11. PROCESSADOR DIDÁCTICO SIMPLES (P8_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.
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;
- Grande número de registos internos;
- Acesso à memória de dados é realizado exclusivamente pelas instruções LOAD e STORE;
- 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 P8_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 P8_V1 tem quatro registos aplicacionais de oito bits, R0 a R3 organizados em vector. Um registo de denominado PSR para conter as flags resultantes das operações aritméticas. Os registos de R0 a R3 servem de operandos e resultado da ALU. Nas operações aritméticas os operandos A e B podem ser quaisquer dos registo R0 a R3, o resultado é sempre depositado num dos registos. O P8_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 P8_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 de memória, composto pelas instruções que transferem dados entre o CPU e a memória;
- Transferência de registo imediato, composto pela instrução que inicia um registo com um valor constante.
- Processamento, constituído pelas instruções que utilizam explicitamente a ALU;
- Controlo de Fluxo, composto pelas instruções que modificam de forma condicional ou incondicional o valor do registo PC provocando, quebra de sequencialidade;

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

Mnemónica	Parâmetro	Descrição	Exemplo
ldr	rd,direct3	rd = data_mem[direct3]	`ldr r0,var1`
ldr	rd,[rb]	rd = data_mem [rb]	`ldr r1,[r3]`
str	rs,direct3	data_mem [direct3] = rs	`str r2,var1`
str	rs,[rb]	data_mem [rb] = rs	`str r0,[r1]`
mov	rd,imm3	rd = imm3	`mov r1,6`
add	rn,rm	rn = rn + rm	`add r0,r2`
sub	rn,rm	rn = rn - rm	`sub r1,r3`
bzs/beq	offset5	if (z==1) pc+=offset5	`bzs label`
bcs/blo	offset5	if (cy==1) pc+=offset5	`bcs label`
b	offset5	pc+=offset5	`b label`

Tabela 11-1 – Conjunto de instruções

rd registo destino

rs registo fonte
rb registo base.
direct3 endereço da memória de dados especificado a três bits.
imm3 constante imediata constituída por três bits.
offset5 inteiro de 5 bits com sinal [-16 a +15].

Nota: Apenas as instruções aritméticas afectam as *flags*.

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

- **CY** reflecte uma de duas situações:
  - Cy – arrasto da soma
  - Bw – défice da subtracção
- **Z** indica que resultado da operação foi igual a zero.

11.2.1 Instruções de Transferência

Estas instruções permitem transferir valores entre os registos R0 a R3 e a memória de dados. Os valores em memória podem ser acedidos de forma directa ou indirecta. A transferência pode ser ainda de modo imediato, ou seja, carregar um registo com uma constante contida na instrução.
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 7 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 é ultrapassada recorrendo a instruções especificamente disponíveis para o efeito. As instruções de transferência não afectam o registo PSR.

ldr (***l**oa**d** **r**egister*) transfere para um registo (r0 a r3) um conteúdo de memória.

str (***st**ore* ***r**egister*) transfere o conteúdo de um registo (r0 a r3) para uma posição de memória.

mov move para um registo R0 a R3 uma constante.

`ldr rd,direct3` ; (***l**oad **d**irect*) carrega no registo **rd** o valor lido da memória de dados cujo endereço é estabelecido pelo parâmetro **`direc3`** (3 bits), estendido com 5 zeros à esquerda.

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

`str rs,direct3` ; (***st**ore Direct*) escreve o valor contido no registo **rs** na memória de dados cujo endereço é estabelecido pelo parâmetro **`direc3`** (3 bits), estendido com 5 zeros à esquerda.

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

`mov rd,imm3` ; (***mov***) carrega o registo **rd** com a constante **`imm3`** (3 bits) estendida a zeros.

Exemplos:

```
x=3;
       mov    r2,3   ;r2=3
       str    r2,x   ;x=r2
y=x;
       ldr    r0,x   ;r0=x
       str    r0,y   ;y=r0
```

11.2.2 Instruções de Processamento

Estas instruções determinam as operações sobre a ALU. As operações aritméticas têm sempre como operandos e resultado os registos R0 a R3. O registo onde é guardado o resultado é sempre o registo especificado à esquerda. 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 que informa se existiu arrasto na soma ou falta na subtracção. Esta informação denominada por *flag* é guardada no registo PSR (*Program Status Register*) sempre que é realizada uma instrução aritmética. A informação guardada

no registo PSR 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.

Aritméticas

`add rn,rm` ; (***add****ition*) adiciona os valores de **rn** e **rm**, e regista o resultado no registo **rn**.

`sub rn,rm` ; (***sub****traction*) subtrai ao valor de **rn** o valor de **rm** e regista o resultado em **rn**.

Exemplos:

```
x=y-z;
      ldr     r0,y
      ldr     r1,z
      sub     r0,r1
      str     r0,x
x++;
      ldr     r0,x
      mov     r1,1
      add     r0,r1
      str     r0,x
x=-4;
      mov     r0,0
      mov     r1,4   ;r0=00000100b
      sub     r0,r1  ;r0=11111100b
      str     r0,x   ;x =11111100b=-4 código de complementos a 8 bits
y=x[i];
      mov     r0,x     ;r0= endereço base do array x
      ldr     r1,i     ;r1=i
      add     r0,r1
      ldr     r2,[r0]  ;r2=dataMem[r0] r2=x[i]
      str     r2,y     ;y=x[i]

x[i]=10; considerando que x não se encontra no espaço directo
xaddr:
       .byte  x      ;variável iniciada com endereço base do array x
Const10:
       .byte  10
       ldr    r0,x_addr
       ldr    r1,i
       add    r0,r1    ; r0= endereço de x[i]
       ldr    r1,const10
       str    r1,[r0]  ;x[i]=10
x:
       .space X_DIM
```

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 próxima instrução a ser executada, alterar o seu valor é saltar para essa instrução. As instruções de *branch* podem ser condicionais ou incondicionais. As condicionais implicam o teste de uma *flag* do PSR e caso a condição 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 *branch*, é calculado relativamente ao valor corrente do PC.

`bzs offset5` ; (***b**rach if* **z** is **s**et) se a *flag* **z** for igual a "1", 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 **beq** (***b**ranch if **eq**ual*).

`bcs offset5` ; (*branch if* **cy** is **set**) se a *flag* **cy** for igual a "1", 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 **blo** (***b**ranch **lo**w*).

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

Exemplos:

```
x=(y==0)?2:4
       ldr    r0,y
       mov    r1,0
       add    r0,r1
       bzs    L1
       mov    r0,4
       b      L2
L1:    mov    r0,2
L2:    str    r0,x
```

```
if (x>y)
   z=-1;
else
  z=k+3;
```

```
       ldr    r0,y
       ldr    r1,x
       sub    r0,r1
       blo    then   ;if borrow x>y
else:  ldr    r0,k
       mov    r1,3
       add    r0,r1  ;r0=k+3
       b      L2
then:  mov    r0,0
       mov    r1,1
       sub    r0,r1  ;r0=-1
L2:    str    r0,z
```

Exercícios:

EX_1: Contar quantos zeros existem no *array* x com dimensão DIM.

```
char x[DIM];
int total=0;
for (i=0; i < DIM; i++)
  if (x[i]=='\0') total++;
```

```
      .equ        LENGTH,8
x_addr:
      .byte x
total:
      .space 1
i:    .space 1

x:    .space LENGTH
main: ldr   r0,x_addr
      mov   r1,0
      str   r1,total ;total=0
      str   r1,i     ;i=0
for:  mov   r2,LENGTH
      sub   r1,r2    ;i-LENGTH
      blo   for1
      b     .
for1: ldr   r1,i
      add   r1,r0    ;x_addr+i
      ldr   r2,[r1]  ;r2=x[i]
      mov   r3,0
      add   r3,r2
      bzs   for2     ;if(x[i]==0)
for3: mov   r3,1
      ldr   r1,i
      add   r1,r3
      str   r1,i     ;i++
      b     for
for2: mov   r3,1
      ldr   r1,total
      add   r1,r3
      str   r1,total ;total++
      b     for3     ;i++
```

EX_2: Procurar o maior valor contido no **`int x[]`** e colocá-lo como conteúdo da variável **`maior.`**

```
byte i, maior , x[6];
maior=x[0];
for (i=1; i < x.length() ; i++)
  if (x[i] > maior) maior = x[i];
```

```
       .equ        LENGTH,8
x_addr:
       .byte x
maior:
       .space 1
i:     .space 1

x:     .space LENGTH

main:
       ldr    r0,x_addr
       ldr    r3,[r0]  ;r3=x[0]=maior
       mov    r1,1
       str    r1,i   ;i=1
for:   mov    r2,LENGTH
       sub    r1,r2  ;i-LENGTH
       blo    for1   ;if (i < x.length())
       b      for_end  ;if(i >= x.length())
long_branch:
       b      for
for1:  ldr    r1,i
       add    r1,r0  ;r1=x_addr+i
       ldr    r2,[r1]  ;r2=x[i]
       sub    r2,r3    ; x[i]-maior
       blo    else   ;if(x[i] < maior)
       beq    else   ;if (x[i]==maior)
then:  ldr    r3,[r1]  ;     maior=x[i]
else:  ldr    r1,i
       mov    r2,1
       add    r1,r2
       str    r1,i   ;i++
       b      long_branch   ;distancia para a label for superior a 15
for_end:
       str    r3,maior
       b      .
```

EX_3: 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;
```

```
M:     .space 1
m:     .space 1
Pl:    .space 1
Ph:    .space 1

main:
       mov    r0,0
       str    r0,P_l
       str    r0,P_h ;P=0
       ldr    r1,M
       add    r1,r0
       bzs    long_for_end   ; if(M==0)
       ldr    r2,m
       add    r2,r0
for:   bzs    long_for_end   ; if(m==0)
       ldr    r1,P_l
       ldr    r3,M
       add    r1,r3
       str    r1,P_l ;P_l=P_l+M
       bcs    for_1  ; if(P_l+M>255) P_h+1
for_2:
       mov    r1,1
       sub    r2,r1  ;--m
       b      for
long_for_end:
       b      for_end
for_1:
       mov    r1,1
       ldr    r3,P_h
       add    r3,r1
       str    r3,P_h ;P=P+M (16 bits)
       b      for_2
for_end
       b      .
```

11.3 Implementação do P8_V1

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

Módulo Funcional

Na Figura 11-1 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 do *register file* do PSR e da ALU, os caminhos para a transferência de dados não só entre o *register file* e a ALU, como entre o *register file* e a memória.

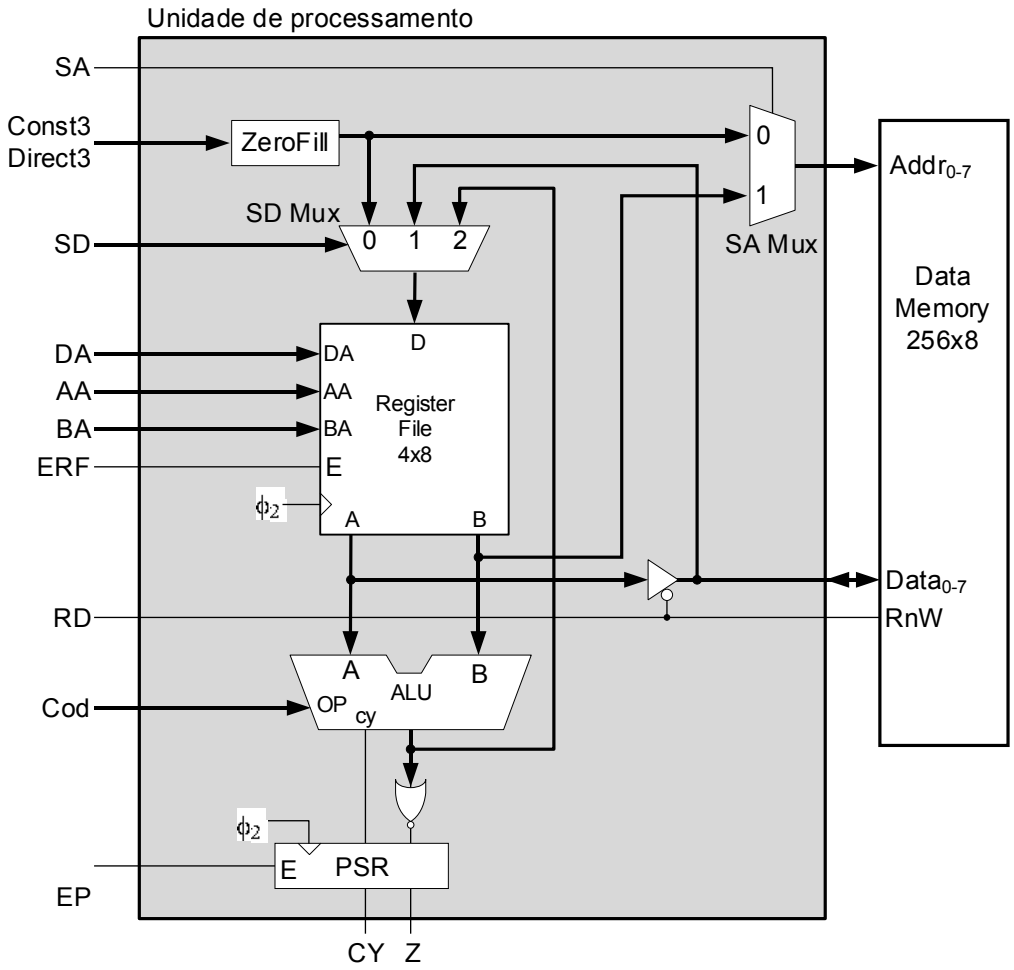
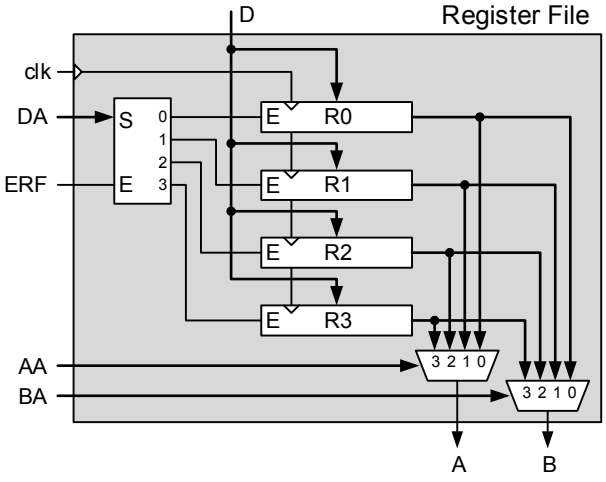


Figura 11-1 – Módulo Funcional

Unidade de processamento

Descrição dos vários elementos que constituem a unidade de processamnto.

Register file Constituído por 4 registos de 8 bits organizados em vector e que permite que sejam seleccionados em simultâneo quaisquer três registos: um registo a ser escrito e dois a serem lidos. Os registos são síncronos (*edge trigger*) com controlo de *Enable*. A necessidade destes elementos serem *edge trigger*, deve-se ao facto de nas operações aritméticas um mesmo registo ser simultaneamente operando e resultado.
Na Figura 11-2 está representada a estrutura base de um *register file*.



[corrigir desenho R0 R1 R2 R3]

Figura 11-2

ALU Unidade que realiza as operações ADD e SUB seleccionadas por 1 bit conforme a codificação apresentada na Tabela 11-2.

OP	Operação	Descrição
0	A + B	Adição de A com B
1	A – B	Subtracção de B a A

Tabela 11-2 - Códigos de operação da ALU

PSR registo para conter as *flag* CY e Z resultantes das operações aritméticas e que servem de base às instruções de controlo de fluxo. As *flags* só são registadas quando é executada uma operação aritmética.

SD mux multiplexer que determina qual a origem da informação que vai ser escrita no *register file*. Como se pode ver na Figura 11-1, esta informação pode ter origens em bits da instrução no caso da instrução `mov rd,imm3`, na memória de dados aquando da execução das instruções `ldr`, ou na ALU, ao executar as instruções aritméticas `add` ou `sub`.

SA mux multiplexer que determina qual a origem da informação que estabelece o endereço da memória de dados, de onde ou para onde vai ser transferida informação. O endereço pode ter origem no registo **rb** no endereçamento indirecto ou em informação contida na instrução, no caso do endereçamento directo.

ZeroFill estende para oito bits o parâmetro imm3 e direct3 acrescentando cinco zeros à esquerda.

Dado que a memória de dados utilizada tem bus de dados bidireccional, 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 (RnW=1). 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.

Módulo de Controlo

O controlo é formado pela memória de código e pela unidade de controlo.

No módulo de controlo mostrado na Figura 11-3, podemos identificar a memória de código e o sequenciador formado pelos elementos: somador, registo PC, *Index Mux*, *Offset Mux* . Na unidade de controlo, o sequenciador, a cada transição ascendente evolui de uma unidade e põe disponível uma nova instrução a ser executada pelo módulo funcional. Esta sequência pode ser quebrada pela execução de uma instrução *branch*. Esta quebra de sequência é determinada pela soma do valor corrente do registo PC com um *offset* através do multiplexer (Offset Mux). Como se pode observar é a instrução que determina qual a flag a ser testada ou se é incondicional. O módulo **SigExt** realiza a extensão do sinal do parâmetro offset5 para que este possa ser somado com o registo PC que é de oito bits em caso de *branch*.

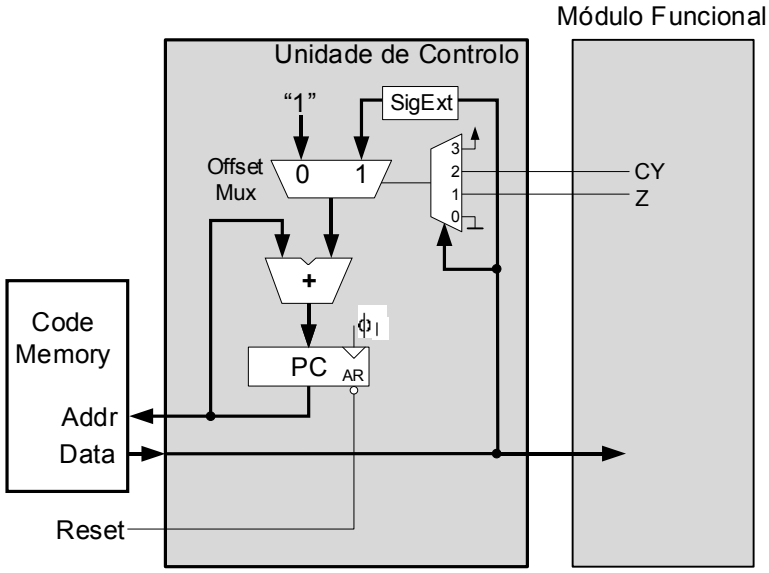


Figura 11-3 – Módulo de Controlo emendar F por CY

11.3.1 Estrutura do P8_V1

Na Figura 11-4 é apresentada a estrutura completa do P8_V1, onde se pode observar a interligação entre o módulo funcional e módulo de controlo.

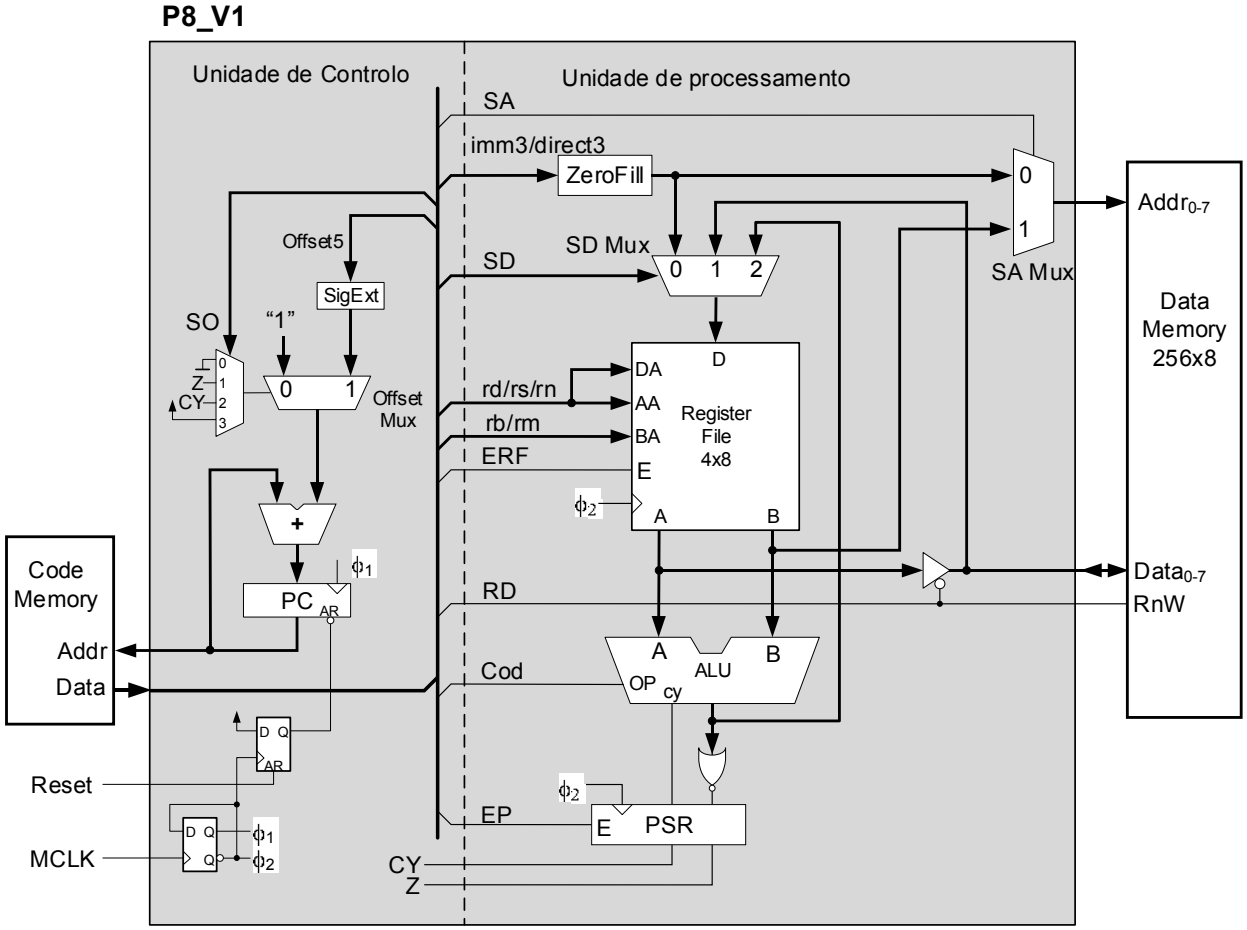


Figura 11-4 – P8_V1, Diagrama de blocos com micro programação horizontal

Descrição dos sinais de entrada externos:

MCLK Esta entrada determina o ritmo de funcionamento do P8_V1. O P8_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 Entrada de natureza assíncrona que quando activada 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. Para garantir que a primeira instrução após reset é executada, a entrada AR no registo PC só é desactivada na fase ϕ2.

Descrição dos sinais de saída do módulo de controlo:

- **SO** (*Select Offset*) determina se o próximo *fetch* se realiza no endereço dado por PC+1, ou em PC mais o parâmetro offset5 contido nas instruções de *branch* condicional e incondicional;
- **ERF** (*Enable Registers*) permite a escrita no *register file*. Este sinal é activo na execução das instruções **`ldr, mov, add`** e **`sub.`**
- **EP** (*Enable* PSR) controla a escrita das *flags* no registo PSR aquando das operações de processamento **add** e **sub**;
- **SD** (*Select Data*) selecciona qual a informação a carregar no *register file*. Para a instrução **`mov`** selecciona a constante imm3 contida na instrução. Na instrução **`ldr`** 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 de acesso à memória de dados. No caso de **`ldr/str direct3`**, selecciona o parâmetro contido na instrução. Caso o modo de endereçamento seja indirecto, selecciona a saída B do *register file*.
- **DA** (*Destination Address*) é utilizado nas instruções **`mov`**, **`ldr`**, **`add`** e **`sub`** para seleccionar qual dos registos do *register file* recebe o valor presente na entrada **D** do *register File*. Nas instruções de str determina qual o registo do *register file* que fornece o valor a ser escrito na memória de dados.
- **AA** (*A Address*) estabelece no *register file* qual o registo que fornece o valor para a saída A.
- **BA** (B *Address*) estabelece no *register file* qual o registo que fornece o valor para a saída B.
- **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.

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 14 bits como mostra a Tabela 11-3.

IInstruçã		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
		SO		EP	ERF	RD	SA	SD		Code	Parâmetros				
ldr	rd,direct3	0	0	0	1	1	0	0	1	-	d	d	d	rd	rd
ldr	rd,[rb]	0	0	0	1	1	1	0	1	-	-	rb	rb	rd	rd
str	rs,direct3	0	0	0	0	0	0	-	-	-	d	d	d	rs	rs
str	rs,[rb]	0	0	0	0	0	1	-	-	-	-	rb	rb	rs	rs
mov	rd,imm3	0	0	0	1	1	-	0	0	-	i	i	i	rd	rd
add	rn,rm	0	0	1	1	1	-	1	0	0	-	rm	rm	rn	rn
sub	rn,rm	0	0	1	1	1	-	1	0	1	-	rm	rm	rn	rn
bzs/beq	offset5	0	1	0	0	1	-	-	-	-	off	off	off	off	off
bcs/blo	offset5	1	0	0	0	1	-	-	-	-	off	off	off	off	off
b	offset5	1	1	0	0	1	-	-	-	-	off	off	off	off	off

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 14 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_{13} são constantes. Tratando-se de apenas 10 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 P8_V1. Dado que o P8_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 P8, 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 P8_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
`ldr rd,direct3`	0	0	0	d	d	d	rd	rd
`ldr rd,[rb]`	0	0	1	0	rb	rb	rd	rd
`str rs,direct3`	0	1	0	d	d	d	rs	rs
`str rs,[rb]`	0	0	1	1	rb	rb	rs	rs
`mov rd,imm3`	0	1	1	i	i	i	rd	rd
`add rn,rm`	1	0	0	0	rm	rm	rn	rn
`sub rn,rm`	1	0	0	1	rm	rm	rn	rn
`bzs/beq offset5`	1	0	1	off	off	off	off	off
`bcs/blo offset5`	1	1	0	off	off	off	off	off
`b offset5`	1	1	1	off	off	off	off	off

Tabela 11-4 - Codificação das instruções

11.3.4 Modulo Descodificador (*Instruction Decoder*)

Dado que as instruções chegam ao CPU codificadas, o P8_V1 passa a apresentar a estrutura mostrada na Figura 11-5.

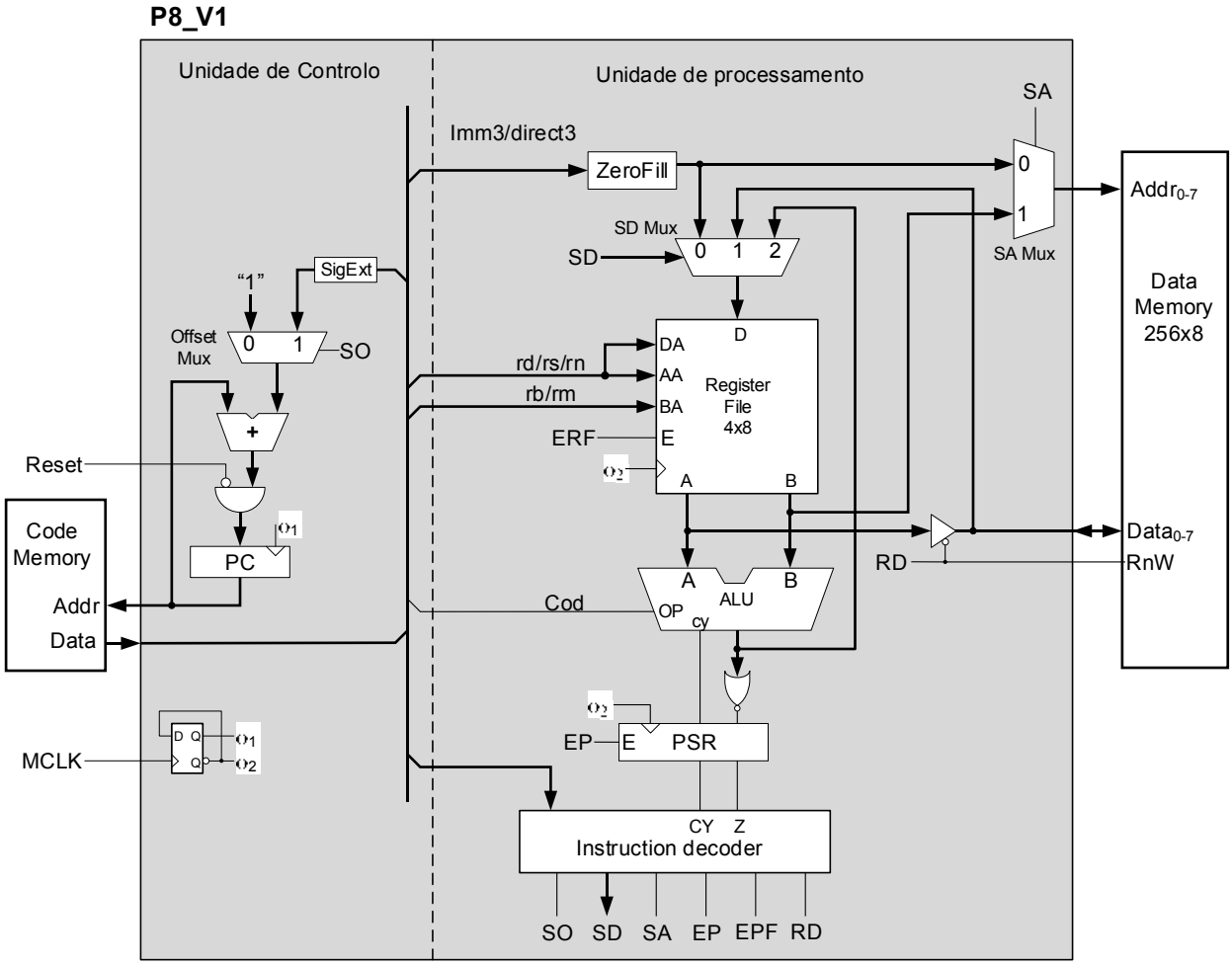


Figura 11-5 – P8_V1, Diagrama de blocos com micro programação vertical

É necessário conceber um módulo descodificador, que tendo como entrada o código da instrução mais as *flags* CY e Z, produza as micro-operações necessárias 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 64x7 cuja programação é apresentada na Tabela 11-5. Constituem endereço da ROM, os 4 bits mais significativos da instrução, conjuntamente com as *flags* CY e Z.

	I_7	I_6	I_5	I_4	CY	Z	SO	SA	SD	SD	RD	ERF	EP
ldr rd,direct3	0	0	0	-	-	-	0	0	0	1	1	1	0
ldr rd,[rb]	0	0	1	0	-	-	0	1	0	1	1	1	0
str rs,[rb]	0	0	1	1	-	-	0	1	-	-	0	0	0
str rs,direct3	0	1	0	-	-	-	0	0	-	-	0	0	0
mov rs,imm3	0	1	1	-	-	-	0	-	0	0	1	1	0
add/sub	1	0	0	-	-	-	0	-	1	0	1	1	1
bzs/beq offset5	1	0	1	-	-	0	0	-	-	-	1	0	0
bzs/beq offset5	1	0	1	-	-	1	1	-	-	-	1	0	0
bcs/blo offset5	1	1	0	-	0	-	0	-	-	-	1	0	0
bcs/blo offset5	1	1	0	-	1	-	1	-	-	-	1	0	0
b offset5	1	1	1	-	-	-	1	-	-	-	1	0	0

Tabela 11-5 – Módulo Descodificador

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:	ldr	r0,x_addr	0x00	0000 0000	0x00
	ldr	r3,[r0]	0x01	0010 0011	0x23
	str	r3,maior	0x02	0100 0111	0x47
	mov	r1,1	0x03	0110 0101	0x65
	str	r1,i	0x04	0100 1001	0x49
for:	mov	r2,6	0x05	0111 1010	0x7a
	sub	r1,r2	0x06	1001 1001	0x99
	blo	for_1	0x07	1100 0011	0xC3
	b	for_end	0x08	1110 1110	0xee
long_for:	b	for	0x09	1111 1100	0xfc
for_1:	ldr	r1,i	0x0A	0000 1001	0x09
	add	r1,r0	0x0B	1000 0001	0x81
	ldr	r2,[r1]	0x0C	0010 0110	0x26
	sub	r2,r3	0x0D	1001 1110	0x9e
	blo	else	0x0E	1100 0011	0xc3
	beq	else	0x0F	1010 0010	0xa2
then:	ldr	r3,[r1]	0x10	0010 0111	0x27
else:	ldr	r1,i	0x11	0000 0101	0x05
	mov	r2,1	0x12	0110 0110	0x66
	add	r1,r2	0x13	1000 1001	0x89
	str	r1,i	0x14	0100 1001	0x49
	b	long_for	0x15	1111 0100	0xf4
for_end:	str	r3,maior	0x16	0100 0111	0x47
	b	.	0x17	1110 0000	0xe0

Data Memory		
addr	Label	data
0x00	x_addr:	0x20
0x01	maior:	
0x02	i:	
0x20	x[0]	0x05
		0xb0
		0x08
		0x1f
		0x55
0x25	x[5]	0xaa