Coleção UAB–UFSCar

Sistemas de Informação

José Hiroki Saito

Introdução à arquitetura e à organização de computadores

Síntese do processador MIPS

Introdução à arquitetura e à organização de computadores

Síntese do processador MIPS

UAB-UFSCar
Universidade Federal de São Carlos
Rodovia Washington Luís, km 235
13565-905 - São Carlos, SP, Brasil
Telefax (16) 3351-8420
www.uab.ufscar.br
uab@ufscar.br

EdUFSCar
Universidade Federal de São Carlos
Rodovia Washington Luís, km 235
13565-905 - São Carlos, SP, Brasil
Telefax (16) 3351-8137
www.editora.ufscar.br
edufscar@ufscar.br

José Hiroki Saito

Introdução à arquitetura e à organização de computadores

Síntese do processador MIPS

São Carlos

EdUFSCar

2013

Concepção Pedagógica
Daniel Mill

Supervisão
Douglas Henrique Perez Pino

Revisão Linguística
Clarissa Galvão Bengtson
Daniel William Ferreira de Camargo
Gabriel Hayashi
Juliana Carolina Barcelli

Editoração Eletrônica
Izis Cavalcanti

Ilustração
Maria Julia Barbieri Mantoanelli

Capa e Projeto Gráfico
Luís Gustavo Sousa Sguissardi

Ficha catalográfica elaborada pelo DePT da Biblioteca Comunitária da UFSCar

S158i

Saito, José Hiroki.
Introdução à arquitetura e à organização de computadores : Síntese do processador MIPS / José Hiroki Saito. -- São Carlos : EdUFSCar, 2010.
191 p. -- (Coleção UAB-UFSCar).

ISBN – 978-85-7600-207-9

1. Organização de computador. 2. Aritmética de computador. 3. Computadores canalizados. 4. Sistemas de memória de computadores. I. Título.

CDD – 004.22 (20ª)
CDU – 681.32.02

SUMÁRIO

UNIDADE 4: Unidade lógica e aritmética no MIPS

UNIDADE 5: Desempenho de computadores

APRESENTAÇÃO

O presente livro foi desenvolvido para estudar dois aspectos dos computadores: a arquitetura e a organização.

Entende-se por arquitetura os aspectos funcionais dos computadores, sem entrar no mérito da sua construção. São aspectos de arquitetura, por exemplo, o tamanho dos dados, o tamanho máximo de memória e o conjunto de instruções executadas. A organização refere-se à metodologia de construção da máquina, tal como a tecnologia de circuito eletrônico utilizada, a distribuição dos caminhos de dados, o posicionamento das unidades funcionais, bem como a técnica de implementação da unidade de controle. Apesar da diferenciação mencionada anteriormente, a arquitetura e a organização são, na prática, indissociáveis, razão pela qual se justifica o nome dado ao livro.

O livro está dividido em unidades, sendo a primeira delas uma introdução à arquitetura e à organização dos computadores, apresentando um histórico e um panorama da tecnologia dos computadores e processadores.

A Unidade 2 aborda os conceitos de circuitos digitais, que são fundamentais para a manipulação de dígitos binários, empregados na construção dos computadores, uma vez que é usada a lógica binária para essas máquinas. Nota-se que existem basicamente dois tipos de circuitos digitais. O primeiro tipo corresponde aos circuitos combinacionais, cujas saídas dependem apenas dos valores das entradas, ou seja, quando se alteram as entradas, eles resultam em saídas preestabelecidas. O segundo tipo corresponde aos circuitos sequenciais cujas saídas dependem das entradas e do estado interno. Será abordado o conceito do relógio (*clock*), que é um sinal básico que altera o estado interno dos circuitos sequenciais.

Na Unidade 3, será estudada a interface entre a implementação física, *hardware*, e a programação, *software*. Essa interface coincide com as instruções que o processador executa. Um *software* refere-se ao sequenciamento lógico dessas instruções para a execução de uma determinada tarefa. O *hardware* deve executar as instruções do *software*, uma a uma na sequência apresentada.

A Unidade 4 consiste no estudo das operações aritméticas. São vistos os principais tipos de representações usados para os dados, bem como os algoritmos para a execução das operações. Nos computadores de uso geral existem a representação de números inteiros e a representação de ponto-flutuante, a qual corresponde aos números fracionários e tem um intervalo bem maior de representação em relação aos números inteiros.

Na Unidade 5, é estudado o conceito de desempenho e como podem ser medidos os desempenhos dos computadores. É vista a lei de Amdahl que calcula

o *speedup* para uma máquina melhorada em relação a uma máquina original. O *speedup* corresponde ao ganho obtido em decorrência do aumento do desempenho. É visto o uso de *benchmark,* que consiste em programas padronizados para a comparação de desempenho entre processadores diferentes. Um *benchmark* bastante conhecido é o SPEC (*System Performance Evaluation Cooperative*), cujo primeiro turno começou em 1989 e atualmente se encontra no quinto turno lançado em 2006.

A Unidade 6 refere-se ao estudo da organização dos computadores, que proporciona o entendimento da construção de um computador, juntando as diversas unidades funcionais, como unidade aritmética, banco de registradores e memória. São verificados o fluxo de dados e o controle para a execução das instruções vistas na Unidade 3. É também visto o funcionamento de unidades de controle fixo (*hardwired*) apropriadas para uso em computadores RISC (*Reduced Instruction Set Computer*) e de unidades de controle microprogramado apropriado para computadores CISC (*Complex Instruction Set Computer*), com grande número e variedade de formatos de instruções.

As unidades seguintes descrevem alguns aspectos de evolução das arquiteturas de computadores, bem como mecanismos usados para explorar melhor as diversas tecnologias de memória e também os mecanismos de entrada e saída nos computadores.

Assim, na Unidade 7, são estudados os mecanismos que são introduzidos na arquitetura e na organização dos computadores para a melhoria do desempenho. Um desses mecanismos é o *pipeline*, em que as instruções são processadas em alguns estágios subsequentes e várias instruções podem ser processadas simultaneamente, porém, cada uma ocupando um determinado estágio.

Na Unidade 8, é visto o sistema de hierarquia de memória, conceito importante que permite a coexistência de memórias de pequena capacidade, com alto custo por *bits*, porém de alta velocidade, e memórias de grande capacidade, com baixo custo por *bits*, porém de baixa velocidade. Entende-se por capacidade a quantidade armazenada de *bits*, e por velocidade a rapidez com que uma palavra é lida ou escrita na memória. Essas memórias são dispostas no sistema de forma que o processador fique mais próximo da memória de alta velocidade que da memória de baixa velocidade. Quanto mais o processador acessar as palavras contidas na memória mais próxima, de alta velocidade, o desempenho global é maior.

Finalmente, na Unidade 9, é feito o estudo da entrada e da saída. A entrada/saída é importante pois, sem ela, o computador seria inútil, pois não teríamos como introduzir dados, nem programas de usuário, e tampouco obter resultados do processamento.

É importante salientar que o estudo sobre arquitetura e organização de computadores é de fundamental importância para a formação profissional em sistemas de informação. Em diversos momentos o profissional dessa área pode enfrentar situações que exigirão conhecimento profundo da máquina. Essa situação pode ser em nível gerencial, por exemplo, na decisão de implantação de um sistema computacional, ou em projeto de sistemas computacionais, por exemplo, na otimização de programas para a melhoria do desempenho.

O autor agradece a Denis Henrique Pinheiro Salvadeo pela colaboração na elaboração da Unidade 2; a Mário Lizier e Renato de Oliveira Violin, pelas sugestões ao longo do texto; a Luisa Theruko Utuni pelas correções gramaticais; a Gislaine Micheloti e Sandra Abib pelas orientações; à equipe de Material Impresso e às demais pessoas que direta ou indiretamente colaboraram na elaboração deste livro.

UNIDADE 1

Introdução à tecnologia e à evolução dos computadores

1.1 Primeiras palavras

A Unidade 1 consiste numa introdução à arquitetura e à organização de computadores. É feita uma apresentação histórica da tecnologia, e traçado um panorama geral das tecnologias atuais.

1.2 Problematizando o tema

Antes da problematização do tema, definiremos termos muito usados: computador e processador. Refere-se ao termo computador o sistema de processamento completo, incluindo a memória e a unidade de processamento, que aceita a interligação de diversos periféricos como impressora, teclado, monitor de vídeo, unidade de disco rígido e modem. A rigor, o termo processador refere-se a apenas uma parte da máquina, a unidade central de processamento, sem incluir a memória. O processador também é conhecido como Unidade Central de Processamento (*Central Processing Unit* – CPU, em inglês). Como atualmente um processador é implementado, na maioria das vezes, em circuito integrado (*chip*), pode-se confundir o processador com o próprio *chip* que o contém. Porém, à medida que os *chips* aumentam de conteúdo, o mesmo pode conter vários processadores, como os *chips multicore* atuais.

Gordon E. Moore, ex-presidente da Intel, certa vez observou que o poder de processamento dos *chips*, bem como o número de transistores, dobrava a cada 18 meses. Essa profecia tornou-se realidade e acabou ganhando o nome de lei de Moore. Essa observação tem servido de parâmetro para uma elevada gama de dispositivos digitais além das CPUs. Analisando, porém, em detalhes, os processadores a partir de 1978, nota-se que no período inicial o aumento no poder de processamento (*desempenho*) era de 25% ao ano. A partir de 1986 o aumento foi mais acentuado, 52% ao ano e, a partir de 2002 o aumento tem sido de 20% ao ano, conforme mostra a Figura 1.

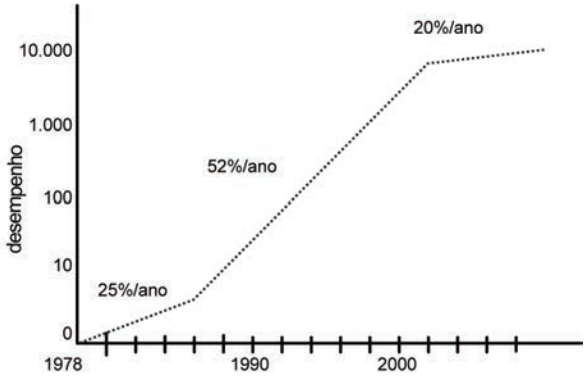


Figura 1 Curva de desempenho dos processadores a partir de 1978.

O aumento inicial de 25% anuais deve-se principalmente ao avanço da tecnologia de fabricação de circuitos integrados, que é a tecnologia usada para os processadores atuais. O aumento de 52%, no período de 1986 a 2002, deve-se não só ao avanço da tecnologia, mas também ao melhoramento da arquitetura e organização a partir da introdução de conceitos como a arquitetura superescalar, com despacho e execução de várias instruções simultaneamente. No período a partir de 2002, é notado um crescimento menor em relação aos períodos anteriores, devido à saturação na exploração dos mecanismos de melhoria de desempenho de execução de instruções, problemas de latência de memória e de dissipação de energia nos circuitos integrados cada vez mais densos.

O presente livro pretende abordar a arquitetura e a organização de processadores e computadores de forma a esclarecer os principais mecanismos que resultam na curva da Figura 1.

1.3 Introdução à tecnologia dos processadores

O item descreve a evolução da tecnologia, uma breve explanação sobre a fabricação de *chips*, a evolução da memória dinâmica, o problema da dissipação de energia nos circuitos integrados e as classes de computadores.

1.3.1 Evolução da tecnologia eletrônica

Os processadores e demais componentes de um computador tiveram uma melhoria de velocidade, em grande parte, por causa da evolução da tecnologia de construção. A Tabela 1 mostra as tecnologias usadas a partir de 1951, época em que estavam em desenvolvimento os primeiros computadores.

Tabela 1 Tecnologia de fabricação dos computadores.

Ano	Tecnologia	Desempenho/custo
1951	válvula	1
1965	transistor	35
1975	circuito integrado	900
1995	circuito VLSI	2.400.000
2005	circuito ULSI	6.200.000.000

Fonte: adaptada de Hennessy & Patterson (2008).

A primeira coluna representa o ano em que a tecnologia atinge o desempenho/custo explicitado. Embora a tecnologia eletromecânica tenha sido usada para os primeiros computadores desenvolvidos em universidades, ela não consta

na Tabela 1 pelo fato de ser muito lenta se comparada à tecnologia eletrônica. As tecnologias apresentadas são todas eletrônicas e são comparadas com a primeira que é a válvula, portanto, com valor de desempenho/custo igual a 1. As válvulas são componentes de forma e tamanho semelhantes a uma lâmpada incandescente, com vácuo interno, que atingem centenas de graus centígrados de temperatura durante o funcionamento. A segunda tecnologia é o transistor construído a partir de materiais semicondutores. Os primeiros computadores a transistores usaram esses mesmos componentes, construídos discretamente (em empacotamento individual para cada transistor). As linhas de interconexão de cobre impressas sobre as superfícies desses cartões interligavam os terminais dos transistores com outros componentes formando os circuitos digitais. A tecnologia de circuito integrado refere-se àqueles denominados na época de pequena (*small scale integration* – SSI), média (*medium scale integration* – MSI) e larga (*large scale integration* – LSI) integração, contendo em média algumas unidades, dezenas, ou centenas de transistores, respectivamente. Para se construir um computador, era necessário o uso de centenas ou milhares desses circuitos integrados de pequena, média ou larga escala de integração. O circuito VLSI (*Very Large Scale Integration*) é, como o próprio nome diz, integrado numa escala muito larga. Essa é a tecnologia em que os microprocessadores são construídos num único *chip*. O circuito ULSI (*Ultra Large Scale Integration*) é um circuito ainda mais denso em relação ao anterior e tem atualmente a capacidade de conter vários processadores internamente.

1.3.2 Tecnologia de fabricação de *chips*

A fabricação de um *chip* começa com o silício contido num cristal de quartzo. O silício é um material semicondutor, caracterizado por se situar entre os materiais condutores de eletricidade e os isolantes. É possível, por meio de processos químicos, adicionar materiais no silício e transformar pequenas áreas em: a) excelentes condutores; b) excelentes isolantes; e c) áreas que conduzem ou não sob controle, conhecidas como transistores. Um circuito VLSI, ou ULSI, é, portanto, uma combinação de enorme quantidade dessas três áreas.

A Figura 2 ilustra um processo de fabricação de um circuito integrado VLSI ou ULSI: (a) o lingote de silício, cujas dimensões atualmente variam entre 30 e 60 cm de comprimento e 20 e 30 cm de diâmetro; (b) após passar por um fatiador, as fatias, conhecidas como *wafers*, têm aproximadamente 0,25 cm de espessura; (c) as fatias virgens passam por várias etapas de processos químicos para a criação de transistores, condutores, e isolantes; (d) as fatias com os circuitos já implementados são testadas, e os elementos marcados com *x* referem-se aos circuitos que apresentaram falhas; (e) os circuitos (*dies*), também conhecidos como *chips*, são separados individualmente por um cortador (*dicer*); (f) os circuitos

separados e aprovados são encapsulados; (g) os circuitos encapsulados são testados novamente, pois o processo de encapsulamento pode provocar danos; (h) os circuitos que passaram pelo teste final podem ser comercializados.

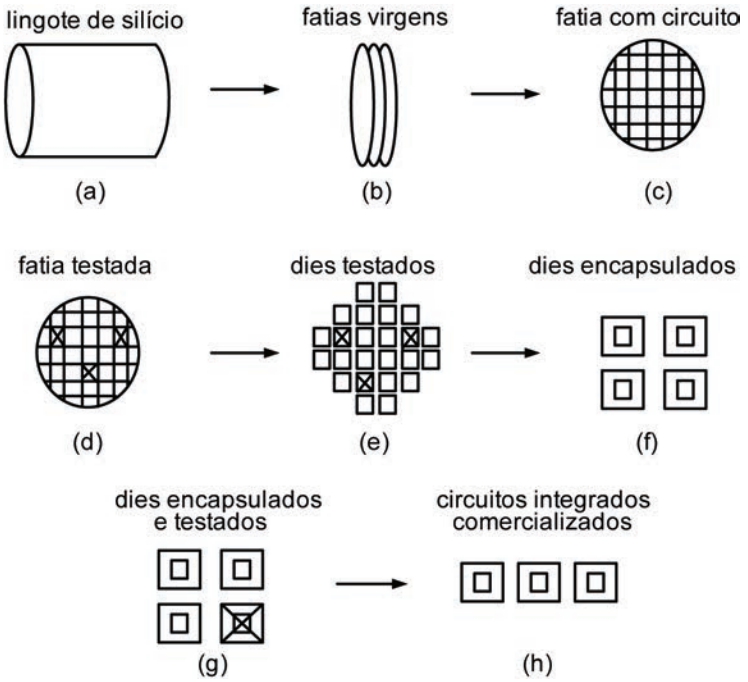


Figura 2 Processo de fabricação de um circuito integrado VLSI ou ULSI.

Fonte: adaptada de Hennessy & Patterson (2008).

1.3.3 Evolução da tecnologia de memória dinâmica

A tecnologia atual de memória principal dos computadores é conhecida como RAM (*Random Access Memory*) dinâmica.

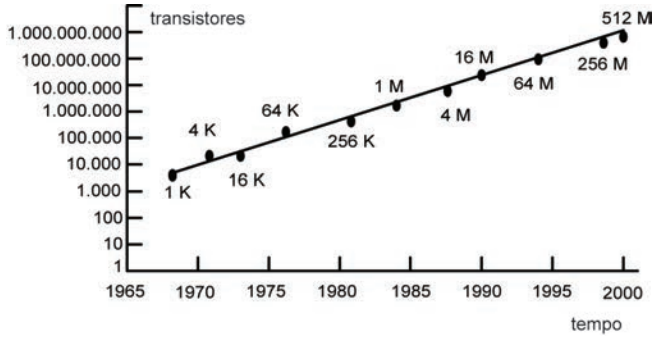


Figura 3 Diagrama ilustrativo da evolução da tecnologia de memória RAM dinâmica, ao longo do tempo, em relação ao número de transistores usados nos chips (K = 1024 bits, M = 1024 Kbits).

Fonte: adaptada de Hennessy & Patterson (2008).

A evolução da tecnologia dessa memória se caracteriza pela quadruplicação da capacidade a cada novo lançamento, ao longo dos anos. A Figura 3 ilustra um diagrama no qual se pode verificar a linearidade dessa evolução, que está de acordo com a lei de Moore. Apesar de a capacidade das memórias RAM dinâmicas ter sido duplicada a cada 18 meses, conforme a lei de Moore, a velocidade de acesso dessas memórias tem evoluído na razão de duplicação a cada 10 anos.

1.3.4 Consumo de energia nos circuitos integrados

O consumo de energia nos circuitos integrados equivale à dissipação de calor. Para os *chips* de CMOS (*Complementary Metal Oxide Semiconductor*), que constituem a tecnologia atual de fabricação dos microprocessadores, o consumo de energia principal tem sido, na comutação dos transistores, chamado de potência dinâmica. A potência dinâmica (em watts) exigida por um transistor é dada pela equação:

$$Potência = ½ \times carga\ capacitiva \times voltagem^2 \times frequência.$$

Assim, a potência dinâmica é diminuída reduzindo-se a voltagem, e, por isso, as voltagens caíram de 5 V para pouco mais de 1 V em 20 anos. O problema de potência tem ficado cada vez mais crítico na tecnologia de circuitos integrados com o aumento dos transistores nos *chips*, tem-se notado dissipadores cada vez mais sofisticados sobre os processadores.

1.3.5 Classes de aplicação dos processadores

Neste item será feito um breve resumo sobre as classes de aplicações de processadores. A primeira classe de aplicação é a do computador *desktop*, em que o computador é projetado para uso pessoal e normalmente é acompanhado de um monitor gráfico, um teclado, um mouse e interfaces para impressoras, redes e outros periféricos. A outra classe de aplicação é a do computador *servidor*, que é usado para executar grandes programas de múltiplos usuários, normalmente conectados em rede. Esses servidores podem ser computadores semelhantes aos *desktops* usados para armazenamento de arquivos, para pequenas aplicações comerciais e serviços *web* simples. Podem ser também computadores enormes, chamados de supercomputadores, contendo centenas até milhares de processadores, *gigabytes* até *terabytes* de memória e *terabytes* até *petabytes* de disco rígido, normalmente usados para cálculos científicos e de engenharia, como previsão metereológica, exploração de petróleo e pesquisas

de estruturas de proteínas. Outra classe de aplicação de processadores é a de computadores embutidos, dedicados ou embarcados. Essa é a classe com maior diversidade e quantidade, e inclui os microprocessadores encontrados em controles de eletrodomésticos e de veículos, em telefones celulares, em televisões digitais, entre outros. Esses processadores, como estão embutidos em dispositivos diversos, são usados para executar uma aplicação predeterminada. Exemplos desses processadores estão, também, em cartões de congratulações e em *chips* de identificação por radiofrequência, RFID (*Radio Frequency Identification*), usados para identificações de veículos, animais e produtos em supermercados. Muitos *chips* usados para controle incluem uma grande quantidade de interfaces, para facilitar a construção de aplicações, e nesses casos são denominados microcontroladores. Outros processadores, denominados Processadores de Sinais Digitais (*Digital Signal Processor – DSP*), são projetados para o processamento de filtros digitais em tempo real.

1.4 Considerações finais

Nesse texto foi feita uma introdução à *arquitetura e à organização de computadores*, sobre as classes de processadores atuais, a tecnologia de fabricação e sua evolução. As descrições mais detalhadas podem ser encontradas nas bibliografias e referências.

1.5 Estudos complementares

Para complementar os estudos introdutórios sobre a tecnologia de construção de computadores, sugere-se a leitura do primeiro capítulo, *Fundamentos do projeto de computadores*, do livro de Hennessy & Patterson (2007). Sugere-se também o primeiro capítulo, *Computer Abstractions and Technology*, do livro de Hennessy & Patterson (2008). Recomenda-se a leitura do primeiro capítulo, *Introdução*, do livro de Tanenbaum (2007). O artigo de J. A. N. Lee (1996) apresenta uma visão cronológica da história da computação.

UNIDADE 2

Fundamentos de circuitos digitais

2.1 Primeiras palavras

A Unidade 2 consiste no estudo da lógica digital que é fundamental para a compreensão da organização de computadores. É feita uma apresentação simplificada do assunto, porém suficiente para o objetivo mencionado.

2.2 Problematizando o tema

A lógica digital descrita nesta unidade é bastante simplificada. Destacamos basicamente dois tipos de circuitos: combinatórios, ou combinacionais, e sequenciais.

Iniciamos o estudo com os circuitos combinatórios, ou combinacionais, a partir do exemplo da Figura 4, que apresenta um diagrama com três entradas à esquerda, denominadas x_2, x_1 e x_0 e, à direita, uma saída z. A identificação de entrada e saída, na figura, pode ser feita em função das respectivas setas.

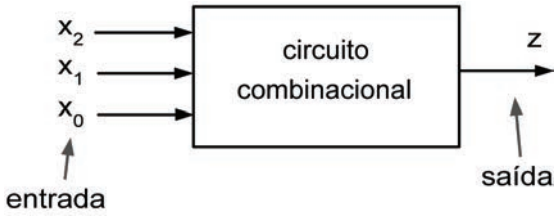


Figura 4 Circuito combinacional.

Tabela 2 Tabela-verdade.

x_2	x_1	x_0	z
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Os circuitos digitais utilizam apenas dois valores de voltagens, normalmente uma voltagem nula, denotada zero, e uma voltagem diferente de zero, denotada um. O valor da voltagem nula seria correspondente a zero volt. O valor da outra voltagem varia de acordo com a tecnologia usada, podendo ser cinco volts, dois volts, etc. Para entender a função do circuito combinacional da Figura 4, é necessário conhecer qual é a saída para cada combinação de entrada. Na Tabela 2 há o que denominamos tabela-verdade, na qual estão ilustradas as combinações de entradas e suas respectivas saídas para o circuito da Figura 4.

O que caracteriza um circuito combinacional é o uso de uma tabela, como a da Tabela 2, para se saber completamente a sua função. Portanto, a saída depende apenas da combinação de entradas.

Por outro lado, o circuito sequencial é um circuito que, além da combinação das entradas, depende também do seu estado interno.

A presente unidade faz uma descrição de como desenvolver os circuitos combinacionais nos primeiros itens e, em seguida, como desenvolver os circuitos sequenciais, que na sua construção fazem uso de circuitos combinacionais.

2.3 Circuitos combinacionais

O item descreve as portas lógicas principais usadas em lógica digital, ou circuitos digitais. Essas portas lógicas são os elementos primitivos usados para a construção de quaisquer circuitos combinacionais. Será vista a álgebra booleana cujos fundamentos são aplicados para o estudo do comportamento dos circuitos combinacionais. Serão também apresentados os conceitos de expressão de funções em forma canônica e circuitos equivalentes.

2.3.1 Portas lógicas *not*, *and* e *or*

Para a construção dos circuitos combinacionais fazemos o uso de portas lógicas, que são circuitos combinacionais simples. A partir de um conjunto de portas lógicas simples é possível construir um circuito combinacional complexo.

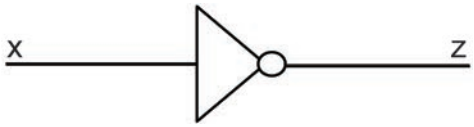


Figura 5 Porta lógica *not*, ou *não*.

A Figura 5 ilustra uma porta lógica *not*, ou *não*, que tem como entrada a variável x, e como saída a variável z. A função da mesma é inverter o valor da entrada x. Assim, quando $x = 0$, a saída $z = 1$, e vice-versa.

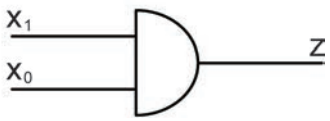


Figura 6 Porta lógica *and*, ou *e*.

A Figura 6 mostra uma porta lógica *and*, ou *e*, que tem como entradas as variáveis x_1 e x_0, e como saída a variável z. A função da porta *and* é facilmente entendida pela Tabela 3.

Tabela 3 Porta lógica *and*.

x_1	x_0	z
0	0	0
0	1	0
1	0	0
1	1	1

Quando as entradas x_1 e x_0 forem iguais a 1, a saída z é igual a 1.

As portas lógicas *and* podem ter mais que duas entradas. Nesse caso, a saída z é igual a 1 somente quando todas as entradas forem iguais a 1.

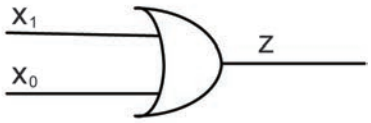


Figura 7 Porta lógica *or*, ou *ou*.

A Figura 7 mostra uma porta lógica *or*, ou *ou*, que tem como entradas as variáveis x_1 e x_0, e como saída a variável z. A função da porta *or* está resumida na Tabela 4.

Tabela 4 Porta lógica *or*.

x_1	x_0	z
0	0	0
0	1	1
1	0	1
1	1	1

Quando a entrada x_1 ou x_0 for igual a 1, a saída z é igual a 1. Quando ambas as entradas forem iguais a 1, a saída z é, também, igual a 1.

Uma porta lógica *or* pode ter mais que duas entradas. Nesse caso, a saída z é igual a 1 quando pelo menos uma das entradas for igual a 1.

O conjunto de portas *not*, *and* e *or* é suficiente para a construção de qualquer circuito combinacional. Por isso, esse conjunto é denotado conjunto universal de portas lógicas.

2.3.2 Portas *nand*, *nor*, *xor* e *xnor*

A porta *nand* comporta-se com uma combinação da porta *and* com uma porta *not*. Portanto, a saída z é igual a 0 quando todas as entradas forem iguais a 1. A Figura 8 mostra o símbolo de uma porta *nand* de duas entradas.

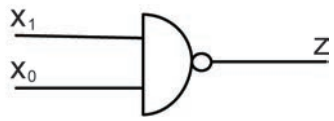


Figura 8 Porta lógica *nand*.

A porta *nor* comporta-se com uma combinação da porta *or* com uma porta *not*. Portanto, a saída z é igual a 0 quando pelo menos uma das entradas for igual a 1. A Figura 9 mostra o símbolo de uma porta *nor* de duas entradas.

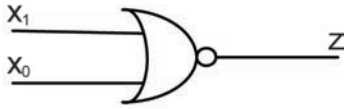


Figura 9 Porta *nor*.

A porta *xor* (Figura 10) realiza a função conhecida como *ou-exclusivo*. Na Tabela 5 é mostrada a função *xor*, em que a saída é 1 quando uma das entradas é 1, porém não ambas.

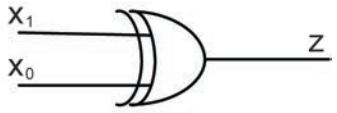


Figura 10 Porta *xor*.

Tabela 5 Porta lógica *xor*.

x_1	x_0	z
0	0	0
0	1	1
1	0	1
1	1	0

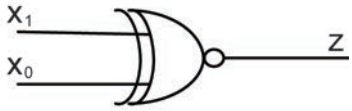


Figura 11 Porta *xnor*.

A porta *xnor* (Figura 11) é uma combinação da porta *xor* com a porta *not*. A sua saída é igual a 1 quando as variáveis de entrada coincidem em 0 ou em 1.

2.3.3 Álgebra booleana

Originalmente definida, com seus postulados e teoremas, por George Boole, a álgebra booleana é usada para modelar operações lógicas. Um caso específico da álgebra booleana é a álgebra de chaveamento, que é utilizada para descrever circuitos combinacionais por meio de expressões de chaveamento. Portanto, os teoremas da álgebra booleana são válidos para a álgebra de chaveamento.

A álgebra booleana é composta por um conjunto de elementos *B* e operações binárias + (*or*) e · (*and*), aplicadas sobre os elementos de *B*. Vale ressaltar que *and* tem precedência sobre *or*. Além disso, podem ser utilizados os parênteses para "quebrar" a precedência e um operador unário denominado *not* (negação). Dessa forma, a precedência completa fica nesta ordem (), *not*, *and* e *or*. Chamamos atenção ao fato de as operações booleanas, *or* e *and*, serem diferentes da soma e multiplicação aritmética, apesar do uso dos mesmos símbolos (+ e ·).

Os principais postulados e propriedades da álgebra booleana são apresentados a seguir. Considera-se a, b e $c \in B$ e a' como sendo o operador *not* (o operador de negação pode ainda ser representado por uma barra, como $\overline{a}$):

1. Comutatividade: a ordem dos elementos a e b em uma operação não altera o resultado final.

$$a + b = b + a$$

$$a \cdot b = b \cdot a$$

2. Distributividade: em equações é semelhante ao que conhecemos em álgebra para a aritmética.

$$a + (b \cdot c) = (a + b) \cdot (a + c)$$

$$a \cdot (b + c) = (a \cdot b) + (a \cdot c)$$

Porém, em circuitos, o resultado para a primeira equação é o da Figura 12.

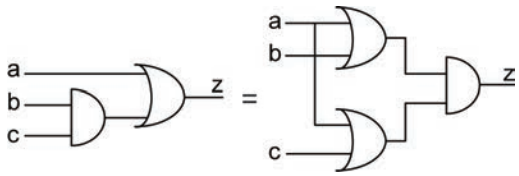


Figura 12 Distributividade $a+(b\cdot c)=(a+b)\cdot(a+c)$.

E para a segunda equação, o resultado é o da Figura 13.

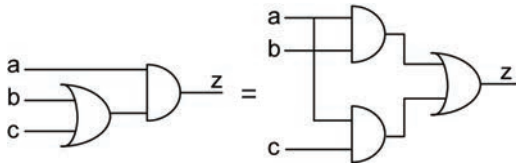


Figura 13 Distributividade $a\cdot(b+c)=(a\cdot b)+(a\cdot c)$.

3. Associatividade: novamente, a associatividade é semelhante à álgebra para a aritmética.

$$a+(b+c)=(a+b)+c=a+b+c$$

$$a\cdot(b\cdot c)=(a\cdot b)\cdot c=a\cdot b\cdot c$$

4. Lei da Idempotência: se os operandos forem iguais em uma operação binária, resulta no próprio operando.

$$a+a=a$$

$$a\cdot a=a$$

5. Complemento: as operações sobre os complementos têm os seguintes resultados.

$$a+a'=1$$

$$a\cdot a'=0$$

6. Os elementos 0 e 1: fazem a função de identidade aditiva e multiplicativa.

$0 + a = a + 0 = a$

$1 \cdot a = a \cdot 1 = a$

7. Lei da Involução: a operação *not* aplicada duas vezes anula o efeito.

$(a')' = a$

8. Lei da Absorção: é muito usada para simplificar expressões.

$a + a \cdot b = a$

$a \cdot (a + b) = a$

9. Lei da Simplificação: idem, para simplificar expressões.

$a + a' \cdot b = a + b$

$a \cdot (a' + b) = a \cdot b$

10. Lei de DeMorgan: é muito usada para a obtenção de expressões duais.

$(a + b)' = a' \cdot b'$

$(a \cdot b)' = a' + b'$

11. Princípio da Dualidade: para uma equação booleana, se intercambiarmos as operações *and* e *or* e os elementos de identidade zero e um, obteremos uma equação booleana denominada dual.

Exemplo: Considerando-se a expressão booleana $a + 0 = a$, se trocarmos *or* por *and* e zero por um, obteremos $a \cdot 1 = a$, que corresponde à propriedade 6.

2.3.4 Função de chaveamento em forma canônica

Notamos que uma forma de obter a função de chaveamento é verificar as linhas da tabela-verdade, nas quais a saída z é igual a 1 (Figura 14).

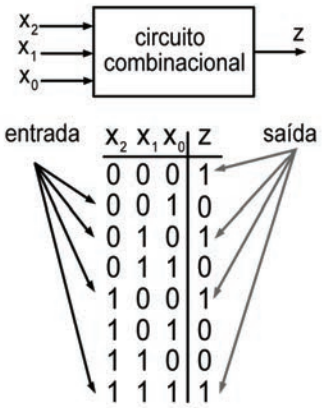


Figura 14 Exemplo de circuito combinacional.

Assim, $z = 1$ para as entradas $x_2x_1x_0 = 000$, 010, 100 e 111. Representando cada uma dessas combinações de entrada, negando a variável de entrada quando o valor é 0, temos: $x_2'x_1'x_0'$, $x_2'x_1x_0'$, $x_2x_1'x_0'$ e $x_2x_1x_0$. Podemos entender cada uma das combinações como uma porta *and*, de três entradas. Por exemplo, no caso de $x_2'x_1'x_0'$, podemos dizer que a saída será $z = 1$ quando $x_2 = 0$, $x_1 = 0$ e $x_0 = 0$. Como a função de chaveamento deve representar o conjunto de todas essas quatro combinações de entrada, podemos expressar a função com *or* dessas combinações de entrada, conforme a equação:

$$z = x_2'x_1'x_0' + x_2'x_1x_0' + x_2x_1'x_0' + x_2x_1x_0$$

Essa forma de expressar uma função de chaveamento é chamada de forma canônica, pois ela é única. Cada uma das combinações de entrada é chamada de *mintermo*, e a função de chaveamento é obtida como uma soma de *mintermos*. Para a construção do circuito da equação acima associamos à operação produto a porta *and*, e à operação soma a porta *or*. A Figura 15 ilustra a construção do circuito combinacional usando a equação na forma canônica.

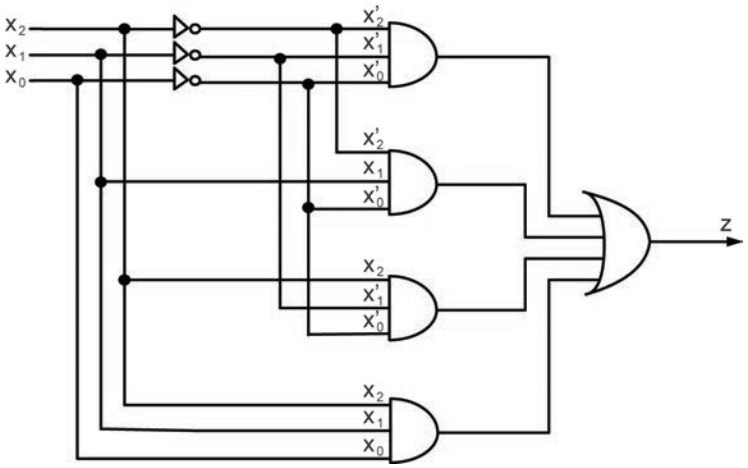


Figura 15 Construção do circuito combinacional da Figura 14.

Assim, qualquer circuito combinacional pode ser construído a partir de uma tabela-verdade, obtendo-se sua função de chaveamento na forma canônica. Construir um circuito combinacional na sua forma canônica implica, normalmente, num circuito grande, sem simplificações.

Evidentemente, essa expressão da função de chaveamento pode ser simplificada usando as propriedades vistas no item anterior.

2.3.5 Equivalência de circuitos

Um circuito é equivalente a outro por representar a mesma função lógica. Utilizando os postulados e as propriedades da álgebra booleana é possível encontrar uma função booleana simplificada equivalente. Ou seja, se dois circuitos chegam à mesma função booleana simplificada, então são equivalentes. Além disso, se compararmos as tabelas-verdade veremos que para as mesmas entradas elas apontam para as mesmas saídas.

Por exemplo, considerando os circuitos das Figuras 16 e 17, após algumas operações de simplificação, é possível verificar se a expressão de chaveamento do circuito da Figura 16 equivale ao circuito simplificado da Figura 17.

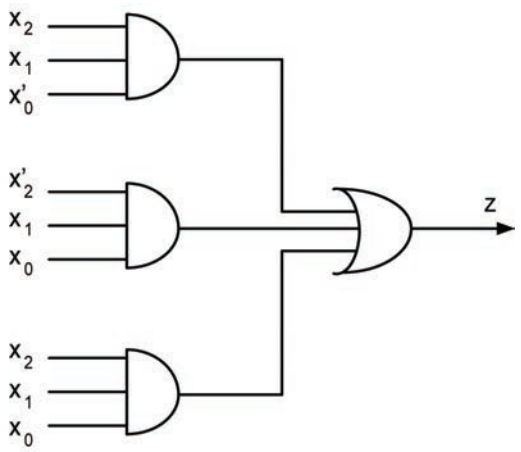


Figura 16 Circuito combinacional em forma canônica (o desenho dos circuitos inversores foram omitidos).

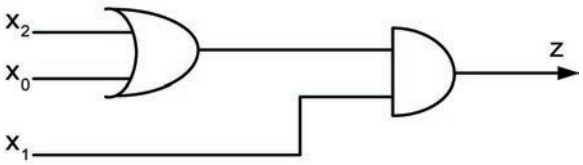


Figura 17 Circuito combinacional equivalente ao da Figura 16.

2.4 Circuitos sequenciais

Os circuitos sequenciais são diferentes dos combinacionais pela existência de estados internos, combinações de *bits* que ficam armazenados internamente.Como o circuito de saída, assim como o de entrada, faz uso dessa combinação de *bits*, um valor de saída não pode ser obtido em função de uma simples tabela-verdade, como nos circuitos combinacionais.

Dependendo dos instantes de tempo em que as entradas e saídas são consideradas, os circuitos sequenciais podem ser classificados como síncronos e assíncronos. Os circuitos síncronos utilizam um sinal de sincronização denominado *clock* (CLK) ou relógio, que define o tempo como uma variável discreta. A Figura 18 apresenta uma variável de saída $z(t)$, sendo alterada em função da variável de entrada $x(t)$, sincronizada com o *clock*. A variável $z(t)$ muda do estado 1 para 0, na borda de descida do *clock* no instante 1; do estado 0 para 1, na borda de descida do *clock* no instante 3, em função da variável $x(t)$.

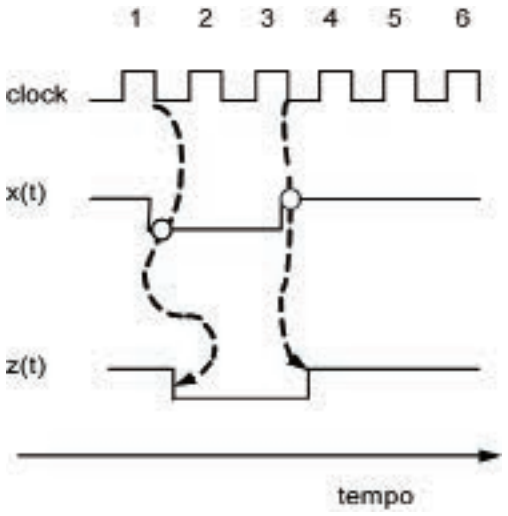


Figura 18 Relação entre a entrada x(t) e a saída z(t) sincronizada com o *clock*. As linhas tracejadas representam as posições do *clock* quando a saída z(t) altera em função da entrada x(t).

Os circuitos assíncronos não usam o *clock*. A variável de saída $z(t)$ varia em função da entrada $x(t)$, sem discretizar o tempo, como mostra a Figura 19.

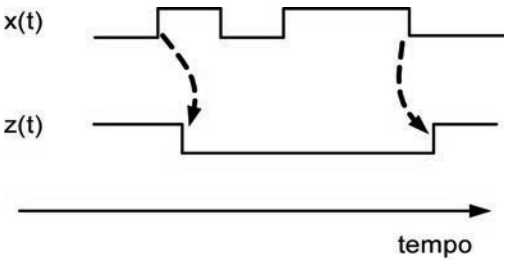


Figura 19 Relação entre a entrada x(t) e a saída z(t) num circuito assíncrono.

A descrição seguinte refere-se aos elementos de estado, que são os circuitos que armazenam os *bits*, e à forma de construção dos circuitos sequenciais síncronos, uma vez que sistemas complexos como os computadores normalmente usam circuitos síncronos.

2.4.1 Elementos de estado

Latches e *flip-flops* são células binárias utilizadas em circuitos sequenciais. A diferença básica entre elas é o momento de acionamento de um novo estado da célula. Nos *latches* uma célula é sensível ao nível (ou seja, o acionamento ocorre quando a mudança no *clock* está consolidada), enquanto nos *flip-flops* ela ocorre nas bordas de subida ou descida do *clock* (sensível à borda).

A Figura 20 mostra um *latch* SR constituído de dois circuitos *nor*. Quando as suas entradas S (*set*) e R (*reset*) têm valores zero, o *latch* SR apresenta o comportamento de dois inversores interligados, como na Figura 21, em que a saída $Q = 0$, ou como na Figura 22, em que a saída $Q = 1$. Os estados dos circuitos das Figuras 21 e 22 são estáveis, ou seja, o valor de Q nunca se altera, ficando sempre no estado atual.

No *latch* SR, a mudança de estado é feita introduzindo o valor 1 em uma das entradas S ou R. Assim, se introduzirmos o valor 1 em S, a saída Q passa para o valor 1. Se introduzirmos o valor 1 em R, a saída Q passa para o valor zero. A introdução do valor 1 em S e R simultaneamente não é permitida no *latch* SR, pois isso implica fazer $Q = 1$ e $Q = 0$ ao mesmo tempo, o que não faz sentido. Além disso, o estado resultante do circuito, nesse caso, é incerto.

Observa-se que nos *latches* e *flip-flops* temos saídas Q e Q'. Notamos pelos circuitos das Figuras 21 ou 22 que Q e Q' ficam em posições simétricas e separadas por um inversor. Portanto, o circuito sempre se estabiliza com Q e Q' contendo valores invertidos entre si.

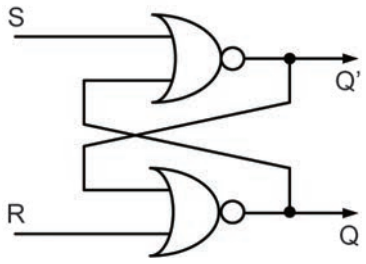


Figura 20 *Latch* SR.

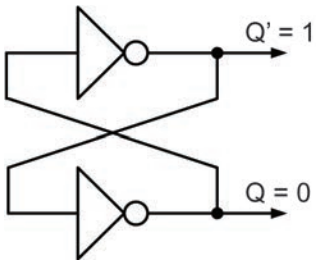


Figura 21 Um possível comportamento do *Latch* SR, quando as entradas S e R são iguais a zero, com $Q' = 1$ e $Q = 0$.

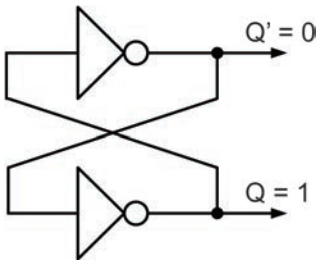


Figura 22 Outro possível comportamento do *Latch* SR, quando as entradas S e R são iguais a zero, com $Q' = 0$ e $Q = 1$.

Os *latches* e *flip-flops* abaixo serão representados por seus símbolos, funções lógicas e tabelas-transição. Vale ressaltar que $Q(t)$ corresponde ao estado atual e $Q(t+1)$ ou $Q(t+t_p)$ corresponde ao próximo estado.

2.4.2 *Latches* e *flip-flops*

O *latch D* é denominado *latch* controlado. O seu símbolo lógico (Figura 23), a sua implementação (Figura 24) e a sua função são apresentados a seguir.

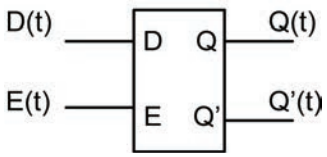


Figura 23 Símbolo lógico do *latch D*.

A sua equação de saída é dada por:

$Q(t+t_p) = D(t) \cdot E(t) + Q(t) \cdot E'(t)$, em que t_p é o tempo de atraso de propagação.

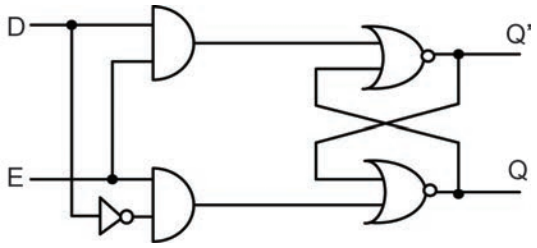


Figura 24 Implementação do *latch D*.

Flip-flop D

O *flip-flop D* é acionado na borda do *clock* (subida ou descida). O símbolo lógico é mostrado na Figura 25.

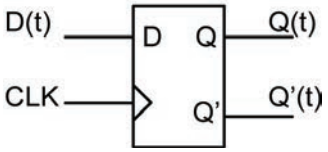


Figura 25 Símbolo lógico do *flip-flop D.*

O *flip-flop D* pode ser construído usando dois *latches D*, como mostra a Figura 26.

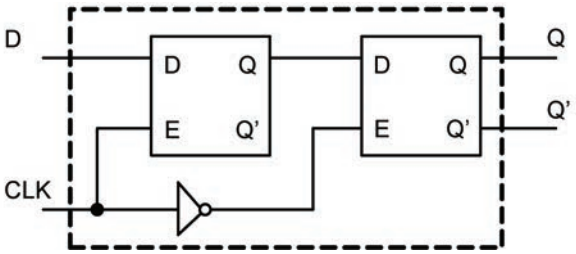


Figura 26 Construção de um *flip-flop D* usando dois *latches D*.

A sua equação de saída é dada por: $Q\left(t+t_p\right)=D\left(t\right)$. Nota-se que a saída $Q\left(t+t_p\right)$ do circuito é função da entrada $Q\left(t\right)$, independente do estado atual $Q\left(t\right)$.

Na Figura 27 um diagrama de tempo é representado ilustrando os momentos em que a saída *Q*, do *flip-flop D* da Figura 23, muda de estado, instante da borda de descida do *clock* (CLK).

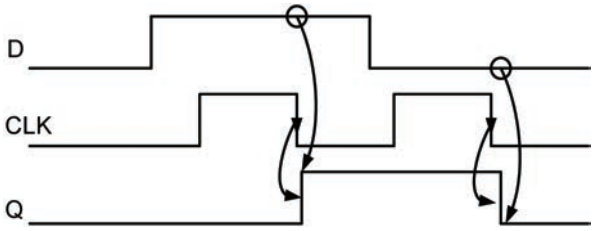


Figura 27 Diagrama de tempo para o *flip-flop D*.

2.4.3 Projeto de um circuito sequencial

Como descrito inicialmente, a saída de um circuito sequencial depende das entradas e do estado interno. Vamos exemplificar um circuito sequencial síncrono projetando um circuito contador que usa dois *flip-flops D*, cuja sequência de saída é dada por $00 \rightarrow 01 \rightarrow 10 \rightarrow 11 \rightarrow 00 \ldots$, e assim por diante. O esboço desse circuito contador é ilustrado na Figura 28.

As saídas do circuito são Q_1 e Q_0. O circuito não apresenta nenhuma entrada, a não ser o *clock* (CLK), que é um sinal de sincronização. Como um *flip-flop D* copia o valor da entrada D para a saída Q na borda do *clock*, o projeto do circuito contador consiste basicamente em projetar o circuito combinacional tendo como entradas os valores atuais de Q_1 e Q_0, obtendo-se os próximos valores. Esse circuito é resultado da sua tabela-verdade (Tabela 6).

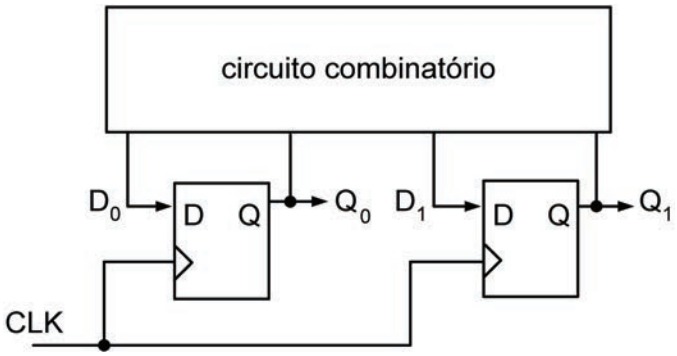


Figura 28 Esboço de um circuito contador que usa dois *flip-flops D*.

Tabela 6 Tabela-verdade para o circuito combinacional do circuito contador.

estado atual $Q_1\ Q_0$	$D_1\ D_0$
0 0	0 1
0 1	1 0
1 0	1 1
1 1	0 0

Nota-se que esse circuito combinacional pode ser entendido como duas funções de chaveamento, uma para D_0 (Figura 27) e outra para D_1 (Figura 30). A Figura 31 mostra o exemplo de circuito sequencial completo do contador.

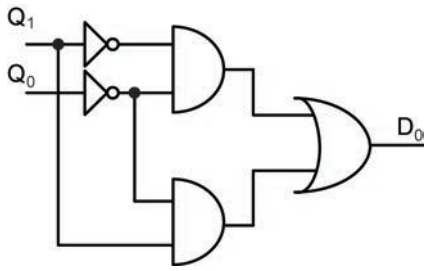


Figura 29 Circuito combinacional para D_0.

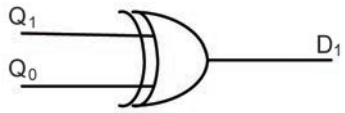


Figura 30 Circuito combinacional para D_1.

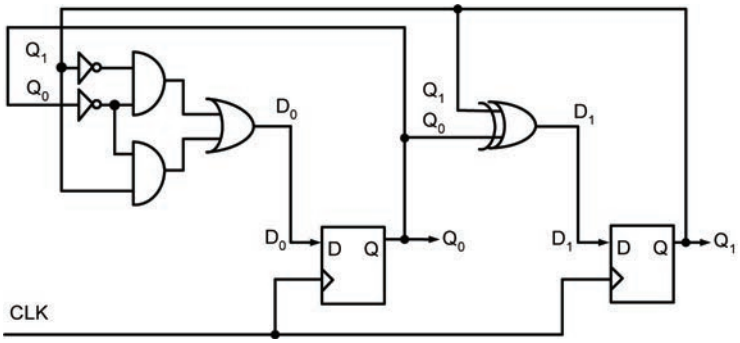


Figura 31 Circuito contador completo.

2.5 Considerações finais

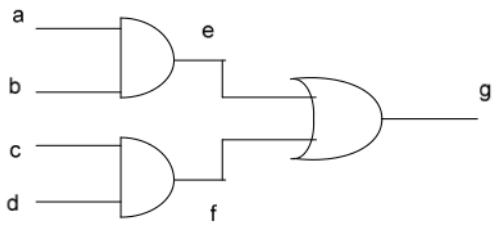
Circuitos combinacionais são úteis em projetos de unidades aritméticas e estruturas como somadores e comparadores. Circuitos sequenciais são utilizados em estruturas de controle. Dessa forma, unindo circuitos combinatórios e sequenciais, conseguimos projetar as organizações de computadores.

2.6 Estudos complementares

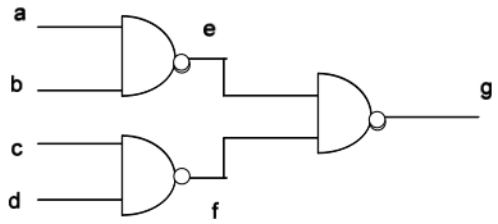
Para complementar os estudos sobre os circuitos digitais, sugere-se a leitura do terceiro capítulo, *Nível Lógico Digital*, do livro de Tanenbaum (2007). Outra referência específica de circuitos digitais é o livro dos autores Ercegovac, Lang & Moreno (2000). O livro de Floyd (2007) também apresenta exaustivamente a matéria.

2.7 Exercícios

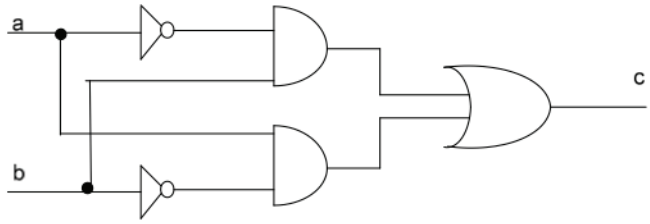
1. Dado um circuito combinacional de 3 entradas (a, b, c) e uma saída (d):
   a) Mostrar todas as combinações de valores binários (0/1) de entradas possíveis;
   b) Mostrar para cada uma das combinações, do item anterior, a expressão da sua representação na função de chaveamento.
2. Dada uma porta AND de 3 entradas (x_2, x_1, x_0) e saída (f), e mostrar a sua tabela-verdade.
3. Dada uma porta OR de 3 entradas (x_2, x_1, x_0) e saída (f), e mostrar a sua tabela-verdade.
4. Para a porta AND de 3 entradas da questão 2, mostrar a expressão da sua função de chaveamento.
5. Para a porta OR de 3 entradas da questão 3, mostrar a expressão da sua função de chaveamento.
6. Aplicando as identidades ou propriedades da álgebra booleana simplificar as expressões:
   a) $x_2' x_1' x_0 + x_2' x_1 x_0$
   b) $x_2' x_1 x_0' + x_2' x_1 x_0$
   c) $x_2 x_1' x_0' + x_2 x_1' x_0$
   d) $x_2 x_1 x_0' + x_2 x_1 x_0$
   e) $x_2 x_1' x_0 + x_2 x_1 x_0$
7. Idem, simplificar a função de chaveamento:
   a) $Z = x_2' x_0 + x_2' x_1 + x_2 x_1' + x_2 x_1 + x_2 x_0$
8. Dado o circuito com 2 portas AND de duas entradas e uma porta OR de duas entradas, conforme figura, obter a tabela verdade correspondente à saída g.



9. Mostrar que o circuito de 3 portas NAND da figura é equivalente ao circuito do exercício 8.



10. Mostrar que o circuito da figura é equivalente a uma porta XOR.



11. Dado um circuito digital com 3 entradas e 1 saída, cuja tabela-verdade é dada abaixo:
    a) Obter a função de chaveamento em forma canônica de c_o;
    b) Desenhar o circuito digital resultante usando portas lógicas.

a	b	c_i	c_o
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

12. Dada a função de chaveamento em forma canônica:

$$C_o = a'bc_i + ab'c_i + abc_i' + abc_i$$

c) Aplicar a idempotência sobre o último termo, replicando-o duas vezes;

d) Aplicar a propriedade distributiva (ao contrário) colocando em evidência os termos em comum, para cada par de termos formados por um dos 3 primeiros termos e o termo replicado no item (a), por ex.: $a'bc_i + abc_i = (a' + a)bc_i$

e) Aplicar as identidades: complementos e multiplicativo, e mostrar a expressão simplificada da função de chaveamento obtida.

13. Dada a função de chaveamento:

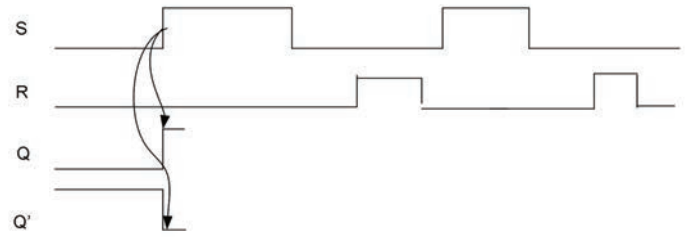
$$C_o = a \cdot b + (a \oplus b) \cdot c_i$$

a) Obter a tabela verdade dessa função;

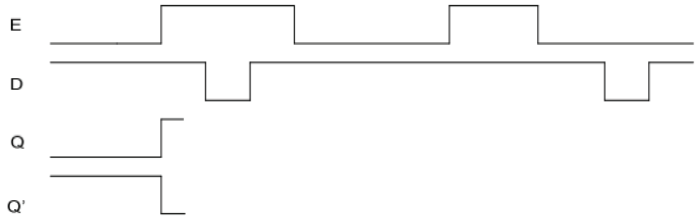
b) Verificar que o resultado é o mesmo da tabela do exercício 11;

c) Mostrar que a expressão simplificada obtida no exercício 12 é, também, equivalente.

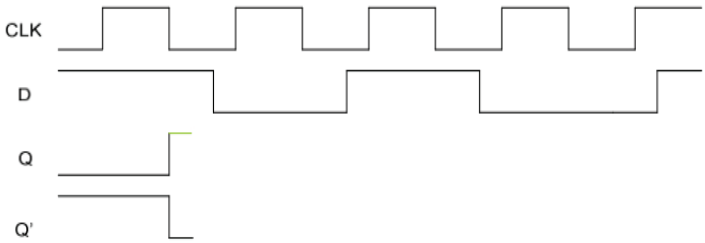
14. Dado o diagrama de tempo dos sinais de entrada e saída para um latch SR, completar o mesmo para as saídas Q e Q'.



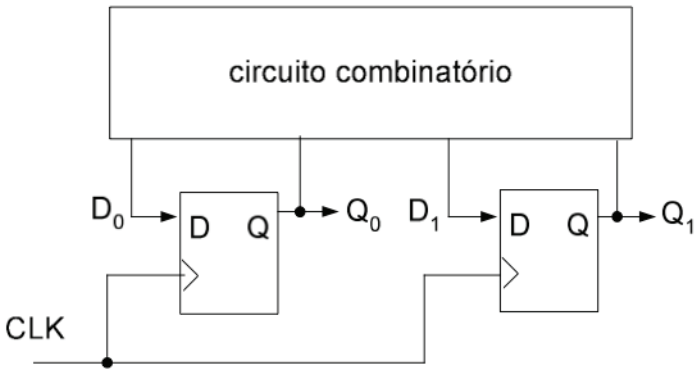
15. Dado o diagrama de tempo dos sinais de entrada e saída para um latch D, completar o mesmo para as saídas Q e Q'.



16. Dado o diagrama de tempo dos sinais de entrada e saída para um flip-flop D, de borda de descida, completar o mesmo para as saídas Q e Q'.



17. Dada a figura de um circuito sequencial, e dado que a sequência de saída, a cada borda de descida do clock, é de: 11 → 10 → 01 → 00 → 11, projetar o circuito combinatório para D_0 e D_1, mostrando o desenho dos mesmos. Nota-se que está sendo usado flip-flop D de borda de descida.



UNIDADE 3

Linguagem de máquina:

interface *hardware*/*software*

3.1 Primeiras palavras

A Unidade 3 descreve a função do computador quanto à execução de instruções típicas envolvendo unidades funcionais que compõem o *hardware*. Para a descrição das instruções típicas é usada a linguagem *Assembly* e a linguagem de máquina.

3.2 Problematizando o tema

A presente unidade refere-se ao estudo da interface entre o *software* e o *hardware*. O *software* corresponde aos programas que são escritos em linguagens de programação e pode ser encontrado em vários níveis de abstração. O nível mais próximo da máquina é a linguagem de máquina, cujas instruções ocorrem em nível de *bits*. Para que a linguagem fique mais compreensível para o programador, foi inventada uma linguagem que representa em mnemônicos as instruções do computador. Essa é a linguagem *Assembly*. Nesta unidade veremos algumas instruções típicas de computadores em linguagem *Assembly* e linguagem de máquina. Como essas instruções dependem da máquina, vamos adotar um computador denominado MIPS, projetado na Universidade de Stanford, atualmente usado em muitos produtos comercialmente disponíveis.

A arquitetura básica de um computador moderno usa, normalmente, o conceito de programa armazenado, significando que o programa fica armazenado na memória e o processador faz a leitura das instruções contidas no programa, uma a uma, e as executa na sequência de armazenamento.

O processador, ao ser iniciado, começa com a leitura de uma instrução na memória e a sua execução. Aqui surge outra questão. Que instruções o processador vai ler da memória, se a mesma acaba de ser energizada? A resposta é que existe uma memória do tipo não volátil, que não perde o conteúdo quando desligada a energia. O processador começa lendo as instruções dessa memória, que já tem um conteúdo armazenado ao ser energizada.

Cada uma das instruções deve realizar uma pequena parte da tarefa atribuída ao programa em execução pelo computador. É claro que para se executar as tarefas, conforme as necessidades do usuário, o *software* ou o programa deve conter instruções que acabam realizando, passo a passo, todos os detalhes da tarefa.

3.3 Linguagem de máquina e linguagem *Assembly* do MIPS

Serão apresentadas as instruções típicas do MIPS, quanto a suas funções, seus operandos e formatos, na linguagem de máquina e na linguagem *Assembly*.

A linguagem *Assembly* fica num nível intermediário entre a linguagem de alto nível C e a linguagem de máquina. Ao final da unidade, o leitor terá visto, também, o aspecto funcional do computador MIPS.

3.3.1 Linguagem de máquina, *Assembly* e C

O processador faz a leitura de uma instrução na memória e a executa. Considerando um processador de *32 bits*, uma máquina binária, podemos exemplificar uma instrução lida na memória como sendo uma cadeia de *bits*, por exemplo:

```
00000000101000010000000000011000
```

Esse é um exemplo de uma instrução no código de máquina, linguagem de máquina, ou código binário.

Para se escrever um programa, deve estabelecer uma sequência de instruções que compõem o mesmo. Um exemplo dessa sequência é o seguinte:

```
00000000101000010000000000011000
10001100011000100000000000000000
10001100111100100000000000000100
10101100111100100000000000000000
10101100011000100000000000000100
00000011111000000000000000001000
```

Como a máquina está projetada para executar tais instruções, essa é a linguagem mais adequada para ela. Para o programador, no entanto, é difícil escrever um programa na linguagem de máquina. Se ele quiser escrever um programa nessa sequência de instruções, existe um *software* chamado programa montador (*assembler*, em inglês) que permite escrever para cada linha as instruções em forma mais compreensível, sendo possível perceber qual operação será executada. Essa linguagem é denominada linguagem montadora (*Assembly*, em inglês), e o programa *assembler* recebe, como entrada, um programa nessa linguagem e o converte em linguagem de máquina. O programa em *Assembly* correspondente ao programa anterior, para o computador MIPS, é o seguinte:

```
swap: muli  $2,     $5,   4
      add   $2,     $4,   $2
      lw    $15,  0($2)
      lw    $16,  4($2)
      sw    $16,  0($2)
      sw    $15,  4($2)
      jr    $31
```

Mesmo que a linguagem *Assembly* seja mais manipulável para o programador que a linguagem de máquina, ela é muito próxima da máquina, pois cada uma das linhas corresponde a uma instrução de máquina.

O primeiro campo corresponde ao mnemônico da instrução, que indica a operação a ser executada. Por exemplo, *add*, *lw* e *sw*. Os demais campos correspondem aos operandos e a sua definição depende das instruções. Na linguagem *Assembly*, existe a possibilidade de uso de símbolos, bem como rótulos (*labels*). Os rótulos são usados no início de uma linha, indicando a instrução, e são seguidos de dois-pontos. Exemplo: "*swap:*". Os símbolos substituem os números e facilitam a escrita do programa, porém devem ser definidos previamente. A linguagem *Assembly* é usada por programadores de sistemas para desenvolverem programas otimizados, por exemplo, *drivers* para manipulação de periféricos de computadores.

Normalmente, o desenvolvimento de *softwares* é feito usando linguagens num nível mais elevado, conhecidas como linguagens de alto nível, como a linguagem C. O programa em C, correspondente aos programas em *Assembly* e código de máquina anterior, é dado por:

```
swap (int v[ ], int k)
{
    int temp;
    temp = v[k];
    v[k] = v[k + 1];
    v[k + 1] = temp;
}
```

O *software* que converte um programa em linguagem de alto nível, como C, para a linguagem *Assembly* é chamado de compilador.

3.3.2 Ciclo de instruções

Num computador sequencial, ao ser executada uma instrução, o computador deve automaticamente fazer a busca da próxima instrução sequencial, executá-la, e assim por diante. Para fazer a busca da próxima instrução, o computador deve fornecer à memória o endereço da próxima instrução. Chamamos o componente que contém o endereço da próxima instrução de apontador de instrução (*instruction pointer*) ou contador de programa (*program counter*).

Podemos delinear o que se chama de ciclo de instrução, o conjunto completo de operações referentes a uma instrução, nas seguintes etapas:

- Etapa 1: ler uma instrução na memória usando o conteúdo do contador de programa como endereço;
- Etapa 2: ler os operandos no banco de registradores, indicados pela instrução lida na memória na etapa 1;
- Etapa 3: executar a operação aritmética sobre os operandos lidos na etapa 2;
- Etapa 4: escrever o resultado da operação aritmética (etapa 3) no registrador destino;
- Etapa 5: atualizar o conteúdo do contador de programa com o endereço da próxima instrução.

Observa-se que num ciclo de instrução a memória é lida para a busca da instrução. Essa etapa é também conhecida como *instruction fetch*, em inglês. Também, após a instrução ser executada, o contador de programa deve ser atualizado para conter o endereço da próxima instrução. Essas duas etapas são sempre presentes num ciclo de instrução. As demais etapas podem existir ou não.

3.3.3 As principais instruções do MIPS

Conforme exemplo do item 3.3.1, em linguagem *Assembly*, representado novamente aqui:

```
swap: muli  $2,     $5,   4
      add   $2,     $4,   $2
      lw    $15,   0($2)
      lw    $16,   4($2)
      sw    $16,   0($2)
      sw    $15,   4($2)
      jr    $31.
```

notamos algumas instruções típicas do MIPS, que serão descritas a seguir.

Uma instrução típica no MIPS é a aritmética, que usa três operandos registradores. Um registrador é um circuito contido no processador e, no caso do MIPS, permite armazenar uma palavra de *32 bits*. Os registradores são denotados com o prefixo *$* e um número. O número identifica o registrador de 0 a 31, contido no banco de registradores.

No exemplo, a instrução aritmética é escrita *add $2, $4, $2*. Isso significa que o conteúdo do registrador *$4* deve ser somado ao conteúdo do registrador *$2* e guardado no registrador destino *$2*. Nota-se que o registrador destino é o primeiro que aparece na instrução. No exemplo, o registrador *$2* é também usado como registrador de operando fonte, que aparece no final da instrução.

Outra instrução muito usada é a instrução *lw* (*load-word*), que faz a leitura de um dado na memória e guarda o dado lido num registrador. No exemplo, temos duas instruções *lw*. A primeira *lw $15, 0 ($2)* indica que o endereço de memória *0 + $2* deve ser lido, e o dado lido deve ser guardado no registrador *$15*. Nota-se que o endereço *0 + $2* é calculado usando o conteúdo do registrador base *$2* e somando com o valor 0. No MIPS, o valor a ser somado é obtido de *16 bits* contidos na instrução.

Mais um exemplo de instrução é *sw* (*store-word*), que escreve na memória um dado contido num registrador. No programa temos duas instruções *sw*, sendo a primeira *sw $16, 0 ($2)*, indicando que o conteúdo do registrador *$16* deve ser escrito na memória no endereço *0 + $2*.

No nosso programa, ainda temos uma instrução *muli*, que significa multiplicação com um dado imediato. Assim, *muli $2, $5, 4* significa que o conteúdo do registrador *$5* deve ser multiplicado pelo valor imediato *4*, e o resultado deve ser guardado no registrador *$2.*

A última instrução do nosso programa é *jr $31*, a qual indica que o contador de programa deve ser atualizado com o valor contido no registrador *$31* para executar a próxima instrução.

Além das instruções vistas, existem outras bastante relevantes no MIPS. São, por exemplo, as de controle de fluxo de instruções, das quais se destaca a instrução de desvio. Destas, temos duas principais: *bne* e *beq*. A instrução *bne* (*branch if not equal*) faz com que o programa desvie, caso os valores de dois registradores não sejam iguais. Exemplo:

```
bne     $3,     $4,     5
```

A instrução de desvio compara o conteúdo dos registradores *$3* e *$4* e, caso não resultar em igualdade, desvia 5 instruções.

A instrução *beq* (*branch if equal*) é semelhante à anterior, porém, desvia caso o conteúdo dos dois registradores for igual.

As instruções *bne* e *beq* são de desvio condicional, ou seja, quando uma condição é satisfeita, ocorre um desvio no programa. Essas instruções são imprescindíveis em computadores para a tomada de decisões.

Outra instrução importante é a *jump* ou apenas *j*. Essa instrução faz o desvio sem verificar qualquer condição. É, portanto, um desvio incondicional. Exemplo:

```
j       200
```

desvia para a instrução *200*.

Outra instrução importante do MIPS é a *slt* (*set-on-less-than*). Ela compara dois registradores, por exemplo, *$a* e *$b,* e caso *$a* < *$b*, guarda o valor *1* no registrador destino; caso contrário, guarda o valor *0*. Exemplo:

```
slt     $2,     $3,     $4
```

se *$3* < *$4*, guarda *1* em *$2*; caso contrário, guarda *0*. A instrução *slt* usa três registradores e, portanto, classifica-se como uma instrução aritmética.

Há também a classe de instruções de valores constantes, ou valores imediatos. Por exemplo, a instrução *addi* (soma imediata) faz uma operação de adição do conteúdo de um registrador com o conteúdo imediato. Ou seja,

```
addi    $2,     $3,     1
```

faz com que o conteúdo do registrador *$3* seja somado com o valor imediato *1*, e o resultado é guardado no registrador *$2*. Outros exemplos de instruções com operandos imediatos são: *ori*, *andi* e *slti*, que são *or* imediato, *and* imediato e *slt* imediato, respectivamente.

3.3.4 Formato das principais instruções do MIPS em linguagem de máquina

Neste item veremos os formatos das instruções do MIPS em linguagem de máquina. Notemos que todas as instruções têm um formato de *32 bits*. Esses *32 bits* são divididos em campos, sendo o primeiro campo constituído de *6 bits*, destinado para o *opcode*. A definição dos demais campos depende da instrução. Basicamente, veremos três tipos de formatos: formato R, formato I e formato J.

Formato R: a primeira instrução vista no item anterior é a instrução aritmética, usando três operandos registradores, sendo dois deles operandos fonte e um operando destino. O formato dessas instruções aritméticas é chamado R, devido ao uso apenas de registradores. Na Figura 32 temos um diagrama ilustrativo do formato R. Nota-se que os três registradores são apontados por campos de *5 bits* denotados *rs*, *rt* e *rd*, respectivamente. O primeiro campo é denotado *op* (*operation code*, ou *opcode*), o seguinte aos registradores é *shamt*, que indica a quantidade de deslocamento para instruções de *shift* (não usadas neste livro) e, finalmente, o campo *function*, que indica a função aritmética a ser realizada. No caso do MIPS, para uma instrução aritmética, no formato R, podemos ter as seguintes funções: *add*, *sub*, *and*, *or* e *slt*. Essas funções são, respectivamente, adição, subtração, operação lógica *and*, operação lógica *or* e operação *set-on-less-than*.

op	rs	rt	rd	shamt	function
000000	*10001*	*10010*	*01000*	*00000*	*100000*
op	*rs*	*rt*	*rd*	*shamt*	*function*

Figura 32 Formato R, para uma instrução aritmética.

Formato I: o formato I é usado pela maioria das instruções do MIPS: *load-word*, *store-word*, *bne*, *beq*, *addi*, *muli* e outras.

A instrução de memória *load-word* usa dois registradores e um número, por exemplo:

lw $2, 4($15)

O formato dessa instrução, já separado em campos, é dado pela Figura 33.

op	rs	rd	16 bits
100011	*01111*	*00010*	*0000000000000100*
op	*rs*	*rd*	*deslocamento*

Figura 33 Formato para uma instrução *lw*.

Nesse caso, o campo de operação *op* indica explicitamente a operação *lw*, com o valor *100011*.

A instrução de memória *store-word* usa dois registradores e um número, como na instrução *load-word*:

sw $3, 4($15)

O formato dessa instrução é dado pela Figura 34.

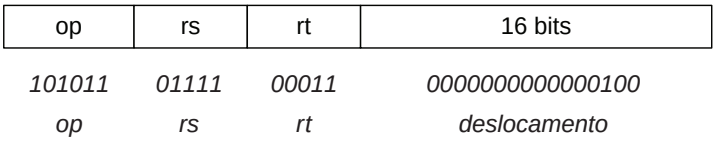


Figura 34 Formato para uma instrução *sw*.

Nesse caso, o campo de operação *op* indica explicitamente a operação *sw*, com o valor *101011*.

A instrução de desvio *bne* usa dois registradores e um número:

```
bne     $3,     $5,     4
```

O formato dessa instrução, separado em campos, é dado pela Figura 35.

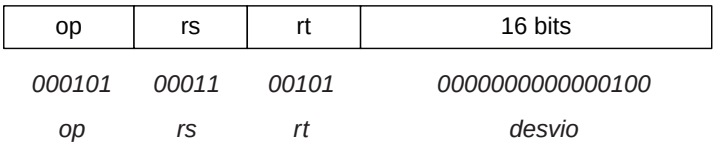


Figura 35 Formato da instrução *bne*.

Os registradores *rs* e *rt* são comparados e, caso não sejam iguais, o contador de programa é atualizado com o valor atual acrescido de quatro posições de instruções.

O formato da instrução *beq* é igual ao da instrução *bne*, apenas difere no valor do código de operação, *op*.

O formato da instrução *addi* (soma imediata), bem como da instrução *muli* (multiplicação imediata), é também igual ao formato *I*.

Formato J: esse formato é usado pela instrução de desvio incondicional jump, *j*.

O formato da instrução de desvio incondicional *j* é dado pela Figura 36.

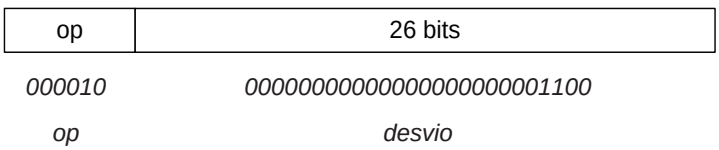


Figura 36 Formato da instrução *j*.

O primeiro campo é o código de operação com o valor *000010*, que identifica a instrução *j*.

Como o campo de código de operação contém *6 bits*, o restante da instrução é o endereço de desvio de *26 bits*.

3.3.5 Considerações sobre as características do MIPS

O MIPS é um computador que usa a arquitetura RISC (*Reduced Instruction Set Computer*), cuja característica inclui o uso de pequena quantidade de formatos de instruções. Nesse sentido, vimos o uso apenas de três formatos de instruções: formato R, formato I e formato J. Outra característica é o uso de instruções aritméticas com operandos em registradores, como em instruções de formato R e operandos imediatos. As operações de referência à memória são apenas *lw* (*load-word*) e *sw* (*store-word*). Nos computadores RISC costuma-se usar um conjunto grande de registradores para guardar as variáveis. No MIPS é usado um conjunto de 32 registradores, de *32 bits*. O uso desses registradores é predefinido, conforme mostra a Tabela 7. Assim, além do uso do número dos registradores para a sua identificação, é possível usar as denominações citadas na primeira coluna da tabela, quando se usa a linguagem *Assembly* do MIPS. Uma observação importante é que o registrador *$0* ou *$zero* contém o valor constante *0*. Esse é o único registrador cujo valor não pode ser alterado.

Tabela 7 Notação dos registradores no *Assembly* do MIPS.

Notação	Número do registrador	Uso
$zero	0	Valor constante 0
$v0 - $v1	2-3	Resultados
$a0 - $a3	4-7	Argumentos
$t0 - $t7	8-15	Temporários
$s0 - $s7	16-23	Endereços
$t8 - $t9	24-25	Temporários
$gp	28	*Global pointer*
$sp	29	*Stack pointer*
$fp	30	*Frame pointer*
$ra	31	*Return address*

Podemos perguntar: o que acontece se, num programa, o número de variáveis é maior do que o número de registradores? Obviamente uma forma é guardar esses valores na memória principal. A memória é vista pelo processador MIPS como um vetor linear de *bytes* (*8 bits*). A memória é endereçada pelo processador por um endereço de *32 bits*. Isso significa que a memória contém 2^{32} *bytes*, cujo endereço varia de *0* a $2^{32} - 1$.

Como as instruções têm *32 bits*, usam *4 bytes*. Uma palavra de dado, lida na memória por uma instrução *load-word*, ou uma palavra escrita por uma instrução *store-word*, usa também *4 bytes*. Assim, tanto as instruções como os dados são armazenados na memória a partir dos endereços de memória múltiplos de *4*. A Figura 37 mostra a organização das palavras de *32 bits* na memória, em posições de *4* em *4 bytes*.

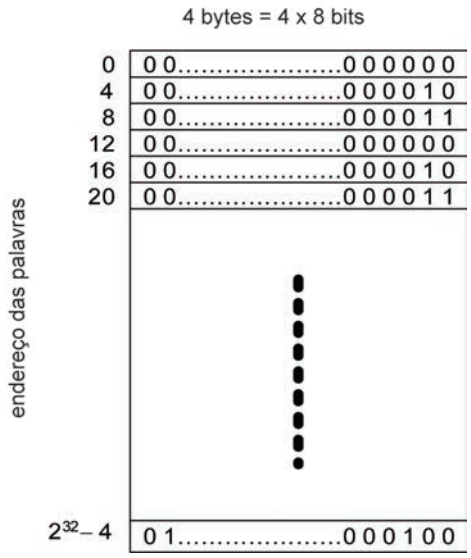


Figura 37 Memória como um vetor linear de palavras. O endereço das palavras varia em múltiplos de *4*.

Em *Assembly* do MIPS, uma variável simples é sempre considerada como contida num registrador. Quando a variável é um conteúdo de memória, usa-se uma variável indexada. Assim, a variável $h(0)$ significa o primeiro elemento do vetor *h*, na memória. A variável $h(1)$ significa o segundo elemento do vetor *h*. Se o primeiro elemento $h(0)$ é guardado a partir do endereço *16* da memória, $h(1)$ é guardado a partir do endereço *20*, $h(2)$ no endereço *24*, e assim por diante.

3.3.6 Exemplos de programas em *Assembly* do MIPS

Exemplo 1

Seja a tarefa de somar variáveis *b*, *c*, *d* e *e* na variável *a*. Como cada instrução de adição faz uma soma de duas variáveis, precisamos realizar a tarefa em algumas instruções. Uma solução é dada por:

add a, b, c	*# soma*	$b + c$	*é colocada em a*
add a, a, d	*# soma*	$b + c + d$	*é colocada em a*
add a, a, e	*# soma*	$b + c + d + e$	*é colocada em a*

em que *a*, *b*, *c*, *d* e *e* são variáveis que estão contidas em registradores. Na notação de registradores, a sequência de instruções seria, por exemplo:

```
add $4, $5, $6    # b + c = a,  $5 + $6 = $4
add $4, $4, $7    # a + d = a,  $4 + $7 = $4
add $4, $4, $8    # a + e = a,  $4 + $8 = $4
```

considerando-se que *$4* corresponde à variável *a*, *$5*, à variável *b*, *$6*, à variável *c*, *$7,* à variável *d* e *$8*, à variável *e*.

Exemplo 2

Para executar a instrução em código C,

$$A[8] = h + A[8],$$

um exemplo de programa em *Assembly* seria:

```
lw    $t0,    32($s3)
add   $t0,       $s2,     $t0
sw    $t0,    32($s3)
```

em que a variável *h* é contida no registrador *$s2*. A memória contém os vetores de dados, sendo $A[8]$ um elemento do vetor com índice *8*. Como o índice é *8*, e a memória é endereçada em *bytes*, devemos multiplicar o índice por *4*, resultando em *32*. Assim, o primeiro elemento do vetor $A[0]$ é dado pelo conteúdo do endereço $(0 + \$s3)$, bem como o elemento $A[8]$ é dado pelo conteúdo do endereço $(32 + \$s3)$.

Exemplo 3

Analisemos agora o nosso primeiro exemplo, do programa em C:

```
swap (int v[ ], int k)
{
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}.
```

Como o próprio nome diz (*swap* significa troca), o programa faz a troca dos valores contidos na memória nas posições $v[k]$ e $v[k+1]$.

O programa em *Assembly* consiste na seguinte sequência de instruções:

```
swap :  muli    $2,     $5,     4
        add     $2,     $4,     $2
        lw      $15,    0($2)
        lw      $16,    4($2)
        sw      $16,    0($2)
        sw      $15,    4($2)
        jr      $31
```

em que a primeira linha corresponde ao rótulo com o nome do programa (*swap*) e não faz parte das instruções.

Para facilitar o estudo, vamos analisar o programa pelas instruções de memória, que estão no meio da sequência:

```
lw      $15,    0($2)
lw      $16,    4($2)
sw      $16,    0($2)
sw      $15,    4($2)
```

Como podemos observar, as duas instruções *lw* fazem a leitura dos dados da memória nos endereços $0(\$2)$ e $4(\$2)$, e carregam os registradores *$15* e *$16*, respectivamente. As duas instruções *sw* fazem a escrita dos dados contidos nos registradores *$16* e *$15*, e os escrevem na memória nos endereços $0(\$2)$ e $4(\$2)$. Portanto, o resultado do processamento dessas quatro instruções consiste na troca do conteúdo de memória dos endereços $0(\$2)$ e $4(\$2)$. Essa troca em linguagem C é verificada nas instruções:

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

em que *temp* é uma variável temporária. Não existe correspondência direta entre a variável *temp* e os registradores usados, porém, é possível notar que existe uma correspondência entre o trecho do programa em C acima, com as quatro instruções de referência à memória. Portanto, podemos concluir que $v[k]$ corres-

ponde a $0(\$2)$, e $v[k+1]$ corresponde a $4(\$2)$. A partir dessa conclusão, podemos analisar a função das duas instruções no início do programa:

```
muli    $2,    $5,    4
add     $2,    $4,    $2
```

que correspondem à passagem de parâmetros contidos na instrução

$swap\,(int\ v[\ ], int\ k)$.

Lembrando que as palavras subsequentes de memória no MIPS têm endereços diferentes de *4*, dado um elemento do vetor de memória $v[k]$, com índice *k*, correspondente ao endereço $0(\$2)$, a palavra subsequente $v[k+1]$ deve ter endereço $4(\$2)$. Por outro lado, o primeiro elemento e o elemento *k* devem diferir em endereço de $4 \cdot k$. Podemos concluir, portanto, que a primeira instrução

```
muli    $2,    $5,    4
```

está realizando a multiplicação $4 \cdot k$, portanto, *$5* corresponde ao índice *k*.

A segunda instrução, *add $2, $4, $2*, faz a adição do conteúdo do registrador *$4* com o conteúdo do registrador *$2*. Podemos concluir que o conteúdo do registrador *$4* corresponde ao endereço inicial do vetor *v*.

Concluindo a análise do programa, a última instrução

```
jr      $31
```

corresponde à instrução que retorna o computador para onde o procedimento analisado foi chamado.

Exemplo 4

Para se implementar uma instrução condicional do tipo $if\,(i == j)\,h = i + j$ em linguagem C, a sequência em *Assembly* é dada por:

```
        bne     $s0,   $s1,    Label
        add     $s3,   $s0,    $s1
Label:  ...
```

Exemplo 5

Seja a implementação do código em C:

```
if (i != j)
   h = i + j;
else
   h = i – j;
```

Em linguagem *Assembly*, a implementação resulta em

```
        beq   $s4,  $s5,  Lab1
        add   $s3,  $s4,  $s5
        j     Lab2

Lab1:   sub   $s3,  $s4,  $s5
Lab2:   ...
```

A variável *i* é representada por *$s4* e a variável *j* por *$s5*. Caso os dois registradores sejam iguais, o fluxo de instruções desvia para *Lab1*, que aponta para a instrução *sub $s3, $s4, $s5*, em que *$s3* representa a variável *h*. Se a condição de igualdade não for satisfeita, não há desvio, portanto, é executada a instrução sequencial: *add $s3, $s4, $s5*. No entanto, após executar essa instrução, existe a necessidade de desviar para *Lab2*; caso contrário, a instrução referente a *Lab1* será também executada. O desvio para *Lab2* é feito usando a instrução *j Lab2*.

Exemplo 6

O programa seguinte zera o vetor *A*, de *n* elementos. O registrador *$4* é o ponteiro para o início do vetor *A*.

```
        sub   $2,    $2,     $2     # zera $2
L1:     muli  $14,   $2,     4      # offset = i * 4
        add   $3,    $4,     $14    # $3 = end A[i]
        sw    $0,    0($3)           # A[i] = 0
        addi  $2,    $2,     1      # i = I +1
        slt   $7,    $2,     $6     # se i < n   $7 =1
        bne   $7,    $0,     L1     # desvia se $7 <> 0
```

A parte importante deste exemplo é o uso da instrução *slt* para verificar o valor do índice contido no registrador *$2*. Caso o conteúdo de *$2* for menor

que o conteúdo de *$6*, será guardado o valor *1* no registrador *$7*. Em seguida, compara-se o valor do registrador *$7* com o valor 0, contido no registrador *$0*. Caso os valores não sejam iguais, desvia para *L1*, pois ainda *i* < *n*.

3.4 Considerações finais

Nesta unidade foi apresentada a interface *hardware/software* do computador MIPS. Foram vistas as instruções típicas do computador em nível de máquina, e as suas correspondências com as instruções na linguagem *Assembly*. Foram vistos os aspectos funcionais do computador MIPS na execução dessas instruções.

3.5 Estudos complementares

Para complementar os estudos sobre a interface *hardware/software*, os leitores podem se reportar ao segundo capítulo, *Instructions: Language of the Computer*, do livro de Hennessy & Patterson (2008), ou buscar o mesmo capítulo na versão traduzida por Daniel Vieira (HENNESSY & PATTERSON, 2005).

3.6 Exercícios

1. Dada a instrução muli $14, $2, 4, mostrar o conteúdo dos registradores após a execução da mesma, sendo que o valor inicial do registrador $2 é igual a 0, e de $14 é igual a 3.
2. Dada a instrução add $3, $4, $14, mostrar o conteúdo dos registradores envolvidos após a execução da mesma, sendo que o valor inicial de $3 é igual a 0, de $4 é igual a 68, e de $14 é igual a 0.
3. Dada a instrução sw $0, 0($3), mostrar o conteúdo de memória no endereço 68, após a sua execução, dado que o registrador $3 contém o valor 68.
4. Dada a instrução addi $2, $2, 1, mostrar o conteúdo do registrador $2, após a execução da mesma, considerando-se que o conteúdo inicial de $2 é igual a 0.
5. Dada a instrução slt $7, $2, $6, dar o conteúdo do registrador $7, após a sua execução, se o conteúdo de $2 é igual a 1 e de $6 é igual a 3.
6. Dada a instrução bne $7, $0, L1, verificar se a instrução seguinte é sequencial, ou do endereço L1, dado o conteúdo de $7 igual a 1.
7. Escrever a instrução, ou menor conjunto de instruções em Assembly do MIPS, que realiza a operação em código C, seguintes:
   a) a = a – 1
   b) a = 0

c) v[10] = 0

d) a = v[10]

e) if (a < b) goto L1;

f) if (a > 0) goto L1;

8. Dada a instrução add $4, $5, $6 preencher os campos no diagrama da figura abaixo.

op	rs	rt	rd	shamt	function
000000				00000	100000

9. Dada a instrução lw $4, 5($3) preencher os campos no diagrama da figura abaixo.

op	rs	rd	offset
100011			

10. Dada a instrução bne $3, $4, 5 preencher os campos no diagrama da figura abaixo.

op	rs	rd	offset
000101			

11. Dado o programa a seguir, em Assembly do MIPS, executá-lo passo a passo, mostrando os valores de registradores e memória a cada iteração.

```
.data
v :
.word  1, 2, 3, 4, 5, 6, 7, 8, 9, 10
.text
        li      $3,     0
        li      $2,     10
        la      $4,     v
L1 :
        lw      $5,     ($4)
        add     $3,     $3,     $5
        sw      $3,     ($4)
        add     $4,     $4,     4
        sub     $2,     $2,     1
        bgtz    $2,     L1
```

12. Escrever um programa em Assembly do MIPS que faz a soma de dois vetores A e B, colocando o resultado em A, ou seja, $A[i] = A[i] + B[i]$, para i = 0 .. 10.

13. Escrever um programa em Assembly do MIPS que imprime os 10 primeiros elementos da série de Fibonacci (0, 1, 1, 2, ...).

UNIDADE 4

Unidade lógica e aritmética no MIPS

4.1 Primeiras palavras

A Unidade 4 apresenta a função da unidade aritmética e a realização das operações aritméticas representadas por números naturais, inteiros e reais. Será vista uma implementação da unidade lógica e aritmética (ULA) para o computador MIPS.

4.2 Problematizando o tema

Os números nos computadores binários são representados por uma combinação de *bits*. A forma mais simples dessas combinações é a de representação de números naturais (inteiros sem sinal), que são números positivos de 0 até o limite máximo. Embora existam infinitos números positivos na matemática, no computador, a quantidade de números representáveis é finita devido ao uso de registradores de tamanho limitado.

Por exemplo, considerando um registrador de *3 bits*, teremos oito combinações de *3 bits*: *000*, *001*, *010*, *011*, *100*, *101*, *110* e *111*. Propositalmente, essas oito combinações foram escritas numa certa ordem, em que cada combinação varia em relação à anterior, a partir do primeiro *bit* à direita. Assim, se a primeira combinação é *000*, a segunda será *001*. Em seguida, como as duas possibilidades de comutação do primeiro *bit* já foram apresentadas, varia-se o segundo *bit* para *1* e repete-se a comutação do primeiro *bit*, sendo as combinações, portanto: *010* seguida de *011*. Novamente, como todas as combinações dos dois primeiros *bits* foram satisfeitas, varia-se o terceiro *bit* para *1* e repete-se as comutações dos *bits* anteriores. Assim, 4 novas combinações são obtidas.

O procedimento de ordenação anterior pode ser aplicado para números decimais. Ordenando 3 dígitos decimais, temos: *000*, *001*, *002*, *003*,..., *009*, *010*, *011*, *012*, *013*,..., *019*, *020*, *021*, *022*, *023*,..., *029*, *030*, *031*, *032*, *033*,..., *039*,..., *998* e *999*. Cada combinação varia em relação à anterior, a partir do primeiro dígito à direita. Assim, se a primeira combinação é *000*, a segunda será *001*. Como no caso decimal temos nove valores para um dígito, todas as variações do primeiro dígito terminam com a combinação *009*. A seguir, varia-se o segundo dígito para *1* e repete-se a variação do primeiro dígito de *0* a *9*, e assim por diante, até obter todas as combinações. Considerando que as combinações foram ordenadas de forma crescente, podemos concluir que cada dígito à esquerda tem mais peso que o dígito da direita. Dizemos que o dígito da direita é o menos significativo.

Assim, é possível entender por qual motivo calcula-se o valor de um número decimal a partir da soma dos múltiplos de potências de *10*. Por exemplo, dada uma combinação 456 de dígitos decimais, para se saber o valor representado, calcula-se: $456 = 4 \cdot 10^2 + 5 \cdot 10^1 + 6 \cdot 10^0 = 400 + 50 + 6$.

Ao mesmo tempo, calcula-se o valor de um número binário por meio da soma dos múltiplos de potências de 2. Por exemplo, dada a combinação *101* de dígitos binários, para se saber o valor representado, calcula-se: $101 = 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 1 \cdot 4 + 0 \cdot 2 + 1 \cdot 1 = 5$.

Portanto, as combinações *000*, *001*, *010*, *011*, *100*, *101*, *110* e *111* representam, em binário, os valores decimais: *0*, *1*, *2*, *3*, *4*, *5*, *6* e *7*, respectivamente.

Podemos concluir também que, usando *n = 3 bits*, o valor máximo representável é dado por $2^n - 1 = 2^3 - 1 = 8 - 1 = 7$.

Um número hexadecimal utiliza dígitos hexadecimais para a sua representação. Os dígitos hexadecimais são denotados: 0, 1, 2,..., 9, A, B, C, D, E e F. Portanto, os 10 primeiros dígitos hexadecimais coincidem com as notações de dígitos decimais. Seguem mais seis dígitos representados por A, B, C, D, E e F, sendo os seus valores em decimal 10, 11, 12, 13, 14 e 15, respectivamente.

Calcula-se o valor de um número hexadecimal por meio da soma dos múltiplos de potências de 16. Por exemplo, dada a combinação *01F* de dígitos hexadecimais, o valor representado é calculado por:

$$01F = 0 \cdot 16^2 + 1 \cdot 16^1 + F \cdot 16^0 = 0 \cdot 256 + 1 \cdot 16 + F \cdot 1 = 0 + 16 + 15 = 31.$$

Um número hexadecimal é de particular interesse para a computação devido à possibilidade de poder representar cada dígito hexadecimal usando *4 bits*.

4.3 Representação de números inteiros

A representação dos números vistos no item anterior é de números naturais, pois não envolve números negativos. Neste item veremos a representação de números inteiros, que envolvem números negativos e positivos. Um número inteiro necessita incluir, na própria combinação de *bits*, um *bit* que representa o sinal, o qual é situado mais à esquerda. Um número assim representado é positivo quando o *bit* de sinal é igual a 0, e negativo, caso contrário. A Figura 38 ilustra três representações de números inteiros de *3 bits*: sinal e magnitude, complemento de um e complemento de dois.

Sinal e magnitude	Complemento de um	Complemento de dois
000 = *+0*	000 = *+0*	000 = *+0*
001 = *+1*	001 = *+1*	001 = *+1*
010 = *+2*	010 = *+2*	010 = *+2*
011 = *+3*	011 = *+3*	011 = *+3*
100 = *–0*	100 = *–3*	100 = *–4*
101 = *–1*	101 = *–2*	101 = *–3*
110 = *–2*	110 = *–1*	110 = *–2*
111 = *–3*	111 = *–0*	111 = *–1*

Figura 38 Representações de números inteiros.

A primeira representação é conhecida como sinal e magnitude. Nela, o primeiro *bit* é o sinal e os *bits* restantes representam a magnitude. Por exemplo, o número 011 representa um número positivo, pois o primeiro *bit* é igual a 0. Além disso, a magnitude é dada pela parte restante, 11, cujo valor é *3*. Portanto, 011 representa o número *+3*. Analogamente, 111 representa o número *–3*. A representação sinal e magnitude tem duas possibilidades de expressar o número zero: 000 e 100, que significam *+0* e *–0*, respectivamente.

A segunda representação é conhecida como complemento de um. Nela, um número negativo é obtido a partir do respectivo número positivo, invertendo cada um de seus *bits*. Os números positivos são representados como os números em sinal e magnitude. Por exemplo, para representar o número *–3*, partimos do número positivo *+3*, cuja representação é 011. Ao inverter todos os seus *bits*, tem-se 100, que é a representação de *–3* em complemento de um. Esse tipo de representação também tem duas possibilidades de expressar o número zero: 000 e 111, significando *+0* e *–0*, respectivamente.

A terceira representação é conhecida como complemento de dois. Nessa representação, um número negativo é obtido a partir do respectivo número positivo, calculando o seu complemento de um e somando 1 ao resultado. Os números positivos são representados como anteriormente. Assim, para representar o número *–3*, partimos do número positivo *+3*, cuja representação é 011. Invertemos, em seguida, todos os seus *bits*, obtendo o complemento de um, 100, e somando 1, o resultado é 101, que é a representação do número *–3* em complemento de dois. Uma característica importante dessa representação é que o procedimento para obtenção do número positivo, a partir do negativo, é igual ao procedimento usado para a obtenção do número negativo, a partir do positivo. Outra característica dessa representação é que existe apenas um número zero. Por essa razão, existe um valor a mais representado nessa abordagem em relação às anteriores; na Figura 38 é o número *–4*. Nota-se que, aplicando o procedimento para a obtenção do número positivo correspondente a *–4*, o resultado é o próprio número *–4*.

No MIPS, os números inteiros são representados em *32 bits*, em complemento de dois. A Figura 39 mostra alguns valores dessa representação. O valor máximo representado é *+2.147.483.647*, e o menor valor é *–2.147.483.648*.

0000 0000 0000 0000 0000 0000 0000 0000 = *0*
0000 0000 0000 0000 0000 0000 0000 0001 = *+1*
...
0111 1111 1111 1111 1111 1111 1111 1111 = *+ 2.147.483.647*
1000 0000 0000 0000 0000 0000 0000 0000 = *– 2.147.483.648*
...
1111 1111 1111 1111 1111 1111 1111 1101 = *– 3*
1111 1111 1111 1111 1111 1111 1111 1110 = *– 2*
1111 1111 1111 1111 1111 1111 1111 1111 = *– 1*

Figura 39 Representação de números inteiros de *32 bits* em complemento de dois.

4.3.1 Conversão de um número de n *bits* em um número com mais de n *bits*

A conversão de um número de n *bits* em um número com mais de n *bits* é uma operação muitas vezes necessária para compatibilizar o tamanho dos operandos com as unidades de processamento. Essa conversão, para números inteiros, deve levar em consideração o sinal do número.

Um dado imediato de *16 bits* do MIPS é convertido para *32 bits* na unidade aritmética. Como a representação usada no MIPS é de complemento de dois, o procedimento para essa conversão é copiar o *bit* mais significativo (o *bit* de sinal) para os *bits* estendidos. Por exemplo, para o número de quatro *bits*, 0010, a sua extensão para *8 bits* é dada por: 0000 0010.

A operação de conversão não deve afetar o valor do número. Por exemplo, para um número negativo, como 1010, cujo valor em decimal é *–6*, a sua extensão é 1111 1010, que representa o mesmo número *–6*.

4.3.2 Adição e subtração

A adição e a subtração de números naturais podem ser feitas operando dígito a dígito, da direita para a esquerda, como aprendemos em aritmética elementar.

Por exemplo, para calcular 0111 + 0110, dispomos as duas parcelas, uma abaixo da outra, fazendo coincidir os dígitos de mesma potência de 2, conforme

Figura 40, item a). A primeira operação, na Figura 40, item b), resulta na soma $1 + 0 = 1$. A segunda operação, na Figura 40, item c), é a soma $1 + 1$. Como essa soma resulta em 2, não pode ser representada num *bit*; portanto, o resultado é igual a *0* e *vai-um* = *1*. O *vai-um* é chamado em inglês de *carry-out*, ou simplesmente *carry*. A terceira operação, na Figura 40, item d), é novamente $1 + 1$, porém, deve ser somado também o *vai-um* da operação anterior, que para a terceira operação seria o *vem-um* (*carry-in* em inglês). Portanto, o resultado da soma é igual a *3*, que não pode ser representado num *bit*. Assim, o resultado é igual a *1* e *vai-um* = *1*. A quarta operação, na Figura 40, item e), é $0 + 0$, porém, somado com o *vem-um* da operação anterior, resulta em *1*. Assim termina a adição.

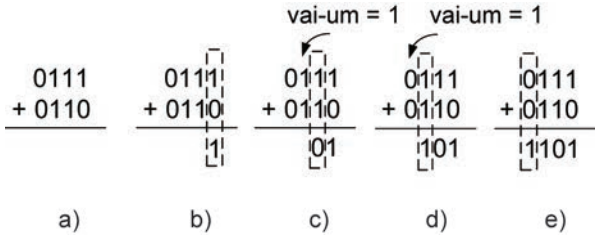


Figura 40 Adição de números naturais.

Um dos métodos para a subtração de números naturais é semelhante à adição. No entanto, no próximo item, veremos uma forma de efetuar a subtração usando a adição do negativo do subtrator ao subtraendo.

4.3.3 Subtração pela adição do negativo do subtrator

Quando se usa a representação de complemento de dois, uma subtração pode ser feita somando ao subtraendo o negativo do subtrator. Seja a subtração do número 0111, *7* em decimal, com o número 0110, *6* em decimal. Esse cálculo pode ser feito negativando o subtrator, 6, e somando ao subtraendo; portanto, calculando $7 + (-6)$. O cálculo fica simples, uma vez que a operação que deve ser feita passa a ser a adição. No exemplo, negativando o subtrator 0110, tem-se 1010. O cálculo final é 0111 + 1010, mostrado na Figura 41.

Iniciamos com a disposição dos operandos em duas linhas, como na Figura 41, item a). O item b) da Figura 41 mostra a primeira operação, $1 + 0$, que resulta em *1*. O item c) da Figura 41 mostra a segunda operação, $1 + 1$, cujo resultado é *0* e *vai-um* = *1*. O item d) da Figura 41 mostra a terceira operação, $1 + 0$, na qual o *vai-um* da operação anterior deve ser levado em consideração, somando *1*. O resultado, portanto, é *0*, e *vai-um* = *1*. O item e) da Figura 41 mostra a última operação, $0 + 1$, e o *vai-um* = *1*. O resultado é, portanto, igual a *0* e o *vai-um* = *1*. O resultado final é 0001, igual a *1*.

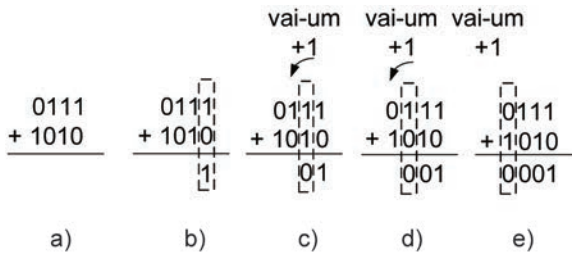


Figura 41 Subtração pela adição do negativo do subtrator ao subtraendo.

4.3.4 *Overflow*

O *overflow*, ou transbordo, ocorre quando o resultado é muito grande para um computador de tamanho de palavra finito. Um exemplo de *overflow* para a representação de números em complemento de dois, de *4 bits*, é a adição de 0111 e 0001. Realizando a operação, o resultado é dado por 1000. Como as parcelas 0111 e 0001 são números positivos, a sua soma deveria ser um número positivo. No entanto, o resultado é 1000, que é um número negativo, portanto, fica evidente o *overflow*.

Analisando as possibilidades de ocorrência de *overflow*, pode-se concluir que:

a) não ocorre *overflow* quando se soma um número positivo e um negativo, ou quando os sinais são os mesmos para a subtração.

b) ocorre *overflow* quando o valor afeta o sinal: quando somando dois positivos resulta num negativo; somando dois negativos resulta num positivo; subtraindo um negativo de um positivo e dá um negativo; ou subtraindo um positivo de um negativo e dá um positivo.

Quando um *overflow* ocorre num computador, existe o mecanismo de *interrupção*, também conhecido como de *exceção*, que faz com que o computador pare, momentaneamente, de executar o programa em andamento e passe a executar uma rotina de exceção que trata do *overflow*. Essa rotina pode, por exemplo, mostrar uma mensagem de texto na tela do computador, indicando a ocorrência de *overflow*, e encerrar a tarefa. Outra possibilidade é o computador indicar a ocorrência de *overflow*, na tela, porém, continuar a tarefa. Existem casos nos quais a detecção de *overflow* não é importante. Para esses casos foram criadas, no MIPS, instruções para ignorar o *overflow*. São as instruções *addu*, *addiu*, *subu*, *sltu*, em que o caractere “u” significa *unsigned*, ou sem sinal.

4.3.5 Projeto de uma unidade lógica e aritmética

A unidade lógica e aritmética (ULA), ALU (*Arithmetic and Logic Unit*, em inglês), é um circuito combinatório que suporta as operações para as instruções aritméticas e lógicas. Neste item vamos desenvolver o projeto de uma ULA que suporta as instruções aritméticas e lógicas vistas na unidade anterior.

Começamos com o projeto de uma ULA para suportar as instruções *add*, *and* e *or*. Primeiro desenvolvemos o projeto de operação para um *bit* e, posteriormente, replicamos esse projeto 32 vezes, para obter uma ULA de *32 bits*. A Figura 42 mostra o diagrama de um circuito para operações de um *bit*, em que *a* e *b* são os dois operandos de entrada de *1 bit* e o "resultado" é a saída de um *bit* do resultado da operação. A seleção da operação é feita pela entrada denotada "operação". Caso existam apenas duas operações selecionáveis, por exemplo, *and* e *or*, a entrada "operação" deve ser de *1 bit*. Para *operação = 0* seleciona-se uma operação, por exemplo, *and*. A entrada *operação = 1* seleciona a operação *or*.

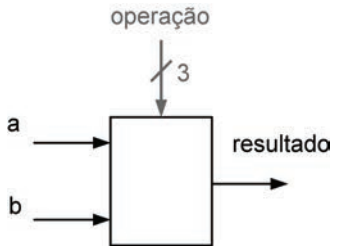


Figura 42 Esboço de uma ULA de um *bit*.

Quando existem três ou quatro operações, precisamos de *2 bits* para a entrada "operação". Assim, a entrada 00 seleciona a operação *and*, 01 seleciona a operação *or*, 10 seleciona a operação de *adição*, e 11, outra operação. Analogamente, quando existem de cinco a oito operações, precisamos de *3 bits*. A Figura 42 ilustra um diagrama para *3 bits* de operação.

Um circuito combinacional útil para selecionar uma das entradas é o multiplexador (MUX). O diagrama da Figura 43 mostra um multiplexador de entradas *a* e *b*, de *1 bit*. Quando a seleção *s = 0*, a entrada *a* é selecionada, e quando *s = 1* seleciona-se a entrada *b*. No multiplexador, a entrada selecionada é transmitida para a saída. Assim, quando a entrada *a* é selecionada, a saída *c* é igual à entrada *a*.

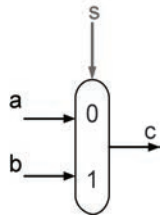


Figura 43 Multiplexador de duas entradas.

Para o circuito de operações selecionáveis de *1 bit*, suportando adição, *and* e *or*, podemos usar um multiplexador de quatro entradas de *1 bit*, conforme mostra a Figura 44. A *operação* = *00* seleciona *and*; a *operação* = *01*, *or*; e a *operação* = *10*, a adição. A *operação* = *11* não está sendo usada.

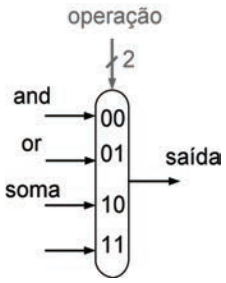


Figura 44 Multiplexador para quatro entradas.

O projeto agora deve prosseguir na realização das operações *and*, *or* e *adição*, cujas entradas são *a* e *b*. As duas primeiras operações são fáceis de serem implementadas, bastando usar diretamente os circuitos *and* e *or*, vistos na Unidade 2. A Figura 45 mostra esses dois circuitos.

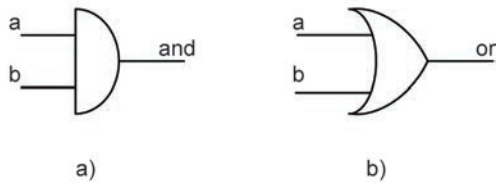


Figura 45 Circuitos a) *and* e b) *or*, de duas entradas.

O circuito de adição de um *bit*, cujo diagrama de bloco é mostrado na Figura 46, tem entradas *a* e *b*, além do *vem-um* (*carry-in*) do estágio anterior. Como saídas, tem o resultado da soma de *1 bit* e o *vai-um* (*carry-out*). Esse circuito de soma é chamado de *somador completo*.

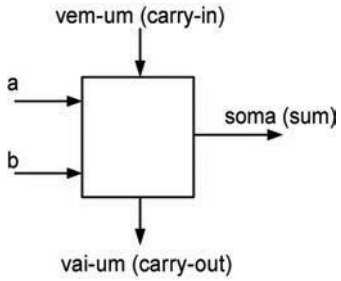


Figura 46 Diagrama de bloco do circuito de adição de *1 bit* (*somador completo*).

A saída soma é obtida pela equação:

$$soma = a \oplus b \oplus cin$$

e a saída *vai-um* (*carry-out*) é dada por:

$$cout = a\ b + a\ cin + b\ cin,$$

em que *cin* é a notação simplificada para *vem-um* (*carry-in*) e *cout* é a notação para *vai-um* (*carry-out*). A Tabela 8 mostra a tabela-verdade para essas duas funções.

Tabela 8 Tabela-verdade para *soma* e *cout*.

a	b	cin	soma	cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

A Figura 47 mostra o diagrama completo de um circuito da ULA de *1 bit*, com três operações, e a Figura 48, uma ULA de *32 bits*, constituída de 32 ULAs de *1 bit*, ULA_i, $i = 0,...,31$.

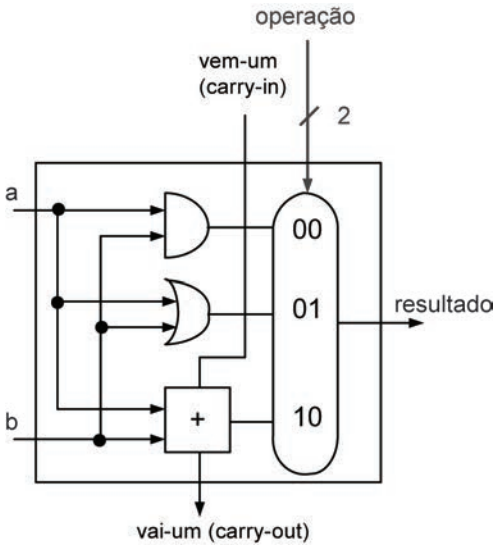


Figura 47 Circuito de ULA de *1 bit* para três operações selecionáveis.

No circuito de ULA da Figura 49, a operação de subtração é feita somando o negativo do subtrator ao subtraendo. Para tanto, inverte-se o *bit b* da ULA e acrescenta-se *1*, pela entrada de *vem-um*. A operação de subtração descrita é selecionada pela operação = 110. O *bit* mais à esquerda da seleção de "operação" corresponde ao "inverte-b", que para a subtração deve ter valor *1*. Os demais *bits* correspondem à seleção das operações *and*, *or* e *soma*. A introdução de *vem-um* = *1* deve ser feita no primeiro *bit* da ULA.

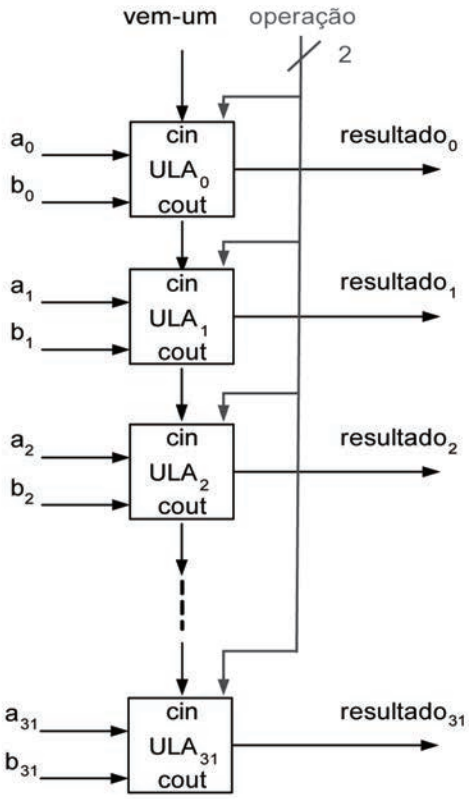


Figura 48 Circuito de uma ULA de *32 bits*.

4.3.6 ULA com operação de subtração

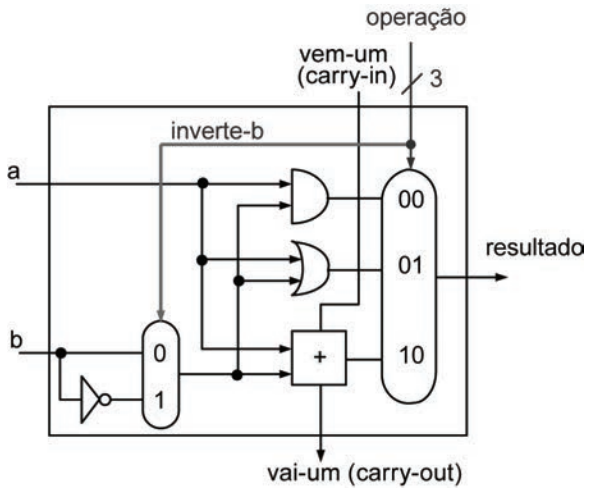


Figura 49 ULA com operação de subtração.

4.3.7 ULA com operação *set-on-less-than*

Na instrução *slt* (*set-on-less-than*), o resultado pode ter valor 0 ou 1 em *32 bits*:

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ou
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1.

Nota-se que o único *bit* que diferencia os dois resultados é o *bit* que se encontra mais a direita.

A instrução *slt* executa o comando em linguagem C:

if $\$s1 < \$s2$ *then*
  $\$t0 = 1$
else
  $\$t0 = 0$.

Portanto, ao selecionar a operação *slt*, a ULA deve realizar uma operação que resulta em 0 ou 1 de *32 bits*. Esse resultado é posteriormente carregado num registrador, no exemplo, *$t0*, porém, numa etapa posterior à ULA.

A operação a ser realizada na ULA é de comparação $(\$s1 < \$s2)$ e, para realizar essa comparação, podemos fazer a subtração $\$s1 - \$s2$. Se o resultado for negativo, $\$s1 < \$s2$, a saída da ULA deve ser igual a 1, caso contrário, a saída deve ser igual a 0.

Quando o resultado for negativo, o *bit* de sinal é igual a 1 e a saída deve ser igual a 1. Quando o resultado for positivo, ou zero, o *bit* de sinal é igual a 0 e a saída deve ser igual a 0. A solução é realizar a operação de subtração e fazer com que o *bit* de sinal seja igual ao *bit* do resultado$_0$, conforme Figura 50. Os outros *bits* de saída devem ser iguais a zero, conforme Figura 51. A Figura 52 mostra um diagrama final de *32 bits*. Como usamos a entrada 11 do multiplexador de saída e a inversão do *bit* b, a seleção da operação *slt* é igual a 111.

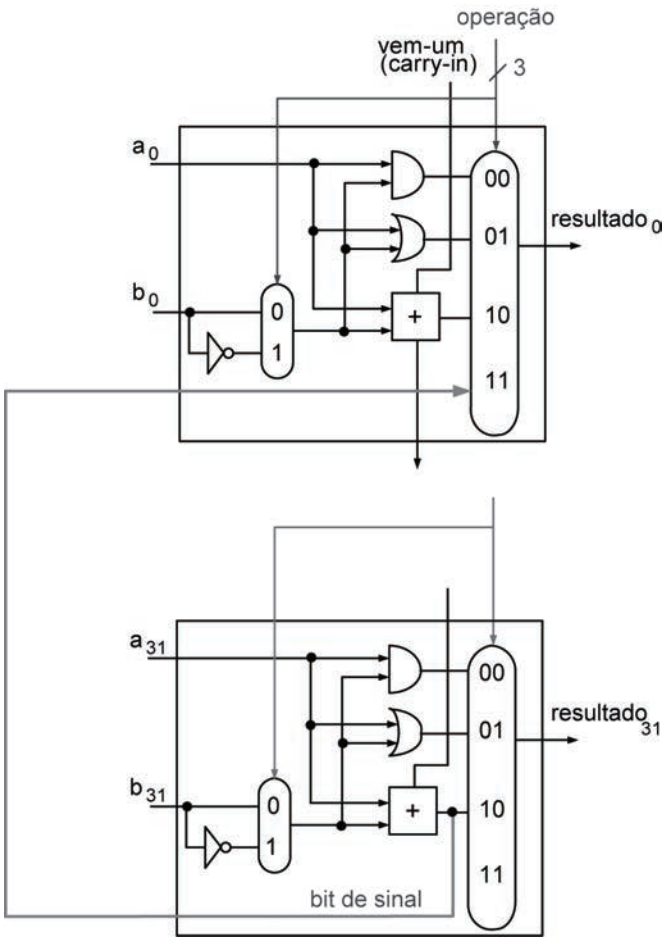


Figura 50 Obtenção do resultado do *bit* mais à direita, em função do *bit* de sinal, para *slt*.

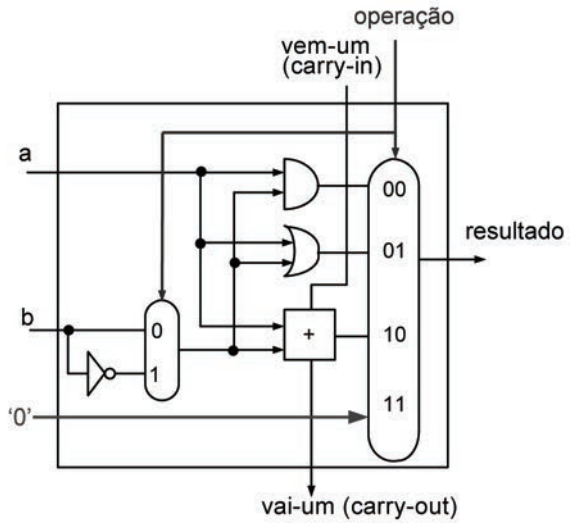


Figura 51 Obtenção dos *bits*, excluindo o *bit* mais à direita, para a operação *slt*.

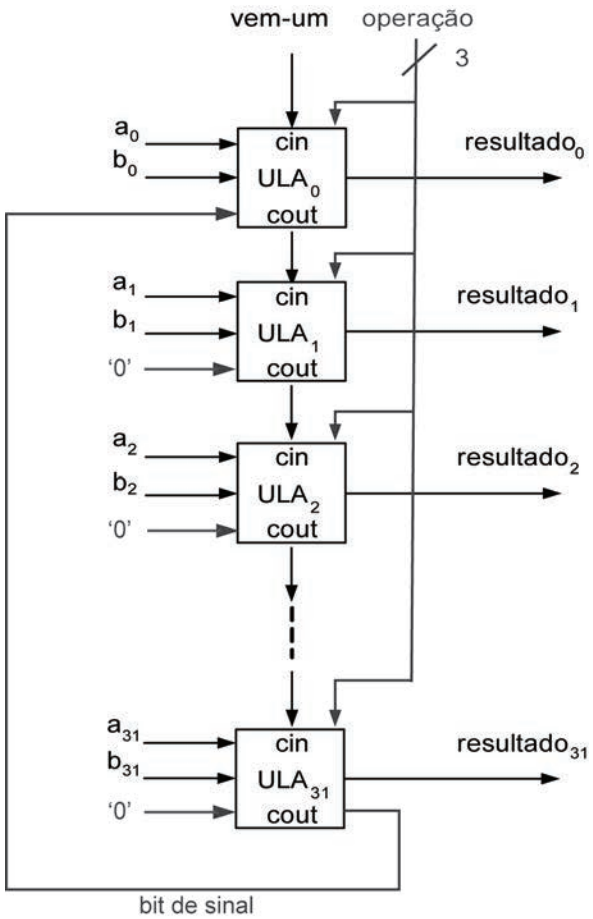


Figura 52 Diagrama de uma ULA de *32 bits* suportando operação *slt*.

4.3.8 ULA com *flag* de zero

Finalmente, para completar a ULA, vamos acrescentar o *flag* de zero. A função deste é sinalizar quando o resultado da operação aritmética é zero. É importante notar que ele indica a obtenção do zero quando o seu valor é igual a 1, ou seja, indica que o resultado é zero, com o valor 1. Uma solução para a sua implementação é um circuito *or* com todas as saídas da ULA, com inversão (*or* com inversão = *nor*), conforme Figura 53.

A Figura 54 ilustra um diagrama de bloco de uma ULA mostrando as entradas e saídas, que iremos usar nas unidades posteriores.

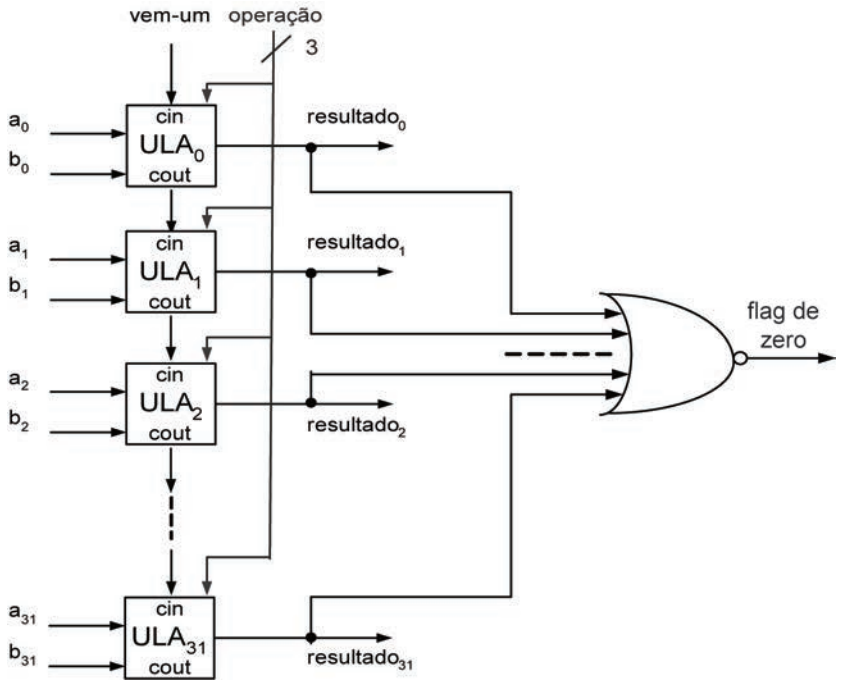


Figura 53 ULA de *32 bits* com *flag* de zero.

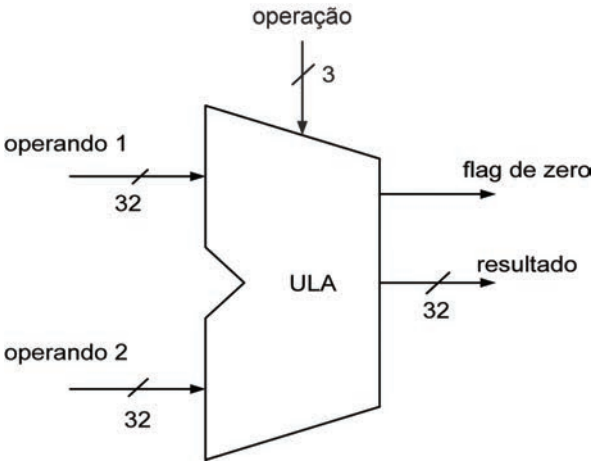


Figura 54 Diagrama de blocos de uma ULA de *32 bits* para o processador MIPS.

Concluimos o item afirmando que a ULA projetada é um circuito combinacional. Uma vez energizado, o circuito da ULA começa a trabalhar, ininterruptamente. Para se obter o resultado de uma operação aritmética, com novos operandos, precisamos alterar os operandos e selecionar a operação. O resultado é obtido depois de certo tempo, denominado tempo de atraso, ou latência. Esse tempo é função da quantidade de portas lógicas em série. No caso da operação de adição, um circuito de soma gera um *vai-um*, que é introduzido no circuito somador do *bit* seguinte. Portanto, para a obtenção do resultado da soma, 31 *vai-um* devem ser calculados e propagados até que o último *bit* possa ser calculado. Esse tipo de somador é chamado de *vai-um propagado* (*ripple carry*, em inglês). Na prática, são usados circuitos específicos para calcular antecipadamente os *vai-um*, a fim de diminuir a latência para a adição. Tais circuitos específicos são chamados de circuitos de *vai-um antecipado* (*look-ahead carry*, em inglês).

O projeto de uma unidade aritmética por completo é muito extenso. Para a obtenção de resultados rápidos, seria interessante a implementação de circuitos de multiplicação e de outras operações. No entanto, para o objetivo do presente livro, concluimos o nosso projeto da ULA, passando a descrever a representação de números reais.

4.4 Representação de números reais: ponto flutuante

O ponto flutuante é uma forma de representar números em intervalos bem maiores se comparados à representação de números inteiros, incluindo números fracionários. Essa representação é, portanto, uma forma de representar números reais, com relativa precisão nos computadores.

A ideia básica é separar os *bits* em três campos: sinal, expoente e significando. O sinal ocupa apenas *1 bit*, sendo 1 para negativo e 0 para positivo, como nos números inteiros. O campo de expoente proporciona a definição do intervalo no qual um número pode ser representado, portanto, quanto maior o número de *bits* do expoente, maior o intervalo de representação. O campo de significando é usado para determinar a mantissa, e quanto maior o número de *bits* do significando, maior a resolução**.**

O módulo de um número é calculado multiplicando a mantissa pela potência de 2, definida pelo expoente:

$$módulo = mantissa \cdot 2^{expoente}$$

e a representação completa do número é dada por:

$$(-1)^{sinal} \cdot módulo = (-1)^{sinal} \cdot mantissa \cdot 2^{expoente}.$$

4.4.1 Padronização IEEE 754 para ponto flutuante

Praticamente todos os computadores modernos usam a padronização IEEE 754 de ponto flutuante. Nesse padrão, existem duas possibilidades:

a) precisão simples: expoente de *8 bits*, significando de *23 bits*;

b) precisão dupla: expoente de *11 bits*, significando de *52 bits*.

A forma de calcular o valor de um número binário, como soma de múltiplos de potências de 2, foi vista, como segue:

$$101 = 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 1 \cdot 4 + 0 \cdot 2 + 1 \cdot 1 = 5,$$

da mesma forma que calculamos o valor de um número decimal, como soma de múltiplos de potências de 10:

$$456 = 4 \cdot 10^2 + 5 \cdot 10^1 + 6 \cdot 10^0 = 400 + 50 + 6.$$

Calculemos agora o valor de um número fracionário em decimal e verifiquemos como poderia ser feita uma analogia com um número binário fracionário. Seja, por exemplo, o valor de *0.456*:

$$0.456 = 4 \cdot 10^{-1} + 5 \cdot 10^{-2} + 6 \cdot 10^{-3}$$

ou seja, *0.456* é igual a 4 décimos, mais 5 centésimos e mais 6 milésimos.

Calculemos agora o valor da fração binária *0.111*:

$$0.111 = 1 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdot 2^{-3}.$$

Assim, temos que *0.111* é igual a um meio, mais um quarto e mais um oitavo, pois $2^{-1} = 1/2$; $2^{-2} = 1/4$ e $2^{-3} = 1/8$.

Obtenção da mantissa

A mantissa é obtida pelo campo do significando, como parte fracionária somada de *1*. Como exemplo, considera-se que o campo de significando de um número seja de *3 bits* e tenha o valor 111. O valor da mantissa é a parte fracionária somada de *1*, portanto igual a: $1 + 0.111 = 1.111$. Assim, o valor da mantissa fica

entre *1* e $2-\varepsilon$, em que $\times$ representa um valor muito pequeno. Um número assim representado, com o valor da mantissa entre *1* e $2-\varepsilon$, é chamado de número normalizado. Num número normalizado, o valor *1* que precede a parte fracionária sempre existe para ser somado à parte fracionária e não aparece na representação. Esse valor *1*, chamado de *leading bit*, é dito implícito.

Obtenção do expoente

O expoente da representação ponto flutuante do padrão IEEE 754 é polarizado, ou seja, o campo de expoente é um número positivo, sem sinal. A polarização é feita adicionando um valor (*bias*, em inglês) ao expoente, para que o mesmo se torne positivo. Para o padrão IEEE 754 precisão simples, o *bias* é igual a *127*, e para a precisão dupla, *1023*. Assim, para a obtenção do expoente, para o cálculo do número, devemos subtrair o *bias* do campo de expoente da representação. A forma de cálculo para o padrão IEEE 754 fica:

$$(-1)^{sinal} \cdot (1 + significando) \cdot 2^{expoente\text{-}bias}$$

Exemplo: Seja o número fracionário binário –0.11, que em decimal é igual a *–0,75*. O *bit* de sinal da representação ponto flutuante do número é *1*, pois o número é negativo. Consideramos agora somente o módulo 0.11 para simplificação. Como esse número é menor que *1*, não está normalizado. Para a normalização, multiplicamos o número por 2 e "dividimos por 2" , de forma que o número multiplicado por 2 passe a ser a mantissa e a divisão por 2 passe a ser considerada na parte do expoente:

$$0.11 = 1.1 \cdot 2^{-1}$$

em que *1.1* é a mantissa, e 2^{-1} é a parte considerada no expoente, ou seja, o expoente é *–1*.

Na representação precisão simples, o *bias* é *127*, portanto, o campo de expoente, para o exemplo, é dado por $-1 + 127 = 126$, que em binário é igual a *01111110*.

A mantissa é igual a *1.1*, portanto, como o *leading bit* é implícito, somente a parte fracionária aparece no campo do significando. Sendo assim, o padrão de *bits* para o número fracionário binário –0.11 é dado pelos campos:

a) sinal = 1,

b) expoente = 01111110 e

c) significando = 10000000000000000000000, conforme segue:
10111111010000000000000000000000.

4.5 Considerações finais

Nesta unidade foram vistos os fundamentos sobre a representação dos números naturais, inteiros e reais, em computadores, e a construção de uma ULA simples de *32 bits* usando portas lógicas, para ser usada no computador MIPS.

4.6 Estudos complementares

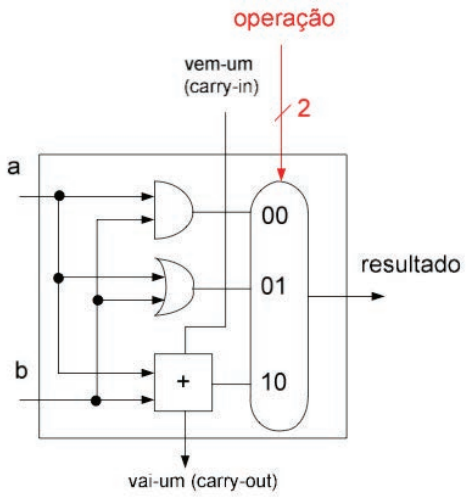
Para complementar os estudos sobre a aritmética, os leitores podem se reportar ao terceiro capítulo, *Arithmetic for Computers*, do livro de Hennessy & Patterson (2008), ou ao mesmo capítulo da versão traduzida por Daniel Vieira (HENNESSY & PATTERSON, 2005). Para quem deseja um estudo mais profundo sobre aritmética, sugere-se o livro de Deschamps, Bioul & Sutter (2006).

4.7 Exercícios

1. Dar a representação binária do número natural 63, ou em outras palavras inteiro sem sinal, numa palavra de 8 bits.
2. Dar a representação binária do número inteiro –63, numa palavra de 8 bits em:
   a) Sinal e magnitude;
   b) Complemento de um;
   c) Complemento de dois.
3. Representar o número –63 em palavra de 16 bits, para:
   a) Sinal e magnitude;
   b) Complemento de um;
   c) Complemento de dois.
4. Realizar a adição binária bit a bit dos números naturais 17 e 18, em 8 bits.
5. Realizar a adição dos números inteiros em complemento de dois de 8 bits seguintes: 11101111 e 00010010. Obter os valores decimais e conferir se o resultado está correto.
6. Fazer a adição dos números 126 e 64 em binário, e verificar se ocorre overflow em 8 bits na representação complemento de dois.

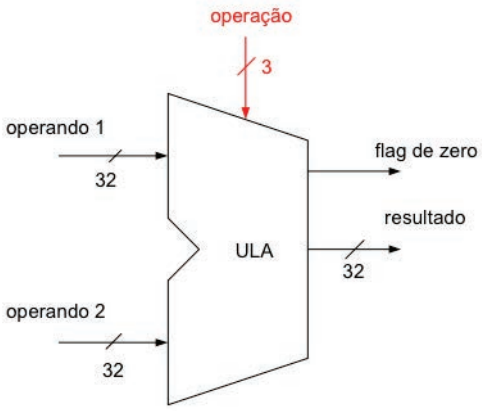
7. Fazer a adição dos números –126 e –64 em binário, e verificar se ocorre overflow em 8 bits na representação complemento de dois.

8. Dado o circuito da ULA de 1 bit abaixo, como seria possível selecionar a execução de uma operação XOR entre a e b, sem alterar o circuito? Nota-se que essa operação XOR é apenas para a ULA de 1 bit, portanto não precisa funcionar quando se encadeia 32 ULAs de 1 bit.



9. Um circuito somador completo faz a soma de dois bits a e b considerando-se o vem-um, ou carry-in, tendo como resultado a soma s e o vai-um, ou carry--out. Projetar as equações para os circuitos lógicos do somador completo, a partir da tabela-verdade, e desenhar o circuito.

10. Dado o diagrama simplificado da ULA do MIPS conforme figura abaixo, preencher a tabela de operações.



Operação	Operando 1	Operando 2	Resultado	Flag de zero
000	32	36		
001	32	36		
010	32	32		
110	32	32		
111	32	32		

11. A operação slt resulta em 1 se a < b e 0, caso contrário. Como seria possível construir uma ULA com uma operação slt modificada, em que resulte em -1 se a < b e 0, caso contrário?
12. Dada uma representação em ponto flutuante de um número no padrão IEEE 754, precisão simples, 11000000011000000000000000000000 separar os campos (sinal, expoente e significando) e calcular o valor em decimal do número representado.
13. Dada uma representação em ponto flutuante de um número no padrão IEEE 754, precisão simples, 01000000100010000000000000000000 separar os campos (sinal, expoente e significando) e calcular o valor em decimal do número representado.
14. Dada uma representação em ponto flutuante de um número no padrão IEEE 754, precisão simples, 01000000100110000000000000000000 separar os campos (sinal, expoente e significando) e calcular o valor em decimal do número representado.
15. Representar o número 3,75 no padrão IEEE 754 usando precisão simples.
16. Representar o número –7,25 no padrão IEEE 754 usando precisão simples.

UNIDADE 5

Desempenho de computadores

5.1 Primeiras palavras

A presente unidade é focada na definição de desempenho e sua aplicação na avaliação de computadores.

5.2 Problematizando o tema

Na primeira unidade deste livro, apresentamos gráficos demonstrando que o desempenho dos computadores tem melhorado ao longo do tempo. Nesta unidade, vamos fazer uma definição formal do desempenho, apresentar formas de medi-lo, ou comparar computadores quanto a seu desempenho. Podemos, assim, entender por que alguns *hardwares* são melhores que outros para programas diferentes, quais fatores de desempenho do sistema são relacionados ao *hardware*, e quais, ao sistema operacional. Outra questão é como a arquitetura do conjunto de instruções (*Instruction Set Architecture* – ISA) de uma máquina afeta o desempenho.

A Tabela 9 faz comparação de aviões comerciais (não necessariamente em atividade) quanto à capacidade de assentos para passageiros, autonomia de vôo em milhas, velocidade em milhas por hora. Já a Tabela 10 apresenta o *throughput* em passageiros x milha por hora. *Throughput* é uma medida de tarefa realizada por unidade de tempo. No exemplo, é a quantidade de passageiros transportados numa distância correspondente a uma milha, no tempo correspondente a uma hora. É possível obter essa medida multiplicando a capacidade de passageiros, coluna 2, pela velocidade, coluna 4, da Tabela 9.

Tabela 9 Capacidade, autonomia e velocidade de aviões comerciais.

Avião	Capacidade (passageiros)	Autonomia (milhas)	Velocidade (milhas/hora)
Boeing 777	375	4.630	610
Boeing 747	470	4.150	610
BAC/Sud Concorde	132	4.000	1.350
Douglas DC-8-50	146	8.720	544

Fonte: adaptada de Hennessy & Patterson (2005).

Tabela 10 *Throughput* para os aviões comerciais da Tabela 9.

Avião	*Throughput* (passageiros x milhas/hora)
Boeing 777	228.750
Boeing 747	286.700
BAC/Sud Concorde	178.200
Douglas DC-8-50	79.424

É importante notar, nos dados da Tabela 9, a complexidade em comparar os aviões e afirmar qual é o melhor. Por exemplo, se o critério de comparação é a velocidade, o Concorde é melhor. Mas se o critério é a capacidade de passageiros, o Boeing 747 ganha. E se levar em consideração a autonomia de vôo, o DC-8-50 é o vencedor.

Outra questão é a medida de quão rápido é um avião em relação a outro, ou quanto um avião é maior em capacidade de passageiros em relação a outro. Essas questões serão analisadas em relação aos computadores na presente unidade.

5.3 Desempenho de computadores

Neste item serão vistos os principais conceitos para a medida de desempenho em computadores, os conceitos de *benchmark* e a lei de Amdahl, que permite obter o ganho em *speedup*, em função de uma melhoria de desempenho numa parte do computador.

5.3.1 Medida de tempo

A medida de tempo de resposta, também chamada de latência, é muito usada em computação. As questões são variadas, por exemplo, medindo: a) o tempo que leva para executar a tarefa do usuário; b) o tempo gasto para executar uma tarefa; c) o tempo dispendido para uma consulta a um banco de dados.

Outras questões são relativas à quantidade de tarefas realizadas, denominada *throughput*, como por exemplo: a) quantas tarefas a máquina pode executar de uma vez; b) significado da taxa média de execução; c) quantidade de trabalho realizado.

As respostas às questões acima podem resolver outros itens de âmbito mais geral como o impacto: a) se atualizarmos uma máquina com um novo processador; b) se adicionarmos uma nova máquina no laboratório.

Uma medida de tempo é o tempo decorrido, que conta todas as operações feitas no computador, envolvendo acesso a memória e disco, E/S (entrada/saída), etc. Essa medida é útil, mas nem sempre boa para o propósito de comparação. Outra medida é o tempo do processador, ou tempo de CPU. Este não conta a E/S ou tempo dispendido rodando outros programas e pode ser dividido em tempo do sistema e tempo do usuário.

O nosso foco é a medição do tempo de CPU do usuário, que é o tempo dispendido executando linhas de código que estão no programa do usuário.

5.3.2 Medida de desempenho

O desempenho é definido como o inverso do tempo de execução. Assim, para certo programa que roda numa máquina X, temos:

$$desempenho_x = \frac{1}{tempo\ de\ execução_x}$$

A medida de desempenho reflete quantas tarefas, com uma determinada duração, dada pelo tempo de execução, são realizadas por unidade de tempo.

Para fazermos uma comparação de quanto uma máquina X é mais rápida que uma máquina Y, dividimos os respectivos desempenhos e obtemos *n*, que é a quantidade de vezes que a máquina X é mais rápida que a máquina Y. Se a divisão for menor que 1, a máquina X é mais lenta que a máquina Y. Portanto,

$$n = \frac{desempenho_X}{desempenho_Y}$$

Exercício proposto sobre desempenho: calcular quanto a máquina A é mais rápida em relação à máquina B, dado que:

- máquina A executa um programa em 20 segundos;
- máquina B executa o mesmo programa em 25 segundos.

5.3.3 Ciclos de *clock*

O tempo de processador é função de um conceito conhecido como ciclo de *clock*. O *clock* é uma sequência de pulsos que determina o ciclo básico da máquina. A Figura 55 mostra uma ilustração de um *clock*. Na parte superior é mostrado um sinal (onda quadrada), que representa a variação da voltagem entre o nível superior e o inferior do *clock*, e na parte inferior é mostrado o momento em que ocorre a transição de voltagem de nível inferior para superior, denotado *tick*. Um *tick* de *clock* indica quando se inicia uma atividade.

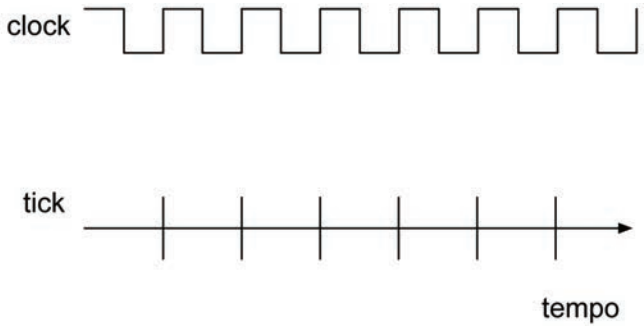


Figura 55 Ilustração do *clock* e do *tick*.

Definimos tempo de ciclo, ou período do ciclo, o tempo entre *ticks*, em *segundos*.

A taxa de *clock*, frequência de *clock*, velocidade de *clock*, ou simplesmente *clock*, é a frequência que representa o inverso do tempo de ciclo, medida na unidade *hertz* ou *ciclos por segundo*:

$$taxa\ de\ clock = \frac{1}{tempo\ de\ ciclo}.$$

Obviamente, o inverso da taxa de *clock* é o tempo de ciclo:

$$tempo\ de\ ciclo = \frac{1}{taxa\ de\ clock}.$$

Exercício envolvendo ciclos de clock: calcular o tempo de ciclo, quando a taxa de *clock* é de *200 MHz*.

Solução:

$$\frac{1}{200 \cdot 10^{6}} = 5 \cdot 10^{-9}\ segundos.$$

O tempo de processador pode ser expresso em termos de ciclo, conforme a equação:

$$\frac{segundos}{programa} = \frac{ciclos}{programa} \times \frac{segundos}{ciclo},$$

significando que o tempo de execução de um programa, em segundos (*segundos/programa*), é igual ao produto do número de ciclos do programa (*ciclos/programa*) pelo tempo de um ciclo em segundos (*segundos/ciclo*).

5.3.4 Ciclos por programa

Normalmente, as instruções do computador levam tempos diferentes, como ilustrado na Figura 56. Assim, a multiplicação leva mais tempo que a soma, as operações de ponto flutuante levam mais tempo que operações de números inteiros. Acessar a memória leva mais tempo que acessar os registradores. Outro fator importante é que se tentarmos diminuir o tempo de ciclo, muitas instruções podem necessitar de mais ciclos para a sua execução.

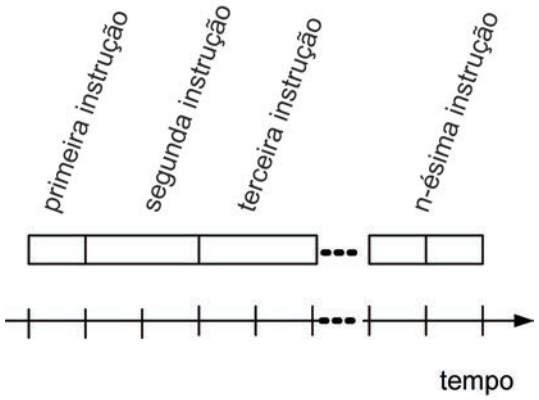


Figura 56 Diferentes instruções levam número de ciclos de *clock* diferentes.

Exercício sobre ciclos: um programa é executado em *10 s* num computador A, que tem um *clock* de *400 MHz*. Um projetista deve construir uma máquina B, que execute o mesmo programa em *6 s*. O projetista usará uma nova tecnologia para aumentar substancialmente a taxa de *clock*, que afeta o resto do projeto da CPU, obrigando a máquina B a requerer 1,2 vezes o número de ciclos de *clock* que a máquina A para o mesmo programa. Que taxa de *clock* o projetista deve usar como meta?

Solução: podemos aplicar a equação que relaciona o tempo de execução de um programa em termos de ciclos de *clock*:

$$\frac{segundos}{programa} = \frac{ciclos}{programa} \times \frac{segundos}{ciclo}$$

para ambas as máquinas, sendo segundos/ciclo o inverso da taxa de *clock*. Para o computador A, tem-se, portanto:

$$10\ s = \frac{ciclos}{programa} \cdot \frac{1}{400\ MHz}$$

e $\frac{ciclos}{programa} = 10\ s \cdot 400\ MHz$

Para o computador B, podemos substituir o tempo de execução para *6 s*:

$$6\ s = \frac{ciclos}{programa} \times \frac{1}{taxa\ de\ clock}.$$

Além disso, tem-se o número de ciclos requeridos, 1,2 vezes o número de ciclos para o computador A:

$$6\ s = 1,2 \cdot (10\ s \cdot 400\ MHz) \cdot \frac{1}{taxa\ de\ clock}.$$

Portanto, isolando-se a taxa de *clock*, obtemos o resultado desejado de *800 MHz*:

$$taxa\ de\ clock = 1,2 \cdot (10\ s \cdot 400\ MHz) \cdot \frac{1}{6\ s} = 800\ MHz\ .$$

5.3.5 Ciclos por instrução

Uma medida muito útil na análise de desempenho é o CPI (ciclos por instrução). Assim, um programa com intenso uso de instruções de ponto flutuante deve ter alto CPI. Quanto maior o CPI, menor a quantidade de instruções executadas pelo computador por unidade de tempo. Uma medida dessa quantidade de instruções por unidade de tempo, também conhecida como *throughput* de instruções, são os MIPS (milhões de instruções por segundo). Evidentemente, quanto mais simples forem as instruções, maior será o valor dos MIPS (não confundir os MIPS de *throughput* com o computador MIPS).

Exercício sobre CPI: considerar que temos duas implementações da mesma arquitetura do conjunto de instruções. Para certo programa, a máquina A tem um tempo de ciclo de *10 ns* e CPI de 2.0, e a máquina B tem um tempo de ciclo de *20 ns* e um CPI de 1.2. Qual máquina é mais rápida para esse programa, e quanto?

Solução: se as duas máquinas têm o mesmo conjunto de instruções, o número de instruções X é igual para as duas máquinas rodando esse programa. Considerando que o CPI indica o número médio de ciclos por instrução, podemos calcular o número de ciclos em função de X. Multiplicando esse número de ciclos pelo tempo de ciclo, temos o tempo de execução do programa em função de X. Para a máquina A temos:

$$tempo\ de\ programa\ A = X \cdot 2{,}0 \cdot 10\ ns = X \cdot 20\ ns,$$

e para a máquina B, temos:

$$tempo\ de\ programa\ B = X \cdot 1{,}2 \cdot 20\ ns = X \cdot 24\ ns.$$

Comparando os resultados, temos que a máquina A é mais rápida. Para sabermos o quanto, fazemos a divisão do tempo de programa B pelo tempo de programa A, e obtemos:

$$\frac{desempenho\ A}{desempenho\ B} = \frac{\frac{1}{tempo\ de\ programa\ A}}{\frac{1}{tempo\ de\ programa\ B}} = \frac{X \cdot 24\ ns}{X \cdot 20\ ns} = 1{,}2.$$

Exercício sobre número de instruções: um projetista está tentando decidir entre duas sequências de código para uma máquina particular. Numa implementação de *hardware*, existem três classes diferentes de instruções: Classe A, Classe B e Classe C, que requerem um, dois e três ciclos, respectivamente. A primeira sequência tem cinco instruções: 2 de A, 1 de B e 2 de C, e a segunda sequência tem seis instruções: 4 de A, 1 de B e 1 de C. Qual sequência será mais rápida? Quanto? Qual o CPI para cada sequência?

Solução: calculando o número de ciclos para a primeira sequência, temos:

$$2 \cdot 1 + 1 \cdot 2 + 2 \cdot 3 = 10\ ciclos,$$

e, para a segunda sequência, temos:

$$4 \cdot 1 + 1 \cdot 2 + 1 \cdot 3 = 9\ ciclos.$$

A sequência mais rápida é a que leva menos ciclos, portanto a segunda se-quência é mais rápida. Para calcularmos quanto mais rápida ela é, dividimos o número de ciclos da primeira sequência pelo número de ciclos da segunda sequência e obtemos $10/9 = 1{,}1$. O CPI de cada sequência é calculado dividindo o número de ciclos pelo número de instruções, obtendo para a primeira sequência $10/5 = 2$, e para a segunda sequência, $9/6 = 1{,}5$.

Exercício sobre MIPS: dois diferentes compiladores estão sendo testados numa máquina de *100 MHz* com três diferentes classes de instruções: Classe A,

Classe B e Classe C, que requerem um, dois e três ciclos, respectivamente. Ambos os compiladores são usados para produzir códigos para uma parte grande de *software*. O primeiro código compilado usa 5 milhões de instruções Classe A, 1 milhão de Classe B e 1 milhão de Classe C. O segundo código compilado usa 10 milhões de instruções Classe A, 1 milhão de Classe B e 1 milhão de Classe C. Qual sequência será mais rápida, levando-se em consideração o valor dos MIPS? E em relação ao tempo de execução?

Solução: calculemos o número de ciclos para os dois códigos e façamos comparações.

Primeiro código: *5 milhões* $\times$ *ciclos de A* + *1 milhão* $\times$ *ciclos de B* + *1 milhão* $\times$ *ciclos de C* = *5 milhões* $\times$ *1* + *1 milhão* $\times$ *2* + *1 mihão* $\times$ *3* = *10 milhões de ciclos*.

Segundo código: *10 milhões* $\times$ *ciclos de A* + *1 milhão* $\times$ *ciclos de B* + *1 milhão* $\times$ *ciclos de C* = *10 milhões* $\times$ *1* + *1 milhão* $\times$ *2* + *1 mihão* $\times$ *3* = *15 milhões de ciclos*.

Podemos calcular o tempo de programa usando a equação:

$$\frac{segundos}{programa} = \frac{ciclos}{programa} \times \frac{1}{taxa\ de\ ciclo}$$

e taxa de ciclo de *100 MHz*.

Para o primeiro código:

$$\frac{segundos}{programa} = 10\ milhões \cdot \frac{1}{100\ MHz} = \frac{10 \cdot 10^6}{100 \cdot 10^6} = 0{,}1\ s.$$

E para o segundo código:

$$\frac{segundos}{programa} = 15\ milhões \cdot \frac{1}{100\ MHz} = \frac{15 \cdot 10^6}{100 \cdot 10^6} = 0{,}15\ s.$$

Para calcular os MIPS dividimos os milhões de instruções pelo tempo calculado. Para o primeiro código temos:

$$MIPS = \frac{7\ milhões}{0{,}1\ s} = 70.$$

E para o segundo código:

$$MIPS = \frac{12\ milhões}{0{,}15\ s} = 80.$$

Portanto, o segundo código é mais rápido, de acordo com os MIPS, pois executa mais milhões de instruções por segundo que o primeiro código.

Mas quanto ao tempo de execução, o primeiro código é mais rápido, levando *0,1 s*, enquanto o segundo código leva *0,15 s*.

5.3.6 *Benchmarks*

Para a determinação de desempenho, executando aplicações reais, é comum usar programas de teste, denominados *benchmarks*, típicos de carga de trabalho (*workload*), ou programas típicos de classes de aplicações desejadas, como: compiladores/editores, aplicações científicas, gráficos, etc. Os pequenos *benchmarks* são bons para arquitetos e projetistas de computadores, é de fácil padronização, porém, com o uso incorreto, resultados errôneos podem ser obtidos.

Um *benchmark* muito usado é o SPEC (*System Performance Evaluation Cooperative*). O primeiro turno do SPEC ocorreu em 1989, contendo 10 programas produzindo um único número "SPECmarks". O segundo turno foi lançado em 1992, contendo programas SPECInt92 (6 programas em inteiros) e SPECfp92 (14 programas em ponto flutuante). O terceiro turno foi lançado em 1995, consistindo em um novo conjunto de programas: SPECint95 (8 programas em inteiros) e SPECfp95 (10 programas em ponto flutuante). Os quarto e quinto turnos foram lançados em 2000 e 2006, respectivamente, atualizando o conjunto de programas anteriores.

O CINT2006[1] serve para medir e comparar o desempenho de processamento de inteiros [CINT, 2006], e o CFP2006,[2] para medir e comparar o desempenho de processamento de ponto flutuante [CFP, 2006].

5.3.7 Lei de Amdahl

A lei de Amdahl define o tempo de execução novo, quando se faz um aperfeiçoamento para melhorar o desempenho de uma máquina, em função do tempo de execução antigo que é dividido em tempo relativo à parte não afetada pelo aperfeiçoamento e à parte afetada pelo aperfeiçoamento.

O tempo de execução antigo da parte não afetada continua contribuindo da mesma forma no tempo de execução novo. O tempo de execução antigo da parte afetada é dividido pela razão de melhoramento, que corresponde a quanto o aperfeiçoamento melhorou o desempenho:

1 Mais informações sobre o CINT2006 estão disponíveis em: <http://www.spec.org/cpu2006/CINT2006/>.

2 Mais informações sobre o CFP2006 estão disponíveis em: <http://www.spec.org/cpu2006/CFP2006/>.

$$Tempo\ de\ Execução_{novo} = Tempo\ de\ Execução_{parte\ não\ afetada} + \frac{Tempo\ de\ Execução_{parte\ afetada}}{Razão\ de\ Melhoramento}$$

Exercício sobre a lei de Amdahl: suponha que um programa é executado em *100 s* numa máquina, com instruções de multiplicação responsáveis *por 80 s* desse tempo. Quanto devemos melhorar a velocidade das instruções de multiplicação se desejamos que o programa seja executado 4 vezes mais rápido? E para executá-lo 5 vezes mais rápido?

Solução: o tempo de execução da parte afetada é de *80 s*, portanto, o tempo de execução da parte não afetada é *20 s*, que é a diferença entre o tempo de *100 s* e o tempo da parte afetada. A velocidade de execução nova deve ser 4 vezes mais rápida ou seja, o tempo de execução novo deve ser *100 s* dividido por 4, que é igual a *25 s*. Portanto, substituindo esses dados na equação, temos:

$$25\ s = 20\ s + \frac{80\ s}{razão\ de\ melhoramento}.$$

Isolando a razão de melhoramento, temos o resultado desejado:

$$razão\ de\ melhoramento = \frac{80\ s}{25\ s - 20\ s} = \frac{80\ s}{5\ s} = 16.$$

E para executar o programa 5 vezes mais rápido? Para calcular o valor do novo tempo, dividimos *100 s* por 5, obtendo o resultado *20 s*. Substituindo esse valor na equação da razão de melhoramento temos:

$$razão\ de\ melhoramento = \frac{80\ s}{20\ s - 20\ s} = \frac{80\ s}{0\ s} = \infty,$$

o que implica num valor infinito. Esse resultado mostra que existe um limite para a redução do tempo, acima do qual chegamos a uma razão de melhoramento impossível.

Exercícios Complementares

1) Suponhamos que uma máquina foi melhorada fazendo todas as instruções de ponto flutuante serem executadas 5 vezes mais rápido. Se o tempo de execução de certo *benchmark* antes do melhoramento é de *10 s*, qual seria o *speedup* se metade dos *10 s* é dispendida em instruções de ponto flutuante?

2) Estamos procurando um *benchmark* para testar a nova unidade de ponto flutuante acima, e queremos que o *benchmark* mostre um *speedup* de 3. Um *benchmark* é executado em *100 s*, com o antigo *hardware* de ponto flutuante. A quanto do tempo de execução as instruções de ponto flutuante devem corresponder nesse programa para que possamos produzir o *speedup* desejado nesse *benchmark*?

5.4 Considerações finais

Nesta unidade foram apresentados os principais conceitos para a avaliação do desempenho de computadores. Foram vistos os conceitos de tempo de execução, o *throughput*, quantidade de ciclos por instrução, *benchmarks* e a lei de Amdahl.

5.5 Estudos complementares

Para complementar os estudos sobre o desempenho de computadores, os leitores podem se reportar ao primeiro capítulo, *Computer Abstractions and Technology*, do livro de Hennessy & Patterson (2008), ou ao quarto capítulo, *Avaliando e Compreendendo o Desempenho*, da versão traduzida por Daniel Vieira (HENNESSY & PATTERSON, 2005).

5.6 Exercícios

1. Um computador C1 executa um programa A em 10 segundos. Um computador C2 executa o mesmo programa A em 15 segundos. Quais são os desempenhos dos dois computadores para a execução do programa A?
2. Se o computador C1 executa o programa A do exercício anterior em 20 milhões de ciclos, qual é a taxa de clock do processador?
3. Dado um computador com taxa de clock de 500 MHz, e um programa com 6 milhões de instruções classe A de 1 ciclo, 3 milhões de instruções classe B de 2 ciclos, e 2 milhões de instruções classe C de 3 ciclos, calcular:
   a) Tempo de ciclo do processador;
   b) Número de ciclos do programa;
   c) Tempo de execução do programa;
   d) CPI médio do programa;
   e) Throughput em MIPS para esse programa.

4. Considerando-se a execução do programa abaixo com n = $10x10^6$, se as instruções tipo-R levam 4 ciclos; lw, 5 ciclos; sw, 4 ciclos; bne, 3 ciclos e de operandos imediatos 4 ciclos, calcular o desempenho para um computador MIPS de 1GHz.

```
        sub   $2,   $2,   $2      #   i = 0
loop1 : muli  $14,  $2,   4       #   $14 = i × 4
        add   $3,   $4,   $14     #   $3 = endereço de A[i]
        sw    $0,   0($3)         #   A[i] = 0
        addi  $2,   $2,   1       #   i = i + 1
        slt   $7,   $2,   $6      #   $7 = (i < n)
        bne   $7,   $0,   loop1   #   se (i < n) vai para loop1
```

5. Supor que a execução sequencial de um programa num computador, leva um tempo de 100 segundos. Considerando-se que todas as instruções tem um mesmo número de ciclos, e que seja possível melhorar o tempo de execução de uma parte desse programa correspondente a 40 segundos do tempo de execução sequencial, fazendo o computador executar simultaneamente duas instruções, calcular:
   a) O novo tempo de execução;
   b) O *speedup* obtido.

6. Suponhamos que melhoramos uma máquina fazendo todas as instruções de ponto flutuante serem executadas 5 vezes mais rápido. Se o tempo de execução de certo *benchmark* antes do melhoramento é de 10 segundos, qual seria o *speedup* se metade dos 10 segundos é dispendida em instruções de ponto flutuante?

7. Estamos procurando um *benchmark* para testar a nova unidade de ponto-flutuante do exercício anterior, e queremos que o *benchmark* todo mostre um *speedup* de 3. Um *benchmark* é executado em 100 segundos, com o antigo hardware de ponto flutuante. Qual a porcentagem das instruções de ponto flutuante nesse programa para que possamos produzir o *speedup* desejado nesse *benchmark*?

UNIDADE 6

Organização do computador MIPS

6.1 Primeiras palavras

A presente unidade descreve os componentes e a organização do computador MIPS simplificado. Mostra os elementos funcionais, os caminhos de fluxo de dados e o fluxo de controle para as instruções típicas do computador MIPS, em monociclo e multiciclo.

6.2 Problematizando o tema

Nesta unidade serão feitos estudos de implementação do computador MIPS. Serão vistas as implementações das instruções aritméticas e lógicas: *add*, *sub*, *and*, *or* e *slt*; instruções de referência à memória: *lw* e *sw*; e instruções de fluxo de controle: *beq* e *j*.

Vimos, na Unidade 3, que podemos delinear o ciclo de uma instrução aritmética (formato-R) nas seguintes etapas:

1. ler a instrução usando o conteúdo do contador de programa (*program counter*, PC);
2. ler os operandos no banco de registradores;
3. executar a operação aritmética;
4. escrever o resultado no registrador destino;
5. atualizar o conteúdo do contador de programa para a próxima instrução.

As etapas 1 e 2, bem como a etapa 5, são iguais para as outras instruções. Assim, para a instrução *lw* (*load-word*), têm-se as seguintes etapas:

1. ler a instrução usando o conteúdo do contador de programa;
2. ler os operandos no banco de registradores;
3. calcular o endereço de memória, somando o valor do registrador base ao deslocamento;
4. ler a memória usando o endereço calculado;
5. escrever o dado lido na memória no registrador destino;
6. atualizar o conteúdo do contador de programa para a próxima instrução.

Para a instrução *sw* (*store-word*), o ciclo de instrução difere nas etapas 4 e 5 em relação à instrução *load-word*, já que passam a ser uma única etapa: escrever na memória o conteúdo do registrador de origem, usando o endereço calculado.

Um ciclo de instrução, para uma instrução de desvio condicional do tipo *beq*, é dado por:

1. ler a instrução usando o conteúdo do contador de programa;
2. ler os operandos no banco de registradores;
3. calcular a condição de desvio e o endereço de desvio;
4. se a condição for satisfeita, atualizar o conteúdo do contador de programa com o endereço de desvio calculado, se não, atualizar com o endereço sequencial.

Finalmente, o ciclo para a instrução de desvio incondicional *j* é dado por:

1. ler a instrução usando o conteúdo do contador de programa;
2. atualizar o conteúdo do contador de programa com o endereço de desvio.

Nota-se também que, num computador típico, essas etapas envolvem vários componentes como processador e memória, que na Figura 57 estão distribuídos em extremidades opostas da placa-mãe de um computador pessoal típico. Assim, as descrições nos itens seguintes podem ser interpretadas como descrições lógicas, e o tamanho dos elementos funcionais desenhados pode não corresponder ao tamanho físico destes. O mesmo acontece com a disposição física dos componentes.

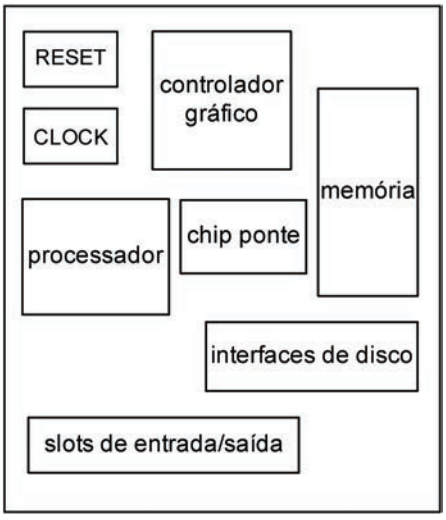


Figura 57 Placa-mãe de um computador pessoal típico.

6.3 Diagrama simplificado

Neste item será apresentado o diagrama da implementação simplificada do MIPS. Compreendê-la facilita o estudo dos itens seguintes que, basicamente, descrevem seus detalhes.

6.3.1 Fluxo de dados simplificado

A forma de interconexão dos principais elementos funcionais do computador pode ser vista na Figura 58. Os componentes mostrados estão dispostos da esquerda para a direita, na ordem de ocorrência das etapas de um ciclo de instrução, exceto o banco de registradores, que é usado na etapa de leitura dos operandos e na escrita do resultado, e o contador de programa, usado na leitura da instrução e na atualização com o endereço da próxima instrução.

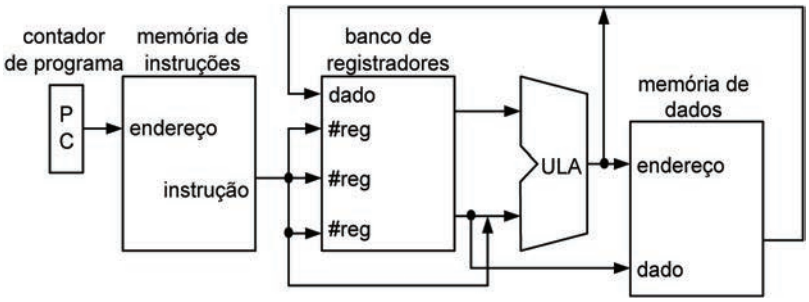


Figura 58 Fluxo de dados simplificado, com os principais elementos funcionais do computador MIPS.

Nota-se o uso da memória de instruções e da memória de dados independentes. Historicamente, uma arquitetura de computador com memória de instruções e de dados independentes é conhecida como arquitetura Harvard. Na descrição apresentada nesta unidade, essas duas memórias são usadas independentemente, para simplificação, e não para exemplificar uma arquitetura Harvard. O usuário pode pensar numa memória que fisicamente possa ser implementada em um único bloco, porém, com possibilidade de separar logicamente as duas funções a ela atribuídas: a leitura de instruções e a leitura e escrita de dados. Essa maneira de pensar remete ao modelo de von Neumann, que prevê uma única memória para instruções e dados.

As duas primeiras etapas de um ciclo de instrução são iguais para todas as instruções e ocorrem na metade esquerda do diagrama da Figura 58. A etapa 1, ler a instrução usando o conteúdo do contador de programa, envolve o contador de programa (PC) e a memória de instruções. A etapa 2, ler os operandos no banco de registradores, envolve o banco de registradores.

As etapas seguintes dependem das instruções. Assim, no caso de uma instrução aritmética (formato-R) ocorrem as seguintes etapas: 3) executar a operação aritmética; 4) escrever o resultado no registrador destino; 5) atualizar o conteúdo do contador de programa para a próxima instrução. Nesse caso, são envolvidos a ULA para executar a operação aritmética na etapa 3; o banco de registradores para a escrita do resultado no registrador destino; e o contador de programa para receber o endereço da próxima instrução. Nota-se que a memória de dados não é envolvida numa instrução aritmética. O resultado do cálculo na saída da ULA é enviado diretamente para a entrada de dados do banco de registradores.

No caso de uma instrução de referência à memória, após as duas primeiras etapas, segue outra etapa 3, calcular o endereço de memória, somando o valor do registrador base ao deslocamento. Essa etapa usa, como operandos da ULA, valores provenientes do banco de registradores e da instrução, que são mostrados no diagrama da Figura 58 pela linha abaixo do banco de registradores e que se associa à entrada inferior da ULA.

Para a instrução *lw* (*load-word*) é feita a leitura da memória de dados e a escrita no registrador destino, como segue: 4) ler a memória usando o endereço calculado; 5) escrever o dado lido na memória no registrador destino. A leitura da memória de dados é feita apontando, diretamente na memória, o endereço calculado na etapa 3, e após um tempo suficiente para a leitura do dado, este é escrito no registrador destino, no banco de registradores. A etapa seguinte é a atualização do contador de programa para a próxima instrução, igual à instrução aritmética.

Para a instrução *sw* (*store-word*), o ciclo de instrução difere nas etapas 4 e 5, que passam a ser uma única etapa: escrever na memória o conteúdo do registrador de origem, usando o endereço calculado. Para essa escrita, o endereço de memória é igual à instrução *lw*, usando o endereço calculado na etapa 3. O dado a ser escrito é proveniente do banco de registradores, obtido a partir da saída inferior do diagrama da Figura 58, e é enviado à entrada de dado da memória. A etapa de atualização do contador de programa é igual às instruções anteriores.

Para as *instruções de desvio*, o diagrama simplificado da Figura 58 não fornece informações suficientes, portanto, postergamos a descrição dessas instruções.

Os itens seguintes mostram o detalhamento do diagrama da Figura 58, descrevendo cada uma das unidades funcionais do computador MIPS.

6.4 Componentes do fluxo de dados do MIPS

Chamamos de componentes do fluxo de dados todos os componentes por onde os dados fluem durante a execução das instruções num computador. Na

descrição a seguir, são apresentados os principais componentes do fluxo de dados do MIPS.

Existem dois tipos de unidades funcionais: o primeiro tipo refere-se àquelas unidades que dependem somente dos dados de entrada e funcionam como circuitos combinacionais, não necessitam de sinais de controle para acionamento, apenas de dados de entrada, ou entradas de seleção de operação ou multiplexadores. São exemplos a memória de instruções, a leitura do banco de registradores e ULA. Assim, essas unidades mudam as suas saídas em função de alguma entrada de dados ou seleção alterada.

O outro tipo de unidades funcionais se refere àquelas que contêm estado interno e são conhecidas como circuitos sequenciais. São exemplos o contador de programa, a escrita no banco de registradores e a memória de dados. Essas unidades necessitam de sinais de controle para acionamento, além dos dados de entrada e entradas de seleção de operações ou seleção de multiplexadores. Sem o acionamento, usando sinais de controle, essas unidades continuam sempre no mesmo estado, com a mesma saída.

A seguir vamos descrever com detalhes as unidades funcionais que deverão compor uma implementação do computador MIPS.

6.4.1 Memória

A memória de um computador é atualmente um sistema complexo, envolvendo componentes rápidos como a memória *cache*, implementada em circuitos de memória estática SRAM (*Static Random Access Memory*, em inglês), componentes mais lentos como a memória DRAM (*Dynamic Random Access Memory*, em inglês) e memória em disco magnético. O estudo detalhado do sistema de memória será visto na Unidade 8.

Para possibilitar o estudo inicial da organização do MIPS, vamos considerar dois módulos de memória, a memória de instruções e a memória de dados. Conforme descrição anterior, na arquitetura von Neumann, esses dois módulos fazem parte de um único sistema.

Memória de instruções

Para o MIPS, essa memória contém uma entrada de endereço de *32 bits* e uma saída de instrução também de *32 bits*, conforme Figura 59. Considerando que essa memória já contenha as instruções, para facilitar o estudo, o funcionamento desse módulo é do tipo circuito combinacional. Assim, a memória de instruções está sempre com um endereço na entrada e uma instrução na saída. Quando é fornecido um novo endereço na entrada, após um tempo de acesso, a instrução correspondente é disponível na saída. Não é usado nenhum sinal de controle.

Figura 59 Memória de instruções.

Memória de dados

Como o computador deve ler e escrever dados na memória usando instruções *lw* (*load-word*) e *sw* (*store-word*), a memória de dados, representada pela Figura 60, é uma unidade que permite a escrita e a leitura de dados.

Diferente da memória de instruções, a memória de dados necessita de um sinal de controle para acionar uma das operações: para a escrita, deve acionar o sinal *escrever memória* (*EscrMem*, simplificadamente); e para a leitura, o sinal *ler memória* (*LerMem*). Para operação de escrita, deve ser fornecido o endereço da posição ou palavra de memória a ser escrito, e o dado a escrever (*DadoEscr*), seguido do sinal de escrita (*EscrMem*). Após um tempo de escrita, o dado está escrito na memória, na posição desejada. Para a operação de leitura, deve ser fornecido o endereço da posição, da palavra de memória a ser lida, seguido do sinal de leitura (*LerMem*). Após um tempo de acesso, o dado será disponível na saída.

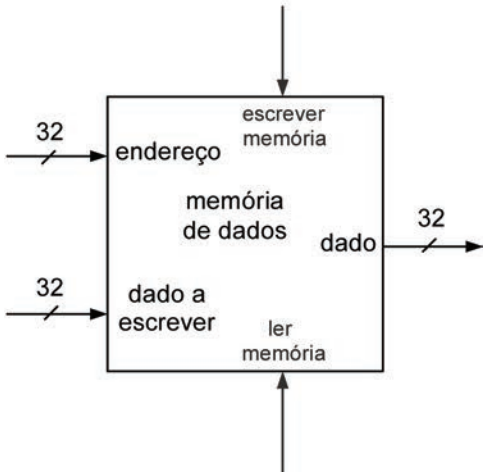


Figura 60 Memória de dados.

6.4.2 Multiplexadores

A Figura 61 mostra um multiplexador de duas entradas, a Figura 62 a) traz um multiplexador de 4 entradas, e a Figura 62 b), uma representação alternativa de um multiplexador de 4 entradas, com as entradas rotuladas em números decimais. A função dos multiplexadores é de selecionar uma das entradas para a saída, baseado numa entrada de seleção, *S*. Essa entrada está sempre funcionando, desde que a energia esteja ligada. Para o caso do multiplexador, ou mux, de duas entradas, da Figura 61, quando *S = 0*, a saída é igual à entrada *a* e, caso contrário, igual à entrada *b*. Assim, quando desejamos que uma determinada entrada seja selecionada, atribuimos um valor a *S* convenientemente. Porém, quando não estamos interessados no uso da saída de um multiplexador, o valor de *S* não importa, e nesse caso dizemos que *S* é irrelevante, simbolizando o valor irrelevante como x, portanto, *S = x*.

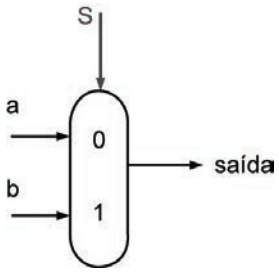


Figura 61 Multiplexador de duas entradas.

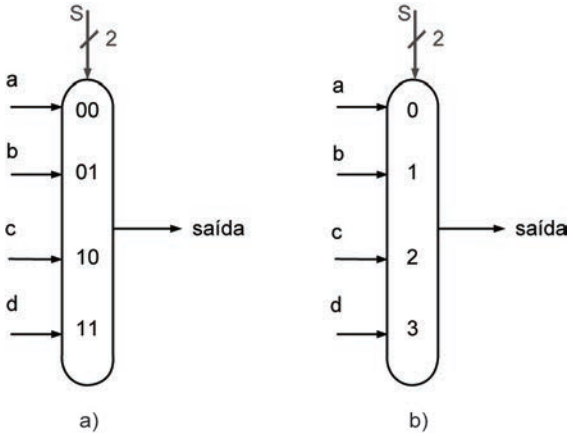


Figura 62 a) multiplexador de 4 entradas; b) representação alternativa do multiplexador de 4 entradas.

6.4.3 Banco de registradores

No MIPS, o banco de registradores contém 32 registradores de *32 bits*. Existem duas operações sobre os registradores: leitura e escrita. A Figura 63 ilustra o diagrama do banco de registradores.

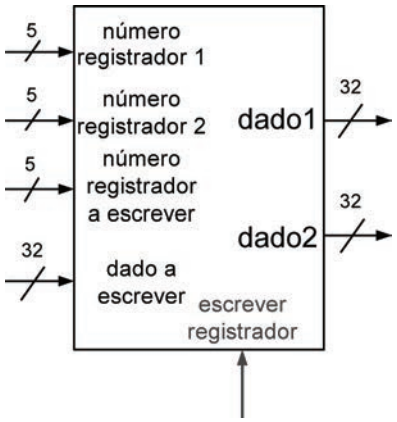


Figura 63 Diagrama de bloco do banco de registradores.

Leitura de registradores

A operação de leitura é feita simultaneamente para dois registradores. O circuito da Figura 64 mostra o esquema usado para essa operação, e a Figura 65, uma representação simplificada que será usada nos itens seguintes.

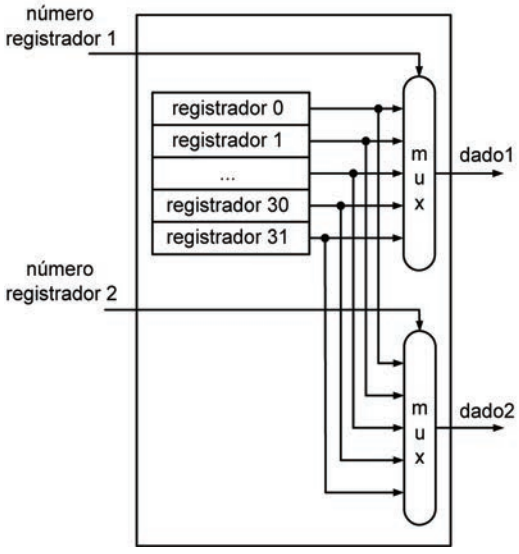


Figura 64 Circuito de leitura de registradores.

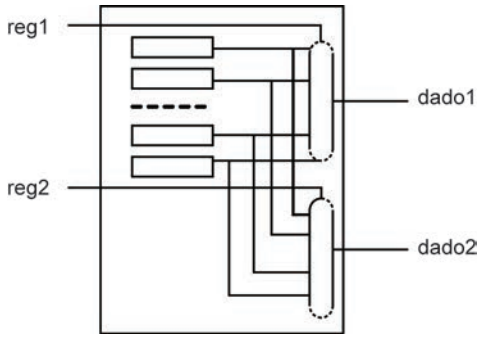


Figura 65 Diagrama simplificado de leitura de registradores.

Basicamente, são usados dois multiplexadores de 32 entradas, uma entrada para cada registrador. A seleção do multiplexador permite a saída dos valores de um registrador. Como o multiplexador é um circuito combinacional, está sempre em atividade, selecionando dois registradores para as saídas. Para realizar uma nova operação de leitura, devem ser fornecidos os números dos registradores (*reg1* e *reg2*). Após um tempo de seleção de multiplexadores, os dados (*dado1* e *dado2*) são disponíveis. O número de um registrador é constituído de *5 bits*, pois a partir deles é possível obter 32 combinações. A operação de leitura dos registradores é, portanto, do tipo combinacional e não necessita de sinal de controle de acionamento.

Escrita de registrador

A operação de escrita é feita apenas para um registrador. O circuito da Figura 66 mostra o esquema usado para essa operação, e a Figura 67 mostra uma representação simplificada que será usada nos itens seguintes.

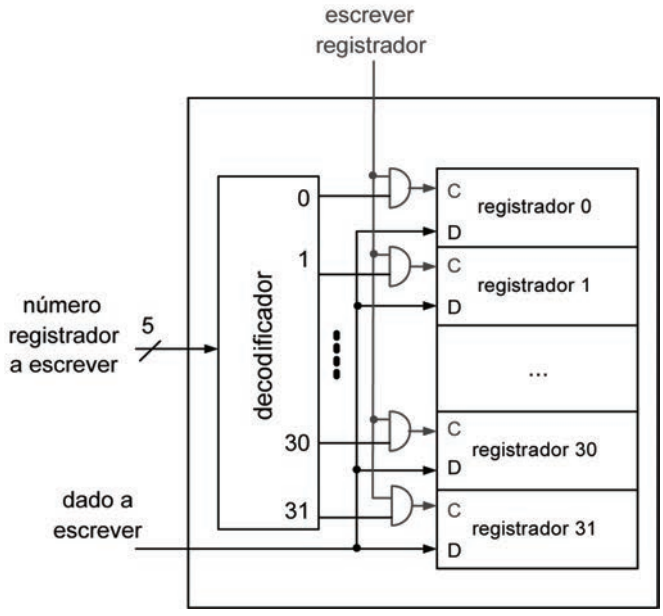


Figura 66 Circuito de escrita do registrador.

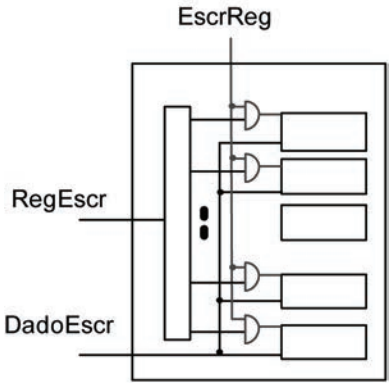


Figura 67 Diagrama simplificado para escrita de registrador.

Para a operação de escrita precisamos fornecer o dado a ser escrito e selecionar o registrador a ser escrito. Essa seleção é feita pelo circuito decodificador, que tem a função de decodificar o número do registrador de escrita (*RegEscr*) e apontar para um dos 32 registradores. O dado a ser escrito (*DadoEscr*) é conectado à entrada de todos os registradores, porém, apenas o registrador selecionado receberá o novo dado. Para finalizar a operação de escrita é preciso acionar o sinal de controle de escrita, *escrever registrador* (*EscrReg*). Como a escrita no registrador envolve alteração do conteúdo de um registrador, a operação é do tipo sequencial.

6.4.4 Circuito da ULA e a lógica de seleção de operação

O circuito da ULA, Figura 68, foi descrito na Unidade 4. A ULA é um circuito combinacional com seleção de operação, e como tal, está sempre funcionando, fornecendo resultados. Para que a ULA execute uma nova operação, é necessário fornecer duas entradas de dados (*operando1* e *operando2*), de *32 bits*, e a seleção de operação, de *3 bits*. Após o tempo de execução, a ULA fornece na sua saída o resultado de *32 bits* e o *flag* de zero. Lembramos que o *flag* de zero é igual a 1 quando o resultado é igual a zero.

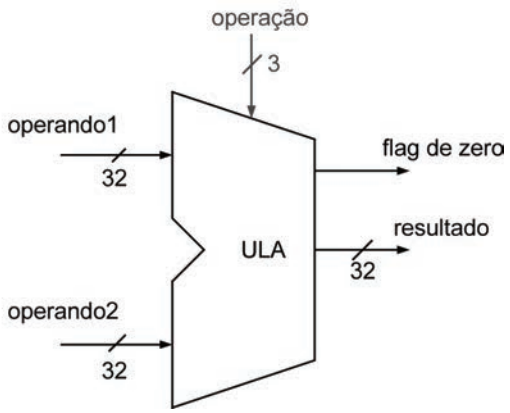


Figura 68 Diagrama da ULA.

No MIPS, os sinais de seleção de operação da ULA são obtidos usando um circuito chamado *lógica de seleção de operação*, que combina os sinais obtidos da decodificação do código de operação da instrução (*op*), *bits 26* a *31* da instrução ([31-26]), com os *bits 0* a *5* da instrução ([5-0]). Isso porque uma instrução aritmética, de formato R, usa os *bits 0* a *5* da instrução para selecionar a operação, conforme o formato de instrução e a função da lógica de seleção de operação ilustrados na Figura 69.

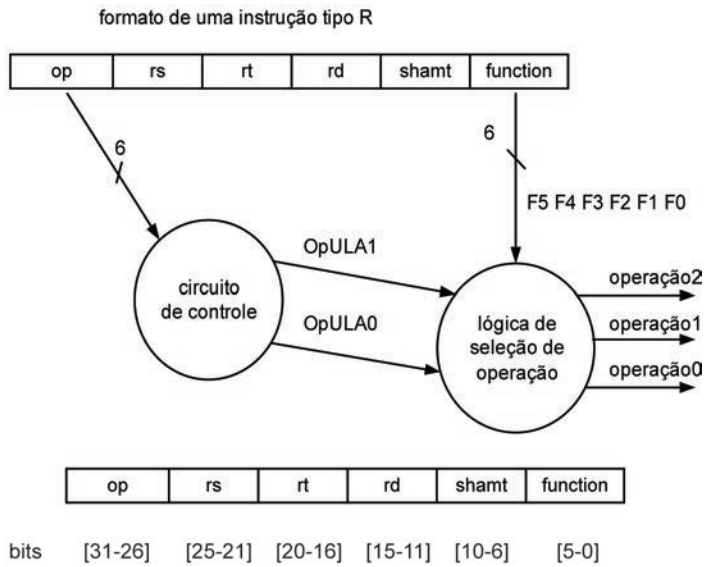


Figura 69 Formato de uma instrução aritmética e a função da lógica de seleção de operação.

Lógica de seleção de operação da ULA

A Figura 70 ilustra o circuito de lógica de seleção de operação que, basicamente, combina os dois *bits* de seleção (*OpULA1* e *OpULA0*) provenientes do circuito de controle que decodifica o *opcode* (*bits* [31-26]) da instrução, com os *6 bits* [5-0] da instrução denotados F5,..., F0.

Nota-se que as entradas F5,..., F0 só terão efeito sobre as saídas (*operação2, operação1 e operação0*) quando *OpULA1 = 1*. Para as instruções abordadas neste livro, os *bits* F4 e F5 não são usados. A Tabela 11 (tabela-verdade) ilustra o funcionamento da lógica de seleção de operação. Nas duas primeiras combinações de *OpULA1* e *OpULA0*, em que *OpULA1 = 0*, as entradas do campo de função não têm efeito sobre as saídas, portanto, os valores de F3, F2, F1 e F0 são marcados por *x* (irrelevante).

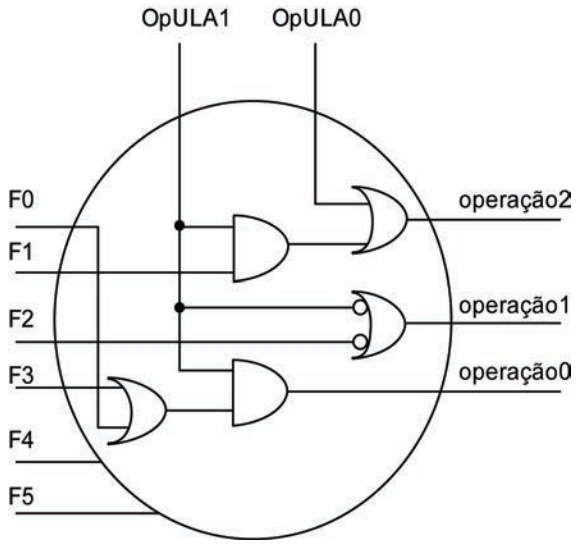


Figura 70 Lógica de seleção de operação da ULA.

Tabela 11 Tabela-verdade da lógica de seleção de operação da ULA.

OpULA		Campo de função				Operação	
OpULA1	OpULA0	F3	F2	F1	F0		
0	0	X	X	X	X	010	adição
0	1	X	X	X	X	110	subtração
1	0	0	0	0	0	010	adição
1	0	0	0	1	0	110	subtração
1	0	0	1	0	0	000	and
1	0	0	1	0	1	001	or
1	0	1	0	1	0	111	slt

6.4.5 Somadores de *32 bits*

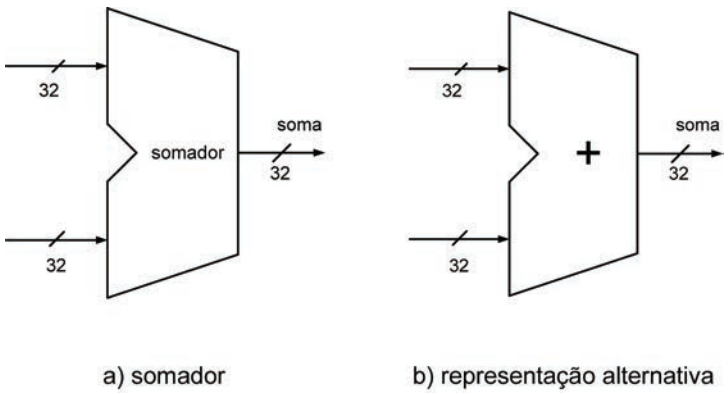


Figura 71 Somador de *32 bits*.

Os somadores de *32 bits* são circuitos combinacionais que somam dois operandos de *32 bits*, resultando num valor de *32 bits*, conforme diagrama da Figura 71.

6.4.6 Extensão de sinal e deslocamento à esquerda

Extensão de sinal: o circuito de extensão de sinal usado no MIPS, Figura 72, é outro circuito combinacional. Serve para converter um número de *n bits* em números com mais de *n bits*. No caso do MIPS, um dado imediato de *16 bits* é convertido para *32 bits* em aritmética, ou um campo de *16 bits* de endereço de memória é convertido para *32 bits* de deslocamento (*offset*) de endereço. O circuito copia o *bit* mais significativo (*bit* de sinal) para os *bits* de extensão.

Figura 72 Circuito de extensão de sinal.

Circuito de deslocamento de 2 bits à esquerda: o circuito de deslocamento de *2 bits* à esquerda permite a multiplicação de um número por 4. Como visto na Unidade 4, um deslocamento à esquerda em binário representa uma multiplicação por 2, portanto, dois deslocamentos à esquerda representam uma multiplicação por 4. O diagrama da Figura 73 ilustra a representação do circuito de deslocamento de *2 bits* à esquerda, usado para o MIPS.

Figura 73 Circuito de deslocamento de *2 bits* à esquerda.

6.4.7 Contador de programa

O circuito do contador de programa (*Program Counter*, PC) é um registrador de *32 bits*, no qual pode ser introduzido um novo valor ao acionar o sinal de controle de escrita. O circuito do contador de programa é mostrado na Figura 74, e o seu diagrama simplificado é ilustrado na Figura 75.

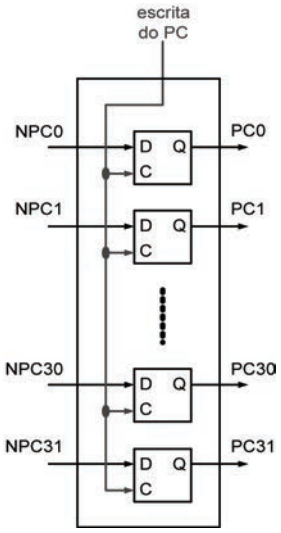


Figura 74 Contador de programa.

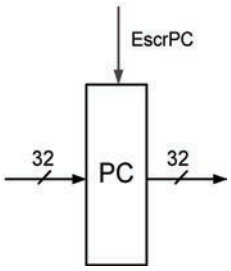


Figura 75 Diagrama simplificado do contador de programa.

6.5 Componentes do fluxo de controle do MIPS

Todos os componentes envolvidos no controle do computador são componentes do fluxo de controle. Esse controle é efetivado pelo uso dos sinais de controle em instantes adequados. Na descrição a seguir, é apresentado um dos principais componentes do fluxo de controle, o circuito de decodificação do código de operação e geração dos sinais de controle.

6.5.1 Circuito de controle: decodificação e geração dos sinais de controle para o MIPS monociclo

O circuito da Figura 76 mostra um circuito combinacional de decodificação do código de operação (*op* ou *opcode*), instrução [31-26]. A primeira porta *and* superior à esquerda decodifica o código 000000, fazendo com que sua saída, formato R, seja 1. Isso faz com que os sinais de controle ligados à linha da saída de formato R, sejam iguais a 1. São eles: *RegDst*, *EscrReg* e *OpULA1*.

Analogamente, quando o *opcode* é igual a 100011, a segunda porta *and* é acionada, fazendo com que os sinais ligados a sua saída sejam acionados para a execução da instrução *lw*. Os sinais acionados são os seguintes: *OrigULA*, *MempReg*, *EscrReg* e *LerMem*. Os sinais de controle acionados para o *opcode* 101011, correspondente à instrução *sw*, são: *OrigULA* e *EscrMem*. E, finalmente, os sinais de controle acionados para o *opcode* 000100, da instrução de desvio (*beq*), são: *Desvio* e *OpULA0*.

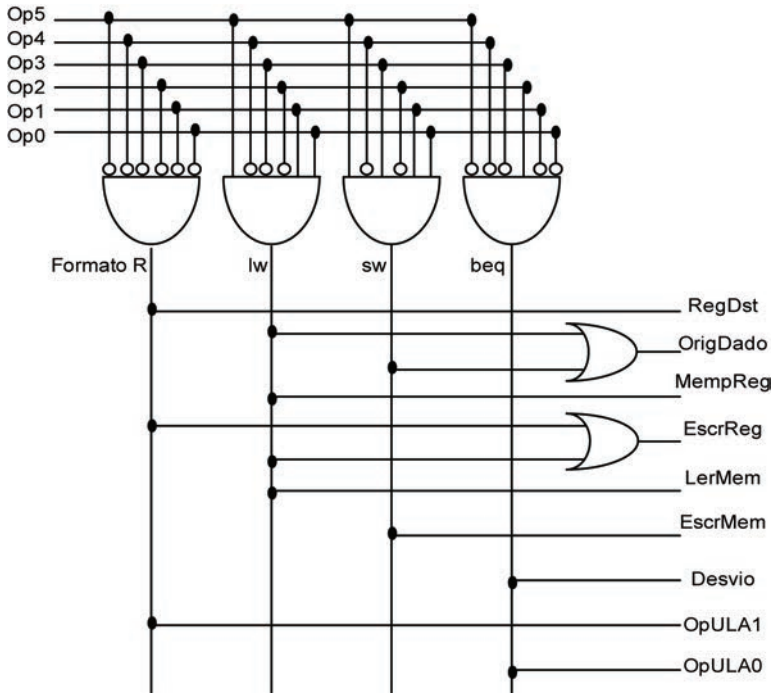


Figura 76 Circuito de decodificação do *opcode* e geração dos sinais de controle.

A Tabela 12 resume os sinais de controle gerados pelos *opcodes* decodificados pelo circuito da Figura 76.

Tabela 12 Sinais de controle e as respectivas instruções.

instrução	*RegDst*	*OrigDado*	*MempReg*	*EscrReg*	*LerMem*	*EscrMem*	*Desvio*	*OpULA1*	*OpULA0*
Formato R	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	x	1	x	0	0	1	0	0	0
beq	x	0	x	0	0	0	1	0	1

6.6 Projeto da organização do computador MIPS

A implementação da organização de um computador consiste em projetar todas as suas unidades funcionais, fazer as interconexões que possibilitem o processamento de todas as instruções previstas (*Instruction Set Architecture* – ISA), projetar o circuito de controle e interligar os sinais de controle, nos pontos de controle das unidades funcionais. São chamados pontos de controle os locais em que os sinais de controle, gerados pela unidade de controle, são interligados.

Esse projeto depende, no entanto, do número de ciclos que deve ter cada instrução. Num ciclo ocorre um conjunto de operações, resultantes dos sinais de controle acionados. Assim, um computador pode ser projetado para executar todas as instruções num ciclo. O computador resultante é monociclo. No caso do MIPS, lembrando que cada instrução vista no início do capítulo tem certo número de etapas a vencer, todas elas devem ocorrer num único ciclo. Assim, ao iniciar um ciclo, o computador tem um novo endereço de instrução no PC.

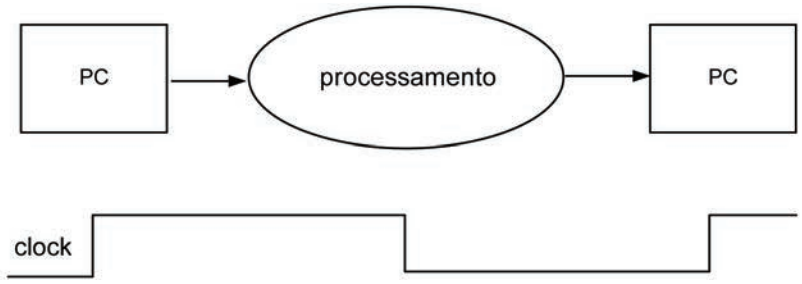


Figura 77 Diagrama de processamento num computador monociclo.

O diagrama da Figura 77 está representando o processador monociclo, que processa uma instrução num único ciclo e atualiza o contador de programa ao final deste.

Os computadores, no entanto, geralmente são construídos para executarem as instruções em múltiplos ciclos. Isso porque normalmente existem instruções simples e complexas nos computadores, e a implementação monociclo faz com que todas as instruções tenham uma mesma duração, que corresponde ao período do *clock*. Assim, mesmo as instruções que poderiam ter uma duração menor acabam ocupando um mesmo período de tempo num computador monociclo, pois esse período precisa ser obtido em função da instrução mais demorada.

Os computadores multiciclos dividem um ciclo de instrução em algumas etapas e faz com que cada etapa corresponda a um período de *clock*. Essas etapas são semelhantes às vistas no início da unidade para as diferentes classes de instruções do MIPS. Para que o fluxo de dados do computador execute em cada etapa uma determinada tarefa, os resultados do processamento anterior

são guardados em elementos de estado, para serem usados no ciclo de processamento seguinte. A Figura 78 mostra um diagrama ilustrativo de um processamento multiciclo de instruções, que é realizado por etapas.

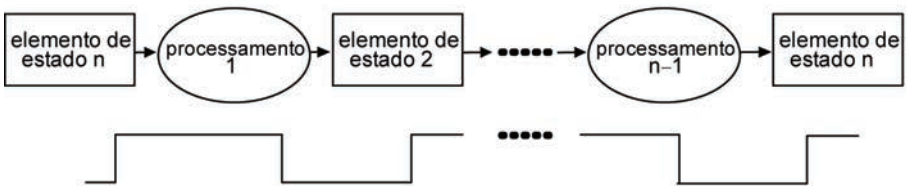


Figura 78 Diagrama de processamento num computador multiciclo.

Nos itens seguintes serão abordadas as implementações do MIPS. Inicialmente será apresentado o MIPS monociclo e, em seguida, um delineamento do MIPS multiciclo.

6.6.1 Organização do MIPS monociclo

O primeiro exemplo de implementação a ser apresentado é o MIPS monociclo; nele todas as instruções previstas são processadas num único ciclo.

Um ciclo de instrução se inicia com um novo endereço de instrução no contador de programa. Todas as unidades funcionais do computador, com exceção do PC, serão consideradas um único circuito que, durante o período de um ciclo, realiza o processamento da instrução. Após esse ciclo, o contador de programa recebe o endereço da próxima instrução.

A Figura 79 mostra um possível diagrama de fluxo de dados. Nele não estão presentes os circuitos de decodificação de *opcode* e geração dos sinais de controle, nem o circuito da lógica de seleção de operação da ULA.

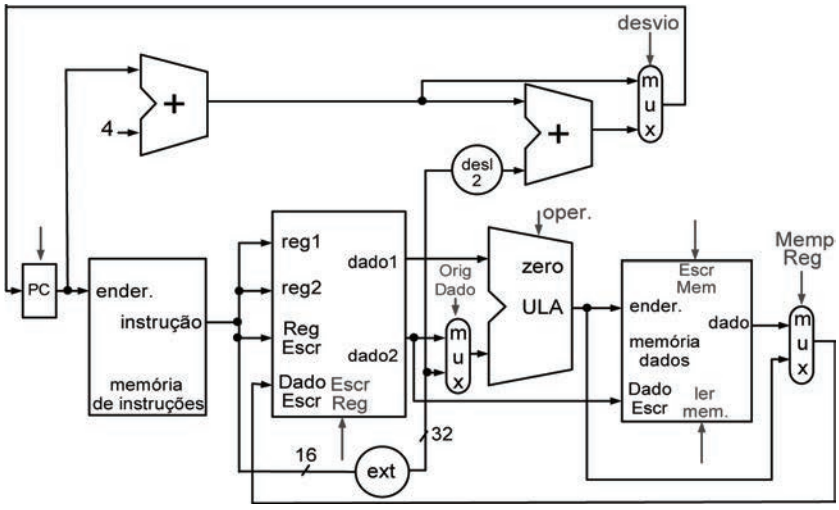


Figura 79 Construindo um fluxo de dados.

Nota-se que, à primeira vista, o diagrama é complexo, uma vez que corresponde à implementação da maioria das instruções vistas até o momento. Para que o diagrama se torne mais compreensível, o leitor deve tentar verificar, etapa por etapa, as instruções implementadas: aritméticas tipo *R*, *lw*, *sw* e *beq*, seguindo o funcionamento de cada uma das unidades funcionais. Os sinais de controle usados para cada uma das instruções implementadas são dados pela Tabela 12, que também define a seleção dos multiplexadores utilizados. Recomenda-se que o leitor tente descobrir tais seleções. Outro detalhe a ser observado é que foi usado um circuito somador de 4 na parte posterior ao PC. Esse circuito é usado para incrementar o PC para apontar para a próxima instrução sequencial, que fica a *4 bytes* posterior ao início da instrução atual.

Na Figura 80 foi feito um desdobramento das operações de leitura e escrita dos registradores.

O diagrama da figura seguinte mostra a implementação completa do MIPS monociclo para as instruções previstas, com o circuito de decodificação de *opcode* e geração dos sinais de controle e lógica de seleção de operação da ULA.

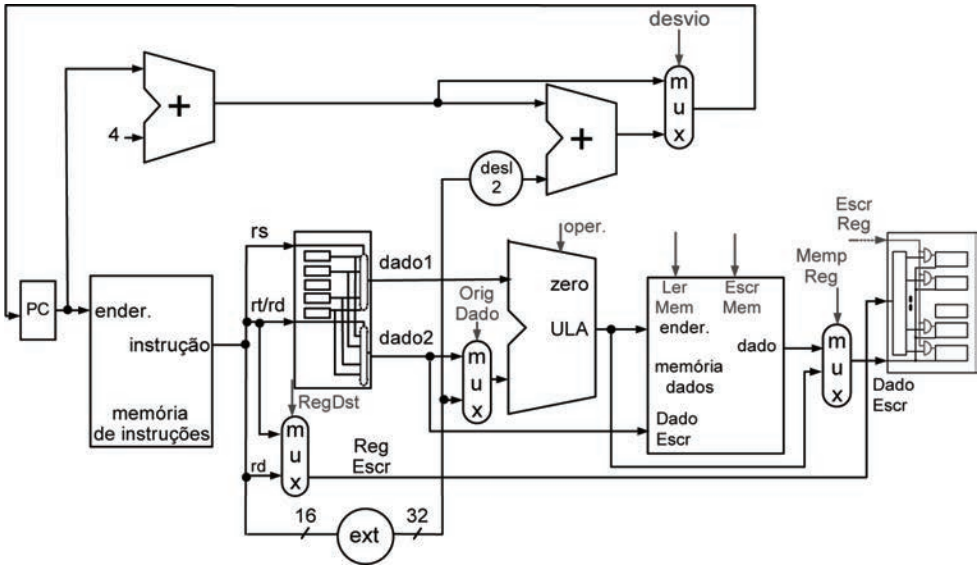


Figura 80 Representação alternativa da implementação monociclo do MIPS.

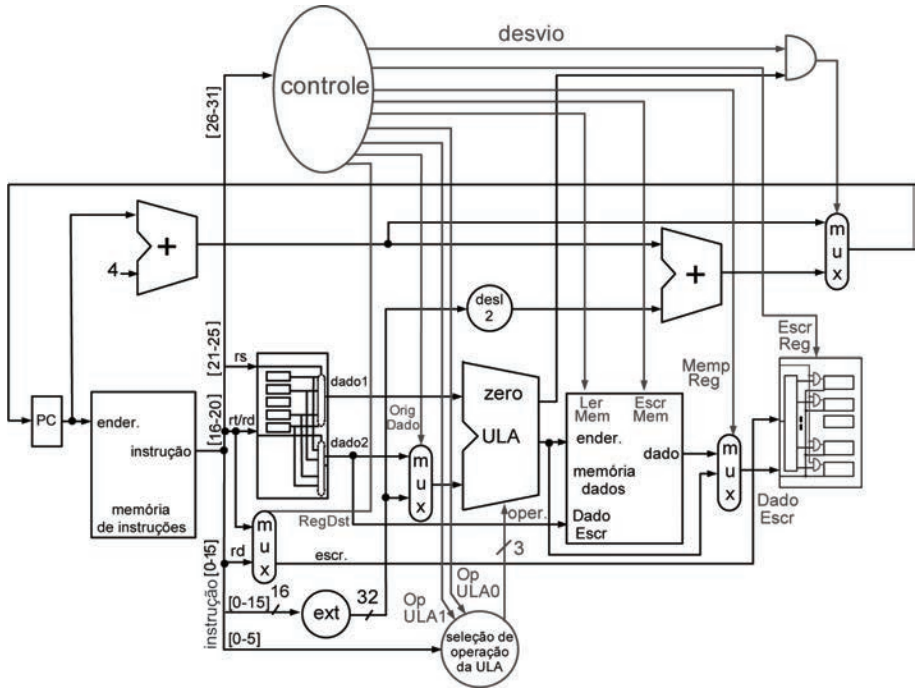


Figura 81 Representação completa da organização do MIPS monociclo.

A implementação monociclo considera que todas as instruções tenham um mesmo tempo de processamento, determinante do tempo de um ciclo, período de *clock*. Como na prática há instruções que levam mais tempo para processamento que outras, o período de *clock* é determinado pelo tempo de processamento da instrução mais morosa. Isso implica num período de *clock* grande e, consequentemente, baixo desempenho. A vantagem da implementação monociclo é a simplicidade da implementação.

6.6.2 Organização do MIPS multiciclo

A implementação de instruções em múltiplos ciclos é uma forma de melhorar o desempenho de um computador. Assim, uma instrução que leva mais tempo de processamento pode ser processada em mais ciclos relativamente a uma instrução com menos tempo de processamento.

A Figura 82 apresenta um diagrama que permite a implementação do MIPS multiciclo.

A diferença em relação à implementação monociclo é que foram acrescentados os elementos de estados intermediários: registrador de instrução RI, registradores A e B, registrador ULA e registrador RDM (Registrador de Dado de Memória). A cada início de um período de *clock*, um elemento de estado correspondente recebe um novo valor de entrada e o circuito correspondente realiza uma operação, carregando o elemento de estado seguinte com novo valor no final do ciclo. O PC, apesar de estar desenhado na extremidade esquerda do

diagrama, deve receber um novo endereço no final da execução das instruções, as quais usam números de ciclos diferentes.

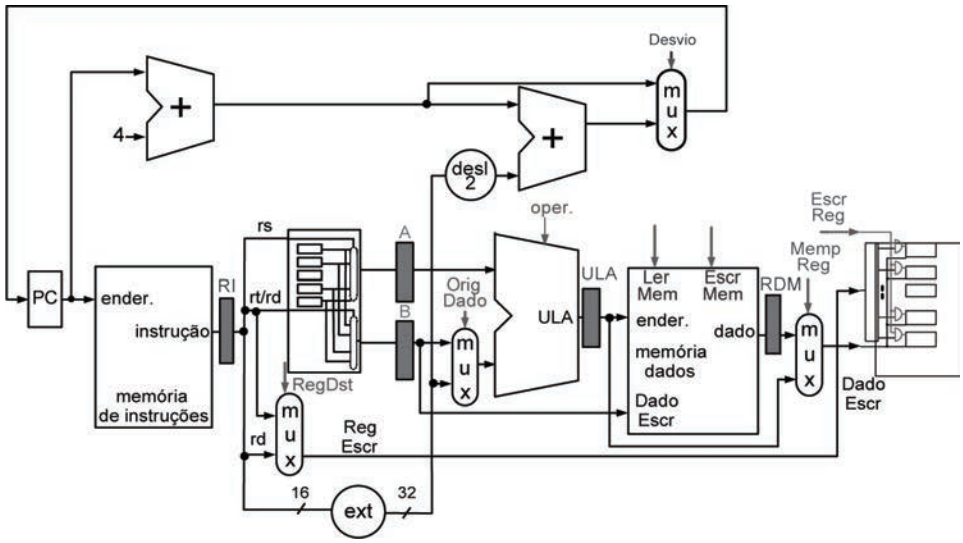


Figura 82 Diagrama do fluxo de dados do MIPS multiciclo.

A execução de uma instrução apresenta os seguintes passos ou ciclos:

1. leitura da instrução (*instruction fetch*), resultando em carga de RI, registrador de instrução;
2. leitura do banco de registradores, resultando em carga dos registradores A e B. Durante este passo, o *opcod*e da instrução é decodificado e os sinais de controle são gerados;
3. execução da instrução aritmética, computação do endereço de memória ou conclusão de uma instrução de desvio;
4. acesso à memória ou conclusão de uma instrução aritmética, formato R;
5. escrita no registrador (*write back*) pela instrução *lw*.

Cada passo implica em sinais de controle apropriados para as operações. Os dois primeiros passos são comuns para todas as instruções e implicam apenas no acionamento do *clock* nos elementos de estado RI e registradores A e B. A partir do passo 3, as operações dependem da instrução.

Assim, se a instrução for aritmética, de formato R, deve ser realizada a operação aritmética na ULA, no passo 3. Os dados de entrada são provenientes dos elementos de estado A e B e o resultado deve ser guardado no elemento de estado, *registrador ULA*. Os sinais de controle usados são: origem de dado para a ULA, *OrigDado = 0* para selecionar o multiplexador com entrada do registrador B, *OpULA1 =1* e *OpULA0 = 0* para ativar o circuito da lógica de seleção de operação

aritmética, por meio do campo de função com os *bits* F0,..., F5, *instrução [5-0]*. No passo 4, o resultado do passo 3 que está guardado no elemento de estado ULA deve ser escrito no registrador destino. Além disso, o próximo elemento de estado, o contador de programa, deve receber o endereço da próxima instrução. Os sinais de controle usados nesse passo são: *RegDst = 1* para selecionar o multiplexador quanto ao campo de instrução que indica o registrador para escrita (que no caso é o campo *instrução [15-11]*); *MempReg = 0* para selecionar o multiplexador quanto ao dado proveniente da ULA, para a escrita no registrador; e finalmente, o sinal de controle para escrita no banco de registradores, *EscrReg = 1*.

Para a instrução *lw*, após o passo 2, são necessários os seguintes passos. No passo 3 deve ser computado o endereço de leitura da memória de dados. Como entrada, são usados os valores do registrador *A* e os *16 bits* da instrução, *instrução [15-0]*, estendido em sinal para *32 bits*. Como saída é usado o *registrador ULA*. Os seguintes sinais de controle são usados: *OrigDado = 1*, para selecionar os *16 bits* da instrução, *instrução [15-0]*, estendida em sinal, como a segunda entrada da ULA, e *OpULA1 = 0* e *OpULA0 = 0* para selecionar a operação de adição, sem usar os *bits* da instrução, *instrução [5-0]*, uma vez que esse campo está sendo usado como endereço de deslocamento (*offset*) de memória, na instrução *lw*. O resultado do cálculo do endereço é guardado após o passo 3 no elemento de estado *ULA*. No passo 4, deve ser feita a leitura da memória de dados, usando como entrada o endereço calculado no passo 3 e guardado no elemento de estado, *registrador ULA*. Após a leitura, guardar o resultado no elemento de estado *RDM*. O sinal de controle é: *LerMem* para acionar a memória quanto à leitura. No passo 5, o dado lido na memória no passo 4, guardado no registrador RDM, é escrito no registrador destino, após o qual o PC recebe novo endereço de instrução. Os sinais de controle usados no passo 5 são: *RegDst = 0*, para indicar o campo de instrução (*instrução [20-16]*) como sendo o endereço do registrador destino, no multiplexador; *MempReg = 1* para indicar o dado lido na memória, contido no RDM, como dado a ser escrito no registrador destino; *EscrReg = 1* para acionar a escrita no banco de registradores.

Para a instrução *sw*, são os seguintes passos necessários a partir do passo 3. No passo 3 deve ser computado o endereço de escrita da memória de dados. Esse passo é exatamente igual ao passo 3 da instrução *lw*. No passo 4, deve ser feita a escrita da memória de dados usando, como entrada, o endereço calculado no passo 3 e guardado no *registrador ULA*, além do dado contido no registrador B. Após a escrita, o PC deve receber o novo endereço de instrução. O sinal de controle usado é: *EscrMem* = 1 para acionar a memória quanto à escrita.

Para a instrução *beq*, o passo 3 é o último. A operação realizada é calcular a diferença entre os operandos contidos nos elementos de estado de entrada, A e B, e caso a diferença resultar em zero, selecionar o multiplexador de seleção de

endereço de desvio para atualizar o novo endereço de PC. Os sinais de controle usados são: *OrigDado = 0* para selecionar o registrador B, como origem do dado para a ULA; *OpULA1 = 0* e *OpULA0 = 1* para selecionar a operação de subtração na ULA, sem a interferência do campo de função, *instrução [5-0]*, uma vez que esses campos estão sendo usados para computar o endereço de desvio, para ser usado caso haja desvio; *Desvio = 1*, para selecionar o multiplexador com o endereço de desvio, caso o resultado da operação na ULA seja 0.

Implementação da unidade de controle usando uma máquina de estado finito

As máquinas de estado finito são implementadas usando circuitos sequenciais e constituídas de um conjunto de estados: função de próximo estado (determinado pelo estado atual e entrada) e função de saída (determinada pelo estado atual e possivelmente entrada).

Usaremos uma máquina Moore (saída baseada somente no estado atual). O diagrama da Figura 83 mostra uma máquina de estado finito.

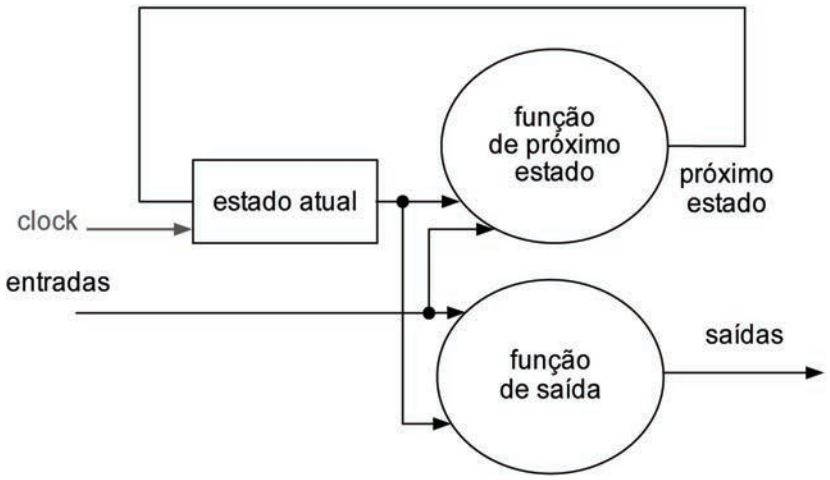


Figura 83 Diagrama de uma máquina de estado finito.

Diagrama de estado para a implementação das instruções do MIPS

O diagrama da Figura 84 mostra os sinais de controle necessários para a implementação do MIPS multiciclo, separados por estados, correspondentes aos passos ou ciclos. Nota-se que os dois primeiros estados, 0 e 1, são comuns a todas as instruções, correspondendo à busca da instrução na memória e leitura dos operandos. Os sinais de controle necessários nesses dois primeiros estados não estão detalhados no diagrama da Figura 84, tampouco no diagrama de fluxo de dados da Figura 82. Nota-se que um sinal de acionamento do registrador de instrução RI no estado 0, escrRI, é necessário para que o RI seja carregado com uma nova instrução apenas nesse estado. Assim, no estado seguinte, o PC pode ser incrementado apontando para a próxima instrução.

Os estados seguintes dependem das instruções. Assim, para a execução da instrução *lw*, são usados os estados 2, 3 e 4. Após o estado 4, o computador volta para o estado 0 e inicia a busca de uma nova instrução. Para a instrução *sw*, o estado 5 executa a mesma operação do estado 2 para a instrução *lw*, cálculo do endereço de memória, portanto os sinais de controle são iguais e os dois estados poderiam ser um único. A separação em dois estados é uma questão didática. A instrução *sw* termina no estado 6, após o qual o computador volta para o estado 0. A instrução aritmética com formato R passa pelos estados 7 e 8, e a instrução *beq* pelo estado 9.

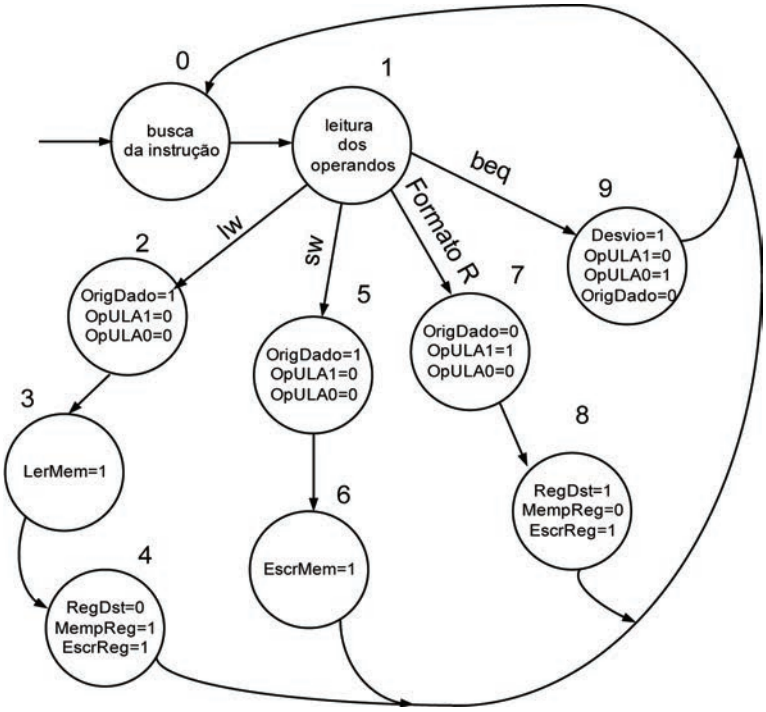


Figura 84 Diagrama de estados para a implementação do MIPS multiciclo.

Para a implementação da unidade de controle do MIPS multiciclo devemos construir um circuito que represente o diagrama de estados mostrado na Figura 84. Esse circuito é uma máquina sequencial, que passa pelos estados do diagrama em função do código de operação da instrução *opcode* (*instrução [31-26]*), e do estado atual. Em cada estado, deve gerar um conjunto de sinais de controle correspondentes para acionar as unidades funcionais convenientemente. Como o diagrama de estado contém 9 ciclos, o número de *bits* que representa cada um deles pode ser igual a 4. Representamos esses *bits* como: S3, S2, S1 e S0.

A Tabela 13 mostra as entradas, o estado atual, o próximo estado e as saídas da máquina sequencial do circuito de controle. As entradas são o *opcode* de *6 bits* (Op5, Op4, Op3, Op2, Op1, Op0). O estado atual é formado por *4 bits* (S3, S2, S1 e S0). As saídas são ao todo nove sinais de controle (*RegDst*, *OrigDado*, *MempReg*, *EscrReg*, *LerMem*, *EscrMem*, *Desvio*, *OpULA1* e *OpULA0*). Na

implementação da máquina sequencial de controle, Figura 85, o estado atual e as entradas devem gerar o próximo estado de *4 bits* (PS3, PS2, PS1 e PS0), por meio de um circuito combinacional (do próximo estado). As saídas somente dependem do estado atual e são geradas por um outro circuito combinacional (de saída).

Tabela 13 Estado atual, entrada (*opcode*), próximo estado e saídas (sinais de controle).

Estado atual	Opcode (Op5, Op4, Op3, Op2, Op1, Op0)						Próximo estado	Sinais de controle
0	X	X	X	X	X	X	1	(escrita no RI)
1	1	0	0	0	1	1	2	(incremento de PC)
2	X	X	X	X	X	X	3	OrigDado = 1, OpULA1 = 0, OpULA0 = 0
3	X	X	X	X	X	X	4	LerMem = 1
4	X	X	X	X	X	X	0	RegDst = 0, MempReg = 1, EscrReg = 1
1	1	0	1	0	1	1	5	(incremento de PC)
5	X	X	X	X	X	X	6	OrigDado = 1, OpULA1 = 0, OpULA0 = 0
6	X	X	X	X	X	X	0	EscrMem = 1
1	0	0	0	0	0	0	7	(incremento de PC)
7	X	X	X	X	X	X	8	OrigDado = 0, OpULA1 = 1, OpULA0 = 0
8	X	X	X	X	X	X	0	RegDst = 1, MempReg = 0, EscrReg = 1
1	0	0	0	1	0	0	9	(incremento de PC)
9	X	X	X	X	X	X	0	Desvio = 1, OpULA1 = 0, OpULA0 = 1, OrigDado = 0

A Tabela 14 corresponde à tabela-verdade do circuito combinacional de saída. As entradas são os *bits* do estado atual (S3, S2, S1 e S0) e as saídas são os sinais de controle. A Tabela 15 corresponde à tabela-verdade do circuito combinacional do próximo estado, em que as entradas são: o estado atual e o código de operação (*opcode*). A saída corresponde aos *4 bits* de próximo estado (PS3, PS2, PS1 e PS0).

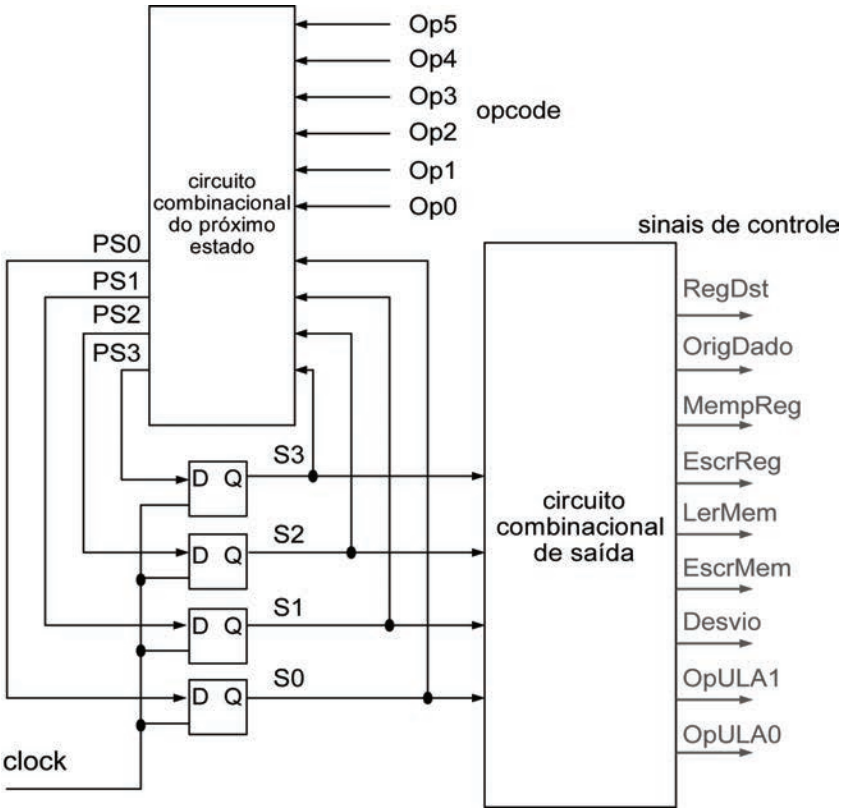


Figura 85 Diagrama da máquina de estado sequencial.

Tabela 14 Tabela-verdade do circuito combinacional de saída.

Entradas estado atual	Saídas Sinais de controle								
	Reg Dst	Orig Dado	Memp Reg	Escr Reg	Ler Mem	Escr Mem	Desvio	Op ULA1	Op ULA0
0	X	X	X	0	0	0	0	X	X
1	X	X	X	0	0	0	0	X	X
2	X	1	X	0	0	0	0	0	0
3	X	X	X	0	1	0	0	X	X
4	0	X	1	1	0	0	0	X	X
5	X	1	X	0	0	0	0	0	0
6	X	X	X	0	0	1	0	X	X
7	X	0	X	0	0	0	0	1	0
8	1	X	0	1	0	0	0	X	X
9	X	0	X	0	0	0	1	0	1

A implementação da Figura 85 é conhecida como controle fixo ou *hardwired*. Esse tipo de circuito de controle é eficiente, porém, quando o número de instruções aumenta, a complexidade também aumenta. Como os circuitos combinatórios de saída e de próximo estado são projetados em função do conjunto de instruções, quando alteramos o conjunto de instruções o projeto deve ser refeito, portanto, a implementação de controle fixo é inflexível.

Tabela 15 Tabela-verdade do circuito combinacional do próximo estado.

Entrada Estado atual e Opcode							Saída
Estado atual	Op5	Op4	Op3	Op2	Op1	Op0	Próximo estado
0	X	X	X	X	X	X	1
1	1	0	0	0	1	1	2
2	X	X	X	X	X	X	3
3	X	X	X	X	X	X	4
4	X	X	X	X	X	X	0
1	1	0	1	0	1	1	5
5	X	X	X	X	X	X	6
6	X	X	X	X	X	X	0
1	0	0	0	0	0	0	7
7	X	X	X	X	X	X	8
8	X	X	X	X	X	X	0
1	0	0	0	1	0	0	9
9	X	X	X	X	X	X	0

6.6.3 Unidade de controle microprogramado

Uma alternativa para melhorar a flexibilidade e a complexidade do circuito de controle é a implementação proposta por Maurice Wilkes em 1951, denominada de controle microprogramado.

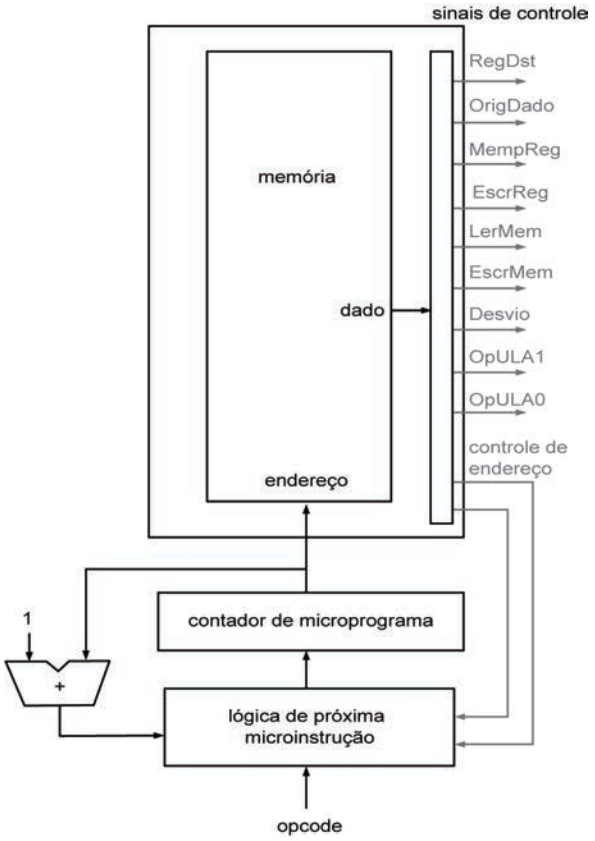


Figura 86 Unidade de controle microprogramado.

Conforme pode ser visto na Figura 86, o controle microprogramado usa uma memória para guardar os sinais de linhas de controle para cada ciclo, e, em cada um deles, uma nova palavra, denotada microinstrução, é lida nessa memória. Uma microinstrução contém alguns sinais de controle para o fluxo de dados e outros para o cálculo do endereço da próxima microinstrução. O endereço de leitura de uma nova micronstrução é contido num registrador chamado contador de microprograma. Esse endereço é obtido em função do endereço atual (estado atual) e de linhas de controle de endereço, contidos na microinstrução atual. Esse novo endereço pode ser sequencial, caso o endereço atual seja incrementado usando um somador de 1, ou pode não ser sequencial, gerado pela lógica da próxima microinstrução, que tem como entradas o *opcode* da instrução atual.

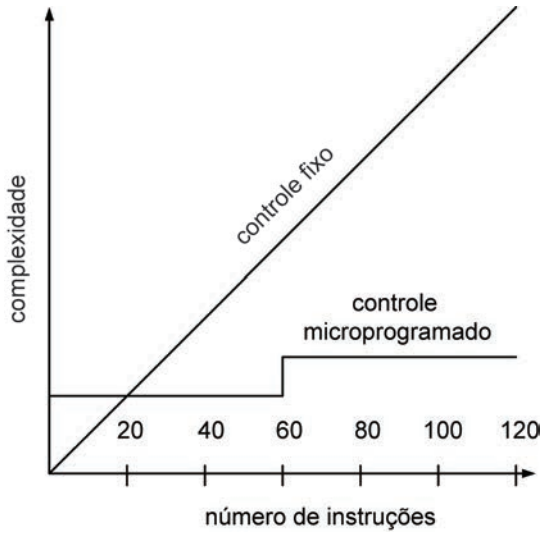


Figura 87 Gráfico comparativo da complexidade do controle fixo em função do controle microprogramado.

Uma vantagem do controle microprogramado é a flexibilidade, pois para aumentar o número de instruções no computador basta reescrever o conteúdo da memória de microinstruções, que é chamado de microprograma. Além disso, a complexidade da unidade de controle não aumenta com o número de instruções. A desvantagem em relação ao controle fixo é o desempenho, uma vez que a geração dos sinais de controle implica na leitura da memória de microprograma, o que faz com que o controle fique lento.

A Figura 87 mostra um diagrama comparativo da complexidade das duas unidades de controle. Nota-se que a complexidade da unidade de controle fixo cresce linearmente com o número de instruções, enquanto o controle microprogramado é constante para um grande incremento de instruções, variando levemente após certo número de instruções, em razão da alteração da memória usada.

6.6.4 Controle de exceções e interrupções

Exceções e interrupções são eventos de desvios diferentes da instrução *beq*, que mudam o fluxo normal de execução de instruções. A exceção é um evento inesperado que causa uma mudança no fluxo de controle, e tem origem dentro do processador (por exemplo, *overflow* aritmético). A interrupção é um evento inesperado que causa uma mudança inesperada no fluxo de controle, e tem origem fora do processador (por exemplo, digitação de um caractere no teclado). A arquitetura 80 x 86 (ou IA-32, Intel) usa o termo interrupção para todos esses eventos. Para evitar confusões, caracterizamos alguns eventos típicos de exceção e interrupção na Tabela 16.

Como manipular interrupções e exceções no MIPS

Quando ocorre uma exceção, as ações básicas a serem tomadas pela máquina são:

1. salvar o endereço da instrução afetada (instrução atual) num registrador especial EPC (*Exception Program Counter*);
2. transferir o controle (atualizar o endereço de PC) para um endereço específico, que possibilite ao computador executar a rotina de tratamento de exceção.

A rotina de tratamento de exceção depende do tipo de exceção. Um exemplo de rotina é reportar ao usuário que ocorreu um erro de *overflow*. Após a execução da rotina, pode-se encerrar ou continuar o programa que estava executando antes da exceção. O EPC é usado para determinar o endereço de reinício da execução. Para o computador manipular uma exceção, deve conhecer sua causa. A informação da causa pode ser obtida de duas formas: 1) usar o registrador de *status* (registrador *causa*) que indica a causa ou 2) usar interrupção vetorizada, caso o endereço de desvio para executar a exceção seja determinado pela causa da exceção.

Tabela 16 Caracterização dos tipos de evento entre exceção e interrupção no MIPS.

Tipo de evento	Origem	Terminologia
Solicitação de dispositivo de entrada/saída	externa	interrupção
Chamada de sistema operacional (SO) pelo programa de usuário	interna	exceção
Overflow aritmético	interna	exceção
Uso de uma instrução indefinida	interna	exceção
Mau funcionamento do hardware	ambas	exceção ou interrupção

Implementação de exceção no MIPS

Para o nosso estudo serão implementadas duas exceções: *overflow* aritmético e instrução indefinida. A causa é determinada pelo valor do registrador *causa* de *32 bits*: *causa* = 00 00 00 00 (em hexa), *overflow* aritmético; c*ausa* = 00 00 00 01 (em hexa), instrução indefinida, conforme Figura 88.

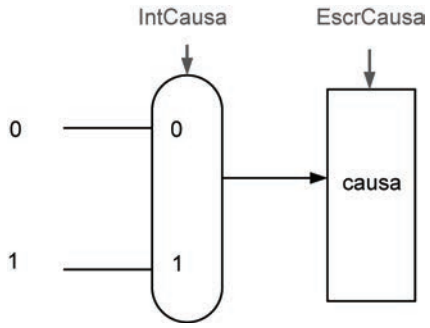


Figura 88 Registrador *causa* e o seu carregamento.

O endereço da instrução afetada é guardado no EPC (Figura 89). Nota-se que, caso o PC seja incrementado de 4 no segundo ciclo da instrução, logo após a carga do registrador de instrução RI, esse valor deve ser decrementado de 4, antes de ser escrito no EPC, para que o retorno da rotina de exceção ocorra no endereço da instrução interrompida.

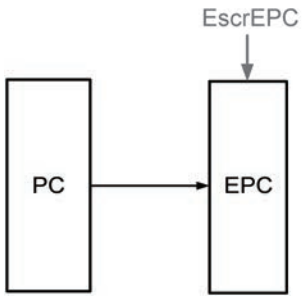


Figura 89 Registrador EPC para guardar o endereço de desvio.

O endereço de desvio para o início da execução da rotina de tratamento de exceção é dado por C0 00 00 00 (em hexa), conforme Figura 90.

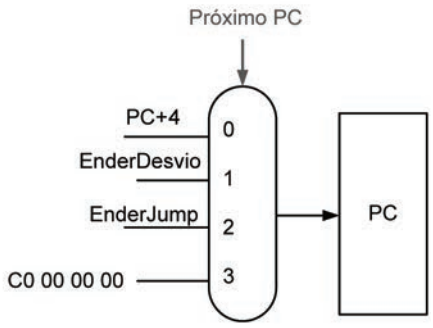


Figura 90 Seleção do endereço de desvio para a rotina de exceção.

A Figura 91 mostra o diagrama de fluxo de dados do MIPS multiciclo com os dispositivos de tratamento de exceção apresentados, no canto superior direito.

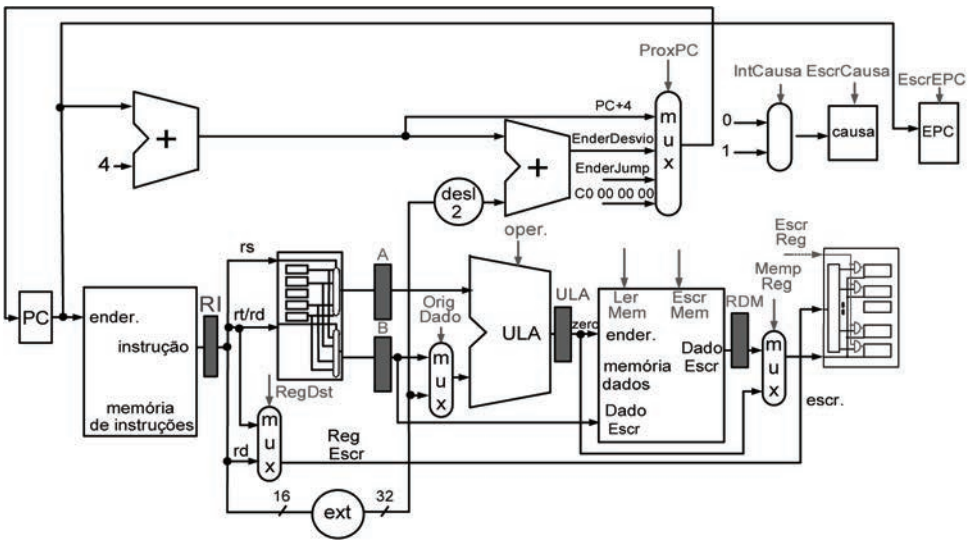


Figura 91 Diagrama de fluxo do MIPS multiciclo com os dispositivos de exceção.

A Figura 92 mostra o diagrama de estado com a inclusão de dois estados: 10 para a exceção de *overflow* aritmético e 11 para a exceção de instrução indefinida.

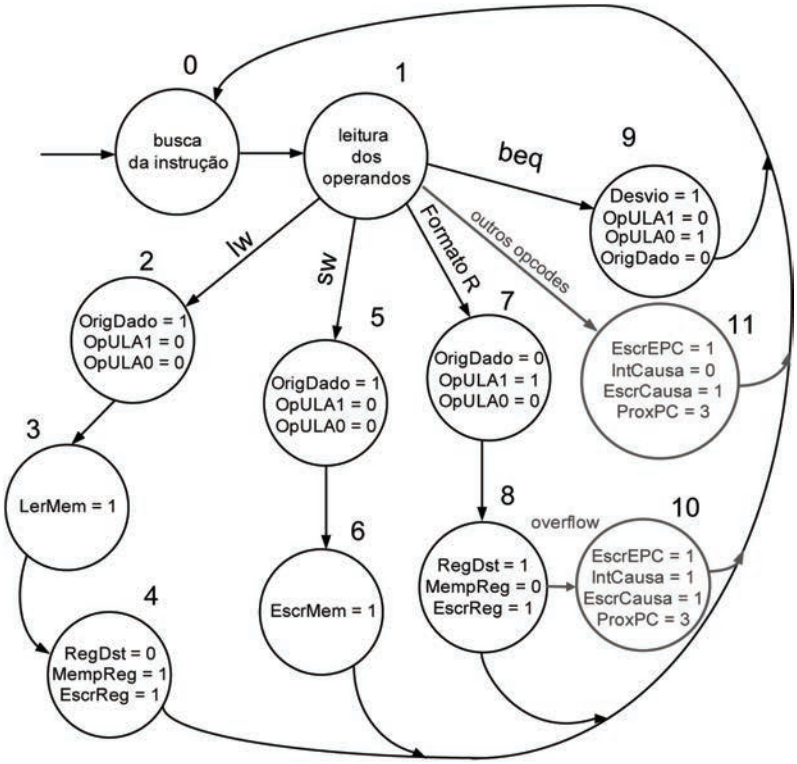


Figura 92 Diagrama de estado do MIPS multiciclo incluindo os dois estados de exceção (10 e 11).

6.7 Considerações finais

Nesta unidade foram descritos os principais tópicos relativos à implementação do MIPS, em monociclo e multiciclo. Foi também descrita a forma de manipulação de exceções pelo MIPS.

6.8 Estudos complementares

Para complementar os estudos sobre a organização do MIPS, os leitores podem se reportar ao quarto capítulo, *The Processor*, do livro de Hennessy & Patterson (2008), ou ao quinto capítulo, *O Processador: Caminho de Dados e Controle*, da versão traduzida por Daniel Vieira (HENNESSY & PATTERSON, 2005). Outras referências, como o livro de Weber (2000), podem auxiliar no entendimento da organização de computadores.

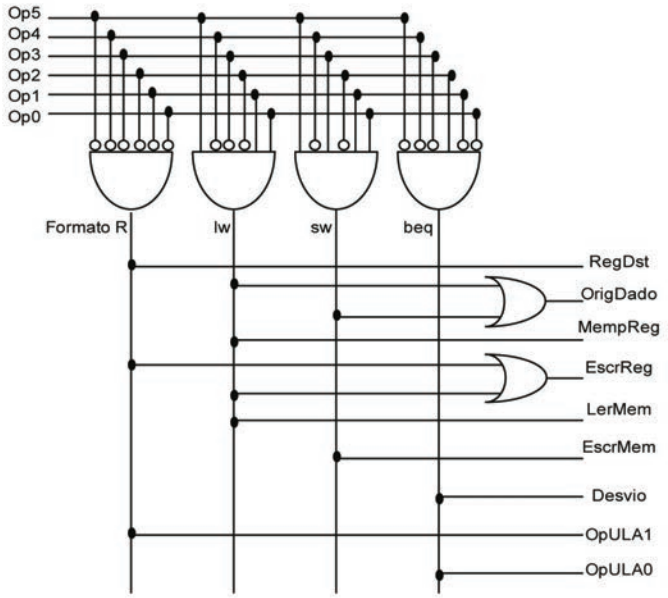
6.9 Exercícios

1. Descrever a função do contador de programa (Program Counter - PC) em um computador.

2. Descrever o formato binário das instruções aritméticas tipo R do MIPS.
3. Descrever o formato binário das instruções de referência à memória do MIPS.
4. Descrever o formato binário das instruções de desvio condicional do MIPS.
5. Descrever qual o artifício usado pela instrução beq ou bne para executar um desvio dentro de um programa.
6. Descrever o formato binário da instrução de desvio incondicional do MIPS.
7. Descrever qual o artifício usado pela instrução j para executar um desvio dentro de um programa.
8. Dada a tabela verdade da lógica de seleção de operação da ULA, transcrita abaixo, explicar por que as operações de adição e subtração se repetem na coluna de operação na extrema direita.

OpULA		Campo de função				operação	
OpULA1	OpULA0	F3	F2	F1	F0		
0	0	x	x	x	x	0 1 0	adição
0	1	x	x	x	x	1 1 0	subtração
1	0	0	0	0	0	0 1 0	adição
1	0	0	0	1	0	1 1 0	subtração
1	0	0	1	0	0	0 0 0	and
1	0	0	1	0	1	0 0 1	or
1	0	1	0	1	0	1 1 1	slt

9. Explicar para que serve o circuito de extensão de sinal de 16 para 32 bits no MIPS,
10. Explicar para que serve o circuito de deslocamento de 2 no MIPS.
11. Dado o circuito combinatório de controle do MIPS monociclo, na figura a seguir, verificar os valores dos sinais de controle quando o opcode é igual a:



a) 000000 (formato R)

b) 100011 (lw)

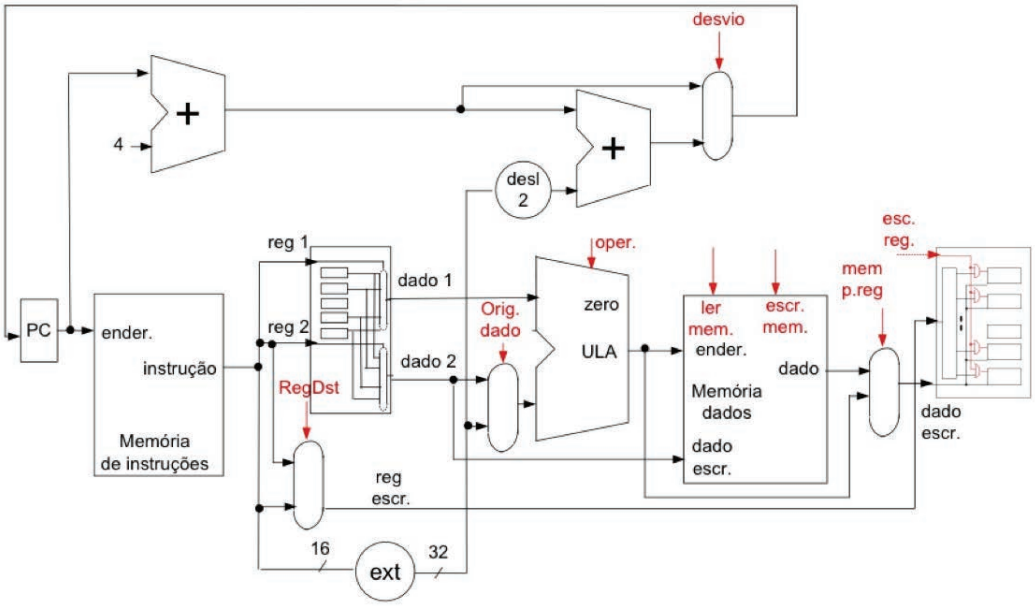
c) 101011 (sw)

d) 000100 (beq)

12. Analisar o fluxo de dados da figura a seguir e descrever:

    a) Para uma instrução aritmética, cada um dos multiplexadores RegDst, OrigDado e MemP.Reg tem 0 na parte superior e 1 na parte inferior ou 1 na parte superior e 0 na parte inferior?

    b) Para uma instrução beq o multiplexador de desvio tem 0 na parte superior e 1 na parte inferior ou 1 na parte superior e 0 na parte inferior?



13. Por que é necessário o multiplexador RegDst, no fluxo de dados do MIPS monociclo?

14. Por que é necessário o multiplexador OrigDado, no fluxo de dados do MIPS monociclo?

15. Por que é necessário o multiplexador Mem P.Reg, no fluxo de dados do MIPS monociclo?

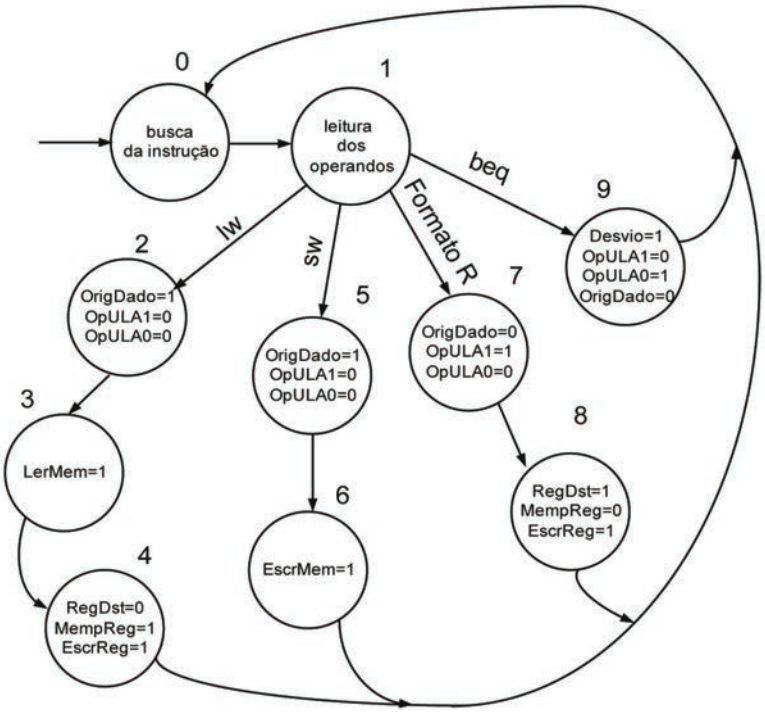
16. Por que é necessário o multiplexador de desvio, no fluxo de dados do MIPS monociclo?

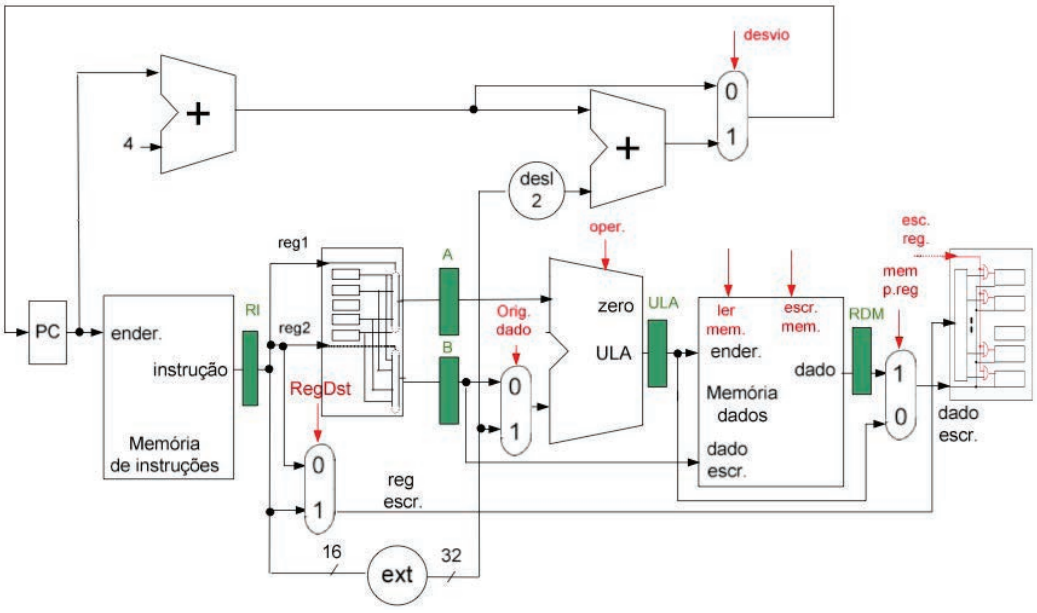
17. Quais são os registradores (elementos de estado) acrescentados na implementação do MIPS multiciclo, e por que eles são necessários?

18. Dado o diagrama de estado da unidade de controle do MIPS multiciclo e seu respectivo diagrama de fluxo de dados nas figuras a seguir, responder:

a) Quais são os operandos da ULA e qual é a operação aritmética no estado 2;

b) Qual é o dado a ser escrito no registrador no estado 4?





19. Considerando os diagramas da questão anterior, descrever os estados a serem adicionados, as operações realizadas nesses estados, e os respectivos sinais de controle necessários para implementar uma instrução addi, no MIPS multiciclo. Lembrando-se que essa instrução tem um formato tipo I, conforme mostrado abaixo:

opcode	rs	rd	1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0

20. Descrever as principais diferenças entre o controle fixo e o microprogramado.

21. Para implementar uma exceção no MIPS, como de overflow aritmético, qual a função dos registradores CAUSA e EPC, e como eles são usados?

UNIDADE 7

Pipeline e outras arquiteturas

7.1 Primeiras palavras

A Unidade 7 contempla a descrição dos princípios de *pipeline*, bem como outros mecanismos de paralelização aplicados em computadores.

7.2 Problematizando o tema

Vimos duas formas de implementação de computadores MIPS: monociclo e multiciclo. No MIPS monociclo, todas as instruções são executadas em um ciclo, cujo período de tempo é calculado em função da instrução mais demorada. Portanto, para uma instrução mais simples sobra um tempo de ociosidade até terminar um período de *clock* e iniciar uma nova instrução.

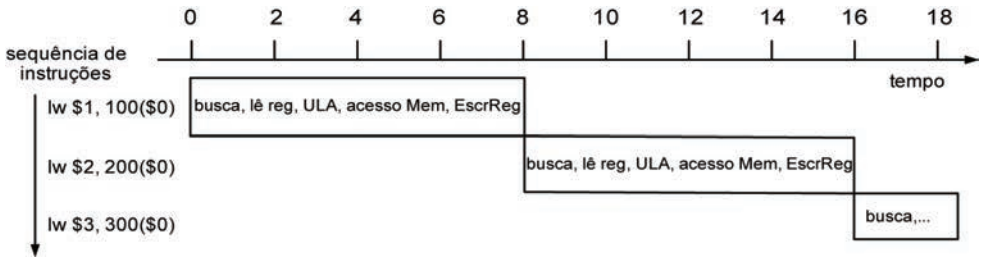


Figura 93 Sequência de execução de instruções no MIPS monociclo.

A Figura 93 representa um diagrama de tempo que mostra várias instruções *load-word*, executadas em sequência num MIPS monociclo. Para o tempo de ciclo, a busca da instrução, o acesso à memória de dados e uma operação na ULA foram considerados de *2 ns*. O tempo de leitura simultânea dos dois registradores do banco de registradores, bem como o tempo de escrita de um registrador foram considerados de *1 ns*. Portanto, um ciclo de instrução, sendo a soma de todos esses tempos, resulta em *8 ns*, e três instruções resultam em *24 ns*.

No MIPS multiciclo uma instrução é implementada em vários ciclos, com períodos menores em relação ao monociclo, de forma a usar mais ciclos para instruções mais demoradas, e menos ciclos para instruções mais simples. Apesar de implementar as instruções em vários ciclos, durante um determinado ciclo, apenas uma parte do processador funciona, enquanto outras partes ficam ociosas. Considerando o tempo de ciclo igual ao das etapas mais demoradas do monociclo, ou seja, *2 ns*, o tempo de execução de uma instrução *load-word* no MIPS multiciclo é de *10 ns*, ligeiramente superior aos *8 ns* do monociclo, e de *30 ns* para as três instruções (Figura 24). Nota-se que a diferença de tempo se dá pelo fato de o MIPS monociclo levar vantagem no tempo de leitura e escrita de registradores.

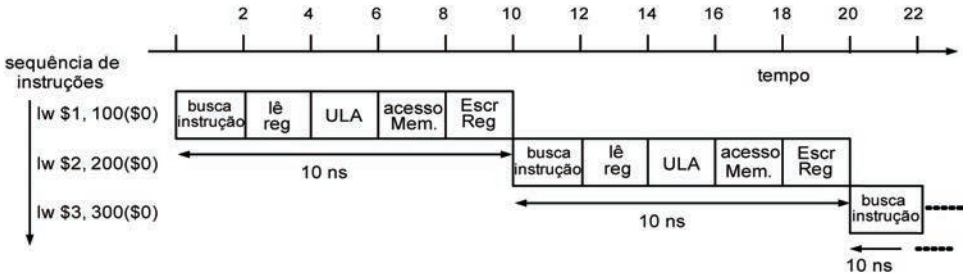


Figura 94 Sequência de execução de instruções no MIPS multiciclo.

Nesta unidade, uma nova forma de implementação denominada *pipeline* será descrita. Como o próprio nome diz, essa forma de implementação tenta desenvolver uma analogia com um duto (*pipe*) que conduz ininterruptamente alguma matéria. A cada instante, numa extremidade do duto ocorre a injeção (entrada) da substância, enquanto na outra extremidade ocorre a ejeção (saída) da matéria, e todas as partes do duto conduzem simultaneamente porções diferentes dessa matéria. Assim, o processamento de uma instrução é dividido em estágios e, em cada ciclo, é permitida a entrada de uma nova instrução enquanto outras instruções ainda estejam sendo executadas, cada qual num estágio diferente. Em média, quando uma instrução está se iniciando, outra está terminando a execução, e outras instruções estão em estágios intermediários. Veja a Figura 95.

O tempo de execução de três instruções *load-word* no MIPS *pipeline* é de *14 ns*, considerando as mesmas unidades funcionais das formas de implementação anteriores. Comparado com as duas formas anteriores (monociclo, *24 ns*, multiciclo, *30 ns*) essa forma de implementação é bem mais eficiente.

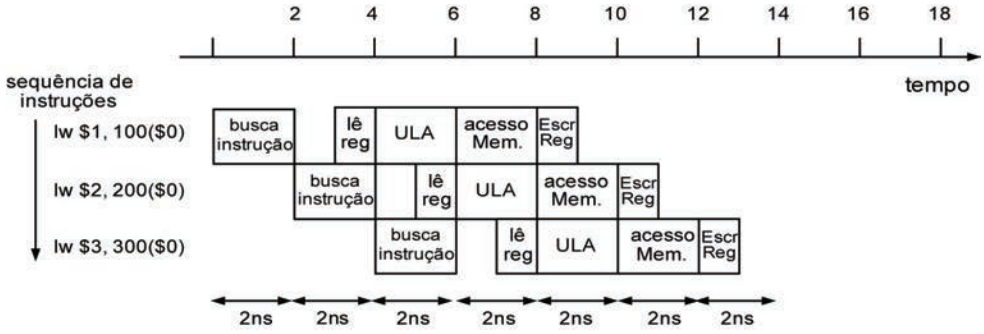


Figura 95 Sequência de execução de instruções no MIPS *pipeline*.

7.3 *Pipeline* em computadores

Um computador executa bilhões de instruções por segundo, portanto, uma medida de desempenho importante é o *throughput*, que é a taxa de execução de instruções. Podemos calcular o *speedup* para um programa de *n* instruções, num computador *pipeline* de *k* estágios, relativo a um computador multiciclo de *k* ciclos, considerando o mesmo tempo de ciclo para os dois computadores. O tempo de processamento para o computador multiciclo é obtido calculando o número de ciclos, $n \cdot k$, e multiplicando o resultado pelo tempo de ciclo. O tempo de processamento no computador *pipeline* é calculado somando o número de ciclos necessários à obtenção do primeiro resultado, *k*, com o número de ciclos que levam as outras instruções, $n-1$, e multiplicando o resultado pelo tempo de ciclo. O *speedup* é calculado dividindo o tempo de processamento no computador multiciclo pelo tempo de processamento no computador *pipeline*:

$$Speedup = \frac{n \cdot k}{k + n - 1}.$$

Para *n* grande, o *speedup* aproxima-se de *k*.

Para o computador *pipeline*, algumas características são importantes para facilitar a implementação: 1) todas as instruções têm a mesma duração; 2) poucos formatos de instruções; 3) operandos de memória aparecem somente em *loads* e *stores*.

Outras características dificultam a implementação: 1) conflitos estruturais: suponha-se que temos somente uma memória; 2) conflitos de controle: preocupar-se com instruções de desvio; 3) conflitos de dados: uma instrução depende de uma instrução prévia; 4) manipulação de exceções.

7.3.1 *Pipeline* no MIPS

A Figura 96 representa uma implementação do MIPS *pipeline*. Basicamente consiste na inserção de um *latch*, entre os estágios do *pipeline*, guardando todos os resultados de cada estágio, uma vez que, a cada ciclo, todos os estágios funcionam simultaneamente, diferentemente do MIPS multiciclo, no qual a cada ciclo, apenas parte do processador é ativa. Os *latches* inseridos no MIPS *pipeline* estão entre os: estágios de busca de instrução (IF, *instruction fetch*) e de decodificação (ID, *instruction decodification*), denotado IF/ID; estágios ID e execução (EX), denotado ID/EX; estágios EX e de memória de dados (M), denotado EX/M; e estágios M e escrita no registrador (WB, *write back*), denotado M/WB.

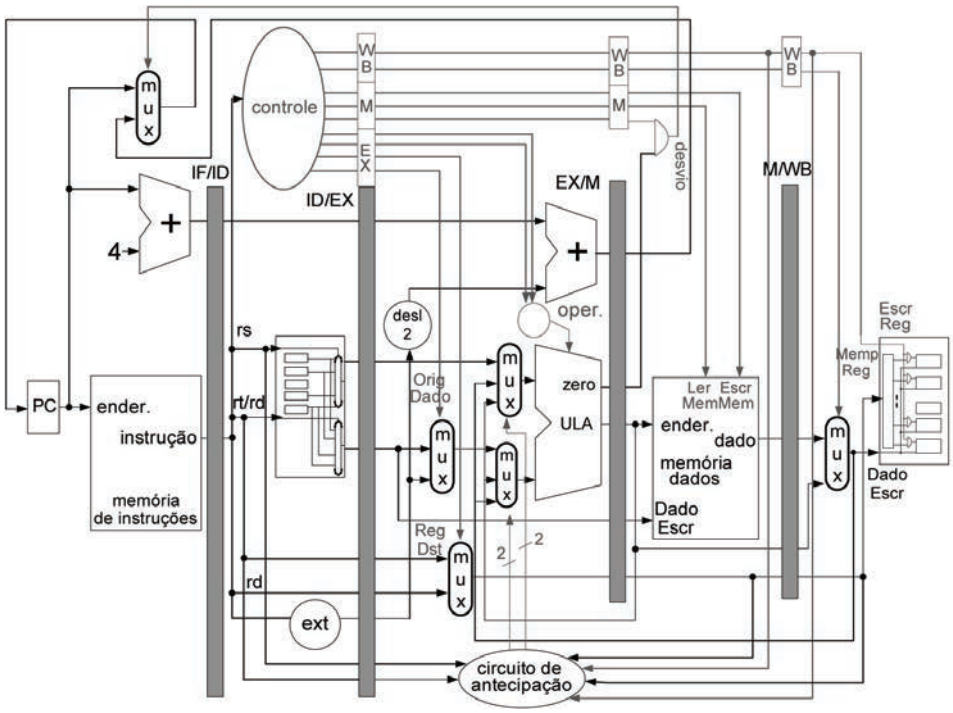


Figura 96 Implementação do MIPS *pipeline*.

Fonte: adaptada de Hennessy & Patterson (2008).

O contador de programa (PC) deve ser atualizado a cada ciclo para a busca da próxima instrução sequencial, usando PC+4 (isso é mostrado no canto esquerdo superior da Figura 96). Quando há uma instrução de desvio, o endereço de desvio é calculado no estágio EX e fornecido ao multiplexador de desvio, que fica acima do *latch* IF/ID no desenho. O banco de registradores é dividido em duas partes para ser compatível com a sequência dos estágios, uma vez que, no estágio ID, os registradores (reg1 e reg2) são lidos, enquanto, no estágio WB, o registrador para escrita (*RegEscr*) é escrito com o dado (*DadoEscr*).

Uma observação importante é que todas as instruções passam por todos os estágios do *pipeline*, tornando difícil terminar uma instrução pulando o estágio posterior; isso se dá pelo fato de esse estágio estar ocupado pela instrução antecessora.

A Figura 97 mostra um diagrama de tempo do processamento *pipeline* do MIPS em cinco estágios. Nota-se que todas as instruções usam um mesmo número de ciclos, daí a representação ser igual para cada uma das instruções, porém uma instrução subsequente é iniciada por um ciclo após a antecessora.

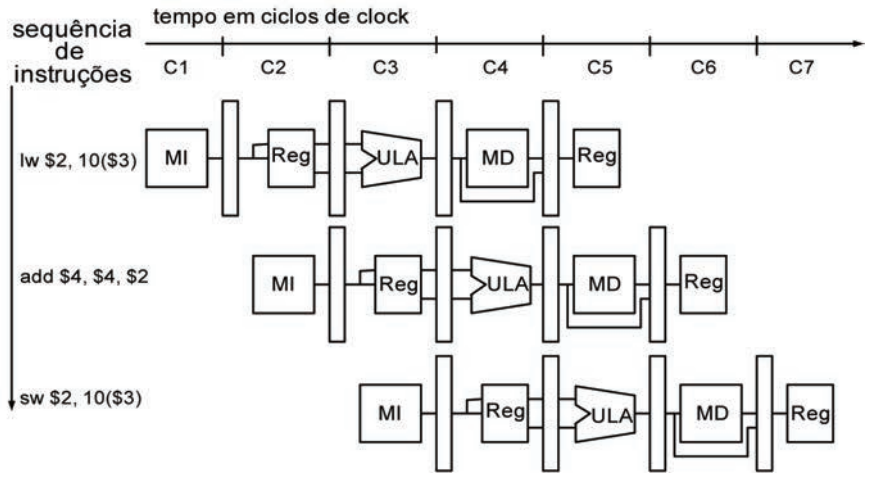


Figura 97 Diagrama de tempo do processamento no MIPS *pipeline*.

Fonte: adaptada de Hennessy & Patterson (2008).

7.3.2 Controle do *pipeline* no MIPS

A organização de controle do MIPS *pipeline* é diferente dos MIPS mono-ciclo e multiciclo, uma vez que num determinado instante podem existir cinco instruções, cada qual necessitando de sinais de controle específicos. Cada instrução usa sinais de controle específicos, porém em estágios subsequentes em tempos subsequentes.

Lembremos os sinais de controle já vistos nas outras implementações. Para o estágio EX são usados: *OrigDado* para selecionar o dado a ser operado na ULA; *OpULA1* e *OpULA0* para selecionar a operação aritmética; e RegDst para selecionar qual campo da instrução será usado para indicar o registrador destino ([20-16] ou [15-11]). Para o estágio M são usados: *LerMem* para ler a memória; *EscrMem* para escrever na memória; e *desvio* para selecionar o endereço de desvio. E, finalmente, para o estágio WB são usados os sinais: *EscrReg* para escrever no registrador e *MempReg* para selecionar o dado a escrever no registrador.

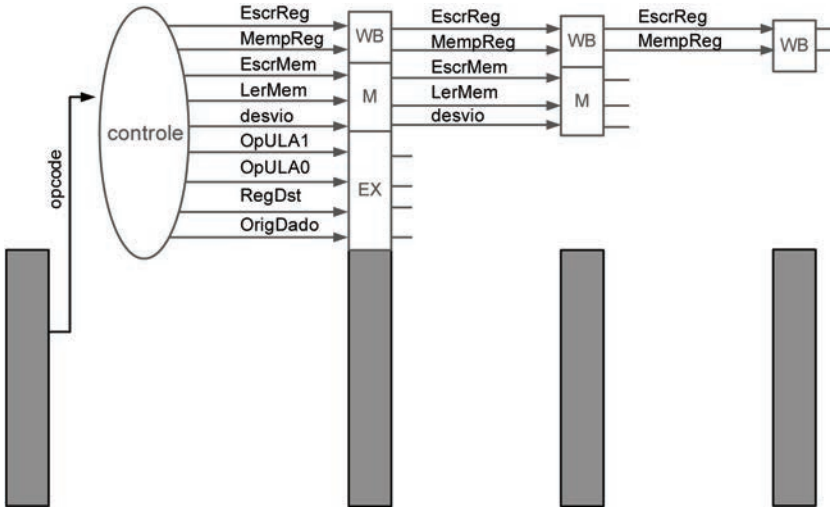


Figura 98 Geração e transmissão de sinais de controle no MIPS em *pipeline*.

Fonte: adaptada de Hennessy & Patterson (2008).

Uma forma de implementar o controle do MIPS em *pipeline* é gerar os sinais de controle no estágio ID e transmiti-los ao longo dos estágios, paralelamente aos dados resultantes de cada estágio. À medida que a instrução avança, vai consumindo os sinais de controle correspondentes aos estágios. A Figura 98 mostra o esquema de geração e transmissão dos sinais de controle no MIPS em *pipeline*, e a Figura 99 mostra o diagrama completo com os sinais de controle atuando sobre os pontos de controle.

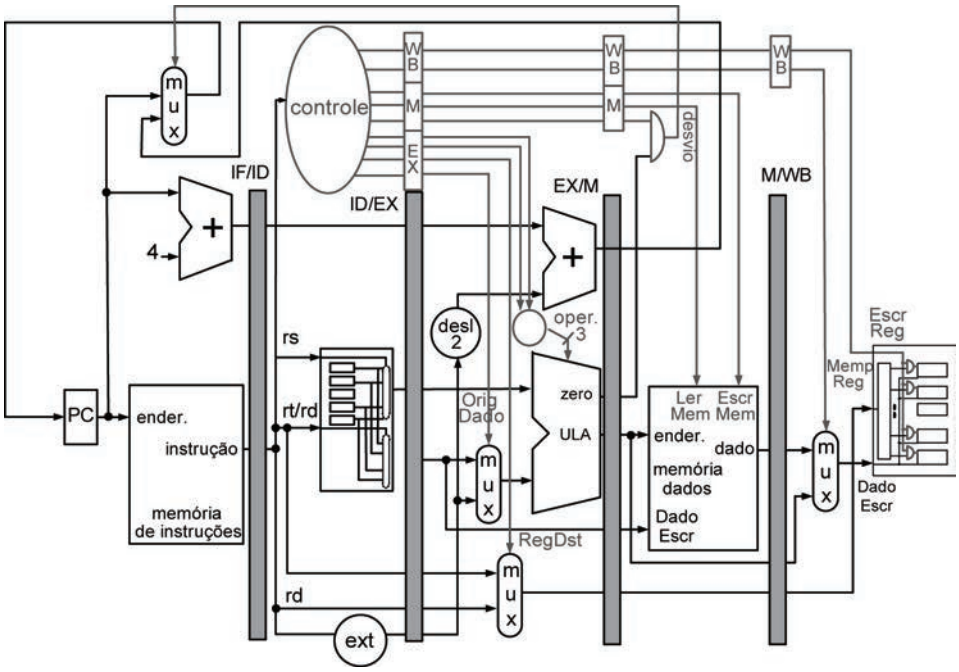


Figura 99 Diagrama do MIPS *pipeline* completo.

Fonte: adaptada de Hennessy & Patterson (2008).

7.4 Problemas que podem ocorrer com a sobreposição de instruções em *pipeline*

Na implementação do *pipeline* podem ocorrer alguns problemas (*hazards*, em inglês) que serão descritos neste item.

O primeiro problema pode ocorrer quando uma instrução escreve num registrador que é usado por instruções seguintes. Como a escrita no registrador corresponde ao último estágio do *pipeline*, se a próxima instrução depender desse registrador para leitura, o mesmo ainda não estará escrito. Esse problema, denominado dependência de dados, é ilustrado na Figura 100, na qual a segunda instrução (*and*) depende da primeira (*sub*), em relação ao registrador *$2*. Essa dependência existe também para a terceira instrução (*or*) da Figura 100. Para a quarta instrução (*add*) já não existe dependência, pois a escrita do registrador acontece na primeira metade do tempo de ciclo, e a leitura do registrador acontece após esse tempo. O mesmo acontece com a quinta instrução (*sw*).

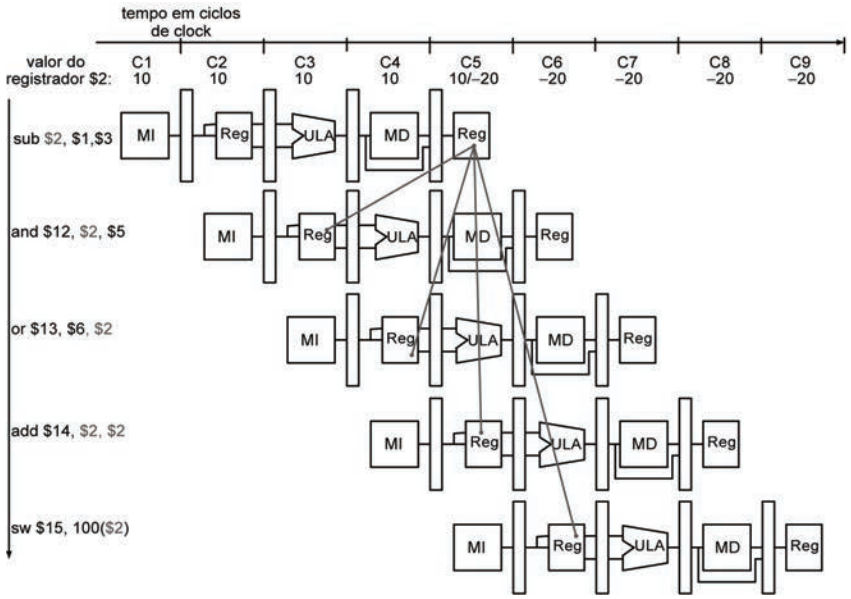


Figura 100 Dependência de dados no *pipeline*.

Fonte: adaptada de Hennessy & Patterson (2008).

7.4.1 Solução para a dependência de dados no MIPS *pipeline* usando antecipação

A solução para o problema de dependência no MIPS *pipeline* é a inclusão de um circuito de antecipação de dados. Isso é possível porque, apesar de o registrador de destino ainda não estar escrito no tempo em que a instrução posterior necessita do dado, este já está calculado pela ULA. Assim, o dado pode ser

antecipado da saída da ULA (relativa à instrução anterior *sub*) para a entrada da ULA (relativa à instrução subsequente *and*), conforme mostra a Figura 101, pela ligação entre as representações da primeira e da segunda instrução. Da mesma forma, o dado pode ser antecipado da saída do estágio de memória (relativa à instrução *sub*) para a entrada da ULA (relativa à terceira instrução). Como já observado, para a quarta instrução não existe necessidade de antecipação.

A Figura 102 mostra o diagrama do MIPS em *pipeline* completo com o circuito de antecipação inserido, mostrado na parte inferior no estágio EX do diagrama.

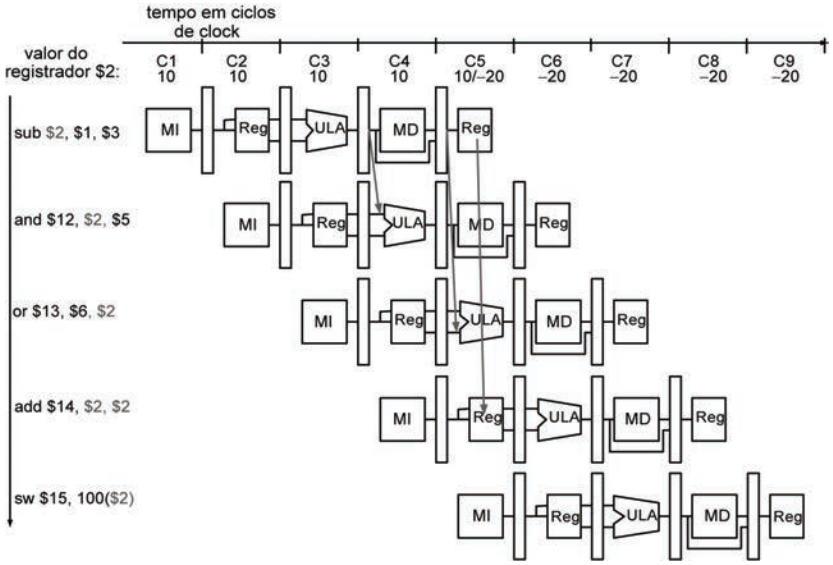


Figura 101 Antecipação de dados para solução da dependência de dados.

Fonte: adaptada de Hennessy & Patterson (2008).

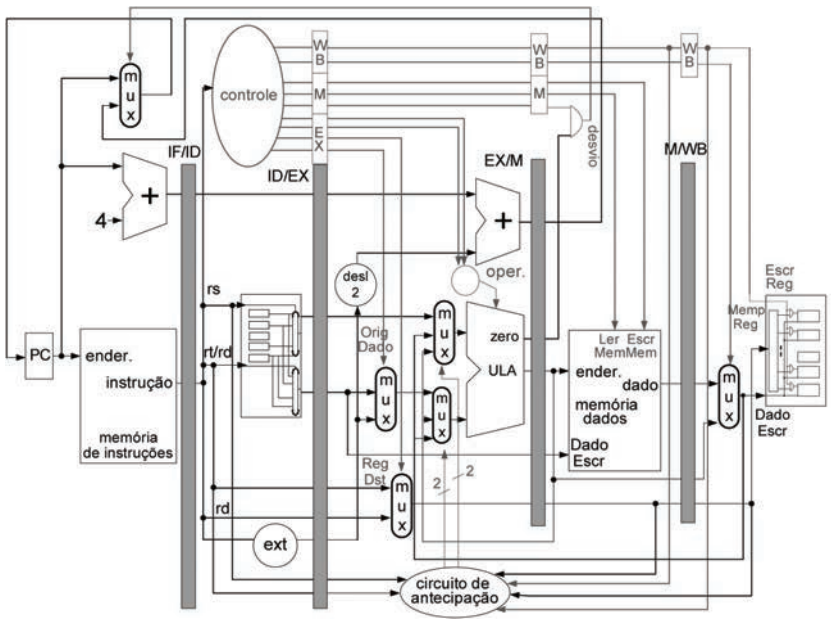


Figura 102 Diagrama do MIPS em *pipeline* com o circuito de antecipação no estágio EX.

Fonte: adaptada de Hennessy & Patterson (2008).

Nota-se que o circuito de antecipação deve interceptar o dado no estágio de execução. O circuito deve verificar se uma ou ambas as instruções anteriores escrevem num registrador, percebendo os sinais de controle de *EscrReg* respectivos. Caso escrevam, é preciso verificar se existe coincidência entre o número do registrador a ser escrito (*RegDst*) e o número do registrador que está sendo usado no estágio de execução EX. Caso exista coincidência, o dado correspondente deve ser antecipado, usando multiplexadores que foram inseridos nas entradas da ULA.

7.4.2 Problema de dependência do dado escrito pela instrução *load-word*

Nem sempre é possível antecipar um dado usando o circuito de antecipação. No caso da instrução *load-word*, como o dado a ser escrito é disponível apenas no estágio M, após a leitura da memória, se a instrução subsequente depender desse dado, não há como antecipá-lo para a entrada da ULA. A solução é paralisar um ciclo, as instruções seguintes à *load-word*. A Figura 103 apresenta um diagrama mostrando a dependência de dados relativos à instrução *load-word*. A segunda instrução, *and*, depende do valor do registrador *$2*, que é escrito pela primeira instrução *load-word*. A terceira instrução, *or*, também depende da primeira instrução *load-word*, porém, nesse caso existe a possibilidade de antecipação, como pode ser verificado no diagrama. O mesmo acontece com a quarta instrução, *add*.

A solução para a dependência à instrução *load-word* da instrução subsequente é a parada (*stall*, em inglês) de um ciclo para as instruções subsequentes. A Figura 104 mostra um diagrama em que as duas instruções subsequentes à *load-word* param enquanto as demais instruções prosseguem. A parada implica na inserção de uma “bolha” no diagrama, repetindo os mesmos estágios para as duas instruções. Assim, a instrução *load-word* é adiantada em dois estágios em relação à instrução subsequente, permitindo a antecipação de dados após a leitura de memória.

Quando se faz uma inserção de parada, é necessário zerar os sinais de controle para o estágio subsequente à instrução *load-word*, para garantir que nada será feito nos estágios logo após a instrução. O diagrama da Figura 105 mostra os estágios das instruções com a inserção da instrução *nop* (*no operation*), que corresponde ao zeramento dos sinais de controle.

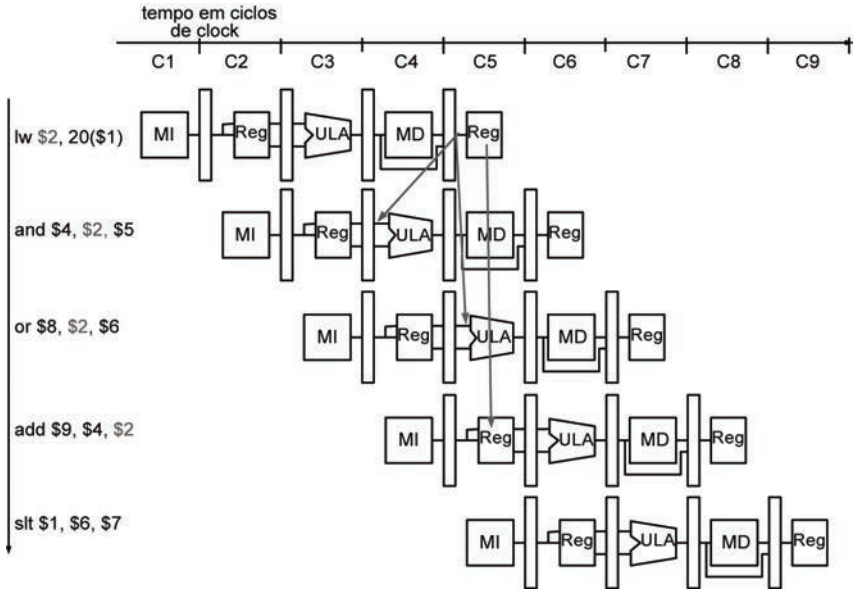


Figura 103 Diagrama mostrando a dependência de dados para a instrução *lw*.

Fonte: adaptada de Hennessy & Patterson (2008).

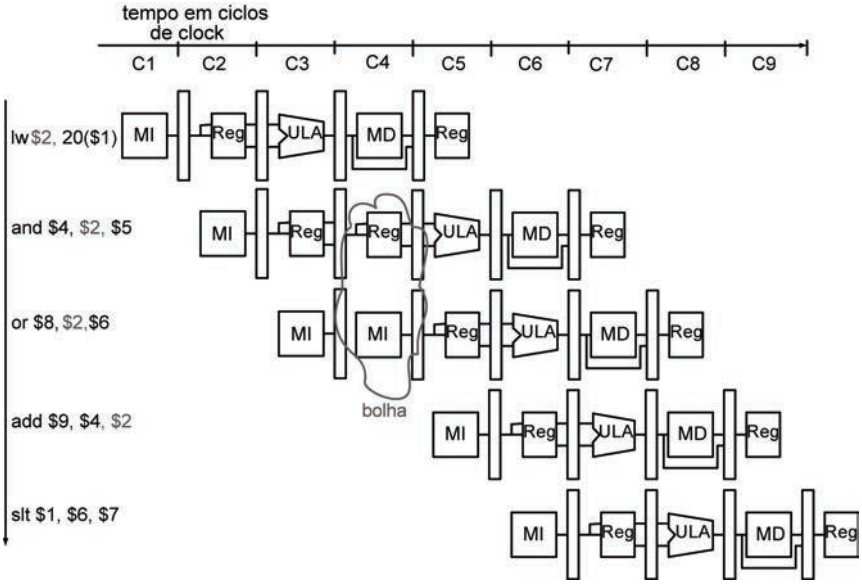


Figura 104 Inserção de parada.

Fonte: adaptada de Hennessy & Patterson (2008).

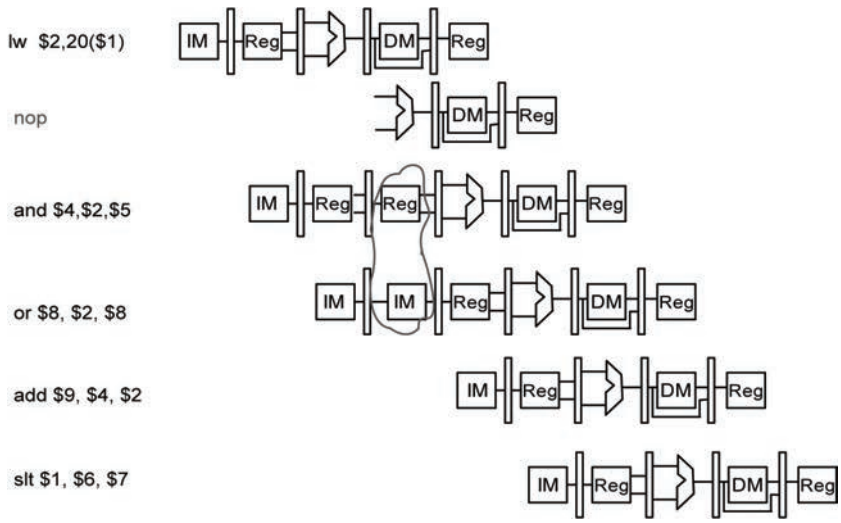


Figura 105 Inserção de estágios para *nop* (*no operation*).

Fonte: adaptada de Hennessy & Patterson (2008).

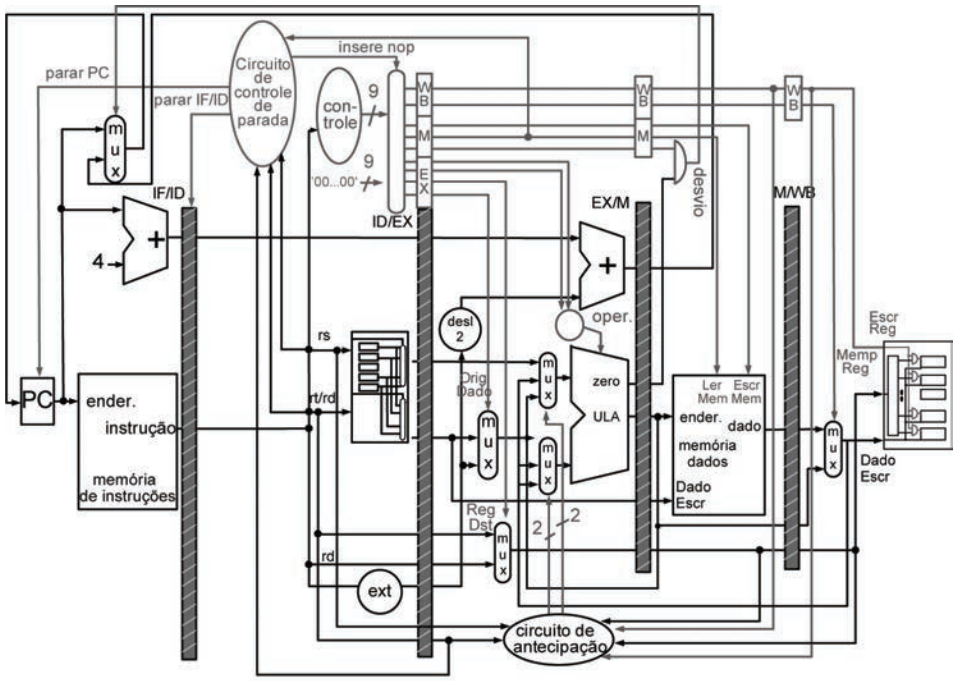


Figura 106 Circuito de controle de parada inserido no lado superior do estágio ID.

Fonte: adaptada de Hennessy & Patterson (2008).

A detecção do problema é feita no estágio ID, no qual a instrução subsequente se encontra quando a instrução *load-word* está no estágio EX. Caso o sinal de controle *LerMem* estiver ativo no estágio EX significa que a instrução é *load-word*. O circuito de controle de parada verifica se um dos registradores usados no estágio ID coincide com o registrador de escrita da instrução *load-word*. Caso isso aconteça, o circuito de controle de parada toma providências para as paradas dos estágios IF e ID, além de inserir os sinais de controle '00...00'

usando um multiplexador após o circuito de controle para inserir *nop* no estágio subsequente à instrução *load-word*. A Figura 106 mostra o circuito de controle de parada, inserido no lado superior do estágio ID.

7.4.3 Problema da instrução de desvio

Numa instrução de desvio condicional, quando é decidido pelo desvio, outras instruções estão em *pipeline*. Assim, as instruções subsequentes são necessárias desde que não ocorra desvio, caso contrário, elas devem ser descartadas. O diagrama da Figura 107 destaca as instruções subsequentes à instrução *beq* que devem ser descartadas e aponta para a instrução *lw* para a qual a execução deve ser desviada, caso a condição de desvio seja confirmada. Como essa confirmação ocorre no estágio M, três instruções subsequentes devem ser descartadas.

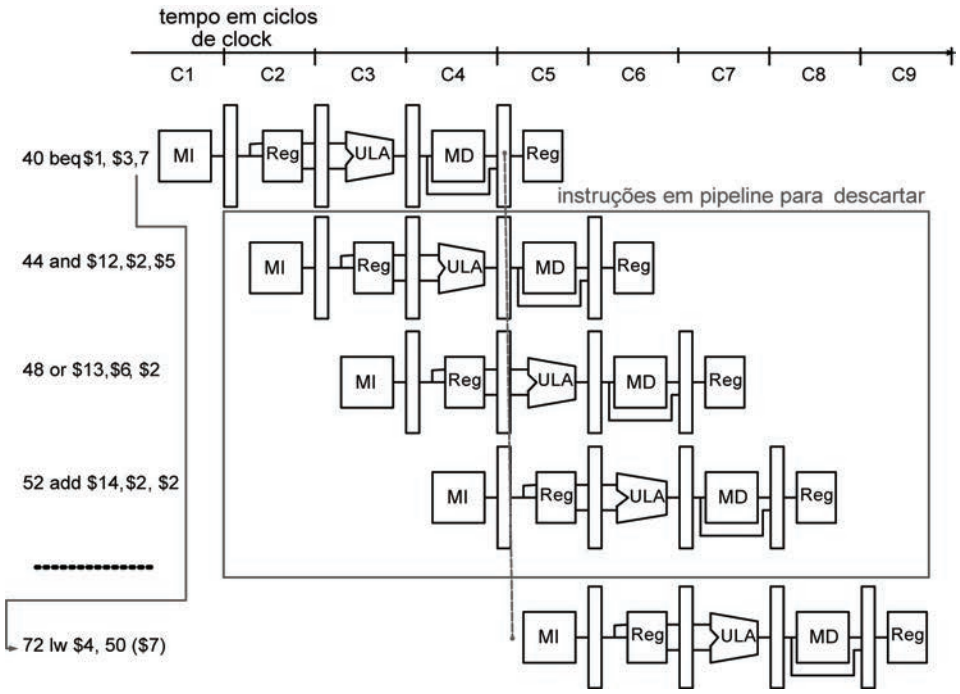


Figura 107 Diagrama de conflito para uma instrução de desvio.

Fonte: adaptada de Hennessy & Patterson (2008).

Para solucionar a instrução de desvio, basta incluir um circuito que descarte as instruções posteriores caso se confirme o desvio. Quanto mais cedo ocorrer a confirmação, melhor, pois menos instruções precisarão ser descartadas. Uma forma de aperfeiçoar a solução é inserir o circuito no estágio ID, antecipando a verificação da condição de desvio e o cálculo do endereço.

O diagrama da Figura 108 mostra o circuito de detecção da condição de desvio (que é um circuito de comparação de igualdade), ligado aos dois valores de

registradores, ativado pelo sinal de controle de desvio proveniente do circuito de decodificação e controle, que decodifica a instrução de desvio. O circuito de detecção, ao confirmar o desvio, limpa a instrução lida pelo estágio IF e atualiza o PC com o endereço de desvio. Assim, apenas uma instrução, contida no IF, é descartada.

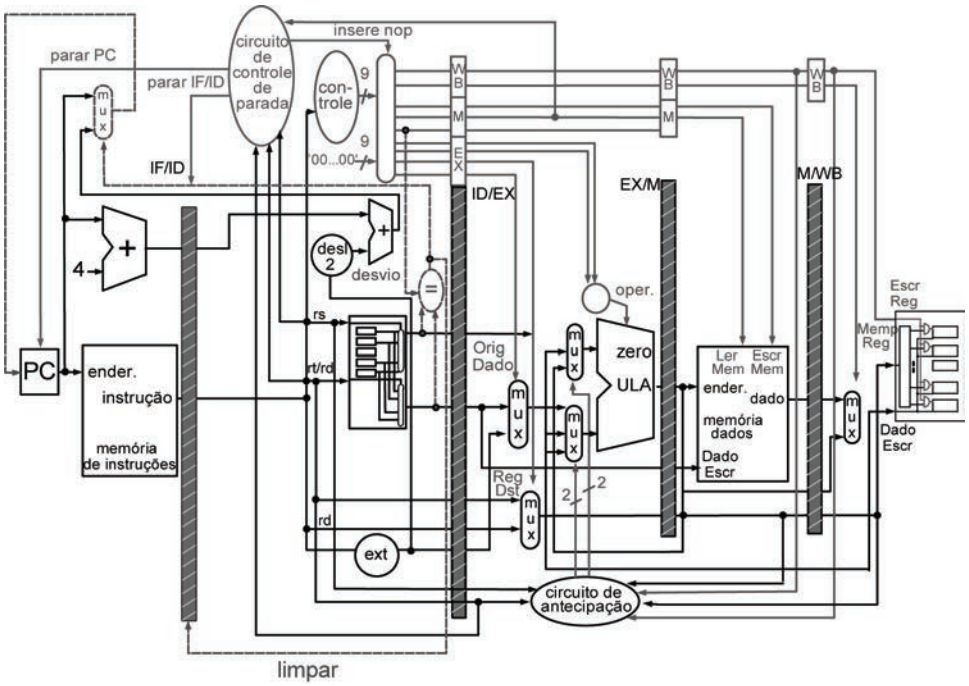


Figura 108 MIPS *pipeline* com circuito de detecção de condição de desvio.

Fonte: adaptada de Hennessy & Patterson (2008).

O diagrama da Figura 109 mostra os estágios das instruções após uma instrução de desvio *beq*, com o uso do circuito de detecção de condição de desvio anteriormente descrito.

Existem formas para melhorar o desempenho de arquiteturas *pipeline*, como fazendo uso de inserção de instrução após *beq*, a qual sempre deve ser executada, haja ou não desvio. Com a inclusão dessa instrução que ocupa a posição subsequente à *beq* (*branch delay slot*), não existe necessidade de limpar a instrução subsequente pelo circuito de detecção de condição de desvio. Quem deve inserir essa instrução para o *branch delay slot* é o compilador.

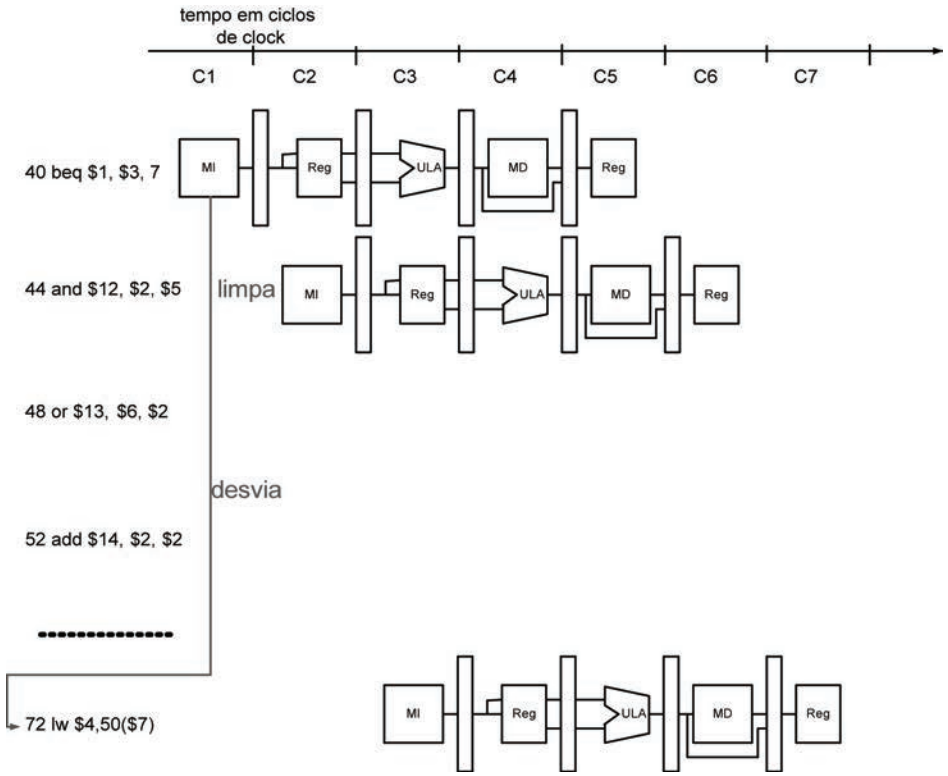


Figura 109 Diagrama dos estágios das instruções posteriores à instrução *beq*.

Fonte: adaptada de Hennessy & Patterson (2008).

7.5 Outras arquiteturas

Neste item, serão descritas algumas evoluções nas arquiteturas de computadores, como os conceitos de processamento superescalar, escalação dinâmica de instruções, processamento VLIW e multiprocessamento.

7.5.1 Processador superescalar

A arquitetura de processamento superescalar está presente na maioria dos processadores de uso geral, e caracteriza-se pela capacidade de iniciar e executar mais de uma instrução num mesmo ciclo. No caso do MIPS, tal capacidade pode ser derivada da memória de instruções, permitindo a leitura simultânea de duas instruções. O novo banco de registradores teria uma capacidade de acesso à leitura pelas duas instruções, permitindo quatro leituras de registradores simultaneamente. O estágio seguinte seria composto de duas ULAs, uma para as instruções de referência à memória calcularem o endereço e a outra para demais instruções. No último estágio a escrita de registrador seria duplicada, podendo ser usada por uma instrução aritmética ou pela instrução *load-word*. Com essa arquitetura, as instruções de referência à memória (*lw* e *sw*) podem ser executadas

paralelamente às demais instruções. A Figura 110 mostra um diagrama simplificado de um MIPS superescalar.

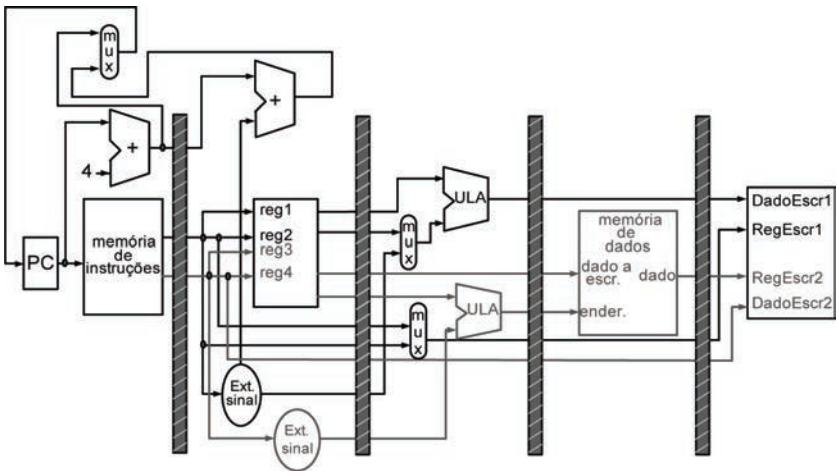


Figura 110 Diagrama de um MIPS superescalar.

Tipo de instrução	Estágios do *pipeline*							
R ou desvio	IF	ID	EX	MEM	WB			
Load/store	IF	ID	EX	MEM	WB			
R ou desvio		IF	ID	EX	MEM	WB		
Load/store		IF	ID	EX	MEM	WB		
R ou desvio			IF	ID	EX	MEM	WB	
Load/store			IF	ID	EX	MEM	WB	
R ou desvio				IF	ID	EX	MEM	WB
Load/store				IF	ID	EX	MEM	WB

Figura 111 Diagrama de execução de instruções na arquitetura superescalar.

O diagrama de estágios da Figura 111 ilustra as instruções de memória (*Load* e *Store*) processadas paralelamente às demais instruções (R ou desvio). A evolução dos ciclos é representada na horizontal, e a emissão de novas instruções, na vertical. Cada coluna do diagrama representa um ciclo.

7.5.2 Escalação dinâmica de instruções

Outra característica presente nas arquiteturas de uso geral é a escalação dinâmica de instruções, realizada pelo *hardware*. O *hardware* tenta encontrar instruções para executar. Nesse caso é possível a execução fora da ordem original do programa. É possível também a execução especulativa e a previsão dinâmica de desvio. Resume-se na Figura 112 um diagrama de fluxo de uma arquitetura com escalação dinâmica.

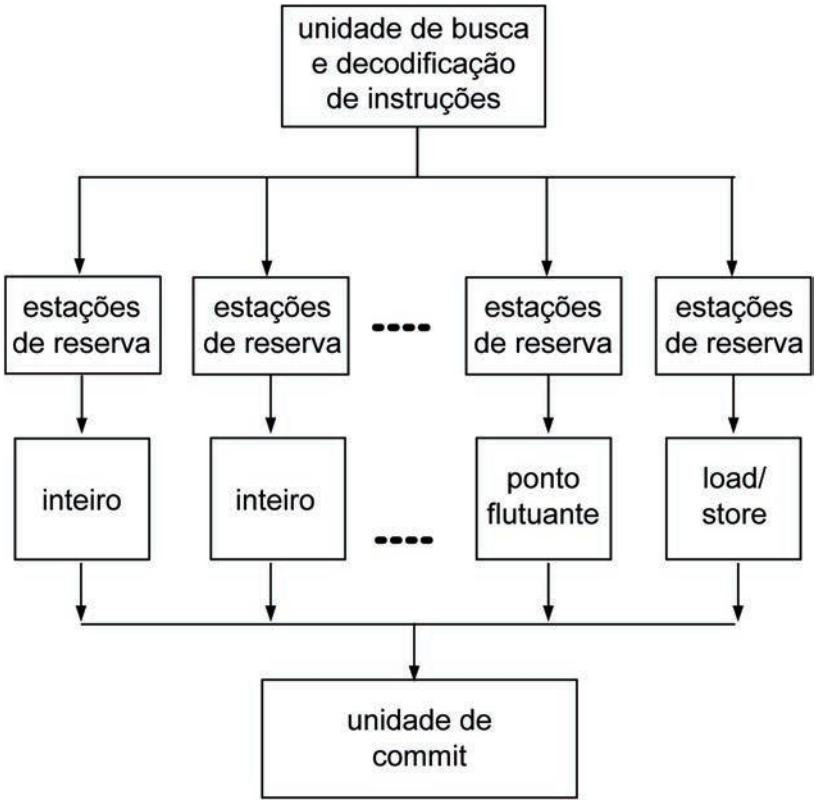


Figura 112 Diagrama de fluxo de execução de instruções com escalação dinâmica.

Na parte superior é verificada a unidade de busca e decodificação de instruções. As instruções decodificadas são despachadas para as unidades funcionais por meio das estações de reserva, onde as instruções aguardam os operandos ficarem disponíveis, momento no qual as instruções começam a execução.

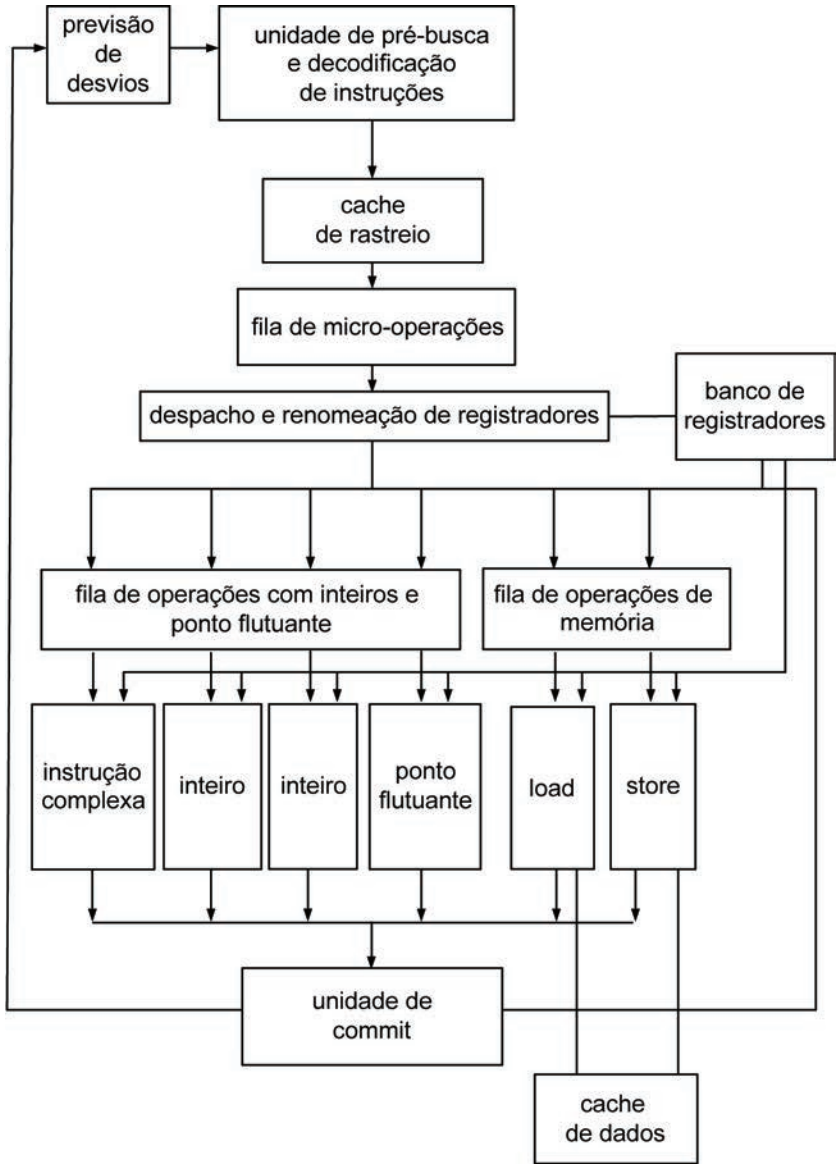


Figura 113 Arquitetura de um Pentium 4.

Após a execução, os resultados são disponibilizados para as estações de reserva que estão no aguardo dos mesmos, e uma unidade denominada *commit* determina o comprometimento desses resultados. O comprometimento implica em aceitar as instruções executadas por especulação. Um exemplo de uma arquitetura com uma característica semelhante à da Figura 112 é a do Pentium 4, mostrada na Figura 113.

7.5.3 Arquiteturas VLIW

VLIW (*Very Long Instruction Word*) é uma alternativa de arquitetura na qual as operações podem ser paralelizadas. O princípio dessa arquitetura é a detecção das operações paralelizáveis por *software* (compilador) seguida da disposição

dessas operações em formato de instruções longas. Estas contêm todas as operações que podem ser executadas num mesmo ciclo, de onde se obtém o nome *Very Long Instruction Word*, VLIW. Uma instrução VLIW é executada usando várias unidades funcionais, uma para cada operação. Como exemplo de operações paralelas em VLIW temos: duas operações inteiras, duas operações ponto flutuante, duas referências à memória e um desvio. Cada campo da instrução, correspondente a uma operação, pode ter tipicamente de *16* a *24 bits*. Para uma arquitetura com sete campos têm-se 7 x 16, ou *112 bits,* a 7 x 24, ou *168 bits,* para uma instrução VLIW. Um exemplo desse tipo de arquitetura é o Intel/HP-IA-64 (ITANIUM), *Explicitly Parallel Instruction Computer* (EPIC).

7.5.4 Multiprocessadores

Apesar dos avanços no desempenho dos processadores, a capacidade de processamento de um único processador é limitada. Uma possibilidade para melhorar o desempenho é usar, para a execução de uma aplicação, vários processadores em paralelo. Esse tipo de arquitetura é denominado multiprocessador.

Um multiprocessador pode ser de memória compartilhada centralizada (Figura 114) ou de memória distribuída (Figura 115).

No multiprocessador de memória centralizada, observamos, na parte superior, os processadores que se interligam a um barramento único e, na parte inferior, a memória e E/S (Entrada/Saída). Denominamos barramento uma forma de interconexão, na qual certo número de dispositivos se comunica diretamente compartilhando um mesmo circuito. Assim, um processador comunica-se com uma memória ou E/S pelo mesmo barramento. Com os processadores compartilhando uma mesma memória, o tempo de acesso à memória se torna lento. Uma forma de evitar esse problema é incluir o *cache* entre o processador e a memória, como mostrado no diagrama. O *cache* é uma memória rápida que, geralmente, contém os dados mais usados pelo processador, e que evita o acesso à memória principal tanto quanto possível. O estudo mais detalhado do *cache* será feito na unidade seguinte.

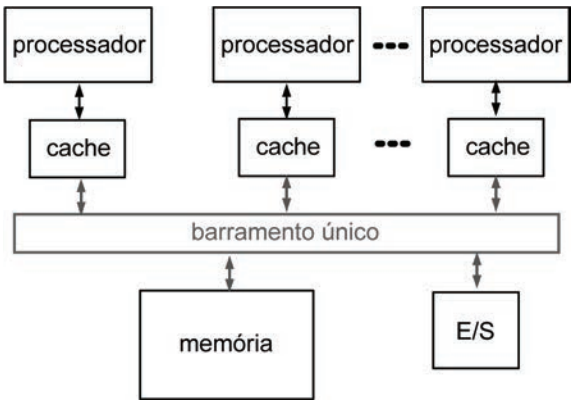


Figura 114 Multiprocessador de memória compartilhada centralizada.

No multiprocessador de memória distribuída, cada nó ou processador possui uma memória própria, conforme Figura 115. Assim, cada nó processa os dados contidos na sua memória e, quando existe a necessidade de comunicação com outro processador, ela é feita a partir de um sistema de rede de interconexão. Esta, dependendo da dimensão do multiprocessador, apresenta topologias variadas.

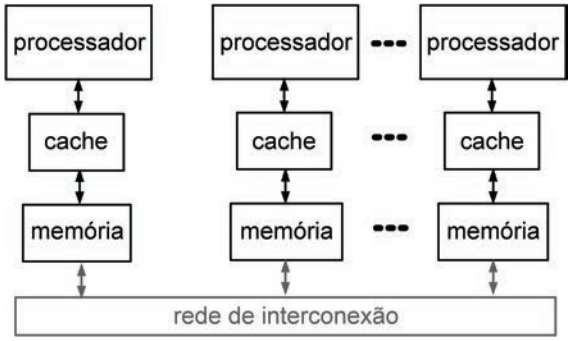


Figura 115 Multiprocessador de memória distribuída.

7.6 Considerações finais

Na presente unidade foi visto o processamento *pipeline* de instruções e os problemas típicos que surgem nessas implementações. Foram descritas rapidamente outras arquiteturas, como superescalares, VLIW e multiprocessadores. Evidentemente, o aprofundamento no estudo dessas arquiteturas fica fora do escopo da unidade, porém, fica explicitada a importância das arquiteturas alternativas como a de multiprocessamento, uma vez que as tendências caminham no sentido do desenvolvimento de arquiteturas como a *multi-core*, que implica na implementação de multiprocessamento interno aos *chips*, usando técnicas de redes (*network-on-chip*).

7.7 Estudos complementares

Para complementar os estudos sobre o *pipeline* e outras arquiteturas, os leitores podem se reportar ao quarto capítulo, *The Processor*, do livro de Hennessy & Patterson (2008), ou ao sexto capítulo, *Melhorando o Desempenho com Pipelining*, da versão traduzida por Daniel Vieira (HENNESSY & PATTERSON, 2005). Outra sugestão é o oitavo capítulo, *Arquiteturas de computadores paralelos*, do livro de Tanenbaum (2007). Uma descrição exaustiva sobre as arquiteturas modernas é também encontrada no livro de Hennessy & Patterson (2007). Sugere-se também a leitura do livro de Dantas (2005).

7.8 Exercícios

1. Considerando o tempo de 1 ns para a leitura de registradores, 1 ns para a escrita de registrador, 2 ns para busca de instrução, 2 ns para ULA e 2 ns para acesso a memória de dados, calcular:
   a) O tempo necessário para executar um instrução tipo R no MIPS monociclo, lembrando-se da existência da instrução lw;
   b) O tempo necessário para executar uma instrução tipo R no MIPS multiciclo;
   c) O tempo necessário para executar uma instrução tipo R no MIPS pipeline.

   Observação: desconsiderar o tempo de cálculo de endereço da próxima instrução e a latência em latches usados em MIPS multiciclo e pipeline.

2. Seja um computador executando as instruções em pipeline de 6 estágios, de forma ideal.
   a) Calcular o *speedup* em relação a um computador não-pipeline executando as mesmas instruções. Considerar o número de instruções n = 20.
   b) Calcular qual seria o *speedup* quando o número de instruções é bem maior do que o número de estágios? Verificar isso para n = 2000.

3. Quais são os latches usados no MIPS pipeline, quais são os sinais de controle neles presentes, e quais são os sinais que são usados no estágio imediatamente seguinte?

4. Dada a seqüência de instruções:

```
lw    $5,  0($6)
add   $2,  $3,  $4
sw    $6,  0($6)
and   $2,  $5,  $7
or    $4,  $5,  $6
```

a) Em que estágios estão as instruções quando a última instrução está no estágio IF (busca de instrução)?

b) Quais são os sinais de controle presentes nos latches entre os estágios do pipeline quando a última instrução está no estágio IF?

c) Dos sinais listados no item b, quais são usados imediatamente nesse mesmo ciclo, ou seja, no estágio subsequente?

d) Quais são os dados contidos no latch MEM/WB nesse mesmo ciclo?

5. Para a seqüência de instruções seguintes, verificar as dependências/conflitos e indicar as soluções para os mesmos:

```
sub   $2,   $4,   $6
add   $5,   $2,   $4
or    $7,   $2,   $8
lw    $9,   10    ($7)
sw    $7,   10    ($5)
and   $3,   $5,   $7
```

6. Definir um processador superescalar, e como poderia construir um MIPS superescalar de grau 2.

UNIDADE 8

Hierarquia de memória

8.1 Primeiras palavras

A Unidade 8 refere-se ao estudo do sistema hierárquico de memória, incluindo os conceitos de memória *cache* e memória virtual.

8.2 Problematizando o tema

As memórias estáticas SRAM (*Static Random Access Memory*) são constituídas de *bits* armazenados em pares de portas inversoras, conforme se pode observar na Figura 116. Nota-se que o inversor 1 está com a saída 1 e o inversor 2, com a saída 0. Como a saída do inversor 2, com valor 0, está ligada à entrada do inversor 1, com o mesmo valor, os valores do par de portas inversoras é estável, e podemos dizer que o inversor 2 está no estado 0.

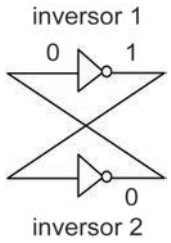


Figura 116 Par de portas inversoras interligadas, formando um *bit* de memória.

O mesmo par de portas inversoras do exemplo anterior pode estar com o inversor 2 no estado 1, como na Figura 117. Nesse caso o par de portas inversoras é também estável. A partir dos dois exemplos mostramos que o par de portas inversoras é capaz de armazenar um *bit*, que pode estar no estado 0 ou 1.

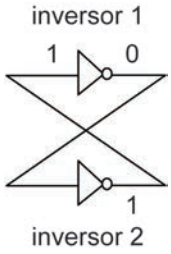


Figura 117 Par de portas inversoras com o inversor 2 no estado 1.

Um circuito de memória estática SRAM é mais rápido que os circuitos de memórias dinâmicas DRAM (*Dynamic Random Access Memory*), cujos *bits* são armazenados como cargas em capacitores. Esses capacitores se descarregam com o tempo, portanto, o conteúdo das memórias DRAM deve ser reavivado (*refreshed*) em intervalos de tempo de alguns milissegundos. O circuito de memória DRAM

é mais lento (5 a 10 vezes) que o circuito de memória SRAM. Porém um circuito DRAM é menor e, portanto, de custo, por *bit*, menor que um circuito SRAM.

Resumindo, o tempo de acesso de SRAM varia em torno de *2* a *25 ns* e o custo por *Mbyte* varia em torno de 100 reais. O tempo de acesso de DRAM varia em torno de *60* a *120 ns* e o custo varia em torno de 1 real por *Mbyte*. Por outro lado, um disco magnético que também é usado para armazenar informações apresenta um tempo de acesso de *10* a *20 milhões* de nanossegundos e um custo de alguns centavos de reais por *Mbyte*.

Os processadores precisam de memórias de grande capacidade (quantidade de *bits*) e rápidas. Para que o quesito capacidade seja atendida, o processador deve usar os discos magnéticos, porém, caso almeje rapidez no acesso, deve usar as memórias SRAM. Para conciliar o uso dos tipos de memória acima, é proposta a construção de um sistema de hierarquia de memória, onde várias tecnologias de memória convivem para servir o processador.

8.3 Sistema hierárquico de memória

Um sistema hierárquico de memória consiste em vários tipos de memória, cada tipo num nível de hierarquia, numerados de 1 a *n*. O nível 1, que fica mais próximo da CPU, tem menor capacidade, porém maior velocidade. O nível *n*, que fica mais longe da CPU, deve ter maior capacidade, porém menor velocidade. Na Figura 118 temos o diagrama de um sistema hierárquico de memória. Nesse diagrama, à medida que o nível de hierarquia desce, a capacidade aumenta e a velocidade diminui.

A memória é referenciada pela CPU, em busca de um conteúdo; este está sempre presente por inteiro na memória de nível *n*. Nos níveis superiores, estão presentes cópias de partes do conteúdo.

A referência à memória começa sempre pelo nível superior, mais próximo da CPU, nível 1. Se a memória de nível 1 consiste no conteúdo referenciado, a referência termina. Caso um nível superior não contenha o dado referenciado, um nível imediatamente inferior deverá ser consultado. Esse processo termina quando a consulta for satisfeita por um dos níveis da hierarquia. Após a consulta ser satisfeita, a cópia da porção da memória referenciada, denominada bloco, é transferida para todos os níveis superiores. Um bloco, normalmente, contém mais do que uma palavra referenciada pela CPU, e o seu tamanho aumenta à medida que o nível hierárquico da memória se torna inferior. Como as memórias de nível superior têm capacidade menor, mantêm apenas algumas cópias de blocos. Assim, quando existe a transferência de bloco e não há mais espaço num nível superior, um bloco desse nível deve ser substituído pelo novo bloco.

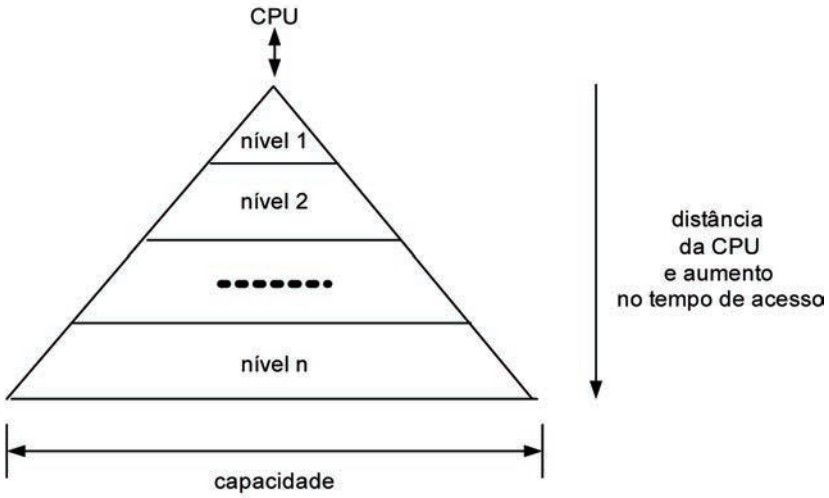


Figura 118 Diagrama de um sistema hierárquico de memória.

Fonte: adaptada de Hennessy & Patterson (2008).

Para que um sistema de hierarquia de memória seja eficiente, a CPU deve fazer a maioria de suas referências nas memórias de nível superior. Uma possibilidade de tornar eficiente o sistema de hierarquia de memória é a CPU referenciar, repetidamente, os mesmos blocos já carregados nas memórias de nível superior. O princípio da localidade garante a eficiência do sistema hierárquico de memória. Segundo esse princípio, a CPU faz referências aos mesmos blocos por duas razões, a primeira é a localidade temporal, que consiste na propriedade da CPU de fazer referências repetidas à mesma localidade, e a segunda é a localidade espacial, em que as referências subsequentes da CPU são os endereços próximos ou mesmo subsequentes da memória.

A localidade espacial existe porque os programas são sequenciais, e as instruções sequenciais ficam armazenadas em posições subsequentes na memória. Além disso, a manipulação de matrizes, vetores e outras estruturas de dados implica em referências a endereços subsequentes. Devemos observar que para poder usufruir da localidade espacial, um bloco deve conter mais que uma palavra de memória referenciada pela CPU. Assim, quando a CPU faz uma referência, um bloco contendo várias palavras é carregado na memória de nível superior, e existe a probabilidade de que outra palavra contida no mesmo bloco seja referenciada subsequentemente.

Por outro lado, a localidade temporal existe, por exemplo, pela implementação de repetições (*loops*), sub-rotinas ou outros controles que acabam provocando a repetição de um mesmo código.

Resumindo, na hierarquia de memória há os níveis superior e inferior. Um nível superior contém cópias de dados contidos num nível inferior. Sempre que a CPU necessita de um dado, procura primeiro no nível superior mais próximo. Um

bloco é a quantidade de dados transferida de um nível inferior para o superior. O acerto (*hit*) consiste em um dado requisitado estar no nível superior. A falta (*miss*) consiste em o dado requisitado não estar no nível superior.

8.3.1 Memória *cache*

Cache (nível superior) é uma memória rápida, normalmente implementada em SRAM, que contém cópias dos dados da memória principal (nível inferior), implementada em DRAM. Como a memória principal está num nível inferior em relação ao *cache*, contém mais blocos que este pode guardar. Assim, diferentes blocos da memória compartilham posições (*slots*) no *cache*. Surgem então as questões: como sabemos se um dado item está no *cache*? E se estiver, como encontrá-lo? Isso depende do tipo de mapeamento da memória principal para o *cache*, que pode ser: mapeamento direto, mapeamento associativo e mapeamento associativo por conjunto.

8.3.2 *Cache* em mapeamento direto

Para a carga do *cache* com as palavras mais referenciadas pela CPU, a memória principal é dividida logicamente em certo número de blocos de tamanho fixo. Denominamos módulo um conjunto de blocos cujo tamanho coincide com o tamanho do *cache*. Assim podemos dizer, também, que a memória é dividida em certo número de módulos de tamanho fixo. O *cache* é, também, dividido em *slots* de mesmo tamanho dos blocos. Cada bloco subsequente da memória principal é carregado num *slot* subsequente no *cache* quando referenciado pela CPU. Assim, se carregarmos blocos subsequentes da memória principal no *cache*, eles vão ocupando *slots* subsequentes, até ficar totalmente carregado. Para carregar os blocos subsequentes ao módulo anterior, os *slots* começam novamente, a partir do primeiro *slot* do *cache*. Evidentemente, o conteúdo anterior contido no *cache* é substituído. A numeração dos módulos consecutivos é usada como rótulos, *tags*, para identificação dos blocos contidos nos *slots* do *cache*. No mapeamento direto, o número do bloco de memória, dentro de um módulo, coincide com o número do *slot* do *cache*. Para calcular o *slot* a ser usado, deve-se dividir o endereço de memória pelo número de *slots*. O resto da divisão é o número do *slot*. Exemplo: num *cache* de *8 slots*, o endereço 9 é copiado no *slot 1*, pois:

9 mod 8 = 1 (a operação *mod* indica o resto da divisão de 9 por 8).

A Figura 119 mostra um exemplo de mapeamento direto para um *cache* de *8 slots*, numerados de 0 a 7, na parte superior do desenho. Assim, o bloco de número 1 da memória desenhada horizontalmente, na parte inferior, é mapeado no *slot* de número *1*. O bloco de número 9 é também mapeado no *slot* de número *1*, bem como os blocos de número 17, e assim por diante.

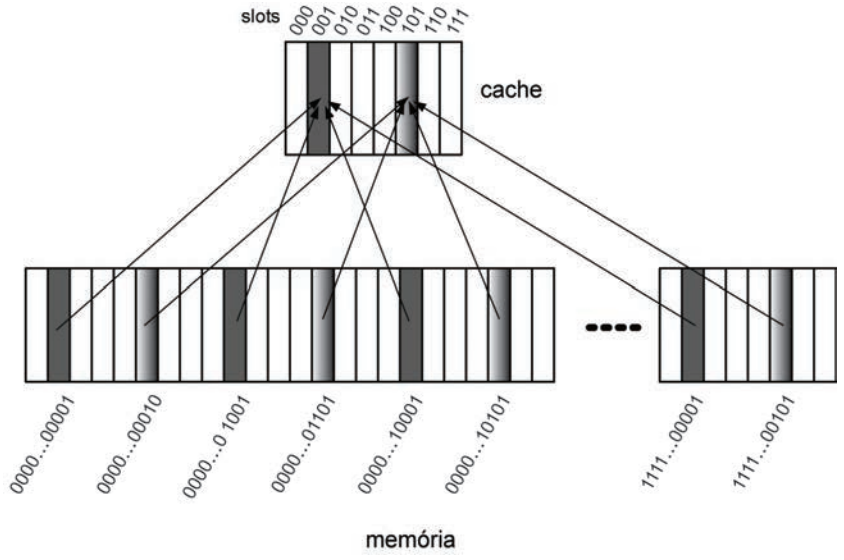


Figura 119 Diagrama ilustrativo de um mapeamento direto num *cache* de *8 slots*.

Fonte: adaptada de Hennessy & Patterson (2008).

Assim, cada bloco de memória tem apenas um *slot* no *cache* para o mapeamento. Para verificar se a cópia do bloco está no *cache* é preciso verificar o conteúdo do *slot* respectivo. Porém, como vários blocos de memória compartilham um mesmo *slot* do *cache*, o problema é saber qual bloco de memória está no *slot*.

A Figura 120 mostra um *cache* de *slots* de *32 bits*, no qual o endereço fornecido pela CPU aponta para o *byte* menos significativo da palavra. A referência feita pela CPU é usada na consulta do *cache*, para ver se o bloco está no *slot* correspondente. Nesse exemplo, o bloco é constituído de apenas uma palavra de *32 bits*.

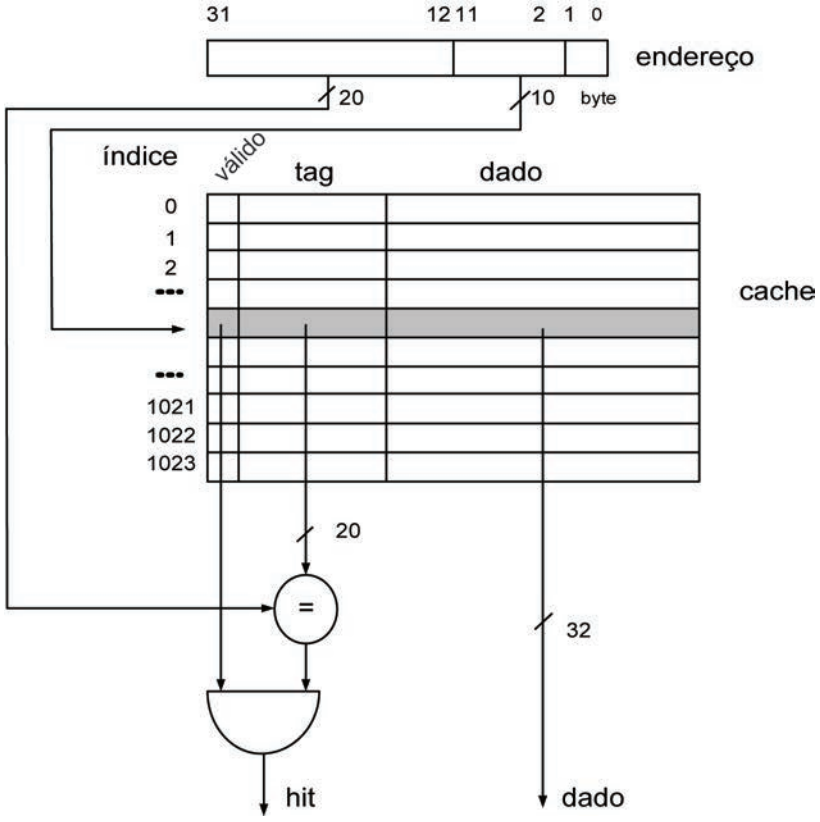


Figura 120 Diagrama da organização do *cache* em mapeamento direto.

Fonte: adaptada de Hennessy & Patterson (2008).

Na parte superior da Figura 120, é visto o endereço apresentado pela CPU para o sistema de memória. Os primeiros dois *bits* à direita são referentes ao *byte* dentro da palavra. Os *10 bits* seguintes são usados para apontar para um dos *slots* do *cache*, portanto o *cache* é composto de *1024 slots*. Os *20 bits* seguintes são usados para indicar o número do módulo. No *slot* apontado há três campos: *bit* de válido, *tag* e dado. O *bit* de válido pode ser 1 quando algum bloco válido existe no *slot* respectivo, e 0, caso contrário. O campo *tag* é a informação do módulo ao qual pertence o bloco, presente no campo de dado. Esse campo deve ser comparado com os *20 bits* do endereço apresentado pela CPU, que indicam o número do módulo referenciado. Se coincidirem os dois conteúdos e o *bit* válido for igual a 1, significa acerto no *cache* (*hit*), ou seja, significa que o bloco desejado é o que está carregado no campo de dado.

A Figura 121 mostra um *cache* organizado em mapeamento direto, porém, explorando a localidade espacial. Para que a localidade espacial seja explorada, um bloco de memória deve conter mais de uma palavra referenciada pela CPU. Uma referência feita pela CPU apresenta alta probabilidade de ser encontrada no mesmo *slot* da referência anterior. Na Figura 121, um *slot* indicado pelos *12 bits* de índice do endereço contém quatro palavras. Cada uma delas é selecionada pelo campo de *2 bits* (*bits 2* e *3*), que é ligado a um multiplexador.

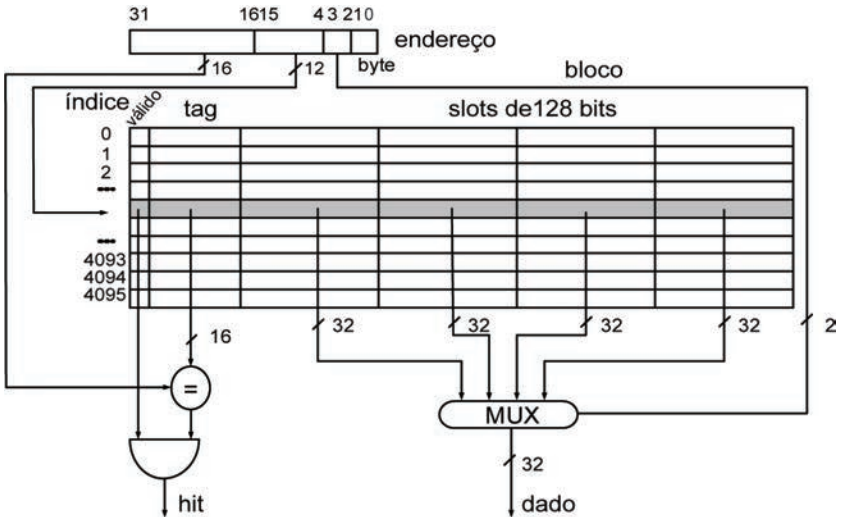


Figura 121 *Cache* organizado em mapeamento direto, com 4 palavras por bloco.

Fonte: adaptada de Hennessy & Patterson (2008).

O que se deseja numa referência à memória é o acerto (*hit*), tanto na leitura como na escrita. Quando acontece a falta (*miss*) de leitura, a CPU para enquanto se faz a busca de um bloco na memória e carrega o *cache*. Quando acontece um acerto de escrita, existem duas alternativas: 1) escrever no *cache* e na memória (abordagem denominada *write-through*); 2) escrever somente no *cache* (a escrita na memória, *write-back*, é feita mais tarde, quando ocorre uma substituição). Quando ocorre uma falta na escrita significa que o bloco correspondente deve ser carregado no *cache*, e, depois, a palavra deve ser escrita usando uma das duas alternativas descritas.

8.3.3 Interconexão do *cache* com a memória principal

A interconexão do *cache* com a memória principal é importante para que um sistema de memória hierárquico seja eficiente.

A Figura 122 mostra um sistema de interconexão da memória principal com o *cache*, num barramento do tamanho de uma palavra. É possível transferir uma única palavra por vez da memória principal para o *cache*, o que corresponde à implementação de *cache*, como na Figura 120. Nesse caso apenas a localidade de referência temporal pode ser explorada, pois o tamanho de um bloco deve ser no máximo de uma palavra.

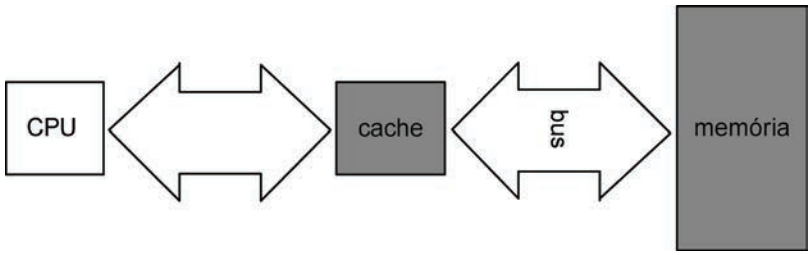


Figura 122 Organização de interconexão memória/*cache* num barramento de uma palavra.

A Figura 123 mostra um sistema de interconexão de quatro palavras, e a transferência para o *cache* pode ser feita em blocos de quatro palavras, o que corresponde à implementação de *cache*, como na Figura 121. Ao encontrar uma cópia do bloco no *cache*, é selecionada uma das palavras do bloco para ser entregue à CPU, no caso de leitura. Essa organização permite a exploração da localidade de referência espacial, além da temporal.

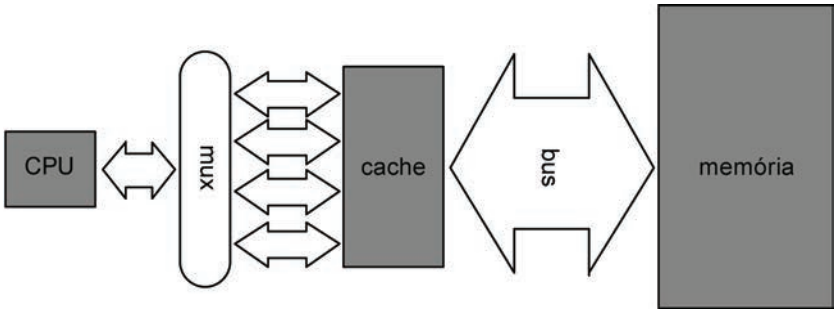


Figura 123 Organização de interconexão com um barramento largo.

A Figura 124 mostra um sistema de memória implementada em vários bancos, cada um contendo uma das palavras do bloco. Assim, para ler um bloco inteiro, todos os bancos devem ser lidos simultaneamente. Esse tipo de implementação de memória é conhecido como intercalação (*interleaving*). Após a leitura, as palavras, uma por vez, são transferidas para o *cache*. Apesar de essa transferência ser sequencial, a latência total é relativamente pequena, pois sua maior parte está no tempo de acesso à memória (que no caso é feito em paralelo), e não no tempo de transferência.

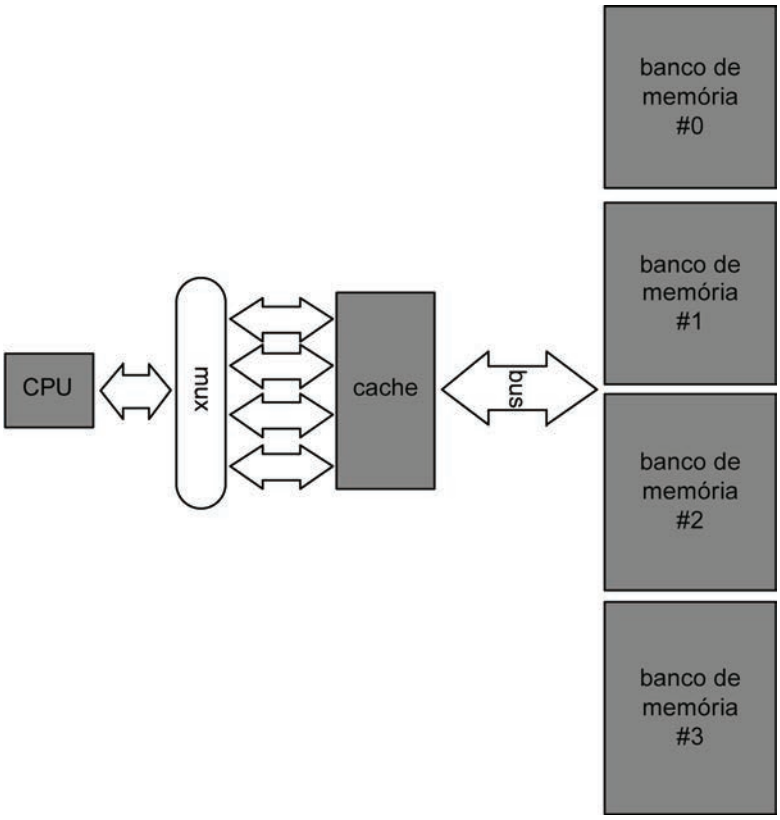


Figura 124 Organização de memória entrelaçada (*interleaving*).

8.3.4 Desempenho do *cache* em mapeamento direto

A Figura 125 mostra um diagrama de taxa de falta no *cache*, em função dos tamanhos do bloco e do *cache*. Nota-se que para tamanho de *4 bytes* por bloco, no início da curva, a taxa de falta é maior que para outras condições. Isso ocorre porque *4 bytes* significam apenas uma palavra de *32 bits.* Como visto, um bloco de apenas uma palavra não explora a localidade de referência espacial num computador de *32 bits*. Quando o tamanho do *cache* é pequeno, de *1 Kbyte*, mesmo a localidade de referência temporal não pode ser explorada, pois o *cache* tem possibilidade de carregar apenas 256 palavras, e os *slots* devem ser compartilhados. Para tamanho maior de *cache*, a localidade de referência temporal é mais bem explorada, diminuindo a taxa de falta.

A taxa de falta diminui à medida que o tamanho de bloco aumenta, até *64 bytes*. Após esse tamanho, ela tende a aumentar. No caso de *256 bytes*, o bloco contém 64 palavras. No *cache* de tamanho igual a 256 palavras há apenas *4 slots* de 64 palavras. Isso implica em pouca opção para explorar a localidade temporal, do que decorre o aumento da taxa de falta. Para os *caches* maiores, o tamanho de bloco de 64 palavras não é tão crítico, por isso, mantém a mesma taxa de falta relativa ao bloco de *64 bytes*, ou 16 palavras.

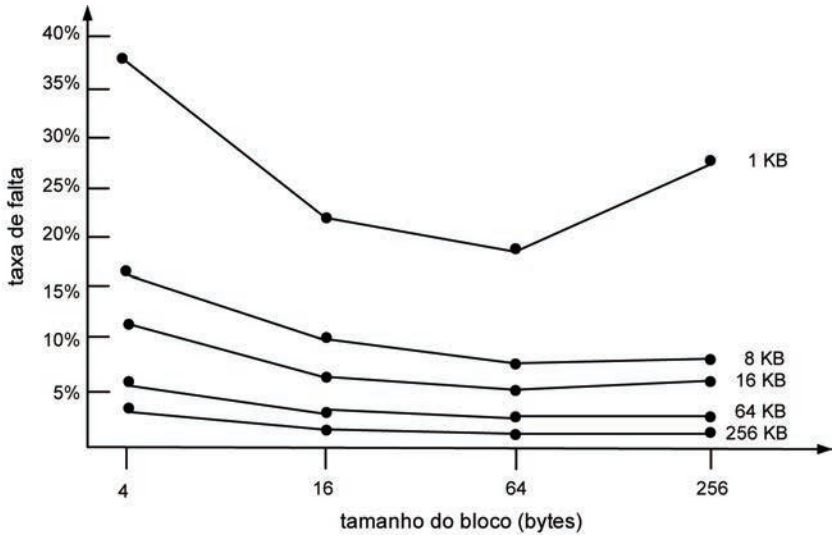


Figura 125 Diagrama de taxa de falta (*miss rate*) em função do tamanho de bloco e tamanho do *cache*.

Fonte: adaptada de Hennessy & Patterson (2008).

Tabela 17 Taxa de falta em *benchmark* SPEC.

Programa	Tamanho do bloco em palavras	Taxa de falta em instruções	Taxa de falta em dados	Taxa de falta combinada
Gcc	1	6,1%	2,1%	5,4%
	4	2,0%	1,7%	1,9%
Spice	1	1,2%	1,3%	1,2%
	4	0,3%	0,6%	0,4%

Fonte: adaptada de Hennessy & Patterson (2008).

A Tabela 17 mostra o resultado da taxa de falta nos *caches* quando estes são divididos em: *cache* de instruções e *cache* de dados. Nota-se que, quando o tamanho do bloco é de quatro palavras, a taxa de falta diminui drasticamente para o *cache* de instruções. Isso significa que as instruções se enquadram na localidade de referência espacial, pelo fato de serem sequenciais.

Podemos equacionar o tempo de execução como:

tempo de execução = (*ciclos de execução* + *ciclos de parada*) x *tempo de ciclo*

em que

ciclos de parada = *número de instruções* x *taxa de falta* x *penalidade*.

Para melhorar o desempenho do computador, existem duas formas: 1) diminuir a taxa de falta e 2) diminuir a penalidade.

8.3.5 *Cache* em mapeamento associativo

A Figura 126 mostra um diagrama simplificado de uma organização de *cache* em associativo por conjunto de 1 via, que coincide com o mapeamento direto. O que caracteriza essa organização é a existência de apenas um *slot* para um bloco da memória principal, tornando o mapeamento inflexível.

Uma alternativa de mapeamento que permite diminuir a taxa de falta é o mapeamento com associatividade maior do que 1 via. Essa associatividade permite o mapeamento de um bloco num conjunto de *slots*, do *cache*. Na Figura 127, temos o caso de um mapeamento associativo por conjunto de 2 vias, em que um índice do *cache* aponta para uma linha de dois *slots*. Um determinado bloco de memória tem um índice específico, como no mapeamento direto. Portanto, o bloco de memória deve ser mapeado numa única linha. A associatividade no caso de 2 vias significa que, naquela linha, a cópia do bloco pode estar em qualquer um dos dois *slots*. Portanto, dois blocos diferentes com mesmo índice podem compartilhar uma mesma linha. No caso do mapeamento direto, uma referência subsequente a esses dois blocos implica na substituição do primeiro para poder carregar o segundo. A Figura 128 mostra o caso de um mapeamento associativo por conjunto de 4 vias, em que um índice do *cache* aponta para uma linha de quatro *slots*. Finalmente, na Figura 129, temos o caso de apenas um índice no *cache*, fazendo com que todos os blocos de memória sejam copiados na mesma linha do *cache*. É o *cache* totalmente associativo.

Quanto maior a associatividade, mais complexo o circuito do *cache*, pois todos os *tags* devem ser comparados com o *tag* do endereço da CPU, uma vez que a cópia do bloco pode estar em qualquer *slot*.

Figura 126 Mapeamento direto.

Fonte: adaptada de Hennessy & Patterson (2008).

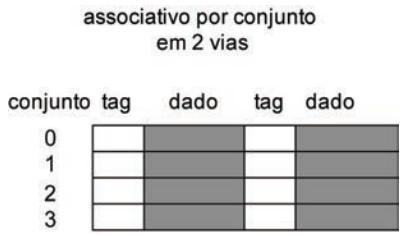


Figura 127 Mapeamento associativo de 2 vias.

Fonte: adaptada de Hennessy & Patterson (2008).

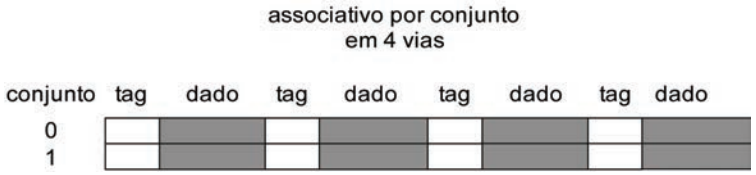


Figura 128 Mapeamento associativo por conjunto de 4 vias.

Fonte: adaptada de Hennessy & Patterson (2008).

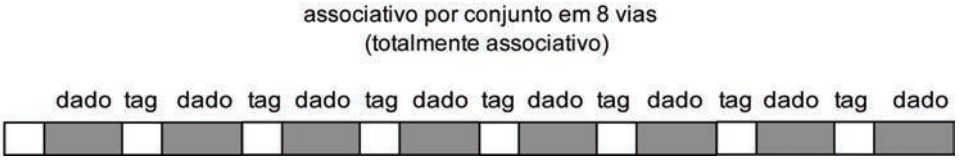


Figura 129 Mapamento totalmente associativo.

Fonte: adaptada de Hennessy & Patterson (2008).

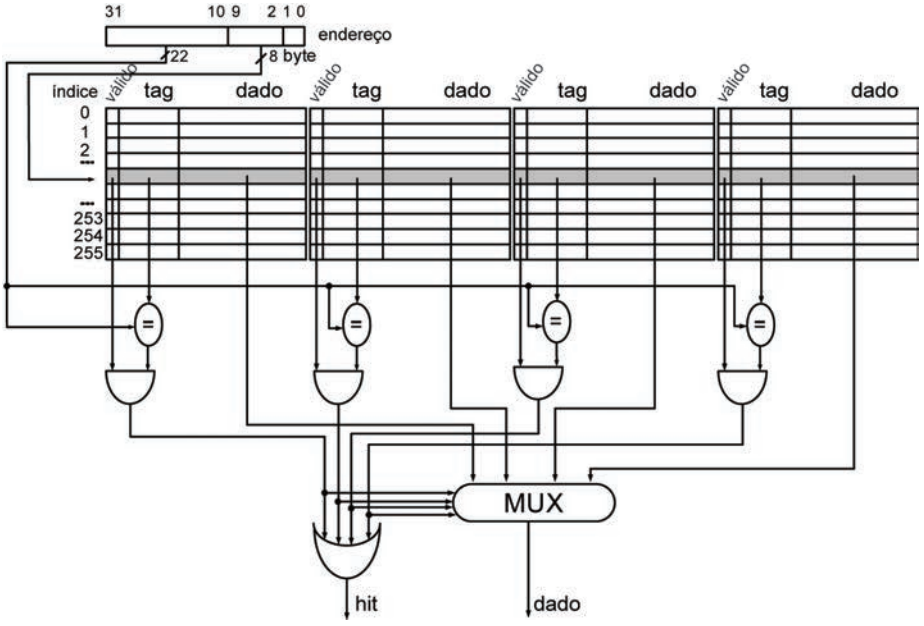


Figura 130 Diagrama de uma organização de *cache* associativo por conjunto de 4 vias.

Fonte: adaptada de Hennessy & Patterson (2008).

A Figura 130 mostra o diagrama de uma organização de *cache* associativo por conjunto de 4 vias para uma memória organizada em *bytes* e CPU referenciando palavras. Nota-se que existem quatro circuitos de comparação dos *tags* do *cache* com o *tag* de endereço da CPU. Considerando que cada circuito deve comparar *22 bits*, 4 implicam em comparar *88 bits*.

A Figura 131 mostra um diagrama de taxa de falta em função da associatividade e tamanho do *cache*. Nota-se que para *caches* pequenos, com o aumento da associatividade, a taxa de falta diminui. Isso se explica, pois, com o aumento da associatividade, um número maior de blocos pode compartilhar uma mesma linha, evitando a substituição. Quando o tamanho do *cache* aumenta, existe menos coincidência dos blocos num mesmo índice, portanto, o efeito da associatividade diminui. Isso explica o desempenho do *cache* de *128 Kbytes*, cuja taxa de falta não se altera com o aumento da associatividade.

Como vimos anteriormente, para melhorar o desempenho do computador, existem duas formas: 1) diminuir a taxa de falta e 2) diminuir a penalidade.

A segunda forma de melhorar o desempenho é, portanto, diminuir a penalidade. Isso implica em diminuir a latência de acesso à memória. Uma possibilidade é adicionar um segundo nível de *cache*. Geralmente o *cache* nível 1 fica no mesmo *chip* do processador. É possível usar SRAMs para adicionar outro *cache* acima da memória principal (DRAM). A penalidade diminui se os dados estão nesse *cache* nível 2.

Exemplo: Numa máquina a *500 MHz* com 5% de taxa de falta e acesso a DRAM de *200 ns*, adicionando *cache* nível 2 de tempo de acesso igual a *20 ns*, diminui a taxa de falta para 2%.

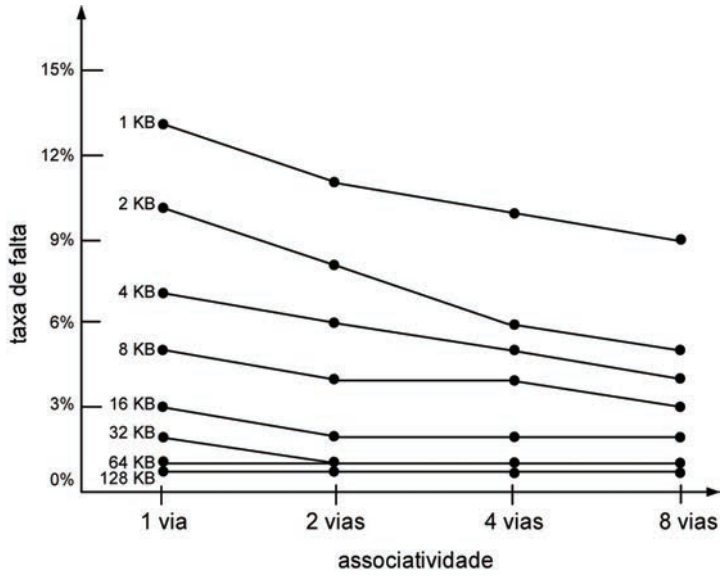


Figura 131 Diagrama de taxa de falta em função da associatividade e tamanho do *cache*.

Fonte: adaptada de Hennessy & Patterson (2008).

8.3.6 Memória virtual

Vimos que é desejável um sistema de memória de grande capacidade e velocidade, e que o *cache* é um nível de memória que resolve o problema da velocidade. Vimos, também, que o disco magnético é um nível de memória que tem uma capacidade grande. Um exemplo de sistema de memória virtual consiste em usar o disco magnético como o nível de memória inferior à memória DRAM (memória principal). Assim, cópias de porções de memória do disco são carregadas na memória DRAM, e quando referenciadas num sistema de memória virtual como cópias de blocos da memória DRAM, são carregadas no *cache*, num sistema de *cache*. Num sistema de memória virtual, a memória DRAM é denominada de memória física.

Esse tipo de sistema apresenta a ilusão de se ter uma memória física enorme, possibilitando a relocação e a proteção de memória. Nesse sistema a CPU manipula a memória com os endereços virtuais. A memória DRAM é chamada de memória física, pois o acesso à memória é feito praticamente na memória DRAM. A memória virtual é implementada no disco magnético, pois todo o conteúdo da memória virtual fica no disco. Quando a CPU faz referência à memória por meio do endereço virtual, o conteúdo pode estar na memória física, caso aquele endereço já tenha sido referenciado. Caso não esteja, o conteúdo deve ser lido a partir do disco. A Figura 132 mostra o mapeamento de endereços virtuais, para os endereços físicos ou para os endereços de disco.

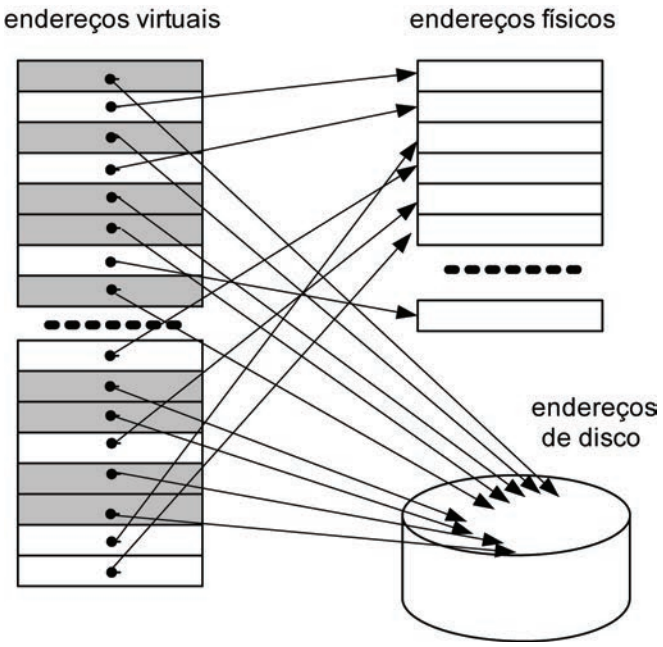


Figura 132 Diagrama básico de endereçamento de memória virtual.

Fonte: adaptada de Hennessy & Patterson (2008).

Num sistema de memória virtual, a porção de dados transferidos de uma vez do disco para a memória física é denominada página. Essa transferência tem uma latência grande devido ao mecanismo usado para a leitura do disco magnético, e tem a característica de transferência por setor. Portanto, uma página tem tamanho de múltiplos setores. Quando ocorre uma falta de página, significa que os dados não estão na memória física e devem ser recuperados do disco. Esse gerenciamento de falta de página é feito, geralmente, por *software*. A penalidade de falta é grande; portanto, para contornar essa situação, usam-se páginas grandes (por exemplo, *4 Kb*), o que é adequado, pois os discos fazem leitura por setores, ou múltiplos setores. É também importante reduzir as faltas de página.

A escrita é feita pela abordagem de *write-back*, em que a escrita no disco é feita somente quando uma página carregada na memória física deve ser substituída. O uso do *write-through* é muito dispendioso, uma vez que nessa abordagem, quando se escreve na memória física, também se escreve na memória virtual, no disco.

A Figura 133 mostra um diagrama de tradução do endereço virtual para o endereço físico. Os *bits* de *0* a *11* do endereço virtual são usados para o deslocamento dentro da página e, portanto, são os mesmos para o endereço virtual e endereço físico. Os *bits 12* a *31* indicam o número de página virtual. Como a memória física é menor que a memória virtual, o número de *slots* existentes na memória física é menor que o número de páginas virtuais. No exemplo a seguir, o número de página física é composto pelos *bits 12* a *29*.

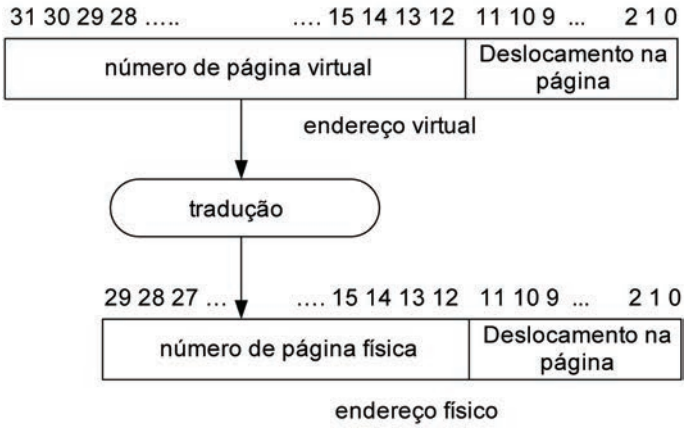


Figura 133 Tradução de endereço virtual para endereço físico.

Fonte: adaptada de Hennessy & Patterson (2008).

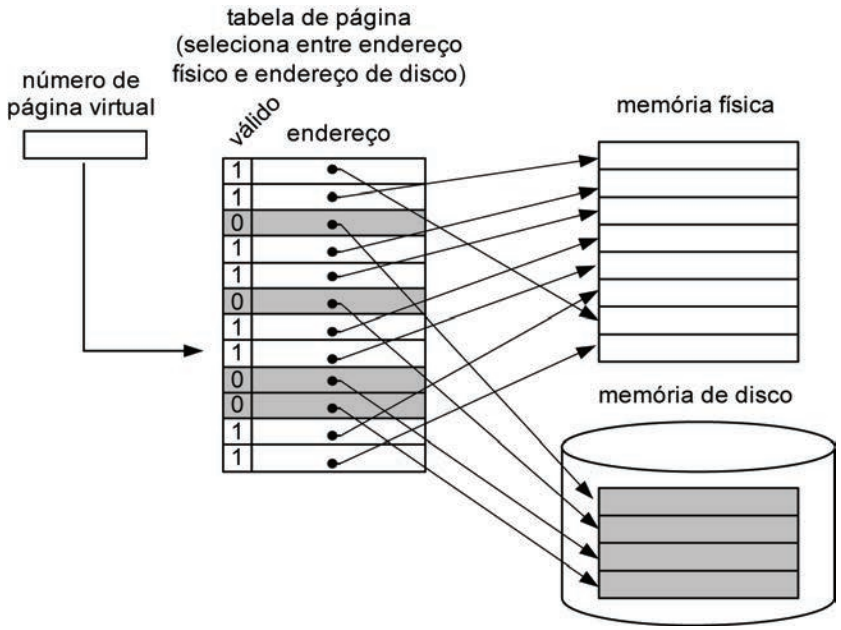


Figura 134 Diagrama de tradução de endereço usando tabela de página.

Fonte: adaptada de Hennessy & Patterson (2008).

Em tradução de endereço virtual para endereço físico é usada uma tabela de página, conforme Figura 134. Essa tabela contém dois campos para cada entrada. O primeiro é do *bit* válido, em que o valor 1 indica que a página virtual correspondente está presente na memória física. O valor 0 indica o contrário. O segundo campo contém o número da página física, caso o *bit* válido seja 1.

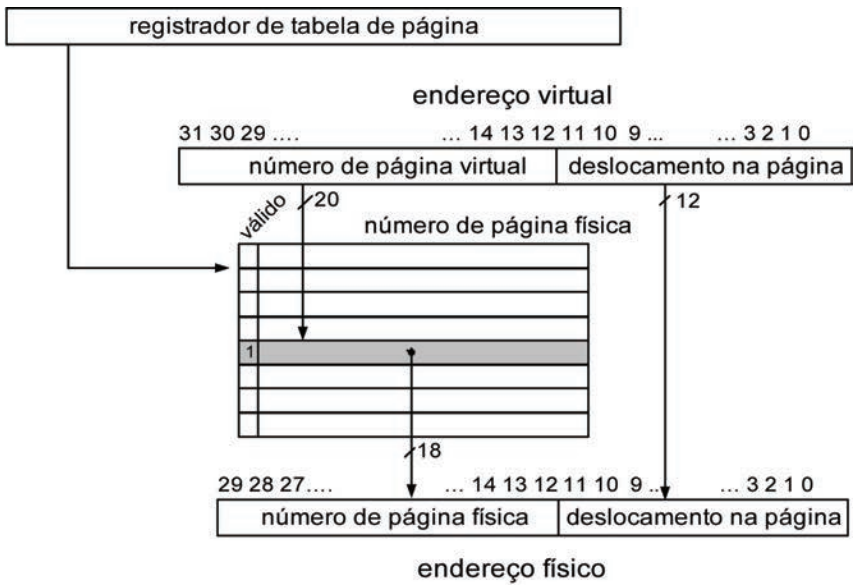


Figura 135 Diagrama de tradução de endereços usando uma tabela de página apontada por um registrador de tabela de página.

Fonte: adaptada de Hennessy & Patterson (2008).

A Figura 135 mostra um diagrama de tradução de número de página virtual para número de página física usando uma tabela de página, apontada por um registrador de tabela de página. Esse diagrama ilustra a possibilidade de existência de tabelas de página diferentes em ambientes multitarefas, por exemplo. Cada tabela seria usada por uma determinada tarefa.

Nota-se que uma tabela de página é grande para conter o número de todas as páginas virtuais e, portanto, deve ficar na memória física. Num sistema de memória virtual devemos consultar a tabela de página para verificar o número de página física, o que implica num acesso à memória. Após a consulta, a palavra referenciada deve ser lida ou escrita por meio de outro acesso à memória. Isso significa que uma referência à memória feita pela CPU resulta em dois acessos à memória, o que torna o sistema de memória lento. Uma forma de melhorar esse tempo é usar um *cache* específico para a tradução de página, TLB. Assim a maioria das consultas à tabela de página ocorre no TLB, diminuindo a latência de tradução.

A Figura 136 mostra o uso de um *cache* específico para tradução de página, denominado *Translation Look-aside Buffer*, TLB.

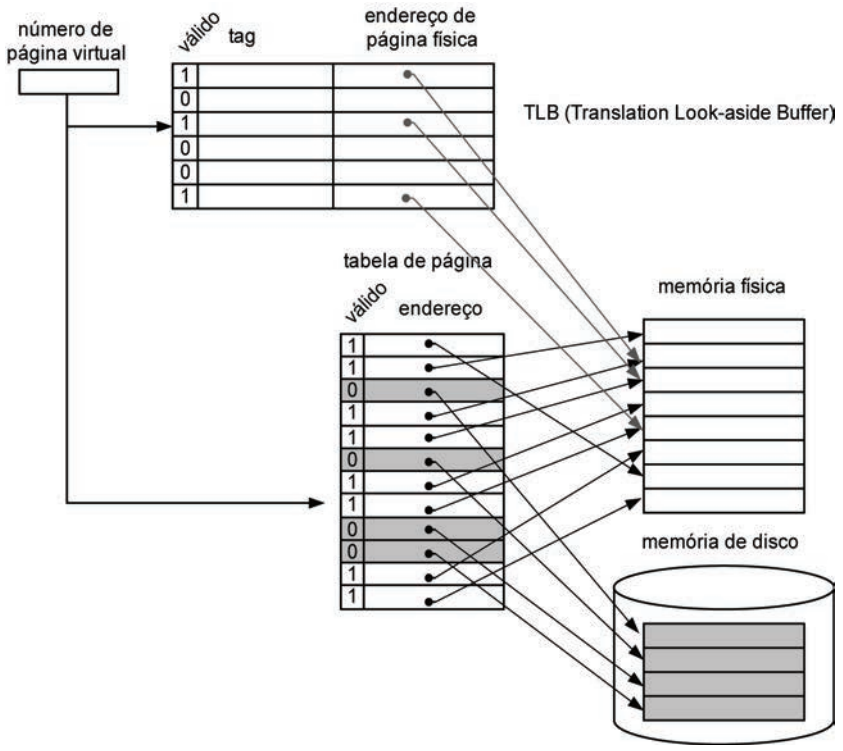


Figura 136 Aumentar a velocidade de tradução usando um *cache* de tabela de página (*Translation Look-aside Buffer*).

Fonte: adaptada de Hennessy & Patterson (2008).

A Figura 137 mostra um fluxograma de acesso à memória virtual. A primeira operação do fluxograma é o acesso ao TLB. Se o resultado der em acerto, o endereço físico é obtido e o dado pode ser acessado pela CPU, por leitura ou escrita. A forma de acesso ao endereço físico é normal, com o uso de *cache*. Se o acesso resultar em falta, a consulta à tabela de página deve continuar na memória física, por meio da exceção de falta no TLB.

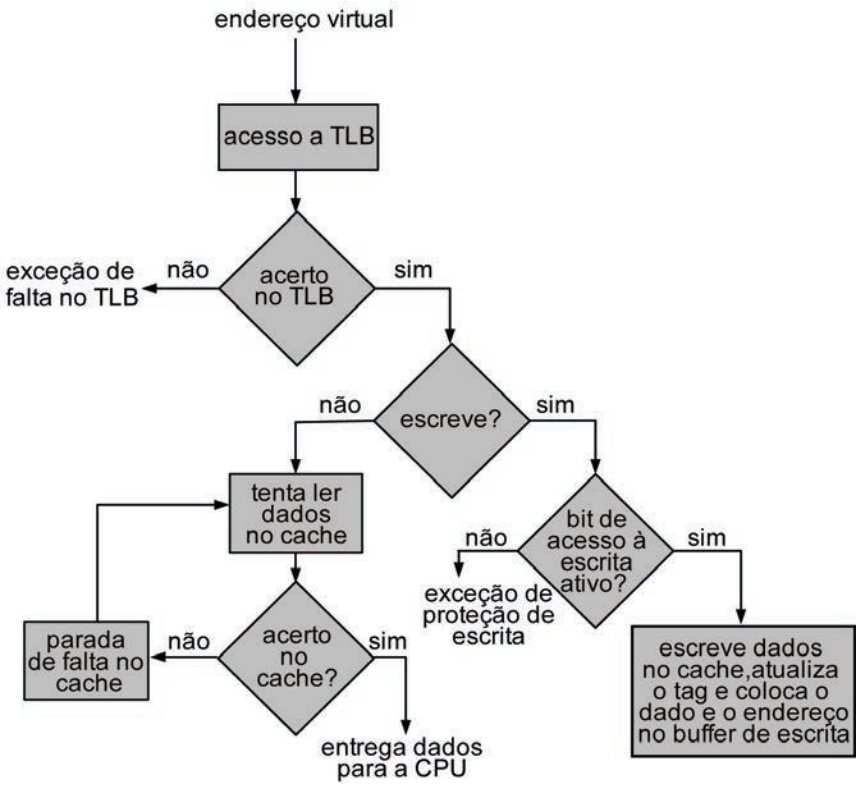


Figura 137 Fluxograma de acesso à memória virtual.

Fonte: adaptada de Hennessy & Patterson (2008).

8.4 Considerações finais

Nesta unidade foram vistos os princípios de um sistema hierárquico de memória que envolve várias tecnologias, desde memórias com alta velocidade de acesso, SRAMs, memórias semicondutoras mais lentas, porém de alta densidade, DRAMs e memórias de disco, também conhecidas como memórias secundárias. O princípio que rege o uso dessas memórias pelo processador é conhecido como princípio de localidade de referência, que pode ser: espacial e temporal. Esse princípio garante que as memórias SRAMs, mais rápidas, sejam mais referenciadas pelo processador que as memórias DRAMs, mais lentas. No caso de memória virtual, esse mesmo princípio garante que a memória física seja mais referenciada pelo processador que a memória em disco. Foram vistas as

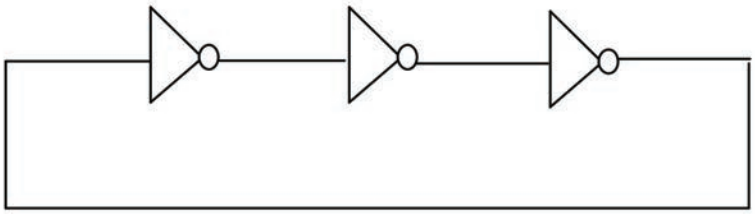
formas de implementação dos *cach*es, as vantagens e desvantagens de cada uma. Foram também apresentadas a memória virtual e a forma mais eficiente de referenciá-la, usando tabela de página na memória.

8.5 Estudos complementares

Para complementar os estudos sobre sistemas hierárquicos de memória, os leitores podem se reportar ao quinto capítulo, *Large and Fast: Exploiting Memory Hierarchy*, do livro de Hennessy & Patterson (2008), ou ao sétimo capítulo da versão traduzida por Daniel Vieira (HENNESSY & PATTERSON, 2005). Outra fonte é o quinto capítulo, *Projeto de hierarquia de memória*, do livro de Hennessy & Patterson (2007).

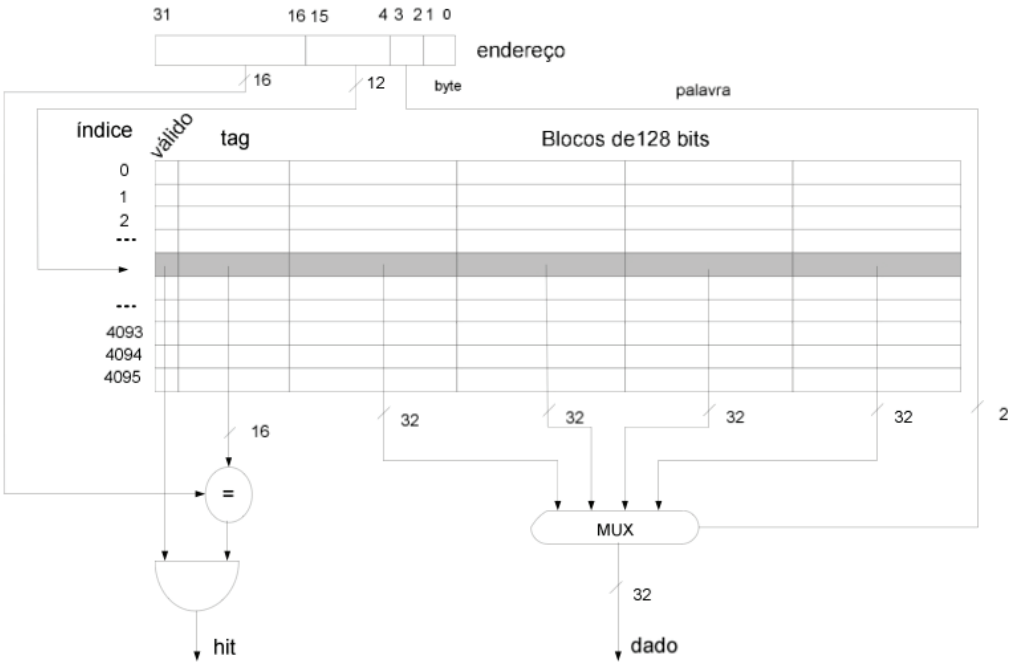
8.6 Exercícios

1. Uma interligação de 3 inversores, como na figura a seguir, consegue guardar um bit? Se a resposta é não, em que resulta esse circuito?



2. Por que as memórias estáticas são usadas se elas tem o custo por Mbyte cerca de 100 vezes maior do que o das memórias dinâmicas?
3. Por que as memórias dinâmicas são lentas e precisam ser reavivadas a um intervalo de alguns milisegundos?
4. O que significa princípio de localidade de referência? Quais tipos de localidade de referência existem?
5. Como funciona o cache de mapeamento direto?
6. Como funciona o cache de mapeamento associativo por conjunto?
7. O que significa um mapeamento totalmente associativo?
8. Para que serve o campo tag do endereço de memória? Como o campo tag é usado para verificar se um dado bloco está no cache?
9. O que acontece quando o processador faz uma referência de leitura a uma palavra cujo bloco de memória não é encontrado no cache? O que acontece se no slot do cache onde o bloco deve ser carregado existir um bloco válido?

10. O que acontece quando um processador faz uma referência de escrita a uma palavra?
11. O que se entende por intercalação de bancos de memória, ou interleaving?
12. Por que no sistema de memória virtual, a atualização do disco usa o critério de write-back?
13. Para que serve a tabela de páginas num sistema de memória virtual?
14. Veja o exemplo do cache em mapeamento direto da figura a seguir, responder:
    a) Qual o tamanho de um bloco em bytes?
    b) Onde as palavras de endereços 0, 4, 16 e 64 são mapeados?
    c) Onde o bloco de número 4096 é mapeado? qual é o tag nesse caso?
    d) Qual o endereço da primeira palavra do bloco de número 4096?

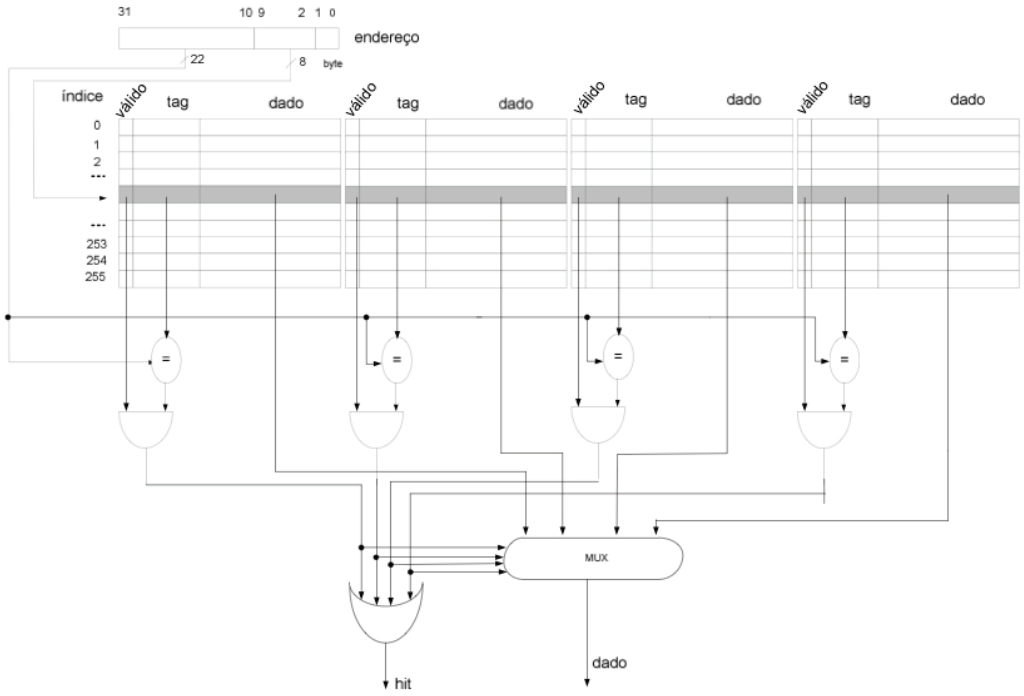


Considerando-se que é executado um programa em um computador MIPS, que lê quatro vezes em seguida, um vetor na memória, de comprimento 20, que se inicia no endereço 16, e que o cache em mapeamento direto da figura anterio é usado na hierarquia de memória, calcular a quantidade de acertos (hit) e de erros (miss), apenas referente ao acesso às palavras do vetor. Observação: considerar inicialmente, o cache totalmente vazio.

15. Ainda, usando o mesmo cache em mapeamento direto da figura anterior, executando um programa no computador MIPS, que lê duas vezes, alternadamente, dois vetores na memória, A e B, de mesmo comprimento, 20, sendo que o vetor A se inicia no endereço 16, e o vetor B, no endereço

16400, calcular a quantidade de acertos e erros, no que se refere ao acesso às palavras dos vetores A e B (Observação: 1) considerar inicialmente o cache totalmente vazio; 2) sequência de leitura dos vetores -> A, B, A, B); 3) o mapeamento do vetor B coincide com o do vetor A, pois o endereço 16400 é igual a 4 x 4096 +16.

16. Dado o cache associativo por conjunto da figura a seguir, responder:
    a) Onde se carregam os blocos de números 0, 256, 512 e 1024? quais são os respectivos tags?
    b) Como poderia melhorar esse cache para explorar a localidade espacial?
    c) Explicar como seria a seleção do dado quando um bloco fosse constituído de 2 palavras.



17. Usando o cache em mapeamento associativo por conjunto da figura anterior, executando um programa no computador MIPS, que lê duas vezes, alternadamente, dois vetores na memória, A e B, de mesmo comprimento, 20, sendo que o vetor A se inicia no endereço 16, e o vetor B, no endereço 16400, calcular a quantidade de acertos e erros, no que se refere ao acesso às palavras dos vetores A e B.

UNIDADE 9

Entradas e Saídas

9.1 Primeiras palavras

A presente unidade visa descrever os mecanismos usados para a entrada e saída de dados dos computadores, os dispositivos de armazenamento e outros periféricos.

9.2 Problematizando o tema

Conforme verificamos na Tabela 18, existe uma diversidade de dispositivos de entrada/saída (E/S) presentes nos computadores. Esses dispositivos se caracterizam pela diversidade de taxa de dados, que variam desde *0,0001 Mbits/s* ou *100 bits/s* do teclado, até *10.000,0000 Mbits/s*, ou *10 Gbits/s*, da rede/LAN (*Local Area Network*). O comportamento dos dispositivos pode ser de entrada (E), saída (S), ou ambos, entrada/saída (E/S). Os dispositivos de armazenamento normalmente usam acesso direto à memória, DMA (*Direct Memory Access*), que é uma forma especial de entrada/saída no computador.

Na presente unidade serão vistas as principais técnicas para a entrada e saída de dados do computador, usando esses dispositivos.

Tabela 18 Diversidade de dispositivos de entrada/saída.

Dispositivo	Comportamento	Taxa de dados Mbit/s
teclado	entrada	0,0001
mouse	entrada	0,0038
entrada voz	entrada	0,2640
entrada som	entrada	3,0000
Scanner	entrada	3,2000
saída voz	saída	0,2640
saída som	saída	8,0000
impressora laser	saída	3,2000
visor gráfico	saída	800,0000-8.000,0000
modem de cabo	entrada/saída	0,1280-6,0000
rede/LAN	entrada/saída	100,0000-10.000,0000
rede/sem fio LAN	entrada/saída	11,0000-54,0000
disco óptico	armazenamento	80,0000-220,0000
fita magnética	armazenamento	5,0000-120,0000
memória flash	armazenamento	32,0000-200,0000
disco magnético	armazenamento	800,0000-3.000,0000

Fonte: adaptada de Hennessy & Patterson (2008).

9.3 Técnicas de E/S do ponto de vista do processador

Existem basicamente três técnicas de entrada e saída de dados, do ponto de vista do computador: *polling*, interrupção e acesso direto à memória.

9.3.1 *Polling*

O *polling* é a forma mais simples para um dispositivo de E/S se comunicar com o processador. A Figura 138 mostra um fluxograma para a leitura de uma palavra de dados usando a técnica de *polling*.

Figura 138 Fluxograma de uma rotina de entrada de dados por *polling*.

O dispositivo de E/S coloca a informação num registrador de *status*, e o processador deve ler essa informação usando uma instrução. A desvantagem do *polling* é o tempo requerido pelo processador para a consulta repetida do *status*, pois os processadores são muito mais rápidos que os dispositivos de E/S, e o processador lê o registrador de *status* muitas vezes enquanto o dispositivo não completa uma operação de E/S.

9.3.2 Interrupção

A técnica de interrupção dispõe de um artifício usado para notificar o processador quando um dispositivo de E/S necessita de atenção. A notificação é

feita usando um circuito especial que sinaliza ao processador a necessidade de interrupção. Uma interrupção de E/S é assíncrona em relação às instruções, pois o dispositivo externo não é sincronizado com o processador. A unidade de controle do processador só verifica uma interrupção de E/S no momento em que começa uma nova instrução. Quando uma interrupção de E/S ocorre, são transmitidas informações adicionais, como a identidade do dispositivo que está gerando a interrupção, pela interface de E/S. Normalmente as interrupções são solicitadas por dispositivos que podem ter diferentes prioridades em relação ao seu atendimento.

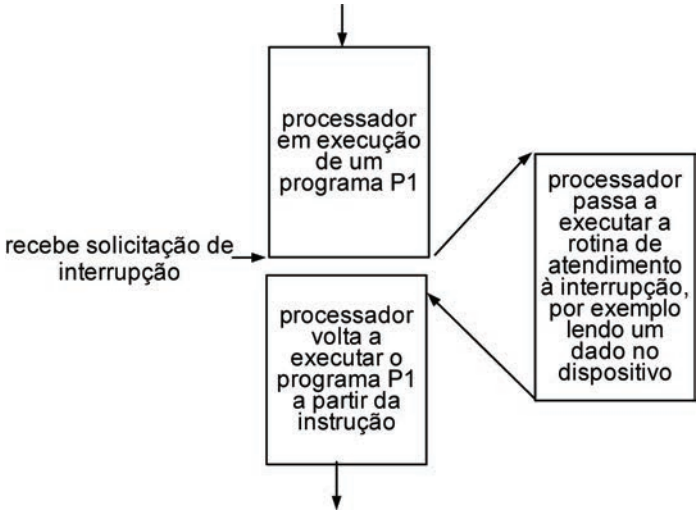


Figura 139 Diagrama de fluxo de execução do processador para atendimento a uma interrupção.

A Figura 139 mostra um diagrama no qual se verifica o papel do processador ao receber uma notificação de interrupção. Quando recebe uma solicitação de interrupção, o processador se encontra executando um programa P1. Nesse momento, caso o processador esteja habilitado a ser interrompido e a prioridade da solicitação for satisfeita, ele interrompe a execução do programa P1 e passa a executar a rotina de atendimento à interrupção, por exemplo, lendo um dado pronto no dispositivo. Ao realizar a leitura do dado, o dispositivo de entrada é liberado e o processador volta a executar o programa interrompido P1, a partir da posição da instrução interrompida.

Nota-se que, diferentemente da técnica de *polling*, no uso da interrupção, o processador não perde tempo fazendo leituras repetidas do *status* do dispositivo. O processador é avisado pelo dispositivo quando o mesmo estiver pronto por meio do artifício de interrupção; portanto, a técnica de entrada/saída por interrupção é mais eficiente que a técnica de *polling*.

9.3.3 Acesso direto à memória – DMA

Apesar da técnica de interrupção ser eficiente em relação à técnica de *polling*, uma operação de leitura de dado de um dispositivo pelo processador é ineficiente se considerarmos um número grande de dados que devem ser transferidos para a memória, ou da memória. São os casos de dispositivos de largura de banda alta (*bandwidth*), como discos rígidos, cujas transferências consistem em blocos de dados relativamente grandes (centenas a milhares de *bytes*).

No caso da leitura de um setor do disco rígido para a memória do computador, o procedimento de rotina de atendimento à interrupção, ilustrado na Figura 139, deve se repetir a cada dado e, além da leitura do dado do dispositivo, o processador deve guardá-lo na memória usando uma instrução de *store-word*. Nos casos de *polling* e interrupção, o trabalho de guardar dados fica a cargo do processador.

A técnica de acesso direto à memória, DMA (*Direct Memory Access*), faz a transferência dos dados do dispositivo diretamente à memória, usando um dispositivo especial chamado de controlador de DMA. Na técnica de DMA, a interrupção é também usada, mas somente no início e término da transferência de dados ou quando ocorre um erro. Nessa técnica é usado um controlador especializado, que transfere dados entre um dispositivo de E/S e a memória, independentemente do processador.

9.4 Barramentos

A técnica mais comumente usada para a interconexão de dispositivos de E/S em computadores é chamada de barramento. Neste item apresentamos alguns conceitos gerais associados aos barramentos, uma vez que cada tipo tem uma característica específica, cujas informações são descritas pelos seus fabricantes.

9.4.1 Características gerais

Os barramentos constituem o meio de interconexão mais simples e, portanto, mais usado entre os diversos componentes de um computador. Contêm um conjunto de linhas de controle e de linhas de dados, cujo acesso é compartilhado entre os dispositivos que são conectados, que podem ser, em geral: processador, memória e dispositivos de E/S. As linhas de controle são usadas para sinalizar solicitações e confirmações e, também, para indicar que tipo de informação se encontra nas linhas de dados. As linhas de dados transportam informações entre a origem e o destino, que são compostas por dados, comandos complexos ou endereços.

O projeto do barramento deve levar em consideração alguns parâmetros, tais como:

1. apenas um dispositivo por vez deve ter controle do barramento, pelo fato de diversos dispositivos compartilharem um mesmo barramento. Isso pode gerar um gargalo (*bottleneck*), quando o número de dispositivos for muito grande;
2. o comprimento do barramento tem limitação física, devido a características físicas dos circuitos;
3. o número de dispositivos deve ser limitado devido aos dois itens anteriores.

Existem vários tipos de barramentos:

1. processador-memória (caracteriza-se por ser pequeno, de alta velocidade e de projeto específico);
2. *backplane* (alta velocidade, em geral padronizado, por exemplo, barramento PCI);
3. E/S (dispositivos diferentes, padronizados, por exemplo, barramento SCSI);
4. síncrono, que usa *clock* e um protocolo síncrono. É rápido e pequeno, mas todos os dispositivos devem operar a uma mesma taxa, e a distorção (*skew*) do *clock* requer barramento curto;
5. assíncrono, que não usa *clock* e sim *handshaking*, que é um protocolo de sinalização para troca de dados no barramento.

Num barramento são realizadas transações de transmissão de dados dos dispositivos origem para os dispositivos destino, e para essas transações deve-se ter um controle. O dispositivo mestre é quem controla o barramento e, normalmente, é o módulo do processador. O dispositivo escravo é controlado e, normalmente, é o módulo de memória e E/S. Mestre temporário é quem assume o controle temporariamente.

A Figura 140 mostra um diagrama de um sistema de computação interligando, por meio de barramentos, os dispositivos de E/S e memória.

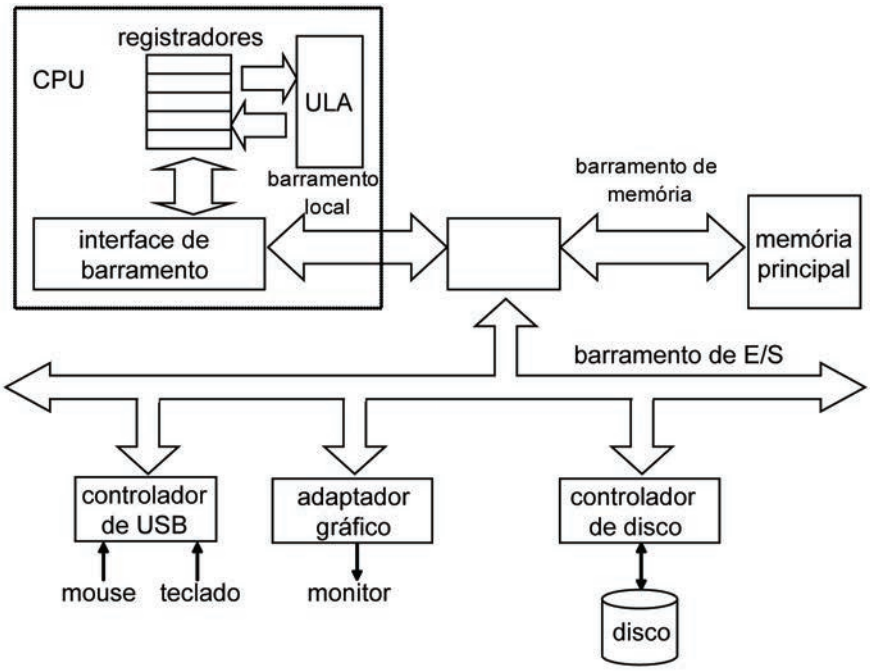


Figura 140 Diagrama de um sistema de computação interligando os dispositivos de E/S e memória usando barramentos.

Na parte superior à esquerda do diagrama, nota-se que a CPU é interligada aos demais dispositivos por um barramento local. A memória principal se interliga à CPU pelo barramento de memória e os dispositivos de E/S são interligados pelo barramento de E/S.

9.4.2 Disco magnético e o uso do mecanismo de DMA e interrupção para leitura

A Figura 141 ilustra um disco magnético composto de vários discos individuais, que armazenam informações em trilhas concêntricas divididas, por sua vez, em setores. Para fazer acesso a dados são necessárias as seguintes etapas:

1. busca (*seek*): posicionar a cabeça sobre a trilha (*8* a *20 ms*);
2. latência rotacional: espera por um setor desejado (*0,5/rpm*);
3. transferência: leitura dos dados (um ou mais setores) em *30* a *80 Mbits/s*.

Numa trilha, a latência média para a informação desejada está a meio caminho, ao redor do disco. Como os discos giram entre *5.400* e *15.000 rpm*, a latência rotacional média está entre:

$0,5/5.400\ rpm = 0,5/(5.400/60) = 5,6\ ms$ e

$0,5/15.000\ rpm = 0,5/(15.000/60) = 2,0\ ms$.

O tempo de transferência é uma função do tamanho do setor, da velocidade de rotação e da densidade de gravação de uma trilha.

Exemplo: *30* a *80 Mbits/s*.

A maioria dos controladores tem uma memória *cache* interna que armazena setores, e suas taxas de transferência são da ordem de *320 Mbits/s*.

Exercício: Qual é o tempo médio para ler ou escrever um setor de *512 bytes* em um disco rígido girando a *10.000 rpm*? O tempo de busca médio (*seek*) anunciado é de *6 ms*, a taxa de transferência é de *50 Mbytes/s* e o *overhead* da controladora é de *0,2 ms*. Suponha que o disco esteja ocioso de modo que não exista um tempo de espera.

Solução: O tempo médio de acesso é igual ao tempo médio de busca + latência rotacional média + tempo de transferência média + *overhead* da controladora.

$$6,0\ ms + 0,5/(10/60) + 512/50.000 + 0,2\ ms =$$

$$6,0\ ms + 3,0\ ms + 0,01\ ms + 0,2\ ms = 9,21\ ms$$

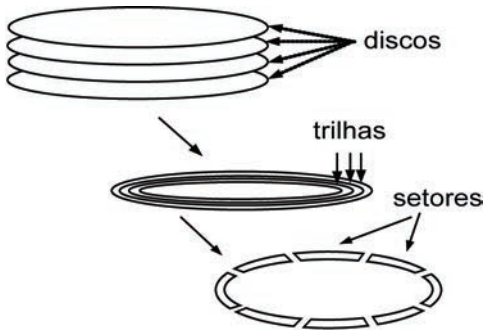


Figura 141 Ilustração de armazenamento em disco.

Mostra-se a seguir as etapas para a leitura de um setor num disco magnético. A Figura 142 mostra a primeira etapa, quando a CPU inicia uma leitura de disco escrevendo um comando, um número de bloco lógico e um endereço de memória num porto associado a um controlador de disco. Um porto é um registrador acessível pelo processador por meio de instruções e que fica na interface do dispositivo, no caso, controlador de disco.

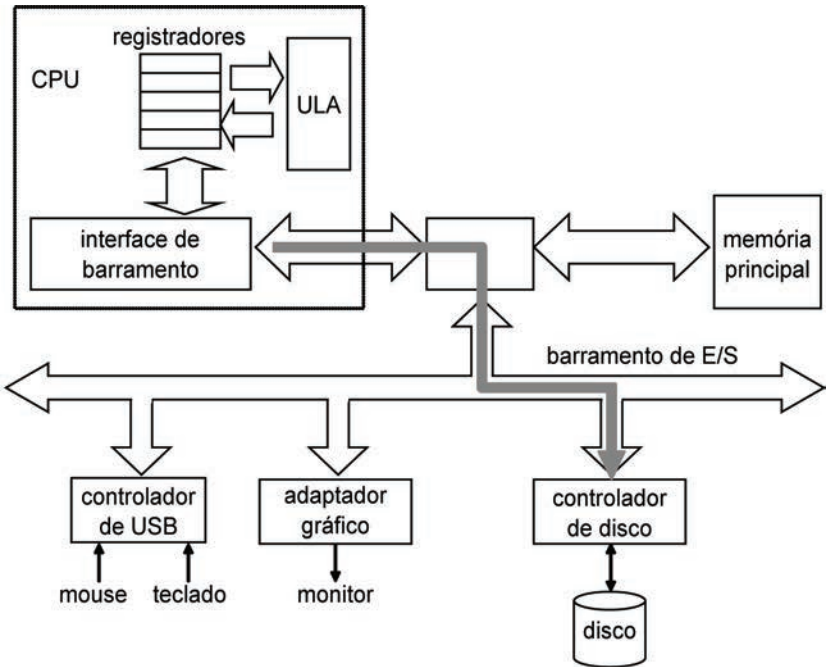


Figura 142 Etapa 1 para leitura de um setor do disco.

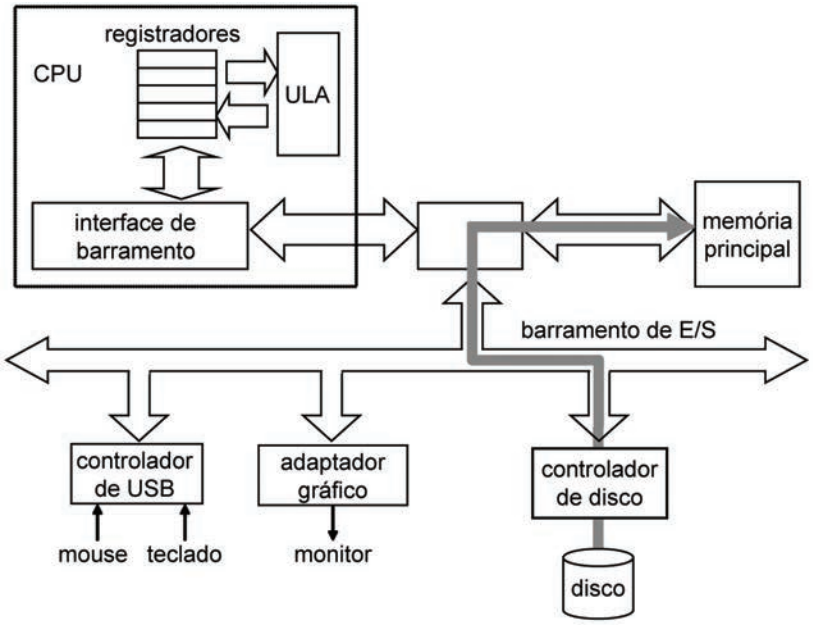


Figura 143 Etapa 2 para leitura de um setor de disco.

A Figura 143 mostra a segunda etapa, em que o dispositivo de disco reconhece o comando de leitura deixado pela CPU na primeira etapa e faz a transferência de dados lidos do disco no bloco lógico correspondente, para o respectivo endereço de memória.

A Figura 144 mostra a etapa final, em que a transferência usando DMA termina. O controlador de disco notifica a CPU usando interrupção (sinaliza a interrupção usando um pino especial na CPU).

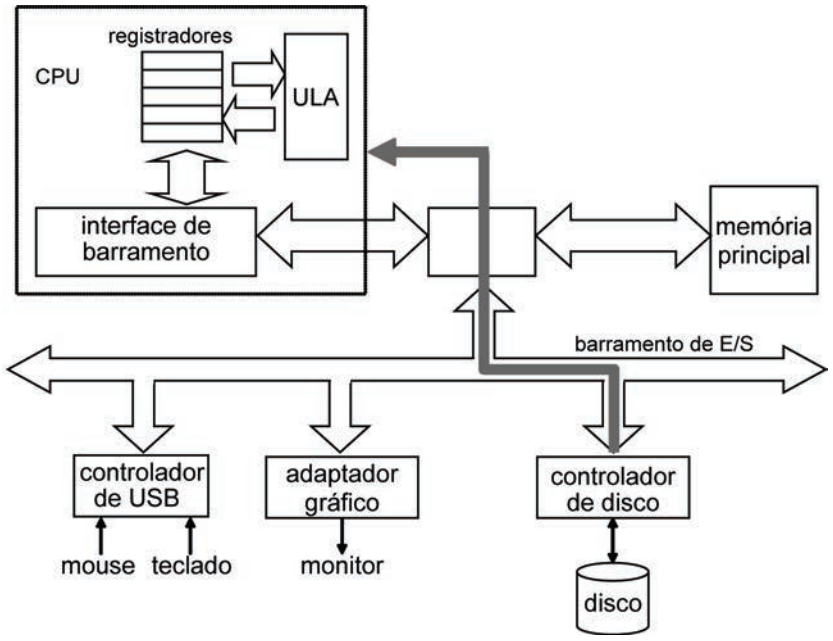


Figura 144 Etapa final para leitura de um setor de disco.

9.5 Sistema Pentium 4 de interconexões

No final da década de 1990, a resolução dos monitores tinha aumentado, em alguns casos para 1600 x 1200 *pixels*, e, em função disso, a demanda pelos gráficos aumentou. Nesse contexto, a Intel lançou um novo barramento apenas para comandar a placa gráfica, denominado AGP (*Accelerated Graphics Port Bus*). A versão inicial, AGP 1.0, funcionou a *264 Mbits/s*, o que foi definido por 1x. Com o passar dos anos surgiram novas versões, com AGP 3.0 funcionando a *2,1 Gbits/s* (8x). Um sistema Pentium 4, Figura 145, contém agora um *chip* ponte central e conecta as cinco peças mais importantes do sistema: a CPU, a memória, a placa gráfica, o controlador ATAPI (*AT Attachment Packet Interface*) e o barramento PCI. Em algumas variações há também suporte para rede *Ethernet*. Os dispositivos de menor velocidade são ligados ao barramento PCI. Internamente, o *chip* ponte é dividido em duas partes: a ponte de memória e a ponte de E/S. A ponte de memória conecta a CPU à memória e ao adaptador gráfico. A ponte de E/S conecta o controlador ATAPI, o barramento PCI e, opcionalmente, outros dispositivos rápidos de E/S, com conexão direta de ponte. As duas pontes são conectadas por uma interconexão de velocidade muito alta e às vezes são integradas num único *chip*. A Figura 145 mostra um sistema Pentium 4 com diversos barramentos associados.

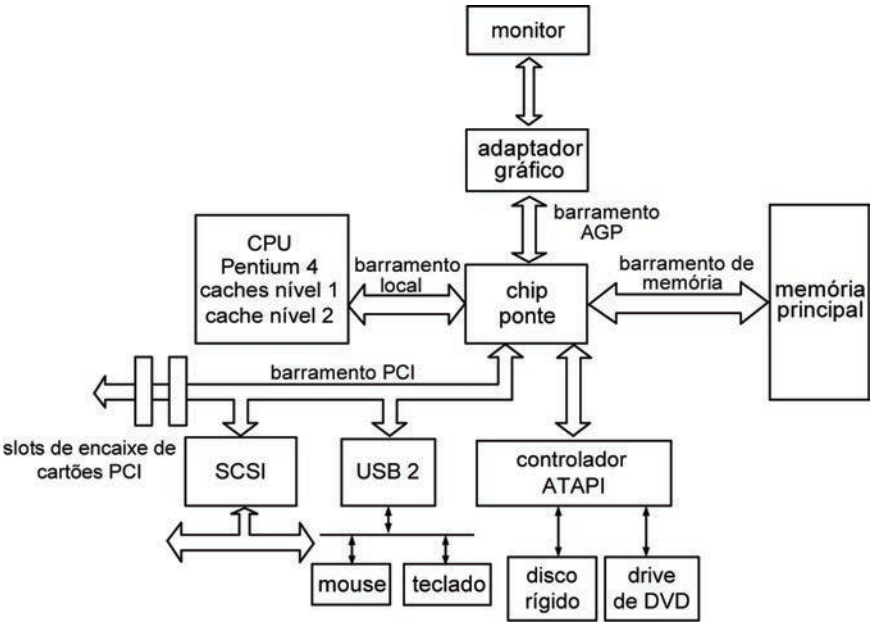


Figura 145 Configuração de interconexão de um sistema Pentium 4.

Fonte: adaptada de Tanenbaum (2007).

9.5.1 Barramento PCI

No IBM PC a maioria das aplicações era baseada em texto. Gradualmente, com a introdução do Windows, interfaces gráficas de usuário começaram a ser usadas.

Com o tempo, muitas aplicações, em especial jogos em multimídia, começaram a ser feitas em computadores para usar vídeo de tela inteira e movimento total. Uma tela de 1024 x 768 *pixels*, em cores reais (*3 bytes* por *pixel*), exige *2,30 Mbytes* de dados. Para a visualização de movimentos suaves são necessários 30 quadros por segundo, ou *69,12 Mbytes* de dados por segundo. Como o conteúdo a ser visualizado fica no disco magnético, CD-ROM ou DVD, os dados devem passar do acionador (*drive*) de disco para o barramento e ir até a memória. Para a apresentação, os dados devem percorrer o barramento novamente e ir até o adaptador de vídeo gráfico. Portanto, precisa-se de um barramento de *135 Mbytes* por segundo só para o vídeo, sem contar a largura de banda que a CPU e outros dispositivos necessitam.

Em 1990, a Intel, percebendo a situação, desenvolveu um novo barramento com largura de banda muito mais alta, denominado PCI (*Peripheral Component Interconnect Bus*). Para incentivar seu uso, a Intel patenteou o barramento PCI e passou todas as patentes para domínio público. A Intel também organizou um consórcio de empresas, *PCI Interest Group*, para gerenciar o futuro do barramento PCI. Como resultado, muitos computadores usam o PCI, mesmo os que não usam *chips* da Intel, como o UltraSPARC.

Na Figura 145, o barramento PCI está ligando uma interface SCSI, USB2, e *slots* de encaixe de cartões de PCI. A interface SCSI (*Small Computer Systems Interface*) é um padrão de conexão de periféricos como: discos rígidos, CD-ROM, CD-R, DVD, ZIP *drives*, *scanners* de mesa, fitas *streammer*, etc. Permite conectar até 15 periféricos usando o SCSI-3, e usar cabos de até 6 metros.

9.5.2 Interface USB

A interface USB (*Universal Serial Bus*) foi lançada em 1998 pelas empresas COMPAQ, DEC, IBM, Microsoft, NEC, Northern Telecom, entre outras. Pode conectar até 127 dispositivos ao barramento. A grande vantagem dessa interface é que o próprio usuário pode instalar um novo periférico, sem conflito ou acidente. Cada cabo USB pode ter até 5 m de comprimento em cada trecho, entre um periférico e uma tomada. Como cada periférico concentrador amplifica o sinal do barramento que vem pelo cabo, pode-se ter um barramento muito grande. O barramento é *plug-and-play*, ou seja, é possível encaixar e desencaixar periféricos com o computador ligado. Utiliza três taxas de transferência:

- USB 3.1: *10 Gbits/s*
- USB 3.0: *5 Gbits/s*
- USB 2.0: *480 Mbits/s*, usada para dados de áudio e MPEG-2;
- USB 1.1: *12 Mbits/s*, usada por periféricos como câmeras digitais, *modems*, impressoras e *scanners*;
- USB 1.0: *1,5 Mbits/s*, usada para periféricos mais lentos (teclados, *joysticks* e *mouse*).

Ainda, na Figura 145, o controlador ATAPI, ligado diretamente no *chip* ponte, serve para interfacear os dispositivos de disco com padrão de conexão ATAPI (*AT Attachment Packet Interface*) ou ATA-2. Esse padrão permite conexão de outros dispositivos ao computador, como unidades de CD-ROM, DVD, gravadores de CD-R e CD-RW, unidades Zip, etc.

9.5.3 Barramento PCI *Express*

Embora o funcionamento do PCI seja adequado para a maioria das aplicações existentes, a necessidade de maior largura de banda de E/S é prevista. O problema é que há, cada vez mais, dispositivos de E/S muito rápidos para o barramento PCI. Toda vez que isso ocorre, a Intel acrescenta uma porta especial no *chip* ponte para permitir que o dispositivo desvie do barramento PCI. Outro problema é que as placas são muito grandes, não cabem nos *laptops* e os fabricantes gostariam de produzir dispositivos menores.

Diversas soluções foram propostas, e a que tem sido adotada é denominada PCI *Express*. Ela pouco se relaciona com o barramento PCI e, tampouco é barramento, mas o pessoal do *marketing* quer manter o famoso nome PCI.

A intenção principal do PCI *Express* é se desvencilhar do barramento paralelo com seus muitos mestres e escravos e passar para um projeto baseado em conexões seriais ponto a ponto de alta velocidade. Essa solução representa uma ruptura radical com a tradição do barramento ISA/EISA/PCI e adquire ideias do mundo das redes locais, em especial a rede *Ethernet* com comutação de pacotes. A ideia básica se resume no fato de um computador ser um conjunto de *chips* de CPU, memória e controladores de E/S que precisam ser interconectados. O que o PCI *Express* faz é fornecer um comutador de uso geral para conectar *chips* usando ligações seriais. A Figura 146 mostra o diagrama de um sistema com um barramento PCI *Express*.

A memória e o *cache* estão conectados ao *chip* ponte de forma tradicional. A novidade é um comutador conectado à ponte, possivelmente fazendo parte da ponte. Cada um dos *chips* de E/S tem uma conexão ponto a ponto dedicada ao comutador. Cada conexão consiste em um par de canais unidirecionais, um que vai para o comutador e outro que vem dele. Cada canal é composto de dois fios, um para o sinal e outro para o terra. A arquitetura PCI *Express* tem três pontos diferentes em relação ao barramento PCI:

- um comutador centralizado;
- utilização de conexões seriais ponto a ponto; e
- envio de pacote de dados.

O conceito pacote, que consiste em um cabeçalho e uma carga útil, vem do mundo das redes. O cabeçalho contém informação de controle, que elimina a necessidade de muitos sinais de controle presentes no barramento PCI. O computador com PCI *Express* é uma rede de comutação de pacotes em miniatura.

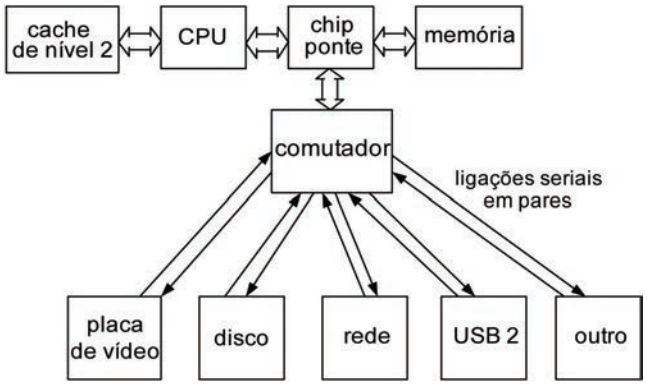


Figura 146 Diagrama de um sistema usando PCI *Express*.

Fonte: adaptada de Tanenbaum (2007).

9.6 Considerações finais

Nesta unidade foram descritos alguns conceitos e exemplos de formas de entrada e saída. Foram descritas também as principais interfaces usadas pelos periféricos.

9.7 Estudos complementares

Para complementar os estudos sobre entrada e saída os leitores podem se reportar ao capítulo *Storage and Other I/O Topics*, do livro de Hennessy & Patterson (2008), ou ao oitavo capítulo da versão traduzida por Daniel Vieira (HENNESSY & PATTERSON, 2005). Outra fonte de estudos é o segundo capítulo, *Organização de sistemas de computadores*, do livro de Tanenbaum (2007).

9.8 Exercícios

1. Quais as diferenças entre as técnicas de entrada/saída por *polling* e por interrupção?
2. Caracterizar a técnica de acesso direto à memória (Direct Memory Access – DMA).
3. Qual é o tempo médio para ler ou escrever um setor de 1K bytes em um disco rígido girando a 5.000 rpm? O tempo de busca (seek) médio anunciado é de 5 ms, a taxa de transferência é de 100 Mbytes/s e o overhead da controladora é de 0,4 ms. Suponha que o disco esteja ocioso, de modo que não exista um tempo de espera inicial.
4. Qual a função do chip ponte nos sistemas como o do Pentium 4?
5. Quais as características de um barramento PCI-Express, e qual a razão pelo qual a interface serial está sendo preferida atualmente, em muitas interfaces como a USB e PCI-Express?

REFERÊNCIAS

DANTAS, M. *Computação Distribuída de Alto Desempenho*: Redes, Clusters e Grids Computacionais. Rio de Janeiro: Axcel Books do Brasil, 2005.

DESCHAMPS, J. P.; BIOUL, G. J. A.; SUTTER, G. D. *Synthesis of Arithmetic Circuits*: FPGA, ASIC and Embedded Systems. New Jersey: John Wiley & Sons, 2006.

ERCEGOVAC, M.; LANG, T.; MORENO, J. *Introdução aos Sistemas Digitais*. Tradução de J. C. B. Santos. Porto Alegre: Bookman, 2000.

FLOYD, T. L. *Sistemas Digitais*: Fundamentos e Aplicações. 9. ed. Porto Alegre: Bookman, 2007.

HENNESSY, J. L.; PATTERSON, D. A. *Organização e Projeto de Computadores*: A Interface Hardware/Software. Tradução de Daniel Vieira. 3. ed. Rio de Janeiro: Campus, 2005.

_____. *Arquitetura de Computadores*: Uma Abordagem Quantitativa. 4. ed. Rio de Janeiro: Campus, 2007.

_____. *Computer Organization and Design*: The Hardware/Software Interface. 4. ed. Burlington: Morgan Kaufmann, 2008.

LEE, J. A. N. Visions and visionaries: Celebrating the history of computing. *IEEE Computer*, Washington, v. 29, n. 10, 1996.

TANENBAUM, A. S. *Organização Estruturada de Computadores*. 5. ed. Tradução de Arlete Simille Marques. São Paulo: Prentice-Hall, 2007.

WEBER, R. F. *Fundamentos de Arquitetura de Computadores*. Porto Alegre: Sagra Luzzatto, 2000.

ANEXO 1

Soluções dos exercícios

UNIDADE 2

1. Solução:

a) 000, 001, 010, 011, 100, 101, 110, 111

b) a'b'c', a'b'c, a'bc', a'bc, ab'c', ab'c, abc', abc

2. Solução:

x_2	x_1	x_0	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

3. Solução:

x_2	x_1	x_0	f
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

4. Solução:

$$f(x_2, x_1, x_0) = x_2 x_1 x_0$$

5. Solução:

$$f(x_2,\ x_1,\ x_0) = x_2 + x_1 + x_0 \quad ou$$
$$f(x_2,\ x_1,\ x_0) = x_2' x_1' x_0 + x_2' x_1 x_0' + x_2' x_1 x_0 + x_2 x_1' x_0' + x_2 x_1' x_0 + x_2 x_1 x_0' + x_2 x_1 x_0$$

6. Solução:

Aplicando a distributividade (ident.2), expressão do complemento $(a + a' = 1)$ (ident.5), e a identidade multiplicativa (ident.7):

a) $x_2' x_1' x_0 + x_2' x_1 x_0 = x_2' (x_1' x_0 + x_1 x_0) = x_2' ((x_1' + x_1) x_0) = x_2' (1 \cdot x_0) = x_2' x_0$

Analogamente:

b) $x_2' x_1 x_0' + x_2' x_1 x_0 = x_2' x_1$

c) $x_2 x_1' x_0' + x_2 x_1' x_0 = x_2 x_1'$

d) $x_2 x_1 x_0' + x_2 x_1 x_0 = x_2 x_1$

e) $x_2 x_1' x_0 + x_2 x_1 x_0 = x_2 x_1$

7. Solução:

Aplicando a comutatividade temos:

$$Z = x_2' x_0 + x_2 x_0 + x_2' x_1 + x_2 x_1 + x_2 x_1'$$

e aplicando a lei da idempotência $(a + a = a)$ (ident. 4), repetindo o penúltimo termo, temos:

$$Z = x_2' x_0 + x_2 x_0 + x_2' x_1 + x_2 x_1 + x_2 x_1 + x_2 x_1'$$

Aplicando a distributividade, o complemento e a identidade multiplicativa, de dois em dois termos subsequentes, temos:

$$z = (x_2' + x_2) x_0 + (x_2' + x_2) x_1 + x_2 (x_1 + x_1') = x_0 + x_1 + x_2$$

8. Solução:

Mostramos na tabela abaixo, as variáveis intermediárias *e* e *f* e a saída *g*:

a b c d	*e f*	*g*
0 0 0 0	0 0	0
0 0 0 1	0 0	0
0 0 1 0	0 0	0
0 0 1 1	0 1	1
0 1 0 0	0 0	0
0 1 0 1	0 0	0
0 1 1 0	0 0	0
0 1 1 1	0 1	1

a b c d	*e f*	*g*
1 0 0 0	0 0	0
1 0 0 1	0 0	0
1 0 1 0	0 0	0
1 0 1 1	0 1	1
1 1 0 0	1 0	1
1 1 0 1	1 0	1
1 1 1 0	1 0	1
1 1 1 1	1 1	1

9. Solução:

Da mesma forma que o exercício 8, preenchemos a tabela obtendo primeiro os valores para as variáveis intermediárias *e* e *f* e posteriormente para a saída *g*.

a b c d	*e f*	*g*
0 0 0 0	1 1	0
0 0 0 1	1 1	0
0 0 1 0	1 1	0
0 0 1 1	1 0	1
0 1 0 0	1 1	0
0 1 0 1	1 1	0
0 1 1 0	1 1	0
0 1 1 1	1 0	1

a b c d	*e f*	*g*
1 0 0 0	1 1	0
1 0 0 1	1 1	0
1 0 1 0	1 1	0
1 0 1 1	1 0	1
1 1 0 0	0 1	1
1 1 0 1	0 1	1
1 1 1 0	0 1	1
1 1 1 1	0 0	1

10. Solução:

Uma porta XOR tem a função de chaveamento

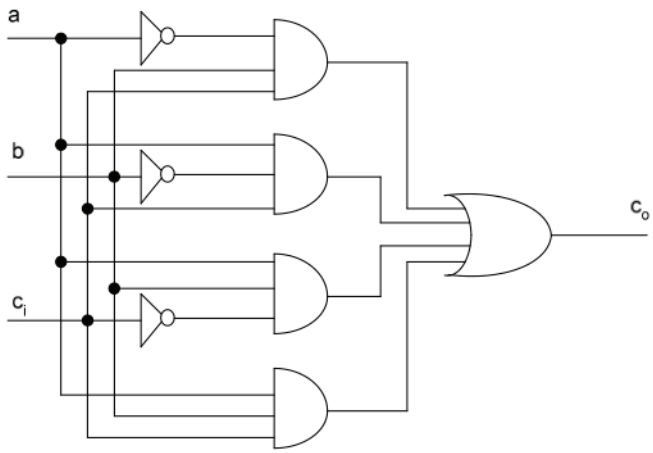
$$c = a'b + ab'$$

que corresponde exatamente ao circuito da figura.

11. Solução:

a) Função de chaveamento: $c_o = a'bc_i + ab'c_i + abc_i' + abc_i$

b) Circuito:



12. Solução:

a) Idempotência:

$$C_o = a'bc_i + ab'c_i + abc_i' + abc_i + abc_i + abc_i$$

b) Propriedade distributiva:

$$C_o = a'bc_i + abc_i + ab'c_i + abc_i + abc_i' + abc_i = (a' + a)bc_i + a(b' + b)c_i + ab(c_i' + c_i)$$

c) Complementos e multiplicativo:

$$C_o = 1 \cdot bc_i + a \cdot 1 \cdot c_i + ab \cdot 1 = bc_i + ac_i + ab = ab + (a + b)c_i$$

13. Solução:

a) Tabela-verdade:

$a\ b\ c_i$	$a \cdot b$	$a \oplus b$	$(a \oplus b) \cdot c_i$	C_o
0 0 0	0	0	0	0
0 0 1	0	0	0	0
0 1 0	0	1	0	0
0 1 1	0	1	1	1

a b c_i	a·b	a ⊕ b	(a ⊕ b)·c_i	C_o
1 0 0	0	1	0	0
1 0 1	0	1	1	1
1 1 0	1	0	0	1
1 1 1	1	0	0	1

b) Tabela verdade e função de chaveamento do exercício 11:

$$C_o = a'bc_i + ab'c_i + abc_i' + abc_i$$

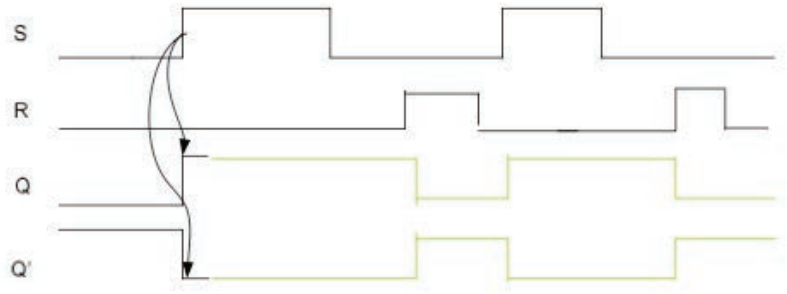
$a\ b\ c_i$	$a'bc_i$	$ab'c_i$	abc_i'	abc_i	C_o
0 0 0	0	0	0	0	0
0 0 1	0	0	0	0	0
0 1 0	0	0	0	0	0
0 1 1	1	0	0	0	1
1 0 0	0	0	0	0	0
1 0 1	0	1	0	0	1
1 1 0	0	0	1	0	1
1 1 1	0	0	0	1	1

c) Função simplificada no exercício 12: $C_o = ab + (a+b)c_i$

$a\ b\ c_i$	ab	$a+b$	$(a+b)c_i$	C_o
0 0 0	0	0	0	0
0 0 1	0	0	0	0
0 1 0	0	1	0	0
0 1 1	0	1	1	1
1 0 0	0	1	0	0
1 0 1	0	1	1	1
1 1 0	1	1	0	1
1 1 1	1	1	1	1

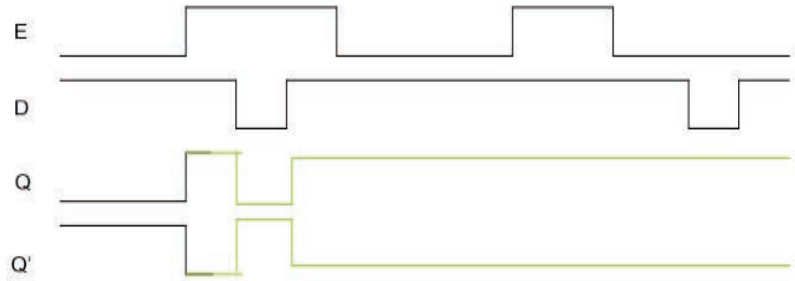
14. Solução:

Considera-se que o nível 1 equivale ao sinal alto, e nível 0, ao sinal baixo. Lembrando que a entrada S = 1 faz com que Q seja igual a 1, e R = 1 faz com que Q seja igual a 0. Após o acionamento com S=1, mesmo que S volte para 0, a saída Q permanece em 1, até que o sinal R seja acionado. A saída Q' é o inverso de Q.



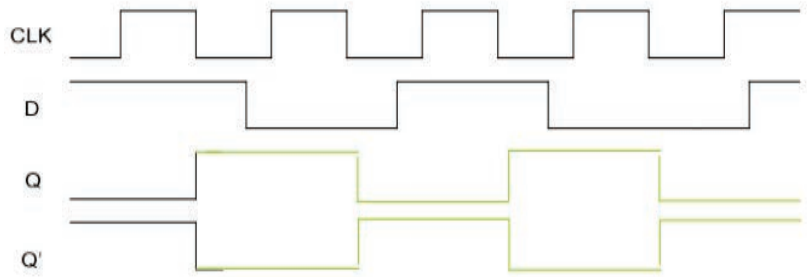
15. Solução:

No latch D, a saída Q é igual a entrada D, desde que a entrada E (controle) esteja em 1. Se a entrada E for zero, a saída Q permanece sem alteração.



16. Solução:

a) O flip-flop D, borda de descida, copia a entrada D, apenas na borda de descida do clock (CLK).



17. Solução:

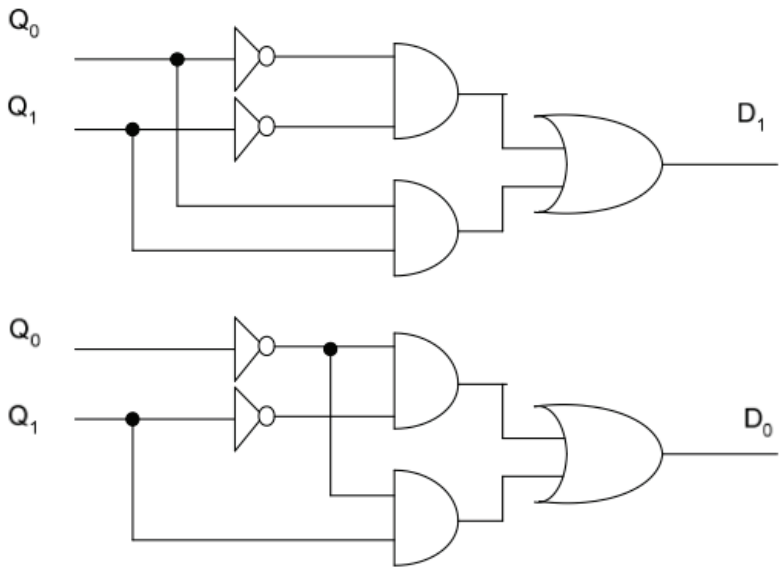
O circuito combinatório é obtido pela tabela de próximo estado, D_1 D_0, em função do estado atual Q_1 Q_0, obtido em função da sequência enunciada: 11 → 10 → 01 → 00 → 11.

Entradas: estado atual	Saídas: entradas dos flip-flops
Q_1 Q_0	D_1D_0
0 0	1 1
0 1	0 0
1 0	0 1
1 1	1 0

Que resultam nas expressões:

$$D_1 = Q_1' \cdot Q_0' + Q_1 Q_0$$
$$D_0 = Q_1' \cdot Q_0' + Q_1 Q_0'$$

Cujos circuitos devem ser:



UNIDADE 3

1. **Solução:** *$2 = 0 e $14 = 0*

2. **Solução:** *$3 = 68, $4 = 68* e *$14 = 0*

3. **Solução:** *0*

4. **Solução:** *$2 = 1*

5. **Solução:** *$7 = 1*

6. **Solução:** A instrução seguinte é a do endereço L1.

7. **Solução:**

   a) Considerando-se a variável a em $t0 a instrução seria sub $t0, $t0, 1 mas como não há subtração imediata ela seria substituída por addi $1, $0, 1 e sub $8, $8, $1.

   b) Considerando-se a variável a em $t0 a instrução seria add $t0, $0, $0.

   c) Considerando-se que o endereço do vetor v esteja armazenado no registrador $4 a instrução seria sw $0, 40($4).

   d) Considerando-se a variável a em $t0 e que o endereço do vetor v esteja armazenado no registrador $4 a instrução seria lw $t0, 40($4).

   e) Considerando-se a variável a em $2, e a variável b em $3, a sequência de instruções seria slt $7, $2, $3 e bne $7, $0, L1.

   f) Considerando-se a variável a em $3 a sequência de instruções seria slt $7, $0, $3 e bne $7, $0, L1.

8. **Solução:**

   Lembrar que o registrador destino é o terceiro no formato

op	rs	rt	rd	shamt	function
000000	00101	00110	00100	00000	100000

9. **Solução:**

op	rs	rd	offset
100011	00011	00100	0000000000000101

10. **Solução:**

op	rs	rd	offset
000101	00011	00100	0000000000000101

11. Solução:

Os valores modificados são apresentados em negrito a cada iteração

registradores				memória									
$2	**$3**	**$4**	**$5**	**v+0**	**v+4**	**v+8**	**v+12**	**v+16**	**v+20**	**v+24**	**v+28**	**v+32**	**v+36**
10	0	v+0	?	1	2	3	4	5	6	7	8	9	10
9	**1**	**v+4**	**1**	**1**	2	3	4	5	6	7	8	9	10
8	**3**	**v+8**	**2**	1	**3**	3	4	5	6	7	8	9	10
7	**6**	**v+12**	**3**	1	3	**6**	4	5	6	7	8	9	10
6	**10**	**v+16**	**4**	1	3	6	**10**	5	6	7	8	9	10
5	**15**	**v+20**	**5**	1	3	6	10	**15**	6	7	8	9	10
4	**21**	**v+24**	**6**	1	3	6	10	15	**21**	7	8	9	10
3	**28**	**v+28**	**7**	1	3	6	10	15	21	**28**	8	9	10
2	**36**	**v+32**	**8**	1	3	6	10	15	21	28	**36**	9	10
1	**45**	**v+36**	**9**	1	3	6	10	15	21	28	36	**45**	10
0	**55**	**v+40**	**10**	1	3	6	10	15	21	28	36	45	55

12. Solução:

```
.data
a:.word 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
b:.word 9, 8, 7, 6, 5, 4, 3, 2, 1, 0

.text
        li      $3,     0               # i = 0
        li      $2,     10              # contator de repeticao
L1:
        lw      $4,     a($3)           # carrega a[i]
        lw      $5,     b($3)           # carrega b[i]
        add     $5,     $5,     $4      # soma
        sw      $5,     a($3)           # armazena a[i]

        add     $3,     $3,     4       # incrementa i
        sub     $2,     $2,     1       # decrementar contador
        bgtz    $2,     L1              # salta se contador maior do que zero
```

13. Solução:

```
.data                       # secao de dados
space:.asciiz  " "

.text
        li      $t0,    0           # Inicializa t0 com 0
        li      $t1,    1           # Inicializa t1 com 1
        li      $t2,    10          # numero de repeticoes
loop:
        move    $a0,    $t0         # move $t0 para ser impresso
        li      $v0,    1           # escolhe impressao de inteiro
        syscall                     # imprime o numero

        la      $a0,    space       # carrega endereco do espaco
        li      $v0,    4           # escolhe impressao de string
        syscall                     # imprime um espaco

        move    $t3,    $t1         # salva $t1 anterior
        add     $t1,    $t1,    $t0 # calcula novo elemento em $t1
        move    $t0,    $t3         # move $t1 anterior para $t0

        addi    $t2,    $t2,    –1  # decrementa contador de loop
        bgtz    $t2,    loop        # repete enquanto maior que 0
```

UNIDADE 4

1. Solução:

Para os números naturais, ou números inteiros sem sinal, todos os bits são significativos, e cada bit tem um peso em potência de 2, começando de 2^0 para o bit mais à direita, portanto, o número 63 é representado por: 00111111, pois:

$$00111111 = 0 \times 2^7 + 0 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$$

$$= 0 \times 128 + 0 \times 64 + 1 \times 32 + 1 \times 16 + 1 \times 8 + 1 \times 4 + 1 \times 2 + 1 \times 1$$

$$= 0 + 0 + 32 + 16 + 8 + 4 + 2 + 1 = 63$$

2. Solução:

a) Sinal e magnitude:

Como o número tem 8 bits, sendo 1 bit de sinal e 7 bits de magnitude, temos: magnitude de 63 em binário, com 7 bits,

$$0111111 = 0 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$$

$$= 0 \times 64 + 1 \times 32 + 1 \times 16 + 1 \times 8 + 1 \times 4 + 1 \times 2 + 1 \times 1 = 0 + 32 + 16 + 8 + 4 + 2 + 1$$

A representação em sinal e magnitude de –63 é: 1 (sinal) e 0111111 (magnitude) = 10111111

b) Complemento de um:

Em complemento de um, um número negativo é obtido complementando todos os bits do seu correspondente positivo:

O número correspondente positivo de –63 é +63, cuja representação em complemento de um, em 8 bits, é 00111111. A representação do número –63 é então: 11000000

c) Complemento de dois:

Em complemento de 2, um número negativo é obtido pelo positivo correspondente, complementando todos os seus bits e somando 1:

Positivo correspondente = **00111111**

Complementando os seus bits = **11000000**

Somando 1 = **11000001**

Portando, a representação em complemento de 2 de –63 é: 11000001

3. Solução:

a) Sinal e magnitude: 1000000000111111

b) Complemento de um: 1111111111000000

c) Complemento de dois: 1111111111000001

4. Solução:

```
 carry       1
 17 =    00010001
+18 =+   00010010
-------------------
 35 =    00100011
```

5. Solução:

```
 carry  1111111
–17 =    11101111
+18 =+   00010010
-------------------
 1  =     00000001
```

O último carry é ignorado. Este e o anterior (destacados), se diferentes, indicam um overflow.

6. Solução:

```
  carry  01
 126 =    01111110
+64 =+    01000000
-------------------
−66 =     10111110
```

Ocorre overflow, pois a soma de dois números positivos resultou em um negativo. Os carrys diferentes indicam esta situação.

7. Solução:

```
 carry        10
–126     =    10000010
+(–64)   =+   11000000
-----------------------
  66     =    01000010
```

Ocorre overflow, pois a soma de dois números negativos resultou em um positivo. Os carrys diferentes indicam esta situação.

8. Solução:

Para o circuito de soma (a + b), considerando carry = 0, temos a seguinte tabela verdade:

a	b	s
0	0	0
0	1	1
1	0	1
1	1	0

Portanto, a solução é usar o circuito de soma, ou seja, operação = 10.

Numa ULA de 32 bits que usa também o carry como entrada da soma, o circuito de soma já não funciona para a operação XOR de 32 bits.

9. Solução:

Dada a tabela-verdade do circuito somador:

a	b	Cin	s	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

O circuito somador pode ser considerado composto de dois circuitos combinatórios, sendo:

$$s = a'b'Cin + a'bCin' + ab'Cin' + abCin \quad e \quad Cout = a'bCin + ab'Cin + abCin' + abCin$$

Colocando em evidência a' e a, para os dois primeiros termos e os dois termos restantes, respectivamente, na equação de s:

$s = a'\left(b'Cin + bCin'\right) + a\left(b'Cin' + bCin\right) = a'\left(b \oplus Cin\right) + a\left(b \oplus Cin\right)'$, portanto

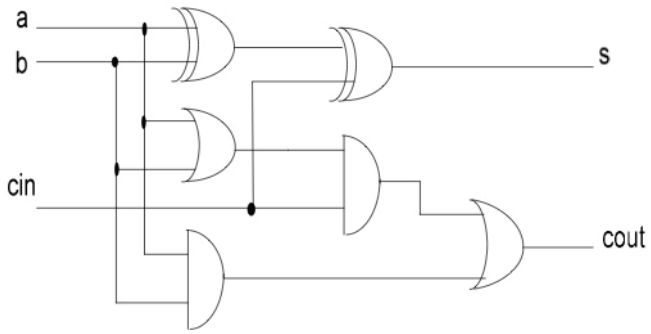
$s = a \oplus b \oplus Cin$.

Para a equação de Cout, replicamos o quarto termo abCin e colocamos em evidência Cin para os dois primeiros termos mais abCin, e colocamos ab em eviência para o dois termos restantes e obtemos:

$Cout = \left(a'b + ab' + ab\right)Cin + ab\left(Cin' + Cin\right)$.

Notamos que a'b + ab' + ab é igual a (a + b), e Cin' + Cin = 1, portanto: Cout = (a+b) Cin + ab.

O circuito resultante seria:

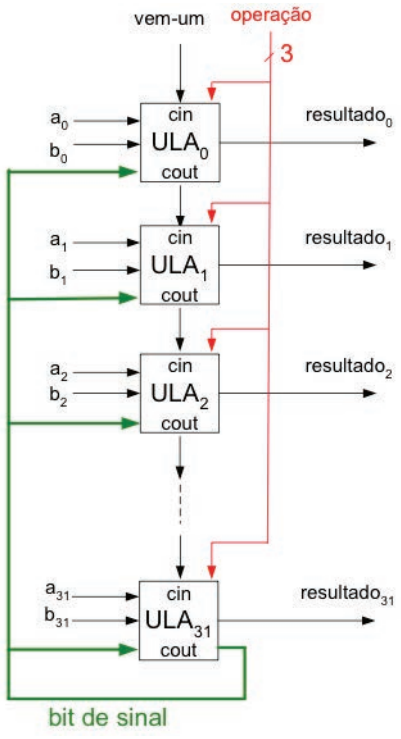


10. Solução:

Operação	Operando 1	Operando 2	Resultado	Flag de zero
000	32	36	32	0
001	32	36	36	0
010	32	32	64	0
110	32	32	0	1
111	32	32	0	1

11. Solução:

Para modificar o slt, levamos em conta que: –1 em binário é 111...11, portanto, todos os bits do resultado devem ser iguais a 1, quando a < b, e todos os bits devem ser iguais a 0, caso contrário, portanto a solução é obtida ligando o bit de sinal na entrada do multiplexador de saída de todas as 32 ULAs conforme a figura:



12. Solução:

1 10000000 11000000000000000000000

Sinal (1 bit) = *1*

Expoente (8 bits) = *10000000*

Significando (23 bits) = *11000000000000000000000*

O valor do número em binário é obtido usando a equação:

Valor em binário = $(-1)^{sinal} \times (1 + significando) \times 2^{expoente - bias}$

O expoente é polarizado, portanto devemos subtrair o bias = 127

expoente = *10000000* = *128*

expoente – bias = *128 – 127 = 1*

O significando tem um bit implícito à esquerda da parte fracionária portanto

1 + significando = *1 + 0,11000000000000000000000 = 1,11000000000000000000000*

Portanto,

Valor em binário $= (-1)^{1} \times 1,11000000000000000000000 \times 2^{1}$

$= (-1) \times 1,11000000000000000000000 \times 2$

$= (-1) \times 11,10000000000000000000000$

E o valor em decimal é obtido fazendo a conversão:

$(-1) \times (1 \times 2^{1} + 1 \times 2^{0} + 1 \times 2^{-1} + 0 \times 2^{-2} + 0 \times 2^{-3} + ...)$

$= (-1) \times (1 \times 2 + 1 \times 1 + 1 \times 0,5 + 0 + 0 + ...)$

$= (-1) \times (2 + 1 + 0,5 + 0 + 0 + ...) = (-1) \times 3,5 = -3,5$

13. Solução: 4,25

14. Solução: 4,75

15. Solução:

$3,75 = 2 + 1 + 0,5 + 0,25 = 2^{1} + 2^{0} + 2^{-1} + 2^{-2} = 11,11 = 2^{1} \times 1,111$ (normalização)

Temos então:

Sinal: 0 (positivo)

Expoente: 10000000 (polarizado, 1 + 127 = 128)

Significando: 11100000000000000000000 (remove-se o 1 depois da vírgula, pois está implícito)

01000000011100000000000000000000

16. Solução: 11000000111010000000000000000000

UNIDADE 5

1. Solução:

$$desempenho_{c1} = \frac{1}{tempo.de.exec_{c1}} = \frac{1}{10} = 0{,}1\ programas/segundo$$

$$desempenho_{c2} = \frac{1}{tempo.de.exec_{c2}} = \frac{1}{15} = 0{,}066\ programas/segundo$$

2. Solução:

$$taxa.de.clock = \frac{1}{tempo.de.ciclo} = \frac{1}{\frac{tempo.de.exec}{num.de.ciclos}} = \frac{num.de.ciclos}{tempo.de.exec} = \frac{20 \times 10^6}{10} = 2\ MHz$$

3. Solução:

a) Tempo de ciclo do processador:

$$tempo.de.ciclo = \frac{1}{taxa.de.clock} = \frac{1}{500 \times 10^6} = 2 \times 10^{-9}\ segundos$$

b) Número de ciclos do programa:

$$\begin{aligned} 6 \times 10^6\ &instruções\ da\ classe\ A,\ de\ 1\ ciclo &&= 6 \times 10^6\ ciclos \\ 3 \times 10^6\ &instruções\ da\ classe\ B,\ de\ 2\ ciclos &&= 6 \times 10^6\ ciclos \\ 2 \times 10^6\ &instruções\ da\ classe\ C,\ de\ 3\ ciclo &&= 6 \times 10^6\ ciclos \\ &Total\ de\ ciclos &&= 18 \times 10^6\ ciclos \end{aligned}$$

c) Tempo de execução do programa:

$$tempo.de.exec = num.de.ciclos \times tempo.de.ciclo = 18 \times 10^6 \times 2 \times 10^{-9} = 36 \times 10^{-3}\ segundos$$

d) CPI médio do programa:

$$CPI_{MÉDIO} = \frac{num.de.ciclos}{num.de.instruções} = \frac{18 \times 10^6}{11 \times 10^6} = \frac{18}{11} = 1{,}63$$

e) Throughput em MIPS para esse programa:

$$throughput_{MIPS} = \frac{num.de.instruções}{tempo.de\,exec \times 10^6} = \frac{11 \times 10^6}{36 \times 10^{-3} \times 10^6} = \frac{11}{36} \times 10^3 = 305\ MIPS$$

4. Solução:

	sub	*$2,*	*$2,*	*$2*	#	*4 ciclos*
loop1:	*muli*	*$14,*	*$2,*	*4*	#	$4 \times 10 \times 10^6$ *ciclos*
	add	*$3,*	*$4,*	*$14*	#	$4 \times 10 \times 10^6$ *ciclos*
	sw	*$0,*	*0*	*($3)*	#	$4 \times 10 \times 10^6$ *ciclos*
	addi	*$2,*	*$2,*	*1*	#	$4 \times 10 \times 10^6$ *ciclos*
	slt	*$7,*	*$2,*	*$6*	#	$4 \times 10 \times 10^6$ *ciclos*
	bne	*$7,*	*$0,*	*loop1*	#	$3 \times 10 \times 10^6$ *ciclos*

$$total.de.ciclos = 23 \times 10^7 + 4\,ciclos \sim 23 \times 10^7\ ciclos$$

$$tempo.de.exec = 23 \times 10^7 \times \frac{1}{10^9} = 23 \times 10^{-2}$$

$$desempenho = \frac{1}{tempo.de.exec} = \frac{1}{23 \times 10^{-2}} = \frac{10^2}{23} = 4{,}34\ programas$$

5. Solução:

a) Novo tempo de execução:

$$tempo.melhorado = 60 + \frac{40}{2} = 60 + 20 = 80\,segundos$$

b) *Speedup* obtido:

$$speed\text{-}up = \frac{tempo.sequencial}{tempo.melhorado} = \frac{100}{80} = 1{,}25\,vezes$$

6. Solução:

O tempo de execução da parte não afetada é de 5 segundos, pois as instruções de ponto flutuante constituem a metade dos 10 segundos. O tempo de execução da parte afetada é também de 5 segundos, e a razão de melhoramento é 5.

Segundo a lei de Amdahl:

$$tempo.de.exec_{novo} = tempo.de.exec_{parte.não.afetada} + \frac{tempo.de.exec_{parte.afetada}}{razão.de.melhoramento} = 5 + \frac{5}{5} = 6$$

Portanto:

$$speedup = \frac{tempo.de.exec_{antigo}}{tempo.de.exec_{novo}} = \frac{10}{6} = 1{,}66\,vezes$$

7. Solução:

Sendo α o percentual de instruções de ponto flutuante procurado, temos:

$$tempo.de.exec_{novo} = (1-\alpha) \times tempo.de.exec_{parte.não.afetada} + \alpha \times \frac{tempo.de.exec_{parte.afetada}}{razão.de.melhoramento}$$

$$speedup = \frac{tempo.de.exec_{antigo}}{tempo.de.exec_{novo}} \quad ou \quad tempo.de.exec_{novo} = \frac{tempo.de.exec_{antigo}}{speedup}$$

$$tempo.de.exec_{novo} = \frac{100}{3} = 33,3$$

$$33,3 = (1-\alpha)100 + \alpha\frac{100}{5} = 100 - 100\alpha + 20\alpha = 100 - 80\alpha$$

$$\alpha = \frac{100 - 33,3}{80} = \frac{66,6}{80} = 0,8325 = 83,25\%$$

UNIDADE 6

1. Solução:

O PC serve para controlar a execução das instruções em um computador. A cada início de um ciclo de instrução, o contador de programa contém o endereço de memória onde está a instrução a ser lida e executada.

2. Solução:

Formato R, usado pelas instruções: add, sub, and, or, slt

Opcode (6 bits)	Número do registrador fonte A (5 bits)	Número do registrador fonte B (5 bits)	Número do registrador de escrita (5 bits)	Shift-amount (não usado) (5 bits)	Função (6 bits)
0 0 0 0 0 0	x x x x x	x x x x x	x x x x x	x x x x x	x x 0 0 0 0 – add x x 0 0 1 0 – sub x x 0 1 0 0 – and x x 0 1 0 1 – or x x 1 0 1 0 – slt

3. Solução:

Formato I, usado pelas instruções: lw, sw

Opcode (6 bits)	Número do Registrador de endereço (5 bits)	Número do Registrador de escrita (lw) ou dado (sw) (5 bits)	Deslocamento (offset) do endereço (16 bits)
1 0 0 0 1 1 – lw 1 0 1 0 1 1 – sw	x x x x x	x x x x x	x x x x x x x x x x x x x x x x

4. Solução:

Formato I, usado pelas instruções: bne e beq

Opcode (6 bits)	Número do registrador A (5 bits)	Número do registrador B (5 bits)	Tamanho do desvio (16 bits)
0 0 0 1 0 0 – beq 0 0 0 1 0 1 – bne	x x x x x	x x x x x	x x x x x x x x x x x x x x x x

5. Solução:

Para executar o desvio dentro de um programa, o contador de programa (PC) deve ser atualizado com um novo valor de endereço que não seja o endereço sequencial, que no caso do MIPS é o endereço contido no PC acrescido de 4. O novo valor de endereço é calculado somando o valor contido no PC, com os 16

bits contidos na instrução, no campo “Tamanho do desvio” (ver exercício anterior), multiplicados por 4. A multiplicação por 4 é necessária para apontar a uma palavra na memória, já que esta é endereçada por byte.

6. Solução:

Formato para a instrução de desvio incondicional: j

Opcode (6 bits)	Tamanho do desvio (26 bits)
0 0 0 0 1 0	x x x x x x x x x x x x x x x x x x x x x x x x x x

7. Solução:

Para executar o desvio dentro de um programa, pela instrução j, o contador de programa (PC) deve ser atualizado com um novo valor calculado concatenando os 4 bits mais significativos (mais à esquerda) do PC, com os 26 bits contidos na instrução, no campo “Tamanho do desvio” (ver exercício anterior), seguidos de dois zeros, na extremidade direita. Os dois zeros representam a multiplicação por 4, para que o desvio seja em endereços múltiplos de 4.

8. Solução:

A tabela mostra como os 3 bits de seleção de operação da ULA são obtidas, ao longo da execução das instruções. Assim, para as instruções aritméticas (formato R), o opcode é igual 000000 para as instruções: add, sub, and, or e slt, e a forma de seleção de operação é pelos bits F3, F2, F1 e F0, que são os últimos 4 bits do campo de função no formato de instrução. Os bits F4 e F5 não estão sendo usados. Os bits OpULA1 = 1 e OpULA0 = 0, indicam que a instrução é de formato R. Para demais instruções o campo de função não existe e portanto, os bits F3, F2, F1, F0 da tabela são irrelevantes (marcados com x). Nesse a seleção de operação é feita exclusivamente pelos bits OpULA1 e OpULA0.

9. Solução:

O circuito serve para converter os 16 bits da instrução para um número de 32 bits. Caso o número seja negativo, o número estendido deve continuar sendo negativo. E caso o número seja positivo, o número estendido deve ser positivo.

10. Solução:

O circuito de deslocamento de 2 no MIPS, desloca, todos os 32 bits, dois bits à esquerda e preenche os espaços resultantes à direita com zeros. Essa operação corresponde a uma multiplicação por 4, e é usada para calcular o desvio de uma palavra, a partir da quantidade de desvios fornecida pela instrução de desvio.

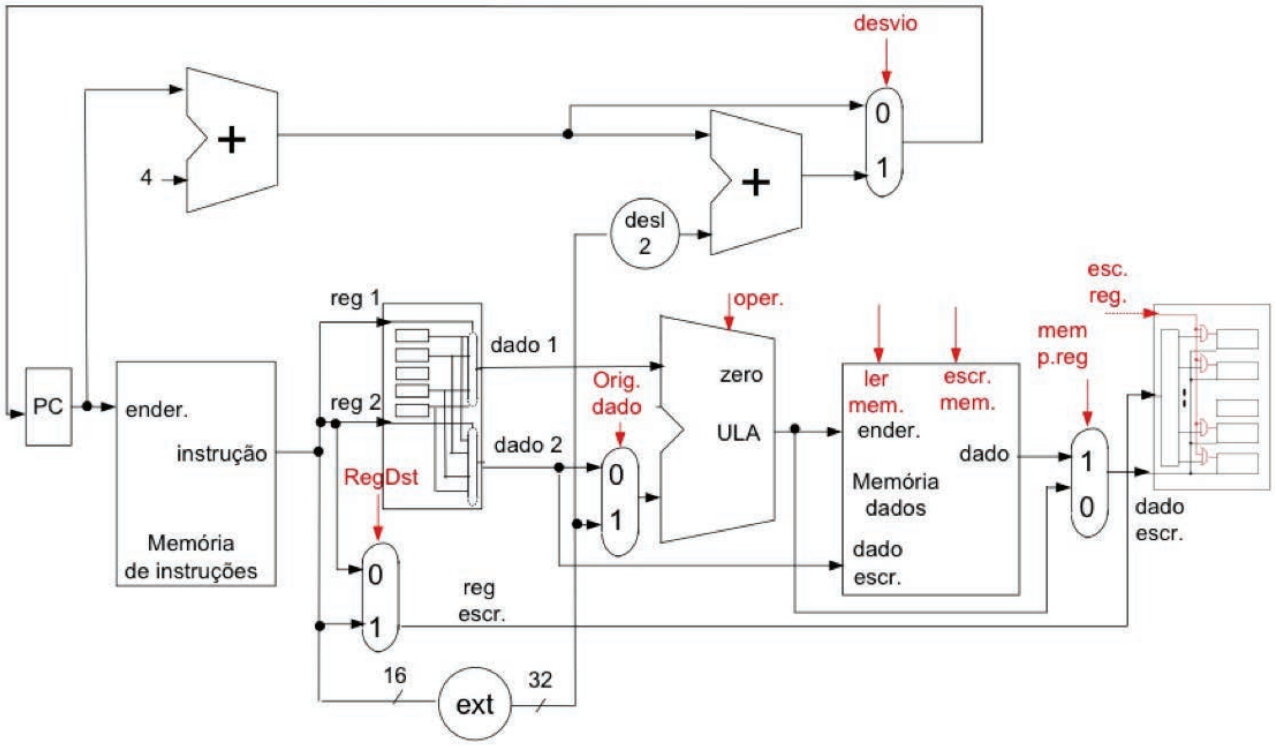
11. Solução:

Opcode	RegDst	OrigDado	MempReg	Escreve-Reg	Le-Mem	Escreve-Mem	Des-vio	OpULA1	OpULA0
000000	1	0	0	1	0	0	0	1	0
100011	0	1	1	1	1	0	0	0	0
101011	0	1	0	0	0	1	0	0	0
000100	0	0	0	0	0	0	1	0	1

12. Solução:

a) Para a instrução aritmética RegDst = 1; OrigDado =0; e MemparaReg = 0.

b) Para a instrução beq, desvio = 1;



13. Solução:

Porque as instruções no formato R usam o terceiro campo de registrador (bit15 ... bit11) para indicar o registrador de escrita, enquanto que a instrução lw usa o segundo campo de registrador (bit20 ... bit16) para indicar o registrador de escrita.

14. Solução:

Porque as instruções no formato R, bem como a instrução beq, usam o segundo registrador (dado2) como o segundo operando da ULA, enquanto que as instruções lw e sw usam os 16 bits estendidos da instrução como segundo operando da ULA.

15. Solução:

Porque as instruções no formato R, usam o resultado de operação da ULA como dado a ser escrito em um registrador, enquanto que a instrução lw, usa o dado lido na memória.

16. Solução:

Porque numa instrução de desvio condicional, caso a condição é satisfeita, a linha de controle de desvio é um, e é enviado um endereço de desvio para o PC, calculado deslocando de 2 à esquerda os 16 bits estendidos da instrução, e somando com o valor contido em PC somado de 4. No caso em que a condição não é satisfeita a linha de controle de desvio é zero e o novo endereço de PC é PC somado de 4.

17. Solução:

- RI (registrador de instrução), para que a no final do ciclo de busca de instrução, uma nova instrução contida em RI represente o novo estado do computador;
- A e B (registradores de dados lidos do banco de registradores), para que no final do segundo ciclo, os dados a serem processados estejam contidos nesses registradores;
- ULA (registrador do resultado da ULA), para que no final do ciclo de operação da ULA, o resultado esteja guardado nesse registrador;
- RDM (registrador de dado da memória), para que após a leitura de um dado da memória, o mesmo seja guardado nesse registrador.

18. Solução:

a) No estado 2, OrigDado = 1, OpULA1 = 0 e OpULA 0 = 0. O operando 1 da ULA é sempre o dado1 do banco de registradores. O operando 2, como OrigDado = 1, são os 16 bits da instrução estendidos para 32 bits. A operação aritmética é dada por OpULA1 = 0 e OpULA0 = 0, que resulta em adição.

b) No estado 4, RegDst = 0, MempReg = 1 e EscrReg = 1. O dado a ser escrito no registrador depende do multiplexador MempReg, que como é igual a 1, deve vir do registrador RDM.

19. Solução:

a) Acrescentar o estado 10 com a seguinte operação – calcular a adição de um operando do registrador com um segundo operando proveniente dos 16 bits da instrução estendidos para 32 bits, colocando o resultado no registrador ULA. Os sinais de controle são OrigDado = 1 e OpULA1 = 0 e OpULA0 = 0, para a operação de adição.

b) Acrescentar o estado 11 em sequência ao estado 10, com a operação de escrita do resultado contido no registrador ULA no registrador destino. Os sinais de controle são MempReg = 0 para selecionar o caminho do registrador ULA para o registrador destino; EscrReg = 1 , para escrever no registrador; e RegDst = 0, para selecionar o registrador destino.

20. Solução:

O controle fixo não usa uma memória para o controle, mas uma implementação fixa de um circuito combinatório. Portanto, não é flexível, e quando existe uma necessidade de alteração na implementação das instruções, o circuito combinatório deve ser refeito. Quando a quantidade de instruções no computador é pequeno, o circuito é relativamente simples.

O controle microprogramado usa uma memória para o controle, portanto é flexível. Quando existe uma necessidade de alteração nas instruções, apenas o conteúdo da memória de controle deve ser alterado. Mesmo quando a quantidade de instruções é grande a unidade de controle é relativamente simples.

Comparando os dois tipos de unidades de controle, quando o número de instruções aumenta, a complexidade aumenta para o controle fixo, enquanto que para o controle microprogramado se mantém relativamente constante. Quanto a latência, o controle fixo é mais eficiente, ao contrário do controle microprogramado que depende da leitura da memória de controle, o que é mais lento que um circuito combinatório diretamente implementado em portas lógicas, como é o caso do controle fixo.

21. Solução:

CAUSA: guarda um código que representa o motivo da exceção. O código é introduzido no instante em que ocorre o ciclo de exceção, que desvia o programa para a rotina de tratamento de exceção.

EPC: guarda o endereço contido no contador de programa, PC, no momento de ocorrência do ciclo de exceção. Esse endereço serve para retornar o programa interrompido, após a execução da rotina de tratamento de exceção.

UNIDADE 7

1. Solução:

a) Para o MIPS monociclo, o tempo de ciclo de uma instrução é determinado pela instrução mais demorada, pois todas as instruções têm o mesmo tempo de ciclo. A instrução mais demorada no MIPS é lw que necessita das operações: busca de instrução (2ns), leitura de registradores (1ns), ULA (2ns), acesso a memória de dados (2ns) e escrita de registrador (1ns). Portanto, uma instrução tipo R também tem o tempo de execução de 8 ns, como a instrução lw.

b) Para o MIPS multiciclo, cada instrução executa uma instrução usando um número de ciclos apropriado, porém cada ciclo deve ter uma mesma duração, que é a maior duração entre as operações. No nosso caso, a maior duração entre as operações é de 2 ns. Portanto, uma instrução tipo R deve usar 4 ciclos: busca de instrução, leitura de registrador, ULA e escrita de registrador, e o tempo de 8 ns.

c) Para o MIPS pipeline, o número de estágios é 5 e igual para todas as instruções. Portanto uma instrução tipo R usa 5 estágios, ou ciclos, de 2 ns, ou 10 ns para execução.

2. Solução:

O *speedup* para o pipeline de instruções é calculado usando a equação

$$speedup = \frac{n.k}{k + n - 1}$$

onde k é o número de estágios e n é o número de instruções do programa.

a) Para n = 20 temos:

$$speedup = \frac{20 \cdot 6}{6 + 20 - 1} = \frac{120}{25} = 4,8$$

b) Para n = 2000 temos:

$$speedup = \frac{2000 \cdot 6}{6 + 2000 - 1} = \frac{12000}{2005} = 5,98$$

3. Solução:

Os latches são da esquerda para a direita, IF/ID, ID/EX, EX/M e M/WB.

Os sinais neles presentes:

- IF/ID: sem sinais
- ID/EX: OpULA0, OpULA1, RegDst, Origdado, LerMem, EscrMem, Desvio, MempReg, EscReg
- EX/M: LerMem, EscrMem, Desvio, MempReg, EscReg
- M/WB: MempReg, EscReg

Os sinais usados no estágio imediatamente seguinte:

- ID/EX: OpULA0, OpULA1, RegDst, Origdado
- EX/M: LerMem, EscrMem
- M/WB: MempReg, EscReg

4. Solução:

a)

```
lw    $5,   0($6)        #  Estagio WB
add   $2,   $3,   $4     #  Estagio M
sw    $6,   0($6)        #  Estagio EX
and   $2,   $5,   $7     #  Estagio ID
or    $4,   $5,   $6     #  Estagio IF
```

b) Os sinais de controle presentes nos latches são:

- ID/EX : sinais de controle para instrução sw
- OpULA0 = 0, OpULA1 = 0, RegDst= x, Origdado = 1; LerMem=0, EscrMem=1, Desvio=0, MempReg=x, EscReg=0
- EX/M: sinais de controle para a instrução add
- LerMem=0, EscrMem=0, Desvio=0, MempReg=0, EscReg=1
- M/WB: sinais de controle para a instrução lw
- MempReg=1, EscReg=1

c) Os sinais usados imediatamente nesse mesmo ciclo, no estágio subsequente são:

- ID/EX : sinais de controle para instrução sw
- OpULA0 = 0, OpULA1 = 0, RegDst= x, Origdado = 1
- EX/M: sinais de controle para a instrução add

- LerMem=0, EscrMem=0, Desvio=0
- M/WB: sinais de controle para a instrução lw
- MempReg=1, EscReg=1

d) Os dados contidos no latch MEM/WB são os referentes à instrução lw $5, 0($6) que está no estágio WB: o dado lido na memória, e o endereço de memória da mesma instrução lw, sendo que o multiplexador MemP.Reg deve selecionar o dado lido na memória, para escrita no registrador.

5. Solução:

A instrução add $5, $2, $4 depende da instrução sub $2, $4, $6 – no registrador $2.

Solução: antecipação.

A instrução or $7, $2, $8 depende da instrução sub $2, $4, $6 – no registrador $2.

Solução: antecipação.

A instrução lw $9, 10($7) depende da instrução or $7, $2, $8 – no registrador $7.

Solução: antecipação.

A instrução sw $7, 10($5) depende da instrução or $7, $2, $8 – no registrador $7.

Solução: antecipação – observando que esse circuito de antecipação não é o mesmo circuito de antecipação colocado no estágio EX, e sim, um circuito que deve ser colocado no estágio M.

6. Solução:

Processador superescalar é um processador com capacidade de despachar mais de uma instrução simultaneamente.

Para construir um MIPS superescalar de grau 2, ou seja com a capacidade de despachar 2 instruções simultaneamente, modificamos a memória de instruções para a leitura simultânea de 2 instruções e o banco de registradores para leitura simultânea de 4 registradores, e escrita de 2 registradores. Construímos duas ULAs, uma para cálculo de endereços para as instruções de lw e sw e outra para demais operações aritméticas. Dessa forma podemos executar as instruções lw e sw em paralelo a demais instruções.

UNIDADE 8

1. Solução:

A interligação de um número ímpar de inversores formando um circuito como na fingura,não conseguem guardar um bit, pois quando o inversor mais à direita introduz o valor de saída para o inversor mais à esquerda, inverte o valor anterior de entrada desse inversor. Assim, todos os inversores terão os seus valores de entrada invertidos até chegar novamente ao inversor mais à direita, que também vai receber um valor de entrada inverso. Isso repete infinitamente. A resposta é não e um circuito de 3 (ou um número impar de) inversores resulta em um oscilador, pois medindo a saída de qualquer um dos inversores teremos sinais variando entre 0 e 1.

2. Solução:

Porque apesar do custo das memórias estáticas ser alto,a velocidade é de 5 a 10 vezes maior, o que compatibiliza o seu uso como memória cache.

3. Solução:

Devido ao armazenamento dos bits em forma de capacitores, cujo efeito de carga e descarga é lento em relação aos transistores usados nos inversores das memórias estáticas. O reavivamento é necessário exatamente porque os capacitores perdem as suas cargas com o tempo, portanto os mesmos devem ser recarregados.

4. Solução:

O princípioda localidade de referência refere-se à forma com que os processadores fazem referência à memória durante a execução de programas. A memória é fisicamente um vetor de palavras, porém, o processador não faz referência (endereçamento) a essas palavras de forma aleatória, mas sim de forma localizada, devido a existência de estruturas de dados, armazenamento de programas em endereços contíguos de memória, a execução sequencial de instruções, existência de laços em programas, etc.

Existem dois tipos de localidade de referência: 1) o primeiro tipo é a localidade espacial.A localidade espacial refere-se aos endereçamentos de palavras próximas fisicamente entre uma referência e outra.2) o segundo tipo é a localidade temporal. A localidade temporal refere-se aos endereçamentos das mesmas palavras referenciadas anteriormente.

5. Solução:

A memória principal é dividida em blocos de tamanho igual ao slot do cache, onde slot é o espaço no cache onde será carregado um bloco. O cache tem um certo número de slots, ordenados sequencialmente.Como a memória principal

é maior que o cache, em cada slot do cache deve ser carregado blocos diferentes da memória. O mapeamento direto é o tipo de mapeamento em que a cada slot consecutivo do cache são destinados os blocos consecutivos da memória para carregamento. Quando é atingido o tamanho máximo do cache, os próximos blocos consecutivos da memóriasão destinados a partir do slot inicial do cache, novamente. Assim, os blocos de endereços múltiplos do tamanho do cache são mapeados no slot inicial do cache, e os blocos subsequentes a esses endereços múltiplos são mapeados nos slots subsequentes do cache.

6. Solução:

O cache de mapeamento associativo por conjunto é diferente em relação ao mapeamento direto, devido a existência de mais do que um slot no cache associado a um blocode memória. Por exemplo, um cache associativo por conjunto de duas vias, tem dois slots que um determinado bloco de memória pode ser mapeado. Um cache associativo por conjunto de quatro vias, tem quatro slots.O circuito do sistema de mapeamento associativo fica um pouco mais complexo em relação ao mapeamento direto, porque quando é preciso descobrir se um determinado bloco está no cache, é preciso verificar todos os slots correspondentes a aquele bloco, pois o bloco pode estar em qualquer um dos slots.

7. Solução:

Significa um mapeamento onde todos os slots do cache estão disponíveis para todos os blocos da memória. Assim, quando é preciso verificar se um determinado bloco está no cache, é preciso consultar todos os slots do cache, pois o bloco pode estar em qualquer um dos slots. Portanto é o sistema de mapeamento mais complexo de custo elevado.

8. Solução:

O campo tag do endereço de memória fornecido pelo processador contém os bits que conseguem identificar se o bloco contido num slot do cache é realmente o bloco procurado. Para a verificação se um determinado bloco de memória está no cache, os bits do campo tag do endereço devem ser comparados com os bits do tag que estão contidos no diretório do cache. O diretório do cache é a parte do circuito do cache onde ficam as informações sobre o conteúdo. Toda vez que o cache recebe um bloco de memória o diretório é atualizado quanto ao slot que recebeu o bloco, com a informação de tag, e com a indicação de slot com bloco válido.

9. Solução:

O sistema de memória deve buscar na memória principal o bloco, carregar esse bloco no cache e disponibilizar a palavra que consta dentro do bloco para o processador. Caso no slot do cache onde o bloco deve ser carregado existir um bloco válido, esse bloco deve ser substituído. Caso esse bloco a ser substituído tivesse

sido escrito, e a memória principal não tivesse sido atualizada, a memória principal deve ser atualizada (write-back).

10. Solução:

Quando o processador faz uma referência de escrita a uma palavra, essa palavra deve ser escrita no cache, no bloco correspondente. Além disso, a palavra deve ser atualizada na memória principal, e nesse caso tem dois critérios: 1) write--through – em que a palavra na memória principal é atualizada a cada vez que é escrita no cache; 2) write-back – em que a palavra na memória principal só é atualizada quando o bloco correspondente no cache deve ser substituído.

11. Solução:

Uma implementação onde existem vários bancos de memória, e as palavras consecutivas estão contidas em bancos diferentes, é denominada de intercalação de bancos. Isso porque para se obter os dados de um determinado bloco referenciado pelo processador todos os bancos de memória devem ser lidos ao mesmo tempo e os resultados de leitura devem ser intercalados para formar o bloco.

12. Solução:

No sistema de memória virtual, os blocos são maiores que no cache, e tem tamanhos da ordem de alguns Kbytes e são denominados páginas. Se a cada escrita na memória física tivesse que atualizar o disco (write-through) o custo seria muito alto, portanto usa-se o write-back, em que o disco só é atualizado quando uma página da memória física deve ser substituída, e essa página tivesse sido escrita.

13. Solução:

Num sistema de memória virtual a memória é dividida em páginas, de tamanho da ordem de alguns Kbytes. Como existe um número muito grande de páginas, as mesmas ficam num sistema de armazenamento como o disco magnético. O processador quando faz uma referência a uma palavra, a página que contém essa palavra é carregada do disco para a memória principal, denominada de memória física. Isso porque um acesso à memória principal é mais rápido que um acesso ao disco. A página fica na memória física até que ela precise ser substituída, usando algum critério de substituição. Assim, as páginas mais referenciadas tem uma cópia na memória física. Para que o sistema saiba se uma determinada palavra referenciada tem uma cópia na memória física é usada a tabela de página. Nela contém informações da existência ou não daquela página virtual na memória física e do número do slot da memória física onde fica a cópia, se for o caso.

14. Solução:

a) O cache contém 4 palavras por bloco, portanto 16 bytes.

b) Cada palavra contém 4 bytes portanto, as palavras de endereços 0, 4, 8 e 12 são subsequentes e estão no mesmo bloco. Usando esse raciocínio, as palavras de endereços 0, 4, 16 e 64 estão nos slots de índices: 0, 0, 1, e 4, respectivamente.

c) O bloco de número 4096 corresponde ao 4096-ésimo bloco de memória. Como o cache contém slots numerados de 0 a 4095, o 4096-ésimo bloco é mapeado no primeiro slot do cache, ou seja, no slot de índice 0, com tag igual a 1. O tag seria 2 se o bloco fosse 8192-ésimo.

d) Lembrando que um bloco contém 4 palavras e cada palavra contém 4 bytes, o número de bloco deve ser multiplicado por 16 para se obter o endereço da primeira palavra do bloco de número 4096, portanto, 4096 x 16 = 65536.

15. Solução:

O vetor contem 20 palavras, em endereços subsequentes a partir de 16. Cada slot do cache contem um bloco de 4 palavras, portanto, são usados 20/4 = 5 slots do cache. No primeiro acesso ao vetor, a cada palavra lida, é carregado um bloco, e ocorre um erro. As 3 palavras subsequentes já estão no cache, portanto, 3 acertos. Subtotal = 5 erros e 15 acertos. Nos acessos subsequentes, ocorrem 20 acertos. Subtotal = 60 acertos. Total geral = 5 erros e 75 acertos.

16. Solução:

No primeiro acesso ao vetor A: 5 erros e 15 acertos. No primeiro acesso ao vetor B: 5 erros e 15 acertos. No segundo acesso ao vetor A: 5 erros e 15 acertos, pois o slot do cache do vetor B é o mesmo para o vetor A. No segundo acesso ao vetor B: 5 erros e 15 acertos. Total geral = 20 erros e 60 acertos.

17. Solução:

a) Esses blocos se carregam nos conjuntos de índice 0. Os respectivos tags são: 0, 1, 2 e 4.

b) Fazendo com que cada bloco contenha mais do que uma palavra.

c) Inserindo 4 multiplexadores 2x1 logo abaixo de cada um dos 4 slots do conjunto, para selecionar a palavra dentro do bloco. A saída desses 4 multiplexadores entram no mux 4 x 1 da figura.

18. Solução:

No primeiro acesso a A, ocorrem 20 erros e nenhum acerto. No primeiro acesso a B, ocorrem 20 erros e nenhum acerto. No segundo acesso a A ocorrem 20 acertos e nenhum erro, pois os slots ocupados por B, pertencem aos mesmos conjuntos de A, mas são diferentes. No segundo acesso a B, ocorrem 20 acertos e nenhum erro. Total: 40 erros e 40 acertos.

UNIDADE 9

1. Solução:

A técnica de polling é baseada na consulta feita pelo processador a um dispositivo de entrada/saída, sobre o seu estado (se já disponibilizou o dado na interface, por exemplo). Enquanto o dispositivo responde negativamente, o processador continua repetindo a consulta, até que o dispositivo responda positivamente, momento em que o processador faz a leitura do dado na interface correspondente. Caracteriza-se pela ocupação do processador com a função de consulta repetitiva ao dispositivo.

A técnica de interrupção usa um circuito especial adicionado ao processador para que os dispositivos periféricos possam sinalizar o processador quanto ao seu estado (se tem um dado a ser lido na interface, por exemplo). O processador quando recebe o sinal entra num estado de interrupção, que consiste em desviar do programa em execução, para executar uma rotina especial para tratamento do dispositivo, como por exemplo, a leitura do dado na interface. Após terminar a execução da rotina especial o processador volta ao programa desviado. Caracteriza-se pelo uso eficiente do processador, uma vez que evita do mesmo executar consultas repetitivas ao dispositivo de entrada/saída como no polling.

2. Solução:

A técnica de DMA é a forma mais eficiente de transportar uma quantidade grande de dados para a memória (ou da memória). Sem o uso do DMA, a forma de transportar dados para a memória (ou da memória) é usando instruções de entrada/saída, que normalmente transporta uma palavra por instrução para um registrador interno ao processador. Em seguida uma outra instrução deve guardar a palavra do registrador interno para a memória. A técnica de DMA consiste em usar um circuito especialmente projetado para transferir dados de um dispositivo (ou para um dispositivo) periférico como um disco magnético para a (ou da) memória principal do computador, sem a interferência do processador durante a transferência dos dados. Esse circuito especialmente projetado deve cuidar de endereçar a memória palavra por palavra e escrever (ou ler) os dados, enquanto faz a transferência. Ao mesmo tempo deve cuidar de ler (ou escrever) um buffer de dados que fornece (ou recebe) os dados transferidos. Após terminar a transferência de dados, o dispositivo de DMA deve sinalizar o processador usando a técnica de interrupção, sobre o fim da transferência.

3. Solução:

Para realizar uma leitura de um setor, a cabeça de leitura deve ser posicionado no início do setor, para que possa ser feita a transferência dos dados. Para o posicionamento, temos:

- Tempo de overhead de 0,4 ms +
- Tempo de busca da trilha de 5 ms +
- Tempo médio de rotação para atingir o início do setor de $(1 / 5000) \times 0,5 \times 60 = 6\ ms$
- Total para posicionamento = 11,4 ms.

Para a transferência temos 1 Kbytes a uma taxa de 100 Mbytes/s, portanto,

- Total para transferência = 1K bytes /100 Mbytes/ s = 0,01 ms.
- Tempo médio para leitura de um setor = 11,4 + 0,01 = 11, 41 ms.

4. Solução:

O chip ponte tem a função de interconectar os componentes do sistema Pentium 4: processador, placa gráfica, memória, barramento PCI e o controlador ATAPI.

5. Solução:

O barramento PCI-Express tem a característica de uso de um comutador que interliga o chip ponte aos periféricos usando o esquema de transmissão serial. O conceito de pacote de dados, com um cabeçalho e uma sequência de dados, é aplicado.

A interface serial está sendo preferida atualmente devido à limitação da velocidade de transmissão paralela pelo problema de diferença de atraso dos bits nas várias linhas paralelas. Na transmissão serial não existe problema de atraso dos bits, uma vez que existe apenas uma linha de transmissão.

SOBRE O AUTOR

José Hiroki Saito

O autor é engenheiro eletricista, graduado na Escola de Engenharia de São Carlos, Universidade de São Paulo, em 1972; mestre em Eletrônica Aplicada, pelo Instituto Tecnológico da Aeronáutica, em 1979; e doutor em Engenharia Elétrica, pela Escola Politécnica da Universidade de São Paulo, em 1983. Fez pós-doutorado na Universidade de Osaka, Japão, de 1997 a 1999. É docente do Departamento de Computação da Universidade Federal de São Carlos (DC-UFSCar) e da Faculdade Campo Limpo Paulista (FACCAMP), tendo atuado em ensino, pesquisa e extensão, na área de Arquiteturas de Computadores, Processamento de Imagens e Sinais, Redes Neurais Artificiais e Circuitos Reconfiguráveis.

Este livro foi impresso pelo Departamento de Produção Gráfica – UFSCar.