Capítulo 10

O Processador Mico XII

Se algo tem rabo de jacaré, couro de jacaré, boca de jacaré, pé de jacaré, olho de jacaré, corpo de jacaré e cabeça de jacaré, como é que não é jacaré?

Leonel Brizola

O autor teve seu primeiro contato com os livros de Hennessy & Patterson em 1990, com [HP90], e vem lecionando com base nas várias edições do livro introdutório desde 1994 [PH94]. Evidentemente, o conteúdo deste capítulo, bem como o projeto do Mico XII, foram muito influenciados por aqueles textos e de suas edições posteriores. O autor é um adepto do método de estudar Arquitetura de Computadores, proposto por Hennessy & Patterson, segundo o qual a linguagem *assembly* é a especificação do processador, e isso se reflete na forma e no conteúdo deste capítulo. Referências explícitas e implícitas àqueles textos são inevitáveis após um contato, quase que diário, de três décadas.

O Mico XII é um processador extremamente simples porém completo. O conjunto de instruções do Mico permite executar programas que contenham funções e que façam acessos a uma memória de dados. Todos os processadores comerciais conhecidos pelo autor (8085, z80, Atmel, 8086, 68000, ARM, MIPS, PowerPC, SPARC) possuem instruções como as do Mico, além de muitas outras que acrescentam pouco quando o que nos interessa é compreender a funcionalidade de um processador. O processador chama-se Mico porque este é uma imitação que tenta macaquear o processador MIPS32. A versão apresentada aqui é o décimo segundo membro de uma linhagem iniciada em 1996.

Iniciamos o capítulo[1] com a programação em linguagem de máquina do Mico, e então prosseguimos com a descrição dos componentes do processador, e de como os componentes são interligados para implementar as instruções. O capítulo encerra com uma discussão sobre a metodologia de sincronização do processador.

10.1 Conjunto de Instruções do Mico XII

A Figura 10.1 mostra a ligação do processador às memórias de instruções e de dados. O programa por executar é armazenado na memória de instruções, que por ora consideraremos como uma memória ROM. Os dados do programa são armazenados (escritos) e recuperados

[1]© Roberto André Hexsel, 2012-2021. Versão de 2 de fevereiro de 2021.

(lidos) de uma memória RAM. A capacidade de cada uma destas memórias é de 64K (2^{16}) palavras de 32 bits.

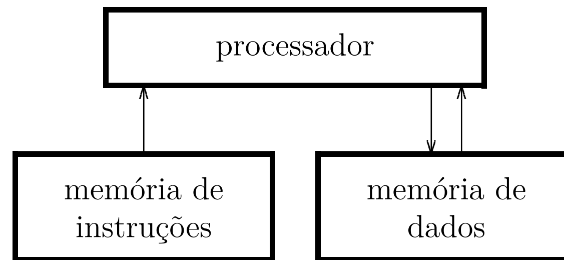


Figura 10.1: Processador e memórias de instruções e de dados.

As *instruções* são as operações indivisíveis efetuadas pelo processador, como uma soma ou uma comparação de magnitude. Um comando em linguagem de alto nível, tal como C ou Pascal, é traduzido pelo compilador para uma sequência de instruções que produz as alterações no estado da computação definidas naquele comando. Cada instrução do Mico é codificada em 32 bits, e em sua maioria, os operandos das instruções também são de 32 bits. As exceções são os endereços, que têm largura de 16 bits.

O processador, mostrado na Figura 10.2, contém quatro componentes principais: (1) um registrador que aponta a próxima instrução a ser executada e é chamado de *instruction pointer* (IP); (2) um circuito que gera os sinais de controle para os demais componentes (*cntrl*), em função da instrução que está sendo executada; (3) um *bloco de registradores* (*regs*) com 16 registradores de 32 bits, que permite observar o conteúdo de dois registradores e atualizar o conteúdo de um terceiro registrador; e (4) uma *unidade de lógica e aritmética* (ULA) na qual são efetuadas todas as operações de lógica e aritmética.

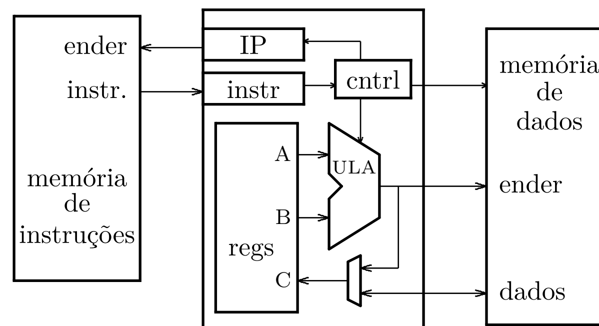


Figura 10.2: Modelo do processador.

A *arquitetura* do Mico é definida com três *espaços de endereçamento* distintos, indicados na Figura 10.3. O primeiro é o *espaço de código* ou *de instruções*, através do qual são acessadas as instruções do programa. O programador tem acesso a um conjunto de instruções que manipulam o registrador IP, e assim controlam a sequência de execução das instruções. Estas instruções são definidas na Seção 10.1.3. A faixa de endereços deste espaço é o intervalo $[0, 2^{16})$, e o registrador IP tem 16 bits de largura. As faixas de endereço são mostradas em hexadecimal na Figura 10.3.

O segundo espaço é o *espaço de dados*, pelo qual são acessadas as variáveis e estruturas de dados dos programas. São duas as instruções para acessar dados, que estão definidas na Seção 10.1.2. A faixa de endereços também é o intervalo $[0, 2^{16})$, e os endereços têm 16 bits de largura.

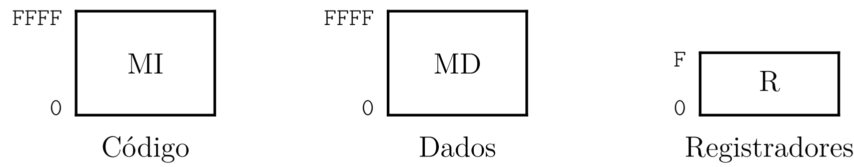


Figura 10.3: Espaços de endereçamento do Mico.

O terceiro espaço de endereçamento, *espaço registradores*, é pequeno se comparado aos outros dois, sendo somente 16 os registradores, referenciados no intervalo $[0, 2^4)$. As instruções que referenciam os registradores são definidas na Seção 10.1.1. Estes 16 registradores são usados para a manipulação aritmética ou lógica das variáveis dos programas.

Notação O caractere '`;`' significa execução sequencial; o caractere '`,`' significa execução em paralelo dos eventos à esquerda e à direita da vírgula; 'R(s)' é a sintaxe de VHDL para indexar um vetor, seja um vetor de bits seja um vetor de bytes ou inteiros; '&' é o operador de concatenação de VHDL, e '←' representa atribuição.

A Figura 10.4 mostra o modelo de execução de uma instrução de soma. No topo da figura está a *instrução* de soma, chamada de **`add`**. Esta instrução tem dois operandos, que são obtidos de dois registradores, identificados por `r1` e `r2`, e o resultado é depositado num terceiro registrador, chamado de *registrador de destino*, que é `r3` neste exemplo. O caractere '`#`' separa a instrução do comentário `r3`←`r1+r2`. O que estiver à direita do '`#`' é ignorado pelo compilador.

```
add r3,r1,r2 # r3 ← r1+r2
```

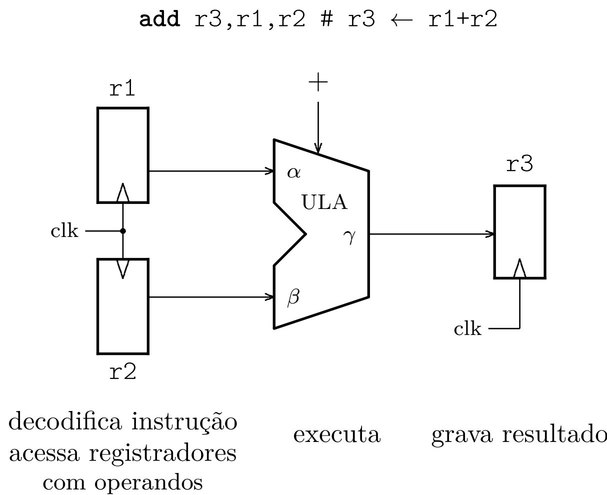


Figura 10.4: Execução de uma instrução de adição.

O circuito de controle *decodifica* a instrução e gera os sinais de controle necessários àquela instrução. No caso de um **`add`**, a operação da ULA deve ser uma soma, e o registrador de destino deve ser atualizado.

Na base da Figura 10.4 estão indicadas três das fases da execução de um **`add`**: (i) a decodificação da instrução e acesso aos seus operandos, (ii) a execução na ULA, e (iii) a gravação do resultado no registrador de destino.

O diagrama não mostra a primeira fase, que é a *busca* da instrução na memória de instruções. A memória de instruções é indexada pelo registrador IP, e o conteúdo daquele endereço é a instrução que deve ser decodificada e executada. No caso de um **`add`**, enquanto a instrução grava o resultado, o IP é incrementado e a instrução seguinte é buscada, decodificada, executada e seu resultado armazenado.

Todas as instruções do Mico são codificadas em cinco campos, como mostra a Figura 10.5. O campo no quarteto mais significativo (`opc`) corresponde ao 'nome' da instrução, e é conhecido como *operation code*, ou *opcode*. Com 4 bits, pode-se enumerar até 16 instruções distintas.

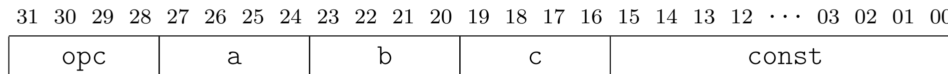


Figura 10.5: Formato das instruções do Mico.

Os próximos dois campos, `a` e `b` correspondem aos operandos das instruções de lógica e aritmética e contêm os números, ou os endereços, de dois registradores. O campo `c` indica o número/endereço do registrador que será atualizado com o resultado da operação. O campo nos 16 bits menos significativos (`const`) contém um endereço, ou uma constante codificada em complemento de dois. A maioria das instruções faz uso de alguns dos campos e ignora outros – a instrução **`add`** ignora o campo `const` enquanto que a instrução **`j`** (*jump*) ignora os campos `a`, `b` e `c`.

10.1.1 Instruções de Lógica e Aritmética

A ULA executa as operações de lógica e aritmética definidas para as instruções. O bloco de registradores contém 16 registradores de 32 bits, duas portas de leitura (A e B) e uma porta de escrita (C). Este bloco de registradores é similar ao da Seção 8.7.2.

As instruções de lógica e aritmética são **`add`**, **`sub`**, **`mul`**, **`and`**, **`or`**, **`xor`**, **`slt`** e **`not`**. Todas, exceto **`not`**, têm dois operandos provenientes de registradores, e depositam o resultado em um registrador. A instrução **`not`** tem um único operando. A Tabela 10.1 mostra as instruções de lógica e aritmética do Mico.

Tabela 10.1: Instruções de lógica e aritmética.

instrução	semântica	
`op c, a, b`	`R(c)` $\leftarrow$ `R(a) op R(b)`	$+, -, \times, \wedge, \vee, \oplus$
`slt` `c, a, b`	`R(c)` $\leftarrow$ `(R(a) < R(b))`	*set on less than*
`not` `c, a`	`R(c)` $\leftarrow$ `not(R(a))`	$\neg$
`addi` `c, a, const`	`R(c)` $\leftarrow$ `R(a) + extSin(const)`	*add immediate*

Na Tabela 10.1, `a,b,c` são nomes de registradores (endereços dos registradores, de 0 a 15) enquanto que `A,B,C` são os conteúdos dos respectivos registradores (`R(a)=A`). O registrador 0 (`r0`) contém a constante zero (`R(0)=0`) e escritas neste registrador não tem nenhum efeito.

A instrução *set on less than*, **`slt`**, faz uma comparação de magnitude entre seus operandos, e armazena os valores `TRUE`=1 ou `FALSE`=0 no registrador de destino, dependendo do resultado da comparação.

A instrução *add immediate*, **addi**, adiciona o conteúdo de um registrador (R(a)) a uma constante, representada em complemento de dois, em 16 bits. A constante é estendida de 16 para 32 bits para que possa ser aplicada à entrada β da ULA – todos os operandos da ULA são de 32 bits. Esta instrução possibilita o uso de constantes positivas ou negativas. A instrução **addi** é frequentemente usada para inicializar uma variável ou registrador com uma constante. Por exemplo,

```
addi r5, r0, K    # r5 <- 0 + K
```

soma o valor de K com zero (r0) e o atribui ao registrador r5.

Vejamos três exemplos simples de tradução de código Pascal para *assembly*. Nestes exemplos usaremos a convenção de chamar os operandos das instruções com os nomes das variáveis, prefixados de 'r' para 'registrador'.

```
Pascal                  assembly

a := b + c;             add ra, rb, rc
```

Há uma correspondência direta entre o comando (simples) em Pascal e sua tradução para *assembly*. Nosso segundo exemplo tira proveito da associatividade da soma. O resultado intermediário é acumulado no registrador ra.

```
Pascal                  assembly

a := b + c + d + e;     add ra, rb, rc    # ra <- rb + rc
                        add ra, ra, rd    # ra <- ra + rd
                        add ra, ra, re    # ra <- ra + re
```

No terceiro exemplo podemos tirar proveito da associatividade, ou então escrever código ligeiramente mais complexo para demonstrar o uso de dois registradores temporários (t0 e t1) para a computação de expressões que não sejam triviais. O código Pascal poderia ser transformado para evitar o uso dos dois registradores temporários.

```
Pascal                  assembly

f := (g+h) - (i+j);     add t0, rg, rh   # t0 <- rg + rh
                        add t1, ri, rj   # t1 <- ri + rj
                        sub rf, t0, t1   # rf <- t0 - t1
```

Espaço em branco proposital.

Exemplo 10.1 Vejamos um exemplo completo de tradução de um programa em Pascal para o seu equivalente lógico em *assembly* do Mico XII. O programa em Pascal calcula a média aritmética de dois números que são lidos do teclado. Nosso processador não opera com frações, então computaremos a aproximação inteira da média.

O código em Pascal lê dois valores do teclado, computa sua média aritmética e escreve o resultado na tela. Espaço em memória é alocado para três variáveis de tipo inteiro (a, b e m). Os parênteses são necessários no cálculo da média porque a precedência da divisão é maior do que a da soma.

Pascal

```
program med2(input,output)

var a, b, m : integer;

begin
  read(a);
  read(b);
  m := (a + b) / 2;
  write(m);
end.
```

assembly

```
# segmento de dados (RAM)
# a em r4 = ra
# b em r5 = rb
# m em r6 = rm

# segmento de código (ROM)
read ra
read rb
add  rm, ra, rb # rm <- a + b
sra  rm, rm, 1  # rm <- rm / 2
show rm
halt
```

Este programa é tão pequeno e simples que não é necessário alocar as variáveis em memória; as variáveis são alocadas em registradores, como indica o comentário do *segmento de dados*.

No segmento de código empregamos duas instruções que não são definidas para o Mico XII: a instrução **read**, que lê um inteiro do teclado; e a instrução **sra** (*shift right arithmetic*) que desloca seu operando de n posições para a direita e preserva o sinal do operando.

A expressão (a+b)/2 é traduzida para o par **add** seguido da divisão por dois – deslocamento de uma posição para a direita. A média é exibida e o programa termina com uma instrução **halt**, que paralisa o processador. ◁

10.1.2 Instruções de Acesso à Memória

A memória de dados do Mico é um vetor de inteiros com 64K elementos, endereçados de 0 a 65.535. A memória é denotada[2] por M() e se comporta como um circuito combinacional nas leituras (x←M(i)) – a porta de leitura sempre mostra o conteúdo da posição apontada pelo endereço i – e como um registrador nas escritas (M(j)←y) – a posição de memória apontada pelo endereço j é atualizada com o valor na porta de escrita na borda de subida do relógio, se a escrita estiver habilitada.

São duas as instruções de acesso à memória, uma que copia o conteúdo de um registrador para a memória (*store to memory*), e outra que copia o conteúdo de uma posição de memória para um registrador (*load from memory*). Nos dois casos, o endereço é computado pela soma do conteúdo de um registrador com uma constante. O registrador é a *base* do endereço, e a constante é um *deslocamento* com relação à base. A constante é representada em complemento de dois, em 16 bits. A Tabela 10.2 mostra as instruções de acesso à memória.

[2] A notação para vetores é aquela da linguagem de descrição de *hardware* VHDL. A indexação de vetores é denotada por parênteses (V(i)) ao invés de colchetes (V[i]).

Tabela 10.2: Instruções de acesso à memória.

instrução	semântica	
`ld c, dsl(a)`	`R(c) ← M( R(a) + extSin(dsl) )`	*load from memory*
`st dsl(a), b`	`M( R(a) + extSin(dsl) ) ← R(b)`	*store to memory*

A instrução **ld** copia o conteúdo da posição indexada pela soma do registrador base com o deslocamento. O deslocamento é uma constante representada em complemento de dois, e pode ser positiva – sendo portanto o endereço maior do que a base, ou negativa – sendo o endereço menor do que a base.

A instrução **st**, copia o conteúdo de um registrador para a posição de memória indexada pela soma do registrador base com o deslocamento. O endereço é computado como num *load*.

Alocação de dados pelo compilador O resultado da compilação de um programa são três *segmentos*: (i) um com as instruções do programa, chamado de *segmento de texto*; (ii) um com os dados, chamado de *segmento de dados*; e (iii) um segmento com a pilha de chamadas de funções, o *segmento de pilha*.

Para gerar o segmento de texto, as instruções que resultam da compilação de todas as funções do programa são agrupadas, tipicamente na ordem em que as funções aparecem no código fonte. Uma vez que todas as funções tenham sido agrupadas, é possível determinar qual é o endereço da primeira instrução de cada função, e portanto o endereço do código pode ser associado ao nome simbólico da função.

Por exemplo, num programa com as funções `f()` com 1000 instruções, e `g()` com 2000 instruções, mais o programa principal com 5000 instruções, a alocação dos endereços das funções seria algo como o mostrado ao lado.

endereço	função
`0000:`	`princ()`
`5000:`	`f()`
`6000:`	`g()`
`8000:`	fim

O mesmo processo é aplicado na alocação das variáveis e estruturas de dados. O compilador agrupa todas as declarações de variáveis no segmento de dados, e então aloca os endereços das variáveis em função dos seus tamanhos.

Considere um programa com uma variável inteira `war`, e dois vetores de inteiros `vet[]` com 100 elementos e `get[]` com 200. A alocação dos endereços das variáveis seria algo como o mostrado ao lado.

endereço	variável
`0000:`	`war`
`0001:`	`vet[]`
`0101:`	`get[]`
`0301:`	fim

O segmento de pilha é vazio porque a pilha cresce dinamicamente em função da execução do programa. O conteúdo da pilha não pode ser alocado em tempo de compilação porque, em geral, não é possível prever o comportamento do programa por causa da incerteza quanto aos dados de entrada.

A Figura 10.6 mostra o cálculo do endereço para acessar o k-ésimo elemento de um vetor, quando o registrador base aponta para o endereço do i-ésimo elemento. O registrador base é `r5`, que aponta para o endereço do elemento `V[i]`. A soma do conteúdo do registrador base com o deslocamento (2) é o endereço efetivo, e o conteúdo daquela posição é copiado para o registrador de destino (`r4`). Elementos contíguos de vetores são alocados em posições contíguas da memória.

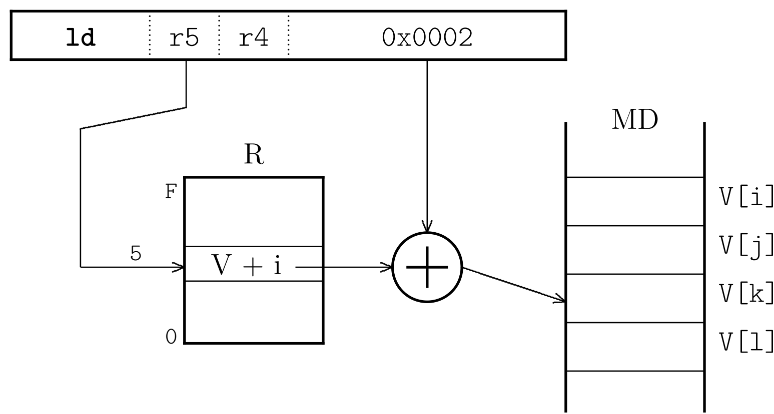


Figura 10.6: Cálculo do endereço para acessar o elemento V[k].

O trecho de código abaixo mostra um exemplo com a instrução que efetua um acesso de leitura no k-ésimo elemento de um vetor de inteiros, indicada abaixo em Pascal e em *assembly*. O endereço de um vetor, na linguagem Pascal ou em *assembly*, é representado pelo nome do vetor, que é `V` no nosso exemplo. No código *assembly*, o endereço base do vetor (`V`) é adicionado ao índice `i`; o k-ésimo elemento é acessado com um deslocamento de 2 com relação ao registrador base (`r5`). Para simplificar o código, os valores de `V` e de `i` são mantidos nos registradores apelidados de `rV` e `ri`.

Pascal

```
k    := i+2;
elem := V[k];
```

assembly

```
add r5, rV, ri # r5 <- V+i
ld  r4, 2(r5)  # acessa V[i+2] = V[k]
```

No próximo exemplo, os dois operandos da soma são copiados para registradores pelos dois **ld**s, e o resultado da adição é armazenado no quinto elemento do vetor com um **st**.

Pascal

```
V[4] := V[0]+V[1];
```

assembly

```
addi r15, r0, V # r15 <- 0 + base de V
ld   r1, 0(r15) # r1 <- M(V+0) = V[0]
ld   r2, 1(r15) # r2 <- M(V+1) = V[1]
add  r3, r1, r2
st   4(r15), r3 # M(V+4) <- V[0]+V[1]
```

Espaço em branco proposital.

A tradução para binário do comando `V[4]:=V[0]+V[1];`, assim como sua alocação em memória, são mostradas na Figura 10.7. A coluna da esquerda mostra os endereços na memória de programa e na memória de dados. Para simplificar a explanação, a primeira instrução é alocada no endereço `0x100` da memória de programa, e o endereço do primeiro elemento do vetor (`V[0]`) é alocado no endereço `0x300` da memória de dados. Neste exemplo, a memória de dados foi inicializada com valores aleatórios.

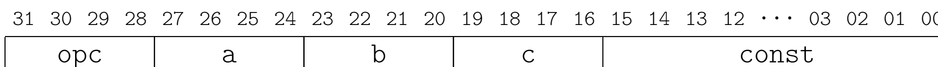
```
# memória de instruções
# end binário    assembly
...
0100: 800f.0300  addi r15, r0, V  # r15 <- 0 + base de V
0101: 9f01.0000  ld   r1, 0(r15)  # r1 <- M(V+0) = V[0]
0102: 9f02.0001  ld   r2, 1(r15)  # r2 <- M(V+1) = V[1]
0103: 0123.0000  add  r3, r1, r2
0104: af30.0004  st   4(r15), r3  # M(V+4) <- V[0]+V[1]
...
# memória de dados
...
0300: 0000.0020  V[0] = 0x20
0301: 0000.0033  V[1] = 0x33
0302: 0000.0504  V[2] = 0x504
0303: ffff.ffff  V[3] = -1
0304: 0000.0053  V[4] = 0x53
...
```

Figura 10.7: Tradução para binário de `V[4]:=V[0]+V[1]`.

A segunda coluna mostra o código binário da instrução, e a terceira coluna mostra a instrução em *assembly* que lhe corresponde. A tradução de *assembly* para binário é deveras simples, embora tediosa. Com base na Tabela 10.3, o código binário correspondente a cada instrução é gerado e acrescentado à sequência de instruções. Para facilitar a leitura, as instruções na Figura 10.7 estão divididas por um "ponto de milhar" em dois campos de 16 bits.

Tabela 10.3: Codificação das instruções do Mico (parcial).

opc	instrução	semântica	descrição
0	`add c, a, b`	`R(c) ← R(a) + R(b)`	soma registradores
8	`addi c, a, K`	`R(c) ← R(a) + extS(K)`	soma com constante
9	`ld c, D(a)`	`R(c) ← M( R(a) + extS(D) )`	*load from memory*
a	`st D(a), b`	`M( R(a) + extS(D) ) ← R(b)`	*store to memory*
e	`beq a, b, E`	`IP ← ((R(a)==R(b)) ? E : IP+1)`	desvio condicional



Na memória de programa, cada instrução ocupa um endereço, e as instruções são executadas na ordem em que aparecem na memória. O IP inicialmente aponta para `0x100`, depois para `0x101`, `0x102`, e assim por diante. Veremos em breve o que ocorre na presença de decisões e laços.

O estado da memória de dados é mostrado após a execução do comando `V[4]:=V[0]+V[1]`, depois que `V[4]`, armazenado em `M(0x304)`, é atualizado com a soma de `V[0]` e `V[1]`, por sua vez armazenados em `M(0x300)` e `M(0x301)`.

É mais fácil para humanos pensar em termos de comandos, vetores e variáveis, que são *nomes simbólicos*, enquanto que o processador trata somente de endereços e instruções simples. A tarefa do programador em *assembly*, e de quem escreve o compilador, é traduzir as construções abstratas de Pascal ou C para as sequências de instruções simples que o processador executa. Enquanto que código em linguagens de alto nível pode ser escrito em "duas dimensões", com construções tais como **`if then else`**, o código em *assembly* é 'unidimensional' e o programador em *assembly* deve traduzir os comandos de alto nível para sequências lineares de instruções.

No comando `V[4]:=V[0]+V[1]`, os índices do vetor são constantes, e o compilador computa os deslocamentos com relação à base do vetor pela mera inspeção do código fonte. Se o programador decide mudar estes índices, então o programa deve ser recompilado. Vejamos como fica o código de acesso ao vetor quando os índices variam em tempo de execução; ao invés de constantes, os índices são variáveis, cujos conteúdos se alteram ao longo da execução do programa.

Pascal

```
V[k] := V[i]+V[j];
```

assembly

```
addi r15, r0, 0300 # r15 <- base de V
add  r14, r15, ri  # V+i
ld   r1, 0(r14)    # r1 <- M(V+i)
add  r13, r15, rj  # V+j
ld   r2, 0(r13)    # r2 <- M(V+j)
add  r3, r1, r2
add  r12, r15, rk  # V+k
st   0(r12), r3    # M(V+k) <- r3
```

Para referenciar o `i`-ésimo elemento do vetor, o compilador deve gerar código para, explicitamente, efetuar a soma do *endereço base* do vetor com o deslocamento armazenado na variável `i`. As três primeiras linhas da versão em assembly o fazem: a base do vetor – endereço 0x0300 – é atribuída a `r15`, e o deslocamento indicado na variável `i` é adicionado e atribuído a `r14`. O conteúdo de `r14`, que é o endereço resultante de *base+deslocamento*, é chamado de *endereço efetivo*, e este é usado na instrução **`ld`** para atribuir `V[i]` a `r1`. O mesmo cálculo deve ser efetuado para computar os endereços de `V[j]` e `V[k]`.

Espaço em branco proposital.

Exemplo 10.2 Vejamos um exemplo mais complexo de acesso a vetores. Neste caso, um elemento do vetor `Y[]` é usado para indexar o vetor `X[]`. `N` e `M` são números razoáveis.

Pascal

```
X[0..(N-1)] : integer;
Y[0..(M-1)] : integer;
...
a := X[ Y[3] ];
...
```

assembly

```
addi ry, r0, Y   # ry <- base de Y
addi r5, ry, 3   # r5 <- Y+3
ld   r6, 0(r5)   # r6 <- Y[3]
addi rx, r0, X   # rx <- base de X
add  r7, rx, r6  # r7 <- X+Y[3]
ld   r8, 0(r7)   # r8 <- X[Y[3]]
addi r9, r0, a   # r9 <- ender a
st   0(r9), r8   # M[a] <- X[Y[3]]
```

Inicialmente, o quarto elemento de `Y[]` é carregado para o registrador `r6` com o primeiro **`ld`**. Seu valor é adicionado à base de `X[]` e o segundo **`ld`** carrega o conteúdo daquela posição para `r8`. Este valor é armazenado na variável `a` com o **`st`**. ◁

Os limites de um vetor podem ser ultrapassados caso o programador cometa um erro, e tal deve ser sinalizado durante a execução do programa Pascal. Suponha que se deseje atribuir valores ao vetor, sem nunca extrapolar seus limites. Isso pode ser facilmente obtido empregando o operador módulo (%): $(x \,\%\, k) < k$ para qualquer x. Considere um vetor declarado como

```
X[0..(N-1)] : integer;
```

quando `N` é uma potência de dois. Para garantirmos que o índice `i` nunca ultrapasse o limite superior, basta acessar o vetor com `X[i MOD N]` porque $i \,\%\, N \in [0, N)$.

O módulo é computado com uma divisão, que é uma operação deveras custosa. O módulo de uma potência de dois pode ser obtido de uma maneira mais econômica empregando-se uma conjunção. Se $N = 16$ então para qualquer i o resto da divisão por 16 é menor ou igual a 15:

$$i \,\%\, 16 \in \{0, 1, 2, \cdots, 13, 14, 15\}.$$

A conjunção bit a bit de qualquer número com 15 (1111_2) é no máximo 15. Logo

$$(i \,\%\, 16) = (i \wedge 15)$$

e a extensão óbvia para qualquer potência de dois é

$$\text{se} \quad N = 2^n \quad \text{então} \quad (i \,\%\, N) = (i \wedge (N - 1))\,.$$

Exemplo 10.3 Vejamos um exemplo em que se garante que o índice não ultrapassa os limites em acessos ao vetor. `N` é uma potência de dois.

Pascal

```
X[0..(N-1)] : integer;
...
a := X[ i MOD N ];
...
```

assembly

```
addi rx, r0, X       # rx <- base de X
addi r4, r0, (N-1)   # r4 <- N-1
and  r5, ri, r4      # r5 <- i AND N-1
add  r6, rx, r5      # r6 <- X+i
ld   r7, 0(r6)       # r7 <- X[i MOD N]
addi r9, r0, a       # r9 <- ender a
st   0(r9), r7       # M(a) <- X[i]
```

A operação de módulo é efetuada com o **`and`** e assim o acesso com o **`ld`** fica garantidamente entre os limites máximo e mínimo do vetor. A constante `N` é definida pelo programador; o endereço inicial do vetor `X[]` é definido pelo compilador ou pelo programador do *assembly*. ◁

10.1.3 Instruções de Controle de Fluxo de Execução

Até aqui vimos instruções que são executadas uma após a outra, na mesma ordem em que estão armazenadas em memória. As instruções que *alteram* o fluxo sequencial de execução nos permitem escrever programas que executam sequências distintas de comandos em função dos dados do programa. A Tabela 10.4 mostra as instruções de *controle de fluxo* do Mico.

Tabela 10.4: Instruções de controle de fluxo de execução.

instrução	semântica	
j ender	IP ← ender	*jump to address*
beq a, b, ender	if (R(a)=R(b)) IP ← ender else IP ← IP+1	*branch if equal*
nop	IP ← IP+1	*no operation*
jal c, ender	IP ← ender , R(c) ← IP+1	*jump and link*
jr a	IP ← R(a)	*jump register*

A instrução *jump* (`j`) é o equivalente em *assembly* do **goto**[3] e a sequência de execução é incondicionalmente desviada para o endereço indicado no operando. O endereço de destino do `j` deve ser o endereço de uma instrução, identificado por um nome simbólico, ou um *label*, que identifica o destino do salto.

A instrução *jump* é um apelido para a instrução `jal r0, ender`, de tal modo que o valor IP+1 é gravado no registrador `r0`, o que não altera seu conteúdo, e o efeito obtido é o de um salto para o endereço de destino. O nome para este tipo de apelido é *pseudoinstrução* porque o programador pode fazer uso de uma instrução que não existe mas que ainda assim tem uma função útil. Em geral, linguagens de montagem empregam uma série de pseudoinstruções porque elas facilitam a vida do programador. Por exemplo, a instrução `addi r5, r0, K` poderia ser apelidada de `copy r5, K`, porque este é o sentido desejado: *copiar a constante K para o registrador r5.* O compilador faz a tradução de pseudoinstruções para as respectivas instruções reais.

A instrução *branch-if-equal* (`beq`) faz uma comparação de igualdade, e se os operandos são iguais, então desvia a execução para o endereço indicado na instrução. Os conteúdos dos dois registradores (`R(a)` e `R(b)`) são comparados; se iguais, a instrução apontada por `ender` é a próxima a ser executada; se diferentes, então a instrução apontada por IP+1 é executada.

A instrução *no-operation* (`nop`) não tem nenhum efeito, a não ser a passagem do tempo para executar uma instrução. Esta instrução parece inútil nas não é, como veremos adiante. As instruções `jal` e `jr` são descritas na Seção 10.1.5.

Espaço em branco proposital.

[3] A programação em *assembly* é o único ambiente em que **goto**'s podem ser utilizados sem causar asco no leitor do código.

Comando `if` Vejamos um exemplo que emprega as três instruções que vimos nesta seção. O trecho em Pascal é um comando **`if`** simples que compara duas variáveis e incrementa uma terceira, se a comparação de igualdade tem resultado verdadeiro. Como antes, os nomes das variáveis são tratados como nomes de registradores.

Pascal

```
if (a = b) then
   c := c + 1;
cont:
```

assembly

```
if:    beq ra, rb, then
       j   cont
then:  addi rc, rc, 1 # c = c + 1
cont:  nop # código após o if(a=b)
```

A comparação do **`beq`** define qual é a instrução seguinte a ser executada. Se `R(a)` ≠ `R(b)`, então a instrução seguinte ao **`beq`** é executada; se os conteúdos dos registradores são iguais, então a próxima instrução a ser executada é aquela apontada pelo campo `ender`. Como só temos comparação de igualdade, o código *assembly* é menos elegante do que poderia: em caso de igualdade, a próxima instrução, que é a primeira da cláusula **`then`**, não pode ser a seguinte ao **`beq`**. Neste exemplo, a instrução seguinte ao **`beq`** é um *jump*, que pula "por cima" do código do **`then`**. A instrução seguinte ao **`beq`** é um **`j`** que salta para a continuação do programa, para além do código que implementa o **`then`**.

Os nomes na margem esquerda do código *assembly* são chamados de *labels*, e servem para nomear o endereço de uma instrução. No exemplo acima o *label* `then:` identifica o endereço da instrução **`addi`**, enquanto que o *label* `cont:` identifica o endereço da instrução seguinte ao comando **`if`**, e neste caso, como não sabemos qual instrução seria, usamos um **`nop`** como um marcador de local. Adiante veremos um exemplo com a tradução de *assembly* para binário de código com saltos e desvios.

Comando `if-then-else` Um comando **`if-then-else`** garante a execução exclusiva de uma das duas cláusulas, ou a clausula do **`then`**, ou a cláusula do **`else`**, mas nunca as duas. O que em Pascal ou C é 'bidimensional', em *assembly* deve ser escrito de forma 'unidimensional', ficando explícitos os saltos sobre a 'outra' cláusula, e isso é mostrado no trecho de código abaixo.

Pascal

```
if (a = b) then
   c := c AND d;
else
   c := c OR d;
cont:
```

assembly

```
       beq ra, rb, then   # a=b -> then
else:    or  rc, rc, rd   # c <- c OR d
       j cont             # pula then
then:    and rc, rc, rd   # c <- c AND d
cont:  nop
```

Aqui, podemos tirar melhor proveito da instrução **`beq`** porque a cláusula **`else`** deve ser executada, se a **`then`** não o for. Se `R(a)` ≠ `R(b)`, então a disjunção (**`else`**) deve ser executada; do contrário, a conjunção (**`then`**) é executada. Se a cláusula **`else`** é executada, então o *jump* garante que a outra cláusula não o é.

Espaço em branco proposital.

Comando `while` Um laço é mostrado no trecho de código abaixo. A variável de redução (`s`) e a de controle do laço (`i`) são inicializadas. O laço **`while`** acumula em `s` a redução por soma dos dez primeiros elementos do vetor `V`.

Pascal

```
s := 0;
i := 0;
while (i <> 10) do
begin
  s := s + V[i];
  i := i + 1;
end
fim:
```

assembly

```
         add  rs, r0, r0    # rs <- 0
         add  ri, r0, r0    # ri <- 0
         addi rv, r0, V     # rv <- V
         addi r10, r0, 10   # r10 <- 10
while:   beq  ri, r10, fim  # i=10 -> fim
           add  r5, rv, ri   # r5 <- V+i
           ld   r6, 0(r5)    # r6 <- M(V+i)
           add  rs, rs, r6   # s <- s+V[i]
           addi ri, ri, 1    # i <- i+1
           j    while        # repete
fim:     nop
```

A tradução para *assembly* inicializa as variáveis `s` e `i`, assim como um apontador para a base do vetor. O registrador `r10` é inicializado com 10, que é 1 além do limite de `i`.

No corpo do laço, o endereço do `i`-ésimo elemento é computado em `r5`, e este endereço é usado para carregá-lo em `r6`, e então somá-lo à variável `s`. Quando `i`=10, o laço termina e a execução continua a partir do endereço `fim`.

A tradução do laço para binário é mostrada na Figura 10.8. A coluna da esquerda é o endereço da instrução, e neste exemplo o código inicia no endereço `0x0010`, e o vetor `V` está alocado no endereço `0x0200` da memória de dados. A segunda coluna é a tradução para binário, e aqui os nomes das variáveis foram traduzidos para números de registradores: `s`, `i` e `v` foram alocados aos registradores `r1`, `r2`, e `r3`, respectivamente. Os *opcodes* das instruções estão listados na Tabela 10.3.

```
...                          ...
0010: 0001.0000              add  r1, r0, r0     # rs <- 0
0011: 0002.0000              add  r2, r0, r0     # ri <- 0
0012: 8003.0200              addi r3, r0, V      # rv <- 0x0200
0013: 800a.000a              addi r10, r0, 10    # r10 <- 10
0014: e2a0.001a  while:      beq  r2, r10, fim   # i=10 -> fim
0015: 0325.0000                add  r5, r3, r2   # r5 <- v+i
0016: 9506.0000                ld   r6, 0(r5)    # r6 <- M(V+i)
0017: 0161.0000                add  r1, r1, r6   # s <- s+V[i]
0018: 8202.0001                addi r2, r2, 1    # i <- i+1
0019: c000.0014                j    while        # repete
001a: 0000.0000  fim:        nop
...                          ...
```

Figura 10.8: Tradução para binário da redução do vetor.

Uma vez que todas as instruções tenham sido traduzidas e colocadas na ordem, os endereços dos *labels* `while:` e `fim:` podem ser computados. O endereço do *label* `while` é `0x0014` e o endereço que corresponde ao *label* `fim` é `0x001a`. Este código traduzido pode ser gravado na memória de programa, e então executado.

Exemplo 10.4 Considere um programa que conta o número de elementos num vetor que são maiores do que zero. A contagem encerra no primeiro elemento negativo encontrado. Vejamos a sua tradução para *assembly*.

Pascal

```
V[0..(N-1)] : integer;
...
i := 0;
c := 0;
x := V[0];
while ((x > 0)AND(i < N))
begin
   c := c + 1;
   i := i + 1;
   x := V[i];
end;
```

assembly

```
   addi ri, r0, 0     # i <- 0
   addi rc, r0, 0     # c <- 0
   addi rn, r0, N     # rn <- N
   addi rv, r0, V     # rv <- base de V
   ld   rx, 0(rv)     # rx <- V[0]
w: slt  r1, r0, rx    # r1 <- (0<V[i])
   beq  r1, r0, f     # 0>=V[i] -> fim
   slt  r1, ri, rn    # r1 <- (i<N)
   beq  r1, r0, f     # i>=N -> fim
     addi rc, rc, 1   # c <- c+1
     addi ri, ri, 1   # i <- i+1
     add  r5, ri, rv  # r5 <- V+i
     ld   rx, 0(r5)   # rx <- V[i]
     j    w           # repete
f: nop
```

A condição contém dois testes; o primeiro compara `V[i]` com zero; caso `V[i]` não seja maior do que zero, o primeiro **`beq`** desvia a execução para o final do laço. O segundo teste compara `i < N` e se resultar em falso, o segundo **`beq`** desvia a execução para o final do laço. A conjunção das duas condições está implícita na sequência dos testes: o corpo do laço é executado somente se o fluxo de execução sobreviver aos dois testes. ◁

Exemplo 10.5 Vejamos um segundo exemplo completo de tradução de um programa em Pascal para o seu equivalente em *assembly* do Mico XII. O programa em Pascal calcula a aproximação inteira da média aritmética de 64 inteiros, que são lidos do teclado.

Para fins de exemplo, o código é ligeiramente mais complexo do que o necessário: no primeiro laço, o código em Pascal lê os valores do teclado e os armazena no vetor `K[]`; no segundo laço, o vetor é então percorrido e a média computada.

Espaço em memória é alocado para quatro variáveis escalares de tipo inteiro (`c`, `v`, `s` e `m`), e para um vetor de 64 inteiros. Repare que as variáveis são sempre inicializadas antes de que sejam lidas.

Espaço em branco proposital.

Pascal

```
program med64(input,output)

var c, v, s, m : integer;
var K : array(0..63)
        of integer;

begin
  c := 0;
  while (c < 64)
  begin
    read(v);
    K[c] := v;
    c := c + 1;
  end;
  c := 0;
  s := 0;
  while (c < 64)
  begin
    s := s + K[c];
    c := c + 1;
  end;
  m := s / 64;
  write(m);
end.
```

assembly

```
# segmento de dados (RAM)
# c em r4 = rc
# v em r5 = rv
# s em r6 = rs
# m em r7 = rm
# K em M(0) até M(63)
# endereço de K[] em re

# segmento de código (ROM)
    addi r8, r0, 64    # limite
    add  rc, r0, r0    # c <- 0
    add  re, r0, r0    # K[0]=M(0)
w1: slt  r9, rc, r8    # r9 <- (c<64)
    beq  r9, r0, f1    # r9=F -> fim
      read rv
      st   0(re), rv   # K[re] <- v
      addi rc, rc, 1   # c <- c+1
      addi re, re, 1   # e <- e+1
      j    w1

f1: add  rc, r0, r0
    add  re, r0, r0
    add  rs, r0, r0
w2: slt  r9, rc, r8    # r9 <- (c<64)
    beq  r9, r0, f2    # r9=F -> fim
      ld   rv, 0(re)   # v <- K[re]
      add  rs, rs, rv  # s <- s+v
      addi rc, rc, 1   # c <- c+1
      addi re, re, 1   # e <- e+1
      j    w2

f2: sra  rm, rs, 6     # rm <- rs/64
    show rm
    halt
```

As variáveis escalares são alocadas em registradores, como indica o comentário do *segmento de dados*. O vetor `K[]` é alocado a partir do endereço zero da RAM, porque não há outras variáveis alocadas em RAM. Se `K[]` fosse alocado em outra posição, o registrador `re` deveria ser inicializado com aquele endereço, ao invés de `0`.

Como no Exemplo 10.1, aqui também empregamos as instruções **`read`** e **`sra`**, que não são definidas para o Mico XII. A instrução **`sra`** desloca seu operando de n posições para a direita e preserva o sinal do operando.

A média $(\sum_{i=0}^{63} K_i)/64$ é traduzida para o segundo laço, que percorre os elementos do vetor e acumula os resultados parciais da soma na variável `s`. Após as 64 iterações, o somatório é dividido por 64 com um deslocamento de seis posições para a direita. A média é exibida na tela e o programa termina com uma instrução **`halt`**. ◁

10.1.4 Outras Instruções

A instrução **show** mostra o conteúdo do registrador `R(a)` e tem efeito similar ao um **writeln**. Processadores "de verdade" não suportam uma instrução tal como **show** porque a operação *escrever na tela* é relativamente complexa e que pode necessitar de centenas, ou milhares, de instruções. Para nossos fins, **show** é perfeitamente adequada.

Tabela 10.5: Outras instruções.

instrução	semântica	
show a	display ← R(a)	exibe valor na tela
halt		paralisa a execução

A instrução **halt** paralisa a execução, e o processador deve ser reinicializado – com a ativação do sinal reset – para voltar a operar. Esta instrução também é útil para interromper a simulação da execução de programas, por exemplo.

10.1.5 Suporte a Funções

Uma *função* é um trecho de código que computa um *valor de retorno* a partir de um ou mais *argumentos*. A função pode ser invocada em vários pontos do programa, em uma ordem arbitrária e que depende dos dados com que o programa executa a cada vez. Logo, o endereço para onde a função retorna só pode ser determinado em tempo de execução porque o compilador não tem como prever a sequência de invocações.

A instrução que *salta para a função* deve fazer duas coisas: (i) saltar para a primeira instrução do código da função; e (ii) registrar o endereço para onde deve retornar, depois de executar todo o código da função. A instrução que *retorna da função* simplesmente salta para o endereço de retorno. A Tabela 10.6 mostra as instruções de *suporte a funções* do Mico.

A instrução *jump and link* (**jal**) tem dois efeitos: (i) salta para o endereço indicado no operando; e (ii) armazena o endereço de retorno em `R(c)`, que é o que faz a ligação do endereço de retorno – este é o *link* para o *return address*.

A instrução *jump register* (**jr**) salta para o endereço de ligação/retorno, que foi armazenado por uma instrução **jal**.

Tabela 10.6: Instruções para suporte a funções.

instrução	semântica	
jal c, ender	IP ← ender , R(c) ← IP+1	*jump and link*
jr a	IP ← R(a)	*jump register*

Espaço em branco proposital.

Considere o trecho de programa abaixo, com uma invocação da função `f()` a partir do programa principal. Por convenção, o endereço de retorno deve ser armazenado no registrador `r15`, que é apelidado de *return addres* ou `ra`. No retorno da função, a instrução **`jr`** usa o `ra` para retornar ao ponto de invocação.

Pascal / *assembly*

```
...                      ...
z := fun(x);             jal ra, fun     # IP <- fun, ra <- ret
...                 ret: add r5, r0, r1  # retorna aqui
                         ...
                         ...
                         ...
                    fun: add r4,r2,r3    # fun()
                         ...
                         jr  ra          # IP <- ret
                         ...
```

Protocolo de invocação de função

Além do mecanismo de saltar para funções, e retornar delas, o código das funções que invocam e o código das funções que são invocadas devem concordar em mais três coisas: (i) a forma de comunicação dos argumentos; (ii) a forma de comunicação do valor de retorno; e (iii) o mecanismo para controlar o aninhamento de funções. Vejamos um protocolo simplificado para resolver os dois primeiros pontos.

Um *protocolo de invocação* extremamente simples é o seguinte:

(1) dois argumentos são transferidos através dos registradores `r2` e `r3`, que são apelidados de `a0` e `a1` ('a' de argumento);

(2) o valor da função é retornado no registrador `r1`, apelidado de `v0` ('v' de valor); e

(3) o endereço de retorno é mantido no registrador `r15`, apelidado de `ra` (*return address*).

Considere o trecho de programa abaixo, com uma invocação da função `f()` de dois argumentos. Os argumentos são preenchidos nas duas primeiras linhas, e o valor da função é usado na atribuição à variável `z` na quarta linha.

Pascal / *assembly*

```
...                      ...
z := fun(x, y);          add  a0, r0, rx # a0 <- x
...                      add  a1, r0, ry # a1 <- y
                         jal  ra, fun    # IP <- fun, ra <- ret
                    ret: add  rz, r0, v0 # z <- f(x,y)
                         ...
                         ...
                    fun: add  ...        # f(a0,a1)
                         ...
                         add  v0, r0, r5 # v0 <- f(x,y)
                         jr   ra         # IP <- ret
                         ...
```

Adiante veremos como empregar funções com mais do que dois argumentos. Vejamos o que fazer quanto ao aninhamento de funções.

Aninhamento de funções

Ainda nos falta definir o mecanismo de *software* para suportar o aninhamento de funções. Em tese, o par **jal**–**jr** nos garante que cada função retornará para o ponto de onde foi invocada. Infelizmente, essa tese é inválida.

Considere o programa abaixo: o programa principal invoca `alfa()`, que invoca `beta()`, que por sua vez invoca `gama()`. Quando a função `alfa()` é invocada, o registrador `ra` é atualizado com o endereço de retorno no programa principal, indicado pelo *label* (`r_0`) na versão em *assembly*. Quando `beta()` é invocada, `ra` é sobrescrito com o endereço de retorno em `alfa()`, indicado por `r_1`. Quando `gama()` é invocada, `ra` é sobrescrito novamente com o endereço de retorno em `beta()`, que é `r_2`. Ao longo da execução deste programa, ao registrador `ra` são atribuídos os seguintes valores: *indefinido* ⇝ `r_0` ⇝ `r_1` ⇝ `r_2` ⇝ `r_1` ⇝ `r_0`.

Pascal

```
program principal(...);

  function alfa(...):integer;
  begin
    ...
    x := beta(...);
    ...
  end;

  function beta(...):integer;
  begin
    ...
    y := gama(...);
    ...
  end;

  function gama(...):integer;
  begin
    ...
  end;

begin
  ...
  z := alfa(...);
  ...
end.
```

assembly

```
alfa: addi sp, sp, -16 # aloca
      st   10(sp), ra  # ra = r_0
      ...
      jal  beta        # ra <- r_1
r_1:  add  rx, r0, r1  # x:=beta()
      ...
      ld   ra, 10(sp)  # ra <- r_0
      addi sp, sp, +16 # devolve
      jr   ra          # IP <- r_0

beta: addi sp, sp, -20 # aloca
      st   18(sp), ra  # ra = r_1
      ...
      jal  gama        # ra <- r_2
r_2:  add  ry, r0, r1  # y:=gama()
      ...
      ld   ra, 18(sp)  # ra <- r1
      addi sp, sp, +20 # devolve
      jr   ra          # IP <- r1

gama: addi sp, sp, -8  # aloca
      ...
      addi sp, sp, +8  # devolve
      jr   ra          # IP <- r_2

principal:
      addi sp,r0,0xffff # inicia
      ...
      jal  alfa        # ra <- r_0
r_0:  add  rz, r0, r1  # z:=alfa()
      ...
      halt             # termina
```

Além do endereço de retorno, é necessário alocar espaço em memória para cada uma das funções, espaço que deve ser desalocado quando a função retorna. Isso é particularmente importante com funções recursivas, que são funções que invocam a si próprias[4].

[4] Funções recursivas estão fora do escopo deste capítulo.

Uso da pilha

A estrutura de dados empregada para suportar funções é uma *pilha* porque esta estrutura reflete o comportamento natural de programas com funções, especialmente quando se emprega funções aninhadas, ou funções que invocam outras funções. Além de refletir o comportamento dos programas, a manutenção da pilha é extremamente simples e só necessita um registrador para apontar o *topo da pilha*. No nosso caso, este registrador é chamado de *stack pointer*, e o registrador empregado é o `r14`, apelidado de `sp`. Por convenção, o `sp` aponta para a última posição ocupada.

Tipicamente, o registrador que aponta para o topo da pilha é inicializado no endereço mais alto da memória de dados, e portanto a pilha *cresce para baixo*: a cada novo registro acrescentado à pilha, o valor do apontador é decrementado, e quando um registro é removido da pilha, o apontador é incrementado. Em breve veremos o que é um *registro*.

O trecho de programa abaixo mostra um programa Pascal com duas funções, `f()` e `g()`. O programa principal necessita de 256 variáveis e portanto aloca 256 posições na pilha, desde o endereço em que a pilha é inicializada (`0xffff`) descendo `0x100` posições até o endereço `0xfeff=0xffff-0x0100`. Quando o código da função principal está a executar, seu *stack pointer* aponta para `0xfeff`.

Pascal

```
program ...
...
   function f(x:int): int;
   begin
      ...
      j := g(k);
      ...
      f := ...;
   end;
   function g(x:int): int;
   begin
      ...
      g := ...;
   end;
begin
   ...
   a := f(b);
   ...
end.
```

uso da pilha

```
# ender.  função
0xffff    principal
0xfffe      256 posições
...         ...
0xff00      ...
0xfeff    <- sp de principal

0xfefe    f( )
0xfefd      32 posições
...         ...
0xfedf      ...
0xfede    <- sp de f()

0xfedd    g( )
0xfedc      16 posições
...         ...
0xfece      ...
0xfecd    <- sp de g()
```

Quando a função `f()`, que necessita 32 posições na pilha para as suas variáveis locais, é invocada, o conteúdo do `sp` é decrementado de 32 posições, passando a apontar para `0xfede=0xfeff-0x0020`. Por sua vez, a função `g()` necessita 16 posições na pilha para suas variáveis locais, e durante a execução das instruções de `g()`, o `sp` aponta para o endereço `0xfecd=0xfede-0x0010`.

No programa da página 355, a primeira instrução do programa principal inicializa o apontador de pilha para o endereço mais alto da RAM, que no caso do Mico XII é `0xffff` ou 65.535. Quando o *hardware* é inicializado, os conteúdos dos registradores são indefinidos e portanto o programa, logo no seu início, deve atribuir os valores adequados aos registradores. No nosso caso, o mais importante é inicializar a pilha com o endereço mais alto da memória RAM.

Registro de ativação

Um *registro de ativação* (*stack frame*) é alocado para cada função. A Figura 10.9 mostra o leiaute de um registro de ativação completo. Nem sempre é necessário alocar todos os campos de um registro de ativação. Por exemplo, se a função não invoca nenhuma outra função, então não é necessário salvar o endereço de retorno. Se não é necessário preservar os valores de `a0` e `a1`, então estes não são preservados no registro de ativação em memória.

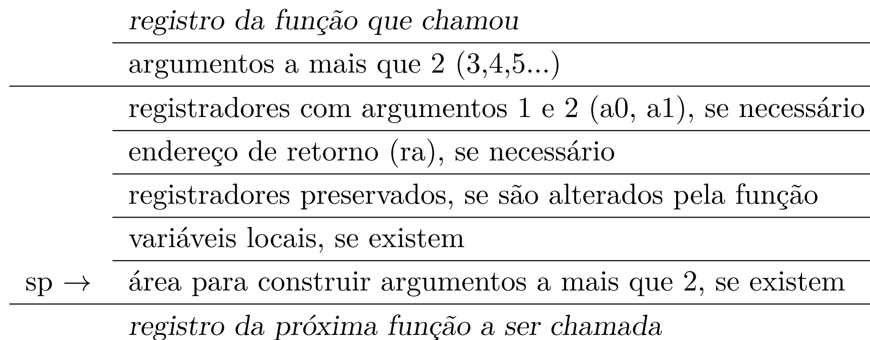


Figura 10.9: Registro de ativação no Mico.

O número de registradores do processador é limitado em 15, ou como veremos em breve, limitado a 10 se a convenção da Tabela 10.7 for observada. Isso significa que o código de uma função pode alterar os conteúdos dos registradores da função que a invocou, já que o número de registradores é limitado. Por isso o registro de ativação contém espaço para que uma função salve na pilha o conteúdo dos registradores que ela altera. Quando a função está prestes a retornar, os conteúdos originais devem ser recuperados da pilha e gravados nos respectivos registradores. Do ponto de vista da função que invoca, a função invocada não pode alterar o conteúdo de *nenhum* registrador, a menos de `v0`, `a0` e `a1`. Voltaremos a este assunto quando apresentarmos a convenção de uso de registradores.

A Figura 10.10 mostra o registro de ativação da função `g()` do exemplo acima, sem detalhar o registro de ativação da função `f()`.

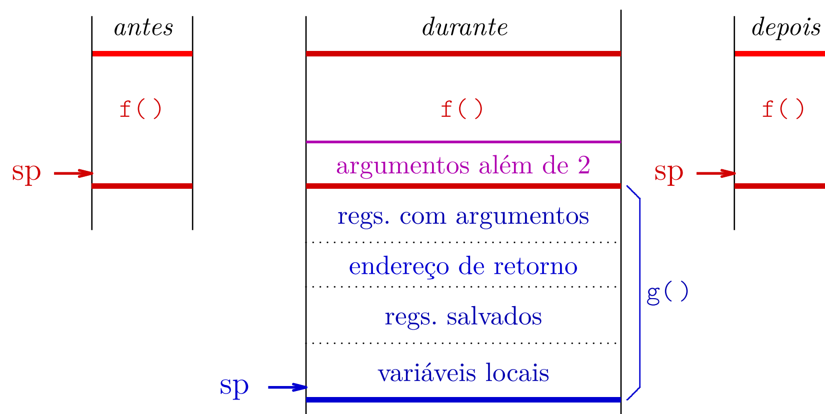


Figura 10.10: Registro de ativação das funções `f()` e `g()`.

Argumentos para além do segundo devem ser gravados no registro de ativação da função que invoca (`f()` no exemplo da Figura 10.10), o terceiro argumento apontado por `sp`, o quarto argumento na posição `sp+1`, e assim por diante.

No topo do registro de ativação são preservados os dois argumentos, seguidos do endereço de retorno. Se a função `g()` altera um ou mais registradores, seus valores originais – necessários na função `f()` – devem ser preservados no registro de ativação. Abaixo dos registradores preservados ficam as variáveis locais da função.

O programador, ou o compilador, é responsável por definir o tamanho e a quantidade de campos no registro de ativação. Para tanto, o programador deve observar quantos e quais os tipos dos argumentos da função, quantos e quais registradores o código da função altera, e quantas e quais os tipos das variáveis locais à função. Com estes dados, é uma tarefa trivial definir o registro de ativação de uma função.

Exemplo 10.6 Vejamos um exemplo com a criação de um registro de ativação para uma função com três argumentos e que emprega três variáveis locais, mostrada no Programa 10.1. A convenção de programação de funções do Mico reserva dois registradores para passar argumentos para uma função, que são os registradores `a0=r2` e `a1=r3`. Se uma função tem três ou mais argumentos, os argumentos para além do segundo devem ser armazenados na pilha *antes* da invocação da função. O terceiro argumento fica na posição apontada por `sp` *antes* do salto para a função, o quarto fica em `sp+1`, o quinto em `sp+2`, e assim por diante.

A Figura 10.11 mostra o estado da pilha durante a execução da função `h(x,y,z)`. ◁

Programa 10.1: Parte do código da função `h(x,y,z)`.

```
   # w = h(x,y,z);
   add  a0,r0,rx    # prepara 3 argumentos
   add  a1,r0,ry
   st   0(sp),rz    # empilha 3o argumento
   jal  ra, h       # salta, ra <- i = IP+1 (link)
i: add  rw,r0,v0    # valor de retorno w <- v0
   ...
   ...
h: addi sp,sp,-6    # espaço para 2 args + ra + 3 vars
   st   4(sp),a0    # empilha a0
   st   5(sp),a1    # empilha a1
   st   3(sp),ra    # empilha endereço de retorno
   # corpo de h()
   ...
   ld   r5, 6(sp)   # carrega valor de z
   ...
   add  v0,r0,r5    # valor de retorno r5 -> v0
   ld   ra,3(sp)    # recompõe endereço de retorno
   ld   a0,4(sp)    # recompõe a0
   ld   a1,5(sp)    # recompõe a1
   addi sp,sp,6     # devolve espaço na pilha
   jr   ra          # retorna para endereço i
```

sp anterior	z	sp + 6
	a1 = y	sp + 5
	a0 = x	sp + 4
	ra = i	sp + 3
	var loc1	sp + 2
	var loc2	sp + 1
sp →	var loc3	sp + 0

Figura 10.11: Registro de ativação do Programa 10.1.

Convenção para escrever funções

Para escrever programas no *assembly* do Mico usaremos uma convenção similar à do processador MIPS, indicada na Tabela 10.7.

Tabela 10.7: Convenção para uso de registradores.

reg.	nome	função
0	r0	sempre zero (*hardware*)
1	v0	valor de retorno de função
2	a0	primeiro argumento de função
3	a1	segundo argumento de função
4..13	r4..r13	registradores de uso geral
14	sp	apontador de pilha
15	ra	endereço de retorno

Além de reservar alguns registradores, a convenção determina que: (i) a pilha acima do registro de ativação da função que está executando seja preservada – a função invocada *não pode* alterar as variáveis da função que a invocou; (ii) que o conteúdo do `sp` recebido pela função seja preservado quando do seu retorno; (iii) o mesmo se aplica ao `ra`; e (iv) os registradores de uso geral cujo conteúdo é destruído pela função invocada devem ser preservados para não interferir com o estado da computação da função que a invocou, a menos do resultado da execução da função. Estas condições estão explicitadas na Tabela 10.8.

Tabela 10.8: Preservação de conteúdos entre chamadas de funções.

PRESERVADOS	NÃO PRESERVADOS
`ra` (*return address*)	`a0,a1` (argumentos)
`sp` (*stack pointer*)	`v0` (valor de retorno)
registradores de uso geral	
pilha acima do reg. de ativação corrente	pilha abaixo do `sp`

O conteúdo dos registradores com argumentos `a0` e `a1`, e o valor de retorno em `v0` podem ser alterados. A pilha abaixo do registro de ativação corrente pode ser alterada – uma função pode invocar outras funções cujos registros de ativação ficarão, na pilha, *abaixo* do registro da função corrente.

O Programa 10.2 mostra o código *assembly* para a implementação do comando `z := f(x)`, tendo `f()` um argumento inteiro e retornando um valor inteiro. A cada chamada de função encontrada num programa, o compilador deve gerar instruções para efetuar os 6 passos listados abaixo. Os números das linhas indicadas referem-se ao Programa 10.2.

1. Alocar os argumentos onde o corpo da função possa encontrá-los (linha 1);
2. transferir controle para a função e armazenar o endereço de retorno (*link*) no registrador `ra` (linha 2);
3. o corpo da função deve alocar o espaço necessário na pilha para computar seu resultado, que são 8 inteiros neste exemplo (linha 5);
4. executar as instruções do corpo da função (linha 6);
5. colocar o valor computado onde a função que chamou possa encontrá-lo, que é o registrador `v0` (linha 7);
6. devolver o espaço alocado em pilha (linha 8); e
7. retornar controle ao ponto de invocação da função (linha 9).

Programa 10.2: Protocolo de invocação de função no comando `z := f(x)`.

```
1     add  a0, r0, rx # prepara argumento em a0=r2
2     jal  ra, f      # salta para f, ra <- r = link
3  r: add  rz, r0, v0 # valor da função em v0=r1
4     ...
5  f: addi sp, sp, -8 # aloca espaço na pilha, 8 inteiros
6     ...             # computa valor de f()
7     add  v0, r0, t0 # prepara valor (v0) a retornar
8     addi sp, sp, 8  # devolve espaço alocado na pilha
9     jr   ra         # retorna, IP <- r
```

10.1.6 Conjunto de Instruções

O processador, a cada ciclo, busca e executa uma instrução de lógica/aritmética, uma instrução de controle de fluxo (salto/desvio), ou uma instrução de acesso à memória. O conjunto de instruções do Mico está definido na Tabela 10.9.

Espaço em branco proposital.

Tabela 10.9: Modelo de programação e instruções do Mico.

Modelo de programação

16 registradores, r0 a r15, r0=0, r1=v0 valor da função, r2=a0, r3=a1 argumentos, r4..r13 uso geral, r14=sp *stack pointer*, r15=ra *return address*, IP *Instruction Pointer*

Memória de programa `MI[0..64K]`, Memória de dados `M[0..64K]`, `MI` e `M` com 32 bits de largura

Conjunto de instruções

opcode	instrução	semântica	descrição
0	`add c, a, b`	`R(c) ← R(a) + R(b) , IP ← IP+1`	soma
1	`sub c, a, b`	`R(c) ← R(a) - R(b) , IP ← IP+1`	subtração
2	`mul c, a, b`	`R(c) ← R(a) * R(b) , IP ← IP+1`	multiplicação
3	`and c, a, b`	`R(c) ← R(a) ∧ R(b) , IP ← IP+1`	conjunção
4	`or c, a, b`	`R(c) ← R(a) ∨ R(b) , IP ← IP+1`	disjunção
5	`xor c, a, b`	`R(c) ← R(a) ⊕ R(b) , IP ← IP+1`	ou-exclusivo
6	`slt c, a, b`	`R(c) ← (R(a) < R(b)) , IP ← IP+1`	*set on less than*
7	`not c, a`	`R(c) ← not(R(a)) , IP ← IP+1`	complemento
8	`addi c, a, K`	`R(c) ← R(a) + extS(K) , IP ← IP+1`	soma constante aritmética
9	`ld c, D(a)`	`R(c) ← M( R(a) + extS(D) ) , IP ← IP+1`	*load from memory*
a	`st D(a), b`	`M( R(a) + extS(D) ) ← R(b) , IP ← IP+1`	*store to memory*
b	`show a`	`display ← R(a) , IP ← IP+1`	exibe valor na saída
c	`j E`	`IP ← E , R(0) ← IP+1`	salto incondicional
c	`jal c, E`	`IP ← E , R(c) ← IP+1`	*jump and link*
d	`jr a`	`IP ← R(a)`	*jump register*
e	`beq a, b, E`	`IP ← ((R(a)==R(b)) ? E : IP+1)`	desvio condicional
f	`halt`	`IP ←IP + 0`	paralisa a execução

Formato das instruções:

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12 ··· 03 02 01 00
opcode	a	b	c	const (K, D, E)

10.2 Componentes do Processador

Quais são os recursos necessários para implementar o processador?

IP
: o *instruction pointer* é um registrador de 16 bits que aponta o endereço da próxima instrução a ser executada;

MI
: uma memória de instruções, com capacidade de 64K palavras, para armazenar o programa por executar;

memória de controle
: uma memória ROM com 16 palavras de C bits que contém a tabela com todos os C sinais de controle do processador;

somador
: para incrementar o IP;

multiplexadores
: para escolher uma dentre as possíveis entradas;

ULA
: uma unidade de lógica e aritmética com operandos e resultados de 32 bits;

extensor de sinal
: para estender o sinal de um número de 16 bits representado em complemento de dois;

R
: um bloco de registradores com 16 registradores, cada um com 32 bits de largura; e

M
: uma memória de dados, com capacidade de 64K palavras, para armazenar os dados do programa.

A Figura 10.12 mostra os circuitos combinacionais necessários para a implementação do processador. A *Unidade de Lógica e Aritmética* (ULA) tem como entradas dois operandos de 32 bits e produz um resultado de 32 bits. O valor na saída depende da entrada func, que define qual a função a ser aplicada aos operandos, que é uma dentre soma, subtração, multiplicação, conjunção, disjunção, ou-exclusivo ou complemento. O *somador* efetua a soma em 16 bits de seus operandos. O *multiplexador* (*mux*) apresenta em sua saída uma dentre as suas entradas, selecionadas pela entrada sel.

A Figura 10.12 também mostra um dos elementos de estado necessários, um *registrador* com atualização que depende de um sinal de controle (habEscr). Quando habEscr=1, o valor em D é capturado na borda ascendente do relógio e mantido em Q. Se habEscr=0, então o valor memorizado anteriormente não se altera. Estas duas situações são mostradas nos diagramas de tempo. O IP é um registrador cujo sinal de habilitação está sempre ativo.

A Figura 10.13 mostra os elementos de estado algo mais complexos do que registradores 'solteiros'. O *bloco de registradores* (*register bank*) contém 16 registradores de 32 bits cada. Dois registradores podem ser acessados simultaneamente para leitura – são duas portas de leitura, A e B – e um registrador pode ser atualizado na borda do relógio, se o sinal de habilitação estiver ativo – que é a porta de escrita C. A Figura 10.13 mostra a interface do bloco de registradores e das memórias. O registrador 0 é *sempre* zero, e escritas neste registrador são ignoradas. A Seção 8.7.2 mostra o projeto detalhado de um bloco de registradores.

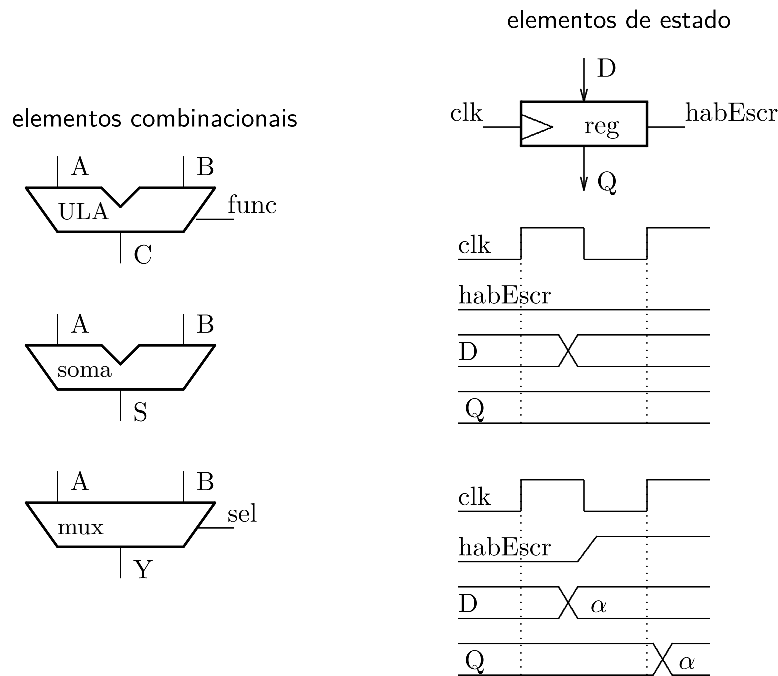


Figura 10.12: Elementos combinacionais e elementos de estado.

Os registradores cujos conteúdos são mostrados nas portas de leitura são selecionados por dois endereços de 4 bits. As portas de leitura se comportam como circuitos combinacionais e podem ser consideradas como dois vetores de inteiros independentes:

```
x <= A(a);
y <= B(b);
```

sendo a e b dois números representados em 4 bits ($\log(16) = 4$). A porta de escrita se comporta como um registrador e o novo valor só é atualizado na borda do relógio se o sinal de habilitação (hab=1) está ativo:

```
if (rising_edge(clk) and hab=1) then C(c) <= z end if;
```

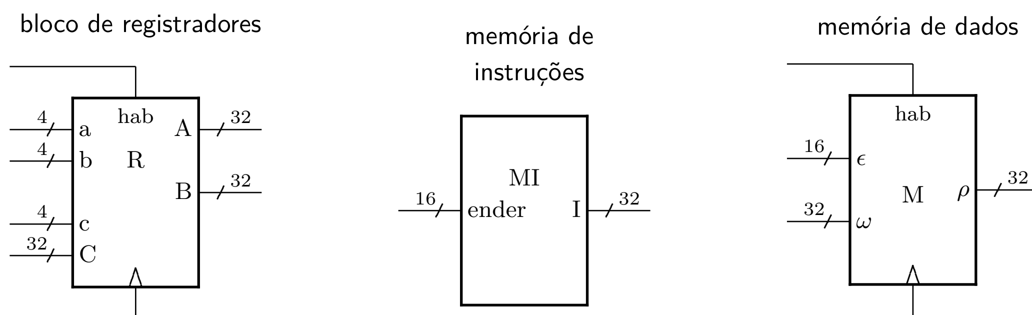


Figura 10.13: Bloco de registradores, memória de instruções e de dados.

A *memória de instruções* (MI) se comporta como um circuito combinacional – uma vez que o endereço estabiliza, passado o tempo de acesso à memória, a instrução fica estável na porta I:
I <= MI(ender);

A *memória de dados* (M) tem um comportamento similar ao banco de registradores. A porta de leitura ρ se comporta como um vetor e a leitura do conteúdo endereçado é combinacional – uma vez que o endereço estabiliza e decorrido o tempo de acesso, o dado fica disponível na porta ρ:

```
rho <= M(epsilon);
```

A atualização da palavra indexada pelo endereço ϵ só é efetivada na borda do relógio, se a escrita estiver habilitada (hab=1):

```
if (rising_edge(clk) and hab=1) then M(epsilon) <= omega end if;
```

A decodificação das instruções é extremamente simples, graças à codificação simples: basta indexar uma tabela de 16 elementos que contém todos os sinais de controle do processador. O *opcode* é um número de 4 bits e é o 'nome' da instrução. Esse número indexa a tabela de controle – implementada com uma memória ROM – e os sinais de controle de todos os componentes do processador são definidos pelos seus campos – logo veremos alguns exemplos.

10.2.1 Busca e Decodificação de Instruções

A memória de instruções (MI) mantém o código binário do programa que está sendo executado. No momento não nos interessa como o programa foi gravado nesta memória; mais adiante no curso estudaremos a carga de programas para execução. A MI é indexada pelo registrador *instruction pointer* (IP) e este registrador é incrementado de 1 a cada ciclo do relógio. A Figura 10.14 mostra um diagrama com o circuito que efetua a busca e a decodificação das instruções.

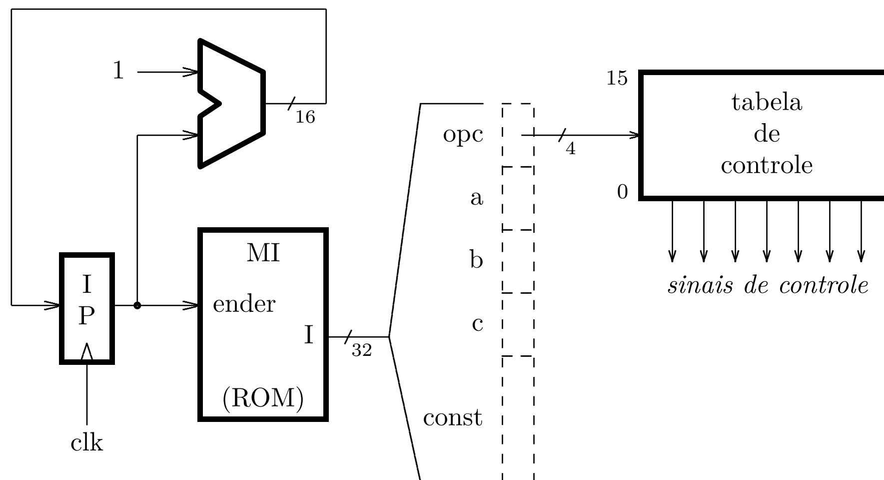


Figura 10.14: Circuito de busca e decodificação de instruções.

A saída do IP é ligada à entrada de endereços da ROM que contém as instruções do programa e a um somador, que incrementa o conteúdo do IP de 1 a cada borda do sinal de relógio (clk). Tanto o IP quanto o somador têm largura de 16 bits.

Uma vez que a saída do IP estabiliza após a borda do relógio, e decorrido o tempo de acesso da MI, uma nova instrução fica disponível para ser decodificada e executada. A *decodificação* da instrução ocorre com a indexação da tabela de controle com o *opcode* – abreviatura para *operation code* (opc), o que ativa os sinais de controle relevantes para a execução da instrução recém buscada.

A largura dos sinais é indicada no diagrama na Figura 10.14 com um traço diagonal e pelo número de bits daquele sinal. O *opcode* tem 4 bits de largura, a instrução I tem 32 bits, e o IP tem 16 bits. As setas indicam o fluxo dos sinais – de saídas para entradas.

Cada uma das instruções de 32 bits é dividida em cinco campos. O *opcode* tem 4 bits de largura e ocupa a posição mais significativa da palavra (bits 28 a 31). Os campos `a`, `b` e `c` também têm 4 bits de largura e indicam quais registradores são os operandos da instrução (`a` e `b`), e em qual registrador (`c`) o resultado deve ser armazenado –este registrador é chamado de o *destino* da instrução. O campo *constante* (`const`) armazena uma constante inteira, representada em complemento de dois em 16 bits, ou um endereço de 16 bits.

A *tabela de controle* é uma memória ROM com 16 palavras de C bits, sendo que cada um dos c_i bits é usado para controlar um dos componentes do processador, tais como multiplexadores e registradores com carga controlada.

O circuito de busca não é mostrado nos próximos diagramas, que descrevem os caminhos de dados das instruções.

10.2.2 Operações de Lógica e Aritmética

As instruções de lógica e aritmética usam dois registradores como fonte dos operandos e um terceiro registrador como destino para o resultado. A Figura 10.15 mostra uma instrução **add** no topo da figura e as ligações necessárias. Os campos da instrução `a` e `b` indexam os registradores e seus conteúdos são apresentados nas portas A e B do banco de registradores (R), e de lá para as entradas α e β da ULA.

O resultado da soma é levado da saída γ da ULA para a porta de escrita (C) do banco de registradores. O registrador que deve ser atualizado é indicado pelo campo `c`, e o sinal escReg é ativado. Na borda ascendente do relógio a soma de `R(a)` com `R(b)` é armazenada em `R(c)`.

```
add c, a, b   # R(c) ← R(a) + R(b)
```

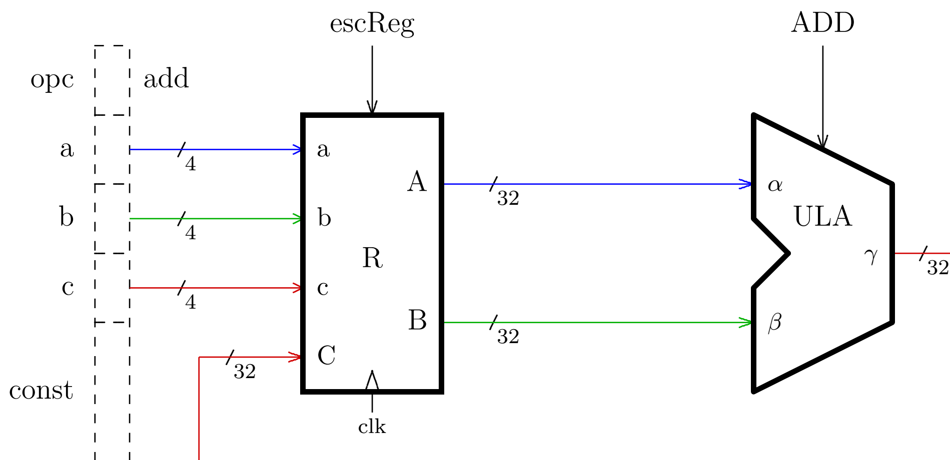


Figura 10.15: Operações de lógica e aritmética – add.

Os sinais de controle escReg e ADD são gerados/ativados na tabela de controle.

10.2.3 Operações de Lógica e Aritmética com Imediato

Há uma versão da instrução de soma que usa uma constante – que é parte da instrução - como um dos operandos, e o outro operando é o conteúdo de um registrador, apontado por a. Esta constante é chamada de *operando imediato*. O registrador de destino é apontado por c. O operando da porta B do banco de registradores não é utilizado nesta instrução e portanto as ligações de b e B são mostradas em tom desbotado. A Figura 10.16 mostra o circuito para executar a instrução de soma com um operando imediato.

A constante é codificada em 16 bits, no campo const, e ela deve ser estendida para 32 bits para que possa ser aplicada à entrada β da ULA. Na operação aritmética (**addi**), a constante é estendida para 32 bits pela replicação do seu bit de sinal.

```
addi c, a, const    # R(c) ← R(a) + extSinal(const)
```

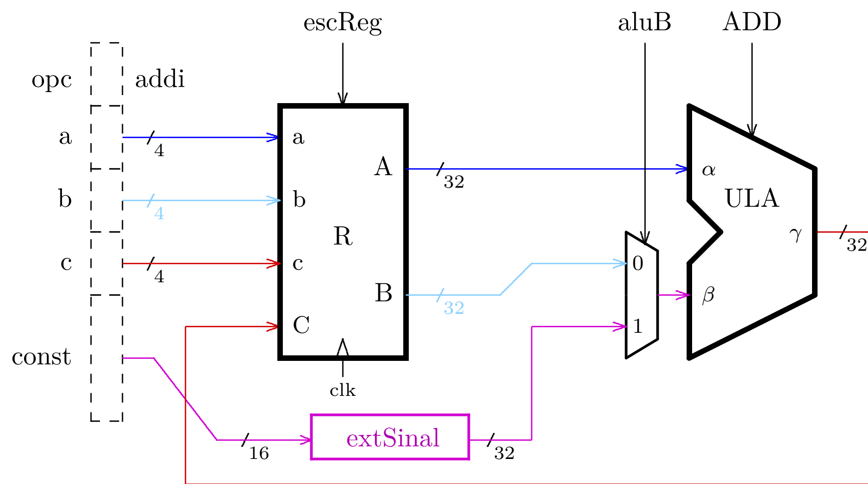


Figura 10.16: Operações de lógica e aritmética com imediato – addi.

O operando apontado por a é apresentado à entrada α da ULA; o segundo operando (const) é estendido para 32 bits e apresentado à entrada β da ULA. A operação da ULA é definida como ADD e o resultado é armazenado no registrador apontado por c, na borda de subida do relógio. Os sinais escReg e aluB devem ser 1, e estes sinais de controle, mais ADD, são ativados na tabela de controle.

Espaço em branco proposital.

10.2.4 Operação de Acesso à Memória – Leitura

A instrução `ld` (*load-word*) copia, para o registrador apontado por `c`, o conteúdo da palavra indexada pela soma de um deslocamento de 16 bits (const) com o conteúdo do registrador apontado por `a`. A soma da constante com um registrador é chamada de *endereço efetivo*. A Figura 10.17 mostra o diagrama do processador com as ligações para a instrução `ld`.

```
ld c, const(a)    # R(c) ← MD( extSinal(const) + R(a) )
```

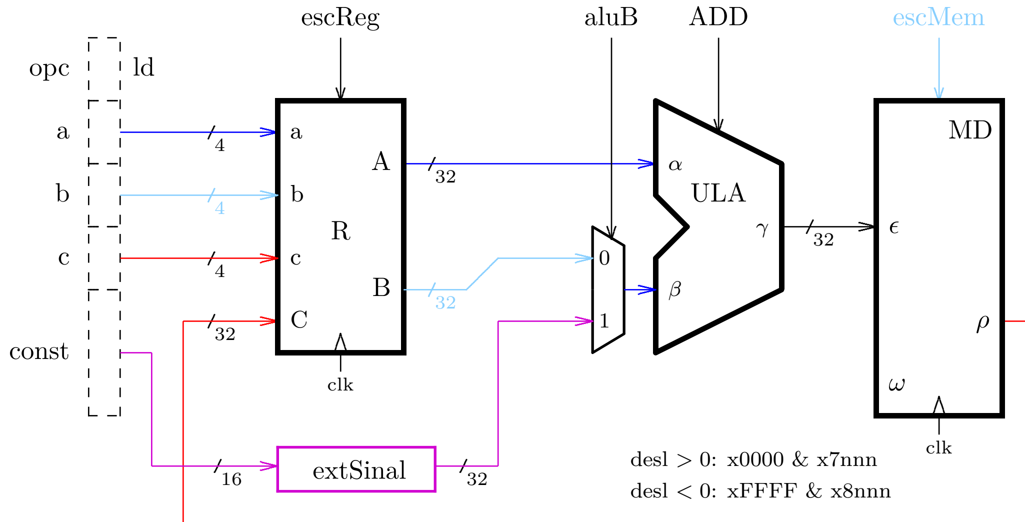


Figura 10.17: Operação de leitura da memória – `ld`.

A constante é estendida com sinal para permitir deslocamentos positivos e negativos com relação ao endereço apontado por R(a). A saída da ULA, que é o endereço efetivo, é aplicada à entrada de endereços da memória de dados. Decorrido o tempo de acesso à memória, a porta de saída ρ contém a cópia do valor lido da memória. Na borda do relógio este valor é armazenado no registrador apontado por `c`.

Os sinais de controle escReg, aluB e ADD são ativados na memória de controle. O sinal escMem não é ativado porque não ocorre uma escrita na RAM, mas somente no bloco de registradores.

Espaço em branco proposital.

10.2.5 Operação de Acesso à Memória – Escrita

A instrução **st** (*store-word*) copia o conteúdo do registrador apontado por b para a palavra indexada pela soma de um deslocamento de 16 bits (const) com o conteúdo do registrador apontado por a. A Figura 10.18 mostra o diagrama do processador com as ligações para a instrução **st**.

```
st const(a), b    # MD( extSinal(const) + R(a) ) ← R(b)
```

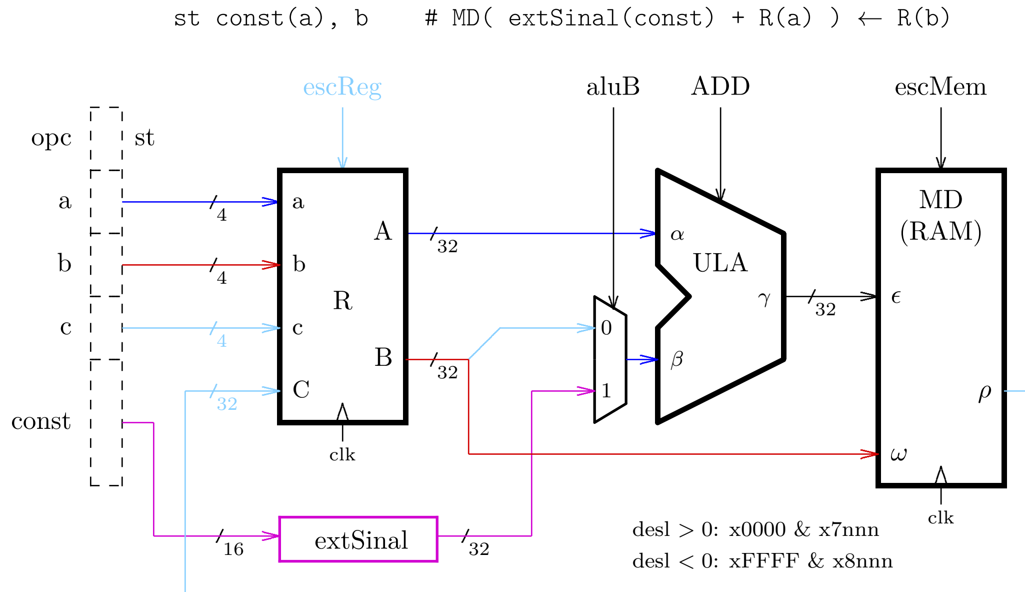


Figura 10.18: Operação de escrita em memória – st.

O endereço efetivo é aplicado à entrada de endereços ϵ da memória de dados e o conteúdo de R(b) é aplicado à porta de escrita ω da memória de dados. O sinal escMem é ativado e na borda ascendente do relógio a posição indexada pelo endereço efetivo é atualizada.

Os sinais de controle escMem, aluB e ADD são ativados na memória de controle. O sinal escReg não é ativado porque não ocorre uma escrita no bloco de registradores, mas somente na RAM.

Espaço em branco proposital.

10.2.6 Desvio Condicional

Num desvio condicional (**beq** – *branch if equal*), o endereço da próxima instrução a ser buscada depende de uma comparação de igualdade. Para tanto, o conteúdo de dois registradores são comparados. A Figura 10.19 mostra o diagrama do processador com as ligações para a instrução **beq**. O sinal iguais indica o resultado da comparação.

```
beq a, b, const   # if (R(a)=R(b)) IP <- const else IP <- IP+1
```

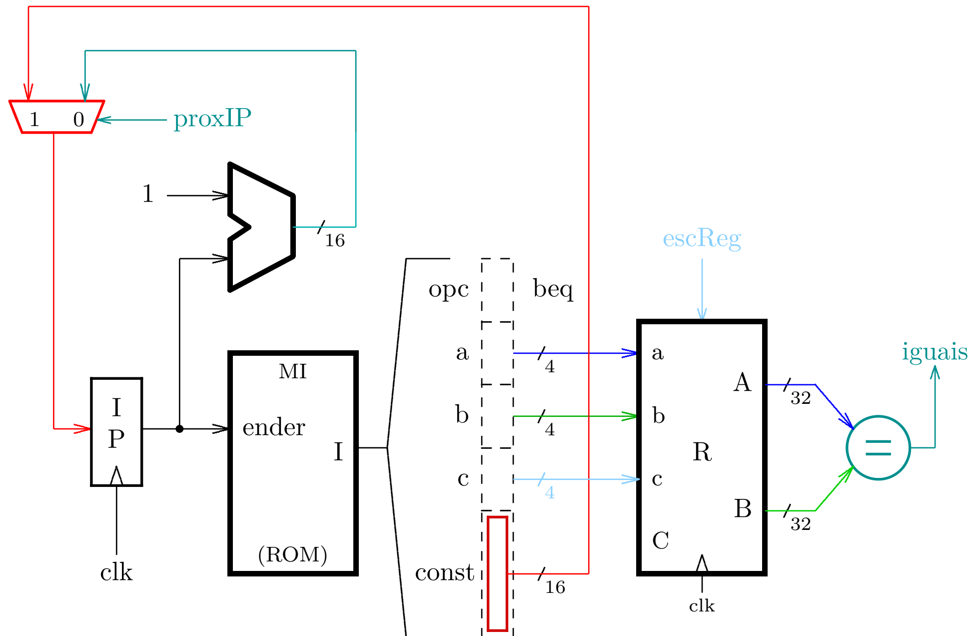


Figura 10.19: Desvio condicional – beq.

O *endereço de destino* de um desvio é a constante const. Se iguais=1 então o endereço de destino é selecionado e aplicado à entrada do IP, e a instrução apontada por aquele endereço será buscada no próximo ciclo. Do contrário, se iguais=0 então IP+1 é aplicado à entrada do IP e a instrução após o **beq** será buscada. O sinal proxIP só é ativado se a instrução for um **beq** e iguais=1.

Espaço em branco proposital.

10.2.7 Salto para Função

Num salto para função (**`jal`** para *jump and link*), o endereço de destino é o endereço da primeira instrução da função, e é codificado no campo `const`. O endereço de retorno, para onde o fluxo de controle retorna ao final da função, deve ser armazenado *ao mesmo tempo* em que ocorre o salto para o corpo da função. O valor de IP+1 é apresentado à entrada C do bloco de registradores. As condições de gravação do endereço de destino são distintas para as instruções **`jal`** e **`beq`** e isso deve ser codificado na tabela de controle. A Figura 10.20 mostra o diagrama do processador com as ligações para a instrução **`jal`**.

De acordo com a convenção de uso da Seção 10.1.5, o endereço de retorno é armazenado no registrador 15, e isso deve ser codificado no campo `c` da instrução.

```
jal 15, const    # IP <- const , R(15) <- IP+1
```

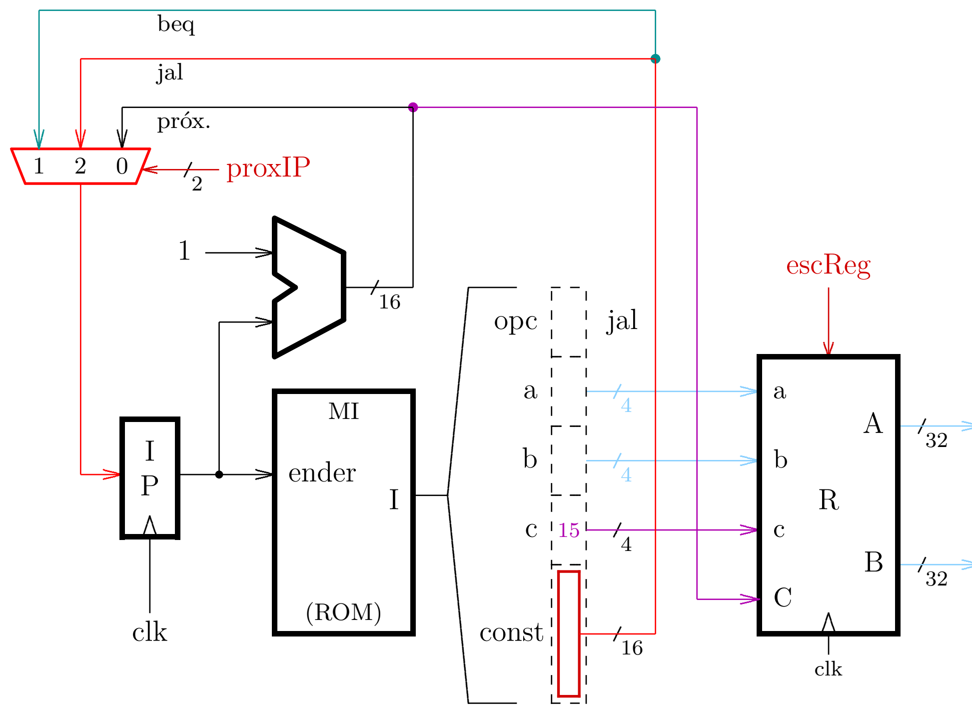


Figura 10.20: Salto para função – `jal`.

10.2.8 Retorno de Função

No retorno de uma função, a instrução *jump-register* (**`jr`**), carrega no IP os 16 bits menos significativos do valor armazenado no registrador `a`. Este registrador, que a convenção de uso recomenda ser o registrador 15, deve conter o endereço de retorno da última função invocada.

10.2.9 Circuito de Dados Completo

O circuito de dados completo é a 'união' dos circuitos necessários para cada classe de instruções e é mostrado na Figura 10.21. O circuito de dados consiste de uma Unidade de Lógica e Aritmética (ULA), de um Bloco de Registradores (R), de uma memória de dados (M), e circuitos auxiliares. O circuito de controle consiste de um apontador de instrução (IP), um somador, uma memória de instruções (MI), e de uma memória ROM com os sinais de controle.

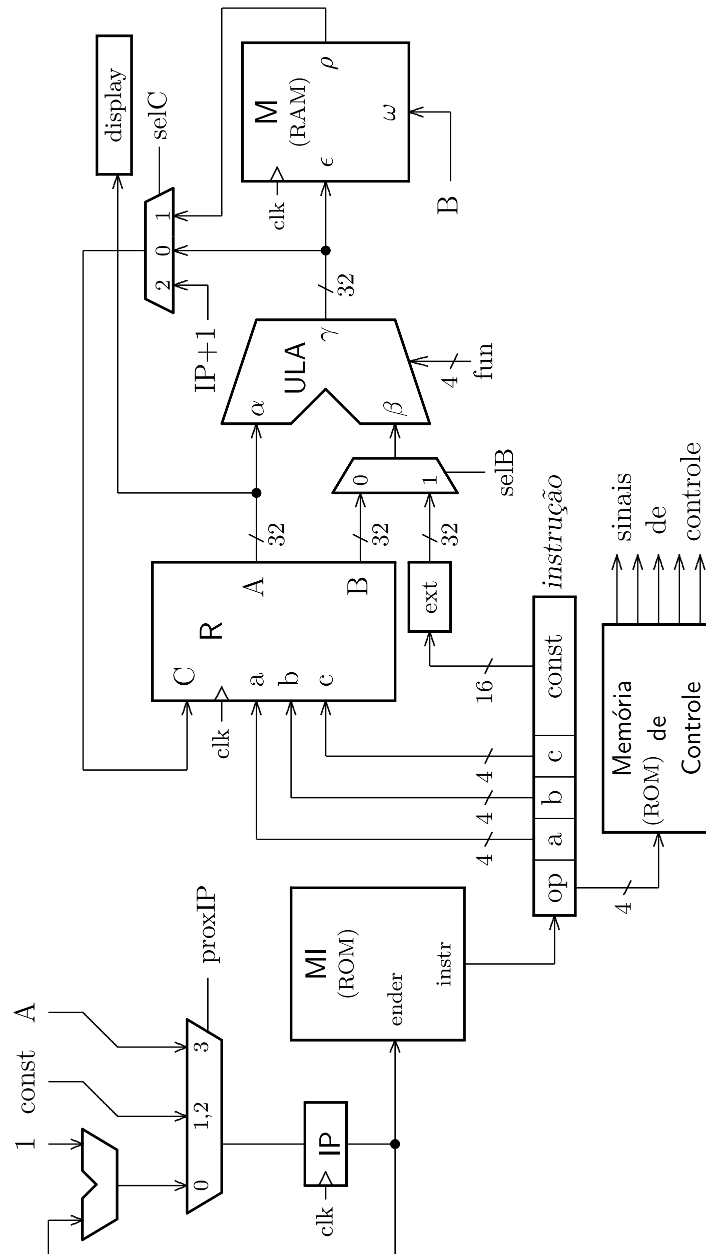


Figura 10.21: Circuito de dados e de controle do Mico.

O valor que será gravado no registrador de destino pode ser aquele na saída da ULA, ou aquele lido da memória ou IP+1. Um multiplexador, controlado pelo sinal selC, define qual dos três valores é apresentado à porta C do bloco de registradores, para ser gravado no registrador de destino.

Um dos operandos da ULA é sempre o registrador apontado por a, que corresponde à porta A do banco de registradores. O outro operando pode ser a porta B do banco de registradores ou a constante estendida. O multiplexador controlado pelo sinal selB determina o operando na porta β da ULA.

10.2.10 Tabela de Controle

O projetista do processador deve elaborar uma *tabela de controle* com todos os sinais que controlam os componentes do processador. A Figura 10.22 mostra o circuito de busca e decodificação com os sinais de controle das unidades funcionais do processador – os nomes estão abreviados no diagrama por causa do espaço. A tabela de controle é indexada pelo *opcode* da instrução recém buscada e o conteúdo do elemento indexado pelo *opcode* ativa somente os sinais de controle apropriados à instrução por executar.

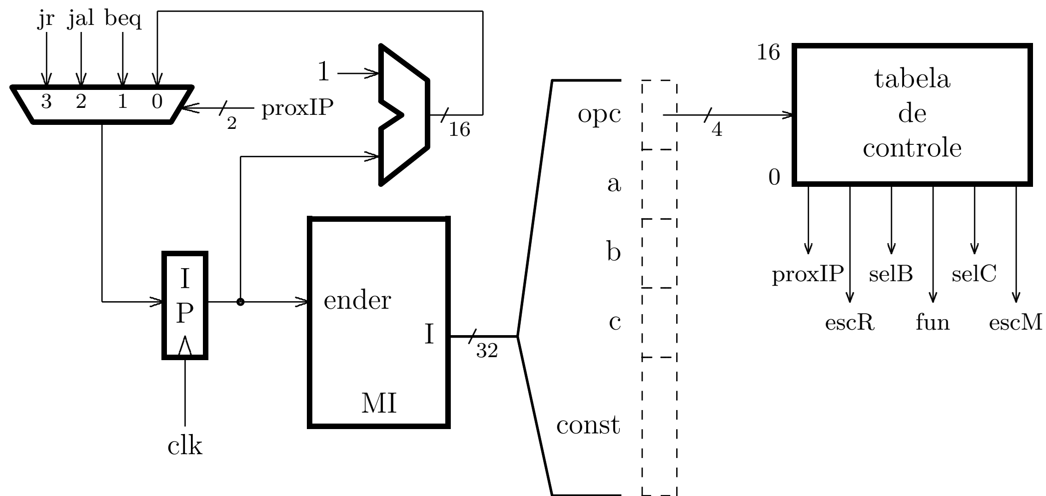


Figura 10.22: Circuito de busca e decodificação com tabela de controle.

A Tabela 10.10 mostra os sinais de controle que devem ser ativados para as instruções vistas nas seções anteriores. Sinais que prescindem de um valor determinado são indicados com um 'X' (*don't care*).

Os sinais de controle que alteram o estado do processador, proxIP, escR e escM, devem estar *sempre* determinados porque não há nenhuma instrução que *talvez* altere a memória, ou que *talvez* altere os registradores.

Espaço em branco proposital.

Tabela 10.10: Tabela de controle para as instruções.

INSTRUÇÃO	proxIP	escR	selB	fun	selC	escM
ALU	0	1	0	oper.	0	0
addi	0	1	1	ADD	0	0
ld	0	1	1	ADD	1	0
st	0	0	1	ADD	X	1
beq	iguais	0	0	X	X	0
jal	2	1	X	X	2	0
jr	3	0	X	X	X	0

10.3 Metodologia de Sincronização

A metodologia de sincronização para esta implementação do conjunto de instruções Mico é chamada de *ciclo longo* porque o período do relógio é longo o bastante para executar uma instrução durante um único ciclo.

10.3.1 ADD

A Figura 10.23 mostra o diagrama de tempo da execução de um **add**. Um diagrama simplificado do processador é mostrado na base da figura, com o bloco de registradores dividido em duas partes: no centro do diagrama está a parte na qual ocorre a leitura dos operandos, e na direita a parte em que ocorre a gravação do resultado.

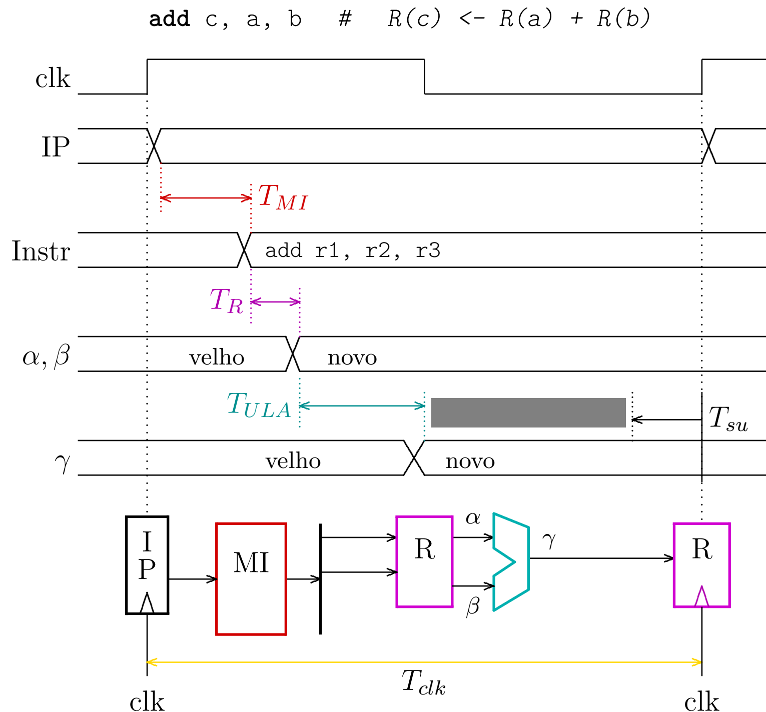


Figura 10.23: Diagrama de tempo de uma operação de ULA.

Decorrido o tempo de acesso à memória de instruções (T_{MI}), a nova instrução é decodificada paralelamente à leitura do banco de registradores (T_R). Com os operandos disponíveis, decorre o tempo de propagação através da ULA (T_{ULA}) até que o resultado esteja disponível no sinal γ. Há folga para respeitar a limitação de *setup* do banco de registradores (T_{su}), como mostra o intervalo entre a disponibilidade do novo valor em γ e a indicação do tempo de estabilização dos sinais (*setup*) – a folga é indicada pela barra de cor cinza.

10.3.2 LD

A Figura 10.24 mostra o diagrama de tempo da execução de um **ld**. O período do relógio deve ser longo o bastante para acomodar o tempo de acesso à memória de instruções (T_{MI}), a leitura do banco de registradores (T_R), o tempo de propagação através da ULA (T_{ULA}), o tempo de acesso para leitura da memória de dados (T_{MD}), e respeitar a limitação de *setup* do banco de registradores (T_{su}),. A folga é mostrada pela barra de cor cinza.

```
ld c, D(a)   #   R(c) <- M( R(a) + extS(D) )
```

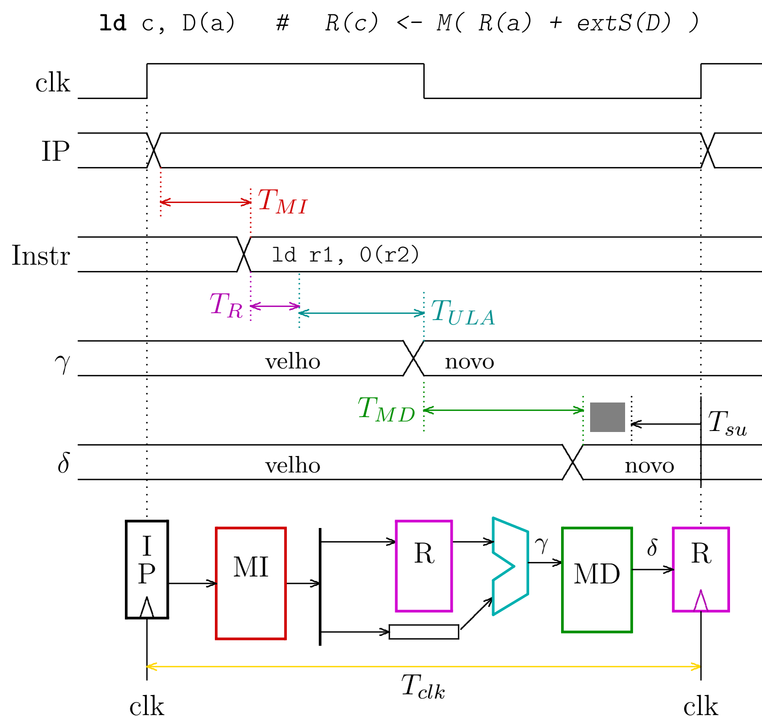


Figura 10.24: Diagrama de tempo de uma leitura em memória.

A instrução **ld** é a instrução mais demorada porque todas as unidades funcionais do processador são usadas em série, MI $\rightsquigarrow$ R $\rightsquigarrow$ ULA $\rightsquigarrow$ M $\rightsquigarrow$ R, e portanto o ciclo do processador é limitado pela temporização do **ld**.

10.3.3 Período Mínimo

As outras instruções não usam todas as unidades funcionais e há algum desperdício de tempo porque o ciclo do relógio deve ser fixado para atender ao pior caso, que é o **ld**. Por exemplo, na instrução **add**, as unidades funcionais utilizadas são MI $\rightsquigarrow$ R $\rightsquigarrow$ ULA $\rightsquigarrow$ R, não há acesso à memória de dados, e o ciclo desta instrução poderia ser encurtado de T_{MD}.

Alterar o ciclo do relógio em função da instrução não é realizável e por isso o período do relógio deve ser tal que acomode a temporização da instrução mais demorada. O Capítulo 12 discute uma implementação mais eficiente do que esta com ciclo longo.

Exemplo 10.7 Considerando os dados da Tabela 10.11, vejamos como computar a duração de um ciclo de relógio do Mico. A instrução mais demorada é o `ld` porque seu fluxo de dados atravessa, em série, todas as unidades funcionais do processador. Na tabela, todos os tempos estão em picossegundos.

O caminho crítico inicia no circuito de busca de instruções, com o registrador IP seguido da memória de instruções: $T_{IP}+T_{MI}$. A instrução indexa o banco de registradores (T_R) para obter o valor de `R(a)`; ao mesmo tempo, o deslocamento deve ser estendido[5] para 32 bits e atravessar o *mux*-2 na entrada β da ULA: $T_b + T_{m2}$. O endereço efetivo é computado na ULA (T_U) e aplicado à entrada de endereços da memória de dados; o conteúdo da posição indexada, através do *mux*-4, é levado à entrada do banco de registradores: $T_M + T_{m4} + T_{su,R}$. O tempo de propagação deste caminho é

$$\begin{aligned} T_{ld} &= T_{IP} + T_{MI} + \max(T_R,\, T_b + T_{m2}) + T_U + T_M + T_{m4} + T_{su,R} \\ &= 40 + 250 + \max(60,\, 5 + 40) + 150 + 250 + 40 + 20 \\ &= 810\,\text{ps}. \end{aligned} \tag{10.1}$$

Deve-se escolher o pior caso, o máximo, dentre dois caminhos paralelos, como é o caso do acesso à `R(a)` que ocorre em paralelo à extensão do sinal do deslocamento. ◁

Tabela 10.11: Tempos de propagação dos componentes do Mico.

Componente	T_{prop}		T_{setup}	
IP	T_{IP}	40	$T_{su,I}$	20
somador de 16 bits	T_s	75		
Mem. Instr (MI)	T_{MI}	250		
registradores (R)	T_R	60	$T_{su,R}$	20
buffer-16	T_b	5		
mux-2 de 32 bits	T_{m2}	20		
mux-4 de 16 bits	T_{m4}	40		
mux-4 de 32 bits	T_{m4}	40		
ULA (soma)	T_U	150		
Mem. Dados (M)	T_M	250	$T_{su,M}$	40

Do ponto de vista da fabricação em grande escala, não é uma boa ideia usar como período de relógio exatamente o valor para o atraso através do caminho crítico. Qualquer desvio no processo de fabricação, ou na temperatura de operação, pode provocar comportamento incorreto por problemas com a temporização. Por isso, deve-se alocar uma folga no período, algo como 5 a 10% do valor calculado na Equação 10.1. Se considerarmos uma folga pessimista de 10%, então o período mínimo seria de $T_{ld} \times 1,1 = 891\,\text{ps} \approx 890\,\text{ps}$. Esse período corresponde a uma frequência de $\approx 1,24\,\text{GHz}$, que é relativamente elevada para aplicações embarcadas. Uma frequência tão elevada implica em grande dispêndio de energia, o que pode inviabilizar aplicações alimentadas por bateria. Para estas aplicações, a frequência de operação deveria ser uma fração dos $1,24\,\text{GHz}$. A Seção 5.3.3 trata da relação entre frequência de operação e dispêndio de energia.

[5] O bit de sinal deve ser replicado 16 vezes; para evitar carga excessiva nesse sinal, empregamos um *buffer* para reproduzir as 16 cópias. As razões para tal são discutidas na Seção 5.4.

Exemplo 10.8 Uma vez definido o período mínimo do relógio, é necessário verificar se todas as demais instruções completam com folga. A instrução **st** aparenta ter um período similar ao do **ld**.

A temporização é a mesma que para o **ld** desde a busca até o cálculo do endereço efetivo. O valor a ser gravado na memória provém de `R(b)`, que fica disponível ao mesmo tempo que `R(a)`. Uma vez computado o endereço, deve-se atender ao *setup* da memória de dados $T_{su,M}$. O tempo do **st** é

$$\begin{aligned} T_{st} &= T_{IP} + T_{MI} + \max(T_R,\, T_b + T_{m2}) + T_U + T_{su,M} \\ &= 40 + 250 + \max(60,\, 5 + 40) + 150 + 40 \\ &= 540\,\text{ps} \end{aligned} \tag{10.2}$$

O caminho crítico do **st** tem $540\,\text{ps}$, o que é $0,67$ daquele do **ld**. Considerando o período de $890\,\text{ps}$, a folga é de $350\,\text{ps}$. ◁

10.3.4 Extensões ao Processador

O departamento de vendas avisa ao departamento de engenharia que os clientes necessitam de instruções adicionais, tais como deslocamentos, e de um desvio condicional que compare desigualdade. Essas modificações não podem alterar (demasiadamente) a codificação das instruções, para permitir que o código antigo e já compilado possa ser usado nos novos processadores. A Tabela 10.9 indica os possíveis caminhos para as adições.

No caso dos desvios a modificação solicitada pode ser implementada facilmente porque o campo para o registrador de destino (`c`) dessas instruções não é usado, o que permitiria adicionar quinze novas instruções de desvio, todas com o mesmo *opcode*. O campo `c` passa a ser o *opcode secundário* dos desvios.

Para implementar a instrução **beq** (*branch if equal*) basta computar a diferença entre os dois operandos e verificar se o resultado é zero. Para implementar a instrução **bzero** (*branch if zero*), o operando apontado por `a` é subtraído de `r0` e se o resultado é zero, então o desvio deve ser tomado. A instrução **ba** (*branch always*) equivale a um salto incondicional e é obtida com **beq** `r0,r0,dest`.

As instruções de ULA usam somente três registradores e ignoram o campo de deslocamento. As instruções de ULA podem ser alocadas no *opcode* `0`, por exemplo, e alguns bits do deslocamento são usados para codificar a operação de ULA desejada. Essa modificação liberaria os *opcodes* `1` a `7` para outras instruções, tais como os deslocamentos. Se forem usados 4 bits do deslocamento, a ULA pode ser expandida para suportar 16 operações ao invés das atuais 8.

Exercícios

Ex. 10.1 Mostre como implementar as instruções abaixo. Para tanto, indique quaisquer modificações que sejam necessárias no processador da Figura 10.21 e mostre a tabela de sinais de controle ativos durante a execução da instrução.

```
                   # jump-register
jr ra              # IP <- R(ra)

                   # jump-and-link register
jalr rc, ra        # IP <- R(ra) , R(rc) <- IP+1
```

Ex. 10.2 Desenhe os diagramas de tempo para as instruções do Ex. 10.1.

Ex. 10.3 Calcule o caminho crítico para as instruções do Ex. 10.1.

Ex. 10.4 Calcule o caminho crítico para as instruções de ULA.

Ex. 10.5 Calcule o caminho crítico para o `beq`. Suponha que a comparação é efetuada na ULA, através de uma subtração.

Instrução	semântica	nome
beq ra,rb,D	$IP \leftarrow D \triangleleft ra = rb \triangleright IP + 1$	*branch if equal*
bne ra,rb,D	$IP \leftarrow D \triangleleft ra \neq rb \triangleright IP + 1$	*branch if not equal*
bzero ra,D	$IP \leftarrow D \triangleleft ra = 0 \triangleright IP + 1$	*branch if zero*
bnz ra,D	$IP \leftarrow D \triangleleft ra \neq 0 \triangleright IP + 1$	*branch if not zero*
bpos ra,D	$IP \leftarrow D \triangleleft ra > 0 \triangleright IP + 1$	*branch if positive*
bneg ra,D	$IP \leftarrow D \triangleleft ra < 0 \triangleright IP + 1$	*branch if negative*
ba D	$IP \leftarrow D$	*branch always*

Tabela 10.12: Instruções adicionais: desvios condicionais

Ex. 10.6 Com base no sugerido na Seção 10.3.4, mostre como implementar as instruções da Tabela 10.12. Sua resposta deve conter a codificação das instruções, indicar como as operações de comparação são efetuadas, e os sinais de controle necessários para as novas instruções. Uma codificação cuidadosa do *opcode* secundário no campo `c` pode simplificar a lógica de controle.

Ex. 10.7 Com base no sugerido na Seção 10.3.4, mostre como implementar as modificações para as instruções de ULA. Sua resposta deve conter a codificação das instruções, e eventuais mudanças na tabela de controle do processador.

Índice Remissivo

Símbolos

T_A, 64
T_D, 120
T_I, 109, 116
T_M, 117, 119, 183
T_P, 116
T_R, 374
T_h, 185, 189, 196, 201
T_p, 170
T_s, 374
$T_{C,F}$, 189
$T_{C,x}$, 118–120, 196, 200, 202
T_{FF}, 189
T_{MD}, 374, 377
T_{MI}, 374, 377
T_{ULA}, 374, 377
T_{min}, 195, 202, 374
T_{skew}, 199
T_{su}, 185, 189, 196, 201, 374, 377
%, 18–20, 347
$\Rightarrow$, 37
$\bigvee$, 47
$\bigwedge$, 47
$\equiv$, 36
$\gg$, 155
$\wedge$, 30, 36
$\triangleleft\ \triangleright$, 36, 42, 66
$\leftarrow$, 190, 339
$\Leftrightarrow$, 36
$\Rightarrow$, 36, 37, 41, 44
$\ll$, 155
$\neg$, 30, 36, 153
$\vee$, 30, 36
$\oplus$, 36, 53, 65
$\mapsto$, 42
$\mathbb{N}$, 146
$\overline{a}$, *veja* $\neg$
π, 25
\, 77
decod, 72
demux, 75
e, 24
mod, 159, 214, 347
mux, 67
&, 339
$\mathbb{B}$, 29–33, 38, 42
$\mathbb{Z}$, 42, 151
$\lceil r \rceil$, 237, 238
$\mathbb{N}$, 42, 43, 151
num, 44, 55, 72, 77, 257, 258, 260, 270
num^{-1}, 55
$\mathbb{R}$, 42
$|\,\mathrm{N}\,|$, 223
$\langle\rangle$, 29, 42, 43, 178, 197, 221
, , 339
; , 339

Números

74148, 75
74163, 214, 268
74374, 228

A

abstração, 27
  bits como sinais, 27–33, 57, 180, 181
  tempo discretizado, 116, 118, 180, 183–185, 188–189, 193–194, 218
acumulação de produtos parciais, 171–176
adição, 153
adiantamento de vai-um, 165–169
alfabeto, 17
Álgebra de Boole, 27
algoritmo, 229
  conversão de base, 18
  conversão de base de frações, 22
amplificador, 125, 136
  diferencial, 138
amplitude, 206
and, *veja* $\wedge$
and array, 128
apontador de pilha, 264
aproximação, 24
arquitetura, 338
árvore, 64, 118
assembly, *veja* ling. de montagem
associatividade, 31
atraso, *veja* tempo de propagação, 57
atribuição, 12
autômatos finitos, 232

B

barramento, 140
barrel shifter, 159
básculo, 138, 181–186
  relógio multi-fase, 222
  SR, 182, 191
binário, 20
bit, 20
  de sinal, 147

bits, 27–37, 65
  definição, 29
  expressões e fórmulas, 30
  expressão, 30
  propriedades da abstração, 31
  variável, 30
bloco de registradores, 260, 362, 372, 373
borda, *veja* relógio
branch, *veja* desvio condicional
buffer, 123, 375
buffer three-state, 140
buraco, 87
busca, 364
busca binária, 270
byte, 11

C

C,
  deslocamento, 160
  overflow, 164
cadeia,
  de portas, 64, 118
  de somadores, 153
caminho crítico, 117
capacitor, 105, 107, 135, 136
capture FF, *veja flip flop*, destino
CAS, 134
célula de RAM, 81
célula, 100
chave,
  analógica, 142
  digital, 92
  normalmente aberta, 92
  normalmente fechada, 78, 92
chip select, 264
ciclo,
  combinacional, 57
    violação, 81, 181, 183, 185
  de trabalho, 221
ciclo longo, 373
circuito,
  combinacional, 57, 65
  de controle, 253
  de dados, 253
  dual, 99
  segmentado, 199
  sequencial síncrono, 193, 205
circuito aberto, 57
clear, 191
clk, *veja* relógio
clock, *veja* relógio
clock skew, *veja skew*
clock-to-output, 189
clock-to-Q, 189
CMOS, 59, 65, 85–142
  buffer three-state, 140
  célula, 100
  inversor, 96
  nand, 99
  nor, 98
  porta de transmissão, 141
  portas inversoras, 99
  sinal restaurado, 125
codificação das instruções,
  Mico, 340
código,
  Gray, 63, 213, 223, 232
Column Address Strobe, *veja* CAS
combinacional,
  ciclo, 57
  circuito, 57
  dispositivo, 57
comentário, 339
comparador,
  de igualdade, 62, 164
  de magnitude, 164, 257, 270
Complementary Metal-Oxide Semiconductor, *veja* CMOS
complemento, *veja* ¬
complemento de dois, 145–151, 153–164
complemento, propriedade, 31
comportamento transitório, *veja* transitório
comutatividade, 31
condicional, *veja* ◁ ▷
condutor, 85
conjunção, *veja* ∧
conjunto mínimo de operadores, 65
contador, 207–219
  74163, 214
  assíncrono, 216
  em anel, 220, 221, 228
  inc-dec, 220, 268
  Johnson, 223
  módulo-16, 211, 214
  módulo-2, 208, 219
  módulo-32, 219
  módulo-4, 208
  módulo-4096, 215
  módulo-8, 209, 217
  ripple, 216, 218
  síncrono, 218
contra-positiva, 41
controlador, 238
  de memória, 136
controle de fluxo, 348–352
conversão de base, 18
conversor,
  paralelo-série, 225
  série-paralelo, 224
corrente, 85, 105, 106, 112, 113
  de fuga, 115
corrida, 123, 125
CSS, 193, 205
curto-circuito, 57

D

datapath, *veja* circuito de dados
decimal, 17

decodificação, 339, 364
decodificador, 72, 78, 82–84, 126
  de linha, 126, 135
  de prioridades, 74
delay, 57
demultiplexador, 75, 120
design unit, *veja* VHDL, unidade de projeto
deslocador exponencial, 156
deslocamento, 155–159
  aritmético, 155, 159, 164
  exponencial, 159, 197
  lógico, 155, 162
  rotação, 159
desvio condicional, 369, 377
detecção de bordas, 123
diagrama de estado, 230
  restrições, 232
disjunção, *veja* $\vee$
dispositivo, 85
  combinacional, 57
distributividade, 31, 34, 51
divisão de frequência, 209, 212
divisão inteira, 44, 269
doador, 87
don't care, 70, 372
dopagem por difusão, 86
dopante, 86
DRAM, 134–137
  controlador, 136
  fase de restauração, 137
  linha,
    de palavra, 135
  linha de bit, 135
  linha de palavra, 136
  página, 135
  refresh, 135
dual, 32, 95, 99
dualidade, 32
duty cycle, 221

E

EEPROM, 133
endereço, 77
  base, 346
  de destino, 348, 369, 370
  de retorno, 353, 370
  efetivo, 346, 367, 368
energia, 100, 105, 112–115
enviesado, relógio, *veja skew*
EPROM, 133
equação característica do FF, 190
equivalência, *veja* $\Leftrightarrow$
erro,
  de representação, 23
espaço,
  de nomes, 338
especificação, 42
estado, 180
estado atual, 193, 205, 208
execução, 365–367, 372
  paralela, 339
  sequencial, 339
exponenciação, 260
expressões, 36
extensão,
  de sinal, 366

F

fan-in, 109–112, 116, 167
fan-out, 82, 84, 109–112, 116, 120, 169, 170
fatorial, 269
fechamento, 31
FET, 91
Field Effect Transistor, *veja* FET
Field Programmable Gate Array, *veja* FPGA
FIFO, 267
fila, 266–269
  circular, 267
filtro digital de ruído, 187
flip-flop, 185–191
  adjacentes, 194
  comportamento, 190
  destino, 194
  fonte, 194, 203
  mestre-escravo, 186
  modelo VHDL, *veja* VHDL, *flip-flop*
  temporização, 189
  tipo JK, 190
  tipo T, 188, 190
  um por estado, *veja* um FF por estado
folga de *hold*, 195
folga de *setup*, 195
forma canônica, 48
formato de instrução,
  Mico, 340
FPGA, 191
frações, *veja* ponto fixo
frequência, 206
frequência máxima, *veja* relógio
função, 30
  aninhamento, 355
  protocolo de invocação, 354
  tipo, 29, 42
função de próximo estado, 229, 236
função de saída, 229, 236
função, aplicação bit a bit, 32
função, tipo (op. infixo), *veja* $\mapsto$
funções, 353–360

G

glitch, *veja* transitório
GND, 93
gramática, 17

H

handshake, 253
hexadecimal, 19
hold time, 185, 193–199, 242
  folga, 195, 200, 201

I

idempotência, 31
identidade, 31
igualdade, 30
imediato, 366
implementação, 42
implicação, *veja* ⇒
informação, 16
inicialização,
  flip-flop, 190
Instrução,
  busca, *veja* busca
instrução, 12, 338
  add, 339, 365, 373, 374
  addi, 341, 366
  beq, 348, 369, 376
  busca, *veja* busca
  com imediato, 366
  decodificação, *veja* decodificação
  execução, *veja* execução
  formato, 340
  j, 348
  jal, 353, 370
  jr, 353, 370
  lógica e aritmética, 365
  ld, 343, 367, 374, 375
  nop, 348
  resultado, *veja* resultado
  slt, 340
  st, 343, 368, 376
interface,
  de rede, 13
  de vídeo, 12
inversor, 96
  tempo de propagação, 109
involução, 31, 61
IP, 338, 362, 364
isolante, 85

J

Joule, 112
jump, *veja* salto incondicional

L

label, 348, 349
latch, *veja* básculo
latch FF, *veja flip flop*, destino
launch FF, *veja flip flop*, fonte
Lei de Joule, 112
Lei de Kirchoff, 106
Lei de Ohm, 105, 106
LIFO, 264
ligação,
  barramento, 140
  em paralelo, 93, 99
  em série, 93, 99
linguagem,
  assembly, *veja* ling. de montagem
  C, *veja* C
  de montagem, 337–361
  Pascal, *veja* Pascal
  VHDL, *veja* VHDL
  Z, 27
linha de endereçamento, 78
literal, 38
logaritmo, 43
lógica restauradora, 125

M

MADD, 197
Mapa de Karnaugh, 49, 124
máquina de estados, 229, 232
  Mealy, 234, 241, 252, 269
  Moore, 233, 240, 252, 265
  projeto, 236
Máquina de Mealy, *veja* máq. de estados
Máquina de Moore, *veja* máq. de estados
máscara, 32
máximo e mínimo, 31
maxtermo, 46
MD, 364
Mealy, *veja* máq. de estados
memória, 179
  atualização, 77
  bit, 181
  de controle, 372
  de dados, 364
  de instruções, 363
  de vídeo, 13
  decodificador de linha, 80
  endereço, 77
  FLASH, 133
  matriz, 129, 132, 134
  multiplexador de coluna, 80
  primária, 13
  RAM, 81, 134, 263
  ROM, 78, 126, 241
  secundária, 13
memória dinâmica, *veja* DRAM
memória estática, *veja* SRAM
metaestabilidade, 181, 185, 188, 191
  defesa contra artes das trevas, 188
metodologia de sincronização, 373
MI, 363
Mico XII, 337
  assembly, 337–361
  processador, 362–377
  temporização, 373–376
microcontrolador, 241–249
microrrotina, 249
mintermo, 45, 124, 126
MIPS32, 337, 359
modelo,
  funcional, 44
  porta lógica, 96
  temporização, 115
módulo de contagem, 217
módulo, *veja* %, *mod*

Moore, *veja* máq. de estados
MOSFET, 91
multiplexador, 61, 66–70, 80, 101, 117, 119, 123–124, 126, 141, 142, 362, 372
  de coluna, 132, 136
multiplicação, 170–176, 258
  acumulação de produtos parciais, 171–176
multiplicador,
  somas repetidas, 258, 269
multiply-add, *veja* MADD

N

número,
  de Euler, 24
negação, *veja* ¬
nível lógico,
  0 e 1, 28
  indeterminado, 28, 109, 116, 141
  terceiro estado, 140
nó, 96
nomes simbólicos, 346
non sequitur, 37
not, *veja* ¬
número primo, 53

O

octal, 18
onda quadrada, 185, 206
opcode, 340, 364, 372
  secundário, 376
operação,
  binária, 29
  bit a bit, 32
  infixada, 42
  MADD, 197
  prefixada, 47
  unária, 29
operação apropriada, 194
operações sobre bits, 29–33
operador,
  binário, 29
  lógico, 36
  unário, 29
operation code, *veja opcode*
or, *veja* ∨
or array, 129
ou exclusivo, *veja* ⊕
ou inclusivo, *veja* ∨
output enable, 264
overflow, 148–150, 155, 163–164, 171, 176

P

paridade,
  ímpar, 49
  par, 49
Pascal, 341–360
  funções, 353–360
  `if`, 349
  `if-then-else`, 349
  `while`, 350
período mínimo, *veja* relógio
pilha, 264–266, 269, 355–360
pipelining, *veja* segmentação, 199
piso, *veja* ⌊v⌋
ponto fixo, 151–152
ponto flutuante, 70
pop, 264
porta,
  de escrita, 261
  de leitura, 261
porta lógica, 65
  and, 59
  carga, *veja fan-out*
  de transmissão, 141, 181
  nand, 60, 99
  nor, 60, 98
  not, 59, 96
  or, 59
  xor, 60, 65, 188
portas complexas, 100
potência, 112–115
  dinâmica, 114
  estática, 115
potenciação, 43
precedência, 30
precisão,
  representação, 23
preset, 190
prioridade,
  decodificador, 74
processador, 12, 360–377
produtório, 33
produto de somas, 46
programa de testes, *veja* VHDL, *testbench*
PROM, 133
propriedades, operações em $\mathbb{B}$, 31
protocolo,
  de sincronização, 253, 269
  invocação de função, 354
prova de equivalência, 40–41
próximo estado, 193, 208
pseudoinstrução, 348
pull-down, 96
pull-up, 96, 126, 141
pulso, 122, 123, 182, 191, 207, 215, 231, 257
  espúrio, *veja* transitório
push, 264

R

raiz quadrada, 270
RAM, 12, 77, 81, 134–139
  célula, 81
  dinâmica, 135
Random Access Memory, *veja* RAM
RAS, 134
Read Only Memory, *veja* ROM
realimentação, 81
receptor, 87
rede, 96

redução, 33, 350
refresh, 137, 139
Register Transfer Language, *veja* RTL
registrador, 191, 228, 246, 260, 338, 362
  carga paralela, 192
  de segmento, 197
  destino, 372
  ra, 359
  simples, 192
  sp, 356, 359
registrador de deslocamento, 224–228
  modelo VHDL, *veja* VHDL, registrador
  paralelo-série, 225
  série-paralelo, 224
  universal, 227
registrador de estado, 193
registro de ativação, 357
relógio, 183, 185, 206–221, 374
  bordas,
    ascendente, 186
    descendente, 186
  ciclo de trabalho, 221
  enviesado, *veja skew*
  frequência máxima, 195
  multi-fase, 221
  período mínimo, 195, 374–376
representação,
  abstrata, 28
  binária, 20
  complemento de dois, 147
  concreta, 27
  decimal, 17
  hexadecimal, 19
  octal, 18
  ponto fixo, 151
  posicional, 17
  precisão, 23
reset, 182, 190, 192
resistência, 87, 100, 105
resultado, 339, 365, 367, 370
retorno de função, 370
return address, 353
ROM, 12, 77–80, 126–133
rotação, 159, 164
Row Address Strobe, *veja* RAS
RTL, 199

S

salto incondicional, 348
salto para função, 370
segmentação, 197
segmento, 343
  dados, 342, 343, 352
  texto, 342, 343
seletor, 72
semântica, 17
semicondutor, 85
  tipo N, 87
  tipo P, 87
set, 182, 190
setup time, 185, 193–199, 242, 374
  folga, 195, 200, 201
silogismo, 37
simplificação de expressões, 38–40
sinal, 27, 42
  analógico, 27
  digital, 27, 28
  fraco, 125, 126, 138, 141
  intensidade, 90, 139
  restaurado, 125, 139
síntese, *veja* VHDL, síntese
skew, 199–203
Solid State Disk, *veja* SSD
soma, *veja* somador
soma de produtos, 45, 51, 126
  completa, 45
somador, 145, 152–153, 362
  adiantamento de vai-um, 165, 228
  cadeia, 153
  completo, 104, 152
  overflow, 163
  parcial, 103
  seleção de vai-um, 169
  serial, 227
  temporização, 197
  teste, 176
somatório, 33
`spice`, 28
SRAM, 138–139
SSD, 14
stack frame, 357
status, 162
subtração, 153–155
superfície equipotencial, 96, 110

T

tabela,
  de controle, 365, 372
  de excitação dos FFs, 190
tabela verdade, 33–35, 45
tamanho, *veja* |N|
tempo,
  de contaminação, 115, 118–121, 181, 189
  de estabilização, 185, 189
  de manutenção, 185, 189
  de propagação, 57–58, 61, 64–65, 73, 77, 83, 102, 104, 109, 115–117, 189, 203, 218, 219
  discretizado, 183, 185, 189, 194, 218
temporização, 104–126, 373–376
tensão, 105
Teorema,
  Absorção, 49
  DeMorgan, 32, 41, 48, 54, 60, 61, 94, 98
  Dualidade, 99
  Simplificação, 49
terceiro estado, 140–141
testbench, *veja* VHDL, *testbench*
teste,

cobertura, 177
de corretude, 176
teto, *veja* ⌈r⌉
three-state, *veja* terceiro estado
tipo,
de sinal, 42
função, 29
Tipo I, *veja* formato
Tipo J, *veja* formato
Tipo R, *veja* formato
transferência entre registradores, *veja* RTL
transistor, 87–91, 95–96
CMOS, 95
corte, 113
gate, 88
saturação, 113
sinal fraco, 90
tipo N, 88
tipo P, 89
Transistor-Transistor Logic, *veja* TTL
transitório, 121–123, 182, 183, 235
transmissão,
serial, 226
transmission gate, *veja* porta de transmissão
TTL,
74148, 75
74163, 214
74374, 228
tupla, *veja* ⟨ ⟩
elemento, 32
largura, 43

U

ULA, 161–164, 178, 260, 338, 362, 366, 372
status, 162
um FF por estado, 249–252
Unidade de Lógica e Aritmética, *veja* ULA

V

valor da função, 30
VCC, 93
vetor de bits, *veja* ⟨ ⟩, 32
largura, 43
VHDL, 191, 339, 342
design unit, *veja* VHDL, unidade de projeto
síntese, 203
std_logic, 140
tipos, 42

W

Watt, 112
write back, *veja* resultado

X

xor, *veja* ⊕

Z

Z, linguagem, 27