

Capítulo 12

Duas Implementações para MIPS32r2

Before you design, you must understand the components.

Yale Patt.

Esta seção¹ descreve duas implementações do conjunto de instruções MIPS32, uma que é simples e sem paralelismo na execução, e outra que tira proveito de paralelismo e portanto tem melhor desempenho. A implementação com paralelismo introduz dois artefatos que podem reduzir a eficiência desejada. Para mais detalhes quanto ao *hardware*, veja [PH14]; para detalhes quanto ao *software*, veja [Swe07].

Evidentemente, a implementação do conjunto completo de instruções não é discutida por ser demasiado complexa; o que se deseja é mostrar os princípios de funcionamento do processador. A implementação de algumas poucas instruções representativas é discutida; outras são indicadas nos exercícios. O código VHDL de uma implementação completa do conjunto de instruções MIPS32r2 pode ser encontrado em [Hex15].

12.1 Um subconjunto das instruções MIPS32r2

O conjunto de instruções a ser estudado aqui é mostrado na Tabela 12.1. Cada uma delas representa uma classe de instruções e a implementação de uma classe completa depende de pequenas variações nos circuitos que são discutidos no que se segue.

Convenção: 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 palavras; ‘&’ é o operador de concatenação de VHDL, e ‘←’ representa atribuição. A operação de alguns dispositivos é definida por código VHDL.

Além de uma Unidade de Lógica e Aritmética (ULA), o processador deve conter um bloco de registradores (R) para manter os resultados intermediários da computação. Um registrador que é usualmente chamado de *contador de programa* (*Program Counter* ou PC) indica qual é a próxima instrução a ser buscada da memória de instruções e então executada. Dependendo do fabricante, esse registrador pode ser chamado de *Instruction Pointer*. Empregaremos o nome mais usual que é PC.

¹© Roberto André Hexsel, 2012-2021.

Tabela 12.1: Subconjunto das instruções do MIPS

INSTRUÇÃO	DESCRIÇÃO
addu rd, rs, rt	$R(rd) \leftarrow R(rs) + R(rt)$, $PC \leftarrow PC+4$
ori rt, rs, im16	$R(rt) \leftarrow R(rs) \text{ OR } \text{extZero}(im16)$, $PC \leftarrow PC+4$
lw rt, des16(rs)	$R(rt) \leftarrow M(R(rs) + \text{extSinal}(des16))$, $PC \leftarrow PC+4$
sw rt, des16(rs)	$M(R(rs) + \text{extSinal}(des16)) \leftarrow R(rt)$, $PC \leftarrow PC+4$
beq rs, rt, des16	if (R(rs) = R(rt)) $PC \leftarrow PC+4 + \text{extSinal}(des16) \& 00$ else $PC \leftarrow PC+4$
j ender26	$PC \leftarrow PC(31:28) \& \text{ender26} \& 00$

Quais são os recursos necessários para implementar as instruções da Tabela 12.1?

Para quase todas elas é necessário um somador para incrementar o PC de 4.

Para quase todas elas é necessário um bloco de registradores que permita a leitura de um ou dois registradores e a escrita em um registrador.

Para as instruções de lógica e aritmética é necessária uma unidade de lógica e aritmética que implemente as operações usuais, tais como soma, subtração, *and*, *or*, *xor* e *not*.

Para a instrução **ori** é necessário um circuito que efetue a disjunção bit a bit – que existe na ULA – e um circuito que estenda com zeros um número de 16 bits para um número com 32 bits.

Para as instruções **lw** e **sw** é necessária uma memória que permita escritas e leituras, um somador – da ULA – e um circuito que estenda o sinal de uma constante de 16 bits, representada em complemento de dois, para 32 bits.

Para **beq** é necessário um comparador de igualdade, para comparar dois números de 32 bits.

Embora não esteja indicado na Tabela 12.1, todas as instruções devem ser buscadas da memória de instruções, para que possam ser decodificadas e então executadas. A memória de instruções é indexada pelo PC. A cada instrução que não seja um desvio ou salto, o PC é incrementado de 4; em desvios e saltos, o valor carregado no PC depende da instrução.

A Figura 12.1 mostra os circuitos combinacionais necessários para a implementação do conjunto de instruções. 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, conjunção, disjunção, ou-exclusivo ou *nor*. O *somador* efetua a soma em 32 bits de seus operandos. O *multiplexador* (*mux*) apresenta em sua saída uma dentre as suas entradas, selecionadas pela entrada *sel*.

A Figura 12.1 também mostra os elementos de estado necessários. Os mais simples são *registradores*, que nada mais são do que vários *flip-flops* tipo D que compartilham os sinais de relógio (*clk*) e habilitação (*habEscr*). Quando *habEscr*=1, na borda ascendente do relógio o valor em D é capturado 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 PC é um registrador cujo sinal de habilitação está sempre ativo e portanto é atualizado em todos os ciclos do relógio.

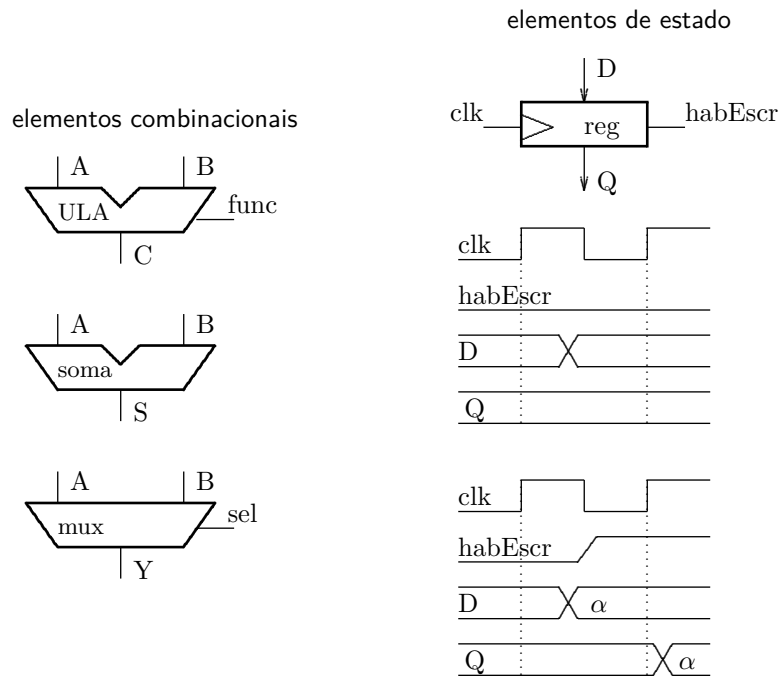


Figura 12.1: Elementos combinacionais e elementos de estado.

O bloco de registradores (*register bank*) contém 32 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 está ativo – que é a porta de escrita C. A Figura 12.2 mostra a interface do banco de registradores e das memórias. O registrador 0 é *sempre* zero, e escritas neste registrador são ignoradas.

Os registradores cujos conteúdos são mostrados nas portas de leitura são selecionados por dois endereços de 5 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 5 bits ($\log(32) = 5$). 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 ;
```

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:
`instr <= MI(ender);`

A *memória de dados* (MD) tem um comportamento similar ao banco de registradores. A porta de leitura L 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 L:
`dado <= MD(ender);`

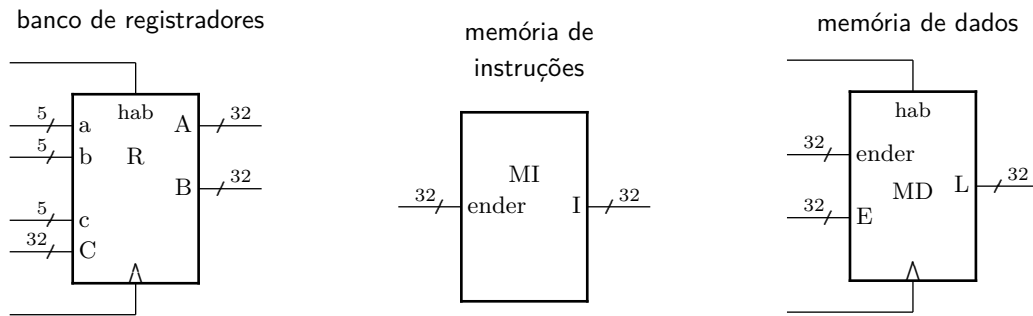


Figura 12.2: Banco de registradores, memória de instruções e de dados.

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

```
if (rising_edge(clk) and hab=1) then MD(ender) <= dado end if;
```

12.1.1 Codificação das instruções

Antes que possamos falar da implementação do processador é necessário examinarmos a codificação das instruções do MIPS32. Todas as instruções tem 32 bits e são codificadas em um dos três formatos: Tipo R, Tipo I e Tipo J, mostrados na Figura 12.3.

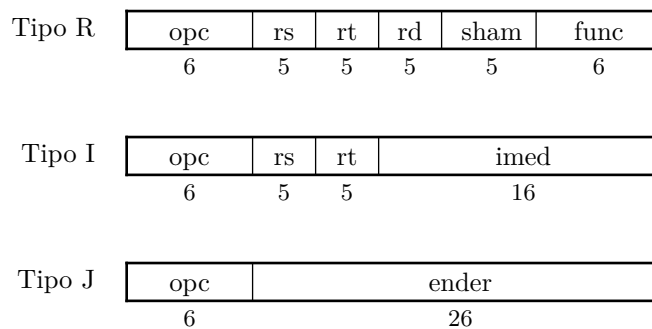


Figura 12.3: Formatos das instruções.

Nos três formatos, o campo *opcode* (opc), abreviatura para *operation code*, é o ‘nome’ da instrução, e em todas elas este campo tem 6 bits. Os campos rs (*register source*), rt (*register target*) e rd (*register destination*) endereçam os registradores com os operandos da instrução. O campo sham (*shift amount*) é um campo de 5 bits que indica o número de posições por deslocar nas instruções de deslocamento. O campo func (*function*) determina a operação da ULA nas instruções de lógica e aritmética quando $opc=0$. O campo imed (*immediate*) é uma constante de 16 bits, que é representada em complemento de dois sempre que isso faça sentido. Finalmente, ender é um endereço de 26 bits.

Na Tabela 12.1, a instrução **addu** é do Tipo R porque a instrução tem seus três operandos em registradores; as instruções **ori**, **lw**, **sw** e **beq** são do tipo I porque todas tem dois registradores como operandos e o terceiro operando é uma constante que faz parte da instrução. As instruções do tipo J são as instruções de saltos, **j** e **jal**.

Esta codificação tem uma característica muito importante que é a sua *regularidade*: todas as instruções tem o mesmo tamanho; todas as instruções tem o *opcode* no mesmo lugar e com

o mesmo tamanho (6 bits), a maioria das instruções tem seus operandos identificados nos campos *rs* e *rt*; a maioria das constantes tem 16 bits (*imed*), e pode ser interpretada como uma constante lógica ou como um inteiro.

Os projetistas do conjunto de instruções reservaram 6 bits para $2^6 = 64$ *opcodes*. Este número é suficiente para codificar todas as instruções do processador? A resposta é *não*, e para aumentar o número de *opcodes* foi usado um truque: quando *opc*=0, a função da ULA é determinada pelo campo *func*, também com 6 bits. Assim, o total de *opcodes* disponíveis é $(2^6 - 1) + 2^6 = 127$.

Como a atenta leitora pode verificar em [MIP05b], a realidade é ligeiramente mais complexa do que o exposto aqui. Por ora, nos interessa mais estudar os princípios de projeto do que as poucas idiosincrasias de um conjunto de instruções real e bem projetado.

A decodificação das instruções é extremamente simples, graças à codificação simples: basta indexar uma tabela de 64 elementos que contém todos os sinais de controle do processador. O *opcode* indexa a tabela e os sinais de controle de todos os componentes do processador são definidos pelos campos dessa tabela – logo veremos alguns exemplos.

12.2 Implementação com relógio de ciclo longo

Vamos, pois, à implementação das instruções da Tabela 12.1. No que segue, os diagramas mostrados estão incompletos para simplificar a explanação. Na Seção 12.2.8 o circuito completo do processador é apresentado.

12.2.1 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 estudaremos a carga de programas para execução. A MI é indexada pelo *program counter* (PC) e este registrador é incrementado de 4 a cada ciclo do relógio. A Figura 12.4 mostra um diagrama com o circuito que efetua a busca e a decodificação das instruções. O PC é incrementado de 4 em 4 porque a memória é indexada byte a byte mas as instruções ocupam uma palavra de 4 bytes.

Uma vez que a saída do PC estabilize após a borda do relógio, e decorrido o tempo de acesso da memória, 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*, o que ativa todos os sinais de controle para a instrução recém buscada.

A largura dos sinais é indicada no diagrama na Figura 12.4 com um traço diagonal e pelo número de bits daquele sinal. O *opcode* tem 6 bits de largura, a instrução *I* e o próximo PC tem 32 bits. As setas indicam a ‘direção’ dos sinais – de saídas para entradas.

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

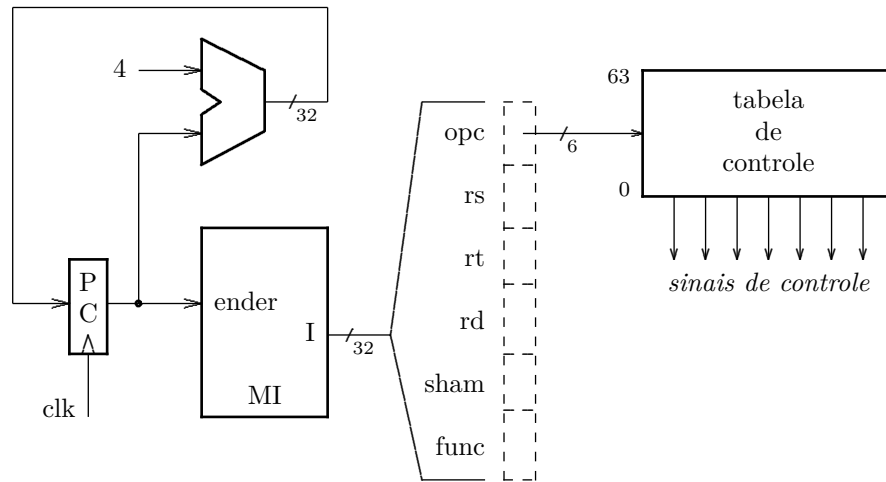


Figura 12.4: Circuito de busca e decodificação de instruções.

12.2.2 Operações de lógica e aritmética

As instruções de lógica e aritmética com formato R usam dois registradores como fonte dos operandos e um terceiro registrador como destino para o resultado. A Figura 12.5 mostra uma instrução **addu** no topo da figura e as ligações necessárias. Os campos da instrução *rs* e *rt* indexam os registradores e seus conteúdos são apresentados nas portas α e β da ULA. O *opcode* desta instrução é zero e portanto o campo *func* é usado para determinar a operação que a ULA efetua em seus operandos, que neste caso é uma soma.

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 *rd*, e o sinal *escReg* é ativado. Na borda ascendente do relógio a soma de $R(rs)$ com $R(rt)$ é armazenada em $R(rd)$.

$$\text{addu rd, rs, rt} \quad \# R(rd) \leftarrow R(rs) + R(rt)$$

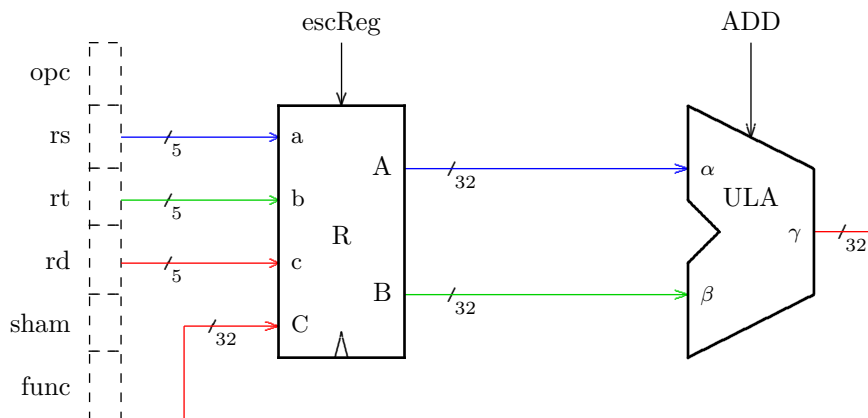


Figura 12.5: Operações de lógica e aritmética – addu.

12.2.3 Operações de lógica e aritmética com imediato

As instruções de lógica e aritmética com um operando imediato usam a constante que é parte da instrução como um dos operandos, e o outro operando é o conteúdo de um registrador, apontado por *rs*. O registrador de destino é apontado por *rt*. O operando da porta B do banco de registradores não é utilizado nesta instrução e portanto as ligações de *b* e *B* não são mostradas. A Figura 12.6 mostra o circuito para executar as instruções com um operando imediato.

A constante é codificada em 16 bits, no campo *im16*, e ela deve ser estendida para 32 bits para que possa ser aplicada à entrada β da ULA. Se a operação é aritmética (**addi** ou **addiu**), então a constante é estendida para 32 bits pela replicação do seu bit de sinal. Se a operação é lógica (**ori** ou **andi**), a constante é estendida com 16 zeros.

```
ori rt, rs, im16    # R(rt) ← R(rs) OR extZero(im16)
```

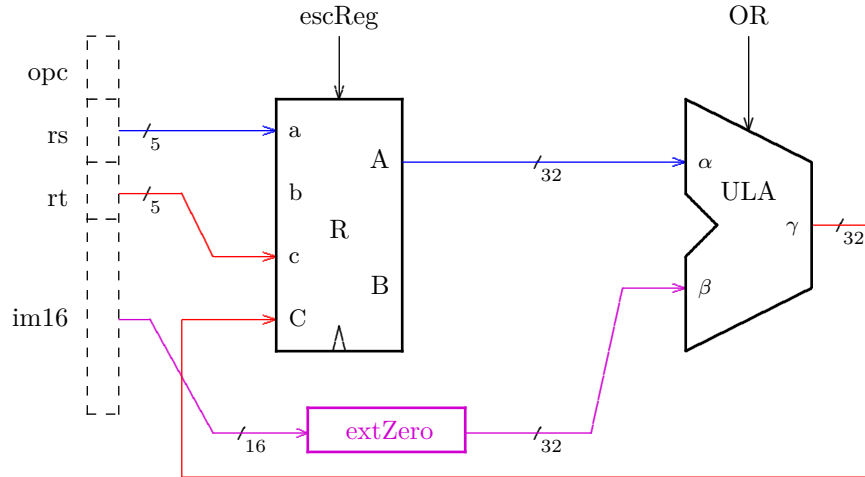


Figura 12.6: Operações de lógica e aritmética com imediato – *ori*.

O operando apontado por *rs* é apresentado à entrada α da ULA; o segundo operando (*im16*) é estendido para 32 bits e apresentado à entrada β da ULA. A operação da ULA é definida como OR e o resultado é armazenado no registrador apontado por *rt*, na borda de subida do relógio, e *escReg* deve ser 1.

As operações aritméticas são similares, exceto que a constante é estendida com sinal ao invés de com zeros.

12.2.4 Operação de acesso à memória – leitura

A instrução **lw** (*load-word*) copia, para o registrador apontado por *rt*, o conteúdo da palavra indexada pela soma de um deslocamento de 16 bits (*des16*) com o conteúdo do registrador apontado por *rs*. A soma da constante com um registrador é chamada de *endereço efetivo*.

A constante é estendida com sinal para permitir deslocamentos positivos e negativos com relação ao endereço apontado por *R(rs)*. 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 *L* contém a cópia do valor lido da memória. Na borda do relógio esse valor é armazenado no registrador apontado por *rt*.

A Figura 12.7 mostra o diagrama do processador com as ligações para a instrução **lw**.

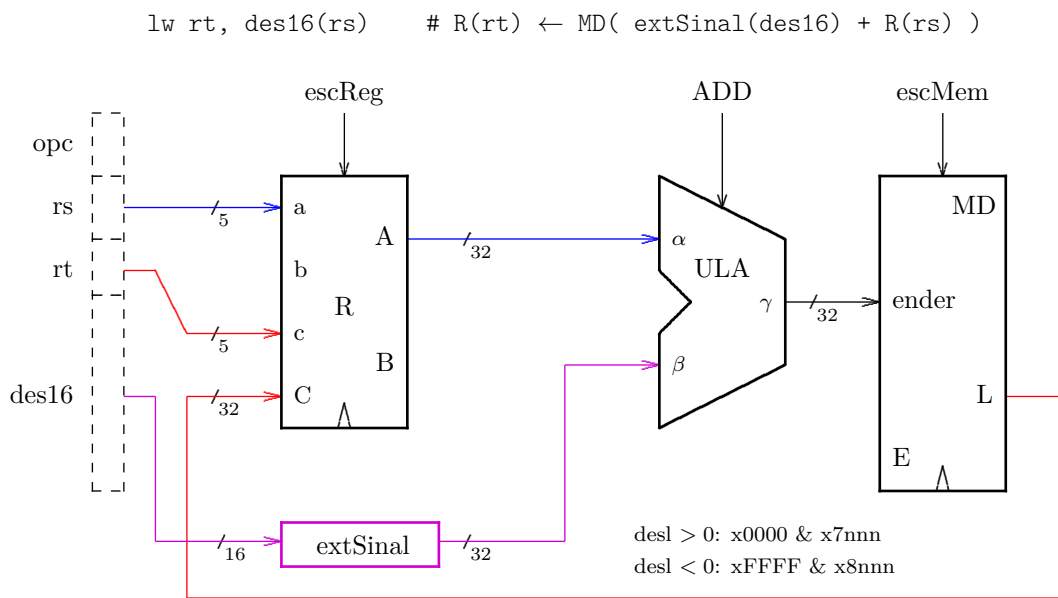


Figura 12.7: Operação de leitura da memória – lw.

12.2.5 Operação de acesso à memória – escrita

A instrução **sw** (*store-word*) copia o conteúdo do registrador apontado por *rt* para a palavra indexada pela soma de um deslocamento de 16 bits (*des16*) com o conteúdo do registrador apontado por *rs*.

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

A Figura 12.8 mostra o diagrama do processador com as ligações para a instrução **sw**.

$$sw\ rt,\ des16(rs)\quad \# MD(\ extSinal(des16) + R(rs)) \leftarrow R(rt)$$

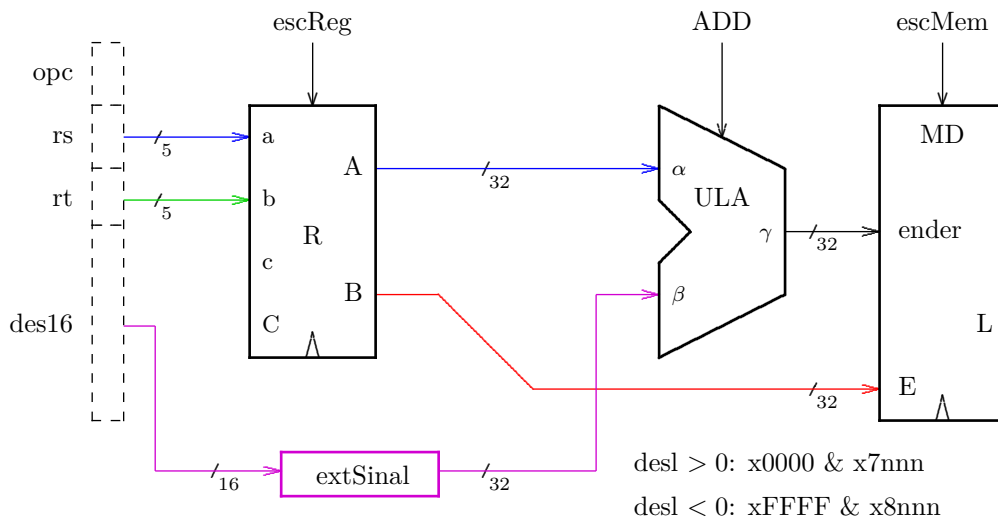


Figura 12.8: Operação de escrita em memória – *sw*.

12.2.6 Desvio condicional

Num desvio condicional (**beq**), o endereço da próxima instrução a ser buscada depende de uma comparação de igualdade. Para tanto, os dois valores por comparar são subtraídos e se o resultado for zero, então eles são iguais. A Figura 12.9 mostra o diagrama do processador com as ligações para a instrução **beq**. A operação na ULA é definida como uma subtração e o sinal **iguais** indica o resultado da comparação.

```
beq rs, rt, des16    # if (R(rs)=R(rt)) PC <- PC+4 + extSinal(des16)*4
```

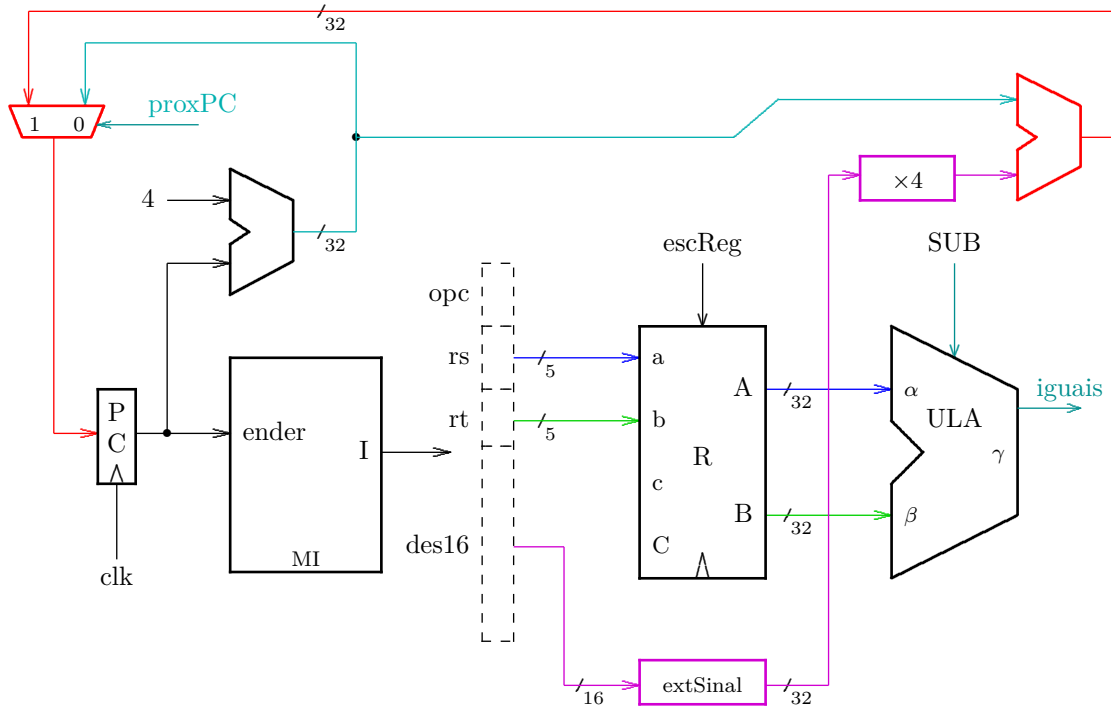


Figura 12.9: Desvio condicional – beq.

O *endereço de destino* de um desvio é obtido pela soma da constante **des16**, estendida para 32 bits com sinal, mais o PC incrementado de quatro. A constante estendida é multiplicada por quatro antes de ser somada ao PC+4. Isso significa que o deslocamento nos desvios é o número de instruções por pular, e não o número de bytes. A multiplicação por quatro é obtida pelo deslocamento de duas posições para a esquerda.

Se **iguais=1** então o endereço de destino é selecionado e aplicado à entrada do PC, e a instrução apontada por aquele endereço será buscada no próximo ciclo. Do contrário, **iguais=0**, PC+4 é aplicado à entrada do PC e a instrução após o **beq** será buscada.

12.2.7 Salto incondicional

Num salto incondicional (*j* para *jump*), o endereço da próxima instrução a ser buscada está codificado na própria instrução. Esta é do Tipo J, com 6 bits para o *opcode* e 26 bits para o endereço de destino.

A especificação do conjunto de instruções determina que as instruções estejam em endereços alinhados, o que significa que os dois bits menos significativos do endereço de destino sejam sempre 00_2 . Assim, os 26 bits da instrução concatenados com os dois 0s compõem um endereço de 28 bits. A faixa de endereços alcançáveis com 28 bits é $2^{28} = 256$ Mbytes, ou 64 M instruções. O folclore nos diz que, na média, um comando em C é implementado com 10 instruções, e portanto uma instrução de salto pode cobrir até 6,4 milhões de linhas de código fonte. Convenhamos que uma ‘perna’ de `if()` com 6 milhões de instruções é assaz incomum, mas não é proibida pela definição da linguagem C e portanto o projetista do *hardware* deve garantir que um salto mais longo que 64 M instruções possa ser usado por programadores néscios.

Para permitir saltos para qualquer distância, os projetistas do MIPS decidiram por usar os 4 bits mais significativos do PC incrementado (PCinc) para completar os 32 bits do endereço. Assim, o endereço de destino de um salto incondicional é

$$\text{PCinc}(31 : 28) \& \text{ender26} \& 00$$

com a concatenação representada por $\&$. Em *hardware*, a concatenação é a mera justaposição dos sinais, e no circuito da Figura 12.10, isso é representado pela “integral quadrada” (*f*), sendo que a parte ‘cortada’ representa o lado mais significativo da justaposição.

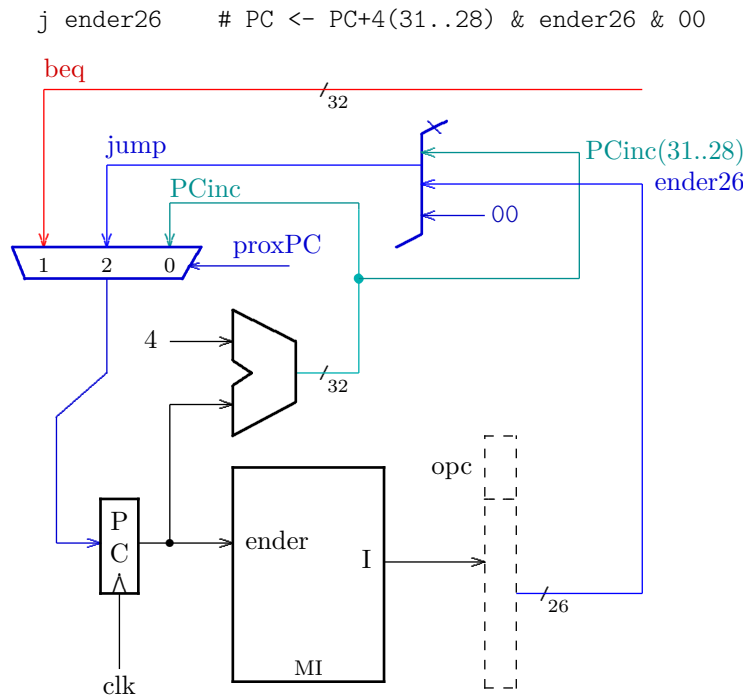


Figura 12.10: Salto incondicional – j.

12.2.8 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 12.11. Para simplificar o diagrama, somente uma parte do circuito para a instrução j é mostrada.

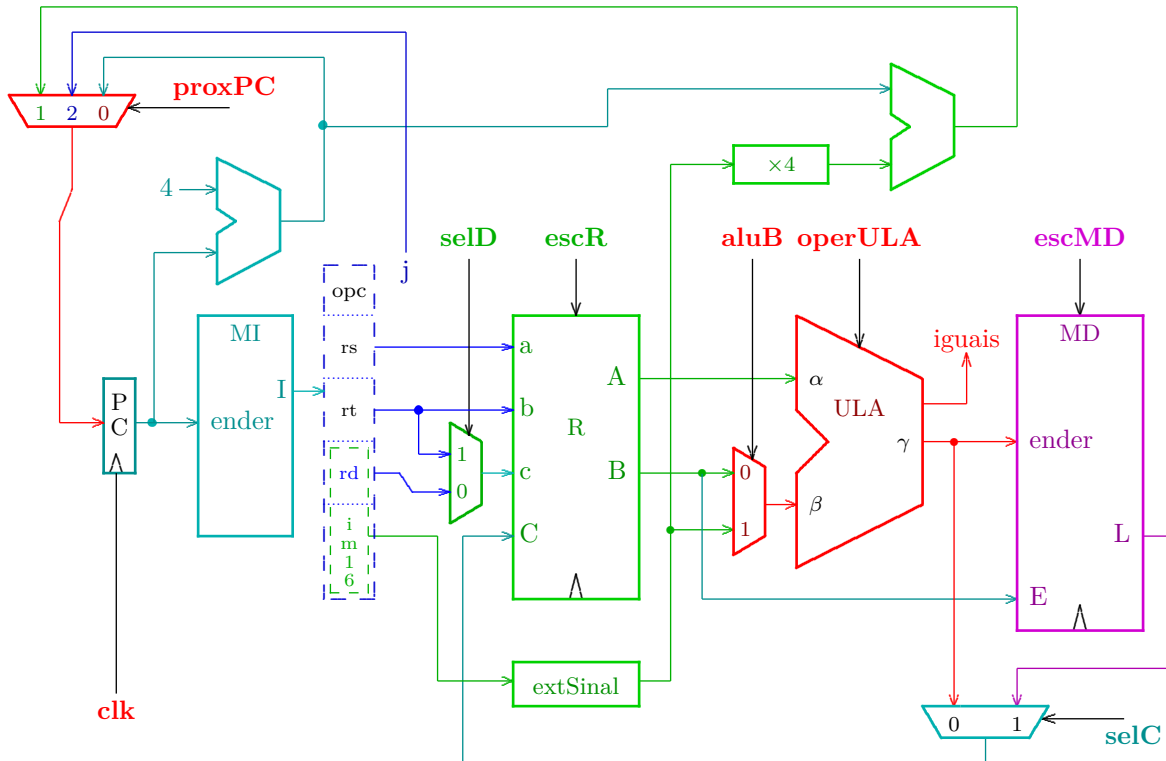


Figura 12.11: Circuito completo do processador.

Nas instruções com imediato, o registrador de destino não pode ser determinado pelo campo rd porque este é uma parte da constante de 16 bits, e por isso o campo rt é usado para determinar o registrador de destino. O registrador de destino, cujo endereço é determinado ora por rd ora por rt , é escolhido por um multiplexador controlado pelo sinal $selD$, cuja saída é ligada à porta c do banco de registradores.

O valor que será gravado no registrador de destino pode ser aquele na saída da ULA, ou aquele lido da memória. Um multiplexador, controlado pelo sinal $selC$, define qual dos dois 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 rs , que é a 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 $aluB$ determina o operando na porta β da ULA.

12.2.9 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 12.12 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.

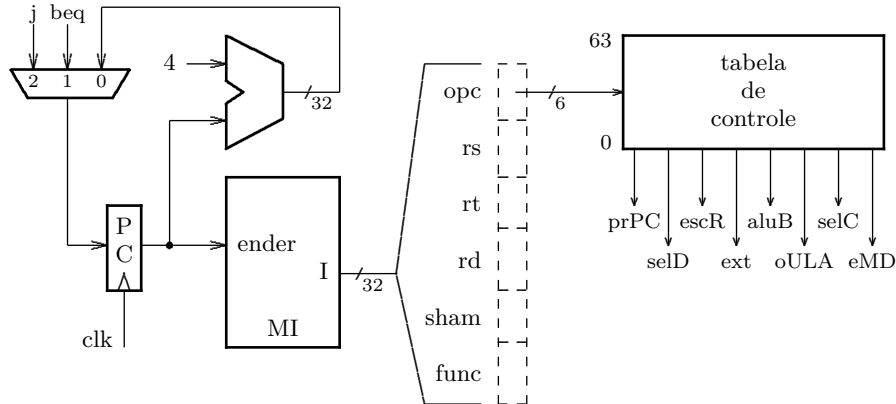


Figura 12.12: Circuito de busca e decodificação com tabela de controle.

A Tabela 12.2 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, *proxPC*, *escR* e *escMD*, devem estar *sempre* determinados porque não há nenhuma instrução que *talvez* altere a memória, ou que *talvez* altere os registradores.

Tabela 12.2: Tabela de controle para as instruções.

INSTRUÇÃO	proxPC	selD	escR	ext	aluB	operULA	selC	escMD
addu	0	0	1	X	0	fun	0	0
ori	0	1	1	zero	1	OR	0	0
lw	0	1	1	sinal	1	ADD	1	0
sw	0	X	0	sinal	1	ADD	X	1
beq	iguais	X	0	sinal	0	SUB	X	0
j	2	X	0	X	X	X	X	0

12.2.10 Metodologia de sincronização

A metodologia de sincronização para esta implementação do conjunto de instruções MIPS32 é chamada de *ciclo longo* porque o período do relógio é longo o bastante para executar uma instrução durante um único ciclo.

Temporização do ADD

A Figura 12.13 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.

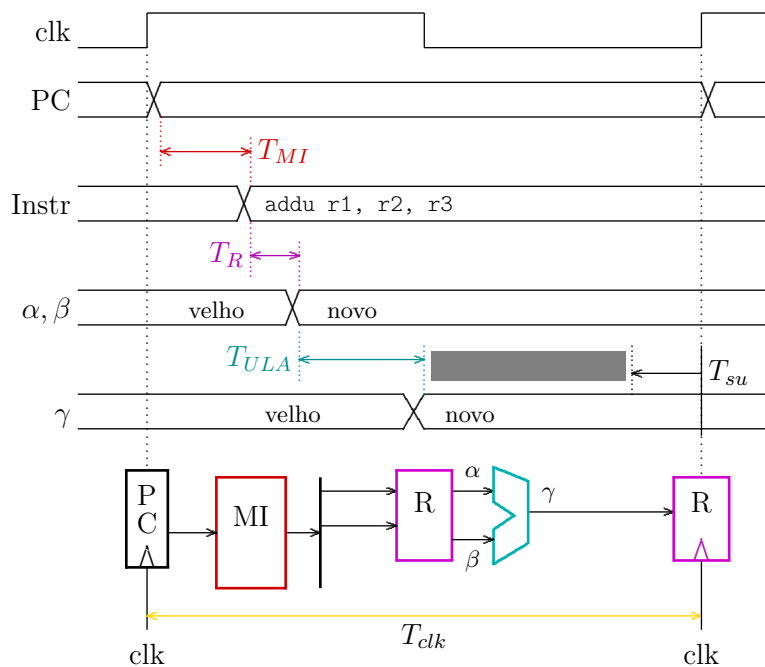


Figura 12.13: Diagrama de tempo de uma operação de ULA.

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.

Temporização do LW

A Figura 12.14 mostra o diagrama de tempo da execução de um **lw**. 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}), e a folga é mostrada pela barra de cor cinza.

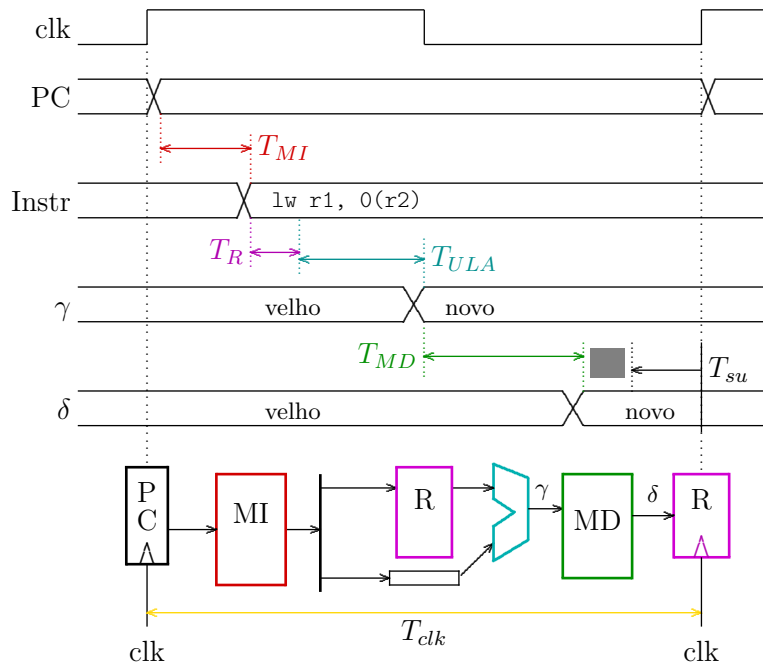


Figura 12.14: Diagrama de tempo de uma leitura em memória.

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

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 **lw**. Por exemplo, na instrução **addu**, 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. Na próxima seção veremos uma implementação mais eficiente do que a do ciclo longo.

Exemplo 12.1 Vejamos como deve ser modificado o processador para acrescentar o circuito de dados para a instrução **BRANCH-AND-LINK** definida abaixo. A instrução **ba1** tem formato I.

ba1 *desl* # R[31] \leftarrow PC+8 , PC \leftarrow (PC+4)+(ext(*desl*) \ll 2)

Esta instrução é uma mistura de uma instrução **ja1** com um “desvio incondicional”: o endereço de ligação é armazenado no registrador *ra* como num **ja1**: R[31] \leftarrow PC+8, e o endereço de destino é computado como numa instrução **beq**: PC \leftarrow (PC+4)+(ext(*desl*) \ll 2).

Quais são as modificações necessárias ao circuito da Figura 12.11?

- (a) é necessário acrescentar uma entrada ao multiplexador controlado pelo sinal *selD*, fixada em 31 para garantir que ao registrador 31 seja atribuído o valor de PC+8;
- (b) deve-se acrescentar um somador para somar 4 ao valor PC+4;
- (c) a saída deste somador deve ser ligada a uma entrada adicional no multiplexador controlado pelo sinal *selC*, para que PC+8 possa ser atribuído ao registrador 31; e
- (d) o sinal (PC+4)+(ext(*desl*) \ll 2) é computado para os desvios, e esse sinal já está ligado à entrada 1 do multiplexador controlado por *proxPC*.

A Figura 12.15 mostra o diagrama de tempos da execução de **ba1**. Assim que o valor no PC estabiliza, decorrido T_{PC} , o incremento de 4 fica disponível após o tempo de propagação do somador da esquerda (T_{add}). O segundo incremento de 4 fica estável após $T_{PC} + 2 \times T_{add}$. A faixa cinza indica que a limitação de *setup* do banco de registradores é respeitada com folga. Decorrido o tempo de acesso à memória de instruções, o endereço de destino fica estável após $T_{PC} + T_{MI} + T_{add}$.

A Tabela 12.3 mostra o estado dos sinais de controle durante a execução de **ba1**. Para fins de comparação, a tabela também mostra os sinais da instrução **beq**. Os sinais que controlam multiplexadores que foram alargados necessitam de dois bits de controle ao invés de um. ◁

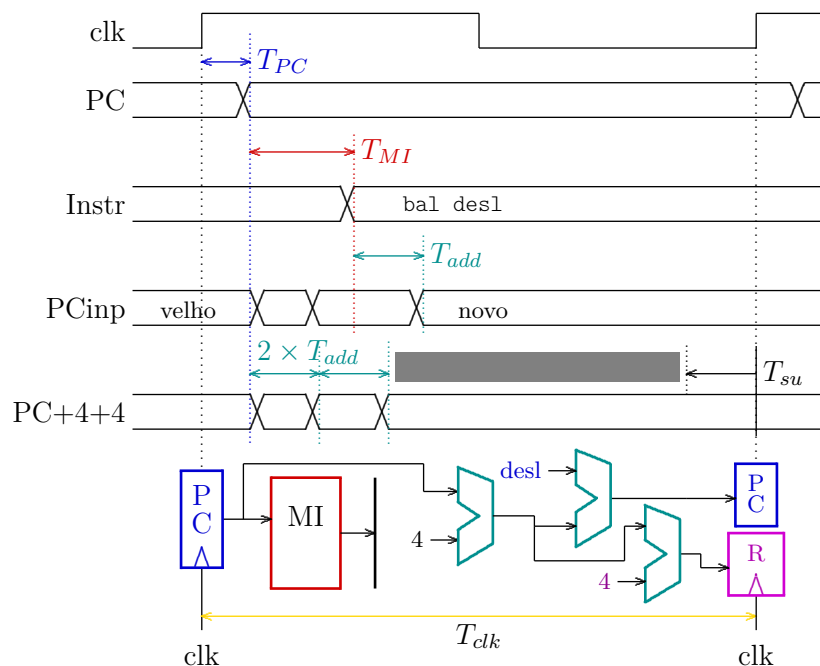


Figura 12.15: Diagrama de tempo da instrução **ba1**.

Tabela 12.3: Tabela de controle para a instrução `bal`.

INSTRUÇÃO	proxPC	selD	escR	ext	aluB	operULA	selC	escMD
<code>beq</code>	iguais	X	0	sinal	0	SUB	X	0
<code>bal</code>	1	2	1	sinal	X	X	2	0

12.2.11 Exercícios

Ex. 12.1 Mostre como implementar as instruções abaixo. Para tanto, indique quaisquer modificações que sejam necessárias no processador da Figura 12.11 e mostre a tabela de sinais de controle ativos durante a execução da instrução. Nos comentários, a vírgula significa “execução simultânea”.

```

jr rs                # jump-register
                    # PC <- R(rs)

jal ender26         # jump-and-link
                    # R(31) <- PC+8 , PC <- PCinc & ender26 & 00

jalr rt,rs         # jump-and-link-register
                    # R(rt) <- PC+8 , PC <- R(rs)

```

Espaço em branco proposital.

12.3 Implementação segmentada

A técnica de *segmentação* (*pipelining*) é usada para aumentar a taxa com que as instruções completam. Suponha que um determinado circuito combinacional tenha tempo de propagação T_p – veja a Figura 12.16. A entrada do circuito é alimentada por um registrador e sua saída é capturada por outro registrador. O período mínimo do relógio para este circuito é $T_{\min} \geq T_p + T_r + T_{su}$, que é a soma do tempo de propagação do circuito ($T_p = n$), do tempo de propagação do registrador de saída (T_r), mais o *setup* do registrador de saída (T_{su}). Para este circuito, a latência é $n + T_r + T_{su}$ segundos, e a vazão é $1/(n + T_r + T_{su})$ resultados por segundo, que pode ser aproximada por $1/n$, porque, no geral, $n \gg (T_r + T_{su})$.

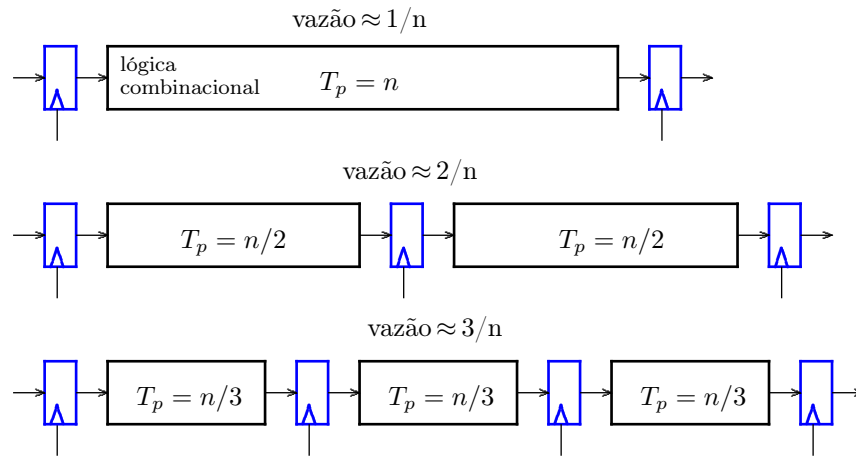


Figura 12.16: Circuito combinacional segmentado.

Se for possível particionar o circuito combinacional em duas metades iguais, o período do relógio pode ser reduzido para a metade se um registrador for inserido entre as duas partes, e neste caso $T'_{\min} \geq n/2 + T_r + T_{su}$, a vazão passa a ser $\approx 2/n$ resultados/s, e a latência permanece em $n + T_r + T_{su}$ s. Se o circuito pode ser particionado em três segmentos, $T''_{\min} \geq n/3 + T_r + T_{su}$, a vazão sobe para $\approx 3/n$ resultados/s enquanto a latência permanece a mesma.

Por “a latência permanece a mesma” entenda-se que o tempo para produzir um resultado não aumenta com a divisão do circuito em dois ou em três segmentos; o que aumenta é a taxa com que novos resultados são produzidos pelo circuito segmentado – a vazão aumenta de forma quase proporcional ao número de segmentos. O ‘quase’ decorre de que dificilmente os circuitos combinacionais possam ser particionados de forma a que o tempo de propagação dos dois segmentos seja exatamente igual, e esse desbalanceamento é a principal causa para que a vazão não seja exatamente proporcional ao número de segmentos.

A analogia mais próxima com a segmentação de um circuito é uma linha de montagem de automóveis, na qual o longo processo de montagem de um carro é subdividido em muitas tarefas simples. Um carro demora algo como 48 horas para ser produzido, mas uma linha de montagem moderna produz um veículo pronto e acabado a cada 15 minutos, dependendo do modelo e da fábrica.

12.3.1 Processador segmentado

A Figura 12.17 mostra o processador de ciclo longo já segmentado em cinco estágios, com registradores de segmento introduzidos nos locais apropriados. A execução de uma instrução é dividida em cinco estágios: (i) busca na memória de instrução e incremento do PC; (ii) decodificação e leitura dos operandos; (iii) execução na ULA; (iv) acesso à memória nos *loads* e *stores*; e (v) gravação do resultado.

Note que o banco de registradores aparece dividido no diagrama: no estágio de decodificação (**decod**), os registradores são lidos, enquanto no estágio de resultado (**result**) o registrador de destino é atualizado.

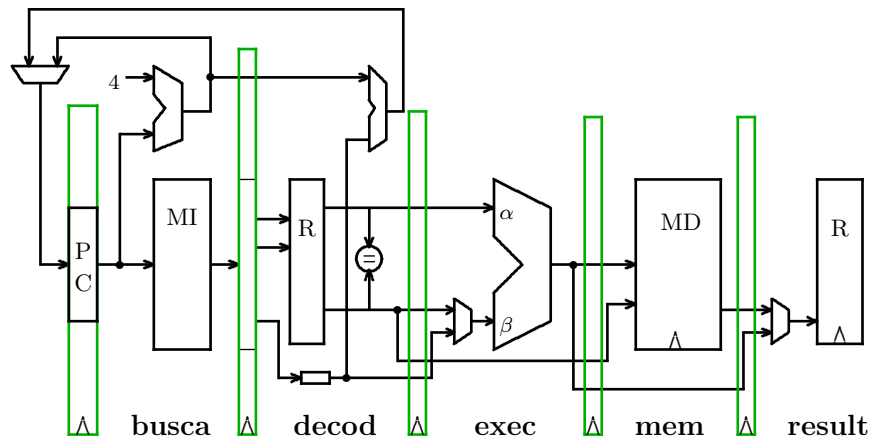


Figura 12.17: Processador segmentado em cinco estágios.

O período do relógio do processador é determinado pelo estágio mais demorado, e este é o estágio de acesso à memória. A latência de uma única instrução é aproximadamente a mesma daquela do processador de ciclo longo, enquanto a vazão, ou a taxa em que uma nova instrução é completada, é aproximadamente cinco vezes maior. A Figura 12.18 mostra dois diagramas de tempo, um do processador de ciclo longo executando duas instruções, e outro do processador segmentado executando três instruções.

Enquanto o processador de ciclo longo emprega 2 ciclos longos para executar um *load* seguido de um *store* – estes dois ciclos longos equivalem a 10 ciclos do processador segmentado, o processador segmentado executaria *seis* instruções neste mesmo intervalo. A primeira instrução completa no ciclo número 5, a segunda no ciclo 6, e a sexta no ciclo 10. Passado o intervalo para encher os segmentos com instruções, uma nova instrução completa a cada ciclo curto. O ganho potencial de desempenho, medido pela taxa de instruções executadas, é de um fator de cinco, que é o número de segmentos.

A Figura 12.19 mostra o quinto ciclo do diagrama de tempos da Figura 12.18: o valor lido da memória pelo **lw** é gravado no registrador de destino; o **sw** escreve na memória de dados; o **add** é computado na ULA; o **xor** é decodificado, r12 e r13 são obtidos do bloco de registradores; o **sub** é buscado da MI e o PC é incrementado de 4.

O ganho de desempenho advém do paralelismo no nível de instrução: cinco instruções executam ao mesmo tempo, cada uma delas num estágio distinto de execução. Os registradores de segmentação separam cada instrução da instrução que a precede, e da instrução que a sucede. A cada borda do relógio as instruções ficam mais perto de completar.

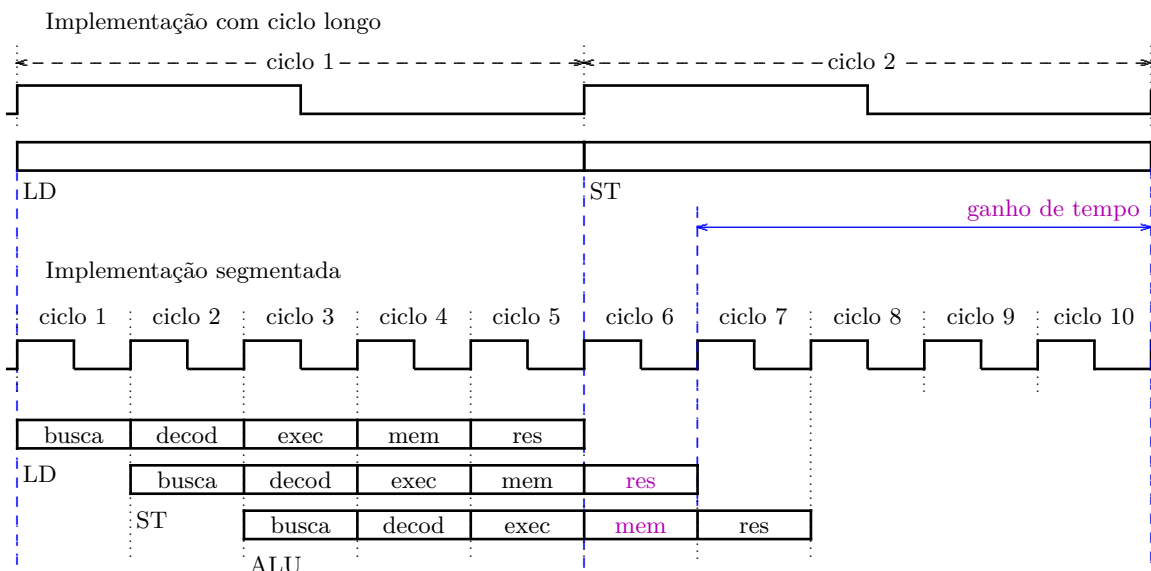


Figura 12.18: Diagrama de tempo do processador de ciclo longo e segmentado.

Compare o diagrama do processador na Figura 12.19 com aquele da Figura 12.11. No processador segmentado, o circuito necessário para as instruções de desvio está todo concentrado em **decod**. Isso é possível com a adição de um comparador de igualdade entre as portas de leitura do banco de registradores para gerar o sinal *iguais*, ao invés de empregar-se a ULA na comparação. Em breve veremos o motivo para resolver os desvios em **decod**.

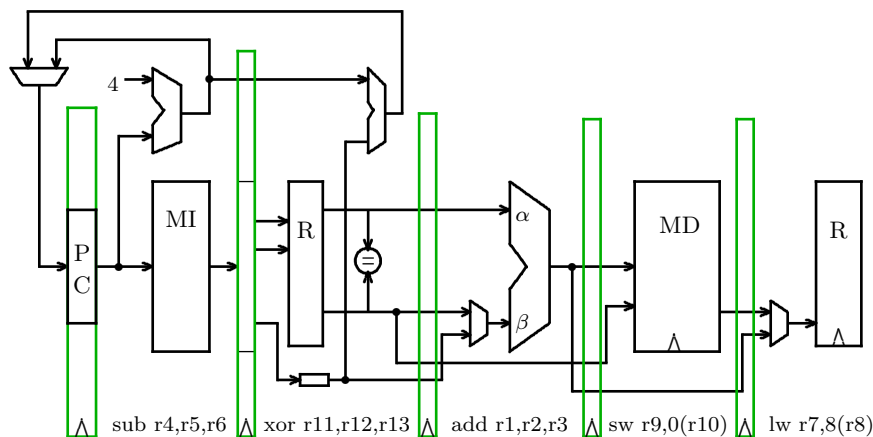


Figura 12.19: Cinco instruções em cinco estágios de execução.

Os registradores de segmento são *invisíveis* ao programador, porque estes registradores não fazem parte do estado da computação como definido pelo conjunto de instruções MIPS32r2. Os *registradores visíveis* são aqueles explicitamente usados como operandos das instruções.

Exemplo 12.2 Vejamos como se dá a execução de um `addu` no processador segmentado. A Figura 12.20 mostra a sequência de execução. O tempo decorre de cima para baixo, ao longo dos cinco ciclos de relógio necessários para completar uma instrução (C_0 a C_4), um ciclo em cada estágio. O registrador “de saída” dos estágios é mostrado em azul, e na próxima borda do relógio, o registrador de saída é carregado com os valores produzidos pelos circuitos combinacionais, alimentados pelo “registrador de entrada” do estágio.

Busca (C_0): o conteúdo do PC indexa a memória de instruções (MI), e no final do ciclo a instrução `addu` é carregada no registrador de saída; em paralelo ao acesso à memória de instruções, o PC é incrementado de 4.

Decod (C_1): a instrução é decodificada – a tabela de controle é indexada com o *opcode*, e o banco de registradores é acessado: $A \leftarrow R(rs)$ e $B \leftarrow R(rt)$. O endereço do registrador de destino ($rd=c$) é transferido para o próximo estágio.

Exec (C_2): às entradas α e β da ULA são atribuídos os conteúdos dos registradores $A = R(rs)$ e $B = R(rt)$, e o resultado é apresentado em γ . O endereço do registrador de destino ($c' \leftarrow c$) é transferido para o próximo estágio.

Mem (C_3): o resultado é transferido sem alteração para o estágio de resultado $\gamma' \leftarrow \gamma$. O endereço do registrador de destino ($c'' \leftarrow c'$) é transferido para o próximo estágio.

Result (C_4): o resultado da operação da ULA ($\gamma'' \leftarrow \gamma'$) é gravado no registrador apontado pelo registrador de destino rd ($c''' \leftarrow c''$): $R(c''') \leftarrow \gamma''$.

O resultado da operação na ULA (γ) é copiado, sem alteração, do registrador de entrada para o registrador de saída no estágio de memória ($\gamma' \leftarrow \gamma$), e este valor ($\gamma'' \leftarrow \gamma'$) é selecionado pelo multiplexador no estágio de resultado, e atribuído ao registrador de destino: $R(rd) \leftarrow (R(rs) + R(rt))$. \triangleleft

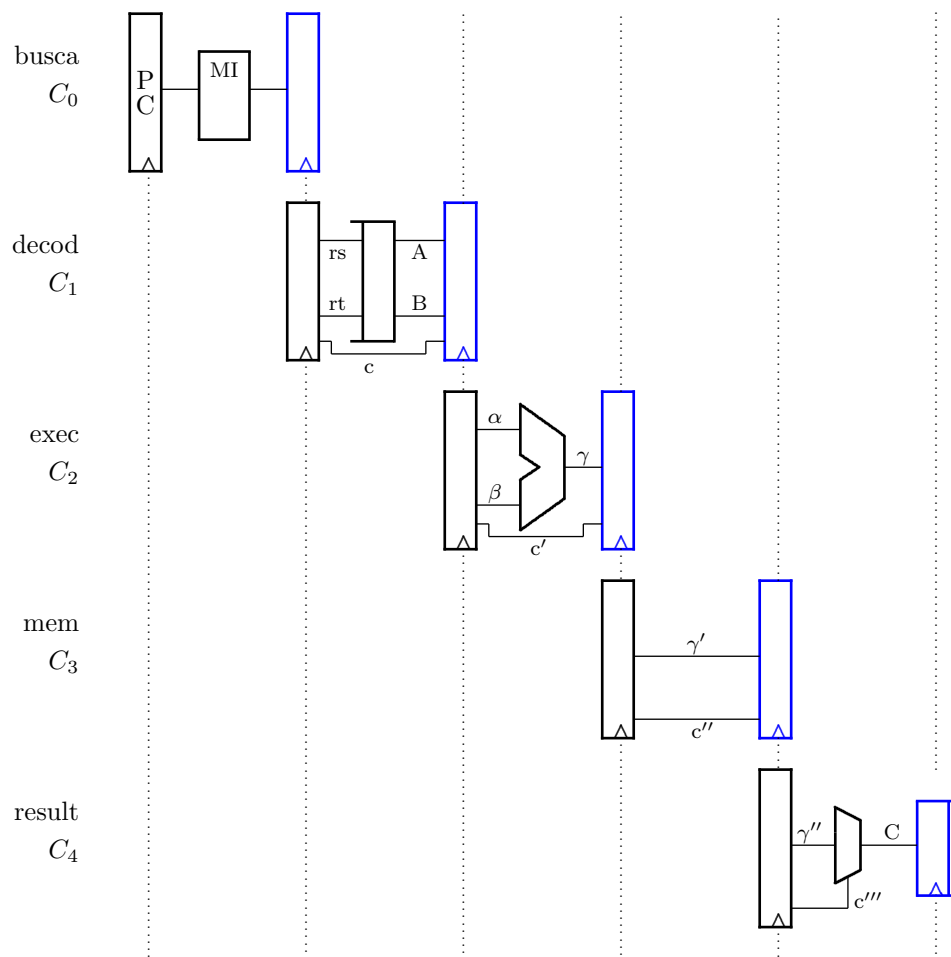


Figura 12.20: Progresso de um addu pelos segmentos.

Espaço em branco proposital.

Exemplo 12.3 Como um segundo exemplo, sigamos a execução de um `lw`. A Figura 12.21 mostra a sequência de execução.

Busca (C_0): o conteúdo do PC indexa a memória de instruções (MI), e no final do ciclo a instrução `lw` é carregada no registrador de saída; em paralelo ao acesso à memória de instruções, o PC é incrementado de 4.

Decod (C_1): a instrução é decodificada – a tabela de controle é indexada com o *opcode*, o banco de registradores é acessado ($A \leftarrow R(rs)$), e o deslocamento é estendido com o sinal ($D \leftarrow extS(desl)$). O endereço do registrador de destino ($rt=c$) é transferido para o próximo estágio.

Exec (C_2): às entradas α e β da ULA são atribuídos os conteúdos dos registradores ($\alpha \leftarrow A$) e do deslocamento ($\beta \leftarrow D$), e o endereço efetivo é apresentado em γ . O endereço do registrador de destino ($c' \leftarrow c$) é transferido para o próximo estágio.

Mem (C_3): o endereço efetivo é aplicado à entrada de endereço da memória de dados (MD), e o conteúdo indexado é atribuído para o registrador de saída ($\delta \leftarrow MD(\gamma')$). O endereço do registrador de destino ($c'' \leftarrow c'$) é transferido para o próximo estágio.

Result (C_4): o valor obtido da memória ($\delta' \leftarrow \delta$) é gravado no registrador apontado pelo registrador de destino ($c''' \leftarrow c''$): $R(rt) \leftarrow MD(R(rs) + extS(desl))$. ◁

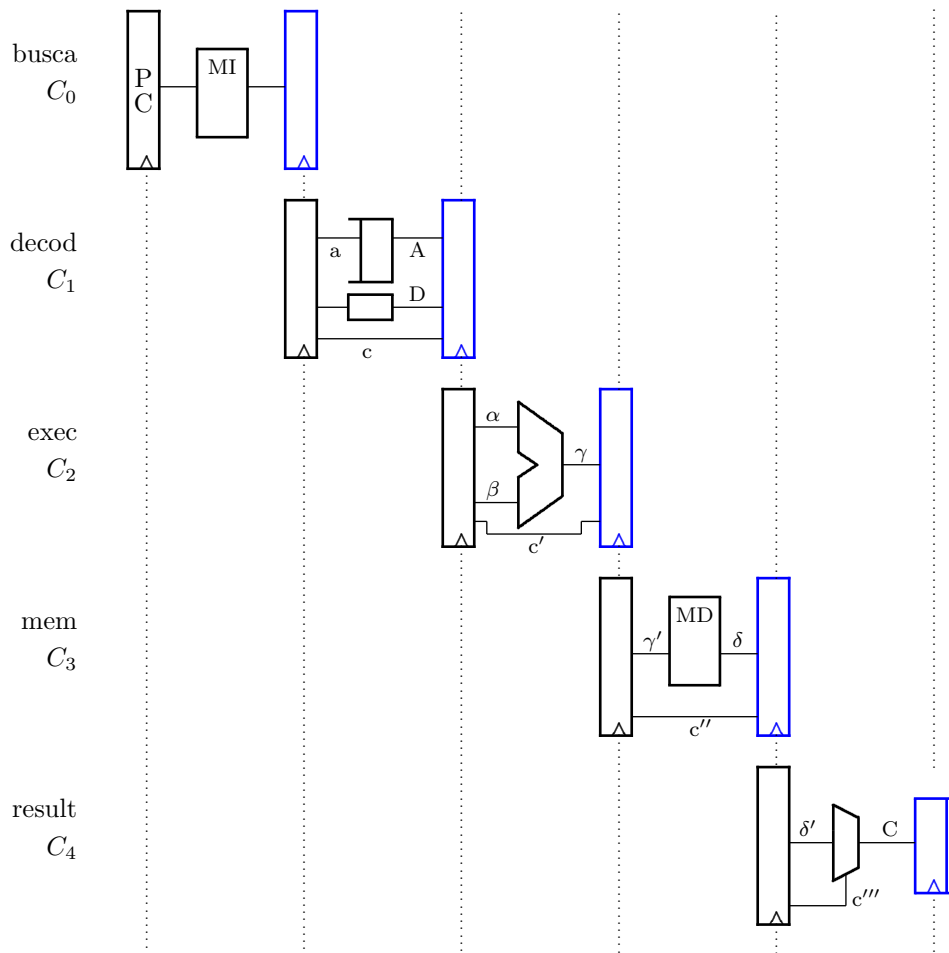


Figura 12.21: Progresso de um `lw` pelos segmentos.

Exemplo 12.4 A instrução `BRANCH-AND-LINK` é similar a um `JUMP-AND-LINK`, exceto que o modo de endereçamento é o mesmo daquele de um desvio condicional: o endereço de destino é um deslocamento com relação ao PC. A especificação da instrução `bal` é:

$$\text{bal des1} \quad \# R(31) \leftarrow PC+8, \quad PC \leftarrow (PC+4) + (\text{ext}(\text{des}) \ll 2)$$

Esta instrução tem formato do Tipo I. O circuito para executar a instrução `BRANCH-AND-LINK` é mostrado na Figura 12.22 – neste diagrama o banco de registradores *não* está separado em uma seção de leitura e outra de escrita.

O endereço de destino independe de qualquer comparação, e o circuito para computá-lo é o mesmo que para as instruções `beq`: a constante é estendida para 32 bits, multiplicada por 4 e então adicionada ao PC+4.

O endereço de retorno pode ser computado no estágio de execução ((PC+4)+4), e deve ser levado até o estágio de resultado, para então ser armazenado no registrador 31. Para tanto, o multiplexador no estágio de resultado ganha uma nova entrada (PC+8), assim como multiplexador do registrador de destino ganha uma entrada fixa em 31.

O endereço de retorno deve ser PC+8 porque este é o endereço da primeira instrução que está além do *branch delay slot* – este assunto deve esperar até a Seção 12.3.2. ◁

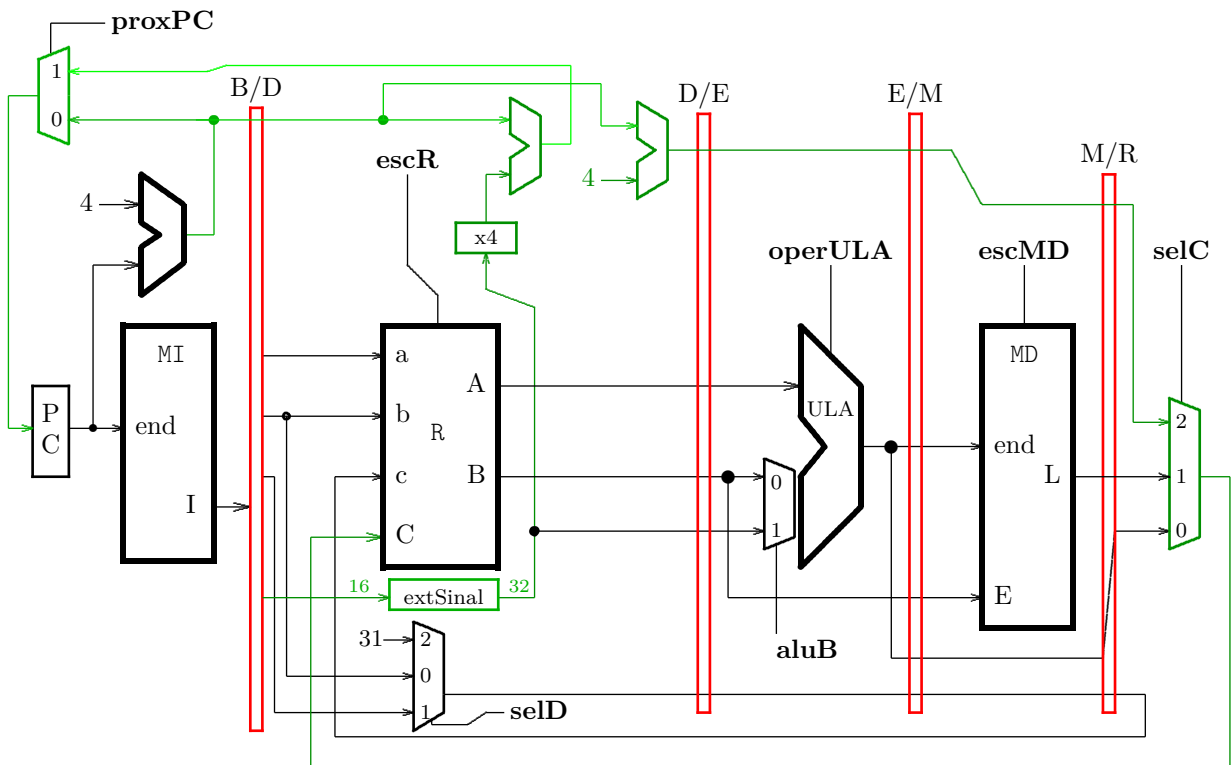


Figura 12.22: Processador com suporte à instrução `bal`.

12.3.2 O ganho é menor que o número de estágios

Ao contrário dos automóveis numa linha de montagem, as instruções de um programa dependem de outras instruções que se encontram adiante na “linha de instruções”, e isso tem impactos significativos no desempenho de processadores segmentados. Nos interessam três situações em particular: dependências de dados, desvios e *loads*.

Usaremos um diagrama de blocos simplificado do processador para examinarmos a execução das instruções através dos segmentos. A Figura 12.23 mostra o modelo simplificado do processador.

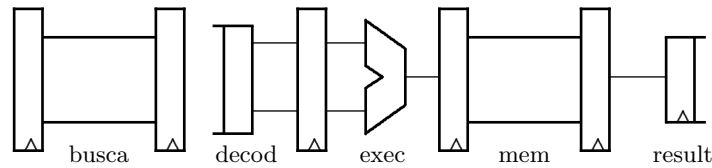


Figura 12.23: Modelo simplificado do processador.

Dependências de dados

Uma das formas de dependência entre instruções é chamada de *dependência de dados* e o nome decorre do fluxo de dados que existe entre duas ou mais instruções. Os dados fluem de uma instrução produtora para instruções consumidoras através dos registradores visíveis ao programador. As instruções consumidoras são chamadas de *instruções dependentes*.

Considere o trecho de programa abaixo, no qual o valor computado pelo **add** em r1 é usado como operando pelo **sub**. O **sub** *depende* do **add** para computar a diferença entre r2 e r1.

```
add r1, r6, r7
sub r3, r2, r1
```

O diagrama do processador na Figura 12.24 mostra o ciclo em que o **add** está no estágio de execução e o **sub** está obtendo seus operandos do banco de registradores. O valor contido em r1, no banco de registradores, é um valor caduco que foi produzido por alguma instrução anterior, e não a soma de r6 com r7, que é o desejado pelo programador.

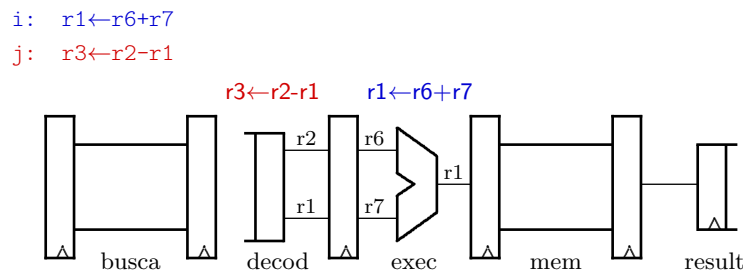


Figura 12.24: Dependência de dados.

A implementação do processador segmentado cria um *risco*: se esta seqüência for executada, existe o risco – neste caso uma certeza – de o programa produzir resultados errados. Esta distinção é importante: as dependências são introduzidas no programa pelo programador; os riscos são introduzidos pelo projetista do processador. Se a implementação pode violar as

dependências entre instruções, então os projetistas do *hardware* e do *software* devem prover os meios para a execução correta dos programas.

No caso acima a solução é simples porém ineficiente: basta afastar a instrução dependente da instrução produtora para garantir que seu resultado seja depositado no banco de registradores *antes* que este valor seja lido pela instrução dependente. No diagrama da Figura 12.25, a instrução dependente está separada da produtora por dois **nops**, o que garante que o valor usado pelo **sub** será aquele produzido pelo **add**.

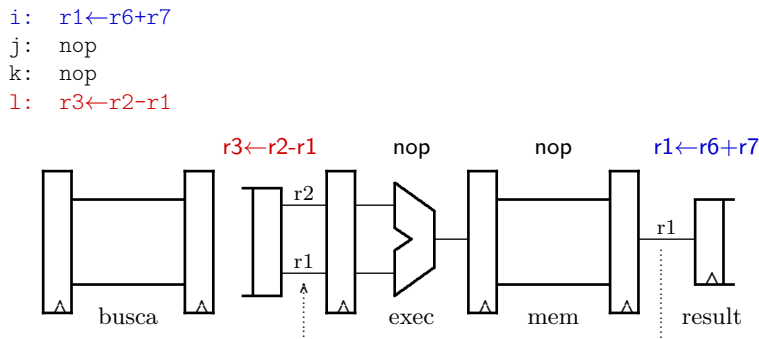


Figura 12.25: Dependência de dados resolvida com dois nops.

A corretude é obtida ao custo de dois ciclos desperdiçados por causa da dependência de dados. Se o compilador reordenar o código, é possível substituir um, ou os dois, **nops** por instruções que efetuem trabalho útil e assim minorar a queda de desempenho causada pela dependência.

Exemplo 12.5 Abaixo estão dois trechos de código que computam o mesmo resultado. No lado esquerdo, o código ignora as dependências de dados entre as instruções. No lado direito está a versão que executaria corretamente no processador da Figura 12.23. Verifique se todas as dependências de dados estão resolvidas com os **nops** que foram adicionados para garantir execução correta.

<pre> # riscos por resolver add r5, r6, r7 sub r8, r9, r5 # r5 addi r2, r8, 9 # r8 xor r10, r8, r5 # r8,r5 add r5, r10, r2 # r10,r2 </pre>	<pre> # riscos resolvidos add r5, r6, r7 nop nop sub r8, r9, r5 nop nop addi r2, r8, 9 xor r10, r8, r5 nop add r5, r1, r2 </pre>
--	--

A resolução dos riscos neste trecho de código custa cinco ciclos adicionais, o que reduz a vazão em 50% neste trecho de código. Felizmente, é possível minimizar as perdas causadas por dependências de dados. <

Existem duas técnicas de *hardware* para minimizar o risco causado pela dependência de dados. Na mais simples, o controle do processador detecta a dependência, ao comparar os números dos registradores de resultado nos estágios **exec** e **mem** com os operandos das instruções em **decod**. Uma vez detectada a dependência, a instrução que está em **decod** fica bloqueada até que a instrução produtora chegue a **result**, quando então o valor atualizado é obtido do banco

de registradores. Há desperdício de tempo, mas o compilador não é obrigado a inflar o código com a inserção dos **nops**. Os ciclos perdidos por causa dos bloqueios são chamados de *bolhas*.

Exemplo 12.6 Vejamos uma parte do circuito que computa as dependências de dados entre as instruções. Este circuito é geralmente implementado no estágio de decodificação, e é usado para decidir se a instrução que está sendo decodificada deve esperar até que seus operandos estejam disponíveis no banco de registradores.

O número do registrador de destino deve ser transportado com a instrução, na medida em que esta progride pelos estágios, para que seja possível ‘casar’ o resultado com o registrador de destino. Chamemos este registrador de *a_c*, e cada um dos registradores de segmento possui um campo *a_c*, *viz.* EXE.a_c, MEM.a_c, e RES.a_c.

O circuito que detecta uma dependência no primeiro operando de uma instrução, DEC.rs pode ser modelado em VHDL como:

```

if ( DEC.rs = EXE.a_c ) and
      ( EXE.a_c != 0 )      and
      ( EXE.regWR = true )
then
    segura_em_decod <= true;
end if ;

```

Se a instrução no estágio de execução escreve num registrador (regWR) e este registrador não é o registrador r0, e o registrador de destino daquela instrução é o operando da instrução que está sendo decodificada, então esta instrução deve permanecer no estágio **decod** (segura_em_decod), para que a instrução produtora avance até o estágio **result**, quando esta gravará o seu resultado no registrador de destino.

Um teste similar deve ser efetuado para a instrução que está em **mem**:

```

if ( ( ( DEC.rs = EXE.a_c ) and
        ( EXE.a_c != 0 )      and
        ( EXE.regWR = true ) )
      or
      ( ( DEC.rs = MEM.a_c ) and
        ( MEM.a_c != 0 )      and
        ( MEM.regWR = true ) ) )
then
    segura_em_decod <= true;
end if ;

```

Se a instrução no estágio de memória produz um operando para a instrução em **decod**, então esta deve esperar até que aquela chegue em **result**.

Estes testes, com as adaptações necessários, devem ser efetuados para o segundo operando das instruções, que é apontado pelo campo *rt*. ◁

A técnica mais eficiente, e com maior complexidade, consiste em adiantar os resultados das instruções produtoras para a instrução dependente. O *adiantamento* de resultados pode eliminar todas as dependências de dados, com excessão do uso do *load*.

Um multiplexador é adicionado a cada entrada da ULA e ligações adiantam os valores que estão disponíveis nos registradores de segmentos nos estágios adiante da instrução que está em **exec**. O circuito de detecção de dependências é usado para acionar as entradas dos multiplexadores

de forma a adiantar os valores dos registradores de segmento diretamente às entradas da ULA. A Figura 12.26 mostra o circuito de adiantamento.

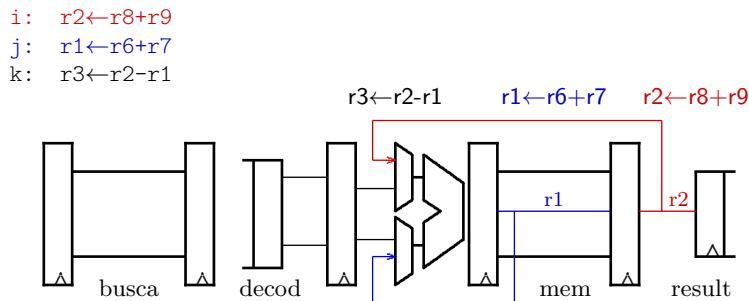


Figura 12.26: Dependência de dados resolvida com adiantamento.

Com adiantamento evitam-se as bolhas e o processador pode executar instruções na máxima taxa possível, completando uma instrução por ciclo. Infelizmente, *loads* e desvios são problemáticos, como mostra a discussão que se segue.

Uso do *load*

A Figura 12.27 mostra um diagrama do processador quando executa um *lw* seguido de um *add*. O resultado do *lw* $r1, 100(r6)$ é gravado no registrador de segmento do estágio *mem* somente no final do ciclo em que a leitura da memória é concluída, porque o circuito de memória é o componente mais lento do processador.

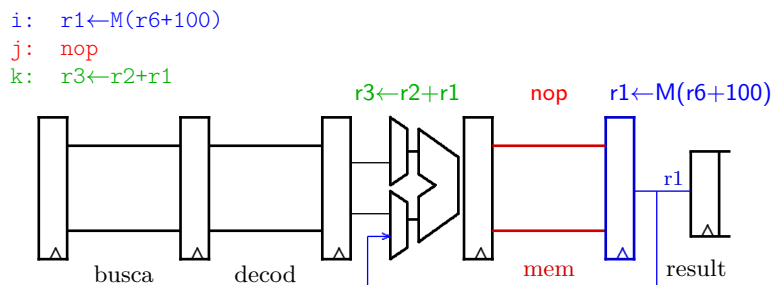


Figura 12.27: Bolha introduzida pelo *load delay-slot*.

Neste mesmo ciclo, a instrução *add* $r3, r2, r1$, que segue o *lw*, não pode usar o valor adiantado do registrador de estágio *exec* porque este valor não é o resultado obtido da leitura da memória. Para evitar o erro decorrente da leitura do valor caduco que está no registrador do segmento *exec*, o *add* deve ser mantido em *exec* por um ciclo, para garantir que o valor no registrador *mem* seja adiantado para a entrada da ULA, como indicado pela seta que adianta $r1$, na Figura 12.27.

O compilador deve inserir um *nop* entre todos os pares de instruções com um *uso do load* para garantir que a instrução dependente do *load* obtenha o valor correto para seu operando. Esta instrução *nop*, ou uma bolha, é necessária para garantir corretude mas é um desperdício de tempo. Se o compilador consegue reordenar o código de forma a preencher o *load delay-slot* com uma instrução útil, então não há desperdício de um ciclo a cada *load*. Evidentemente, a instrução que preenche o *delay-slot* não pode alterar a corretude do programa.

Exemplo 12.7 Os trechos de código abaixo não computam o mesmo resultado. No lado esquerdo, o código não considera o uso do *load*. No lado direito está a versão que executaria corretamente no processador da Figura 12.27.

```
# riscos por resolver                                # riscos resolvidos
  li    r1, 0                                         li    r1, 0
  li    r2, 1024                                       li   r2, 1024
  la    r3, A                                           la    r3, A
  li    r4, 0                                         li    r4, 0

for:  lw   r5, 0(r3)                                    for:  lw   r5, 0(r3)
      add  r4, r4, r5                                  nop
      addi r3, r3, 4                                  add   r4, r4, r5
      addi r1, r1, 4                                  addi  r3, r3, 4
      bne  r1, r2. for                                addi  r1, r1, 4
fim:                                                                    nop
                                                                    nop
                                                                    bne  r1, r2. for
                                                                    fim:
```

Há uma dependência no registrador *r1*, entre o segundo *addi* e o desvio *beq*. Esta dependência é resolvida com a introdução de dois *nops*.

A resolução dos riscos neste trecho de código custa três ciclos adicionais a cada volta do laço, o que reduz a vazão em $9/6=1,5$. É possível reordenar as instruções para eliminar um ou mais *nops*? Se sim, quantos? A reordenação deve manter a corretude do código. ◀

Dependências de controle

As instruções de saltos e desvios causam o seu próprio *delay-slot*. A instrução de desvio só é decodificada, e o destino do desvio determinado, ao final do estágio de decodificação. Isso significa que a instrução que será executada após o desvio, caso este seja tomado, só será conhecida um ciclo após a determinação do destino. No processador segmentado, a instrução que segue o desvio é buscada antes que o destino do desvio seja conhecido. A Figura 12.28 mostra uma instrução de desvio no estágio de decodificação e a instrução que a segue, um *nop*.

Os projetistas do MIPS transformaram o limão em limonada e decretaram que a instrução que segue um desvio é sempre executada. A tarefa de preencher o *branch delay-slot* com uma instrução útil recai sobre o compilador. Se não há instrução que possa ser movida para o *delay-slot*, então este deve ser preenchido com um *nop*. Retornaremos ao tópico de reordenamento de código na Seção ??.

Exemplo 12.8 O trecho de código abaixo é uma versão reordenada daquele do Exemplo 12.7, que resolve a dependência de uso do *load* mas não elimina a bolha causada pela dependência de controle.

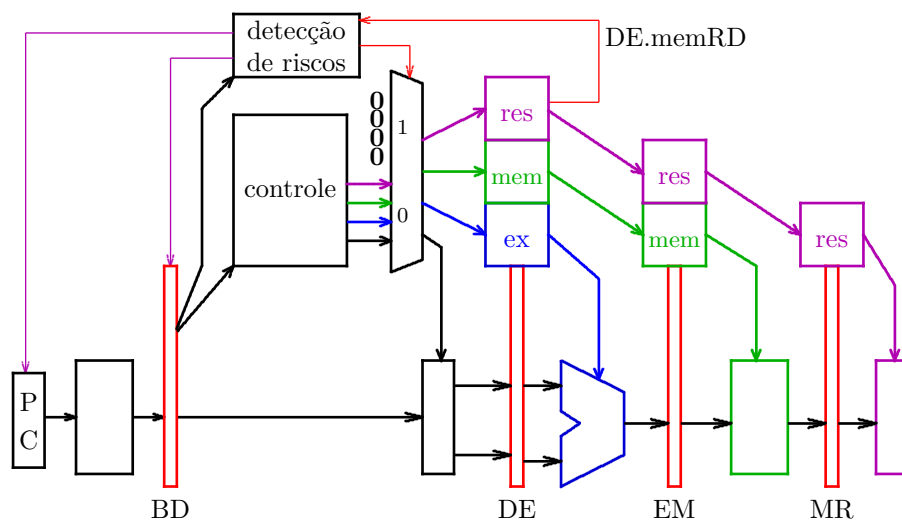


Figura 12.28: Bolha introduzida pelo *branch delay-slot*.

```

li    r1, 0
li    r2, 1024
la    r3, A
li    r4, 0

for:  lw    r5, 0(r3)
      addi  r1, r1, 4      # resolve uso do load
      add   r4, r4, r5
      addi  r3, r3, 4      # resolve dependência em r1
      bne  r1, r2. for
      nop                    # branch delay-slot por preencher
fim:

```

Esta versão reordenada executa com a máxima eficiência possível no processador da Figura 12.28. Este é um caso de “cobertor curto” porque falta uma instrução para preencher o *branch delay-slot*: se a instrução `add r4, r4, r5` for movida para após o `beq`, a dependência em `r1` fica exposta, e deve ser resolvida com um `nop`. ◀

Exemplo 12.9 Vejamos uma implementação da instrução `BRANCH-EQUAL`, que tira proveito do adiantamento dos valores de seus operandos. Os acréscimos ao circuito são mostrados na Figura 12.29 – neste diagrama o banco de registradores *não* está separado em uma seção de leitura e outra de escrita.

A escolha do endereço de destino depende da comparação de dois registradores, e tipicamente, um dos operandos foi computado por uma instrução que precede o `beq`, como mostrado abaixo:

```

addi  r1, r1, 4
sub   r2, r4, r5
bne  r1, r2. dest # depende de r2 e de r1
nop                    # branch delay slot

```

Ao decodificar o `beq`, as dependências são detectadas, e o `beq` é mantido na decodificação até que o `sub` avance para o estágio de acesso à memória, quando então o valor de `r2` é adiantado da saída do registrador de segmento E/M até a entrada do multiplexador controlado por `fwdD`. O valor de `r1` é obtido da porta A do bloco de registradores.

Exercícios

Ex. 12.2 Mostre como implementar a instrução JUMP-REGISTER no processador segmentado. Sua resposta deve conter: (i) uma indicação clara de quais modificações são necessárias no circuito de dados do processador; (ii) quais são os sinais de controle ativos em cada estágio; (iii) uma indicação de quais são os caminhos de adiantamento para esta instrução. O formato é R.

```
jr rs # PC <- R(rs)
```

Ex. 12.3 Repita para a instrução JUMP-AND-LINK-REGISTER, de formato R.

```
jalr rd, rs # R(rd) <- PC+8 , PC <- R(rs)
```

Ex. 12.4 Há vantagem em incrementar o PC de 8 no estágio de decodificação, na instrução BRANCH-AND-LINK? Se for vantajoso, explique quando e o por quê. Veja a Figura 12.22.

Ex. 12.5 Com base na sequência abaixo, que é deveras popular, acrescente quaisquer circuitos de adiantamento que possam ser úteis ao processador da Figura 12.29.

```
sub r2, r4, r5
lw r1, 0(r5)
bne r1, r2, dest # depende de r2 e de r1
nop # branch delay slot
```

Ex. 12.6 Escreva as equações para o controle de bloqueios, necessárias para eliminar os riscos de dados que você descobriu ao resolver o Exercício. 12.5. Veja o Exemplo 12.6.

Ex. 12.7 Para simplificar a exposição, todos os exemplos de código em *assembly* do Capítulo 11 ignoram os *branch delay-slots* e os *load delay-slots*. Revise todos os exemplos, e reordene as instruções para eliminar quaisquer ciclos desperdiçados.

Ex. 12.8 Para responder a esta pergunta, explicitamente quaisquer suposições necessárias.

Considere o programa ao lado, para ser executado no processador segmentado mais simples, sem nenhuma forma de adiantamento e nem bloqueios, mas com *branch delay-slot* e *load delay-slot*. (i) Acrescente ao código o que for necessário para garantir a execução correta deste programa. Qual o número total de ciclos necessários para armazenar a redução contida em r5? (ii) Otimize o código para reduzir o número de ciclos. Qual o novo número total de ciclos? Qual o ganho com relação ao item anterior?

```
la
r20, 0x40000000
li r21, 800
move r5, r0
L: lw r10, 0(r20)
add r5, r5, r10
addiu r20, r20, 4
addiu r21, r21, -4
bne r21, r0, L
nop
sw
r5, -808(r20)
```


Ex. 12.9 Indique, com setas da instrução produtora para a(s) instrução(ões) dependente(s), as dependências entre as instruções do trecho de programa abaixo.

```

# int a, *b, i, j, x[N], y[M], z[P];
# a = 16 / y[ -33000 + i ];
la   ry, Y
lui  at, %hi(-33000)      ; -33.000 > 2**15
ori  rc, at, %lo(-33000)
add  r5, ri, rc
sll  r5, r5, 2           ; indice * 4
add  r5, r5, ry          ; r5 <= Y+4*(i-33000)
lw   r5, 0(r5)
li   r16, 16
div  r16, r5
mflo r6                  ; r6 <= quoc
sw   r6, 0(ra)           ; *a <= y[i-33000]

# b = &(x[ y[ z[j]+2 ] ]);
la   rz, Z
la   ry, Y
la   rx, X
sll  r5, rj, 2           ; indice*4
add  r5, r5, rz          ; r5 <= Z+4*j
lw   r6, 0(r5)           ; r6 <= z[j]
addi r6, r6, 2
sll  r6, r6, 2           ; indice*4
add  r6, r6, ry          ; r6 <= Y+4*([z[j]+2])
lw   r7, 0(r6)           ; r7 <= y[ z[j]+2 ]
sll  r7, r7, 2           ; indice*4
add  r7, r7, rx          ; r6 <= X+4*(y[z[j]+2])
sw   r7, 0(rb)           ; *b <= &( x[y[z[j]+2]] )

```

Índice Remissivo

Símbolos

T_A , 64
 T_D , 120
 T_I , 109, 116
 T_M , 117, 119, 186
 T_P , 116
 T_R , 379, 439, 440
 T_h , 188, 192, 199, 204
 T_p , 171
 T_s , 379, 439, 440
 $T_{C,F}$, 192
 $T_{C,x}$, 118–120, 199, 204, 206
 T_{FF} , 192
 T_{MD} , 379, 382, 440
 T_{MI} , 379, 382, 439, 440
 T_{ULA} , 379, 382, 439, 440
 T_{min} , 199, 206, 379
 T_{skew} , 203
 T_{su} , 188, 192, 199, 204, 379, 382
 $\%$, 18–20, 352, 396
 \Rightarrow , 37
 \bigvee , 47
 \bigwedge , 47
 \equiv , 36
 \gg , 155
 \wedge , 30, 36
 $\triangleleft \triangleright$, 36, 42, 66
 \leftarrow , 193, 344, 426
 \Leftrightarrow , 36
 \Rightarrow , 36, 37, 41, 44
 \ll , 155
 \neg , 30, 36, 154
 \vee , 30, 36
 \oplus , 36, 53, 65
 \mapsto , 42
 \mathbb{N} , 146
 \bar{a} , veja \neg
 π , 25
 \setminus , 77
decod, 72
demux, 75
e, 24
mod, 158, 218, 352
mux, 67
 $\&$, 344, 426
 \mathbb{B} , 29–33, 38, 42
 \mathbb{Z} , 42, 149
 $[r]$, 241, 242
 \mathbb{N} , 42, 43, 149

num, 44, 55, 72, 77, 261, 262, 265, 275
 num^{-1} , 55
 \mathbb{R} , 42
 $|N|$, 227
 $\langle \rangle$, 29, 42, 43, 180, 201, 225
 $,$, 344, 426
 $;$, 344, 426

Números

74148, 75
 74163, 218, 272
 74374, 232

A

ABI, 409
 abstração, 27, 281, 284, 293, 294, 302
 bits como sinais, 27–33, 57, 182, 184
 tempo discretizado, 116, 118, 182, 186–188,
 191–193, 197, 222
 acumulação de produtos parciais, 173–178
 adição, 152
 adiantamento, 452
 adiantamento de vai-um, 164–172, 313–316
 alfabeto, 17
 Álgebra de Boole, 27
 algoritmo, 233
 conversão de base, 18
 conversão de base de frações, 22
 amplificador, 125, 136
 diferencial, 138
 amplitude, 210
 and, veja \wedge
and array, 128
 apontador de pilha, 269
Application Binary Interface, veja ABI
 aproximação, 24
 arquitetura, 343, 385
 arquivo,
 objeto, 418
 árvore, 64, 118
assembler, veja montador
assembly, veja ling. de montagem
 associatividade, 31
 atraso, veja tempo de propagação, 57
 de transporte, 306
 inercial, 305
 atribuição, 12
 autómatos finitos, 236
 auto, 409
 avaliação de desempenho, 317

avaliação preguiçosa, 407

B

barramento, 140
barrel shifter, 158
 básculo, 138, 185–189
 relógio multi-fase, 226
 SR, 185, 194
 binário, 20
 bit, 20
 de sinal, 147
 bits, 27–37, 65
 definição, 29
 expressões e fórmulas, 30
 expressão, 30
 propriedades da abstração, 31
 tipo VHDL, 282
 variável, 30
 bloco de registradores, 265, 367, 377, 378, 427, 428, 437, 439
 bloqueio, 451
 bolha, 452
 borda, *veja* relógio
branch, *veja* desvio condicional
branch delay slot, 449, 454
buffer, 123, 380
buffer three-state, 140
 buraco, 87
 busca, 369, 384, 430, 446
 busca binária, 275
 byte, 11

C

C, 383, 386–409
 classes de armazenagem, 409
 deslocamento, 160
 funções, 407
 if, 397, 400
 if-then-else, 401
 matriz, 392
 overflow, 163
 sizeof, 391
 string, 403
 vetor, 390
 while, 401
 cadeia,
 de portas, 64, 118
 de somadores, 152
 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, 269
 ciclo,
 combinacional, 57
 violação, 81, 184, 186, 188
 de trabalho, 225
 ciclo longo, 378, 439, 444
 circuito,
 combinacional, 57, 65
 de controle, 257
 de dados, 257
 dual, 99
 segmentado, 202
 sequencial síncrono, 196, 209
 circuito aberto, 57
 classes de armazenagem, 409
clear, 194
 clk, *veja* relógio
clock, *veja* relógio, 320
clock skew, *veja skew*
clock-to-output, 192
clock-to-Q, 192
 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, 345
 MIPS, 429
 código,
 Gray, 63, 217, 227, 236
 portável, 408
Column Address Strobe, *veja* CAS
 combinacional,
 ciclo, 57
 circuito, 57
 dispositivo, 57
 comentário, 344, 384, 386
 comparação de magnitude, 399
 comparador, 427
 de igualdade, 62, 164
 de magnitude, 164, 261, 275
 compilador,
 cross, 417
 nativo, 417
Complementary Metal-Oxide Semiconductor, *veja* CMOS
 complemento, *veja* \neg
 complemento de dois, 145–151, 154–164, 427, 432
 complemento, propriedade, 31
 comportamento transitório, *veja* transitório
 comutatividade, 31
 condicional, *veja* $\triangleleft \triangleright$
 condutor, 85
 conjunção, *veja* \wedge

conjunto de instruções, 385, 416, 426
conjunto mínimo de operadores, 65
contador, 211–223
 74163, 218
 assíncrono, 220
 em anel, 224, 225, 232
 inc-dec, 224, 272
 Johnson, 227
 módulo-16, 215, 218
 módulo-2, 212, 223
 módulo-32, 223
 módulo-4, 212
 módulo-4096, 219
 módulo-8, 213, 221
 ripple, 220, 222
 síncrono, 222
contra-positiva, 41
controlador, 242
 de memória, 136
controle de fluxo, 353–357, 397–402
conversão de base, 18
conversor,
 paralelo-série, 229
 série-paralelo, 228
corrente, 85, 105, 106, 112, 113
 de fuga, 115
corrida, 123, 125
cross-compilador, 417
CSS, 196–197, 209
curto-circuito, 57

D

datapath, veja circuito de dados
decimal, 17
decodificação, 344, 369, 384, 430, 446
decodificador, 72, 78, 81–82, 84, 126
 de linha, 126, 135
 de prioridades, 74
delay, 57
DeMorgan, 277
demultiplexador, 75, 120
dependência, 458
 de controle, 454
 de dados, 450, 454
design unit, veja VHDL, unidade de projeto, 279
deslocador exponencial, 156
deslocamento, 155–158, 398
 aritmético, 155, 158, 164
 exponencial, 159, 201
 lógico, 155, 162
 rotação, 158
desvio condicional, 374, 382, 397, 435
detecção de bordas, 123
diagrama de estado, 234
 restrições, 236
diretiva, 418
disjunção, veja \vee
dispositivo, 85
 combinacional, 57

distributividade, 31, 34, 51
divisão de frequência, 213, 216
divisão inteira, 44, 274
doador, 87
don't care, 70, 377
dopagem por difusão, 86
dopante, 86
dot, 418
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, 225

E

EEPROM, 133
efeito colateral, 408
endereço, 77
 alinhado, 390
 base, 351
 de destino, 353, 374, 375, 398, 435, 436, 449, 455
 de retorno, 358, 375, 449
 efetivo, 351, 372, 373, 391, 433, 434
 relativo ao PC, 398, 449
energia, 100, 105, 112–115
enviesado, relógio, veja skew
EPROM, 133
equação característica do FF, 193
equivalência, veja \Leftrightarrow
erro,
 de representação, 23
escopo, 307, 409
espaço,
 de nomes, 343, 386
especificação, 42, 281, 294, 385
estágio,
 busca, 446, 448
 decod, 444–446, 448, 452
 exec, 446, 448, 451
 mem, 446, 448, 452, 453
 result, 444, 446, 448
estado, 182
estado atual, 196, 209, 212
estrutura de dados, 395
excessão, 388
execução, 370–372, 377, 384, 431–433, 437, 446
 paralela, 344, 426
 sequencial, 344, 426
exponenciação, 264
expressões, 36
extensão,
 de sinal, 371, 427, 432, 435

de zeros, 427, 432
extern, 409

F

fan-in, 109–112, 116, 167, 170
fan-out, 82, 84, 109–112, 116, 120, 170–172
fatorial, 274
fechamento, 31
FET, 91
Field Effect Transistor, veja FET
Field Programmable Gate Array, veja FPGA
FIFO, 271
fila, 271–274
 circular, 271
filtro digital de ruído, 190
flip-flop, 188–195
 adjacentes, 198
 comportamento, 193
 destino, 198
 fonte, 197, 198
 mestre-escravo, 189
 modelo VHDL, veja VHDL, *flip-flop*
 temporização, 192
 tipo JK, 193
 tipo T, 191, 193
 um por estado, veja um FF por estado
folga de *hold*, 198
folga de *setup*, 198
forma canônica, 48
formato de instrução,
 I, 429, 449
 J, 429
 Mico, 345
 R, 429, 431
FPGA, 194, 301
frações, veja ponto fixo
frequência, 210
frequência máxima, veja relógio
função, 30
 aninhamento, 360
 convenção, 409
 declaração, 407
 definição, 407
 folha, 410
 protocolo de invocação, 359
 tipo, 29, 42
função de próximo estado, 233, 240, 335
função de saída, 233, 240
função, aplicação bit a bit, 32
função, tipo (op. infix), veja \mapsto
funções, 358–365, 407–414

G

ganho de desempenho, 317
gas, 387
gerador de relógio, 192
ghdl, 276
glitch, veja transitório
GND, 93

gramática, 17
gtkwave, 276, 301, 312

H

half word, 386
handshake, 257
hexadecimal, 19
hold time, 188, 197–202, 246
 folga, 198, 204, 205

I

idempotência, 31
identidade, 31
igualdade, 30
imediato, 371, 429, 432, 437
implementação, 42, 385, 426
 ciclo longo, 430
 segmentada, 443
implicação, veja \Rightarrow
informação, 16
inicialização,
 flip-flop, 194
Instrução,
 busca, veja busca
instrução, 12, 343, 416
 add, 344, 370, 378, 379, 384
 addi, 346, 371, 432
 addu, 427, 431, 439, 440, 446
 andi, 432
 bal, 441, 449, 457
 beq, 353, 374, 381, 398, 427, 435, 455
 busca, veja busca
 com imediato, 371, 432
 corrente, 384
 decodificação, veja decodificação
 execução, veja execução
 formato, 345, 429
 j, 353, 400, 427, 436
 jal, 358, 375, 409
 jalr, 457
 jr, 358, 375, 409, 457
 lógica e aritmética, 370, 431
 ld, 348, 372, 379, 380
 lw, 427, 433, 440, 448, 453
 nop, 353, 451
 ori, 427, 432
 resultado, veja resultado
 slt, 345
 st, 348, 373, 381
 sw, 427, 434
interface,
 de rede, 13
 de vídeo, 12
inversor, 96
 tempo de propagação, 109
involução, 31, 61
IP, 343, 367, 369
isolante, 85

J

Joule, 112
jump, veja salto incondicional

L

laço, 401–404
 infinito, 399
 reordenar instruções, 454
label, 334, 353, 354, 386, 398
 latência, 443
latch, veja basculo
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, 269
 ligação,
 barramento, 140
 em paralelo, 93, 99
 em série, 93, 99
 linguagem,
assembly, veja ling. de montagem
 C, veja C
 de montagem, 342–366, 385–391, 397–402, 409–414
 declarativa, 276, 286
 imperativa, 286
 Pascal, veja Pascal
 Verilog, 277
 VHDL, veja VHDL
 Z, 27
 linha de endereçamento, 78
 literal, 38
load delay slot, 453
location counter, 418
 logaritmo, 43
 lógica restauradora, 125
look-up table, 301
 LUT, 301

M

MADD, 201
 Mapa de Karnaugh, 49, 124
 máquina de estados, 233, 236
 Mealy, 238, 245, 256, 274, 338
 modelo VHDL, 335
 Moore, 237, 244, 256, 271, 336
 projeto, 240
 Máquina de Mealy, veja máq. de estados
 Máquina de Moore, veja máq. de estados
 MARS, 402
 máscara, 32
 matriz, 392
 máximo e mínimo, 31
 maxtermo, 46
 MD, 369, 428, 448
 Mealy, veja máq. de estados
 memória, 181

atualização, 77
 bit, 184
 de controle, 377
 de dados, 369, 428
 de instruções, 368, 427, 428
 de programação de FPGA, 301
 de vídeo, 13
 decodificador de linha, 80
 endereço, 77
 FLASH, 133
 matriz, 129, 132, 134
 multiplexador de coluna, 80
 primária, 13
 RAM, 81, 134, 268
 ROM, 78, 126, 245
 secundária, 13
 memória dinâmica, veja DRAM
 memória estática, veja SRAM
 metaestabilidade, 184, 188, 191, 194
 defesa contra artes das trevas, 191
 metodologia de sincronização, 378, 439
 MI, 368, 428, 446, 448
 Mico XII, 342
assembly, 342–366
 processador, 367–382
 temporização, 378–381
 microarquitetura, 385
 microcontrolador, 245–253
 microrrotina, 253
 mintermo, 45, 124, 126
 MIPS, 385
 MIPS32, 342, 364, 426
 modelagem, 281
 modelo, 281
dataflow, 295
 de von Neumann, 383
 estrutural, 285, 295, 296
 funcional, 44, 294, 296
 porta lógica, 96
 RTL, 295
 temporal, 305
 temporização, 115
 modo de endereçamento, 385
 módulo de contagem, 221
 módulo, veja %, *mod*
 montador, 385, 416–424
 duas passadas, 423
 Moore, veja máq. de estados
 MOSFET, 91
 multiplexador, 61, 66–70, 80, 101, 117, 119, 123–124, 126, 141, 142, 279–280, 285–287, 310–312, 367, 377, 427, 437, 452
 de coluna, 132, 136
 multiplicação, 172–178, 262
 acumulação de produtos parciais, 173–178
 multiplicador,
 somas repetidas, 262, 274
multiply-add, veja MADD

N

número,
 de Euler, 24
 negação, *veja* \neg
net list, 285, 295
 níveis de abstração, *veja* abstração
 nível lógico,
 0 e 1, 28
 indeterminado, 28, 109, 116, 141
 terceiro estado, 140
 nó, 96
 nomes simbólicos, 351
non sequitur, 37
 not, *veja* \neg
 número primo, 53

O

octal, 18
 onda quadrada, 188, 210, 320
opcode, 345, 369, 377, 386, 429, 430, 438
 secundário, 381
 operação,
 binária, 29
 bit a bit, 32
 infixada, 42, 385
 MADD, 201
 prefixada, 47, 280, 385
 unária, 29
 operação apropriada, 197
 operações sobre bits, 29–33
 operador,
 binário, 29
 lógico, 36
 unário, 29
operation code, *veja* *opcode*
 or, *veja* \vee
or array, 129
 otimização de código, 451–454
 ou exclusivo, *veja* \oplus
 ou inclusivo, *veja* \vee
output enable, 269
overflow, 148–150, 154, 162–164, 173, 178, 387

P

paridade, 333
 ímpar, 49
 par, 49
 Pascal, 346–365
 funções, 358–365
 if, 354
 if-then-else, 354
 while, 355
 PC, 383, 427, 430, 435, 449
 período mínimo, *veja* relógio
 pilha, 269–271, 274, 360–365, 410
pipelining, *veja* segmentação, 202, 443
 piso, *veja* [v]
 ponto fixo, 151–152
 ponto flutuante, 70

pop, 269
 porta,
 de escrita, 266
 de leitura, 266
 porta lógica, 65
 and, 59
 carga, *veja* *fan-out*
 de transmissão, 141, 185
 modelo VHDL, 296
 nand, 60, 99
 nor, 60, 98
 not, 59, 96
 or, 59
 xor, 60, 65, 191
 portabilidade, 408
 portas complexas, 100
 potência, 112–115
 dinâmica, 114
 estática, 115
 potenciação, 43
 precedência, 30
 precisão,
 representação, 23
preset, 194
 prioridade,
 decodificador, 74
 processador, 12, 365–382, 430–456
 produtivo, 33
 produto de somas, 46
Program Counter, *veja* PC
 programa de testes, *veja* VHDL, *testbench*
 PROM, 133
 propriedades, operações em \mathbb{B} , 31
 protocolo,
 de sincronização, 257, 274
 invocação de função, 359
 prova de equivalência, 40–41
 próximo estado, 196, 212
 pseudoinstrução, 353, 390
pull-down, 96
pull-up, 96, 126, 141
 pulso, 122, 123, 185, 194, 211, 219, 235, 261
 espúrio, *veja* transitório
push, 269

R

raiz quadrada, 274
 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, 355, 457
refresh, 137, 139
 register, 409

- Register Transfer Language, veja RTL*
- registrador, 195, 232, 250, 265, 343, 367, 427
 \$zero, 387
 a0-a3, 412
 base, 391
 carga paralela, 195
 convenção de uso, 409
 de segmento, 201, 444
 destino, 377, 437
 invisível, 445
 k0,k1, 412
 ra, 364, 412, 449
 s0-s7, 412
 simples, 195
 sp, 361, 364, 412
 t0-t9, 412
 v0,v1, 412
 visível, 387, 445
- registrador de deslocamento, 228–232
 modelo VHDL, *veja* VHDL, registrador, 330
 paralelo-série, 229
 série-paralelo, 228
 universal, 231
- registrador de estado, 196, 335
- registro de ativação, 362, 410
- regra,
 de escopo, 409
- relógio, 186, 188, 192, 210–225, 320, 379, 440, 444
 bordas, 322
 ascendente, 189
 descendente, 189
 ciclo de trabalho, 225
 enviesado, *veja skew*
 frequência máxima, 199
 multi-fase, 225
 período mínimo, 199, 379–381
- reordenação de código, 451
- 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*, 185, 194, 195, 321, 326
- resistência, 87, 100, 105
- resultado, 344, 370, 372, 375, 384, 437, 446
- retorno de função, 375
- return address*, 358
- risco, 450, 451
- ROM, 12, 77–80, 126–133
- rotação, 158, 164
- Row Address Strobe, veja RAS*
- RTL, 202, 278, 295
- rvalue*, 327
- S**
- salto incondicional, 353, 400, 436
- salto para função, 375
- seção, 418
 .bss, 418
 .data, 418
 .text, 418
- segmentação, 201, 443
- segmento, 348
 dados, 347, 348, 357
 texto, 347, 348
- seletor, 72
- semântica, 17, 388
- semicondutor, 85
 tipo N, 87
 tipo P, 87
- set*, 185, 194, 326
- setup time*, 188, 197–202, 246, 379, 439, 440
 folga, 198, 204, 205
- signed*, 388
- silogismo, 37
- simplificação de expressões, 38–40
- simulação,
 eventos discretos, 288
- sinal, 27, 42, 283, 327
 analógico, 27
 digital, 27, 28
 fraco, 125, 126, 138, 141
 intensidade, 90, 139
 restaurado, 125, 139
- síntese, *veja* VHDL, síntese, 301
- sizeof, 391
- skew*, 202–207
- Solid State Disk, veja SSD*
- soma, *veja* somador
- soma de produtos, 45, 51, 126
 completa, 45
- somador, 145, 152–153, 294–301, 367, 427
 adiantamento de vai-um, 165, 232, 312–317
 cadeia, 152
 completo, 104, 152, 296
 modelo VHDL, 296, 313, 315
overflow, 163
 parcial, 103
 seleção de vai-um, 170–172
 serial, 231
 temporização, 201, 312–317
 teste, 178, 299
- somatório, 33
- spice, 28
- SRAM, 138–139
- SSD, 14
- stack frame*, 362, 410
- static, 409
- status*, 162
- string*, 403
- struct, 395
- subtração, 154
- superfície equipotencial, 96, 110

T

tabela,
 de controle, 370, 377, 438
 de excitação dos FFs, 193
 de símbolos, 423
 tabela verdade, 33–35, 45
 tamanho, *veja* |N|
 tempo,
 de compilação, 393
 de contaminação, 115, 118–121, 184, 192, 307
 de estabilização, 188, 192
 de execução, 394
 de manutenção, 188, 192
 de propagação, 57–58, 61, 64–65, 73, 77, 84, 102,
 104, 109, 115–117, 192, 207, 222, 223, 305
 discretizado, 186, 188, 192, 197, 222
 temporização, 104–126, 305–317, 378–381
 somador, 312–317
 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, 179
 de corretude, 178
 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, 186, 239, 312
 transmissão,
 serial, 230
transmission gate, *veja* porta de transmissão
 TTL,
 74148, 75
 74163, 218
 74374, 232
 tupla, *veja* ⟨⟩
 elemento, 32
 largura, 43

U

ULA, 160–164, 180, 265, 343, 367, 371, 377, 385,
 427, 432, 437, 452
status, 162
 um FF por estado, 253–256
 Unidade de Lógica e Aritmética, *veja* ULA
 unidade de projeto, 284
unsigned, 388
 uso do *load*, 452, 453

V

valor da função, 30
 vazão, 443
 VCC, 93
 Verilog, 277
 vetor, 390
 de testes, 290
 endereçamento, 392
 vetor de bits, *veja* ⟨⟩, 32
 largura, 43
 VHDL, 194, 276–341, 344, 347
 &, 282
 (i), 283
 :=, 283
 <=, 282
 =>, 287
 active, 328
 after, 305, 312
 all, 285
 architecture, 279
 área concorrente, 286
 arquitetura, 279
 array, 291
 assert, 293
 associação,
 nominal, 287
 posicional, 286
 atraso,
 de transporte, 306
 inercial, 305
 atributo, 322, 328
 bit, 279, 283, 295
 bit_vector, 279
 boolean, 283
 borda ascendente, 322, 339
 case, 332
 concatenação, 283
 delta, 288–289, 319
design unit, *veja* VHDL, unidade de projeto
 entidade, 279
 entity, 279
 event, 322, 328
 evento, 288
 exit, 334
flip-flop, 322–327
 for, 293, 333
 função, 339–341
 generate, 312
 generic, 309

in, 279, 334
 inertial, 305
 inicialização de sinal, 283
 instanciação, 286
 interface genérica, 309
label, 334
last_value, 328
length, 328
 linguagem, 276
 loop, 333, 334
 mapeamento de portas, 286
 máquina de estados, 335
 modelo executável, 281
 open, 310
 others, 332
 out, 279
 package, 284
 port map, 280, 286, 287
 processo, 288
 combinacional, 329
 sequencial, 330
 range, 328, 334
 record, 290
 registrador, 330
 reject, 306, 312
 report, 293
rising_edge, 339, 340
rvalue, 327
 síntese, 281
 seleção de elemento de vetor, 283
 severity, 293
 simulação, 288
 sinal, 283
 síntese, 207, 301, 305
std_logic, 140
 subtype, 284
testbench, 289–293, 299–301
 time, 283
 tipos, 42, 282, 287
 to, 334
 transação, 288
 transport, 306
 type, 283, 335
 unidade de projeto, 284
 use, 285
 variável, 327–331
 wait, 320, 321, 323
 wait for, 321
 wait on, 321
 wait until, 321
 when, 332
 while, 334
 work, 285

W

Watt, 112
word, 386
write back, veja resultado

X

xor, veja \oplus

Z

Z, linguagem, 27