Manual do Simulador do Processador P3

Guilherme Arroz
José Monteiro
Arlindo Oliveira

Instituto Superior Técnico
Lisboa, Portugal

Fevereiro 2003

Conteúdo

1 Introdução

Este documento descreve a operação do simulador para o processador `P3`. Este programa permite simular a nível funcional o processador descrito nos Capítulos 11 e 12 do livro:

> Introdução aos Sistemas Digitais e Microprocessadores
> G. Arroz, J. Monteiro e A. Oliveira
> IST Press, 1ª Edição, 2004

O simulador `P3` é constituído por dois programas, pelo simulador propriamente dito, `p3sim`, e por um *assembler*, `p3as`. O programa `p3as` converte programas descritos na linguagem *assembly* daquele processador para um ficheiro objecto. Uma vez convertido para este formato objecto, o programa pode ser carregado para o simulador `p3sim`. O simulador `p3sim` permite não só a execução normal e passo-a-passo do programa, mas também a execução de apenas um ciclo de relógio. Este modo de funcionamento é útil para se observar a evolução passo-a-passo do micro-código. De forma a tornar mais interessante a interacção com o micro-processador, foram definidos um conjunto de dispositivos de entrada e saída.

Este documento está dividido em três partes. Na primeira parte introduz-se a arquitectura do processador `P3`. Na segunda parte, descreve-se a utilização do *assembler* `p3as`. Na terceira parte, é apresentado o simulador `p3sim`.

2 Arquitectura do Processador `P3`

2.1 Registos

O processador `P3` contém os seguintes registos visíveis ao programador:

`R0-R7`: registos de uso genérico. De facto, `R0` não é um registo, tem sempre o valor 0.

`PC`: *program counter*, contém o endereço da próxima instrução a executar. Não pode ser acedido directamente com instruções *assembly*, sendo alterado apenas com instruções de salto.

`SP`: *stack pointer*, apontador para o topo da pilha. É utilizado também de forma indirecta, podendo apenas ser manipulado directamente (para a sua inicialização) através de uma instrução `MOV SP, R[1-7]`.

`RE`: *registo de estado*, registo onde estão guardados os bits de estado (*flags*) do processador, descritos na secção seguinte. Também não existem instruções para manipular este registo directamente.

Todos estes registos são inicializados a 0 após um *reset* do processador.

2.2 Bits de Estado

Do ponto de vista do programador, existem 5 bits de estado, ou *flags*, neste processador. Os bits de estado estão guardados nos 5 bits menos significativos do registo `RE`, contendo os restantes bits deste registo o valor 0.

O significado dos bits de estado, do bit de menor para o de maior peso do registo `RE`, é:

`O`: *overflow* ou excesso, indica que o resultado da última operação aritmética excede a capacidade do operando destino. Por outras palavras, o resultado não pode ser representado em complemento para 2 com o número de bits disponíveis no operando destino, ficando este portanto com um valor incorrecto.

`N`: *negative* ou sinal, indica que o resultado da última operação foi negativo, o que em complemento para 2 é equivalente a dizer que o bit mais significativo do operando destino ficou a 1.

`C`: *carry* ou transporte, indica que a última operação gerou um bit de transporte para além da última posição do operando destino. Também pode ser modificado por software através das instruções `STC`, `CLC` e `CMC`.

`Z`: *zero*, indica que o resultado da última operação foi 0.

`E`: *enable interrupts*, permite controlar a aceitação ou não de interrupções, conforme for 1 ou 0, respectivamente. Este bit de estado é diferente no sentido em que o seu valor é directamente controlado pelo programador , através das instruções `ENI` e `DSI`.

2.3 Memória

O espaço de memória endereçável é de 64k palavras (barramento de endereços de 16 bits), em que cada palavra é de 16 bits (largura do barramento de dados). O acesso a uma posição de memória pode ser feito com qualquer instrução, usando o modo de endereçamento apropriado.

2.4 Entradas/Saídas

O espaço de entradas e saídas (I/O) é *memory mapped*. Os endereços de memória a partir de `FF00h` estão reservados para o espaço de entradas/saídas. Assim, qualquer instrução pode ter acesso a um qualquer dispositivo de entrada/saída que esteja mapeado neste espaço superior de memória do processador.

No caso do presente simulador, os dispositivos de entrada/saída disponíveis são:

- janela de texto: dispositivo que permite uma interface com o teclado e monitor do computador. Tem 4 portos de interface:
  - leitura, endereço `FFFFh`: porto que permite receber caracteres teclados na janela de texto;
  - escrita, endereço `FFFEh`: porto que permite escrever um dado caracter na janela de texto;
  - estado, endereço `FFFDh`: porto que permite testar se houve alguma tecla premida na janela de texto;
  - controlo, endereço `FFFCh`: porto que permite posicionar o cursor na janela de texto, posição esta onde será escrito o próximo caracter.
- interruptores, endereço `FFF9h`: conjunto de 8 interruptores cujo estado pode ser obtido por leitura deste endereço.

- LEDs, endereço FFF8h: conjunto de 8 LEDs que podem assumir 4 estados (apagado, vermelho, verde ou amarelo) por escrita neste endereço.
- *display* de 7 segmentos, endereços FFF0, FFF1h, FFF2h e FFF3h: cada um destes portos de escrita controla um conjunto de 8 LEDs que formam um *display*, 7 segmentos para o caracter, mais um LED para o ponto.

O controlo destes dispositivos é explicado em mais detalhe na Secção 4.6.

2.5 Interrupções

O simulador disponibiliza 4 botões para a geração de interrupções externas. Qualquer destas interrupções provoca a activação de um sinal INT, ligado a um dos pinos externos do processador. No final da execução de cada instrução, este sinal é testado para verificar se existe alguma interrupção pendente. Nesse caso, é chamada a rotina de serviço dessa interrupção, determinada pelo vector de interrupção, lido do barramento de dados. Os endereços das rotinas de interrupção encontram-se na Tabela de Vectores de Interrupção, uma tabela com 256 entradas guardada em memória a partir do endereço FE00h. Assim, o contador de programa PC é carregado com o valor da posição de memória M[FE00h+vector]. O vector de interrupção correspondente a cada um dos 4 botões de interrupção deve ser previamente definido pelo utilizador através da interface do simulador.

A chamada à rotina de serviço da interrupção guarda o registo RE na pilha e desabilita as interrupções (E=0). É da responsabilidade do programador salvaguardar qualquer registo que seja modificado nesta rotina. A rotina deve ser terminada com a instrução RTI que repõe o valor de RE a partir da pilha.

3 Assembler

3.1 Evocação

O modo de evocação do *assembler* p3as é simplesmente:

```
$ p3as <nome>.as
```

O nome do ficheiro *assembly* tem que ter extensão .as. Caso não haja erros de *assembly*, são gerados dois ficheiros:

<nome>.exe : ficheiro com o código binário, pronto a ser executado no simulador p3sim.

<nome>.lis : ficheiro com o valor atribuído às referências usadas no programa *assembly*.

3.2 Conjunto de Instruções

As instruções *assembly* aceites pelo *assembler* `p3as` são as apresentadas na Tabela 11.4 do livro. Para além destas instruções, o *assembler* reconhece um conjunto de comandos (chamados de *pseudo-instruções*, Tabela 11.16 do livro) que, embora não gerem código binário, permitem reservar espaço para variáveis ou tornar o código mais legível. O total de instruções reconhecidas pelo `p3as` encontram-se na tabela seguinte, agrupadas por classes:

Pseudo	Aritméticas	Lógicas	Deslocamento	Controlo	Transfer.	Genéricas
ORIG	NEG	COM	SHR	BR	MOV	NOP
EQU	INC	AND	SHL	BR.*cond*	MVBH	ENI
WORD	DEC	OR	SHRA	JMP	MVBL	DSI
STR	ADD	XOR	SHLA	JMP.*cond*	XCH	STC
TAB	ADDC	TEST	ROR	CALL	PUSH	CLC
	SUB		ROL	CALL.*cond*	POP	CMC
	SUBB		RORC	RET		
	CMP		ROLC	RETN		
	MUL			RTI		
	DIV			INT		

A condição *.cond* nas instruções de salto condicional (`BR.`*cond*, `JMP.`*cond* e `CALL.`*cond*) pode ser uma de:

`O`, `NO`: bit de estado excesso (*overflow*)

`N`, `NN`: bit de estado sinal (*negative*)

`C`, `NC`: bit de estado transporte (*carry*)

`Z`, `NZ`: bit de estado zero são atendidas (*enable*)

`I`, `NI`: bit que indica se existe alguma interrupção pendente

`P`, `NP`: resultado positivo ($\texttt{P} = \overline{\texttt{Z}} \wedge \overline{\texttt{N}}$)

Estas combinações permitem testar cada uma destas condições e realizar o salto caso a condição seja a 1 ou a 0, respectivamente.

As instruções aritméticas assumem os operandos em formato de complemento para 2. As excepções a esta regra são a multiplicação e a divisão que assumem números sem sinal. No caso destas duas operações, terá que ser o programador a ter o cuidado de manipular o sinal à parte.

Neste conjunto, há instruções de 0, 1 e 2 operandos. Nas instruções de 2 operandos, um deles tem que ser necessariamente um registo. O outro operando pode ter diversos modos de endereçamento, como se explica em seguida. Os detalhes do funcionamento de cada instrução (a operação realizada e os bits de estado alterados) são também apresentados mais adiante.

3.3 Constantes

O facto do processador P3 ser um processador de 16 bits define os valores máximos possíveis de especificar para uma constante. Assim, o intervalo válido para inteiros positivos será de 0 a $2^{16} - 1$ e para inteiros em complemento para 2 de -2^{15} a $+(2^{15} - 1)$.

Valores constantes podem ser especificados de três formas no código *assembly*:

Valor numérico em binário: para uma constante numérica ser interpretada em binário deve ser terminada com a letra `b`; são válidos valores entre `-1000000000000000b` e `1111111111111111b`.

Valor numérico em octal: para uma constante numérica ser interpretada em octal deve ser terminada com a letra `o`; são válidos valores entre `-100000o` e `177777o`.

Valor numérico em decimal: qualquer valor inteiro entre `-32768` e `65535`. Pode opcionalmente ser terminada com a letra `d`, embora tal seja assumido quando nenhuma outra base for indicada;

Valor numérico em hexadecimal: para uma constante numérica ser interpretada em hexadecimal deve ser terminada com a letra `h`; são válidos valores entre `-8000h` e `ffffh`.

Caracter alfanumérico: um caracter entre plicas, por exemplo, `'g'`, é convertido para o seu código ASCII.

Notar, no entanto, que o uso de constantes no meio do código *assembly* (ou de qualquer outra linguagem de programação) é extremamente desaconselhável. Em vez disso, deve-se usar o comando `EQU` para definir constantes (ver Secção 3.7). Esta prática, por um lado, torna o código mais legível, pois o símbolo associado à constante, se convenientemente escolhido, dá uma pista sobre a acção que se está a tomar, e por outro lado permite uma actualização mais fácil do código, pois constantes que estão associadas não têm que ser alteradas em vários sítios dentro do código (porventura falhando-se alguma), mas simplesmente na linha do comando `EQU`.

3.4 Modos de Endereçamento

Os operandos usados nas instruções *assembly* podem ter 7 modos de endereçamento, a seguir indicados.

O significado dos símbolos usados nesta secção é:

op:	operando;
`Rx`:	registo `Rx`. O processador tem 8 registos visíveis para o programador, portanto $0 \leq x \leq 7$, em que `R0` é sempre igual a 0;
`W`:	constante de valor `W` (de 16 bits);
`M[y]`:	referência à posição de memória com endereço `y`;
`PC`:	registo contador de programa (*program counter*);
`SP`:	registo do apontador para o topo da pilha (*stack pointer*)

Endereçamento por Registo op = Rx

O valor do operando é o conteúdo do registo Rx.

Endereçamento por Registo Indirecto op = M[Rx]

O valor do operando é o conteúdo da posição de memória cujo endereço é o conteúdo do registo Rx.

Endereçamento Imediato op = W

O valor do operando é W. Naturalmente, este modo não pode ser usado como operando destino.

Endereçamento Directo op = M[W]

O valor do operando é o conteúdo da posição de memória com o endereço W.

Endereçamento Indexado op = M[Rx+W]

O valor do operando é o conteúdo da posição de memória com o endereço resultante da soma de W com o conteúdo de Rx, Rx+W. Nota: a versão W+Rx não é aceite pelo *assembler*.

Endereçamento Relativo op = M[PC+W]

O valor do operando é o conteúdo da posição de memória com o endereço resultante da soma de W com o conteúdo de PC, PC+W. Nota: a versão W+PC não é aceite pelo *assembler*.

Endereçamento Baseado op = M[SP+W]

O valor do operando é o conteúdo da posição de memória com o endereço resultante da soma de W com o conteúdo de SP, SP+W. Nota: a versão W+SP não é aceite pelo *assembler*.

Na utilização destes modos de endereçamento, há duas restrições a saber:

- no caso das instruções com 2 operandos, para um deles tem que ser necessariamente usado o endereçamento por registo.
- o modo imediato não pode ser usado como operando destino, por razões óbvias.
- as instruções MUL e DIV, por usarem como destino ambos os operandos (ver descrição adiante), não podem usar o modo imediato em nenhum dos operandos. Além disso, os dois operandos não devem ser o mesmo devido a limitações na arquitectura do processador que provoca que parte do resultado se perca.

3.5 Etiquetas

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

```
VoltaAqui: INC R1
```

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

```
BR VoltaAqui
```

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

Para facilitar a leitura do código *assembly*, convenciona-se que estas etiquetas são palavras capitalizadas todas juntas: primeira letra de cada palavra em maiúsculas e restantes em minúsculas, como no exemplo anterior `VoltaAqui`.

O valor atribuído às etiquetas pode ser consultado no ficheiro com a extensão `.lis`, gerado quando da execução do `p3as`.

3.6 Comentários

Um comentário começa com o caracter `;`, que indica ao *assembler* que todo o texto que se segue nessa linha deverá ser ignorado no processo de tradução do código `assembly`.

3.7 Pseudo-Instruções

Chamam-se pseudo-instruções ao conjunto de comandos reconhecidos pelo *assembler* que não são instruções *assembly*, mas permitem dar ao *assembler* um conjunto de informações e directivas necessárias para a sua correcta execução ou para simplificar a sua utilização. A função das pseudo-instruções é, por um lado, controlar a forma como o código é gerado (por exemplo, indicando as posições de memória onde colocar o executável ou reservando posições de memória para dados), por outro lado, permitir definir símbolos (constantes ou posições de memória) que tornam o código mais legível e mais fácil de programar. Nesta secção descrevem-se as pseudo-instruções usadas pelo *assembler* `p3as`.

ORIG

Formato: `ORIG <endereço>`

Função: o comando `ORIG` permite especificar no campo `<endereço>` a primeira posição de memória em que um bloco de programa ou dados é carregado em memória. Este comando pode aparecer várias vezes no código, permitindo que se definam blocos em diferentes zonas de memória.

EQU

Formato: `<símbolo> EQU <const>`

Função: o comando `EQU` permite associar um valor `const` a um `símbolo`. Convenciona-se que estes símbolos são palavras todas em maiúsculas, com uso possível do caracter de separação '_', por exemplo, `NUM_LINHAS`.

Nota: Este comando associa um nome a uma constante. Isto permite que, no código *assembly*, em vez de um valor numérico que em geral não dá muita informação, se use um nome que pode indicar que tipo de acção se está a tomar nesse ponto do código. Adicionalmente, permite que numa posterior alteração baste alterar a linha do comando `EQU` para que a alteração

se propague pelo código todo.

WORD

Formato: `<etiqueta> WORD <const>`

Função: o comando `WORD` permite reservar uma posição de memória para conter uma variável do programa *assembly*, associando a essa posição o nome especificado em `<etiqueta>`. O campo `const` indica o valor a que essa posição de memória deve ser inicializada. Convenciona-se que estas etiquetas são palavras capitalizadas todas juntas: primeira letra de cada palavra em maiúsculas e restantes em minúsculas, por exemplo `CicloInterno`.

STR

Formato: `<etiqueta> STR 'texto'|<const>[,'texto'|<const>]`

Função: o comando `STR` coloca em posições de memória consecutivas o texto que estiver entre plicas ou o valor de `<const>`. No caso de `<texto>`, o código ASCII de cada caracter entre plicas fica numa posição de memória (portanto usa tantas posições de memória quantos os caracteres em `<texto>`). Podem-se usar mais do que um parâmetro, separados por vírgulas, sendo feita a sua concatenação em memória. `<etiqueta>` fica com o endereço do primeiro caracter. A convenção para os nomes destas etiquetas é o mesmo que para `WORD`.

TAB

Formato: `<etiqueta> TAB <const>`

Função: o comando `TAB` reserva o número de posições de memória especificados no campo `<const>` sem as inicializar com qualquer valor. `<etiqueta>` fica com o endereço da primeira posição. A convenção para os nomes destas etiquetas é o mesmo que para `WORD` e `STR`.

3.8 Instruções Assembly

As instruções *assembly* válidas para o micro-processador `P3` são apresentadas em seguida por ordem alfabética. É indicado o formato da instrução, a função realizada e as *flags* alteradas (Z, zero; C, *carry* ou transporte; N, *negative* ou sinal; O, *overflow* ou excesso; E, *enable* das interrupções).

ADD

Formato: `ADD op1, op2` *Flags*: ZCNO

Acção: *op1* ← *op1* + *op2*, soma a *op1* o valor de *op2*.

ADDC

Formato: `ADDC op1, op2` *Flags*: ZCNO

Acção: *op1* ← *op1* + *op2* + *C*, igual a `ADD` excepto que soma mais um caso o bit de estado transporte esteja a 1.

AND

Formato: `AND op1, op2` *Flags*: ZN

Acção: *op1* ← *op1* ∧ *op2*. Faz o AND lógico bit-a-bit dos dois operandos.

BR

Formato: `BR <deslocamento>` *Flags*: Nenhuma

Acção: *PC* ← *PC* + <*deslocamento*>, *branch*, salto relativo incondicional para `<deslocamento>` posições de memória à frente (ou atrás, se `<deslocamento>` for negativo) da posição actual. O valor de `<deslocamento>` tem que estar compreendido entre -32 e 31. Normalmente `<deslocamento>` é especificado com uma etiqueta.

BR.*cond*

Formato: `BR.cond <deslocamento>` *Flags*: Nenhuma

Acção: salto relativo condicional baseado no valor de um dada condição. As versões disponíveis são:

Condição	Transporte	Sinal	Excesso	Zero	Interrupção	Positivo
Verdade	`BR.C`	`BR.N`	`BR.O`	`BR.Z`	`BR.I`	`BR.P`
Falso	`BR.NC`	`BR.NN`	`BR.NO`	`BR.NZ`	`BR.NI`	`BR.NP`

Caso a condição se verifique, a próxima instrução a ser executada será a do endereço `PC + <deslocamento>` (*PC* ← *PC* + <*deslocamento*>). Caso contrário, funciona como um `NOP`. O valor de `<deslocamento>` tem que estar compreendido entre -32 e 31. Normalmente `<deslocamento>` é especificado com uma etiqueta.

CALL

Formato: `CALL <endereço>` *Flags*: Nenhuma

Acção: *M[SP]* ← *PC*, *SP* ← *SP - 1*, *PC* ← <*endereço*>, chamada a subrotina com início em `<endereço>`. O endereço da instrução seguinte ao `CALL` é colocado na pilha e é feito uma salto para a subrotina. Normalmente `<endereço>` é especificado com uma etiqueta.

CALL.*cond*

Formato: `CALL.cond <endereço>` *Flags*: Nenhuma

Acção: chamada condicional a uma subrotina baseado no valor de um dado bit de estado. As versões disponíveis são:

Condição	Transporte	Sinal	Excesso	Zero	Interrupção	Positivo
Verdade	`CALL.C`	`CALL.N`	`CALL.O`	`CALL.Z`	`CALL.I`	`CALL.P`
Falso	`CALL.NC`	`CALL.NN`	`CALL.NO`	`CALL.NZ`	`CALL.NI`	`CALL.NP`

Caso a condição se verifique, comporta-se como uma instrução `CALL`. Caso contrário, funciona como um `NOP`. Normalmente `<endereço>` é especificado com uma etiqueta.

CLC

Formato: `CLC` *Flags*: C

Acção: *clear C*, coloca o bit de estado transporte a 0.

CMC

Formato: `CMC` *Flags*: C

Acção: complementa o valor do bit de estado transporte.

CMP

Formato: `CMP op1, op2` *Flags*: ZCNO

Acção: compara os operandos *op1* e *op2*, actualizando os bits de estado. Efectua a mesma operação que `SUB op1, op2` sem alterar nenhum dos operandos. É habitualmente seguida no programa por uma instrução `BR.`*cond*, `JMP.`*cond* ou `CALL.`*cond*

COM

Formato: `COM op` *Flags*: ZN

Acção: $op \leftarrow \overline{op}$, faz o complemento bit-a-bit de *op*.

DEC

Formato: `DEC op` *Flags*: ZCNO

Acção: *op* ← *op - 1*, decrementa *op* em uma unidade.

DIV

Formato: `DIV op1, op2` *Flags*: ZCNO

Acção: executa a divisão inteira de *op1* por *op2*, deixando o resultado em *op1* e o resto em *op2*. Assume operandos sem sinal. O bit de estado `O` fica a 1 no caso de divisão por 0. Os bit de estado `C` e `N` ficam sempre a 0. Uma vez que ambos os operandos são usados para guardar o resultado, nenhum deles pode estar no modo imediato. Pela mesma razão, os dois operandos não devem ser o mesmo pois parte do resultado será perdido.

DSI

Formato: `DSI` *Flags*: E

Acção: *disable interrupts*, coloca o bit de estado `E` a 0, inibindo assim as interrupções.

ENI

Formato: `ENI` *Flags*: E

Acção: *enable interrupts*, coloca o bit de estado `E` a 1, permitindo assim as interrupções.

INC

Formato: `INC op` *Flags*: ZCNO

Acção: *op* ← *op* + *1*, incrementa *op* em uma unidade.

INT

Formato: `INT const` *Flags*: ZCNOE

Acção: *M[SP]* ← *RE, SP* ← *SP - 1, M[SP]* ← *PC, SP* ← *SP - 1, RE* ← *0, PC* ← *M[FE00h+const]*, gera uma interrupção com o vector `const`. Este vector tem que estar compreendido entre 0 e 255. Esta interrupção ocorre sempre, independentemente do valor do bit de estado `E`, *enable interrupts*.

JMP

Formato: `JMP <endereço>` *Flags*: Nenhuma

Acção: *PC* ← *<endereço>*, *jump*, salto absoluto incondicional para a posição de memória com o valor `<endereço>`. Normalmente `<endereço>` é especificado com uma etiqueta.

JMP.*cond*

Formato: `JMP.cond <endereço>` *Flags*: Nenhuma

Acção: salto absoluto condicional baseado no valor de um dada condição. As versões disponíveis são:

Condição	Transporte	Sinal	Excesso	Zero	Interrupção	Positivo
Verdade	`JMP.C`	`JMP.N`	`JMP.O`	`JMP.Z`	`JMP.I`	`JMP.P`
Falso	`JMP.NC`	`JMP.NN`	`JMP.NO`	`JMP.NZ`	`JMP.NI`	`JMP.NP`

Caso a condição se verifique, a próxima instrução a ser executada será a apontada por `<endereço>` (*PC* ← *<endereço>*). Caso contrário, funciona como um `NOP`. Normalmente `<endereço>` é especificado com uma etiqueta.

MOV

Formato: `MOV op1, op2` *Flags*: Nenhuma

Acção: *op1* ← *op2*, copia o conteúdo de `op2` para `op1`.

Para além dos modos de endereçamento comuns a todas as intruções (conforme Secção 3.4), esta instrução permite ler e escrever no registo apontador da pilha `SP`, mas apenas em conjunção com o modo de endereçamento por registo: `MOV SP, Rx` e `MOV Rx, SP`. A primeira destas instruções será necessária no início de todos os programas que utilizem a pilha.

MUL

Formato: `MUL op1, op2` *Flags*: ZCNO

Acção: *op1|op2* ← *op1* × *op2*, multiplica `op1` por `op2`, assumindo-os como números sem sinal. Como o resultado necessita de 32 bits são usados os dois operandos para o guardar: `op1` fica com os 16 mais significativos e `op2` com os 16 menos significativos. O bit de estado `Z` é actualizado de acordo com o resultado, os restantes ficam a 0. Uma vez que ambos os operandos são usados para guardar o resultado, nenhum deles pode estar no modo imediato. Pela mesma razão, os dois operandos não devem ser o mesmo pois parte do resultado será perdido.

MVBH

Formato: `MVBH op1, op2` *Flags*: Nenhuma

Acção: *op1* ← *(op1* ∧ *00FFh)* ∨ *(op2* ∧ *FF00h)*, copia o octecto de maior peso de `op2` para o octecto de maior peso de `op1`.

MVBL

Formato: `MVBL op1, op2` *Flags*: Nenhuma

Acção: *op1* ← *(op1* ∧ *FF00h)* ∨ *(op2* ∧ *00FFh)*, copia o octecto de menor peso de `op2` para o octecto de menor peso de `op1`.

NEG

Formato: `NEG op` *Flags*: ZCNO

Acção: *op* ← *-op*, troca o sinal (complemento para 2) do operando `op`.

NOP

Formato: `NOP` *Flags*: Nenhuma

Acção: *no operation*, não altera nada.

OR

Formato: `OR op1, op2` *Flags*: ZN

Acção: *op1* ← *op1* ∨ *op2*, faz o OR lógico bit-a-bit dos dois operandos.

POP

Formato: `POP op` *Flags*: Nenhuma

Acção: *SP* ← *SP* + *1*, *op* ← *M[SP]*, copia o valor do topo da pilha para `op` e reduz o tamanho desta.

PUSH

Formato: `PUSH op` *Flags*: Nenhuma

Acção: *M[SP]* ← *op*, *SP* ← *SP - 1*, coloca `op` no topo da pilha.

RET

Formato: `RET` *Flags*: Nenhuma

Acção: *SP* ← *SP* + *1*, *PC* ← *M[SP]*, retorna de uma subrotina. O endereço de retorno é obtido do topo da pilha.

RETN

Formato: `RETN const` *Flags*: Nenhuma

Acção: *SP* ← *SP* + *1*, *PC* ← *M[SP]*, *SP* ← *SP* + *const*, retorna de uma subrotina libertando `const` posições do topo da pilha. Esta instrução permite retornar de uma subrotina retirando automaticamente parâmetros que tenham sido passados para essa subrotina através da pilha. O valor de `const` tem que estar compreendido entre 0 e 1023 (10 bits).

ROL

Formato: `ROL op, const` *Flags*: ZCN

Acção: *rotate left*, faz a rotação à esquerda dos bits de `op` o número de vezes indicado por `const`. Mesma operação que o deslocamento simples, `SHL`, mas os bits da esquerda não se perdem, sendo colocados nas posições mais à direita de `op`. O valor de `const` tem que estar compreendido entre 1 e 15.

ROLC

Formato: `ROLC op, const` *Flags*: ZCN

Acção: *rotate left with carry*, mesma operação que `ROL`, mas envolvendo o bit de estado transporte: o valor de `C` é colocado na posição mais à direita de `op` e o bit mais à esquerda de `op` é colocado em `C`. O valor de `const` tem que estar compreendido entre 1 e 15.

ROR

Formato: `ROR op, const` *Flags*: ZCN

Acção: *rotate right*, faz a rotação à direita dos bits de `op` o número de vezes indicado por `const`. Mesma operação que o deslocamento simples, `SHR`, mas os bits da direita não se perdem, sendo colocados nas posições mais à esquerda de `op`. O valor de `const` tem que estar compreendido entre 1 e 15.

RORC

Formato: `RORC op, const` *Flags*: ZCN

Acção: *rotate right with carry*, mesma operação que `ROR`, mas envolvendo o bit de estado transporte: o valor de `C` é colocado na posição mais à esquerda de `op` e o bit mais à direita de `op` é colocado em `C`. O valor de `const` tem que estar compreendido entre 1 e 15.

RTI

Formato: `RTI` *Flags*: EZCNO

Acção: *SP ← SP + 1, PC ← M[SP], SP ← SP + 1, RE ← M[SP], return from interrupt*, retorna de uma rotina de serviço a uma interrupção. O endereço de retorno e os bits de estado são obtidos do topo da pilha, por esta ordem.

SHL

Formato: `SHL op, const` *Flags*: ZCN

Acção: *shift left*, deslocamento à esquerda dos bits de `op` o número de vezes indicado por `const`. Os bits mais à esquerda de `op` são perdidos e é colocado 0 nas posições mais à direita. O bit de estado transporte fica com o valor do último bit perdido. O valor de `const` tem que estar compreendido entre 1 e 15.

SHLA

Formato: `SHLA op, const` *Flags*: ZCNO

Acção: *shift left arithmetic*, mesma operação que `SHL`, mas actualizando os bits de estado correspondentes às operações aritméticas. Permite realizar de forma expedita uma multiplicação de `op` por 2^n. O valor de `const` tem que estar compreendido entre 1 e 15.

SHR

Formato: `SHR op, const` *Flags*: ZCN

Acção: *shift right*, deslocamento à direita dos bits de `op` o número de vezes indicado por `const`. Os bits mais à direita de `op` são perdidos e são colocados 0 nas posições mais à esquerda. O bit de estado transporte fica com o valor do último bit perdido. O valor de `const` tem que estar compreendido entre 1 e 15.

SHRA

Formato: `SHRA op, const` *Flags*: ZCNO

Acção: *shift right arithmetic*, deslocamento à direita dos bits de `op`, mas mantendo o bit de sinal. Os bits mais à direita de `op` são perdidos, mas os bits mais à esquerda mantêm o valor anterior. O bit de estado transporte fica com o valor do último bit perdido. Permite realizar de forma expedita uma divisão de `op` por 2^n. `const` entre 1 e 15.

STC

Formato: `STC` *Flags*: C

Acção: *set C*, coloca o bit de estado transporte a 1.

SUB

Formato: `SUB op1, op2` *Flags*: ZCNO

Acção: *op1 ← op1 - op2*, subtrai a *op1* o valor de *op2*.

SUBB

Formato: `SUBB op1, op2` *Flags*: ZCNO

Acção: *op1* ← *op1 - op2 - C*, igual a `SUB` excepto que subtrai mais um caso o bit de estado transporte esteja a 1.

TEST

Formato: `TEST op1, op2` *Flags*: ZN

Acção: testa o bits dos operandos *op1* e *op2*, actualizando os bits de estado. Efectua a mesma operação que `AND op1, op2` sem alterar nenhum dos operandos.

XCH

Formato: `XCH op1, op2` *Flags*: Nenhuma

Acção: *exchange op1/op2, op1* ← *op2, op2* ← *op1*, troca os valores de `op1` e `op2`.

XOR

Formato: `XOR op1, op2` *Flags*: ZN

Acção: *op1* ← *op1* ⊕ *op2*. Faz a operação lógica EXCLUSIVE-OR bit-a-bit dos dois operandos.

4 Simulador

4.1 Evocação

O modo de evocação do *simulador* `p3sim` é simplesmente:

```
$ p3sim [<nome>.exe]
```

em que `<nome>.exe` é o executável gerado pelo assembler `p3as` que se pretende simular. Os parêntesis rectos indicam que o ficheiro <nome>.exe é opcional, o programa a simular pode também ser carregado através da interface do simulador.

Para sair do simulador deve-se escolher a opção *Sai* do menu *Ficheiro*.

4.2 Ambiente

A evocação do simulador lança uma janela como a representada na Figura 1.

Nesta janela existem 6 secções diferentes que se explicam em seguida, no sentido de cima para baixo na janela.

4.2.1 Menus

Na parte superior da janela existem 5 menus que se abrem quando seleccionados: *Ficheiro*, *Definições*, *Comandos*, *Depuração* e *Ver*. Qualquer destes menus pode ser mantido aberto seleccionando a primeira linha (a tracejado).

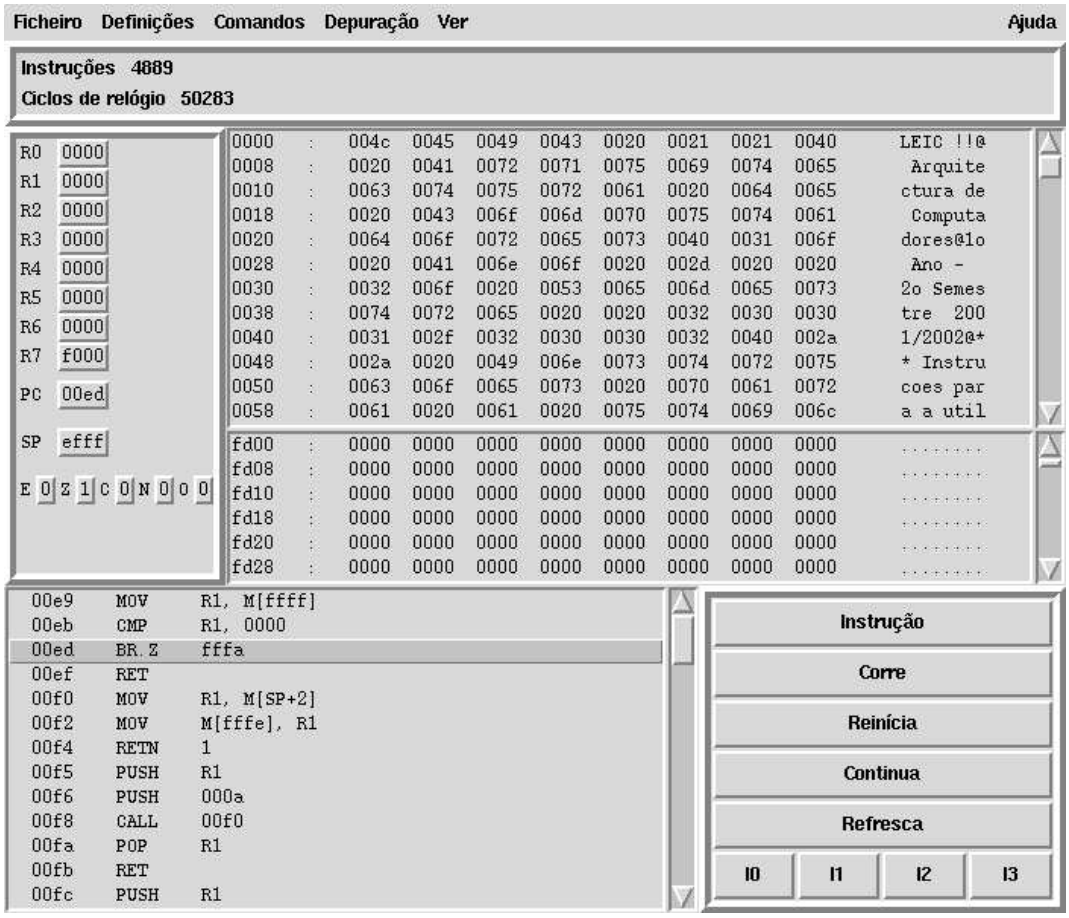


Figura 1: Interface do simulador.

As opções de cada menu são as seguintes:

- **Ficheiro:**
  as opções deste menu estão relacionadas com a manipulação de ficheiros, quer para leitura quer para escrita.

  Carrega Programa permite carregar para o simulador um novo programa gerado pelo `p3as`.

  Escreve Memória escreve para um ficheiro o conteúdo actual da memória. O ficheiro gerado é texto com uma posição de memória por cada linha, com endereço e conteúdo dessa posição. Todos estes valores são de 16 bits e estão em hexadecimal.

  Carrega Memória carrega directamente algumas posições de memória. O ficheiro de entrada deve ser em texto, com o mesmo formato gerado pelo comando *Escreve Memória*, uma posição de memória por cada linha, com endereço e conteúdo dessa posição. Podem especificar-se o número de posições que se quiser e a sua ordem não é importante. Todos estes valores têm que estar em hexadecimal e ser de 16 bits.

  Carrega ROM de Controlo permite alterar ao conteúdo da ROM da unidade de controlo. Esta opção é útil para modificar o micro-programa das instruções. O fi-

cheiro de entrada deve ser texto, com uma posição de memória por linha. Em cada linha deve constar o endereço da posição a alterar (a ROM tem um barramento de endereços de 9 bits, portanto 512 posições de memória) e o novo valor a colocar nessa posição (cada posição desta ROM tem 32 bits), tudo em hexadecimal.

Carrega ROM A permite alterar ao conteúdo da ROM que faz o mapeamento de instruções. A ROM A é endereçada com o campo do código da instrução *assembly* presente no registo de instrução, colocando à saída o endereço de início da micro-rotina que realiza esta instrução na ROM de controlo. Esta opção é útil para acrescentar novas instruções ou modificar o micro-programa de instruções já existentes. O ficheiro de entrada deve ser texto, com uma posição de memória por linha. Em cada linha deve constar o endereço da posição a alterar (esta ROM tem 64 posições) e o novo valor a colocar nessa posição (cada posição desta ROM tem 9 bits), tudo em hexadecimal.

Carrega ROM B permite alterar ao conteúdo da ROM que faz o mapeamento do modo de endereçamento. A ROM B é endereçada com o campo do modo de endereçamento da instrução *assembly* presente no registo de instrução, de acordo com a Figura 12.8 do livro, colocando à saída o endereço da sub-micro-rotina na ROM de controlo que lê/escreve os operandos de acordo com esse modo. Esta opção é útil para acrescentar ou modificar os modos de endereçamento existentes. O ficheiro de entrada deve ser texto, com uma posição de memória por linha. Em cada linha deve constar o endereço da posição a alterar (esta ROM tem 16 posições) e o novo valor a colocar nessa posição (cada posição desta ROM tem 9 bits), tudo em hexadecimal.

Sai saída do programa, perdendo-se toda a informação sobre o contexto da simulação.

- **Definições:**
menu com opções de configuração do próprio simulador.

  Define IVAD define quais os vectores de interrupção activos e permite especificar em hexadecimal os vectores associados a cada interruptor.

  Zona de Memória permite alterar quais as posições de memória visualizadas na secção da memória (ver Secção 4.2.4).

  Zona de Programa permite alterar o número de posições de memória visualizadas na secção de programa desassemblado (ver Secção 4.2.5).

- **Comandos:**
os comandos deste menu são os mesmos que os descritos na Secção 4.2.6. A razão da duplicação é que por vezes pode ser útil ter este menu fixo numa janela pequena e independente.

- **Depuração:**
neste menu estão um conjunto de opções que facilitam a depuração de programas.

  Pontos de Paragem esta opção lista os pontos de paragem (ou *breakpoints*, endereços onde a execução do programa pára) que estão definidos. Para apagar todos os pontos de paragem basta clicar em *Apaga Todos*. Para apagar um determinado

ponto de paragem deve-se clicar sobre ele (quer nesta janela quer na do programa) e depois clicar em *Apaga*. Para definir um novo ponto de paragem numa dada linha do código, deve-se seleccionar essa linha na janela do programa e depois clicar em *Adiciona*.

Escreve Registo permite alterar directamente o conteúdo dos registos. O valor deve estar em hexadecimal.

Escreve Memória permite alterar directamente uma posição de memória. Os valores do endereço e conteúdo devem estar em hexadecimal. Importante: se se alterar o conteúdo de uma posição correspondente ao código, a janela de programa não será actualizada (não há uma nova desassemblagem do programa), e portanto haverá alguma inconsistência.

- **Ver:**
este menu tem opções para activar/desactivar janelas ou informação extra no simulador.

  Ver Controlo estende ou reduz a interface do simulador, permitindo visualizar informação interna da unidade de controlo. Este modo de funcionamento é descrito na Secção 4.4.

  Ver ROMs cria ou elimina uma janela que mostra o conteúdo de cada posição de memória das três ROMs da unidade de controlo: ROM de mapeamento A, ROM de mapeamento B e ROM de controlo.

  Janela Texto cria ou elimina a janela de entrada e saída de texto. Como descrito na Secção 4.6.1, as entradas/saídas para esta janela estão mapeadas nos endereços `FFFCh` a `FFFFh`. Portanto, leituras e escrita para esta gama de endereços controlam este dispositivo de acordo com o descrito nessa secção.

  Janela Interface cria ou elimina a janela de entrada e saída com 4 *displays* de 7 segmentos, 8 LEDs e 8 interruptores. Como descrito na Secção 4.6.2, as entradas/saídas para controlar estes dispositivos estão mapeadas nos endereços `FFF0h` a `FFF3h` para os *displays*, `FFF8h` para os LEDs e `FFF9h` para os interruptores. Portanto, leituras e escrita para estes endereços controlam os dispositivos de acordo com o descrito nessa secção.

4.2.2 Contadores de Instrução e Ciclos de Relógio

Por baixo dos menus, existe uma secção que mostra o número de instruções e o número de ciclos de relógio que decorreram desde que se iniciou a última execução do programa.

4.2.3 Registos

A secção imediatamente abaixo à esquerda indica o valor actual de cada registo da unidade de processamento. São apresentados os registos de uso genérico (`R0` a `R7`), o contador de programa `PC` (*program counter*) e o apontador para o topo da pilha `SP` (*stack pointer*). Todos os valores estão em hexadecimal, com 16 bits.

Estão também indicados os bits de estado (*flags*) do sistema (cujo valor é naturalmente 0 ou 1): `O`, excesso ou *overflow*; `C`, transporte ou *carry*; `N`, sinal ou *negative*; `Z`, zero; e `E`, *enable interrupt*.

4.2.4 Conteúdo da Memória

Nesta secção é mostrado o conteúdo das diferentes posições de memória. Por razões de eficiência, não é possível ter acesso a todas as posições de memória simultaneamente. Assim optou-se por dar acesso a duas zonas diferentes da memória, o que se traduz na divisória ao meio desta secção. Inicialmente, a parte de cima aponta para a zona de memória onde tipicamente estão os dados e a parte de baixo para a zona da pilha e tabela de interrupção, com os valores:

	início	número de posições
parte de cima:	`8000h`	512
parte de baixo:	`FD00h`	512

Pelo menu *Definições*, é possível definir o endereço de início e o número de posições de memória a visualizar em cada uma destas zonas. Um aumento do número de posições a visualizar torna a execução do simulador mais lenta.

Em cada linha são apresentadas 8 posições de memória consecutivas. O endereço da primeira destas posições é o primeiro número da linha. Os seguintes 8 valores são o conteúdo dessas posições. Mais uma vez, todos os valores estão em hexadecimal e são de 16 bits. No final de cada linha estão os 8 caracteres com os códigos ASCII das posições de memória dessa linha. Caso o valor não corresponda ao código ASCII de um caracter alfa-numérico, é usado o caracter `'.'`.

4.2.5 Programa Desassemblado

Na secção em baixo à esquerda é apresentado o programa desassemblado. Sempre que um novo programa é carregado para o simulador, é feita a sua desassemblagem. Este processo consiste em interpretar os valores binários do ficheiro de entrada e imprimir a instrução *assembly* que lhes corresponde. Notar que não se tem acesso às etiquetas usadas no ficheiro *assembly* original, logo todos os valores são numéricos.

A barra escura indica a próxima instrução a ser executada. No entanto, esta pode ser colocada em qualquer instrução, clicando em cima dela. Isto permite que seja aí colocado um ponto de paragem, através da opção *Pontos de Paragem* do menu *Depuração*. As instruções com pontos de paragem são antecedidas no código com o sinal '»'. Para se remover um ponto de paragem pode-se clicar sobre essa instrução e fazer *Apaga* na mesma opção do menu *Depuração*.

Quando o programa se está a executar e pára num dado ponto de paragem, tal é assinalado pela cor vermelha da barra de selecção.

4.2.6 Comandos de Execução e Interrupção

No canto inferior direito estão os comandos que controlam a execução do programa:

Instrução – executa uma única instrução *assembly*.

Corre – reinicia o programa e executa-o indefinidamente ou até parar num ponto de paragem. O utilizador pode parar o programa em qualquer altura clicando no botão *Parar*.

Reinícia – reinicia o processador, colocando todos os registos a 0, excepto o `PC` que é colocado com o valor do endereço de início do programa.

Continua – continua a execução do programa a partir da instrução corrente. Este botão transforma-se num botão *Parar* permitindo ao utilizador parar a execução do programa em qualquer altura.

Refresca – actualiza a janela do programa sem parar a sua execução, mostrando o conteúdo da memória e dos registos na altura em que se clicou neste botão.

Na última linha estão 4 botões que correspondem às 4 interrupções para este processador: clicando num destes botões é gerada uma interrupção no programa com o correspondente vector de interrupção. Inicialmente as 4 interrupções estão desabilitadas. Para habilitar uma delas, tem que se seleccionar a opção *Define IVAD* no menu *Definições* e clicar na respectiva caixa de selecção. É também aqui que se define o vector associado a cada um destes 4 botões de interrupção.

4.3 Depuração

Tipicamente, as ferramentas disponíveis para ajudar na depuração de um programa em *assembly* são muito limitadas. A funcionalidade destas ferramentas é replicada no simulador `p3sim`.

Para testar a funcionalidade de uma secção do código, começa-se por colocar um ponto de paragem (como indicado atrás) no início dessa secção e dá-se o comando *Corre*. Após a sua paragem, executa-se o programa passo-a-passo, verificando se o fluxo do programa é o previsto e se depois de cada instrução os registos, bits de estado e posições de memória foram alterados de acordo com o esperado. Caso tal não aconteça, é possível que se tenha que repetir este procedimento para se tentar perceber porque que é que o comportamento do programa é diferente do esperado.

Por vezes é desejável criar artificialmente as condições que se quer testar. Para isso podem-se carregar os registos/posições de memória com os valores necessários para o teste que se pretende.

Basicamente, são estes os procedimentos a seguir. Portanto, a não ser que se tenha uma intuição muito apurada para depuração de programas *assembly* que dê uma ideia muito boa de onde o erro poderá estar a surgir, é vivamente recomendado que o teste dos programas seja feito módulo a módulo. Só depois de os módulos terem sido testados separadamente sob condições típicas e se ter bastante confiança no seu correcto funcionamento é que se deve começar a juntá-los e a testá-los em conjunto.

4.4 Unidade de Controlo

O simulador `p3sim` faz simulação ao nível do micro-código. Para se ter acesso à informação interna da unidade de controlo (portanto informação que não está disponível a nível da programação *assembly*) deve-se seleccionar a opção *Ver controlo* do menu *Ver*. Após esta selecção a interface é estendida, ficando como mostra a Figura 2.

Em particular, temos mais uma secção na janela da interface com os registos internos da unidade de controlo e mais um botão (*Clock*) na secção de comandos de execução.

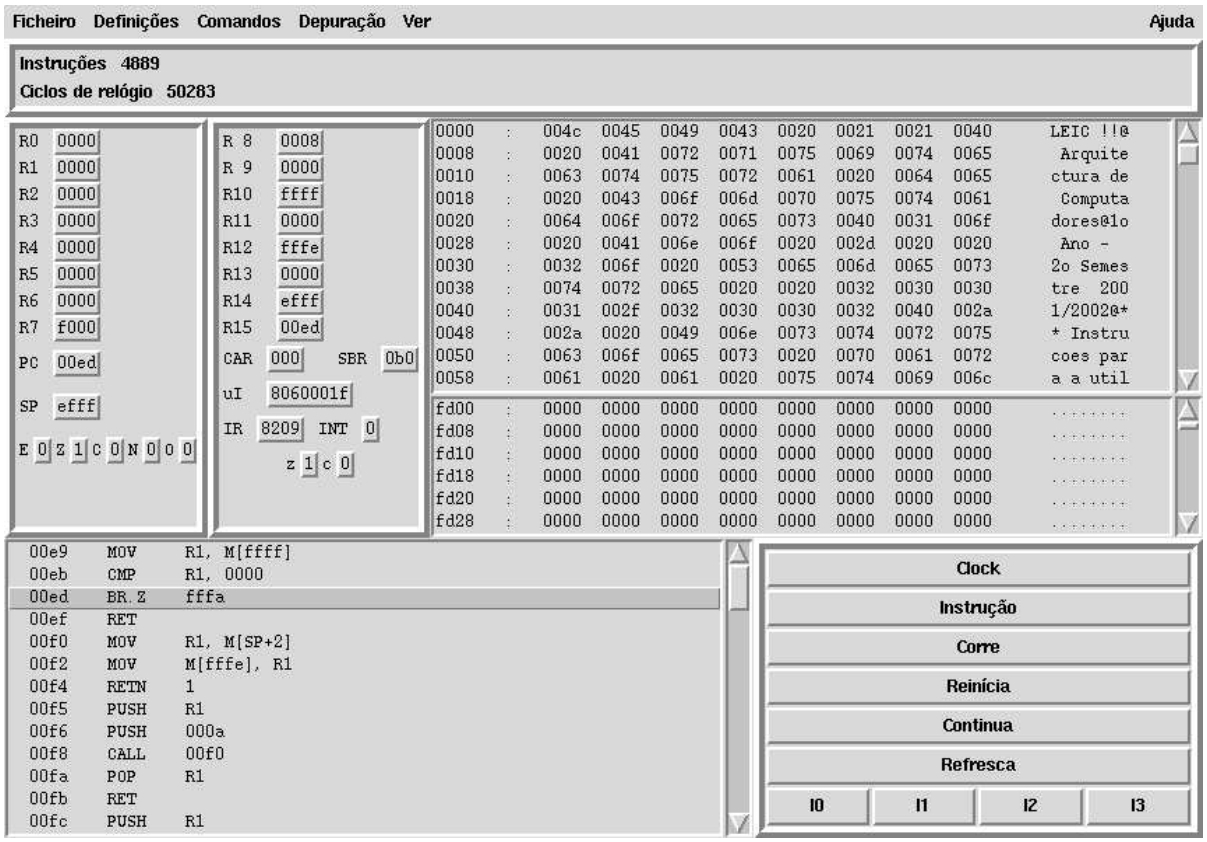


Figura 2: Interface do simulador estendida com a informação de controlo.

4.4.1 Registos Internos à Unidade de Controlo

A secção que aparece entre os registos e o conteúdo da memória mostra os valores dos registos internos à unidade de controlo. São registos que não são vistos pelo programador, mas que são usados pelos micro-programas das instruções do processador.

Os registos apresentados são:

`R8-R13` – conjunto de 6 registos de 16 bits de uso genérico para os micro-programas. Destes, os 3 últimos têm significados especiais pela maneira como são usados na estrutura dos micro-programas do processador:

- `R11` : também chamado de `SD` (*source data*), pois na fase de *operand fetch* da instrução é carregado com o valor do operando origem (*source*).
- `R12` : também chamado de `EA` (*effective address*), pois na fase de *operand fetch* da instrução é carregado com o endereço de memória onde eventualmente se vai buscar um dos operandos (aquele que não é usado em modo registo) e, na fase de *write-back*, onde se guarda o resultado caso o operando destino esteja em memória.
- `R13` : também chamado de `RD` (*result data*), pois na fase de *operand fetch* da instrução é carregado com o valor do operando destino e fica com o valor do resultado, a ser usado na fase de *write-back*.

`R14` – de facto, este é o registo apontador da pilha, `SP`, ou seja, o registo `SP` está no banco de registos e corresponde ao `R14`.

`R15` – de igual forma, este é o registo contador de programa, `PC`.

`CAR` – *control address register*, contém o endereço da micro-instrução a executar no próximo ciclo de relógio. Registo de 9 bits.

`SBR` – *subroutine branch register*, guarda o endereço de retorno quando se executa uma chamada a uma sub-rotina dentro de um micro-programa. Registo de 9 bits.

`uI` – *micro-instruction*, micro-instrução a ser executada no próximo ciclo de relógio. A micro-instrução tem 32 bits.

`IR` – *instruction register*, contém a instrução *assembly* que está a ser executada. Registo de 16 bits.

`INT` – *interrupt*, indica se existe ou não uma interrupção pendente, tomando os valores 1 ou 0 respectivamente.

`z,c` – bits de estado zero e transporte à saída da ULA, invisíveis ao programador e portanto apenas úteis na micro-programação. São actualizados todos os ciclos de relógio, ao contrário dos bits de estado em *assembly* cuja actualização ou não é controlada pelo micro-programa. Tomam os valores 0 ou 1.

4.4.2 Botão Clock

Este botão extra permite executar apenas um ciclo de relógio de cada vez. A sua utilidade é permitir seguir o funcionamento de um micro-programa, micro-instrução a micro-instrução. Para acabar a execução da instrução *assembly* actual pode usar-se o botão *Instrução*, que executa os ciclos de relógio necessários para chegar novamente ao início do ciclo de *fetch*.

Notar que o `PC` pode ficar momentaneamente numa zona inválida quando se carrega no botão *Clock*, o que é indicado pela mensagem "*A posição apontada pelo PC não contém uma instrução válida*". Isto deve-se a que, nas instruções que ocupam duas posições de memória, o `PC` possa ficar momentaneamente a apontar para a segunda posição de memória dessa instrução, que não corresponde a uma instrução *assembly*. Uma vez lida essa posição de memória, o `PC` é de novo incrementado, voltando a uma posição válida.

4.5 Micro-Programação

O simulador `p3sim` está desenvolvido de forma a permitir modificar o funcionamento das instruções *assembly* do processador e mesmo introduzir novas instruções. Este processo envolve modificar algumas posições de memória das ROMs do processador: a ROM de controlo e as ROMs de mapeamento, A e B. O conteúdo destas ROMs é apresentado no Anexo B.

A alteração de uma instrução pode, em princípio, ser feita modificando certas posições da ROM de controlo. Para isso, tem que se analisar o micro-programa da instrução *assembly* a alterar e identificar quais as posições do micro-programa que devem ser alteradas. Basta então criar um ficheiro de texto com uma linha por cada micro-instrução a alterar. Em cada linha deve constar o endereço da ROM de controlo a alterar seguido do valor desejado para

essa posição, todos os valores em hexadecimal. Este ficheiro deve depois ser carregado usando o opção *Carrega ROM de Controlo* do menu *Ficheiro*. O formato usado para as micro-instruções está apresentado no Anexo B.

Para adicionar uma instrução, é necessário:

1. arranjar um código de instrução (*opcode*) único.
2. desenvolver o micro-programa para essa instrução.
3. arranjar um espaço livre na ROM de controlo onde esse micro-programa vai ser colocado. No caso do `p3sim`, as posições livres são a partir do endereço `fah`, inclusivé.
4. carregar o micro-programa, conforme descrito no parágrafo anterior.
5. modificar a ROM de mapeamento A, colocando no endereço correspondente ao código da instrução nova o endereço da ROM de controlo onde se colocou o micro-programa, usando o mesmo procedimento que o usado para a alteração da ROM de controlo.

O conteúdo das ROMs do processador pode ser confirmado seleccionando a opção *Ver ROMs* do menu *Ver*.

A depuração do micro-programa pode ser feito seguindo passo-a-passo (que a este nível é equivalente a ciclo de relógio-a-ciclo de relógio) a sua execução com o botão *Clock* e verificando o fluxo e as alterações que o micro-programa provoca nos diferentes registos.

4.6 Dispositivos de Entrada e Saída

O simulador disponibiliza um conjunto de dispositivos de entrada e saída através de duas janelas que podem ser abertas através do menu *Ver*. Cada um destes dispositivos poderá ser acedido por um ou mais portos. Sendo o espaço de endereçamento de IO mapeado no espaço de endereçamento de memória, a cada porto corresponderá um endereço de memória. Estes portos podem ser de leitura, de escrita ou de leitura e escrita. Escritas para portos só de leitura são ignoradas. Leituras de portos de escrita retornam todos os bits a 1, ou seja, `FFFFh`.

4.6.1 Janela Texto

Esta janela, apresentada na Figura 3, permite uma interface a nível de texto, permitindo ler caracteres do teclado e escrever caracteres para o monitor. Para aceder a este dispositivo estão reservados 4 portos:

porto de leitura, endereço `FFFFh`: uma leitura deste porto retorna o código ASCII do caracter correspondente à última tecla premida sobre a janela de texto. Portanto, no caso de se premir uma tecla antes da leitura da tecla anterior faz com que esta se perca. É possível testar se existe alguma tecla para ler através do porto de estado. Uma leitura deste porto sem que tenha havido uma tecla premida retorna o valor 0.

porto de escrita, endereço `FFFEh`: porto que permite escrever um dado caracter na janela de texto. O caracter com o código ASCII igual ao valor escrito para este porto é ecoado na janela. Esta janela mantém internamente um cursor onde este caracter é escrito. Sempre que se faz uma escrita, este cursor avança. É possível posicionar-se o cursor em qualquer ponto da janela através do porto de controlo.

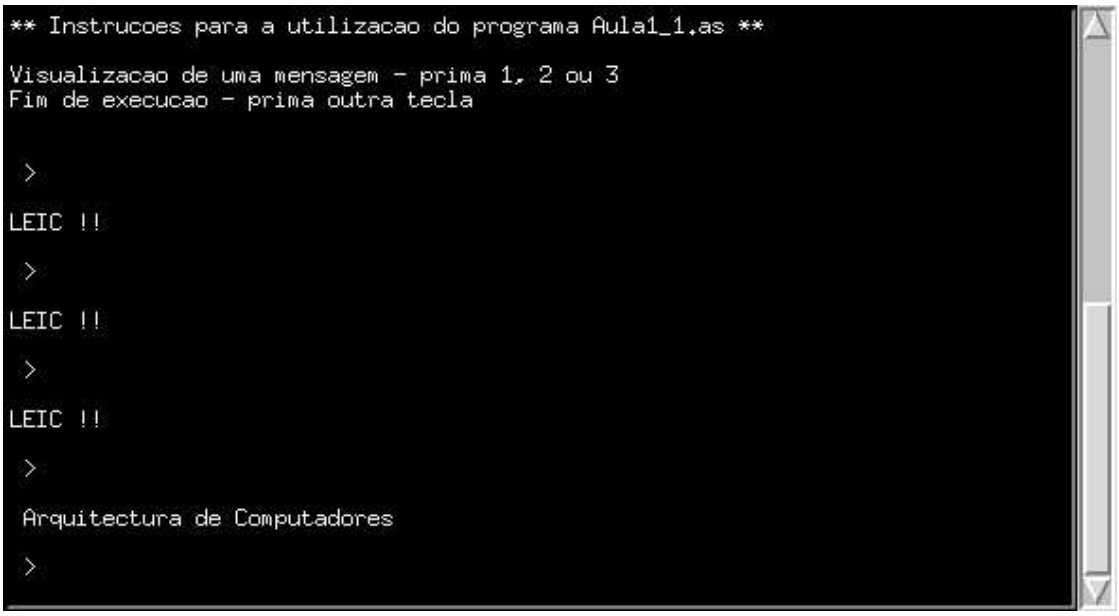
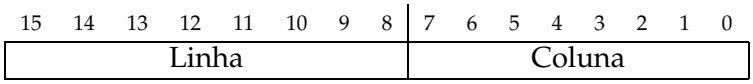


Figura 3: Janela de interface de texto, com 24 linhas e 80 colunas.

porto de estado, endereço FFFDh: porto que permite testar se existe ou não algum caracter para ler na janela de texto. Caso não haja, uma leitura deste porto retorna 0. Caso entretanto tenha sido premida uma tecla, este porto retorna 1. Assim que esta tecla for lida através do porto de leitura, este porto passa novamente a retornar 0.

porto de controlo, endereço FFFCh: porto que permite posicionar o cursor na janela de texto, indicando onde será escrito o próximo caracter. Para tornar possível este posicionamento, tem que ser feita a sua inicialização, conseguida através da escrita do valor FFFFh para este porto[1]. Uma vez inicializado, o cursor pode ser posicionado numa dada linha e coluna escrevendo para este porto um valor em que os 8 bits mais significativos indicam a linha (entre 0 e 23) e os 8 menos significativos a coluna (entre 1 e 80):



4.6.2 Janela Interface

A Figura 4 apresenta a janela de interface que disponibiliza 4 *displays* de 7 segmentos, 8 LEDs multicolores (verde, vermelho e amarelo) e 8 interruptores. Estes dispositivos podem ser acedidos da seguinte forma:

- interruptores, endereço FFF9h: uma leitura deste endereço permite ler em simultâneo o estado do conjunto dos 8 interruptores. A cada interruptor corresponde um bit, correspondendo ao interruptor da direita o bit menos significativo e ao da esquerda o oitavo bit (os oito bits mais significativos vêm sempre a 0). Um interruptor para baixo coloca o bit respectivo a 0 e para cima a 1.

[1] Um efeito secundário desta inicialização é limpar todo o conteúdo da janela.

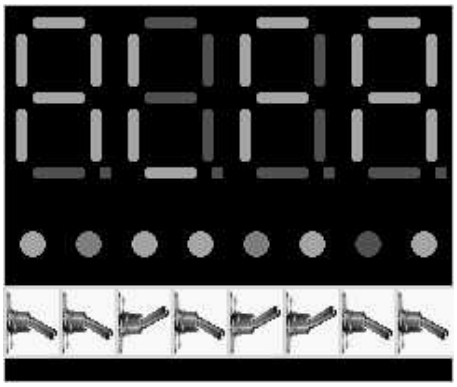


Figura 4: Janela de interface com *displays* de 7 segmentos, LEDs multicolores e interruptores.

- LEDs, endereço `FFF8h`: conjunto de 8 LEDs que podem assumir 4 estados (apagado, vermelho, verde ou amarelo) definidos por uma escrita para este endereço. A cada LED correspondem 2 bits, sendo o LED da direita controlado pelos dois bits menos significativos e os restantes LEDs por cada 2 bits por ordem. A combinação destes dois bits define a cor do LED respectivo:

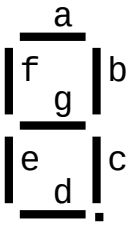
  00 apagado

  01 vermelho

  10 verde

  11 amarelo

- *display* de 7 segmentos, endereços `FFF0`, `FFF1h`, `FFF2h` e `FFF3h`: cada um destes portos de escrita controla, da direita para a esquerda, um conjunto de 8 LEDs que formam um *display*, 7 segmentos para o caracter, mais um LED para o ponto. Estes 8 LEDs são definidos pelos 8 bits menos significativos da palavra escrita para esse porto, com a seguinte correspondência:

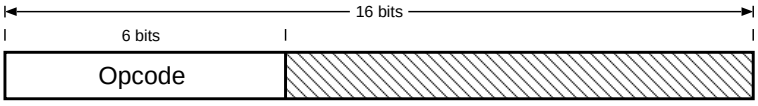


15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Não usados								.	g	f	e	d	c	b	a

A Formatos das Instruções Assembly

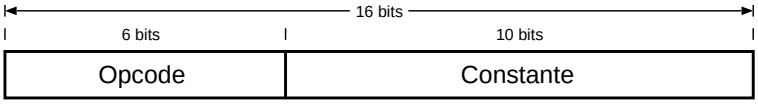
Instruções de 0 operandos

`NOP`, `ENI`, `DSI`, `STC`, `CLC`, `CMC`, `RET` e `RTI`



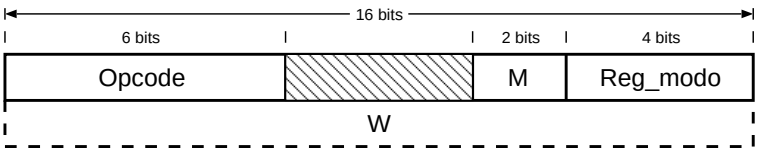
Instruções de 0 operandos com constante

`RETN` e `INT`



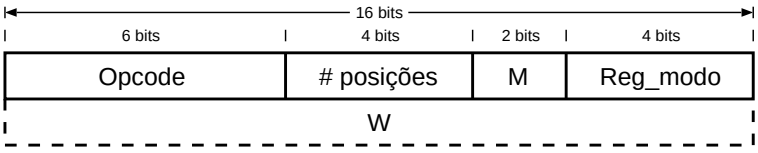
Instruções de 1 operando

`NEG`, `INC`, `DEC`, `COM`, `PUSH` e `POP`



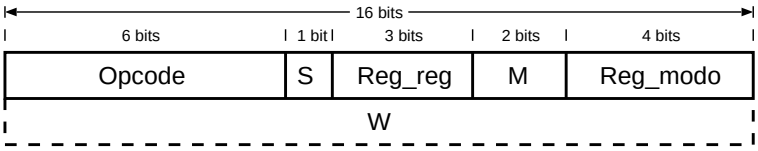
Instruções de 1 operando com constante

`SHR`, `SHL`, `SHRA`, `SHLA`, `ROR`, `ROL`, `RORC`, `ROLC`



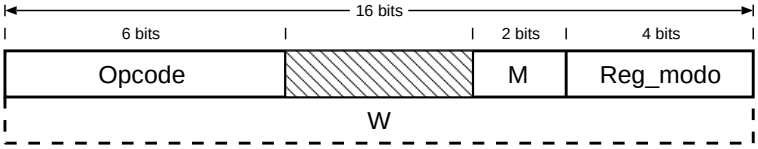
Instruções de 2 operandos

`CMP`, `ADD`, `ADDC`, `SUB`, `SUBB`, `MUL`, `DIV`, `TEST`, `AND`, `OR`, `XOR`, `MOV`, `MVBL`, `MVBH` e `XCH`



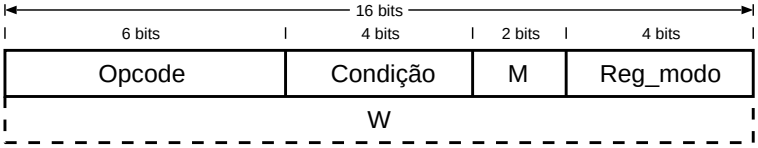
Instruções de salto absoluto incondicional

`JMP`, `CALL`



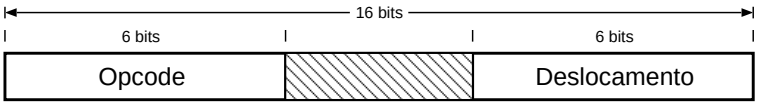
Instruções de salto absoluto condicional

`JMP`.cond, `CALL.`cond



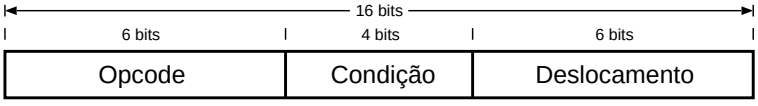
Instrução de salto relativo incondicional

`BR`



Instrução de salto relativo incondicional

`BR.`cond



Codificação da condição de salto

Condição	Mnemónica	Código
Zero	Z	0000
Não zero	NZ	0001
Transporte	C	0010
Não transporte	NC	0011
Negativo	N	0100
Não negativo	NN	0101
Excesso	O	0110
Não excesso	NO	0111
Positivo	P	1000
Não positivo	NP	1001
Interrupção	I	1010
Não interrupção	NI	1011

Códigos de Operação

Mnemónica	Código	Mnemónica	Código
NOP	000000	CMP	100000
ENI	000001	ADD	100001
DSI	000010	ADDC	100010
STC	000011	SUB	100011
CLC	000100	SUBB	100100
CMC	000101	MUL	100101
RET	000110	DIV	100110
RTI	000111	TEST	100111
INT	001000	AND	101000
RETN	001001	OR	101001
NEG	010000	XOR	101010
INC	010001	MOV	101011
DEC	010010	MVBH	101100
COM	010011	MVBL	101101
PUSH	010100	XCH	101110
POP	010101	JMP	110000
SHR	011000	JMP.cond	110001
SHL	011001	CALL	110010
SHRA	011010	CALL.cond	110011
SHLA	011011	BR	111000
ROR	011100	BR.cond	111001
ROL	011101		
RORC	011110		
ROLC	011111		

Modos de Endereçamento

M	Endereçamento	Operação
00	Por registo	`op = RX`
01	Por registo indirecto	`op = M[RX]`
10	Imediato	`op = W`
11	Indexado, directo, relativo ou baseado	`op = M[RX+W]`

Selecção do operando com o modo de endereçamento

S	Operando
0	Destino
1	Origem

B Conteúdo das ROMs de Controlo

Em apêndice, inclui-se a listagem do conteúdo das ROMs da Unidade de Controlo do processador P3. Estas ROMs podem ser modificadas conforme descrito na Secção 4.5 de forma a acrescentar uma instrução *assembly* ou a alterar o comportamento de uma já existente.

ROM B

Endereco	Conteudo	Modo
0 - 0000	0x00a	F1R0
1 - 0001	0x00b	F1RI0
2 - 0010	0x00d	F1IM0
3 - 0011	0x00f	F1IN0
4 - 0100	0x02d	WBR0
5 - 0101	0x02f	WBM0
6 - 0110	0x02d	WBR0
7 - 0111	0x02f	WBM0
8 - 1000	0x013	F2R0
9 - 1001	0x017	F2RI0
10 - 1010	0x01d	F2IM0
11 - 1011	0x023	F2IN0
12 - 1100	0x015	F2RS0
13 - 1101	0x01a	F2RIS0
14 - 1110	0x020	F2IMS0
15 - 1111	0x028	F2INS0

ROM A

Endereco	Conteudo	Instrucao
0 - 000000	0x032	NOP
1 - 000001	0x033	ENI0
2 - 000010	0x037	DSI0
3 - 000011	0x03b	STC0
4 - 000100	0x03e	CLC0
5 - 000101	0x040	CMC0
6 - 000110	0x044	RET0
7 - 000111	0x047	RTI0
8 - 001000	0x04c	INT0
9 - 001001	0x055	RETN0
10 - 001010	0x0	Livre
11 - 001011	0x0	Livre
12 - 001100	0x0	Livre
13 - 001101	0x0	Livre
14 - 001110	0x0	Livre
15 - 001111	0x0	Livre
16 - 010000	0x05b	NEG0
17 - 010001	0x05e	INC0
18 - 010010	0x060	DEC0
19 - 010011	0x062	COM0
20 - 010100	0x064	PUSH0
21 - 010101	0x067	POP0
22 - 010110	0x0	Livre
23 - 010111	0x0	Livre
24 - 011000	0x06a	SHR0
25 - 011001	0x071	SHL0
26 - 011010	0x078	SHRA0
27 - 011011	0x07f	SHLA0
28 - 011100	0x08c	ROR0
29 - 011101	0x093	ROL0
30 - 011110	0x09a	RORC0
31 - 011111	0x0a1	ROLC0
32 - 100000	0x0c2	CMP0
33 - 100001	0x0b4	ADD0
34 - 100010	0x0b6	ADDC0
35 - 100011	0x0b8	SUB0
36 - 100100	0x0ba	SUBB0
37 - 100101	0x0cf	MUL0
38 - 100110	0x0dd	DIV0
39 - 100111	0x0c4	TEST0
40 - 101000	0x0bc	AND0
41 - 101001	0x0be	OR0
42 - 101010	0x0c0	XOR0
43 - 101011	0x0a8	MOV0
44 - 101100	0x0af	MVBH0
45 - 101101	0x0aa	MVBL0
46 - 101110	0x0ca	XCH0
47 - 101111	0x0	Livre
48 - 110000	0x102	JMP.C0
49 - 110001	0x105	JMP
50 - 110010	0x109	CALL.C0
51 - 110011	0x10d	CALL
52 - 110100	0x0	Livre
53 - 110101	0x0	Livre
54 - 110110	0x0	Livre
55 - 110111	0x0	Livre
56 - 111000	0x0f9	BR.C0
57 - 111001	0x0f8	BR
58 - 111010	0x0	Livre
59 - 111011	0x0	Livre
60 - 111100	0x0	Livre
61 - 111101	0x0	Livre
62 - 111110	0x0	Livre
63 - 111111	0x0	Livre

ROM de Controlo

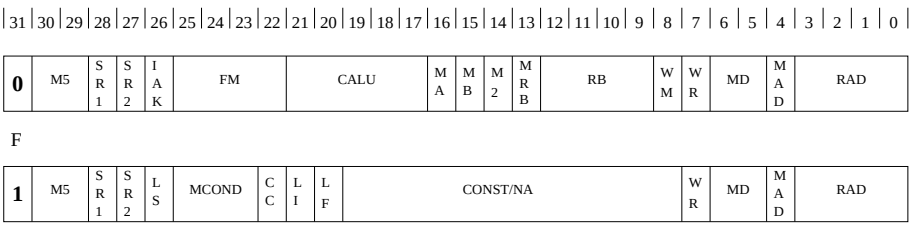


Figura 5: Formato das micro-instruções.

Endereco	Conteudo	Etiqueta	Operacao
000 000h 000000000b	0x8060001f	IF0	IR<-M[PC]
001 001h 000000001b	0x400a009f	IF1	PC<-PC+1, CAR<-ROMA[OP]
002 002h 000000010b	0x81c000d8	IH0	R8<-RE, !EINT?CAR<-IF0
003 003h 000000011b	0x0008319e	IH1	M[SP]<-R8, SP<-SP-1
004 004h 000000100b	0x04083f9e	IH2	M[SP]<-PC, SP<-SP-1, IAK<-1
005 005h 000000101b	0x000000b9	IH3	R9<-INTADDR
006 006h 000000110b	0x804200f8	IH4	R8<-0200h
007 007h 000000111b	0x00023099	IH5	R9<-R9-R8
008 008h 000001000b	0x000132bf	IH6	PC<-M[R9]
009 009h 000001001b	0x80100010	IH7	RE<-R0, CAR<-IF0
010 00ah 000001010b	0x2031009d	F1R0	RD<-R[IR1], CAR<-SBR
011 00bh 000001011b	0x0031009c	F1RI0	EA<-R[IR1]
012 00ch 000001100b	0x200138bd	F1RI1	RD<-M[EA], CAR<-SBR
013 00dh 000001101b	0x00013ebd	F1IM0	RD<-M[PC]
014 00eh 000001110b	0x200a009f	F1IM1	PC<-PC+1, CAR<-SBR
015 00fh 000001111b	0x00013ebc	F1IN0	EA<-M[PC]
016 010h 000010000b	0x000a009f	F1IN1	PC<-PC+1
017 011h 000010001b	0x0000009c	F1IN2	EA<-EA+R[IR1]
018 012h 000010010b	0x200138bd	F1IN3	RD<-M[EA], CAR<-SBR
019 013h 000010011b	0x0031009d	F2R0	RD<-R[IR1]
020 014h 000010100b	0x2031409b	F2R1	SD<-R[IR2], CAR<-SBR
021 015h 000010101b	0x0031009b	F2RS0	SD<-R[IR1]
022 016h 000010110b	0x2031409d	F2RS1	RD<-R[IR2], CAR<-SBR
023 017h 000010111b	0x0031009c	F2RI0	EA<-R[IR1]
024 018h 000011000b	0x000138bd	F2RI1	RD<-M[EA]
025 019h 000011001b	0x2031409b	F2RI2	SD<-R[IR2], CAR<-SBR
026 01ah 000011010b	0x0031009c	F2RIS0	EA<-R[IR1]
027 01bh 000011011b	0x000138bb	F2RIS1	SD<-M[EA]
028 01ch 000011100b	0x2031409d	F2RIS2	RD<-R[IR2], CAR<-SBR
029 01dh 000011101b	0x00013ebd	F2IM0	RD<-M[PC]
030 01eh 000011110b	0x000a009f	F2IM1	PC<-PC+1
031 01fh 000011111b	0x2031409b	F2IM2	SD<-R[IR2], CAR<-SBR
032 020h 000100000b	0x00013ebb	F2IMS0	SD<-M[PC]
033 021h 000100001b	0x000a009f	F2IMS1	PC<-PC+1
034 022h 000100010b	0x2031409d	F2IMS2	RD<-R[IR2], CAR<-SBR
035 023h 000100011b	0x00013ebc	F2IN0	EA<-M[PC]
036 024h 000100100b	0x000a009f	F2IN1	PC<-PC+1
037 025h 000100101b	0x0000009c	F2IN2	EA<-EA+R[IR1]
038 026h 000100110b	0x000138bd	F2IN3	RD<-M[EA]
039 027h 000100111b	0x2031409b	F2IN4	SD<-R[IR2], CAR<-SBR
040 028h 000101000b	0x00013ebc	F2INS0	EA<-M[PC]
041 029h 000101001b	0x000a009f	F2INS1	PC<-PC+1
042 02ah 000101010b	0x0000009c	F2INS2	EA<-EA+R[IR1]
043 02bh 000101011b	0x000138bb	F2INS3	SD<-M[EA]
044 02ch 000101100b	0x2031409d	F2INS4	RD<-R[IR2], CAR<-SBR
045 02dh 000101101b	0x00313a80	WBR0	R[WBR]<-RD

```
046 02eh 000101110b     0x80000200     WBR1      CAR<-IH0
047 02fh 000101111b     0x83002d00     WBM0      S?CAR<-WBR0 (modo no outro)
048 030h 000110000b     0x00003b1c     WBM1      M[EA]<-RD
049 031h 000110001b     0x80000200     WBM2      CAR<-IH0
050 032h 000110010b     0x80000200     NOP0      CAR<-IH0
051 033h 000110011b     0x804010f8     ENI0      R8<-0010h
052 034h 000110100b     0x000000d9     ENI1      R9<-RE
053 035h 000110101b     0x00143298     ENI2      R8<-R8 or R9
054 036h 000110110b     0x80100218     ENI3      RE<-R8, CAR<-IH0
055 037h 000110111b     0x80400ff8     DSI0      R8<-000fh
056 038h 000111000b     0x000000d9     DSI1      R9<-RE
057 039h 000111001b     0x00123298     DSI2      R8<-R8 and R9
058 03ah 000111010b     0x80100218     DSI3      RE<-R8, CAR<-IH0
059 03bh 000111011b     0x00112098     STC0      R8<-not R0
060 03ch 000111100b     0x010a0018     STC1      R8+1, flag C
061 03dh 000111101b     0x80000200     STC2      CAR<-IH0
062 03eh 000111110b     0x01002010     CLC0      R0+R0, flag C
063 03fh 000111111b     0x80000200     CLC1      CAR<-IH0
064 040h 001000000b     0x804004f8     CMC0      R8<-0004
065 041h 001000001b     0x000000d9     CMC1      R9<-RE
066 042h 001000010b     0x00163298     CMC2      R8<-R8 exor R9
067 043h 001000011b     0x80100218     CMC3      RE<-R8, CAR<-IH0
068 044h 001000100b     0x000a009e     RET0      SP<-SP+1
069 045h 001000101b     0x00013cbf     RET1      PC<-M[SP]
070 046h 001000110b     0x80000200     RET2      CAR<-IH0
071 047h 001000111b     0x000a009e     RTI0      SP<-SP+1
072 048h 001001000b     0x00013cbf     RTI1      PC<-M[SP]
073 049h 001001001b     0x000a009e     RTI2      SP<-SP+1
074 04ah 001001010b     0x00013cb8     RTI3      R8<-M[SP]
075 04bh 001001011b     0x80100218     RTI4      RE<-R8, CAR<-IH0
076 04ch 001001100b     0x000000d8     INT0      R8<-RE
077 04dh 001001101b     0x0008319e     INT1      M[SP]<-R8, SP<-SP-1
078 04eh 001001110b     0x00083f9e     INT2      M[SP]<-PC, SP<-SP-1
079 04fh 001001111b     0x8040fff8     INT3      R8<-00ffh
080 050h 001010000b     0x00128098     INT4      R8<-IR and R8
081 051h 001010001b     0x804200f9     INT5      R9<-0200h
082 052h 001010010b     0x00023298     INT6      R8<-R8-R9
083 053h 001010011b     0x000130bf     INT7      PC<-M[R8]
084 054h 001010100b     0x80100010     INT8      RE<-R0, CAR<-IF0
085 055h 001010101b     0x000a009e     RETN0     SP<-SP+1
086 056h 001010110b     0x00013cbf     RETN1     PC<-M[SP]
087 057h 001010111b     0x8043fff8     RETN2     R8<-03ffh
088 058h 001011000b     0x00128098     RETN3     R8<-IR and R8
089 059h 001011001b     0x0000309e     RETN4     SP<-SP+R8
090 05ah 001011010b     0x80000200     RETN5     CAR<-IH0
091 05bh 001011011b     0xe40000f8     NEG0      R8<-0, SBR<-CAR+1, CAR<-F1
092 05ch 001011100b     0x03c23a98     NEG1      R8<-R8-RD, flags ZCNO
093 05dh 001011101b     0x7031309d     NEG2      RD<-R8, CAR<-WB
094 05eh 001011110b     0xe4000000     INC0      SBR<-CAR+1, CAR<-F1
095 05fh 001011111b     0x73ca009d     INC1      RD<-RD+1, flags ZCNO, CAR<-WB
096 060h 001100000b     0xe4000000     DEC0      SBR<-CAR+1, CAR<-F1
097 061h 001100001b     0x73c8009d     DEC1      RD<-RD-1, flags ZCNO, CAR<-WB
098 062h 001100010b     0xe4000000     COM0      SBR<-CAR+1, CAR<-F1
099 063h 001100011b     0x7290009d     COM1      RD<-!RD, flags ZN, CAR<-WB
100 064h 001100100b     0xe4000000     PUSH0     SBR<-CAR+1, CAR<-F1
101 065h 001100101b     0x00083b9e     PUSH1     M[SP]<-RD, SP<-SP-1
102 066h 001100110b     0x80000200     PUSH2     CAR<-IH0
103 067h 001100111b     0xe4000000     POP0      SBR<-CAR+1, CAR<-F1
104 068h 001101000b     0x000a009e     POP1      SP<-SP+1
105 069h 001101001b     0x70013cbd     POP2      RD<-M[SP], CAR<-WB
106 06ah 001101010b     0xe403c0f8     SHR0      R8<-03c0h, SBR<-CAR+1, CAR<-F1
107 06bh 001101011b     0x00128098     SHR1      R8<-R8 and IR
108 06ch 001101100b     0x804040f9     SHR2      R9<-0040h
109 06dh 001101101b     0x03a0009d     SHR3      RD<-shr RD, flags ZCN
110 06eh 001101110b     0x00023298     SHR4      R8<-R8-R9
111 06fh 001101111b     0x80c06d00     SHR5      !z?CAR<-SHR3
```

```
112 070h 001110000b     0x70000000     SHR6      CAR<-WB
113 071h 001110001b     0xe403c0f8     SHL0      R8<-03c0h, SBR<-CAR+1, CAR<-F1
114 072h 001110010b     0x00128098     SHL1      R8<-R8 and IR
115 073h 001110011b     0x804040f9     SHL2      R9<-0040h
116 074h 001110100b     0x03a2009d     SHL3      RD<-shl RD, flags ZCN
117 075h 001110101b     0x00023298     SHL4      R8<-R8-R9
118 076h 001110110b     0x80c07400     SHL5      !z?CAR<-SHL3
119 077h 001110111b     0x70000000     SHL6      CAR<-WB
120 078h 001111000b     0xe403c0f8     SHRA0     R8<-03c0h, SBR<-CAR+1, CAR<-F1
121 079h 001111001b     0x00128098     SHRA1     R8<-R8 and IR
122 07ah 001111010b     0x804040f9     SHRA2     R9<-0040h
123 07bh 001111011b     0x03e4009d     SHRA3     RD<-shra RD, flags ZCNO
124 07ch 001111100b     0x00023298     SHRA4     R8<-R8-R9
125 07dh 001111101b     0x80c07b00     SHRA5     !z?CAR<-SHRA3
126 07eh 001111110b     0x70000000     SHRA6     CAR<-WB
127 07fh 001111111b     0xe403c0f8     SHLA0     R8<-03c0h, SBR<-CAR+1, CAR<-F1
128 080h 010000000b     0x00128098     SHLA1     R8<-R8 and IR
129 081h 010000001b     0x0031209a     SHLA2     R10<-R0
130 082h 010000010b     0x03e6009d     SHLA3     RD<-shla RD, flags ZCNO
131 083h 010000011b     0x000000d9     SHLA4     R9<-RE
132 084h 010000100b     0x0014329a     SHLA5     R10<-R10 or R9
133 085h 010000101b     0x804040f9     SHLA6     R9<-0040h
134 086h 010000110b     0x00023298     SHLA7     R8<-R8-R9
135 087h 010000111b     0x80c082d9     SHLA8     R9<-RE, !z?CAR<-SHLA3
136 088h 010001000b     0x804001f8     SHLA9     R8<-1
137 089h 010001001b     0x0012309a     SHLA10    R10<-R10 and R8
138 08ah 010001010b     0x0014329a     SHLA11    R10<-R10 or R9
139 08bh 010001011b     0xf010001a     SHLA12    RE<-R10, CAR<-WB
140 08ch 010001100b     0xe403c0f8     ROR0      R8<-03c0h, SBR<-CAR+1, CAR<-F1
141 08dh 010001101b     0x00128098     ROR1      R8<-R8 and IR
142 08eh 010001110b     0x804040f9     ROR2      R9<-0040h
143 08fh 010001111b     0x03a8009d     ROR3      RD<-ror RD, flags ZCN
144 090h 010010000b     0x00023298     ROR4      R8<-R8-R9
145 091h 010010001b     0x80c08f00     ROR5      !z?CAR<-ROR3
146 092h 010010010b     0x70000000     ROR6      CAR<-WB
147 093h 010010011b     0xe403c0f8     ROL0      R8<-03c0h, SBR<-CAR+1, CAR<-F1
148 094h 010010100b     0x00128098     ROL1      R8<-R8 and IR
149 095h 010010101b     0x804040f9     ROL2      R9<-0040h
150 096h 010010110b     0x03aa009d     ROL3      RD<-rol RD, flags ZCN
151 097h 010010111b     0x00023298     ROL4      R8<-R8-R9
152 098h 010011000b     0x80c09600     ROL5      !z?CAR<-ROL3
153 099h 010011001b     0x70000000     ROL6      CAR<-WB
154 09ah 010011010b     0xe403c0f8     RORC0     R8<-03c0h, SBR<-CAR+1, CAR<-F1
155 09bh 010011011b     0x00128098     RORC1     R8<-R8 and IR
156 09ch 010011100b     0x804040f9     RORC2     R9<-0040h
157 09dh 010011101b     0x03ac009d     RORC3     RD<-rorc RD, flags ZCN
158 09eh 010011110b     0x00023298     RORC4     R8<-R8-R9
159 09fh 010011111b     0x80c09d00     RORC5     !z?CAR<-RORC3
160 0a0h 010100000b     0x70000000     RORC6     CAR<-WB
161 0a1h 010100001b     0xe403c0f8     ROLC0     R8<-03c0h, SBR<-CAR+1, CAR<-F1
162 0a2h 010100010b     0x00128098     ROLC1     R8<-R8 and IR
163 0a3h 010100011b     0x804040f9     ROLC2     R9<-0040h
164 0a4h 010100100b     0x03ae009d     ROLC3     RD<-rolc RD, flags ZCN
165 0a5h 010100101b     0x00023298     ROLC4     R8<-R8-R9
166 0a6h 010100110b     0x80c0a400     ROLC5     !z?CAR<-ROLC3
167 0a7h 010100111b     0x70000000     ROLC6     CAR<-WB
168 0a8h 010101000b     0xec000000     MOV0      SBR<-CAR+1, CAR<-F2
169 0a9h 010101001b     0x7031369d     MOV1      RD<-SD, CAR<-WB
170 0aah 010101010b     0xec00fff8     MVBL0     R8<-00ffh, SBR<-CAR+1, CAR<-F2
171 0abh 010101011b     0x00113099     MVBL1     R9<-!R8
172 0ach 010101100b     0x0012329d     MVBL2     RD<-RD and R9
173 0adh 010101101b     0x00123698     MVBL3     R8<-R8 and SD
174 0aeh 010101110b     0x7014309d     MVBL4     RD<-RD or R8, CAR<-WB
175 0afh 010101111b     0xec00fff8     MVBH0     R8<-00ffh, SBR<-CAR+1, CAR<-F2
176 0b0h 010110000b     0x00113099     MVBH1     R9<-!R8
177 0b1h 010110001b     0x0012309d     MVBH2     RD<-RD and R8
```

```
178 0b2h 010110010b     0x00123699     MVBH3     R9<-R9 and SD
179 0b3h 010110011b     0x7014329d     MVBH4     RD<-RD or R9, CAR<-WB
180 0b4h 010110100b     0xec000000     ADD0      SBR<-CAR+1, CAR<-F2
181 0b5h 010110101b     0x73c0369d     ADD1      RD<-RD+SD, flags ZCNO, CAR<-WB
182 0b6h 010110110b     0xec000000     ADDC0     SBR<-CAR+1, CAR<-F2
183 0b7h 010110111b     0x73c4369d     ADDC1     RD<-RD+SD+C, flags ZCNO, CAR<-WB
184 0b8h 010111000b     0xec000000     SUB0      SBR<-CAR+1, CAR<-F2
185 0b9h 010111001b     0x73c2369d     SUB1      RD<-RD-SD, flags ZCNO, CAR<-WB
186 0bah 010111010b     0xec000000     SUBB0     SBR<-CAR+1, CAR<-F2
187 0bbh 010111011b     0x73c6369d     SUBB1     RD<-RD-SD-C, flags ZCNO, CAR<-WB
188 0bch 010111100b     0xec000000     AND0      SBR<-CAR+1, CAR<-F2
189 0bdh 010111101b     0x7292369d     AND1      RD<-RD and SD, flags ZN, CAR<-WB
190 0beh 010111110b     0xec000000     OR0       SBR<-CAR+1, CAR<-F2
191 0bfh 010111111b     0x7294369d     OR1       RD<-RD or SD, flags ZN, CAR<-WB
192 0c0h 011000000b     0xec000000     XOR0      SBR<-CAR+1, CAR<-F2
193 0c1h 011000001b     0x7296369d     XOR1      RD<-RD xor SD, flags ZN, CAR<-WB
194 0c2h 011000010b     0xec000000     CMP0      SBR<-CAR+1, CAR<-F2
195 0c3h 011000011b     0x73c2361d     CMP1      RD<-RD-SD, flags ZCNO, CAR<-WB
196 0c4h 011000100b     0xec000000     TEST0     SBR<-CAR+1, CAR<-F2
197 0c5h 011000101b     0x7292361d     TEST1     RD<-RD and SD, flags ZN, CAR<-WB
198 0c6h 011000110b     0x8340c900     WSD0      !S?CAR<-WSD3       (mode on RD)
199 0c7h 011000111b     0x8240c900     WSD1      !M0?CAR<-WSD3 (mode REG or IMM)
200 0c8h 011001000b     0x7000371c     WSD2      M[EA]<-SD, CAR<-WB   (mode MEM)
201 0c9h 011001001b     0x70317680     WSD3      R[!WBR]<-SD, CAR<-WB (mode REG)
202 0cah 011001010b     0xec000000     XCH0      SBR<-CAR+1, CAR<-F2
203 0cbh 011001011b     0x00313a98     XCH1      R8<-RD
204 0cch 011001100b     0x0031369d     XCH2      RD<-SD
205 0cdh 011001101b     0x0031309b     XCH3      SD<-R8
206 0ceh 011001110b     0x8000c600     XCH4      CAR<-WSD0
207 0cfh 011001111b     0xec0010f8     MUL0      R8<-16, SBR<-CAR+1, CAR<-F2
208 0d0h 011010000b     0x000000da     MUL1      R10<-RE
209 0d1h 011010001b     0x0013b09a     MUL2      R10<-R10 and R8   (flag E)
210 0d2h 011010010b     0x00313a99     MUL3      R9<-RD
211 0d3h 011010011b     0x01f1209d     MUL4      RD<-R0, flags CNO (clear flags)
212 0d4h 011010100b     0x002c009b     MUL5      SD<-rorc SD
213 0d5h 011010101b     0x8150d71a     MUL6      RE<-R10, !c?CAR<-MUL8
214 0d6h 011010110b     0x0100329d     MUL7      RD<-RD+R9, flag C
215 0d7h 011010111b     0x012c009d     MUL8      RD<-rorc RD, flag C
216 0d8h 011011000b     0x00080098     MUL9      R8<-R8-1
217 0d9h 011011001b     0x80c0d400     MUL10     !z?CAR<-MUL5
218 0dah 011011010b     0x012c009b     MUL11     SD<-rorc SD, flag C (C=0)
219 0dbh 011011011b     0x0200361d     MUL12     RD+SD, flag Z
220 0dch 011011100b     0x8000c600     MUL13     CAR<-WSD0
221 0ddh 011011101b     0xec0000d8     DIV0      R8<-RE, SBR<-CAR+1, CAR<-F2
222 0deh 011011110b     0x0000201b     DIV1      SD<-SD+R0
223 0dfh 011011111b     0x80c0e300     DIV2      !z?CAR<-DIV6
224 0e0h 011100000b     0x804001f9     DIV3      R9<-0001         (divisao por 0!)
225 0e1h 011100001b     0x00143298     DIV4      R8<-R8 or R9
226 0e2h 011100010b     0x80100218     DIV5      RE<-R8, CAR<-IH0           (O<-1)
227 0e3h 011100011b     0x01c12099     DIV6      R9<-R0+R0, flags CNO (clear flag)
228 0e4h 011100100b     0x0002361d     DIV7      RD-SD
229 0e5h 011100101b     0x8140f500     DIV8      !c?CAR<-DIV24          (result=0)
230 0e6h 011100110b     0x00312098     DIV9      R8<-R0
231 0e7h 011100111b     0x000a0098     DIV10     R8<-R8+1
232 0e8h 011101000b     0x0122009b     DIV11     SD<-shl SD, flag C
233 0e9h 011101001b     0x8100ec00     DIV12     c?CAR<-DIV15
234 0eah 011101010b     0x0002361d     DIV13     RD-SD
235 0ebh 011101011b     0x8100e700     DIV14     c?CAR<-DIV10
236 0ech 011101100b     0x002c009b     DIV15     SD<-rorc SD
237 0edh 011101101b     0x0102369d     DIV16     RD<-RD-SD, flag C
238 0eeh 011101110b     0x8100f100     DIV17     c?CAR<-DIV20
239 0efh 011101111b     0x0000369d     DIV18     RD<-RD+SD              (<0:repoe)
240 0f0h 011110000b     0x01300010     DIV19     R0, flag C             (C<-0)
241 0f1h 011110001b     0x002e0099     DIV20     R9<-rolc R9
242 0f2h 011110010b     0x0020009b     DIV21     SD<-shr SD
243 0f3h 011110011b     0x00080098     DIV22     R8<-R8-1
```

```
244 0f4h 011110100b     0x80c0ed00     DIV23     !z?CAR<-DIV16
245 0f5h 011110101b     0x00313a9b     DIV24     SD<-RD
246 0f6h 011110110b     0x0331329d     DIV25     RD<-R9, flags ZC
247 0f7h 011110111b     0x8000c600     DIV26     CAR<-WSD0
248 0f8h 011111000b     0x83c00200     BR.C0     !COND?CAR<-IH0
249 0f9h 011111001b     0x80403ff8     BR0       R8<-003fh
250 0fah 011111010b     0x0013b099     BR1       R9<-R8 and RI
251 0fbh 011111011b     0x804020fa     BR2       R10<-0020h       (teste do sinal)
252 0fch 011111100b     0x0012329a     BR3       R10<-R10 and R9
253 0fdh 011111101b     0x80810000     BR4       z?CAR<-BR7
254 0feh 011111110b     0x00100098     BR5       R8<-not R8
255 0ffh 011111111b     0x00143099     BR6       R9<-R9 or R8
256 100h 100000000b     0x0000329f     BR7       PC<-PC+R9
257 101h 100000001b     0x80000200     BR8       CAR<-IH0
258 102h 100000010b     0xe4000000     JMP0      SBR<-CAR+1, CAR<-F1
259 103h 100000011b     0x00313a9f     JMP1      PC<-RD
260 104h 100000100b     0x80000200     JMP2      CAR<-IH0
261 105h 100000101b     0xe4000000     JMP.C0    SBR<-CAR+1, CAR<-F1
262 106h 100000110b     0x83c00200     JMP.C1    !COND?CAR<-IH0
263 107h 100000111b     0x00313a9f     JMP.C2    PC<-RD
264 108h 100001000b     0x80000200     JMP.C3    CAR<-IH0
265 109h 100001001b     0xe4000000     CALL0     SBR<-CAR+1, CAR<-F1
266 10ah 100001010b     0x00083f9e     CALL1     M[SP]<-PC, SP<-SP-1
267 10bh 100001011b     0x00313a9f     CALL2     PC<-RD
268 10ch 100001100b     0x80000200     CALL3     CAR<-IH0
269 10dh 100001101b     0xe4000000     CALL.C0   SBR<-CAR+1, CAR<-F1
270 10eh 100001110b     0x83c00200     CALL.C1   !COND?CAR<-IH0
271 10fh 100001111b     0x00083f9e     CALL.C2   M[SP]<-PC, SP<-SP-1
272 110h 100010000b     0x00313a9f     CALL.C3   PC<-RD
273 111h 100010001b     0x80000200     CALL.C4   CAR<-IH0

(livre do endereco 274 ao 511)
```