ANEXO de Arquitectura de Computadores

Informação sobre o Processador P3

Abril 2005

POR FAVOR:
- **não escreva ou danifique este anexo**
- **devolva-o no final do exame**

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 controlo. Este registo está na posição 15 do banco de registos (R15).

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]`. Este registo está na posição 14 do banco de registos (R14).

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.

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.

·	·	·	·	·	·	·	·	·	·	·	E	Z	C	N	O

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.

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.

Anexo AC

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.

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 fornece 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.
- botões de pressão: conjunto de 15 interruptores de pressão. A activação de cada um destes botões gera uma interrupção com o correspondente vector de interrupção.
- interruptores, endereço `FFF9h`: conjunto de 8 interruptores cujo estado pode ser obtido por leitura deste endereço.
- LEDs, endereço `FFF8h`: cada bit da palavra escrita neste porto define quais dos 16 LEDs estão ligados.
- *display* de 7 segmentos, endereços `FFF0`, `FFF1h`, `FFF2h` e `FFF3h`: cada um destes portos de escrita controla um conjunto de 7 LEDs que formam um *display*.
- *display* de cristal líquido ou LCD: *display* de texto com 16 colunas e duas linhas. Tem 2 portos de escrita:
  - endereço `FFF5h`: porto que permite escrever um dado caracter no *display*;
  - endereço `FFF4h`: porto que permite posicionar o cursor no *display*, posição esta onde será escrito o próximo caracter.
- máscara de interrupções, endereço `FFFAh`: posição de um filtro que permite seleccionar individualmente quais dos 16 primeiros vectores de interrupção (de 0 a 15) estão habilitados. Após um *reset*, todos os bits da máscara de interrupção estão a 0.
- temporizador: dispositivo que fornece a geração de uma interrupção ao fim de um intervalo de tempo real, especificado pelo utilizador. Tem 2 portos de interface:
  - controlo, endereço `FFF7h`: porto que permite arrancar (colocando o bit menos significativo a 1) ou parar (colocando esse bit a 0) o temporizador.
  - valor de contagem, endereço `FFF6h`: porto que permite indicar o número de intervalos de 100ms ao fim do qual o temporizador gerará uma interrupção.

Interrupções O simulador disponibiliza 15 botões para a geração de interrupções externas (para além destas, o simulador tem apenas mais uma fonte de interrupções, o temporizador). 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, são efectuados dois testes:

- o bit de estado `E` (*enable interrupts*) tem que estar activo.
- o bit da máscara de interrupções correspondente a este vector de interrupção tem que estar activo.

Caso estas duas condições se verifiquem, 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 posições 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]`.

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.

Conjunto de Instruções A tabela seguinte apresenta todas as instruções *assembly* do `P3`, juntamente com respectivo código de operação.

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

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
`I`, `NI`: 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.

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.

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 $+(2^{15} - 1)$.

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*.

Pseudo-Instruções

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`.

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.

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.

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`.

Instruções Assembly

As instruções *assembly* válidas para o 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: $op \leftarrow op - 1$, 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: $op \leftarrow op + 1$, 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, 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* $\leftarrow$ *op1 - op2*, subtrai a *op1* o valor de *op2*.

SUBB

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

Acção: *op1* $\leftarrow$ *op1 - op2 -* $\overline{C}$, igual a `SUB` excepto que subtrai mais um caso o bit de estado transporte esteja a 0.

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* $\leftarrow$ *op2, op2* $\leftarrow$ *op1*, troca os valores de `op1` e `op2`.

XOR

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

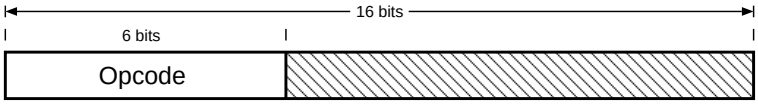
Acção: *op1* $\leftarrow$ *op1* $\oplus$ *op2*. Faz a operação lógica EXCLUSIVE-OR bit-a-bit dos dois operandos.

Anexo AC

Formatos das Instruções Assembly

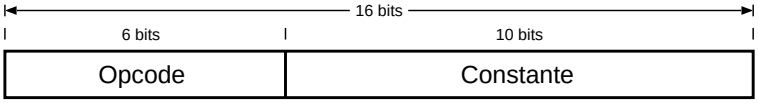
Instruções de 0 operandos

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



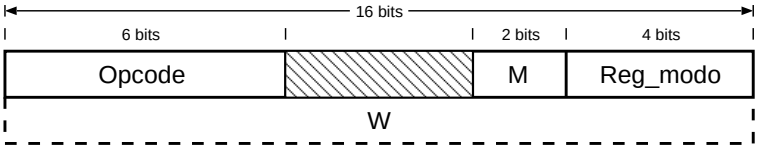
Instruções de 0 operandos com constante

`RETN` e `INT`



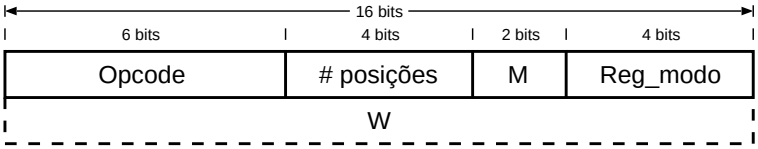
Instruções de 1 operando

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



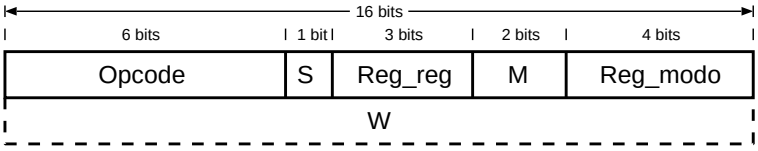
Instruções de 1 operando com constante

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



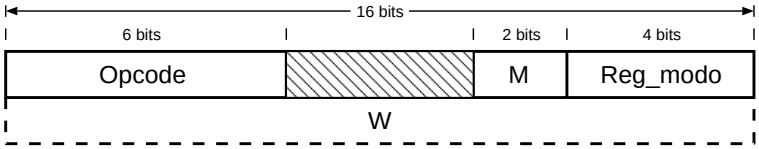
Instruções de 2 operandos

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



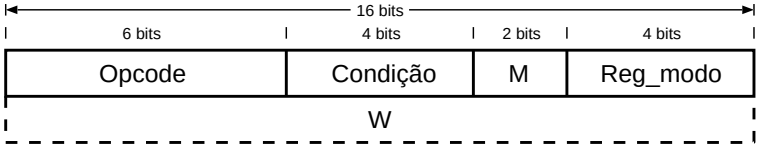
Instruções de salto absoluto incondicional

`JMP`, `CALL`



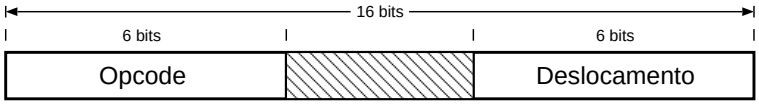
Instruções de salto absoluto condicional

`JMP`.cond, `CALL.`cond



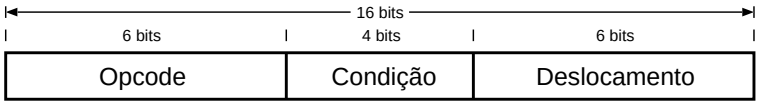
Instrução de salto relativo incondicional

`BR`



Instrução de salto relativo condicional

`BR.`cond



Codificação da condição de salto

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

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]`

Operando com o modo de endereçamento

S	Operando
0	Destino
1	Origem

Anexo AC

Tabela das micro-operações da ULA

$S_4S_3S_2S_1S_0$	Micro-Operação	
00000	$R \leftarrow A + B$	soma
00001	$R \leftarrow A - B$	subtracção
00010	$R \leftarrow A + B + C$	soma com bit transporte
00011	$R \leftarrow A - B - C$	subtracção com bit transporte
00100	$R \leftarrow A - 1$	decremento
00101	$R \leftarrow A + 1$	incremento
00110	$R \leftarrow A - \overline{C}$	decremento, se $C = 0$
00111	$R \leftarrow A + \overline{C}$	incremento, se $C = 0$
01-00	$R \leftarrow \overline{A}$	complemento
01-01	$R \leftarrow A \wedge B$	conjunção
01-10	$R \leftarrow A \vee B$	disjunção
01-11	$R \leftarrow A \oplus B$	ou exclusivo
10000	$R \leftarrow \text{SHR } A$	deslocamento lógico à direita
10001	$R \leftarrow \text{SHL } A$	deslocamento lógico à esquerda
10010	$R \leftarrow \text{SHRA } A$	deslocamento aritmético à direita
10011	$R \leftarrow \text{SHLA } A$	deslocamento aritmético à esquerda
10100	$R \leftarrow \text{ROR } A$	rotação à direita
10101	$R \leftarrow \text{ROL } A$	rotação à esquerda
10110	$R \leftarrow \text{RORC } A$	rotação à direita com transporte
10111	$R \leftarrow \text{ROLC } A$	rotação à esquerda com transporte
11- - -	$R \leftarrow A$	transferência

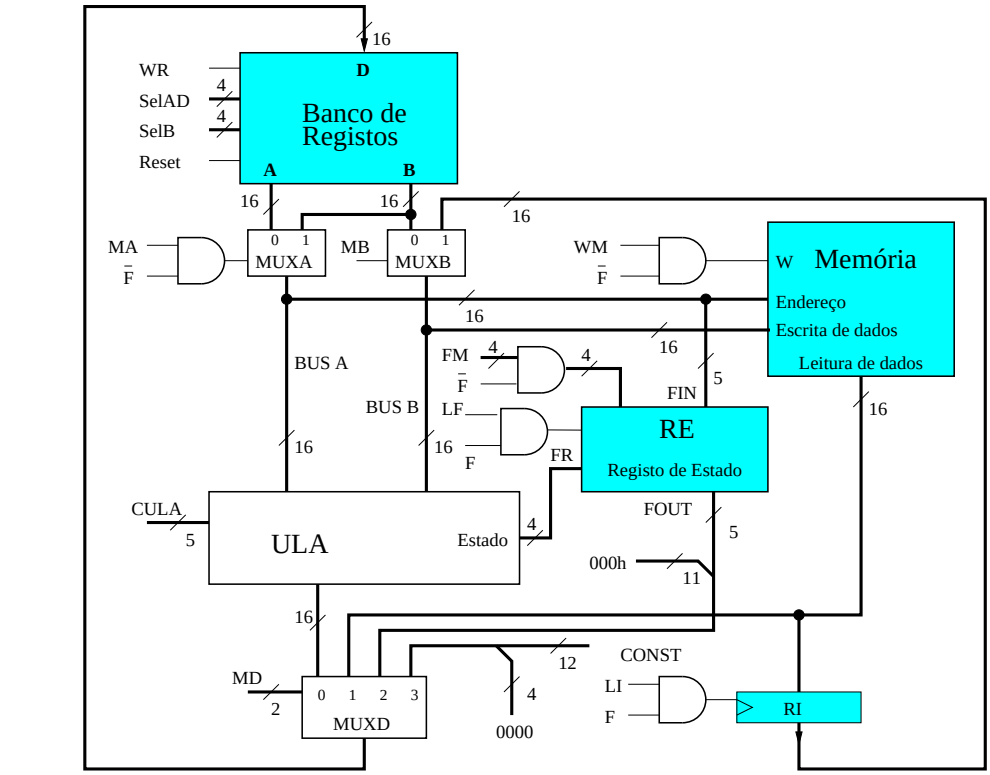
Banco de registos.

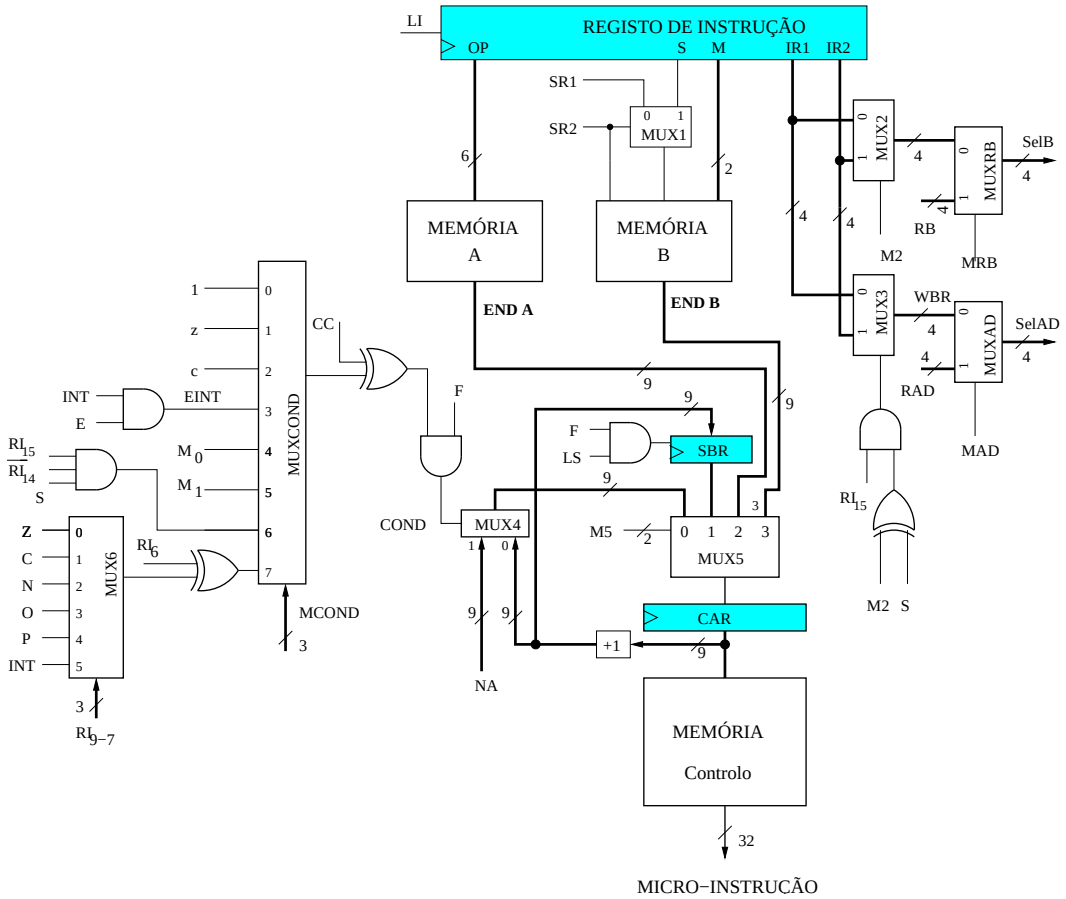
Registo	Descrição
`R0`	Constante 0
`R1` – `R7`	Registos de uso geral
`R8` – `R10`	Registos de uso restrito
`R11`	Operando (`SD`)
`R12`	Endereço destino (`EA`)
`R13`	Resultado (`RD`)
`R14`	Apontador da pilha (`SP`)
`R15`	Contador programa (`PC`)

Organização ROM B

SR2	SR1	S	Endereço seleccionado
0	0	-	Leitura de um operando
0	1	-	Escrita do resultado
1	-	0	Leitura de dois operandos (S=0)
1	-	1	Leitura de dois operandos (S=1)

Unidades de Processamento e Controlo





Formato das Micro-Instruções

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	M5			SR1		SR2		IAK		FM				CALU			MA		MB		M2		MRB		RB			WM		WR		MD		MAD		RAD	

F

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	M5			SR1		SR2		LS		MCOND			CC		LI		LF								CONST/NA					WR		MD		MAD		RAD	

Fluxograma de Execução de uma Instrução Assembly