura possa evoluir no sentido de existir uma única memória para dados e código, a dimensão da instrução está limitada à largura do bus de dados da memória de dados.
No que diz respeito à sequência de acções, a arquitectura terá duas fases:

(1) fase de preparação (*fetch*) que corresponde a estabelecer um novo valor para o registo PC e assim se obter a partir da memória de código a nova instrução a ser executada;
(2) fase de execução (*execute*) onde são registados os valores produzidos pela instrução corrente.

11.2 Conjunto das instruções (ISA)

Uma vez que o PDS8_V1 é uma arquitectura LOAD/STORE, as instruções que constituem o ISA, têm dimensão fixa. Nos bits que constituem a instrução estão contidos o código da instrução e os parâmetros, quando necessários.

Este formato de instrução implica algumas restrições à arquitectura do ISA, como sejam: necessidade de acumulação interna ao CPU, de operandos e de resultados e limitação na dimensão dos operandos.

Embora um CPU possa ter vários registos internos para assegurar o seu funcionamento, só alguns destes registos serão vistos pelo programador e denominam-se por registos aplicacionais. O PDS8_V1 tem três registos aplicacionais, RB, RA e PSW. Os registos RB e RA servem de operandos e resultado da ALU e o PSW para conter as *flags*. Nas operações lógicas ou aritméticas os operandos serão sempre RA e RB para as operações binárias e RB para as unárias. Quanto ao destino do resultado, será o registo RA ou RB, indicado como parâmetro da instrução. O PDS8_V1 é um CPU de oito bits (byte), ou seja, os elementos guardados em memória, o bus de dados e os registos internos são de oito bits.

Na Tabela 11-1 é apresentado o conjunto de todas as instruções a que o PDS8_V1 obedece. Nela se podem observar as mnemónicas, os parâmetros, as descrições das operações realizadas e um exemplo de construção para cada uma delas.

O conjunto das instruções está dividido em três grupos:

- Transferência, composto pelas instruções que transferem dados entre o CPU e a memória;
- Processamento, constituído por todas as instruções que utilizam explicitamente a ALU;
- Controlo de Fluxo, composto pelas instruções que modificam de forma condicional ou incondicional o valor do registo PC;

É com este conjunto de instruções vulgarmente denominado *assembler* que irão ser escritos os programas.

Mnemónica	Parâmetro	Descrição	Exemplo
LD	R,direct4	R=dataMem[direct_4]	`ld ra,var1`
LDI	R,#const4	R=const_4	`ldi rb,#0xf`
LD	R,[RB]	R = dataMem [RB]	`ld ra,[rb]`
LD	R,[RA+RB]	R= dataMem [RA+RB]	`ld ra,[ra+rb]`
ST	R,direct	dataMem [direct_4]=R	`st rb,var1`
ST	R,[RB]	dataMem [RB]=R	`st rb,[rb]`
ADD	R	R=RA+RB	`add ra`
SUB	R	R=RA-RB	`sub`
INC	R	R=RB+1	`inc rb`
DEC	R	R=RB-1	`dec`
ANL	R	R=RA&RB	`anl`
ORL	R	R=RA|RB	`orl ra`
XRL	R	R=RA^RB	`xrl`
SWP	R	$R=RA_{0-3} \leftrightarrow RA_{4-7}$	`swp rb`
JZ/JE	offset5	if (Z) PC+=offset_5	`jz label`
JF	offset5	if (F) PC+=offset_5	`jc label`
JMP	offset5	PC+=offset_5	`jmp lalel`
JMP	RB	PC=RB	`jmp rb`

Tabela 11-1 – Conjunto de instruções

R registo RB ou RA.
direct4 endereço da memória de dados especificado a quatro bits.
const4 constante constituída por quatro bits.
offset5 inteiro de 5 bits com sinal [-16..+15].

Nota: Nas instruções Aritméticas e Lógicas quando não se especifica o registo destino, só são afectadas as *flags*.

Quanto ao registo PSW, constituído por dois bits, armazena as *flags* **F** e **Z** com a seguinte informação:

- **F** reflecte uma de três situações:
  - Cy – arrasto da soma
  - Bw – défice da subtracção
  - *Odd* – resultado da operação lógica contém um número ímpar de 1s.
- **Z** indica que resultado da operação foi igual a zero.

11.2.1 Instruções de Transferência

Estas instruções permitem transferir valores entre os registos RA ou RB e a memória de dados. Os valores em memória podem ser acedidos de forma directa, indirecta e indexada. A instrução LD pode ser ainda de modo imediato, ou seja, carregar um registo com uma constante contida na instrução. O carácter # permite distinguir a instrução que envolve um acesso directo da instrução que envolve um valor imediato.
Como se pode constatar no conjunto destas instruções, os parâmetros estão limitados em dimensão, ou seja, a constante está limitada a valores entre 0 e 15 e não acede de forma directa a todos os conteúdos da memória de dados. Estas são sem dúvida, as mais graves limitações deste tipo de arquitecturas, que como veremos adiante pode ser ultrapassada recorrendo a instruções especificamente disponíveis para o efeito.
As instruções de transferência não afectam o registo PSW.

LD (***L**oa**D***) carrega no registo R uma constante ou um conteúdo de memória.
ST (***ST**ore*) guarda o conteúdo do registo R numa posição de memória.

`ld r,direct4` ; (***l**oad **d**irect*) carrega no registo **r** o valor lido da memória de dados cujo endereço é estabelecido pelo parâmetro **`direc4`** (4 bits), estendido com 4 zeros à esquerda.

`ldi r,#const4` ; (***l**oad **I**mmediate*) carrega o registo **r** com a constante **`const4`** (4 bits) contida no código da instrução.

`ld r,[rb]` ; (***l**oad **I**ndirect*) carrega no registo **r** o conteúdo de memória de dados cujo endereço é dado indirectamente pelo valor do registo **rb**.

`ld r,[ra+rb]` ; (***l**oad **I**ndexed*) carrega no registo **r** o conteúdo da memória de dados cujo endereço é dado pelo valor de **ra+rb**.

st r,direct4 ; (***st****ore Direct*) escreve o valor contido no registo **r** na memória de dados cujo endereço é estabelecido pelo parâmetro **direc4** (4 bits), estendido com 4 zeros à esquerda.

st r,[rb] ; (***st****ore Indirect*) escreve o valor contido no registo **r** na memória de dados cujo endereço é dado indirectamente pelo valor do registo **rb**.

Exemplos de instruções C++/java traduzidas para este assembler

```
x=3;
    ldi  ra,#3   ;ra=3
    st   ra,x    ;x=ra
y=x;
    ld   ra,x    ;ra=x
    st   ra,y    ;y=ra
y=x[i];
    ldi  ra,#x   ;ra= endereço base do array x
    ld   rb,i    ;rb=i
    ld   ra,[ra+rb]     ;ra=dataMem[ra+rb]
    st   ra,y    ;y=x[i]
```

11.2.2 Instruções de Processamento

Estas instruções determinam as operações sobre a ALU. As operações aritméticas ou lógicas têm sempre como operandos e resultado registos internos ao CPU. No caso dos operadores binários os operandos são os registos RB e RA, para os operadores unários o operando é o registo RA ou RB. O registo onde é guardado o resultado é determinado pelo parâmetro R (RA, RA) contido na instrução. A ALU, além do resultado propriamente dito tem mais duas saídas, uma que informa se o resultado da operação é igual a zero, e outra com tripla funcionalidade, que informa se existiu arrasto numa operação aritmética ou se uma operação lógica produziu um resultado contendo um número ímpar de uns. Esta informação denominada por *flags* é guardada no registo PSW (*Program Status Word*) sempre que é realizada uma instrução aritmética ou lógica. A informação guardada no registo PSW será utilizada pelas instruções de controlo de fluxo quando for necessário decidir, por exemplo, qual o conjunto de instruções a realizar caso a última operação aritmética tenha produzido arrasto.

Para podermos efectuar operações na ALU, cujo objectivo seja exclusivamente obter informação acerca da relação entre os operandos RA e RB, e se por razões algorítmicas for conveniente preservar o valor dos dois registos, é introduzida a seguinte funcionalidade: se escrevermos a mnemónica da instrução de processamento omitindo o registo destino, o resultado é desprezado e somente o registo PSW será afectado.

Como se pode observar no ISA proposto, não é natural que as operações INC e DEC tenham como fonte exclusiva o registo RB e SWP o registo RA. Esta restrição está relacionada, como foi referido anteriormente, com a complexidade da arquitectura, ou seja, para que a fonte pudesse ser RA ou RB implicava mais um multiplexer na entrada da ALU. Este custo foi considerado excessivo para beneficiar unicamente estas três instruções.

Aritméticas

`add r` ; (***add****ition*) adiciona os valores de **RA** e **RB**, e regista o resultado no registo **r**.

`sub r` ; (***sub****traction*) subtrai ao valor de **RA** o valor de **RB** e regista o resultado em **r**.

`inc r` ; (***inc****rement*) incrementa de um o valor de **RB** e regista o resultado no registo **r**.

`dec r` ; (***dec****rease*) Decrementa de um o valor de **RB** e regista o resultado em **r**

Lógicas

`anl r` ; (***and l****ogic*) realiza o AND lógico, bit a bit, do registo **RA** com **RB**, e regista o resultado no registo **r**.

`orl r` ; (***or l****ogic*) realiza o OR lógico, bit a bit, do registo **RA** com **RB**, e regista o resultado em **r**.

`xrl r` ; (***xor l****ogic*) realiza o XOR lógico, bit a bit, do registo **RA** com **RB**, e regista o resultado em **r**.

`swp r` ; (***swap***) troca os *nibbles* (o de menor peso com o de maior peso) do registo **RA** e regista o resultado em **r**. O principal objectivo desta instrução é permitir carregar um registo com um número de bits superior a 4.

Exemplos:

```
x=y-z;
   ld  ra,y     ;ra fica com y
   ld  rb,z
   sub ra
   st  ra,x

x++;
   ld  ra,x
   inc ra
   st  ra,x

x=-5;
   ldi rb,#low(-5)    ;rb=00001011b
   ldi ra,#high(-5)   ;ra=00001111b
   swp ra             ;ra=11110000b
   orl ra             ;ra=11111011b
   st  ra,x           ;x=11111011b = -5 em código de complementos a 8 bits

x[i]=10;
   ldi rb,#x   ; rb= endereço base do array x
   ld  ra,i    ;ra=i
   add rb      ;rb= endereço de x[i]
   ldi ra,#10
   st  ra,[rb] ;x[i]=10
```

11.2.3 Instruções de controlo de fluxo

Estas instruções, denominadas de salto (*Jump* ou *branch*), permitem alterar a normal sequência de fluxo do programa (entenda-se por normal sequência, a execução da instrução que está no endereço de memória a seguir à que acabou de ser executada), ou seja, permitem alterar o valor do registo PC. Uma vez que é este registo que determina qual o endereço da memória de código onde reside a instrução a ser executada, alterar o seu valor é saltar para essa instrução. As instruções de *jump* podem ser condicionais ou incondicionais. As condicionais implicam o teste de uma *flag* do PSW e caso esta seja verdadeira altera o valor do PC com o parâmetro incluído na instrução, caso contrário deixa que o PC siga a normal sequência. O endereço destino do *jump* condicional, é calculado relativamente ao valor corrente do PC.

`jz offset5` ; (*jump if* **z**) se a *flag* **Z** for verdadeira, adiciona ao valor do **PC** o parâmetro **offset5** contido na instrução. O valor de **offset5** é estendido para 8 bits mantendo o sinal. Também poderá usar a mnemónica **je** (*equal*).

`jf offset5` ; (*jump if* **f**) se a *flag* **F** for verdadeira, adiciona ao valor do **PC**, o parâmetro **offset5** contido na instrução. Para tornar mais explicita a condição de salto poder-se-á usar a mnemónica **jc** (*carry*), **jbl** (*below*), **jpo** (*parity odd*).

`jmp offset5` ; (*jump unconditional*) adiciona ao valor do **PC**, o parâmetro **offset5.**

`jmp rb` ; (*jump register unconditional*) estabelece como valor do registo **PC**, o conteúdo do registo **RB**.

Exemplos:

```
if (x>y)
   z=-1;
else
  z=k+3;
      ld    ra,y
      ld    rb,x
      sub
      jbl   L1   ;if borrow x>y
      ld    ra,k
      ldi   rb,#3
      add   rb   ;rb=k+3
      jmp   L2
L1:   ldi   rb,#0
      dec   rb        ;rb=-1
L2:   st    rb,z
```

```
x=(y==0)?2:4;
      ld    ra,y
      ldi   rb,#0
      orl
      jz    L1
      ldi   ra,#4
      jmp   L2
L1:   ldi   ra,#2
L2:   st    ra,x
```

Determinar a dimensão de uma *string* C++

```
char * ptr;
int c=0;
while (*(ptr++)!='\0') c++;
```

```
      ldi    rb,#0
      st     rb,c       ;c=0
while:
      ld     rb,ptr
      ld     ra,[rb]    ;ra=*ptr
      inc    rb
      st     rb,ptr     ;ptr++
      ldi    rb,#0
      orl               ; ra | 0
      jz     ewhile     ;if(*ptr==0)
      ld     rb,c
      inc    rb
      st     rb,c       ;c++
      jmp    while
ewhile:
```

Procurar o maior valor contido no **int x[]** e colocá-lo como conteúdo da variável **maior**

```
byte i, maior , x[8];
maior=x[0];
for (i=1; i < x.length() ; i++)
  if (x[i] > maior) maior = x[i];
```

```
      ld   ra,x
      st   ra,maior
      ldi  ra,#1
for:  st   ra,i     ;i=1 e i++
      ldi  rb,#length
      sub
      jf   for_1    ;if i < x.length
      jmp  efor     ;if(i >= x.length)
for_1:
      ldi  rb,#x
      ld   rb,[ra+rb]  ;ra=x[i]
      ld   ra,maior
      sub
      jf   then   ;if(x[i]> maior)
for_2:
      ld   rb,i
      inc  ra         ;i+1
      jmp  for
then:
      st   rb,maior   ; (x[i]>  maior) maior=x[i]
      jmp  for_2
efor:
```

Realizar a multiplicação dos dois operandos de oito bits **M** e **m**, utilizando o algoritmo das somas sucessivas. O resultado é expresso em 16 bits.

```
byte M, m;
int P;

P= 0;
if (M!=0)
  for (; m!=0 ; --m)
    P=P+M;

     ldi  ra,#0
     st   ra,P
     st   ra,P+1
     ld   ra,M
     ldi  rb,#0
     orl  ra
     jz   efor     ; if(M==0)
for: ld   ra,m
     ldi  rb,#0
     orl  rb
     jz   efor     ; if(m==0)
     ld   ra,P
     ld   rb,M
     add  ra
     st   ra,P
     jc   for_1    ; if(P+M>255)
for_2:
     ld   rb,m
     dec  rb         ;--m
     st   rb,m
     jmp  for
for_1:
     ld   rb,P+1
     inc  rb
     st   rb,P+1
     jmp  for_2
efor:jmp  $
```

11.3 Implementação do PDS8_V1

Estabelecido o conjunto das instruções e a acção realizada por cada uma delas estamos em condições de desenhar uma estrutura *hardware* que cumpra os objectivos estabelecidos. A estrutura será constituída por um módulo de controlo e um módulo funcional.

Módulo Funcional

Na Figura 11-3 está representada a estrutura base do módulo funcional. O módulo funcional é formado pela memória de dados e pela unidade de processamento. A unidade de processamento contém, para além dos registos RA, RB e PSW, a ALU, os caminhos para a transferência de dados não só entre os registos e ALU, como entre os registos e a memória.

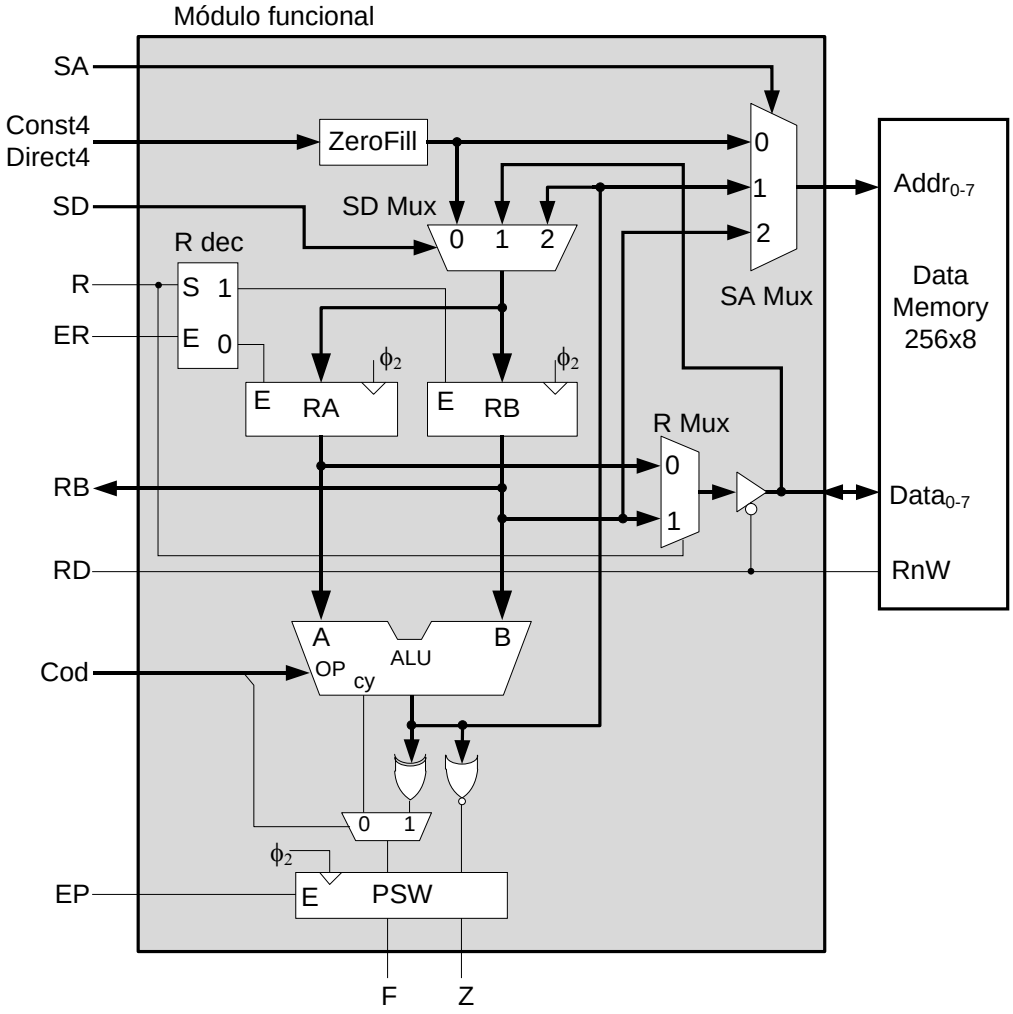

Figura 11-1 – Módulo Funcional

Na unidade de processamento, o multiplexer de dados (SD Mux) determina qual a origem da informação que vai ser escrita no registo RA ou RB. Como se pode ver na Figura 11-1, esta informação pode ter as seguintes origens: em bits da instrução no caso da instrução LDI, na memória de dados aquando da execução das instruções LD, ou na ALU, ao executar instruções de processamento.

O descodificador (R Dec) determina qual dos registos RA ou RB vai ser escrito com a informação proveniente do multiplexer (SD Mux).
O buffer *tri-stat* controla a impedância do bus de dados por parte do CPU. Quando a acção sobre a memória é de leitura, o buffer *tri-stat* é desactivado ficando o bus de dados em alta impedância para permitir que a memória carregue o bus com a informação a ser lida. Quando a acção é de escrita, o buffer *tri-stat* é activado para por presente no bus de dados a informação a ser escrita na memória.
O multiplexer (R Mux) determina qual dos registos RA ou RB são fontes da informação a ser escrita na memória de dados aquando da execução da instrução ST.
O multiplexer (SA Mux) determina qual a origem da informação que estabelece o endereço da memória de dados, de onde ou para onde vai ser transferida informação. A informação que estabelece o endereço pode ter origem na instrução, no caso do endereçamento directo, no registo RB no indirecto ou em RA+RB no endereçamento baseado indexado.
O módulo **ZeroFill** estende para oito bits o parâmetro const4 e direct4 acrescentando quatro zeros à esquerda.
As operações a realizar pela ALU são estabelecidas por três bits que obedecem à codificação apresentada na Tabela 11-2. As *flags* F e Z são registadas no mesmo instante de tempo que é registado o resultado em RA ou RB.

OP	Operação	Descrição
000	A + B	Adição de A com B
001	A – B	Subtracção de B a A
010	B + 1	Incrementa A de 1
011	B – 1	Decrementa A de 1
100	A & B	AND lógico bit a bit de A com B
101	A | B	OR lógico bit a bit de A com B
110	A⊕B	XOR lógico bit a bit de A com B
111	SWAP A	Troca os *nibbles* de A

Tabela 11-2 - Códigos de operação da ALU

Módulo de Controlo

O controlo é formado pela memória de código e pelo módulo de controlo.

No módulo de controlo mostrado na Figura 11-2, podemos identificar o sequenciador formado pelos seguintes elementos: somador, registo PC, *Index Mux*, *Offset Mux* e memória de código. No módulo de controlo, o sequenciador, que a cada transição ascendente evolui de uma unidade e põe disponível uma nova instrução a ser executada pelo módulo funcional, pode ver quebrada a sequência pela execução de uma instrução *jump*. Esta quebra de sequência pode ser determinada pela soma do valor corrente do registo PC com um *offset* através do multiplexer (Offset Mux), ou através do multiplexer (*Index* Mux), em que o registo PC recebe o valor contido no registo RB. O módulo **SigExt** estende o sinal do parâmetro offset5 por este ser considerado um valor inteiro e ter que ser somado com o registo PC que tem um número de bits superior.

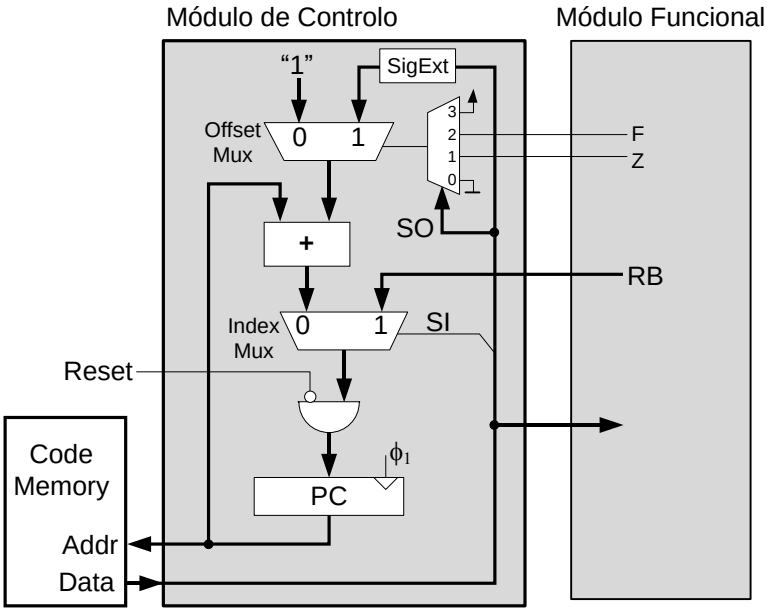

Figura 11-2 – Módulo de Controlo

11.3.1 Estrutura do PDS8_V1

Na Figura 11-3 é apresentada a estrutura completa do PDS8_V1, onde se pode observar a interligação entre o módulo funcional e módulo de controlo.

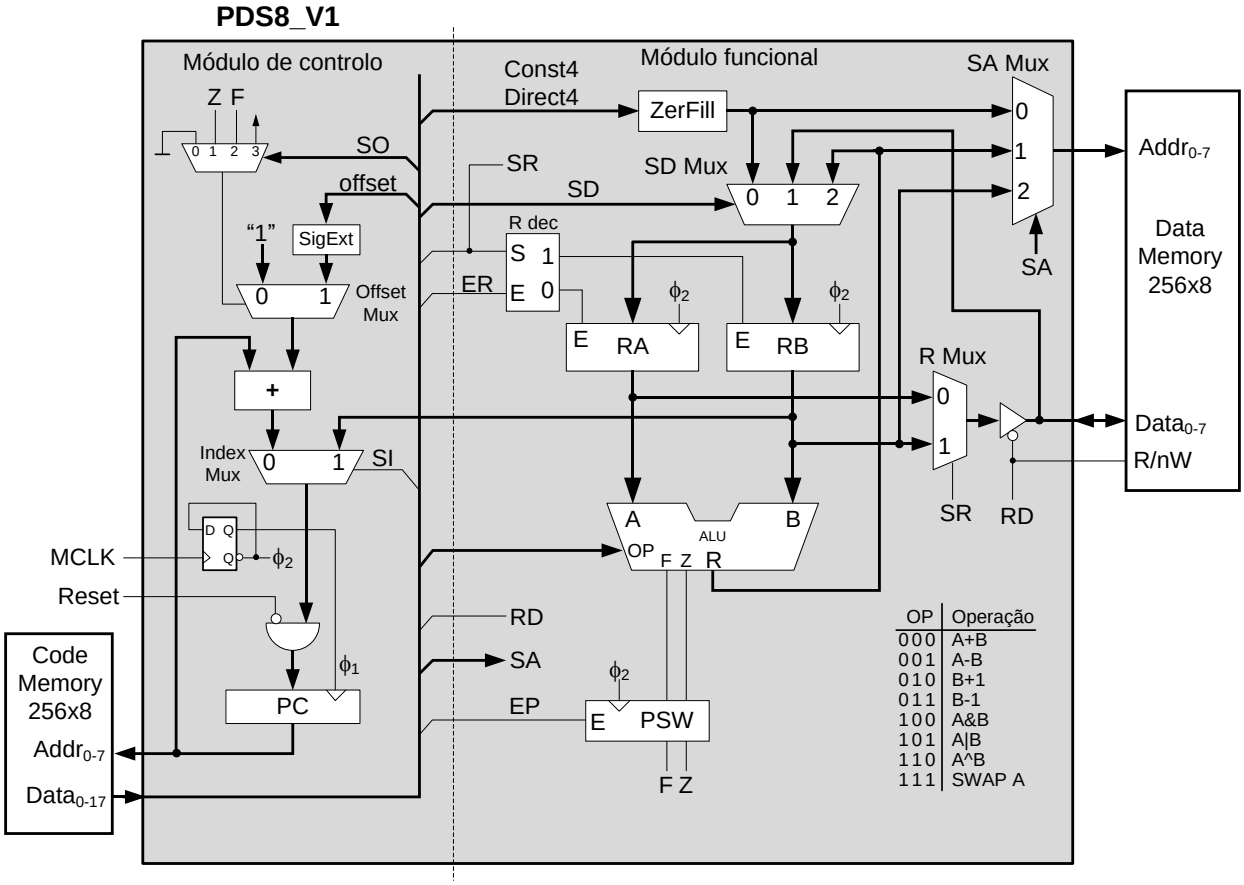

Figura 11-3 – PDS8_V1, Diagrama de blocos com micro programação horizontal

Especificação dos registos que constituem a estrutura:

- RA, RB e PSW são síncronos (*edge trigger*) com controlo de *Enable*. A necessidade destes registos serem *edge trigger*, deve-se ao facto de nas operações Aritméticas e Lógicas um mesmo registo ser simultaneamente operando e resultado.
- RA, RB e PSW são afectados na transição ascendente do sinal de ϕ_2, que corresponde à fase de execução da instrução.
- o registo PC é afectado em todas as transições ascendentes do sinal ϕ_1. Esta é a razão pela qua o registo PC não tem controlo de *Enable.*
- A acção de iniciação do PC é activada pelo sinal **RESET**. Esta acção é síncrona com o *clock* e tem como única consequência colocar o valor do registo PC a zero.

Descrição dos sinais de entrada externos:

- **MCLK** (*Master clock*) Esta entrada determina o ritmo de funcionamento do PDS8_V1. O PDS8_V1 gera internamente dois sinais ϕ_1 e ϕ_2, desfasados de 180°, para garantir que o momento de evolução do Módulo de Controlo não coincide com a acção de escrita no Módulo

Funcional. Estes dois sinais ϕ_1 e ϕ_2 estabelecem duas fases, uma de *fetch* ϕ_1 e outra de *execute* ϕ_2. A existência do *flip*-flop D garante *duty cycle* de 50% para os sinais ϕ_1 e ϕ_2, assegurando desta forma tempos idênticos para a fase *fetch* e para a fase *execute*.

- **Reset** A activação desta entrada leva a que o registo PC tome o valor zero, e consequentemente fique disponível no bus de dados da memória de código, a primeira instrução da aplicação estabelecida pelo programador. Após a libertação da entrada Reset, o sistema dá início à execução do programa.

Descrição dos sinais de saída do módulo de controlo:

- **SI** (*Select index*) permite realizar a instrução **`jmp rb`**, pois quando activo selecciona para a entrada do PC o valor de RB, estabelecendo um novo endereço de *fetch*;
- **SO** (*Select Offset*) determina se o próximo *fetch* se realiza no endereço dado por PC+1, ou em PC mais o parâmetro offset contido nas instruções de *jump* condicional e incondicional;
- **ER** (*Enable Registers*) é utilizado pela instrução **`LD R,(parâmetro)`** e pelas instruções de processamento. No caso da instrução LD, serve para registar em R o parâmetro fonte, estando o parâmetro R (RA ou RB) também contido na instrução. Nas instruções de processamento, para registar em R o resultado da operação realizada pela ALU.
- **EP** (*Enable* PSW) controla a escrita das *flags* no registo PSW aquando das operações de processamento;
- **SD** (*Select Data*) selecciona qual a informação a carregar nos registos RA ou RB. Para a instrução **`LD,#const4`** selecciona a constante que é parâmetro da instrução. Na instrução **`LD,direct4`** selecciona o valor que está a ser lido da memória de dados. Nas instruções de processamento, selecciona o resultado da ALU.
- **SA** (*Select Address*) selecciona qual o parâmetro que estabelece o endereço. No caso de **`LD/ST R,direct4`**, selecciona o parâmetro contido na instrução. Caso o modo de endereçamento seja indirecto, selecciona o registo de indirecção RA. Se o modo é indexado, selecciona o valor calculado pela ALU.
- **SR** (*Select Register*) é utilizado pela instrução **`ST R,(parâmetro)`** para seleccionar qual dos registos RA ou RB vai fornecer o valor a ser escrito na memória de dados.
  Na instrução **`LD R,(parâmetro)`**, selecciona qual dos registos RA ou RB recebe o valor estabelecido pelo multiplex SD Mux. O parâmetro R está contido na instrução.
- **RD** (*Read*) estabelece se a acção sobre a memória de dados é de escrita ou leitura. Caso a instrução não envolva escrita ou leitura da memória de dados, este sinal tem que ser mantido activo, para que não se realize uma escrita na memória.
- **SigExt** (*Signal Extend*) realiza a extensão do sinal da constante inteira **offset5**. Esta operação deve-se a facto de offset5 e o valor de PC serem entendidos como dois inteiros representados em código de complementos para dois e que para serem somados tem que ter o mesmo número de bits.
- **ZerFill** (*Zero Fill*) adiciona quatro zeros nos bits de maior do parâmetro **const4** e **direct4**. Esta operação é necessária, porque o valor a carregar no registo RA ou RB e o endereço da memória de dados, é de oito bits. No caso do **direct4**, a adição de quatro zeros, estabelece para endereço das variáveis de acesso directo as primeiras 16 posições da memória de dados.

11.3.2 Formato das micro-instruções

Para concluirmos o desenho do PDS8_V1, é necessário estabelecer o conteúdo e o formato de cada uma das micro-instruções do módulo de controlo. É conveniente que um mesmo tipo de parâmetro ocupe os mesmos bits em todas as micro-instruções afim de diminuir o número total de bits. Dada a estrutura do Módulo de Controlo e do Módulo Funcional e admitindo que todas as micro-instruções têm o mesmo número de bits e ocupam um único endereço da memória de código, podemos concluir que as micro-instruções são constituídas por 18 bits como mostra a Tabela 11-3.

Inst		D_{17}	D_{16}	D_{15}	D_{14}	D_{13}	D_{12}	D_{11}	D_{10}	D_9	D_8	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
		SI	SO		EP	ER	RD	SA		SD		ALU Cod			Parâmetros				
LD	R,direct4	0	0	0	0	1	1	0	0	0	1	-	-	-	D	D	D	D	R
LDI	R,#const4	0	0	0	0	1	1	-	-	0	0	-	-	-	C	C	C	C	R
LD	R,[RB]	0	0	0	0	1	1	1	0	0	1	-	-	-	-	-	-	-	R
LD	R,[RA+RB]	0	0	0	0	1	1	0	1	0	1	0	0	0	-	-	-	-	R
ST	R,direct	0	0	0	0	0	0	0	0	0	1	-	-	-	D	D	D	D	R
ST	R,[RB]	0	0	0	0	0	0	1	0	0	1	-	-	-	-	-	-	-	R
ADD	R	0	0	0	1	T	1	-	-	1	0	0	0	0	-	-	-	-	R
SUB	R	0	0	0	1	T	1	-	-	1	0	0	0	1	-	-	-	-	R
INC	R	0	0	0	1	T	1	-	-	1	0	0	1	0	-	-	-	-	R
DEC	R	0	0	0	1	T	1	-	-	1	0	0	1	1	-	-	-	-	R
ANL	R	0	0	0	1	T	1	-	-	1	0	1	0	0	-	-	-	-	R
ORL	R	0	0	0	1	T	1	-	-	1	0	1	0	1	-	-	-	-	R
XRL	R	0	0	0	1	T	1	-	-	1	0	1	1	0	-	-	-	-	R
SWP	R	0	0	0	1	T	1	-	-	1	0	1	1	1	-	-	-	-	R
JZ/JE	offset5	0	0	1	0	0	1	-	-	-	-	-	-	-	of	of	of	of	of
JF	offset5	0	1	0	0	0	1	-	-	-	-	-	-	-	of	of	of	of	of
JMP	offset5	0	1	1	0	0	1	-	-	-	-	-	-	-	of	of	of	of	of
JMP	RB	1	-	-	0	0	1	-	-	-	-	-	-	-	-	-	-	-	-

Tabela 11-3 – Formato das micro-instruções

Este modelo de implementação micro programado é denominado por micro programação horizontal, ou seja, cada micro-instrução contém as várias micro-operações. Dado que cada microinstrução é constituída por 18 bits, e considerando que um programa pode ser constituído por um número elevado de instruções, implicaria que a memória de código tivesse uma grande dimensão. Como se pode observar na Tabela 11-3, para cada uma das micro-instruções, os bits de D_5 a D_{17} são constantes. Tratando-se de apenas 18 diferentes instruções, permite realizar uma compressão através da codificação de cada uma das instruções. Este modelo de implementação micro-programado é denominado por micro-programação vertical.

Esta solução implica a adição de uma ROM no módulo de controlo para conter o micro-código. Esta ROM realiza a descompressão por descodificação do código da instrução e gera as respectivas micro-operações. Ao programa expresso através destas sequências de bits é vulgarmente referido como estando em **Código Máquina** sendo cada elemento denominado por instrução em código máquina.

11.3.3 Código das instruções

Codificar as várias instruções é atribuir a cada uma delas um código unívoco que permita ao controlo do CPU distingui-las e assim activar adequadamente as diferentes micro-operações, de forma a garantir a sua execução.

Codificar subentende, normalmente, comprimir e é com essa premissa que iremos codificar as instruções do PDS8_V1. Dado que o PDS8_V1 é um CPU de oito bits (byte), ou seja, os valores guardados em memória, o bus de dados e os registos internos são de oito bits, iremos codificar as instruções tendo em mente esta dimensão no sentido de permitir que numa outra versão do PDS8, a memória de código e a memória de dados, partilhem o mesmo espaço de endereçamento. Assim sendo, o código das instruções mais os parâmetros não podem exceder os oito bits. Com esta restrição, e dada a especificação do ISA e a dimensão de alguns parâmetros, a codificação pode não ser uniforme, nomeadamente, o código não estabelece uma divisão directa entre os vários tipos de instruções, não existe uma relação directa entre os bits do código de instrução e as micro-operações, e por outro lado, o código das várias instruções não tem um número constante de bits. Na Tabela 11-4 é sugerida uma codificação para as várias instruções do PDS8_V1 que cumpre com o ISA proposto e, embora não sendo completamente estruturada, consegue alguma uniformização, por exemplo estabelece uma localização constante para o código da instrução e para os vários campos constituintes da instrução.

Instrução	**Codificação**							
	7	6	5	4	3	2	1	0
LD R,direct4	0	0	1	A	A	A	A	R
LDI R,#const4	0	1	0	C	C	C	C	R
LD R,[RB]	0	0	0	0	1	-	-	R
LD R,[RA+RB]	0	0	0	0	0	0	0	R
ST R,direct	0	1	1	A	A	A	A	R
ST R,[RB]	0	0	0	1	0	-	-	R
ADD R	1	1	1	T	0	0	0	R
SUB R	1	1	1	T	0	0	1	R
INC R	1	1	1	T	0	1	0	R
DEC R	1	1	1	T	0	1	1	R
ANL R	1	1	1	T	1	0	0	R
ORL R	1	1	1	T	1	0	1	R
XRL R	1	1	1	T	1	1	0	R
SWP R	1	1	1	T	1	1	1	R
JZ offset5	1	0	0	Of	Of	Of	Of	Of
JF offset5	1	0	1	Of	Of	Of	Of	Of
JMP offset5	1	1	0	Of	Of	Of	Of	Of
JMP RB	0	0	0	1	1	-	-	-

Tabela 11-4 - Codificação das instruções

Nota: O bit T (**T**este) quando a "1" indica que o registo destino R é afectado.

Nas instruções lógicas e aritméticas, os três bits de índice 1, 2 e 3 constituem o código das operações da ALU e o bit de índice zero contem o parâmetro Rx. Na instrução LD modo indexado, em que o endereço é estabelecido por RA+RB, a estrutura utiliza a ALU para realizar a soma e assim sendo os bits de índice 1,2 e 3 desta instrução devem ter o valor zero, de forma a garantir que ALU executa a adição de A com B.

11.3.4 Modulo Descodificador (*Instruction Decoder*)

Dado que as instruções chegam ao CPU codificadas, o PDS8_V1 passa a apresentar a estrutura da mostrada na Figura 11-4.

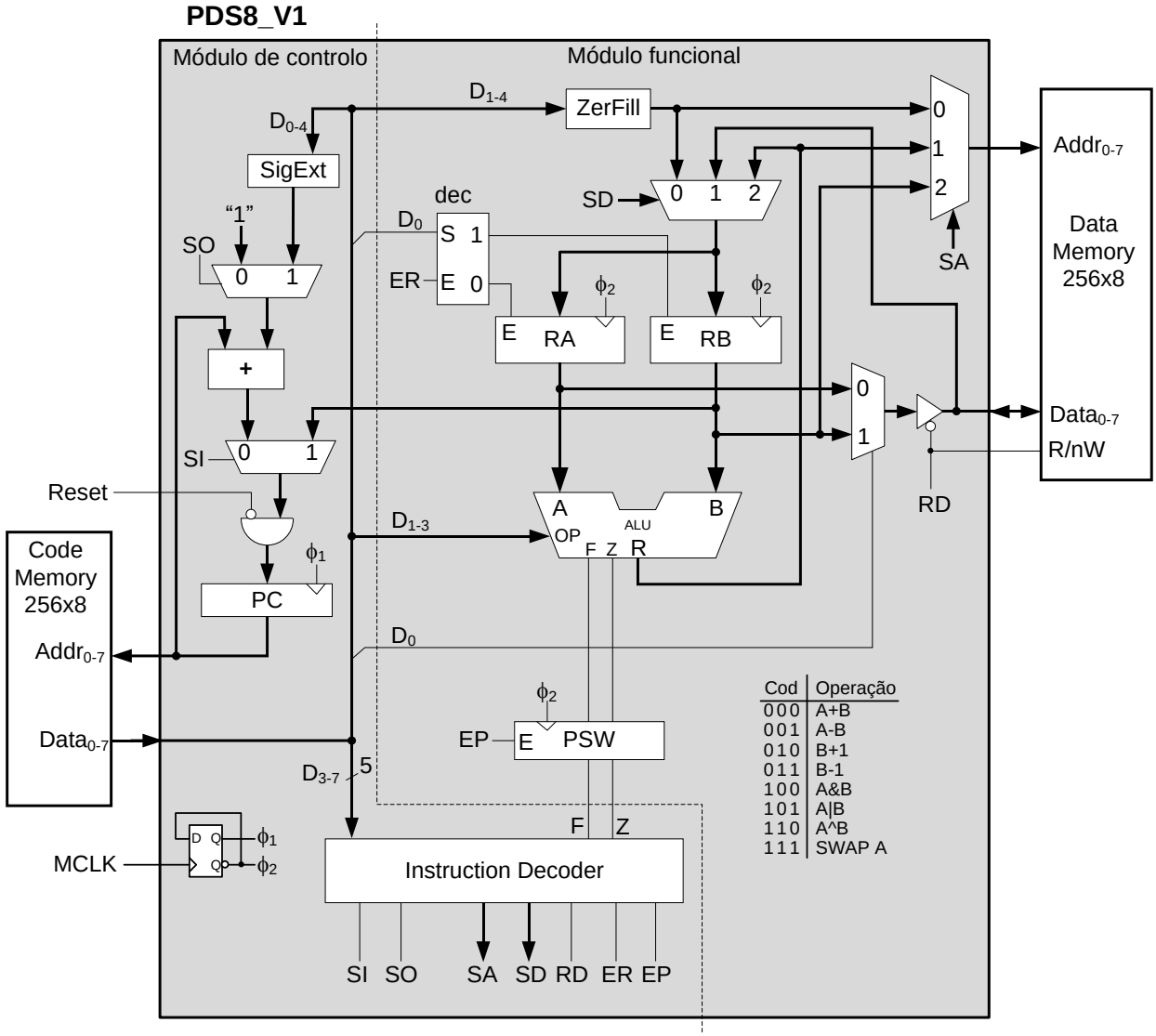

Figura 11-4

É necessário conceber um módulo descodificador, que tendo como entrada o código da instrução mais as *flags* F e Z, produza as micro-operações necessários à execução da instrução.

A implementação do módulo descodificador de instrução, que é responsável por gerar as micro-operações associados à fase de preparação e execução das diferentes instruções, é baseada numa ROM de 128x9 cuja programação é apresentada na Tabela 11-5. Constituem endereço da ROM, os 5 bits mais significativos da instrução, conjuntamente com as *flags* Z e F.

	I_7	I_6	I_5	I_4	I_3	Z	F	SI	SO	SA	SD	R	ER	EP
LD R,[RA+RB]	0	0	0	0	0	-	-	0	0	01	01	1	1	0
LD R,[RB]	0	0	0	0	1	-	-	0	0	10	01	1	1	0
ST R,[RB]	0	0	0	1	0	-	-	0	0	10	01	0	0	0
JMP RB	0	0	0	1	1	-	-	1	-	-	-	1	0	0
LD R,direct4	0	0	1	-	-	-	-	0	0	00	01	1	1	0
LDI R,#const4	0	1	0	-	-	-	-	0	0	-	00	1	1	0
ST R,direct4	0	1	1	-	-	-	-	0	0	00	-	0	0	0
JZ offset5	1	0	0	-	-	0	-	0	0	-	-	1	0	0
JZ offset5	1	0	0	-	-	1	-	0	1	-	-	1	0	0
JF offset5	1	0	1	-	-	-	0	0	0	-	-	1	0	0
JF offset5	1	0	1	-	-	-	1	0	1	-	-	1	0	0
JMP offset5	1	1	0	-	-	-	-	0	1	-	-	1	0	0
ALU	1	1	1	T	-	-	-	0	0	-	10	1	T	1

Tabela 11-5 – Módulo Descodificador

Como curiosidade, podemos observar que, se o CPU ler o código binário 0000010R, leva a que este execute a instrução LD R,[RB+1], descubra outras.

Exercício:
Escrever em Código Máquina um programa para determinar o maior valor contido no *array* **x**.

```
byte i, maior , x[8]; /* inteiros sem sinal */
maior=x[0];
for (i=1; i < x.length(); i++)
  if (x[i] > maior) maior = x[i];
```

Code Memory					
Label	Inst	Parameters	Addr	Code Machine	
main:	ld	ra,x	0x00	0010 0100	0x24
	st	ra,maior	0x01	0110 0010	0x62
	ldi	ra,#1	0x02	0100 0010	0x42
for:	st	ra,i	0x03	0110 0000	0x60
	ldi	rb,#length	0x04	0101 0001	0x51
	sub		0x05	1110 0010	0xE2
	jf	for_1	0x06	1010 0010	0xA2
	jmp	efor	0x07	1100 1011	0xCB
for_1:					
	ldi	rb,#x	0x08	0100 0101	0x45
	ld	rb,[ra+rb]	0x09	0000 0001	0x01
	ld	ra,maior	0x0A	0010 0010	0x22
	sub		0x0B	1110 0010	0xE2
	jf	then	0x0C	1010 0100	0xA4
for_2:					
	ld	ra,i	0x0D	0010 0000	0x20
	inc	ra	0x0E	1111 0100	0xF4
	jmp	for	0x0F	1101 0100	0xD4
then:					
	st	rb,maior	0x10	0110 0011	0x63
	jmp	for_2	0x11	1101 1100	0xDC
efor:	jmp	$	0x12	1100 0000	0xC0

Data Memory		
Label	Addr	Val
i:	0x00	
maior:	0x01	
x:	0x02	x[0]
	0x03	
	0x04	
	0x05	
	0x06	
	0x07	
	0x08	
	0x09	x[7]