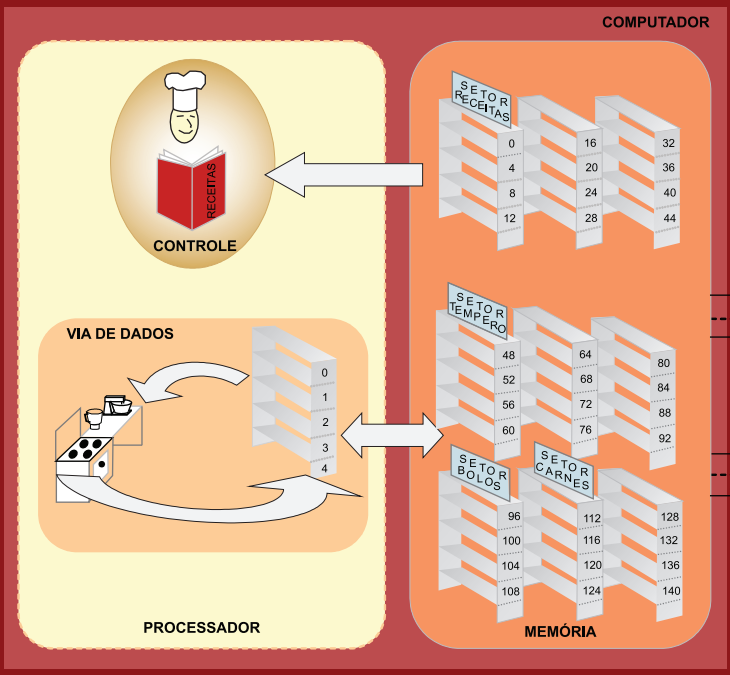
Arquitetura de Computadores

a visão do software



Eduardo Bráulio Wanderley Netto

CEFET-RN Editora

PRESIDENTE DA REPÚBLICA
LUÍS INÁCIO LULA DA SILVA

MINISTRO DA EDUCAÇÃO
FERNANDO HADDAD

SECRETÁRIO DE EDUCAÇÃO PROFISSIONAL E TECNOLÓGICA
ELIEZER PACHECO

DIRETOR GERAL
FRANCISCO DAS CHAGAS DE MARIZ FERNANDES

DIRETOR DA UNIDADE SEDE
ENILSON ARAÚJO PEREIRA

DIRETOR DA UNED-MOSSORÓ
CLÓVIS COSTA DE ARAÚJO

DIRETOR DE ADMINISTRAÇÃO E PLANEJAMENTO
JUSCELINO CARDOSO DE MEDEIROS

DIRETOR DE ENSINO
BELCHIOR DE OLIVEIRA ROCHA

DIRETOR DE PESQUISA
JOSÉ YVAN PEREIRA LEITE

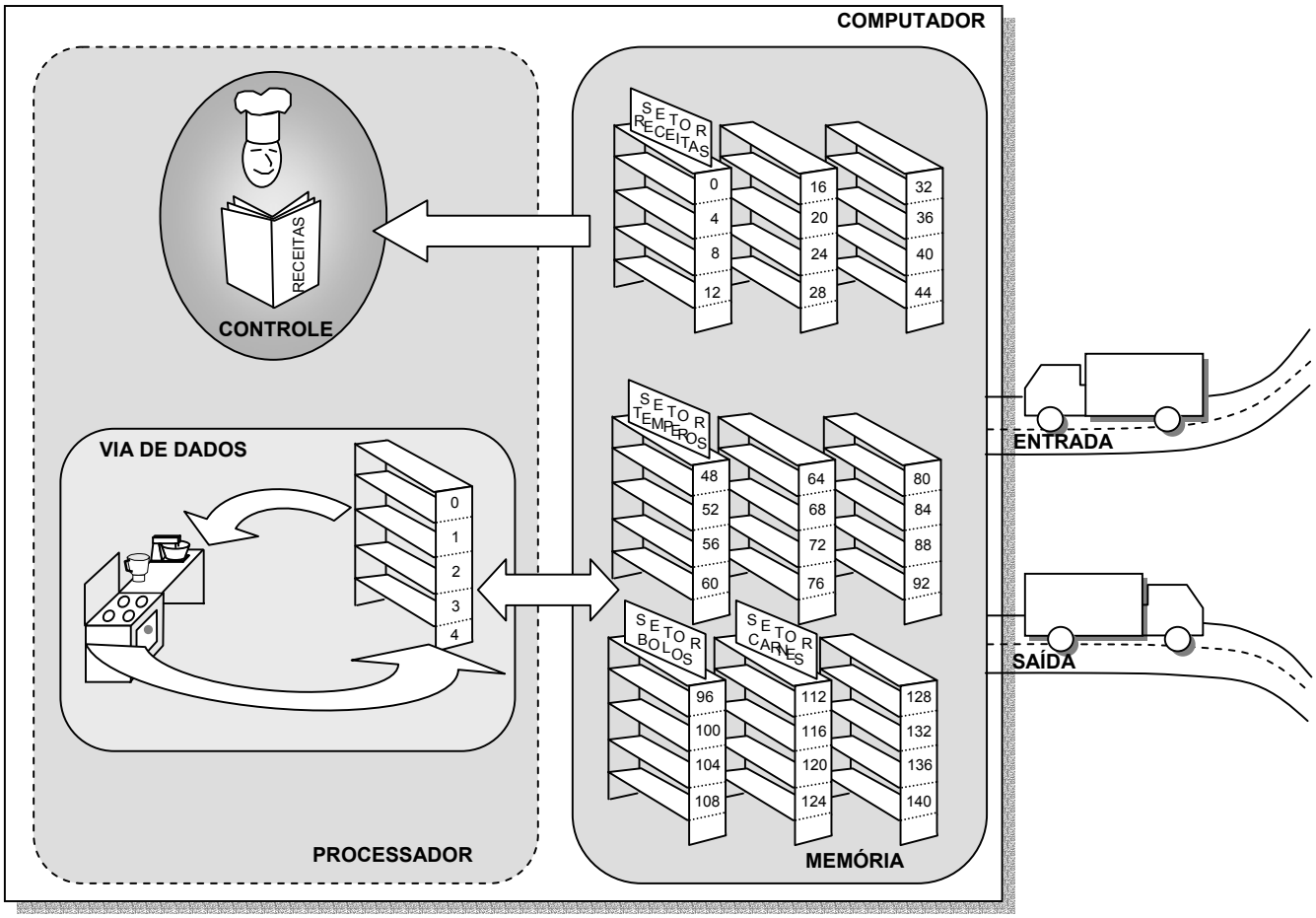
DIRETOR DE RELAÇÕES EMPRESARIAIS E COMUNITÁRIAS
LIZNANDO FERNANDES DA COSTA

GERENTE DE DESENVOLVIMENTO DE RECURSOS HUMANOS
AURIDAN DANTAS DE ARAÚJO

COORDENADOR DA EDITORA
SAMIR CRISTINO DE SOUZA

Arquitetura de Computadores

a visão do software



Eduardo Bráulio Wanderley Netto

http://www.cefetrn.br/~braulio/ACv1/

Divisão de serviços Técnicos
Catalogação da publicação na fonte.
Biblioteca Sebastião Fernandes (BSF) – CEFET/RN

Wanderley Netto, Eduardo Bráulio,
Arquitetura de Computadores: A visão do software/ Eduardo Bráulio Wanderley Netto – Natal: Editora do CEFET-RN, 2005
Xii, 216p.: il.

ISBN 85-89571-06-8

Organizador: Francisco das Chagas Silva Souza.

1. Arquitetura de computadores - Informática. 2. Organização de computadores - Informática . 3. Assembly - Informática . 4. Linguagem de montagem - Informática . 5. Linguagem de máquina - Informática .I. Título.

CDD – 004.22

CEFET/RN/BSF

ARTE DA CAPA
Tânia Carvalho da Silva

Editora do CEFET-RN
Av. Sen. Salgado Filho, 1559, CEP 59015-000
Natal-RN. Fone: (0XX84) 4005-2668, 3215-2733
E-mail: dpeq@cefetrn.br

À Esther e Elker
À minha Mãe e meu Pai (*in memorian*)
À minha família

Prefácio

1.1 – Introdução

O termo arquitetura de computadores tem sua origem em um antigo livro editado por Werner Buzhholz: ***Planning a Computer System: Project Stretch*** de 1962 [1]. Frederick Brooks introduziu o termo no capítulo intitulado "*Architectural Philosophy*". Em 1964 o projeto do system/360 também utilizou o termo, talvez com uma visão mais limitada. Quase duas décadas depois Jean-Loup Baer [2] revisita o termo e o define como um conjunto de conhecimentos que fazem parte da Arquitetura de um Computador: estrutura (modelagem estática das partes); organização (interação dinâmica das partes e o seu gerenciamento); implementação (projeto físico dos blocos construtivos específicos); e avaliação (o estudo do comportamento do sistema e do seu desempenho, seja em parte ou no todo).

Modernamente, entende-se o termo como sendo a visão que um programador deve ter da máquina quando o mesmo está utilizando uma linguagem de montagem e/ou de máquina [3].

À parte da discussão filosófica que envolve a definição do termo e do emaranhado de visões - nem sempre alinhadas - sobre o que significa "Arquitetura de Computadores", procuramos manter o título desta obra de forma expandida contendo o que exatamente se espera do texto: a visão do software sobre a máquina onde ele irá ser executado.

1.2 Motivação

A maioria do material didático sobre o tema [4, 5, 6] costuma apresentar detalhes de projeto em algum nível de abstração, muitas vezes com características tão próximas de uma implementação real (física) que os tornam muito enfadonhos para cursos com ênfase em desenvolvimento de software. Além disto, tais conhecimentos prescinde de requisitos curriculares em

sistemas digitais, o que nem sempre está presente em cursos de graduação curta, focados em seu saber específico. Naturalmente, o material didático atual, carece de um ecletismo particular, pois tem um público abrangente: desde cientistas da computação até engenheiros eletricistas (quando muito alinhados com o tema), mas certamente também todos aqueles que, de alguma forma, querem ou precisam conhecer a máquina que operam de forma mais intelectual.

Neste contexto, um material específico para cursos de graduação tecnológica com ênfase em software, trará os temas mais relevantes da área de Arquitetura de Computadores à luz sob este prisma, dando oportunidade para uma abordagem mais precisa e aprofundada, sem onerar nem avolumar a obra com materiais de interesse secundário.

Neste contexto, o título do livro também explicita seu público, sendo este primordialmente formado por alunos que galgam as primeiras cadeiras de cursos de graduação tecnológica em desenvolvimento de software.

1.3 Abrangência

Dentro da estrutura de cursos da Gerência Educacional de Tecnologia da Informação (GEINF), o primeiro destinatário da obra são os alunos que fazem o curso de Tecnologia em Desenvolvimento de Software. Em sua grade atual a disciplina Organização de Computadores tem os objetivos alinhados com a estrutura proposta para este livro. Ainda no âmbito da GEINF, os cursos técnicos de informática poderão adotar o material como auxiliar ou como tópicos avançados dentro de disciplinas correlatas. Também o curso de Tecnologia em Redes de Computadores (em fase de estudos, mas com perspectivas para seu início em 2006) poderá fazer uso do material.

Dentro do CEFET-RN também as novas disciplinas do curso de Automação poderão utilizar a obra.

Cursos de tecnologia em Desenvolvimento de Software, da rede de ensino tecnológico, assim como o Curso Superior de Tecnologia em Desenvolvimento de Aplicações para Web, do CEFET-RJ e do UFTPR, o curso de Tecnologia em Desenvolvimento de Software do CEFET-PB e o curso de Tecnologia em Sistemas de Informação do CEFET-PE, são exemplos de cursos que poderão utilizar em caráter efetivo o presente material. Afora os demais cursos correlatos que poderiam adotar o livro como cursos de Automação Industrial, Sistemas Distribuídos, etc. embora não seja o foco do trabalho.

1.4 Aspectos Pedagógicos

Existe um certo consenso na comunidade científica que a utilização de ferramentas computacionais de auxílio ao ensino produz um efeito positivo na aprendizagem do aluno. Especificamente, em Arquitetura de Computadores, alguns experimentos foram feitos e publicados [7, 8, 9]. Há divergências sobre quais ferramentas utilizar - comerciais ou didáticas - mas todos concordam que o auxílio é imprescindível ao ensino moderno.

Os livros atuais costumam apresentar um material auxiliar para o aluno explorar (simuladores, sintetizados, etc), mas raramente incentivam o seu uso corriqueiro. Este livro apresenta um conjunto de ferramentas desenvolvidas no NUARQ, dentro do projeto *Visual Architecture* – VA, com vistas ao ensino da disciplina. Desta forma, acreditamos envolver fortemente a pesquisa com o ensino, fortalecendo o seu elo de ligação, tão importante na conjuntura atual.

As ferramentas de simulação podem ser encontradas no *site* do livro: **http://www.cefetrn.br/~braulio/ACv1/**

1.5 MIPS

O MIPS é o processador alvo de nosso livro. Ele foi escolhido por fazer parte de um gênero de processadores simples, mas muito eficientes. A simplicidade nos permite aprofundar os conceitos sem torná-los ininteligíveis. Existem duas famílias de arquiteturas MIPS na nomenclatura atual: MIPS32 e MIPS64. Há também extensões como MIPS16 e outras. Vamos trabalhar com um subconjunto da arquitetura MIPS32, o que nos permitirá produzir pequenos softwares para execução em simuladores.

1.6 Agradecimentos

Escrever um livro didático é tentar aprisionar nas palavras o conhecimento. Isto requer um esforço de concentração muito grande e, portanto, uma devoção exclusiva à obra em andamento. Esta dedicação só me foi possível pela compreensão da importância deste trabalho por minha esposa e por minha filha. Não fora estas protagonistas, um manuscrito incompleto estaria ainda por aparecer. Meus primeiros agradecimentos, depois do Deus Pai, que nos permitiu tamanha travessura e ousadia, são para estas mulheres de minha vida. Não posso deixar de estender também aos meus familiares tal atitude de compreensão.

Em um outro flanco estão os alunos do Núcleo de Pesquisas em Arquiteturas de Computadores do CEFET-RN. Eles foram instigados a encontrar respostas desde cedo para problemas muito maiores que seus conhecimentos. Mas, ao final chegaram com trabalhos que muitas vezes me surpreenderam, bem como ao meu colega Prof. Jorgiano, do mesmo grupo, pelos resultados positivos alcançados.

Por fim, como a arte também precisa de sorte, ou como diria dentro de minha religiosidade, da mão de Deus, gostaria de agradecer aos meus alunos da disciplina de organização de computadores, em especial a turma 1.71.1V de 2005.2, que foram os primeiros a observar o material e apontaram diversos erros nos manuscritos. Esta turma é certamente uma das mais dedicadas que já pude ter o prazer de interagir.

Meus pais são figuras que transcendem aos agradecimentos, tudo me deram, dentro de suas limitações, e não mediram esforços para minha formação. Poderia faltar qualquer coisa dentro de casa, mas nunca faltou o amor e a dedicação deles para comigo e com meu irmão. Não é o material que faz o homem, mas o que ele ganha de valores em um mundo cada vez mais guiado pelo financeiro. A eles minha homenagem, não meu agradecimento!

Eduardo B. Wanderley Netto
Natal/RN, dezembro de 2005

1.7 Bibliografia Específica

[1] Werner Buzhholz, *Planning a Computer System: Project Stretch* , 1962

[2] Jean-Loup Baer, *Computer Systems Architecture,* 1980

[3] Gerrit Blaauw, Frederick Brooks, *Computer Architecture: concepts and evolution,* 1997

[4] Paterson & Henessy, *Computer Organization and Design, 3ed.*, 2004

[5] Tanenbaum, *Structured Computer Organization*, 4 ed., 1998

[6] Willian Stalling, *Arquitetura e Organização de Computadores*, 5ed. 1997

[7] Ney Calazans and Fernando Moraes, *Integrating the teaching of Computer Organization and Architecture with Digital Hardware Design Early in Undergraduate Courses*, **IEEE Transaction on Education Vol 44 no. 2 may 2001, pp109-119**

[8] W. Kleinfelde, D. Gray and G. Dudevoir, *A hierarchical approach to digital design using computer-aided design and Hardware description languages.* In **Proc. 1999 Frontiers Educ. Conf. Nov, 1999. pp 13c6-18~13c6-22**

[9] S. Rigo, M. Juliato, G. Araújo and P. Centoducatte*, Teaching Computer Architecture Using an Architecture Description Language.* **Workshop on Computer Architecture Education (WCAE), June 2004**

Índice

Capítulo 1

Introdução

1.1 – Introdução

O aluno que chega a cursar a disciplina alvo deste livro, costuma apresentar um certo grau de conhecimento sobre programação em alto nível (C, C++, JAVA e/ou outras), ou está exatamente recebendo estas informações. O objetivo deste capítulo é mostrar o que está por trás deste mundo, o que faz com que o programa que foi desenvolvido seja executado em um computador, os papeis do Compilador, do Montador, do Carregador e do Sistema Operacional, SO.

Inicialmente gostaríamos de tratar das múltiplas visões de um computador, ou seja, como nós enxergamos esta máquina que revolucionou nossas vidas. Fico me perguntando como podemos apreciar um *Da Vinci* de forma tão eclética. Seu mais famoso trabalho, a *Mona Lisa*, apresenta uma enorme variedade de interpretações. De fronte à galeria do Louvre que guarda a obra, muitos curiosos enxergam aquela mulher com um ar enigmático como uma obra prima, mas não compreendem a genialidade que existe por trás da tela. Assimetrias ao fundo, técnicas de *sfumato* e a identidade da egéria são apenas vistos por olhos mais curiosos e treinados.

Mas voltemos às nossas máquinas computacionais. Uma criança (e não somente ela) entende um computador como um grande repositório de jogos onde ela pode se divertir. Um usuário compreende um computador como uma ferramenta que lhe ajuda a digitar textos e encontrar informações na Internet. Um analista de sistema vê o que outros não enxergam: o computador como o local onde se materializam seus devaneios (ou dos outros) e de onde ele tira o seu sustento.

O convite neste livro é para irmos além. Abrirmos a máquina ainda é pouco. Vamos precisar radiografá-la, adentrar suas entranhas e descobrir um mundo onde tudo é minúsculo.

O que fazer com isto? Por que uma abstração tão detalhada para um aluno de um curso com ênfase em software? Iremos perceber ao longo deste livro que o usuário de nossos projetos deseja, antes de tudo, um desempenho razoável da aplicação. Imagine chamar (iniciar) um processador de textos e esperar meia hora para que ele esteja pronto para uso! Buscar um arquivo e ter de esperar minutos para que ele esteja disponível. Isto está fora de nossa realidade atual.

E o que faz com que nossos softwares sejam eficientes? É uma combinação de técnicas de **software** e de **hardware**. Usar estruturas de dados adequadas é tão importante como explorar bem o sistema de memórias. Além disto, algumas classes de software exigem um conhecimento profundo da arquitetura do sistema onde eles serão executados. Trata-se dos **softwares básicos** (***System Software***). Nesta classe de softwares estão incluídos montadores, ligadores, carregadores, compiladores e sistemas operacionais. Em breve apresentaremos uma visão geral de cada um deles.

Voltando às nossas abstrações, vamos apresentar como hardware e software podem ser enxergados em camadas. Desta forma definiremos exatamente o escopo deste livro, sobre que camadas há informações no texto. Vamos começar com o Hardware.

A Figura 1.1 mostra as principais camadas de abstração sobre as quais um projetista de hardware pode trabalhar. Na camada de DISPOSITIVOS, são estudados os materiais semi-condutores e suas dopagens com outros elementos para produzir um transistor, elemento básico de funcionamento de um processador. Na camada de CIRCUITO, os transistores são agrupados para formar um circuito. Na camada de PORTAS, uma operação lógica é materializada em um circuito particular. Na camada de MÓDULO, diversas portas são agrupadas para formar um módulo computacional. Finalmente, na camada de SISTEMA os módulos são interligados de tal forma a produzir um sistema, por exemplo, um processador. Para ajudar no processo de construção de um sistema eletrônico existem diversas ferramentas para auxílio a projetos, inclusive ferramentas que permitem a descrição no nível de sistema e que geram automaticamente os outros níveis da abstração. A propósito este processo de gerar níveis mais baixos de abstração do sistema é chamado de **síntese**.

Neste livro abordaremos as camadas de SISTEMA e MÓDULO, com eventuais usos de portas lógicas, para construir um processador simples. De fato, com milhões de transistores presentes em um único chip, em nossos tempos, é uma tarefa árdua, para não dizer impossível, projetar um sistema apenas pensando nos transistores, ou mesmo portas.

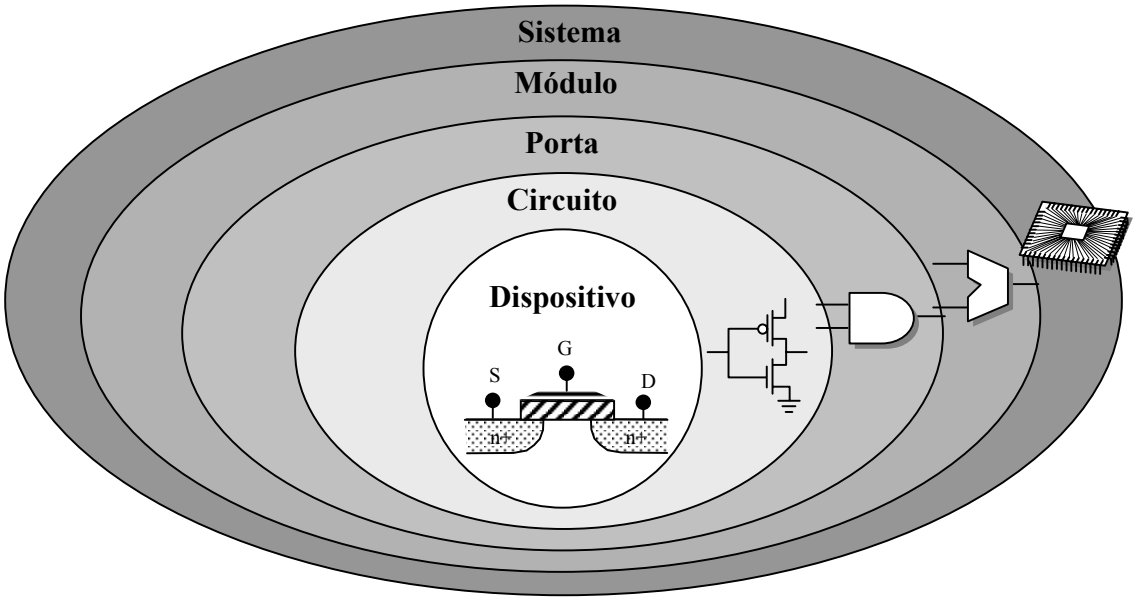


Figura 1.1: Abstração do Projeto de Hardware

Os softwares estão organizados, em relação ao hardware, quanto a sua proximidade da máquina. Um hardware é controlado pelo Sistema Operacional – que presta serviços às aplicações. Quando um programa de usuário requer alguma informação do hardware, ele a solicita ao Sistema Operacional e aguarda até que o SO possa trazê-la. A Figura 1.2 mostra as camadas de software que compõem um sistema computacional. De fato, esta organização em camadas reflete como diversas classes de softwares agem com relação ao hardware.

Um exemplo real é um programa processador de texto. Quando pressionamos alguma tecla no teclado, estamos de fato usando um módulo do hardware que avisa ao sistema operacional que o usuário digitou uma tecla. O SO, por sua vez, informa ao processador de texto que o usuário digitou um caractere. O processador de texto recebe este caractere e pede ao SO para mostrá-lo na tela. O SO envia para o monitor (hardware) o código do caractere a ser mostrado e o monitor o faz. Isto tudo ocorre de forma tão rápida que o usuário tem a sensação de que sua digitação provocou a imediata aparição do caractere na tela.

Um software nada mais é que um conjunto de instruções, logicamente organizado, de tal forma que o hardware possa interpretar e seguir os comandos. O hardware, por sua vez, é uma máquina que faz exatamente o que está escrito em um software. Acontece que o hardware usa uma forma de

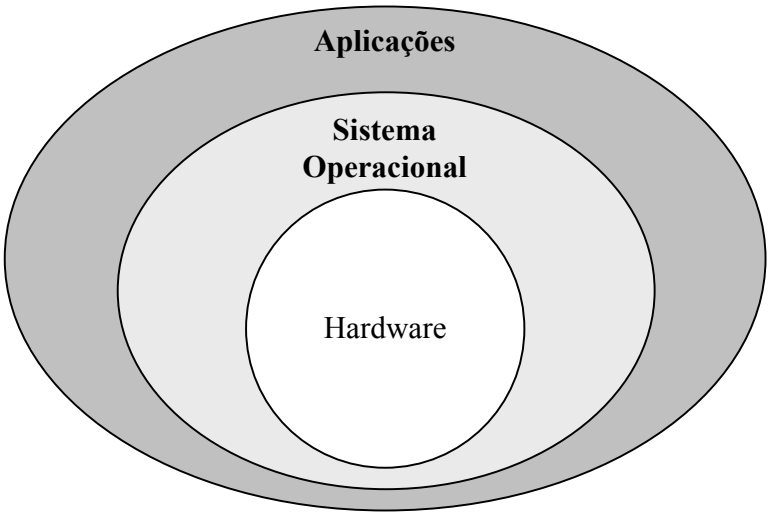


Figura 1.2: Camadas de Softwares

representação de dados diferente do nosso alfabeto convencional. Um hardware tem um alfabeto muito simples, com apenas duas letras: 0 e 1.

Assim como podemos escrever qualquer palavra combinando o nosso alfabeto de 23 ou 26 letras, o hardware também pode atribuir significado à combinações de zeros e uns. Cada letra do alfabeto eletrônico é conhecida como Dígito Binário, ***bit***. Os bits, em termos de circuitos elétricos, são tensões diferentes com padrões predeterminados. Por exemplo, o bit 0 pode ser associado a 0 volts enquanto o bit 1 pode ser associado a 5 volts.

Os comandos inteligíveis para uma máquina são palavras formadas por bits. Existem máquinas que aceitam palavras de tamanhos variados, enquanto outras, mais simples, só admitem um único tamanho de palavras.

Agora, imaginemos a tarefa simples de criar um conjunto de palavras binárias de 32bits que formem um programa. Todos os bits têm significados e devem ser observados. O programa poderia ser algo como:

```
00100010001100111111111111111111
00100010010101000000000000000100
01110010001100010000000000000010
```

Do ponto de vista prático, esta é uma programação praticamente ininteligível e de difícil manutenção. Nós, como seres humanos, dotados de inteligência dita superior, podemos criar símbolos para cada combinação de bits de tal forma que possamos programar com símbolos. Os símbolos mais comumente utilizados são chamados **mnemônicos**. O programa anterior seria então interpretado como

```
addi A, B, -1
addi C, D, 4
mul E, B, B
```

Ora, nesta forma de escrever, algum significado já fica exposto para um possível leitor: as duas primeiras instruções são somas e a terceira instrução é uma multiplicação. Por convenção, `addi A, B, -1` significa A = B + (-1). Por conseguinte, `addi C, D, 4` significa C = D + 4. Ora, mesmo neste princípio de leitura já podermos inferir como construir, de forma simbólica o comando referente a expressão C = A + 8. Tente fazê-lo!

O problema é que estes comandos, `addi` e `mul`, apesar de significativos para nós, não têm como ser expressos para a máquina senão usando bits. Assim, faz-se necessário um processo de conversão. A conversão pode ser manual ou utilizando um software capaz de 'traduzir' esta linguagem mais significativa para nós em uma linguagem que o computador possa entender. Este software é chamado de **Montador** (***Assembler***). Deve existir uma relação biunívoca entre o símbolo e o conjunto de bits correspondente. A linguagem formada pelo conjunto de mnemônicos de um determinado computador é chamada de ***assembly*** ou **linguagem de montagem**. Uma palavra interpretável pelo circuito de um processador é chamada de **instrução**. Podemos então dizer que um montador é um software capaz de converter instruções em *assembly* para instruções de máquina.

Existe ainda mais um nível comum de abstração para programação: linguagens de nível alto (HLL, ***High Level programming Languages***), ou de alto nível. Quando programamos em C ou Java estamos utilizando este outro nível de abstração. Em contrapartida, um programa escrito em *assembly* é dito de nível baixo, ou mais comumente, de baixo nível. Os comandos de uma linguagem de alto nível não têm uma relação unívoca com o *assembly*, ou seja, os comandos podem ser traduzidos de forma diferente dependendo da situação. A programação em HLL se abstrai quase que completamente da plataforma onde será executado o código de máquina gerado. Para traduzir um programa escrito em HLL para um *assembly* específico usamos um software chamado **Compilador**.

A linguagem de alto nível é normalmente independente da máquina para qual nosso programa será traduzido. Já não é o caso do montador. A Figura 1.3 mostra o caso geral para um trecho de programa em C/Java. Se usarmos um compilador para MIPS, o *assembly* gerado será específico do MIPS. O mesmo ocorre para SPARC ou IA32 (Pentium, por exemplo). Daí para baixo, é preciso um montador específico. O código binário (em bits) correspondente em cada máquina é chamado de **código de máquina** ou **linguagem de máquina**. As traduções apresentadas na figura são fictícias.

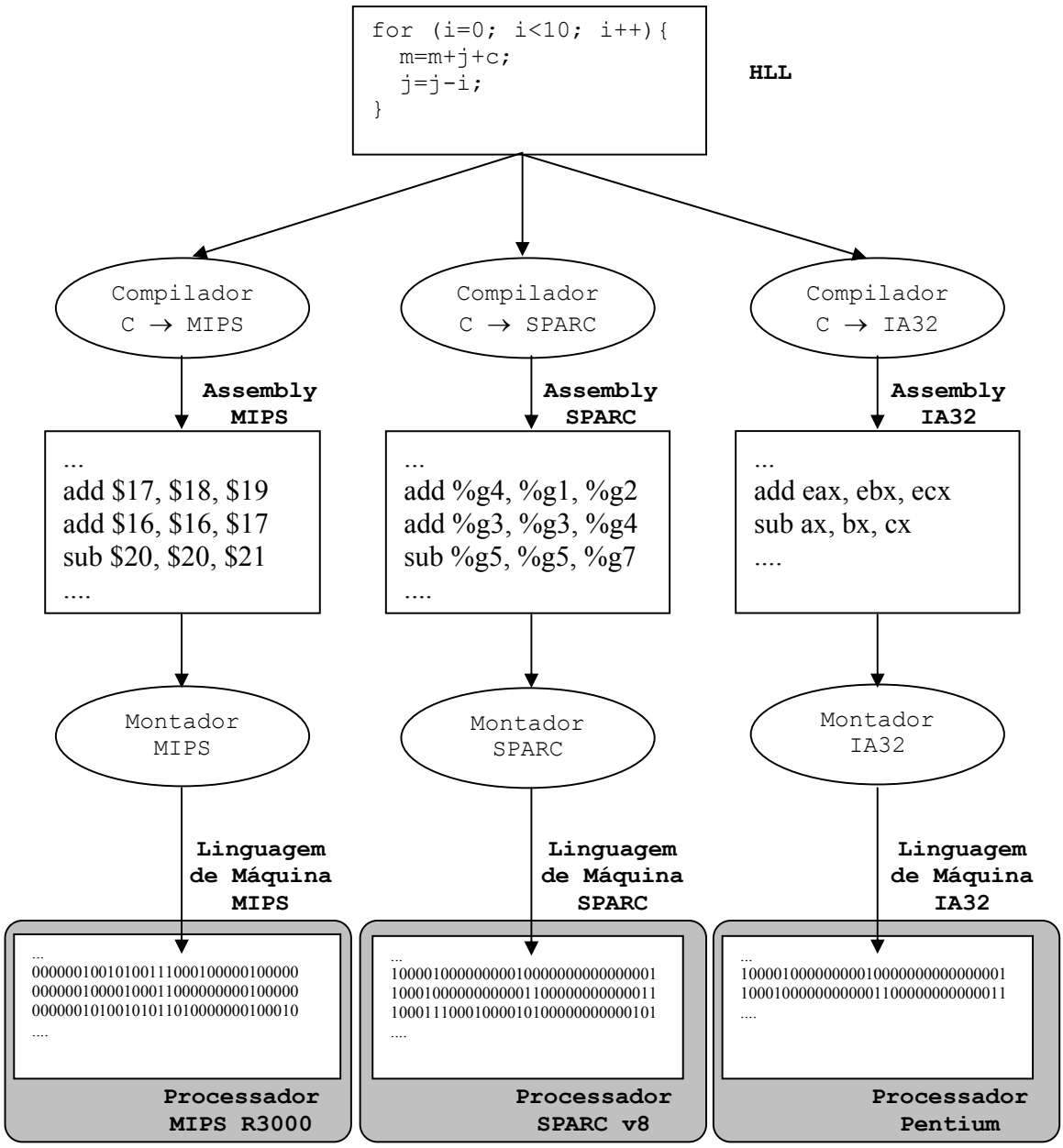


Figura 1.3: Compilação e Montagem de um código em HLL

1.2 – Componentes de um Computador

A abordagem clássica em arquitetura de computadores é apresentar a máquina com cinco componentes básicos. Isto remonta o modelo de computação preconizado por *Von Neumman* e usado até hoje. Neste modelo há um repositório para dados e programas que é a **memória**. Uma **via de dados** (***datapath***) por onde os dados da memória são trazidos para o processador, processados e devolvidos à memória. Uma **unidade de controle**, que gerencia a via de dados baseada no código (programa) que o usuário também depositou na memória. Finalmente um sistema de **entrada** e **saída** (**E/S** ou **I/O,** ***Input*/*Output***), por onde o computador se comunica com o mundo externo. Costumamos chamar a unidade de controle junto com a via de dados de **CPU** (***Central Processing Unit*** ou UCP, Unidade de Processamento Central) ou **processador**. A Figura 1.4 mostra um diagrama dos componentes básicos de um computador e suas inter-relações.

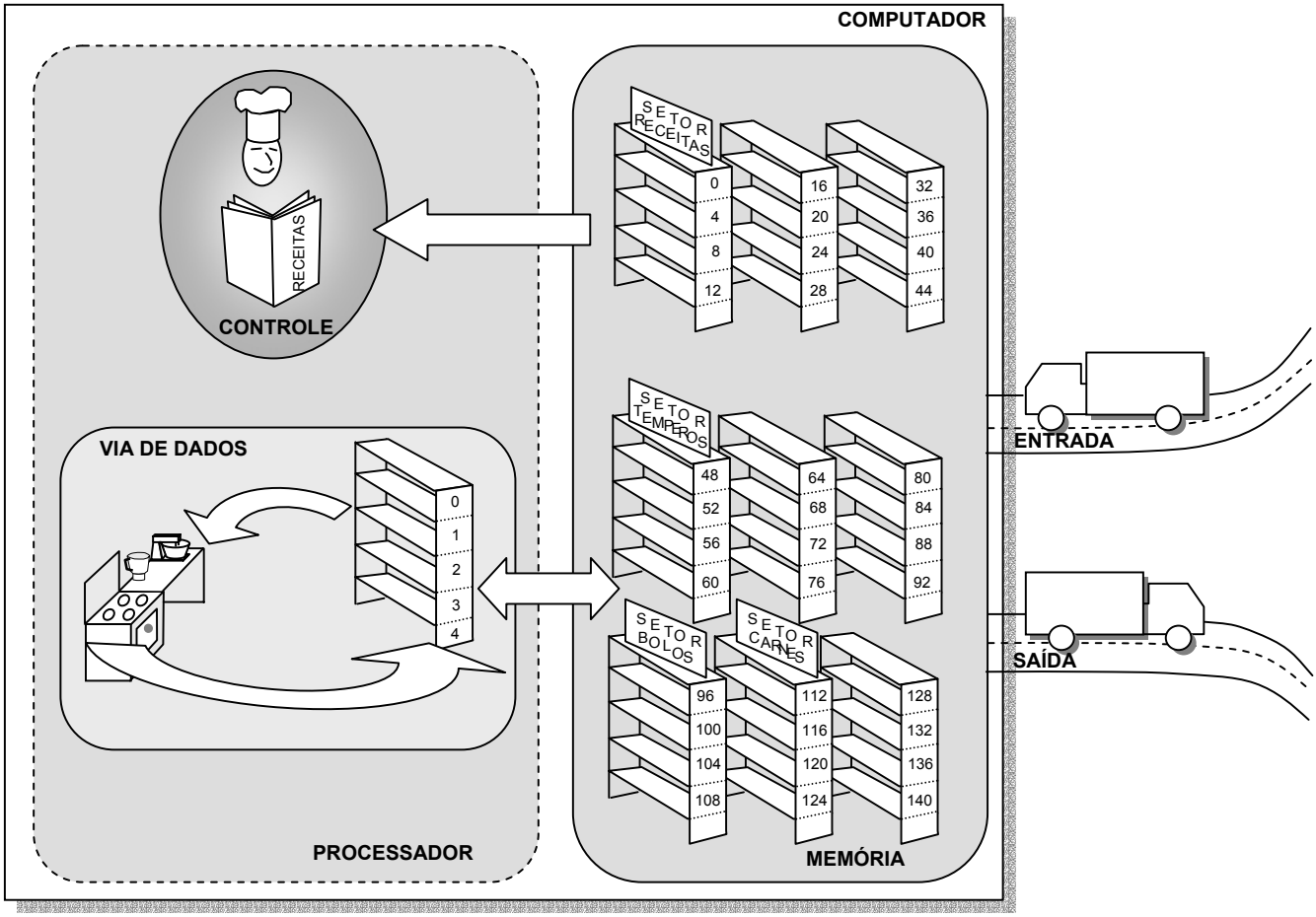


Figura 1.4: Componentes de um Computador

A Figura 1.4 apresenta os componentes de um computador, mas possui também algumas figuras bem peculiares. Trata-se de uma analogia para melhor entender como o computador funciona. Como toda analogia, sempre há imperfeições, mas o raciocínio desenvolvido a seguir ajuda a limitar e direcionar nossa imaginação.

O processador é uma fábrica de bolos. A memória, um supermercado. Dentro do processador está o chefe de cozinha. Ele é o responsável pelo bom funcionamento da cozinha. É ele quem dá as ordens para mistura dos ingredientes. A prateleira, dentro de nossa fábrica de bolos, serve para guardar os ingredientes comprados no supermercado e também os bolos que vão ficando prontos. Esta prateleira, no mundo real, é chamada de **banco de registradores**. O conjunto de utensílios e forno é chamado de **Unidade Lógica e Aritmética**, **ULA** ou **ALU** (***Arithmetic and Logic Unit***).

A memória é um grande supermercado. Nele temos muitas prateleiras. As prateleiras do supermercado são numeradas para que os ingredientes possam ser facilmente encontrados. Existem setores específicos para cada ingrediente no supermercado. O chefe de cozinha adquire no supermercado um livro de receitas e nele estão mostrados os ingredientes e modo de fazer. A receita é análoga ao **programa** (software). Nela está mostrado o modo de preparar e os ingredientes. O chefe, então, instrui seus comandados a comprarem no supermercado os ingredientes e depois lhes informa como misturar. Depois de preparado, o bolo vai para a despensa (prateleiras). O chefe então manda entregar ao supermercado, onde os bolos serão vendidos.

O supermercado, por sua vez, recebe dos produtores, através de uma via de entrada, os ingredientes para abastecer suas prateleiras. Quando existe uma compra em atacado, o supermercado manda entregar os ingredientes e produtos usando uma via de saída. Um exemplo típico é o teclado, que usa a via de entrada de dados e o monitor, que usa a via de saída de dados. Existem dois caminhos entre o processador e a memória: uma para códigos e outro para dados. Estes caminhos são chamados de **barramentos**. Assim como as estradas de entrada e saída, que também são barramentos. Em algumas máquinas existe apenas um barramento para dados e instruções, o que faz com que os mesmos concorram ao recurso. O barramento de dados é bidirecional, ou seja, tanto traz dados da memória, como os leva de volta. O barramento de códigos é unidirecional, dado que o supermercado não aceita livros de receitas usados.

Mais alguns detalhes de nossa analogia: as prateleiras dos supermercados são do mesmo tamanho das prateleiras da despensa na fábrica de bolos. Embora seja possível comprar apenas um ingrediente de uma das prateleiras do supermercado, é comum (veremos porque mais tarde) a

transferência de uma prateleira inteira da memória para o banco de registradores. Os registradores são numerados seqüencialmente a partir do zero. Neste livro vamos utilizar uma despensa com 32 prateleiras. As prateleiras no supermercado também são numeradas, começando do zero, mas, embora sejam também seqüenciais, as numerações salta de quatro em quatro unidades. Em cada prateleira cabem exatamente quatro ingredientes distintos. O ingrediente é a unidade indivisível de informação. No nosso caso usamos um conjunto de 8 bits para o ingrediente. Este conjunto é chamado de **byte**. Então, em cada prateleira existem 4 bytes, o que nos faz concluir que cada prateleira tem capacidade de 32 bits.

Em nosso modelo não admitimos que ingredientes sejam comprados no supermercado e vão direto para o fogão/forno, nem que bolos vão direto ao supermercado. Esta é uma classe de máquinas chamada ***Load*/*Store*** ou **registrador-registrador**. Existem máquinas que flexibilizam esta restrição, permitindo que ingredientes possam vir direto do supermercado (**máquinas memória-registrador**). Existem também máquinas onde o bolo fica em um único registrador especial chamado **Acumulador** (**máquina baseada em acumulador**). Ainda, existem máquinas onde os ingredientes ficam acumulados em uma pilha, em vez de um banco de registradores (**máquina de pilha**, ***stack machine***). Nós estudaremos neste livro a classe de máquinas *load*/*store*.

Ainda dentro de nossa analogia, podemos derivar um importante aspecto das organizações de um computador: o **desempenho**. Perceba que é muito mais trabalhoso deslocar os ingredientes do supermercado para fábrica e então usá-los do que guardar em nossa despensa os ingredientes mais importantes e tão logo seja necessário, imediatamente utilizá-los. Isto também vale, por extrapolação, para quando um produto falta no supermercado. Ai, é preciso esperar ainda mais até que o próximo caminhão de entrega saia do produtor e chegue ao supermercado.

O problema em guardar ingredientes é que a nossa despensa tem uma quantidade de prateleiras limitada, portanto nem todos os ingredientes de um bolo podem estar ao mesmo tempo na despensa. Além disto, as receitas que devem ser preparadas são comandadas pelo usuário, então não há como saber quais os ingredientes mais importantes a serem guardados. Às vezes, algumas receitas exigem o preparo em duas fases e precisamos guardar na despensa a massa para descansar. Isto tudo ocupa lugar na despensa e normalmente apenas buscamos os ingredientes que serão utilizados mais rapidamente para não ocuparmos muitas prateleiras. Em nossa fábrica, o chefe de cozinha só admite a mistura de dois ingredientes por vez, ou seja, nossa ALU opera com dados em 32 bits sempre.

Embora seja ruim em termos de vendas, o supermercado pode, a bem da velocidade de quem faz as compras, reservar uma região no seu espaço físico, onde os ingredientes mais comumente utilizados por seus clientes estejam disponíveis, todos juntos, em prateleiras adjacentes. Isto otimizaria o tempo de quem precisa comprar uma lista de ingredientes enorme. No mundo real dos computadores, esta região do supermercado é chamada de **memória cache**. Nela, o que importa não é a variedade, mas a velocidade, então tudo é otimizado, por construção, para que as compras saiam mais rápidas.

Em uma receita típica de nossa fábrica, primeiro estão especificados os ingredientes e em seguida como misturá-los. È um passo-a-passo. Precisa ser seguido rigorosamente ou a receita não tem o resultado desejado (o bolo não cresce). O seu análogo, o programa, normalmente também é assim: as variáveis são declaradas inicialmente e em seguida vem o código dizendo como as variáveis são processadas. Então, em um programa existe uma região de dados e uma região de código. Normalmente a região de código é apelidada ***text*** e a região de dados ***data***.

Uma outra extrapolação do modelo refere-se à capacidade de armazenamento do supermercado. De fato, esta capacidade é limitada em relação aos grandes silos dos produtores. Mas é muito mais rápido encontrar um produto no supermercado da esquina que esperar que o produtor o transporte até nossa residência. Esta relação de forças entre velocidade e capacidade é muito comum nos sistemas computacionais. Um silo de um produtor pode ser comparado ao **disco rígido** de um computador, o **HD** (***Hard Disk***).

È preciso entender que estamos apenas fazendo analogias. Tudo que foi dito é válido, mas nem tudo que podemos pensar sobre a nossa figura é verdadeiro. Antes de afirmar cabalmente como acontece algo é importante ler o material sobre o assunto, disponível neste livro e em outros da literatura especializada.

1.3 – A visão do software

Ao longo deste livro uma seção especial sempre se faz presente: a visão do software. Estamos descrevendo hardware, mas com enfoque em software, por isto, mesmo que o assunto seja muito voltado ao engenheiro de computação e afins, sempre devemos apontar como o software, mais precisamente o programador, enxerga a máquina e/ou quais softwares podem ser utilizados para extrair o máximo da organização de um computador.

Dentro deste espírito, vamos definir o termo ***Arquitetura de Computadores****, como a visão que um programador de baixo nível tem sobre a máquina para qual ele está codificando. Este programador enxerga então uma abstração da máquina chamada* ***Arquitetura do Conjunto de Instruções****, ISA (***Instruction Set Architecture***). Ela define como é possível fazer a nossa máquina funcionar.*

O ***Conjunto de Instruções*** *de uma máquina é simplesmente o conjunto de palavras que compõem o idioma inteligível ao processador e outras partes da organização do computador. Algumas destas instruções podem afetar a bom funcionamento da máquina, o que, no passado, levou até à perdas de partes do hardware. Para evitar tal situação, algumas das instruções foram selecionadas, na maioria das arquiteturas, para compor um conjunto de instruções privilegiadas. Este conjunto é rigorosamente controlado pelo Sistema Operacional, ou seja, um software que deseja fazer uso de tais instruções precisa de permissão especial. Quem constrói um Sistema Operacional precisa, por sua vez, prover os serviços necessários à proteção da máquina e dos processos que nela são executados. Um software que roda sobre o sistema operacional é dito software do usuário.*

Vamos de forma introdutória mostrar as atribuições básicas do Sistema Operacional. Depois seguiremos com a apresentação de outros softwares básicos.

1.3.1 – O Sistema Operacional

O Sistema Operacional é o software básico de maior importância dentro do sistema computacional. Ele controla as funções básicas do computador, incluindo o gerenciamento de memória e o I/O. O Sistema Operacional é um software baseado em eventos, isto é, ele realiza tarefas em respostas à comandos, à chamadas de programas de usuários, de dispositivos de Entrada e Saída etc.

Os sistemas operacionais modernos têm aparência parecida, mas suas funcionalidades podem diferir muito. Uma possível abordagem é a minimalista, onde apenas as funcionalidades mais básicas estão presentes no SO. Outra abordagem é a completa, onde toda e qualquer funcionalidade deve estar presente. Alguns têm uma interface com usuário bastante amigável, mas falham em desempenho ou confiabilidade. Outros são bastante confiáveis, mas não têm uma boa interface. Em suma, nenhum sistema operacional é completo e melhor que os outros em todos os aspectos.

Uma das principais características dos sistemas operacionais é sua capacidade de gerenciar a execução de diversos softwares do usuário ao mesmo tempo.

Dois componentes são importantes no projeto de um SO: o **kernel** *e os* ***programas de sistema****. O* kernel *é o coração do SO. Ele é responsável pelo escalonamento de tarefas, sincronização, proteção/segurança, gerenciamento de memória e interrupções. O* kernel *tem controle total da máquina, conhece seus registradores, temporizadores, palavras de* status *etc. Muitos são adeptos do microkernel, ou seja, de um kernel com mínimas funcionalidades. Isto transforma boa parte da funcionalidade do SO como um todo em programas de sistema o que permite, praticamente, reiniciar as funcionalidades da máquina sem precisar desligá-la. A alternativa ao microkernel é o SO monolítico, onde todas as funcionalidades necessárias ao funcionamento da máquina estão em um único e grande processo.*

À parte desta discussão, talvez o mais interessante mecanismo do sistema operacional seja a ilusão de óptica que ele nos envolve ao usarmos múltiplas tarefas ao mesmo tempo. Em um sistema com uma única tarefa, a ocupação do processador pode chegar a ser mesmo muito pequena, pois a maior parte do tempo ele está esperando um dado ou algo parecido. Quando executamos mais de uma tarefa, o tempo do processador é dividido (de forma muito sensata, com técnicas especializadas) para os dois processos. Como a velocidade de execução de cada trecho da tarefa é muito alta, nós temos a sensação de estarmos utilizando os dois processos ao mesmo tempo. Isto leva a necessidade de sincronização e controle de recursos. A sensação para um processo, entretanto, é que ele dispõe da máquina inteira apenas para si.

Quando iniciamos um computador (ligamos o botão ON/OFF) o processador é inicializado e depois começa a executar o kernel *do SO. Aí, este toma conta do restante do processo de inicialização. Normalmente ele culmina com a apresentação da Interface gráfica para o usuário. Daí o usuário vai iniciar seus próprios programas, o que significa na prática, solicitar ao Sistema Operacional, espaço para executar aquela tarefa. Vamos ver adiante, que um programador segue um caminho alternativo: ele precisa, depois de confeccionar seu programa, usar um compilador e um montador para gerar um código executável.*

1.3.2 – Outros Softwares Básicos

Já mostramos a idéia geral que está por trás do compilador. Ele tem a tarefa de transformar um código em HLL para um código assembly*. Normalmente uma linha de código em linguagem de alto nível se traduz por*

uma seqüência de linhas de código em assembly, assim programar em alto nível aumenta a produtividade.

No passado, muitos compiladores e sistemas operacionais eram construídos em assembly. *Hoje, a presença de compiladores otimizantes nos permite criar outros compiladores e sistemas operacionais a partir de HLLs. Certamente que os pontos críticos destes softwares básicos são investigados com mais atenção, em assembly, para diminuir qualquer imperfeição (normalmente não na funcionalidade, mas no desempenho) no código gerado.*

O montador também é um software básico que nós já citamos. Ele basicamente tem a missão de gerar um código binário que será carregado na memória do computador para ser executado. O montador, ou melhor, o projetista do montador, precisa conhecer profundamente a máquina em que o software gerado vai ser executado.

O montador também gera um ***tabela de símbolos****, que representam endereços do programa. Nós estudaremos mais adiante as instruções de salto, o que nos fará entender de onde vêm estes símbolos.*

À medida que os programas foram se tornando muito grandes, uma modularização passou a ser importante. Isto significa que podemos construir módulos separados de um programa, compilá-los independentemente e em seguida ligá-los para gerar um código executável. A modularização nos permite criar um conjunto de funções muito usadas e deixá-las compiladas em uma biblioteca. Quando um software do usuário precisa de uma função ele simplesmente liga ao seu código a biblioteca. Um software básico chamado de ***ligador (linker)*** *é o responsável por esta tarefa.*

Finalmente o código executável é gerado. A partir daí é preciso um ***carregador*** *para pôr este código na memória e assim ele ser executado. O carregador é um outro software básico, mas ele normalmente é um módulo específico do SO. Gostaríamos de destacar aqui, mesmo que seja em forma de repetição, que um programa para ser executado ele precisa estar na memória. Este é um conceito muito antigo, chamado conceito de* ***programa armazenado (*****stored program concept***). Veja que em nossa figura da analogia dos componentes do computador, as receitas utilizadas pelo chefe de cozinha vêm dos livros que estão no supermercado, na seção de receitas. Sem estes livros ele não é capaz de misturar os ingredientes para surtir o efeito almejado.*

A Figura 1.5 mostra a inter-relação entre compilador, montador, ligador e cerregador.

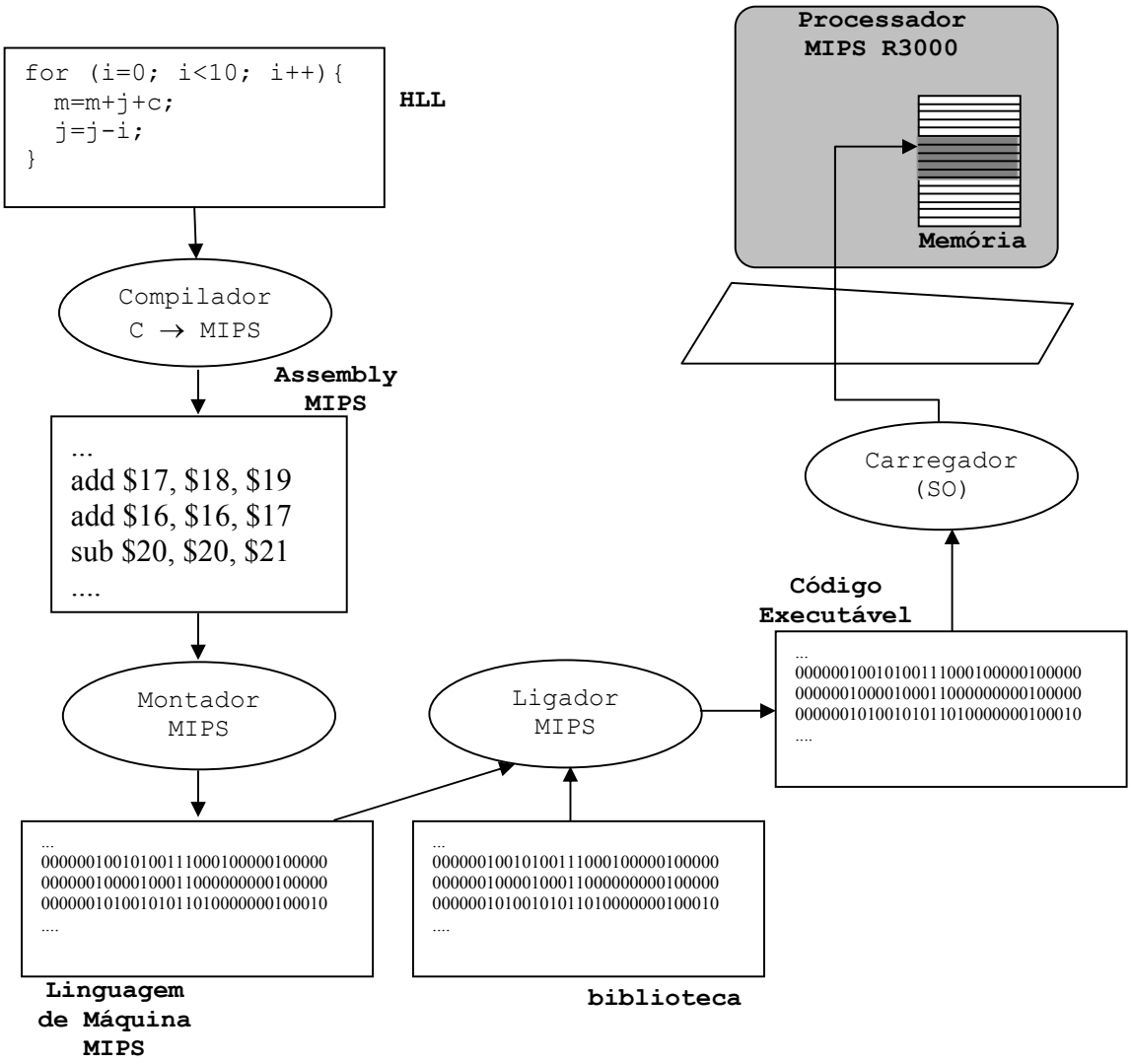


Figura 1.5: Esquema do Compilador/Montador/Ligador/Carregador

*Na ligação apresentada anteriormente, chamada estática, todo o código da biblioteca é anexado ao programa do usuário, mesmo que ele vá utilizar apenas algumas funções. Isto faz com que o programa final fique muito grande. Existe uma outra espécie de ligação muito freqüentemente utilizada: a ligação dinâmica. As bibliotecas de ligação dinâmica, DLLs (***Dynamic Linked Library***) são módulos que são invocados pelo programa do usuário durante sua execução. Isto faz com que o carregador não precise alocar todo o código na memória. As DLLs podem então ser carregadas apenas no momento de seu uso. Outra coisa interessante relativa às DLLs é que um software não precisa ser re-ligado sempre que uma nova DLL (que corrige um defeito, ou que melhora o desempenho) é disponibilizada. O ônus*

natural com as DLLs é que o tempo de ligação é transferido para dentro do tempo de execução do processo.

1.3.3 – A Máquina Virtual Java

A máquina Virtual Java é um software do usuário capaz de interpretar ***bytecodes*** *Java. O termo interpretar é um pouco diferente de executar. A execução de um código só é feita pela máquina real. Os códigos interpretáveis são códigos binários de uma máquina que não existe, idealizada em software e que são inteligíveis para um software simulador de uma arquitetura. A Figura 1.6 mostra como são realizadas a compilação e execução de um código em Java. O compilador Java transforma o programa em bytecodes e gera uma arquivo .class. Os arquivos .class podem ser executados em qualquer plataforma (combinação de Processador e SO), desde que haja uma* ***Máquina Virtual Java****,* ***JVM*** *(Java Virtual Machine) instalada. As bibliotecas Java, também em bytecodes, podem ser carregadas diretamente pela JVM sob demanda.*

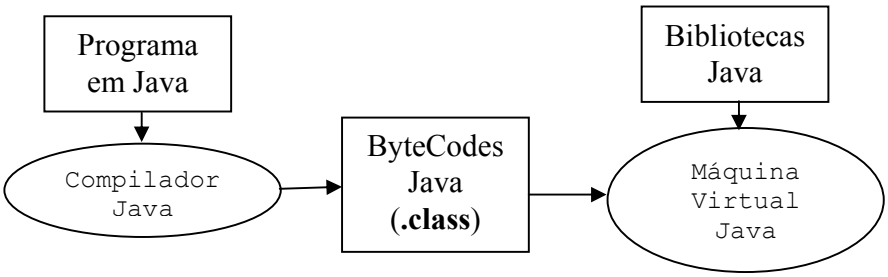


Figura 1.6: Compilação de um código em Java

O sucesso de uma máquina virtual só pode ser alcançado se existirem implementações para muitas plataformas, o que torna a portabilidade do código muito atrativo. Existe, porém, uma limitação importante: por melhor que seja a máquina virtual, a velocidade de interpretação é 10 vezes mais lenta que uma execução com código nativo. Isto levou ao desenvolvimento de um módulo do interpretador com capacidade de converter bytecodes em linguagem de máquina. São os compiladores just-in-time, *JIT (****Just-In-Time compilers****). Toda vez que o interpretador reconhece um trecho de código que é freqüentemente utilizado ele usa o JIT para convertê-lo em código nativo e da próxima vez que for chamado o trecho, ao invés de ser interpretado, ele é executado. A Figura 1.7 mostra a integração do JIT com os demais*

componentes do processo de criação e execução de um software em Java. O interpretador invoca o JIT que converte métodos em códigos executáveis. Quando forem invocados na próxima vez, estes métodos podem ser executados diretamente na arquitetura onde está rodando a JVM.

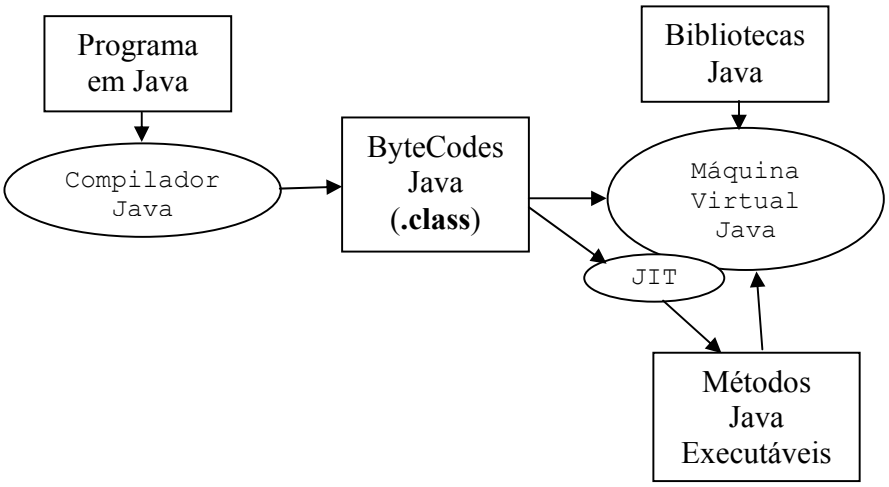


Figura 1.7: Máquina Virtual Java com JIT

1.4 – Conclusões

Vimos que um computador é uma máquina complexa, mas totalmente controlável e determinística[1]. Os componentes básicos de um computador são: memória, controle, via de dados, entrada e saída. Todos os programas em alto nível construídos pelo usuário precisam passar por um processo de conversão para a linguagem da máquina correspondente. Os compiladores, montadores e ligadores, usados em conjunto geram um código que é capaz de ser executado na máquina. Um sistema operacional é um software que blinda o hardware de tal forma a prover controle sobre os processos que são executados. Nele existe um módulo carregador que é encarregado de transferir os programas para a memória, a partir de onde eles podem ser executados. O escopo deste capítulo apresenta o nível de abstração que vamos utilizar, ou seja, um nível de

[1] O processo de construção dos chips por vezes nos levam a algumas imperfeições na implementação de um processador o que permite um certo grau de incerteza na execução dos códigos, mas isto é minimizado pelo processo de qualidade e de testes por onde passam os processadores fabricados.

Arquitetura de Computadores, que envolve o conhecimento mínimo de organização da máquina para dela podermos extrair o máximo de desempenho possível.

Este trabalho está organizado da seguinte forma: o Capítulo 2 introduz o conjunto de instruções do MIPS, uma arquitetura simples e de fácil inteligibilidade, contudo de alto desempenho. O Capítulo 3 descreve como a linguagem *assembly* do MIPS é traduzida para código de máquina e vice-versa. O Capítulo 4 apresenta o hardware capaz de executar nossos programas. Algumas abordagens de como melhorar o desempenho de uma máquina são traçadas. O Capítulo 5, é a extensão natural do 4. Nele os aspectos de desempenho são catalogados e a importância dos testes é realçada. O Capítulo 6 traz toda a teoria sobre o sistema de memória usado convencionalmente nos processadores atuais. O Capítulo 7 mostra como os diversos componentes de um computador são interligados. Por fim, as conclusões são apresentadas no Capítulo 8, bem como o caminho que está por vir, o que esperar dos computadores nas próximas décadas.

Dois grandes capítulos formam a base da leitura: o Capítulo 2 e o Capítulo 6. Todos os capítulos têm um conjunto de exercícios de fixação incluindo temas de pesquisa na web que complementam a leitura do texto.

1.5 – Prática com Simuladores

Existem inúmeros simuladores disponíveis para uso conjunto com os textos deste livro. Vamos utilizar basicamente o SPIM e o dinero, ferramentas amplamente divulgadas na comunidade acadêmica, mas também utilizaremos nossos próprios simuladores, que tentam serem ainda mais didáticos e estão disponíveis para modificação de forma simples. O *site* do livro traz todos os *links* para os simuladores utilizados.

1.6 – Exercícios

1.1 – Pesquise na web e produza um resumo sobre *Firmware*.
1.2 – Pesquise na web e produza um resumo sobre *Plug and Play*.
1.3 – Pesquise na web e produza um resumo sobre *boot process*.
1.4 – Pesquise na web e produza um resumo sobre *BIOS*.
1.5 – Pesquise na web e produza um resumo sobre periféricos.
1.6 – Pesquise na web e produza um resumo sobre *cross compilers*.
1.7 – Pesquise na web e produza um resumo sobre *pseudo-instruções*.
1.8 – Pesquise na web e produza um resumo sobre *a* história dos processadores.

1.9 – Pesquise na web e produza um resumo sobre Máquina Virtuais.
1.10 – Pesquise na web e produza um resumo sobre manufatura de chips.
1.11 – Pesquise na web e produza um resumo sobre RISC.
1.12 – Pesquise na web e produza um resumo sobre CISC.
1.13 – Crie uma figura como a 1.1 onde serão mostrados os níveis de abstração dos programas: HLL, Linguagem de Montagem e Linguagem de máquina.
1.14 – Crie uma figura como a 1.2 onde serão mostradas as camadas de software que contemplem os arquivos.*class*, a JVM, o SO e a máquina.

1.7 – Referências Bibliográficas Específicas

Este livro tem como referências básicas textos consagrados da Organização e Arquitetura de Computadores, porém o material foi cuidadosamente trabalhado para ser apresentado ao leitor com um enfoque bastante realista em termos de software. Uma pitada da experiência deste autor também é expressa e bibliografias particulares são apresentadas ao final de cada capítulo.

[1] Jean-Loup Baer, *Computer Systems Architecture,* 1980
[2] Gerrit Blaauw & Frederick Brooks, *Computer Architecture: concepts and evolution,* 1997
[3] David Paterson & John Henessy, *Computer Organization and Design,* 3ed., 2004
[4] Tanenbaum, *Structured Computer Organization*, 4 ed., 1998
[5] Willian Stalling, *Arquitetura e Organização de Computadores*, 5ed. 1997
[6] Linda Null & Julia Lobur*, The essentials of computer organization and architecture.* 2003
[7] Mostafa Abd-El-Barr, Hesham El-Rewini, *Fundamentals of computer organization and architecture*, 2005

Capítulo 2

Linguagem de Montagem

2.1 – Introdução

Já vimos os fundamentos da programação de computadores, sob o ponto de vista da inteligibilidade dos comandos. Sabemos também que a programação em baixo nível, principalmente em linguagem de máquina, é difícil e de compreensão árdua para nós seres humanos, embora seja inteligível para a máquina.

Neste capítulo vamos conhecer os comandos de linguagem de montagem que nos permitem escrever programas para serem executados no MIPS. Nossa abordagem busca aprender os principais comandos divididos por categorias funcionais: aritmética; transferência de dados; suporte à tomada de decisão; e a procedimentos.

Vamos iniciar pensando um pouco em forma de analogia: aprender um idioma, de uma máquina ou não, requer conhecer sua palavras. Um dicionário é um guia importante para quem está aprendendo as primeiras palavras. Assim como um idioma, uma linguagem de montagem também sofre constantes transformações, normalmente levando a um novo produto no mercado. Um exemplo não muito distante foi a introdução do conjunto MMX nas máquinas Pentium.

A expressividade de um idioma é medida pelo número de verbetes conhecidos da língua. Não se pode afirmar aqui que um indivíduo falante do português, com um vocabulário de 15.000 palavras, seja alguém que não tenha muitas habilidades de comunicação. De fato um adulto usa entre 13.000 e 14.000 palavras no cotidiano e um dicionário pequeno contem 150.000 verbetes. Para termos uma idéia, o vocabulário possível de um computador, que usa palavras de tamanho fixo de 32 bits, é de apenas 4.294.967.296 palavras. Nada mal, não é mesmo? Entretanto, no caso da máquina, existe uma necessidade de tornar esta expressividade uma realidade, ou seja, é preciso

fazer com que um hardware possa executar estas instruções e que exista um compilador/montador eficiente para gerar o código de máquina correspondente. Sob este prisma, a realidade é bem diferente. Vamos aprender ao longo deste texto que a **simplicidade favorece o desempenho**, então o conjunto de instruções que utilizaremos é de cerca de uma centena de palavras com suas flexões.

Esta máxima da arquitetura de computadores foi levantada em tempos remotos, mas ainda hoje é uma realidade. Nós poderíamos utilizar qualquer conjunto de instruções para nosso texto, entretanto, escolhemos o do MIPS por sua simplicidade. Talvez o conjunto de instruções mais convencional e mais estudado seja o da família 80x86, mas para um curso introdutório, o do MIPS nos permite adentrar em detalhes da organização que seriam intratáveis com uma arquitetura da complexidade de um Pentium.

Quando a organização de um computador está em fase de projeto, o formato do conjunto de instruções deve ser determinado o quanto antes. Esta é uma decisão duradoura, visto que uma nova arquitetura, mesmo que melhor que sua antecessora, implica em perda dos softwares que foram projetados anteriormente. Esta **compatibilidade de software** é talvez uma das grandes limitações para o surgimento de novas arquiteturas. Imagine que a Intel descontinuasse completamente o suporte aos softwares atuais, em nome de uma arquitetura que permitisse um ganho de desempenho de 30%. Ora, nenhum grande usuário estaria interessado em perder todo o seu investimento em software para ganhar um pouco de desempenho na máquina. Esta compatibilidade de software é tão forte, que hoje o Pentium IV é enxergado como uma máquina escalar (no máximo uma instrução é executada por vez) pelo software, mas no fundo ela traduz os comandos para um outro patamar, onde mais de uma instrução possa ser executada por vez (máquina super-escalar).

A eficiência de um conjunto de instruções pode ser medida de diversas formas: o tamanho que o programa vai ocupar; a complexidade da decodificação pela máquina; o tamanho das instruções; e o número de instruções. O MIPS utiliza um conjunto cujas instruções são de tamanho fixo, de 32 bits. Isto é bom para decodificação, mas um programa ocupa muito espaço na memória. A propósito, a organização da memória afeta diretamente o formato das instruções. Vamos trabalhar com uma memória que utiliza palavras de 32 bits, mas endereçável a bytes, o que significa que cada byte tem uma posição (endereço) particular. Isto implica que palavras adjacentes diferem de 4 unidades de endereço na memória.

Nas seções seguintes iremos tratar de um tópico eminentemente relacionado à visão do software. Por isto o restante deste capítulo está destacado.

2.2 – A visão do software – Operandos e Operadores

O nome computador nos leva a pensar de imediato em uma máquina que computa, ou seja, calcula. Certamente uma forte ênfase em cálculos é desejada em um computador, o que implica que ele precisa saber realizar operações aritméticas. Ora, uma operação aritmética é composta de ***operadores*** *e* ***operandos****. Os operadores básicos são: soma, subtração, multiplicação e divisão. Nós utilizamos parcelas em pares para guardar os operandos. As parcelas recebem nomes em particular, por exemplo, numa subtração existe o minuendo, o subtraendo e o resto (ou diferença), que é o resultado da operação. Na soma, as parcelas são denominadas 'parcelas'. Enfim, mesmo quando estamos realizando uma soma com três parcelas, nós fazemos as contas primitivas com dois algarismos, invariavelmente. Esta abordagem é seguida pela máquina em seus comandos aritméticos.*

Genericamente falando, uma instrução é composta de um operador e zero ou mais operandos, sendo mais comum quantidades menores ou iguais a três. No MIPS este valor máximo é dois, ou seja, no máximo cada operação pode utilizar dois operandos por vez.

Uma outra característica importante é o local de armazenamento dos operandos. Em nossa analogia, assinalada na Figura 1.4, os operandos (ingredientes) estão sempre guardados na despensa (banco de registradores). Eles, então, são misturados dois a dois para produzir o resultado de uma receita. No MIPS, os operandos de uma instrução são sempre guardados no banco de registradores e os resultados das operações (quando existem), também são endereçados aos registradores. Todavia, existe uma classe de instruções que porta um dos operandos dentro da sua própria codificação binária. São os chamados ***operandos imediatos****. Esta é uma exceção à regra geral, mas muito comumente utilizada. Então, um dos operandos pode vir de um registrador ou da própria instrução. Veja que isto não altera o número de operandos, mas apenas de onde eles vêm. Apenas um operando pode estar contido na codificação de uma instrução. Voltando a nossa analogia, o exemplo seria: quando o chefe de cozinha deseja comprar uma nova receita ele vai ao supermercado e a traz para a fábrica. Ele aproveitaria esta ida ao supermercado para trazer também algum ingrediente junto com ele.*

Os operandos podem ser de tipos e tamanhos diferentes, por exemplo: bytes, palavras, meias palavras (halfword), caracteres, números sinalizados,

números não sinalizados, endereços etc. No MIPS, a ALU só realiza operações com operandos de 32 bits, entretanto os operandos, principalmente imediatos, têm tamanhos diferentes, o que faz com que seja necessário um ajuste para 32 bits antes dos mesmos serem enviados para a ALU.

Um dado em binário não carrega consigo qualquer informação semântica, ou seja, uma seqüência de zeros e uns não indica se tratar de um número sinalizado, um número não sinalizado, uma cadeia de caracteres ou um endereço. Quem dá significado ao operando é a operação. Uma operação que usa operandos sinalizados vai interpretar seus operandos como sendo números sinalizados. Uma outra operação, que usa operandos não sinalizados vai interpretar os mesmos, como números não sinalizados. Por exemplo, o número binário $11111111111111111111111111111110_2$ *pode ser interpretado como -2 ou 4.294.967.294. Isto só depende de qual operação o utiliza.*

Finalmente, existem classes de operações: lógica e aritmética; transferência de dados; suporte à tomada de decisão; e suporte a procedimentos. Estas classes têm um certo impacto em como cada instrução será executada pelo processador.

2.3 – A visão do software – Operações lógicas e aritméticas

*Estudaremos nesta seção as instruções lógicas e aritméticas. Soma, subtração, multiplicação, divisão, e (*and*), ou (*or*), ou exclusivo (*xor*), nou (*nor*),* shift left *e* shift right*. Não trataremos o suporte a operandos em ponto-flutuante (uma representação de máquina para números reais).*

A mais intuitiva operação aritmética é a soma. No MIPS uma soma poderia ser escrita da seguinte forma:

```
add a, b, c
```

isto significa `a = b + c`*. Os operandos possuem ordem predeterminada em uma operação: a primeira variável a aparecer numa soma é, de fato, associada ao resultado da operação, a segunda e terceira variáveis são as parcelas. 'Variáveis' pode ser intuitivo para um programador em alto nível, mas quem trabalha com linguagem de montagem sabe que elas estão associadas a registradores. Já mencionamos que as operações do MIPS são realizadas sobre operados e que os mesmos estão, via de regra, em registradores. Então, quem faz esta associação: registrador - variável? Esta é uma das tarefas do compilador, chamada* ***alocação de registradores****. Por*

enquanto vamos deixar esta discussão fora do nosso caminho e vamos assumir que alguém (ou algo) fez esta associação. Já vimos que os registradores são numerados de 0 a 31, então, vamos utilizar alguns deles para associar nossas variáveis. Por convenção utilizamos os registradores numerados entre 8 e 25. Veremos mais tarde o arrazoado que existe por trás desta convenção. Bem, então vamos reescrever o código acima (de apenas uma instrução) com as seguintes associações: registrador 8 = variável `a`; registrador 9 = variável `b`; e registrador 10 = variável `c`. O código seria então:

```
add $8, $9, $10
```

Veja que os números dos registrados são precedidos de um caractere $ para não restar dúvidas que se trata de um operando que está no banco de registradores e não de um operando imediato.

Um dos problemas encontrados com esta forma de codificar é sua rigidez. Aqui não há espaços para interpretações dúbias. Devemos escrever exatamente de forma inteligível para o montador sob pena de nosso código de máquina gerado não realizar a tarefa desejada. Agora vamos analisar uma expressão aritmética, comum em HLLs, para sabermos como ela pode ser escrita em linguagem de montagem. A expressão seria:

```
a = b + c + d + e
```

Considerando as variáveis `a a e` associadas aos registradores 8 a 12 respectivamente, podemos escrever:

```
add $8, $9, $10  # a = b + c
add $8, $8, $11  # a = a + d => a = (b+c) + d
add $8, $8, $12  # a = a + e => a = (b+c+d) + e
```

Veja que foi necessário fracionar a soma em diversas fases, tomando proveito da propriedade da associatividade da soma. Surgiu também na codificação uma espécie de ajuda após a instrução. O símbolo # indica para um montador que o restante da linha deve ser menosprezado. Isto permite ao programador explicitar sua idéia para um colega de trabalho e/ou para ele mesmo, quando precisa lembrar exatamente como um problema foi resolvido. Usamos estes campos de comentários para explicitar as variáveis, já que as variáveis foram associadas a registradores anteriormente a confecção do código.

Um outro detalhe: não é possível escrever duas instruções em uma única linha de comando. Cada linha de código precisa conter no máximo uma instrução.

Vamos agora aprender a segunda instrução aritmética: a subtração. Por analogia a subtração tem como operador a palavra **sub** *e os operandos são dispostos exatamente como na soma. Então,*

```
sub $8, $9, $10  # $8 = $9 - $10 ou a = b - c
```

significa que a variável associada a $8, por exemplo `a`*, recebe o valor da subtração do minuendo $9, associado a* `b`*, pelo subtraendo $10, associado a variável* `c`*. Assim realizamos a seguinte operação aritmética:* `a = b - c`*.*

Vamos a um exemplo mais elaborado:

```
a = b + c - (d - e)
```

Esta expressão pode ser quebrada em `b + c` *e* `d - e` *e depois os resultados intermediários subtraídos para encontramos o resultado final. A codificação (com as variáveis* `a` *a* `e` *associadas a $8 a $12 respectivamente) seria:*

```
add $8, $9, $10   # a = b + c
sub $13, $11, $12 # temp = d - e
sub $8, $8, $13   # a = a - temp => a = (b+c) - (d-e)
```

Veja nesta solução que precisamos usar um registrador a mais ($13) para guardar um valor intermediário de nossa expressão. Naturalmente, com o nosso conhecimento aritmético poderíamos evitar o uso deste registrador extra.

Mas o que é mais importante neste instante é percebermos que muitas vezes precisamos armazenar valores temporários em registradores. Isto restringe ainda mais o uso de registradores para armazenar as variáveis de um programa. Se um determinado programa tem mais de 32 variáveis (ou, por convenção, mais de 18 variáveis, que é a porção dos registradores disponíveis para uso das variáveis) então alguns valores de variáveis precisam ser guardados na memória. Quando eles são necessários para realização de uma expressão eles são buscados para dentro do banco de registradores e operados normalmente. Vamos ver mais à frente como podemos trocar dados entre a memória e o banco de registradores.

Continuando com nossas expressões aritméticas, vamos introduzir agora mais uma possibilidade de soma. É comum em HLLs encontramos

expressões que somam uma constante a uma expressão, por exemplo: `a = a + 1` *(que é igual a* `a++`*). Podemos então escrever em uma única instrução de soma tal expressão (considerando a variável* `a` *associada ao registrador $8):*

```
addi $8, $8, 1    # a = a + 1
```

Nesta instrução, somamos o valor do registrador a um número chamado imediato. Ele não está armazenado no banco de registradores, mas sim na codificação binária da própria instrução.

Neste ponto precisamos definir os valores máximos de cada operando. Considerando que a nossa ALU só opera números de 32 bits, podemos usar números naturais (não sinalizados) no espaço entre 0 e 4.294.967.295 ou números inteiros (sinalizados) entre –2.147.483.648 e +2.147.483.647. Naturalmente quando operamos com números imediatos, este limite cai para algum valor menor, dado que os 32 bits disponíveis para codificar a instrução são usados não somente para a especificação da operação, mas também são reservados alguns bits para o próprio valor imediato. No caso do MIPS este valor vai de 0 a 65.535 para números naturais e de –32.768 a +32.767 para números inteiros. Para valores de constantes maiores que os limites especificados é preciso armazenar um dado na memória ou construí-lo em parcelas. Veremos como mais tarde.

Bem, agora já conhecemos os limites que podemos trabalhar. Uma informação ainda ficou de fora: como saber se estamos operando números naturais ou inteiros. Existem instruções especiais para operar com números naturais. Uma soma de números naturais seria especificada assim:

```
addu $8, $9, $10    # a = b + c
```

Isto significa que os valores contidos nos registradores $9 e $10 serão tratados como números naturais pela instrução. É preciso um cuidado redobrado ao utilizarmos valores próximos aos limites de armazenamento de dados, isto porque é de responsabilidade do programador cuidar para que o resultado caiba no registrador de destino especificado. Sabemos que freqüentemente uma adição envolvendo dois números naturais de 4 algarismos nos leva a um resultado com cinco algarismos. Ora, sabendo que cada registrador tem apenas 32 bits, o resultado de nossas operações precisa caber nestes 32 bits.

Uma conta que tem como resultado um número maior que 32 bits irá gerar um erro chamado de **overflow**. *Erros de* overflow *podem ser ignorados pela máquina e ai o resultado vai ser mesmo errado. Toda a confiabilidade*

do código depende do programador. Quando um erro de overflow *não é ignorado pela máquina, ela gera o chamado a uma* ***exceção****. Trataremos deste assunto mais tarde, mas ao menos, saberemos que a nossa conta gerou um erro.*

Instruções que operam com números naturais não geram exceções e instruções que operam com números inteiros sempre geram exceções. Portanto, `addu` *não gera exceção e* `add` *gera.*

Por extensão temos ainda a instrução `subu`*. Por exemplo:*

```
subu $8, $9, $10   # a = b - c
```

que realiza uma subtração com números naturais. Mais uma vez lembramos que `subu` *não gera exceção e o programador precisa se precaver para evitar erros em seus programas.*

Finalmente nos deparamos com a instrução `addiu`*. Esta instrução opera com números naturais inclusive sendo um dos operando um valor imediato. Por exemplo, uma constante 4 poderia ser adicionada a um registrador $8 associado a uma variável* `a` *da seguinte forma:*

```
addiu $8, $8, 4   # a = a + 4
```

A instrução `addiu` *não gera exceção quando ocorre* overflow.

Vamos sumarizar na Tabela 2.1 as instruções que conhecemos até o presente momento.

Categoria	Nome	Exemplo	Operação	Comentários
Aritmética	add	add $8, $9, $10	$8 = $9 + $10	*Overflow* gera exceção
	sub	sub $8, $9, $10	$8 = $9 – $10	*Overflow* gera exceção
	addi	addi $8, $9, 40	$8 = $9 + 40	*Overflow* gera exceção Valor do imediato na faixa entre –32.768 e +32.767
	addu	addu $8, $9, $10	$8 = $9 + $10	*Overflow* não gera exceção
	subu	subu $8, $9, $10	$8 = $9 – $10	*Overflow* não gera exceção
	addiu	addiu $8, $9, 40	$8 = $9 + 40	*Overflow* não gera exceção Valor do imediato na faixa entre 0 e 65.535

Tabela 2.1: Operações Aritméticas do MIPS

Nosso próximo passo é conhecer as instruções de multiplicação e divisão. A implementação de uma multiplicação em hardware não é tão simples como um somador/subtrator. A multiplicação costuma ser muito demorada para ser executada e devemos tentar evitá-la ao máximo. A divisão é ainda mais lenta e complexa. Entretanto, não vamos simplesmente abdicar

delas, mas usá-las com restrições, onde elas não podem ser substituídas por expressões mais simples.

Vamos começar pensando em uma multiplicação de dois números de 4 algarismos cada. O resultado desta multiplicação ficará, provavelmente, com 8 algarismos. Isto significa que ao multiplicarmos um número com ***n*** *algarismos por outro com* ***m*** *algarismos, o resultado será um número com* ***n+m*** *algarismos. No caso do MIPS, a operação de multiplicação será realizada sobre dois números de 32 bits. Isto implicará em um resultado de 64 bits. Ora, nenhum registrador tem 64 bits, então para tornar esta operação exeqüível, dois registradores extras foram criados: HI e LO, ambos de 32 bits. HI e LO, usados em conjunto, propiciam que resultados de 64 bits sejam armazenados nele. Assim, a multiplicação tem um destino fixo e obrigatório: o par HI, LO. Por isto ao multiplicarmos dois números precisamos apenas especificar quais os registradores que os guardam. A operação é* `mult`*. Exemplo:*

```
mult $8, $9   # HI, LO = $8 x $9
```

O registrador HI guarda os 32 bits mais significativos (mais à esquerda) do resultado, enquanto o registrador LO guarda os 32 bits menos significativos (mais à direita). Em nenhuma ocasião é tratado overflow. *O desenvolvedor do software deve prover mecanismos suficientes para evitá-lo.* `mult` *considera os dois operandos como sendo números sinalizados. Vamos supor o exemplo acima, onde $8 contem o valor 16 e $9 contem 2.147.483.647. O resultado da multiplicação, em binário, com 64 bits, seria:*

0000 0000 0000 0000 0000 0000 0000 0111 1111 1111 1111 1111 1111 1111 1111 0000$_2$

O registrador HI receberia então:

0000 0000 0000 0000 0000 0000 0000 0111$_2$

e o registrador LO receberia:

1111 1111 1111 1111 1111 1111 1111 0000$_2$*.*

Se o valor de $8 fosse –16, o resultado final seria:

1111 1111 1111 1111 1111 1111 1111 1000 0000 0000 0000 0000 0000 0000 0001 0000$_2$

ou seja, HI receberia 1111 1111 1111 1111 1111 1111 1111 1000_2 e LO receberia 0000 0000 0000 0000 0000 0000 0001 0000_2.

O hardware multiplicador realiza uma tarefa muito parecida com a qual fazemos na escola fundamental. Para operar com números sinalizados, ele simplesmente encontra os valores absolutos e faz a multiplicação. Depois ele analisa os sinais. Se eles divergirem significa que o resultado é negativo, caso contrário o resultado é positivo.

Há ainda uma outra operação de multiplicação que interpreta seus operandos como números não sinalizados. Trata-se da instrução `multu`. `multu` *opera exatamente da mesma forma que* `mult`*, a menos do tratamento dos sinais. Exemplo:*

```
multu $8, $9   # HI, LO = $8 x $9
```

As operações de multiplicação, `mult` *e* `multu` *não fazem qualquer tratamento de* overflow. *Mais uma vez repetimos: cabe ao programador determinar se existe risco na realização das multiplicações.*

Finalmente, existe uma instrução, `mul`*, que opera uma multiplicação como se fosse uma soma ou subtração, ou seja, usa dois registradores e coloca o resultado em um terceiro registrador do banco de registradores. Esta multiplicação é realizada normalmente e o resultado esperado ocorre com 64 bits. Aí, somente os 32 bits menos significativos são transferidos para o registrador especificado. A instrução* `mul` *deve ser usada com cuidado, pois os valores de HI e LO são imprevisíveis (dependem da implementação do hardware multiplicador) e os resultados só apresentam alguma semântica se os operandos forem números pequenos, cujo resultado da multiplicação caiba em 32 bits. Estas restrições normalmente são verificadas em nossos softwares, ou seja, o multiplicando e multiplicador costumam produzir resultados menores que os 32 bits ofertados pelo hardware, mas os casos específicos devem ser tratados pelo programador. A sintaxe de* `mul` *está explicitada no exemplo a seguir.*

```
mul $8, $9, $10   # $8 = $9 x $10
```

Como as demais instruções de multiplicação, `mul` *não gera exceção quando ocorre overflow.*

A divisão é a próxima operação a ser estudada. Existem duas instruções de divisão no MIPS, uma para números sinalizados e outra para números não sinalizados. Como sabemos quando realizamos uma operação de divisão inteira (e/ou natural), existem dois resultados, o quociente e o resto. Diferentemente da multiplicação, quando dividimos duas quantidades de 32 bits, não podemos garantir que quociente e/ou resto sejam quantidade representáveis com menos de 32 bits. Assim, no MIPS, usamos o mesmo par LO, HI para guardar o quociente e o resto da divisão de dois números, respectivamente. As instruções seguem a sintaxe do exemplo a seguir:

```
div $8, $9     # HI = $8 mod $9, LO = $8 div $9
divu $8, $9    # HI = $8 mod $9, LO = $8 div $9
```

`div` opera sobre números sinalizados e `divu` sobre números não sinalizados. `div` e `divu` simplesmente não geram qualquer tipo de exceção de overflow, cabendo mais uma vez ao programador garantir que os resultados são apropriados para os tamanhos especificados. Se o divisor for um valor igual a zero o resultado é imprevisível. Nenhuma exceção é gerada sob esta circunstância. O programador deve sempre checar se a operação tem um divisor igual a zero e avisar ao usuário caso isto ocorra.

Chegamos então ao fim das instruções aritméticas. A Tabela 2.2 mostra agora o compêndio do que foi estudado.

Categoria	Nome	Exemplo	Operação	Comentários
Aritmética	add	add $8, $9, $10	$8 = $9 + $10	*Overflow* gera exceção
	sub	sub $8, $9, $10	$8 = $9 – $10	*Overflow* gera exceção
	addi	addi $8, $9, 40	$8 = $9 + 40	*Overflow* gera exceção Valor do imediato na faixa entre –32.768 e +32.767
	addu	addu $8, $9, $10	$8 = $9 + $10	*Overflow* não gera exceção
	subu	subu $8, $9, $10	$8 = $9 – $10	*Overflow* não gera exceção
	addiu	addiu $8, $9, 40	$8 = $9 + 40	*Overflow* não gera exceção Valor do imediato na faixa entre 0 e 65.535
	mul	mul $8, $9, $10	$8 = $9 x $10	*Overflow* não gera exceção HI, LO imprevisíveis após a operação
	mult	mult $9, $10	HI,LO = $9 x $10	*Overflow* não gera exceção
	multu	multu $9, $10	HI,LO = $9 x $10	*Overflow* não gera exceção
	div	div $9, $10	HI = $9 mod $10 LO = $9 div $10	*Overflow* não gera exceção
	divu	divu $9, $10	HI = $9 mod $10 LO = $9 div $10	*Overflow* não gera exceção

Tabela 2.2: Operações Aritméticas do MIPS

Uma outra classe muito parecida com as operações aritméticas é a classe das operações lógicas. De fato, as operações lógicas são bastante conhecidas em HLLs. Uma operação `or` *sobre dois operandos realiza o* ***ou*** *bit-a-bit. Por exemplo:*

```
or $8, $9, $10     #  $8 = $9 or $10
```

se $9 contém $0001\ 1001\ 1111\ 0000\ 0000\ 1111\ 0011\ 1100_2$ *e*
$10 contém $1111\ 1101\ 1111\ 1110\ 0000\ 1111\ 1100\ 1100_2$*, então o resultado será, em $8,* $1111\ 1101\ 1111\ 1110\ 0000\ 1111\ 1111\ 1100_2$.

As operações, `and`*,* `xor` *e* `nor` *seguem a mesma sintaxe. Existem também operações lógicas envolvendo valores imediatos. Nós já mencionamos que quando operamos com valores imediatos, um dos operandos é especificado em 16 bits dentro da codificação da instrução. Então vamos verificar como podemos operar com um dos operandos em 16 bits e o outro em 32bits. Vamos ver o caso do* `andi`*. Esta instrução realiza um* and *entre o valor em um registrador e o valor imediato especificado na instrução. Assim, por exemplo:*

```
andi $8, $9, 121     #  $8 = $9 and 121.
```

Se $9 contém $0001\ 1001\ 1111\ 0000\ 0000\ 1111\ 0011\ 1100_2$ *então vamos verificar a codificação binária de 121 =* $111\ 1001_2$ *e transformá-la em um valor de 32 bits acrescentando zeros à esquerda. A operação seria então:*

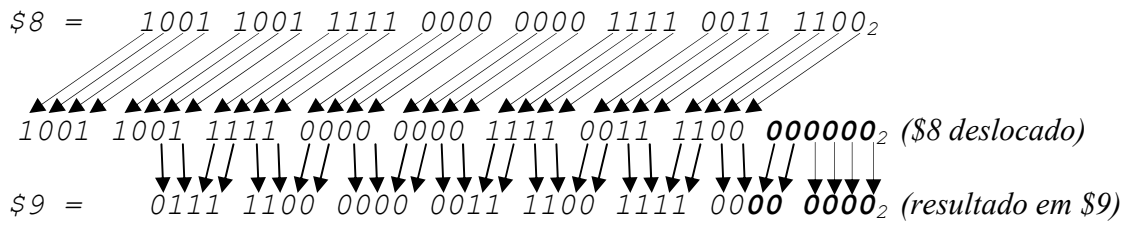
$0001\ 1001\ 1111\ 0000\ 0000\ 1111\ 0011\ 1100_2$ and
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111\ 1001_2$*, cujo resultado seria:*
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0011\ 1000_2$

A operação `ori` *também segue a mesma idéia, acrescentando zeros à esquerda para completar os 32 bits necessários à operação na ALU.*

As últimas operações lógicas a serem estudadas são as de deslocamento à esquerda e à direita. A instrução `sll` *(*shift left logical*) desloca o conteúdo de um registrador em n posições à esquerda, acrescentando zeros à direita. Vamos ver um exemplo com* `sll`*.*

```
sll $9, $8, 6     #  $9 = $8 << 6.
```

Desejamos deslocar o conteúdo do registrador $8 de 6 posições à esquerda. Supondo que originalmente o valor em $8 é `1001 1001 1111 0000 0000 1111 0011 1100`$_2$. *Então:*



Um detalhe no exemplo acima mostra que os 6 bits mais significativos de $8 foram descartados no resultado final.

A instrução `sll` *é muito usada para substituir multiplicações. Já mencionamos que a multiplicação é um processo caro em termos de desempenho da máquina. O deslocamento para esquerda de uma casa binária significa a multiplicação por dois da quantidade armazenada. Se a quantidade a ser deslocada for de 3 casas binárias, estaremos efetivamente multiplicando a quantidade por 8. A regra então é valor do multiplicando = 2^n, onde n é a quantidade de casas a ser deslocada para esquerda.*

Quando realizamos multiplicações desta forma temos de controlar o overflow*, já que as últimas casas binárias (mais à esquerda) são perdidas durante a operação. Outra observação importante é que só podemos realizar multiplicações com* `sll` *quando o multiplicador for uma potência de 2, já que deslocamos uma certa quantidade de casas binárias para esquerda.*

Vamos agora mostrar um exercício onde esta última restrição é diminuída, desde que o multiplicando esteja na circunvizinhança de uma potência de 2.

Desejamos implementar no MIPS um trecho de código que seja capaz de realizar uma multiplicação por um número que difere de uma potência de 2 em apenas 3 unidades (a mais). Como podemos realizar isto com as instruções conhecidas até o momento?

Bem, o problema é realizar a seguinte operação:

R = A x B, onde B está na circunvizinhança de C, que é uma potência de 2. Então,B = C + 3, o que implica em R = A x (C + 3). Assim, R = A.C + 3.A, ou R = A.C + A + A + A. Ora, A.C podemos realizar com `sll` *e a soma A + A + A pode ser feita com três operações de soma.*

Para tornar claro, vamos pôr alguns valores. Desejamos R = 50 x 19. Então, sendo 19 = 16 + 3, podemos realizar a operação como R = 50x(16 + 3). Isto implica em R = 50x16 + 50x3 = 50x2^4 + 50 + 50 + 50.

Vamos agora associar as variáveis aos registradores. Supondo A em $8, B em $9 e R em $10. Fazemos

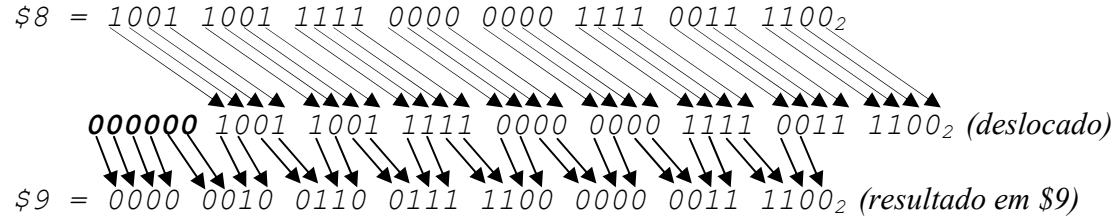
```
sll $10, $8, 4     # $10 = $8 << 4. $10 = $8 x 16
add $10, $10, $8   # $10 = $8 x 16 + $8
add $10, $10, $8   # $10 = $8 x 16 + $8 + $8
add $10, $10, $8   # $10 = $8 x 16 + $8 + $8 + $8
```

Isto pode parecer muito ineficiente já que a multiplicação poderia ser realizada com apenas uma instrução mul $10, $8, $9. O problema é que esta instrução demora 10 vezes mais para ser executada que uma soma ou deslocamento. Neste caso estaríamos trocando 10 unidades de tempo (para executar a multiplicação) por 4 unidades de tempo (para executar as 4 operações). O código fica então muito mais eficiente, embora fique maior.

O oposto à instrução `sll` é a instrução `srl` (shift right logical). Ela realiza o deslocamento de um valor armazenado em um registrador para a direita, preenchendo com zeros os valores à esquerda do número. Por exemplo:

```
srl $9, $8, 6     #  $9 = $8 >> 6.
```

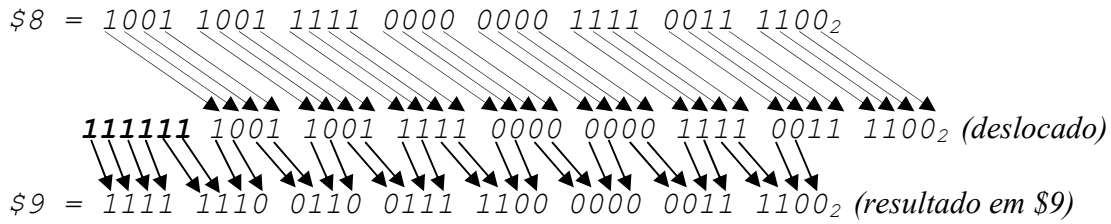
Desejamos deslocar o conteúdo do registrador $8 de 6 posições à direita. Supondo que originalmente o valor em $8 é `1001 1001 1111 0000 0000 1111 0011 1100`$_2$. Então:



Mais uma vez temos de nos preocupar com os bits mais à direita, que são perdidos durante a execução da instrução. A instrução `srl` é interessante para realizar divisões de números não sinalizados, ao molde do que acontece com o `sll`.

Uma restrição à substituição de uma instrução de divisão por uma de deslocamento à direita é quanto à utilização de números sinalizados. Ora, sabemos que o bit de sinal é o mais à esquerda em um número binário. Então ele precisaria ser replicado à esquerda para manter o sinal ao final da operação. Esta limitação levou ao projeto de mais uma instrução: `sra` (shift

right arithmetic). `sra` *opera da mesma forma que* `srl`*, mas ao contrário desta última, os bits que serão adentrados à esquerda são a réplica do bit de sinal: O para um valor positivo e 1 para um valor negativo. Vamos ao exemplo: desejamos deslocar o conteúdo do registrador $8 de 6 posições à direita, mantendo o sinal. Supondo que originalmente o valor em $8 é* `1001 1001 1111 0000 0000 1111 0011 1100`$_2$*. Então:*



Veja que a quantidade a ser deslocada é um número negativo e por isto os bits que completaram o $9 à esquerda foram uns e não zeros.

Fechamos então o bloco das operações lógicas, compendiadas na Tabela 2.3.

Categoria	Nome	Exemplo	Operação	Comentários
lógicas	or	or $8, $9, $10	$8 = $9 or $10	
	and	and $8, $9, $10	$8 = $9 and $10	
	xor	xor $8, $9, 40	$8 = $9 xor 40	
	nor	nor $8, $9, $10	$8 = $9 nor $10	
	andi	andi $8, $9, 5	$8 = $9 and 5	Imediato em 16 bits
	ori	ori $8, $9, 40	$8 = $9 or 40	Imediato em 16 bits
	sll	sll $8, $9, 10	$8 = $9 << 10	Desloc. ≤ 32
	srl	srl $8, $9, 5	$8 = $9 >> 5	Desloc. ≤ 32
	sra	srl $8, $9, 5	$8 = $9 >> 5	Desloc. ≤ 32, preserva sinal

Tabela 2.3: Operações Aritméticas do MIPS

2.4 – A visão do software – transferência de dados

Já temos o conhecimento necessário para realizar as operações aritméticas no MIPS. Entretanto, duas questões ficaram no ar na seção anterior. Trata-se de como movimentar os dados do par de registradores HI, LO para o banco de registradores ou como movimentar dados do banco de/para memória. A primeira questão é bem simples de responder, então vamos começar por ela.

O conjunto de instruções do MIPS apresenta duas instruções especializadas no transporte dos conteúdos dos registradores HI e LO para o banco de registradores: `mfhi` *(*Move From HI*) e* `mflo` *(*Move From LO*).*

Estas instruções apresentam apenas um operando: o registrador de destino. A sintaxe é demonstrada no exemplo a seguir.

```
mfhi $9     #  $9 = HI
mflo $9     #  $9 = LO
```

Os nomes são intuitivos, mas para deixar claro, `mfhi`*, transfere o conteúdo do registrador HI para o registrador de destino especificado na instrução, enquanto,* `mflo`*, transfere o conteúdo do registrador LO para o registrador de destino especificado. Estas instruções podem ser úteis para controlar o* overflow *das operações de multiplicação e para podermos extrair o quociente e o resto de uma divisão.*

Agora vamos partir para a segunda questão. Sabemos que programas em HLLs podem apresentar infinitos dados para serem utilizados durante a execução do código. Sabemos também que o banco de registradores é uma região limitada, portando normalmente até 18 variáveis por vez. Ora, imagine o caso onde um vetor de 128 posições fosse um dos dados a ser utilizado no programa. Sem dúvidas que haveria uma restrição enorme se não fosse a memória.

Nós vamos estudar os detalhes da implementação da memória no Capítulo 6, mas precisamos de um modelo que será utilizado pelo software como sendo este repositório de dados e programas. Então, o software enxerga a memória como um grande vetor onde cada posição é indicada por um endereço. Os endereços são seqüenciais e numerados de 0 a 4.294.967.295. Cada byte ocupa uma destas posições. A Figura 2.1 mostra como são os endereços e dados em uma memória. Os dados estão representados em binário e os endereços em hexadecimal.

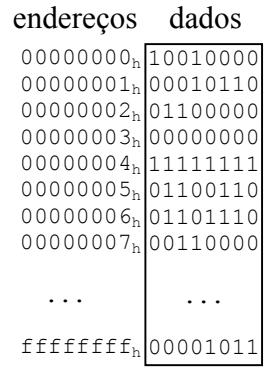


Figura 2.1: Modelo de Memória

Todos os endereços são especificados em 32 bits (4 bytes). Para acessar uma variável na memória é preciso então informar qual o seu endereço e para onde (qual registrador) ela deve ser transferida. No caso de escrita na memória, o processo é invertido. É preciso especificar o endereço de destino e qual o registrador que contém o dado a ser gravado naquele endereço da memória.

Então significa que nós estamos transferindo dados na memória que estão organizados em bytes para registradores que utilizam 32 bits (uma palavra de 4 bytes). Três tamanhos de dados podem ser carregados para os registradores: byte (8 bits), meia-palavra (16 bits) e palavra (32 bits). Considerando que cada registrador comporta uma palavra inteira de 32 bits, vamos mostrar como é feita a transferência de dados de uma palavra na memória para um registrador e vice-versa. A Figura 2.2 estende a compreensão da Figura 2.1. Cada palavra ocupa 4 posições na memória e portanto palavras subseqüentes estão 4 unidades de endereço a seguir. Assim a memória pode ser enxergada como um grande repositório de dados de 32 bits. Esta organização da memória é bastante comum nos processadores modernos. Embora possamos endereçar bytes, a transferência é feita em palavras.

Nesta organização não é possível transferir um valor de 32 bits que tenha seu início em um endereço diferente de um múltiplo de 4. Uma tentativa de transferir uma palavra começando de um endereço cujo último nibble *(à direita) seja diferente de 0, 4, 8 ou 12 (c_h) gera uma exceção e o processamento pára.*

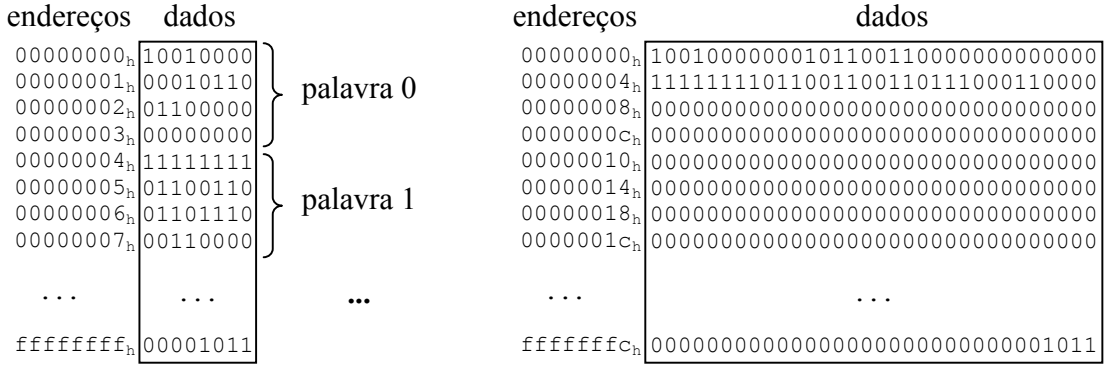


Figura 2.2: Modelo de Memória

Já mencionamos que para transferir um dado da memória para o banco de registradores é preciso especificar o seu endereço efetivo. Para

tanto, o MIPS usa um cálculo de endereço. Basicamente são utilizados um registrador que fornece um ***endereço base*** *e um imediato, especificado na instrução, que fornece o* ***deslocamento*** *a partir deste endereço base. Este* ***modo de endereçamento,*** *com base e deslocamento, tem uma razão bastante significativa: acessos a vetores.*

Vamos conhecer as instruções de transferência de palavras utilizadas pelo MIPS. A instrução `lw`, load word*, transfere dados da memória para o banco de registradores e a instrução* `sw`, store word*, transfere dados do banco de registradores para a memória. A Figura 2.3 mostra as duas instruções de forma pictorial.*

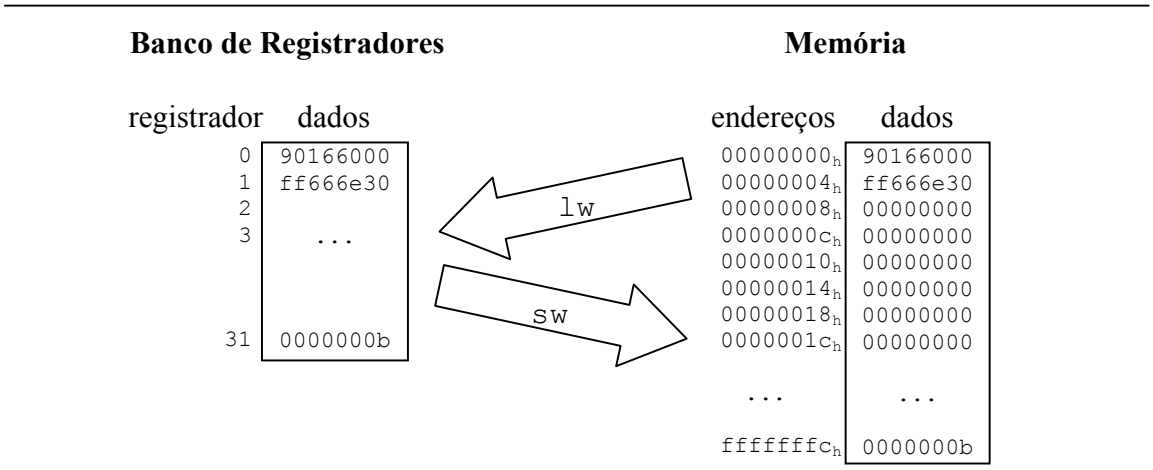


Figura 2.3: Instruções de transferência de dados

O cálculo do endereço efetivo para acesso à memória, nós já mencionamos. Vamos agora elaborar a idéia. Na sintaxe da instrução `lw`*, devem ser especificados um registrador de destino, e um par, registrador base mais um deslocamento imediato. O valor armazenado no registrador base é somado ao valor imediato especificado na instrução formando assim o endereço onde se encontra o dado na memória. Este endereço é então lido e o dado carregado para dentro do banco de registradores, para o registrador especificado na instrução. O cálculo do endereço efetivo pode ser visto na Figura 2.4. Neste exemplo o registrador escolhido foi o registrador 2 que contém o seguinte dado:* $0c_h$*. O valor do deslocamento é 4, assim o endereço efetivo é* $0c_h + 4_h = 10_h$*. O dado que está neste endereço é carregado para o registrador $30, especificado na instrução.*

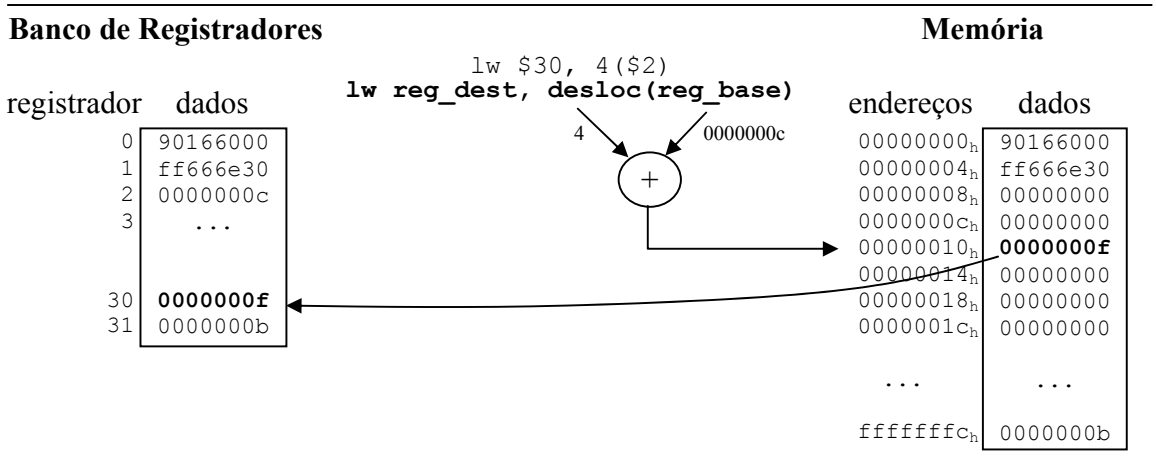


Figura 2.4: Cálculo do endereço efetivo

Para o caso de uma instrução `sw`, o cálculo do endereço efetivo é exatamente igual. A Figura 2.5 mostra um exemplo de `sw`. A única alteração aqui diz respeito ao sentido da transferência dos dados, que saem agora do banco de registradores e vão para memória.

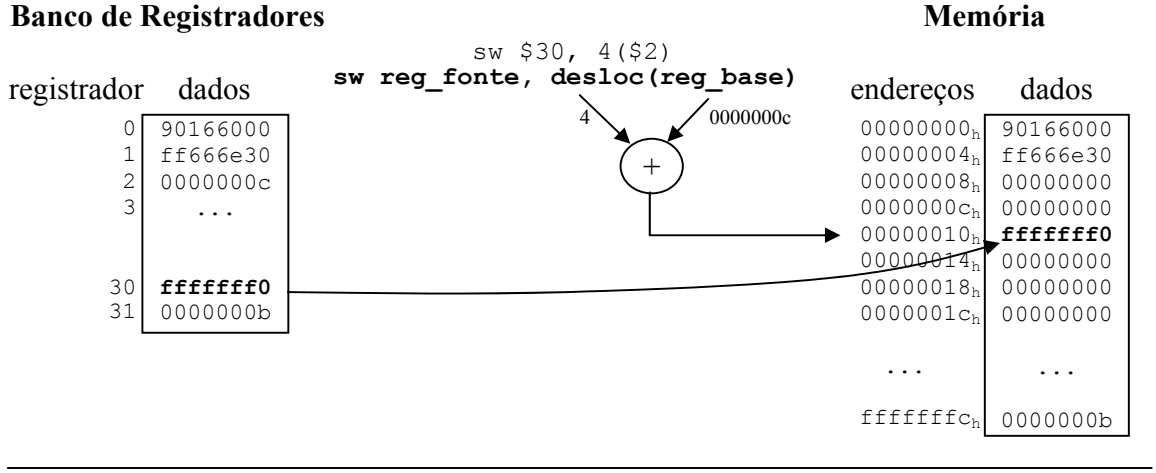


Figura 2.5: Cálculo do endereço efetivo

Então, neste momento nós conhecemos as instruções de transferência de dados, do tamanho de uma palavra, da memória para o banco de registradores e vice-versa. Resta saber como estas palavras de dados, associadas a variáveis vão parar na memória. Via de regra, quando declaramos as variáveis de um programa o compilador cuida para alocar espaço na memória correspondente aos dados. Por exemplo, se declaramos 3 inteiros, `a`, `b` e `c`, na memória são reservados 3 endereços consecutivos para

guardar estas variáveis. Quando inicializamos uma variável durante sua declaração, os endereços correspondentes na memória já devem ter seus valores setados.

Um ponto crítico que merece nossa atenção é a alocação de memória para vetores e matrizes. Um vetor de números inteiros ocupa tantas posições na memória, quantos forem os seus dados. Por exemplo, vamos ver como a memória é alocada para as seguintes declarações de dados: um inteiro não sinalizado, x, inicializado com valor 3, um inteiro y, inicializado com -1 e um vetor de inteiros n com 10 posições. A Figura 2.6 mostra a alocação convencional de dados para este exemplo. Veja que os dados foram alocados em uma região de memória iniciando no endereço 0. Esta região é indiferente para o programador, já que ele acessa a variável pelo nome, mas, de fato, ela é controlada pelo módulo carregador do Sistema Operacional.

As variáveis são então alocadas em seqüência, da forma como foram declaradas. Observe que o vetor n tem um endereço de início, também chamado endereço base e os seus dados podem ser acessados a partir deste endereço. Por exemplo, para acessar o dado n[0] usamos o endereço base de n somado a zero. Para acessar o dado de n[1] usamos o endereço base de n somado ao valor 4 (4x1) para encontramos o endereço efetivo. Para acessar n[2] somamos o endereço base de n com 8 (4x2). Para acessarmos o dado n[m] somamos o endereço base de n com 4 x m. Isto mostra todo potencial do cálculo do endereço efetivo presente na instrução de carga `lw`.

```
...
unsigned int x = 3;
int y = -1;
int n[10];
...
```

Memória

endereços	dados	variáveis
00000000_h	00000003	x
00000004_h	ffffffff	y
00000008_h	00000000	n[0]
$0000000c_h$	00000000	n[1]
00000010_h	00000003	n[2]
00000014_h	00000000	n[3]
00000018_h	00000000	n[4]
$0000001c_h$	00000004	n[5]
00000020_h	00000000	n[6]
00000024_h	00000000	n[7]
00000028_h	00000000	n[8]
$0000002c_h$	00000000	n[9]
...	...	...

Figura 2.6: Alocação de variáveis na memória

Vamos fazer um outro exemplo de cálculo para expressão:

```
n[3] = y * n[5];
```

Ora, a codificação em assembly para esta organização de memória seria:

```
lw $8, 4($0)     # carrega o valor de y em $8
addi $9, $0, 8   # carrega o endereço base de n em $9
lw $10, 20($9)   # carrega o valor de n[5] em $10
mul $8, $8, $10 # multiplica y por n[5]. Resultado em $8
sw $8, 12($9)    # guarda valor da multiplicação em n[3]
```

Neste exemplo utilizamos um registrador que ainda não havíamos trabalhado: o $0. Este registrador tem uma característica especial, o valor dele é sempre zero, independente das instruções que operam com ele. Este registrador é muito importante para calcularmos valores iniciais de endereços e/ou dados. Este código precisaria ser alterado se a memória fosse re-endereçada (relocada), ou seja, se o início dos dados não fosse no endereço 0.

No caso de armazenamento de matrizes, o compilador enxerga uma matriz como um vetor de vetores, portanto, as linhas são postas seqüencialmente na memória. Fica como exercício descobrir como endereçar um elemento n[m,k] de uma matriz de números inteiros.

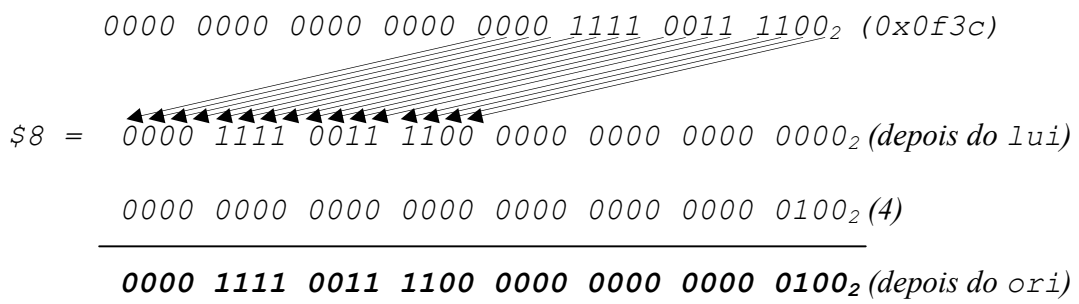
Até o presente mostramos como transportar dados de/ para memória. Existe uma outra instrução, que pode ser interpretada também como uma instrução lógica, que auxilia no cálculo de endereços efetivos. Trata-se da instrução `lui` *(*Load Upper Immediate*). Esta instrução carrega na parte mais significativa (16 bits mais à esquerda) um valor imediato especificado na própria instrução e zera a parte baixa do registrador de destino. Por exemplo, desejamos armazenar o endereço base de um vetor que começa em* $0f3c0004_h$*. Ora, não existe uma forma de armazenar imediatamente os 32 bits que formam este endereço, pois todas as instruções que operam com imediatos só admitem valores de 16 bits.*

A solução é usar uma instrução que compute este endereço base por partes. A instrução `lui` *então pode ser usada para armazenar a parte alta do endereço e uma instrução de* `ori` *pode ser usada para armazenar a parte baixa. A solução para este exemplo seria:*

```
lui $8, 0x0f3c # carrega a parte alta de $8 com 0x0f3c
ori $8, $8,0x0004  # ajusta a parte baixa do endereço
```

Ora, a primeira instrução armazena em $8 o valor $0f3c_h$ *nos 16 bits mais significativos (mais à esquerda). Os 16 bits menos significativos são*

zerados. Depois é feito um or *imediato com o valor 4, o que nos permite construir o endereço base final do vetor, como mostrado a seguir.*



Fechamos então o bloco das operações de transferência de dados com as instruções apresentadas na Tabela 2.4.

Categoria	Nome	Exemplo	Operação	Comentários
Transferência de dados	mfhi	mfhi $8	$8 = HI	
	mflo	mflo $8	$8 = LO	
	lw	lw $8, 4($9)	$8 = MEM[4 + $9]	
	sw	sw $8, 4($9)	MEM[4 + $9] = $8	
	lui	lui $8, 100	$8 = 100 x 2^{16}	Carrega constante na porção alta do registrador de destino. Zera a parte baixa.

Tabela 2.4: Instruções de transferência de dados do MIPS

2.5 – A visão do software – suporte à decisão

Uma das características mais importantes de uma linguagem de alto nível é a possibilidade de escolha de trechos de código para serem executados em detrimento de outros, escolha esta controlada por uma expressão lógica. A referência explícita aqui é das construções condicionais, if – then *e* if – then – else*. Para construir estruturas condicionais no MIPS é preciso antes conhecer como é realizada a execução de um código na memória.*

O conceito de programa armazenado, apresentado no capítulo precedente, é o fundamento de nossa discussão. Na memória de um computador não apenas os dados ficam armazenados, mas também os programas. A memória é dividida logicamente em regiões de dados e de códigos. Esta divisão é meramente uma convenção. Estudaremos o modelo completo no capítulo sobre o sistema de memórias. Por enquanto vamos

visualizar um diagrama que mostra os dados e os códigos nas duas regiões de memória convencionadas para conter dados e programas. A Figura 2.7 apresenta a região de código como alocada entre os endereços 00400000_h *e* $0fffffc_h$*. Nesta região fica guardada a codificação binária (aqui representada em hexadecimal por razões de simplicidade) das instruções. O programa associado está apresentado ao lado apenas por razões pedagógicas. Perceba que cada instrução ocupa necessariamente uma palavra na memória.*

A região compreendida entre os endereços 10000000_h *e* $10007ffc_h$ *guarda os dados. Neste caso, as mesmas variáveis que utilizamos no exemplo anterior, mas agora alocadas em sua região mais convencional.*

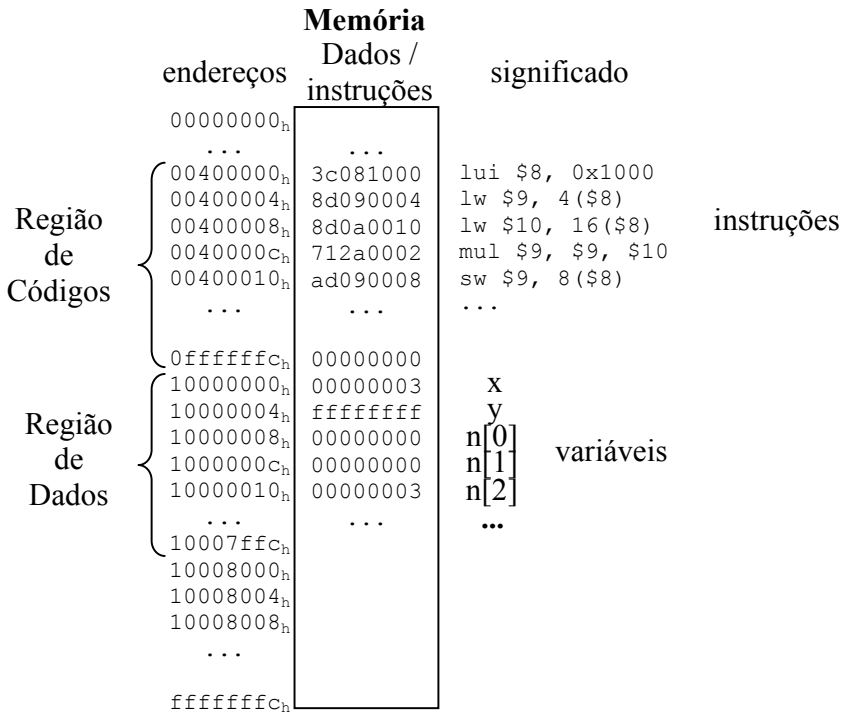


Figura 2.7: Alocação de variáveis/instruções na memória

O programa a ser executado é olhado seqüencialmente, instrução seguida de instrução. Como exercício, descubra o que faz o código apresentado na Figura 2.7 e qual a situação da memória após sua execução.

*Já vimos que os dados para serem manipulados (computados) precisam ser carregados para o banco de registradores. Os códigos também precisam ser carregados em um registrador. Existe um registrador especial chamado IR (*Instruction Register*) que guarda a instrução que deve ser*

*executada em um determinado instante. Vimos também que para acessar um dado na memória é preciso calcular o endereço onde ele se encontra. Para o código, usamos um registrador especial chamado PC (*Program Counter*). Quando estamos executando um programa, o PC contém o endereço da instrução a ser executada. Ela então é lida da memória e executada na via de dados. O PC deve ser iniciado sempre com o valor* 00400000_h*, pois é este endereço onde a região de código começa. A Figura 2.8 mostra os passos para execução de uma instrução de um programa.*

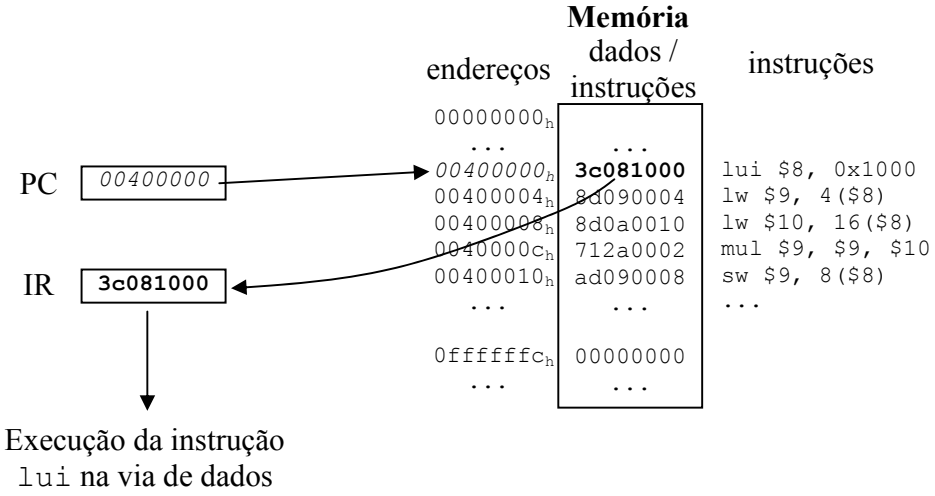


Figura 2.8 Execução de uma instrução de um programa

Como as instruções são executadas seqüencialmente, o valor de PC também é incrementado seqüencialmente. Observe, entretanto, que uma instrução ocupa 32 bits na memória e por isto a próxima instrução está 4 unidades de endereço (bytes) distante. Isto implica que o valor de PC é incrementado de 4 em 4. A Figura 2.9 mostra o restante da execução do código.

Veja que o valor de PC é fundamental para execução do código. Se o valor de PC sair de sua seqüência natural vai provocar um descontrole na seqüência de execução das instruções. É exatamente este o efeito que precisamos para implementar instruções que saltem de um local para outro do código permitindo fazer uma escolha condicional.

Vamos aprender então as instruções que alteram o valor de PC condicionalmente ou não.

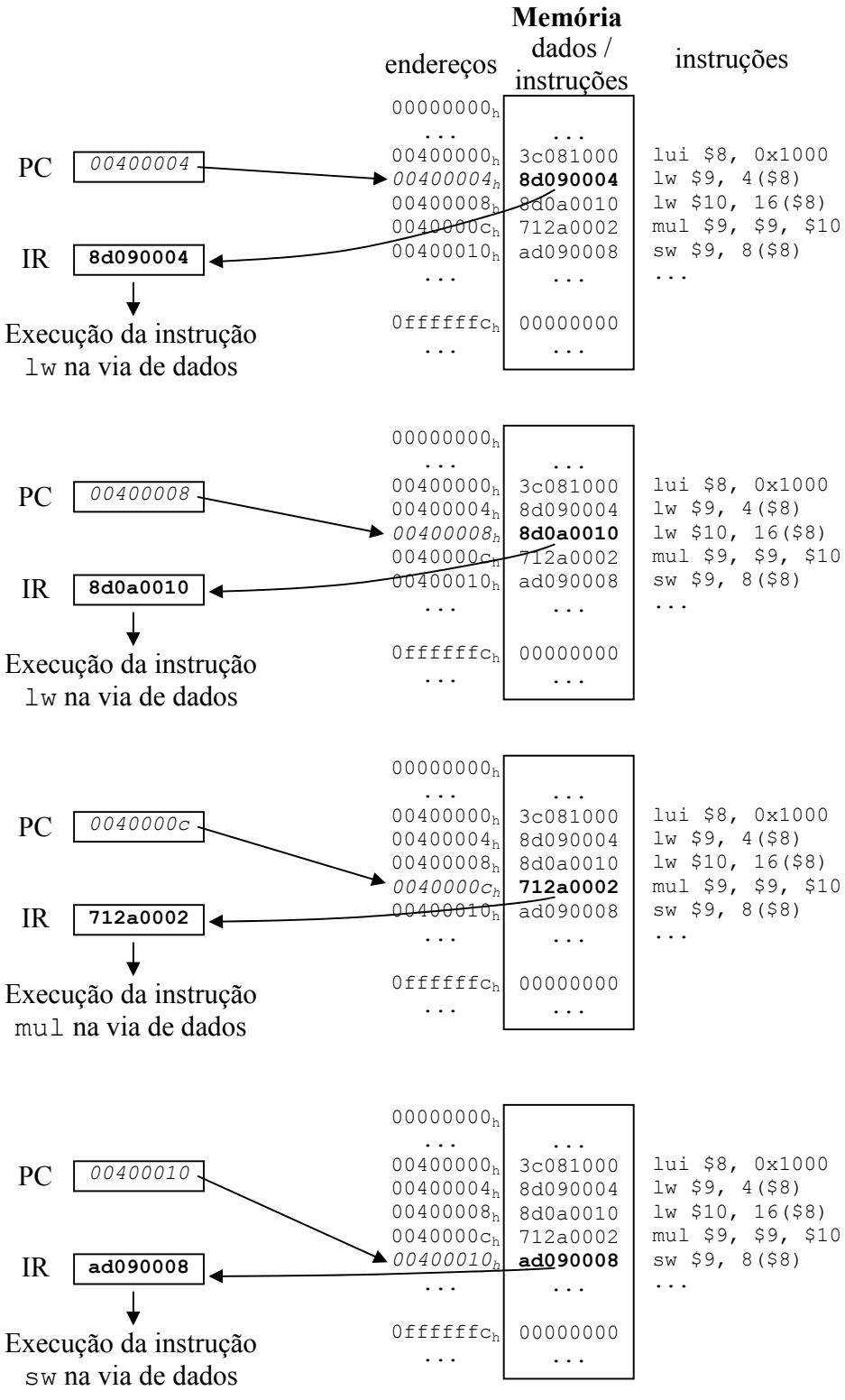


Figura 2.9 Execução de um programa passo a passo

Vamos começar com as instruções `bne` *e* `beq`*.* `bne` *salta para um determinado endereço se os conteúdos dos dois registradores especificados nos operandos não forem iguais (*branch if not equal*).* `beq`*, seu complemento, salta para um determinado endereço se os registradores especificados em seus operados forem iguais (branch if equal). Para evitar a necessidade de cálculo de endereços de destinos usamos símbolos para representá-los. Estes símbolos são associados a endereços através do uso de* ***rótulos****.*

O exemplo a seguir mostra uma estrutura `if-then` *sendo implementada com a instrução* `bne`*. As variáveis* `i`*,* `j` *e* `h` *estão associadas a $8, $9 e $10 respectivamente. O símbolo* `sai` *representa o endereço para onde saltaremos caso a condição de igualdade não se verifique. A condição do* `if` *para que a soma se realize é que os valores de* `i` *e* `j` *sejam idênticos. Se* `i` *e* `j` *não forem iguais iremos saltar a soma (*`h = i + j`*) sem executá-la. É justamente esta última assertiva que implementamos no MIPS.* `bne $8, $9, sai`*, verifica se o conteúdo de $8 é igual ao conteúdo de $9. Caso eles <u>não</u> sejam iguais, a instrução* `bne` *altera o valor de PC para que ele salte para o endereço associado ao símbolo* `sai`*. A Figura 2.10 mostra como um compilador traduziria o* `if` *em um conjunto de instruções em* assembly *do MIPS e ainda, qual o fluxograma das ações esperadas, quando verificada a expressão lógica de (des)igualdade.*

```
if (i == j){              bne $8, $9, sai
    h = i + j;            add $10, $8, $9
}                 sai:    ....
```

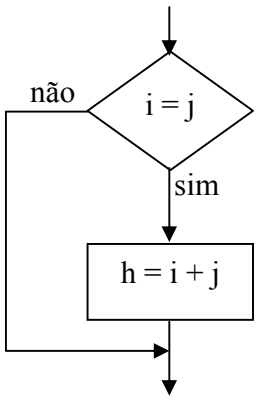


Figura 2.10 Implementação de um comando *if* em MIPS

Vamos agora verificar como estes comandos são alocados na memória e executados no processador. A Figura 2.11 mostra o momento em que a instrução `bne` *é executada. Ela verifica os valores dos registradores $8 e $9*

e percebe que eles são diferentes. A instrução altera o valor de PC para que ele enderece o rótulo `sai`*. A execução segue normalmente depois, com o PC sendo alterado de 4 em 4 unidades.*

A Figura 2.12 também mostra a execução da instrução `bne`*, mas neste caso os registradores $8 e $9 contém o mesmo valor. Isto significa que a instrução* `bne` *não interfere no curso natural do PC, que passa a apontar a instrução de soma. Após executá-la o valor de $10 é alterado e o processamento segue normalmente. A instrução* `nop` *mostrada nas figuras é uma instrução que não realiza nada.*

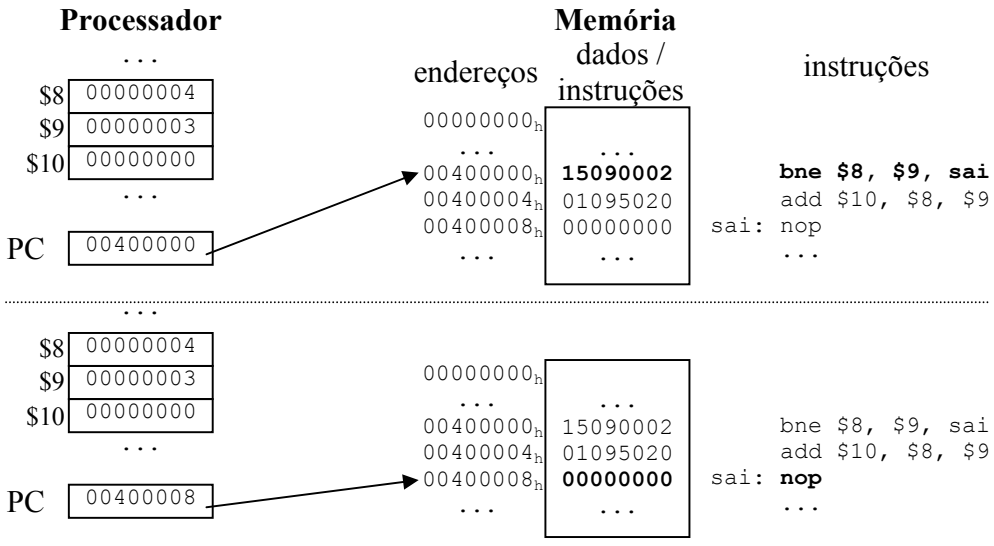


Figura 2.11 Execução de um comando *if* (`bne`) no MIPS

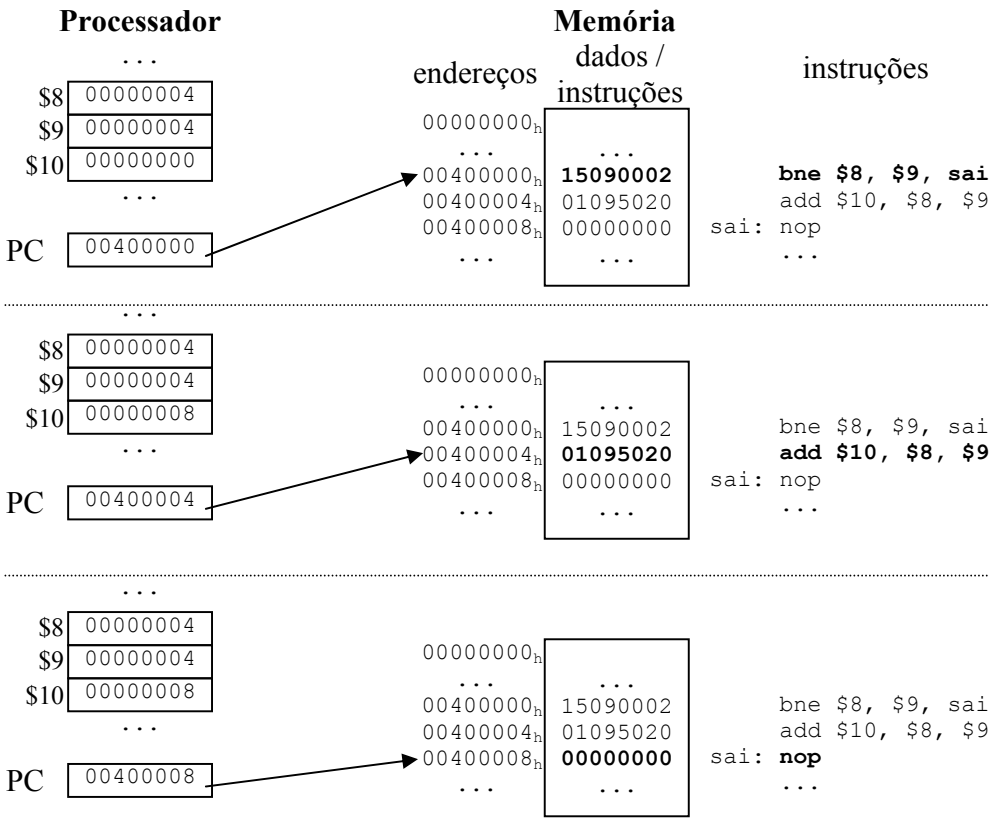


Figura 2.12 Execução de um comando *if* (`bne`) no MIPS

O MIPS também propicia saltos incondicionais. A instrução `j` *(*jump*) salta para um rótulo independente de qualquer condição de valores de registradores. Esta instrução possui como operando apenas o endereço de destino do salto. Ela é útil, entre outras coisas, para implementação da estrutura condicional* if-then-else*. Veja que o bloco* 'then' *quando terminado, precisa sair do* `if` *incondicionalmente, sob pena de o comando executar o* then *e o* else *seqüencialmente. A Figura 2.13 mostra estrutura e a Figura 2.14 e 2.15 a execução do trecho de programa.*

```
if (i == j){              bne $8, $9, else
    h = i + j;            add $10, $8, $9
} else {                  j sai
    h = i - j;    else:   sub  $10, $8, $9
}                 sai:    nop
```

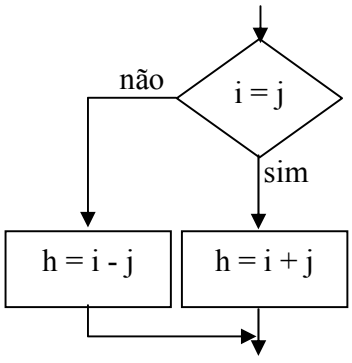


Figura 2.13 Implementação de uma estrutura *if–then–else* em MIPS

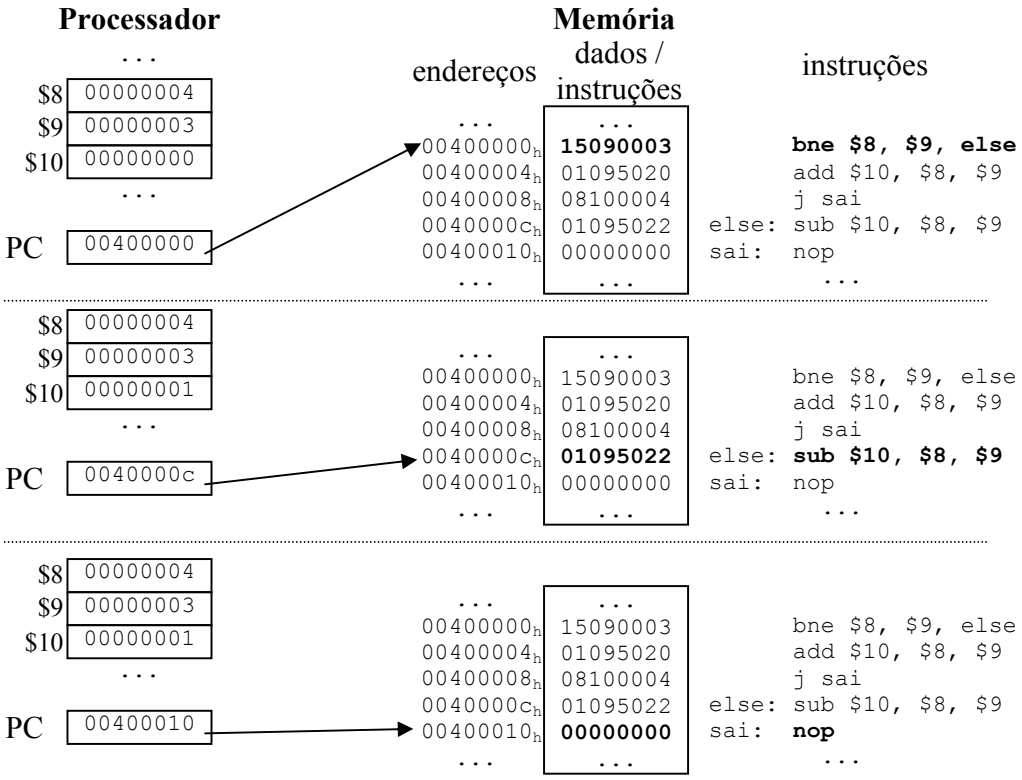


Figura 2.14 Execução de uma estrutura *if-then-else* no MIPS

Na Figura 2.14 o `bne` *provoca uma alteração no PC, de tal forma que a próxima instrução a ser executada é a* `sub`*. Em seguida a execução segue seu curso normal. Já a Figura 2.15 mostra a execução do bloco* then *finalizando com a instrução* `j`*. Quando esta última é executada, ela altera o valor do PC para que o mesmo aponte para o endereço do rótulo* `sai`*. Este salto é incondicional, não depende de nenhum valor de registrador.*

Depois de executado o jump *o fluxo de execução segue normalmente.*

Até o presente aprendemos como construir estruturas condicionais e como desviar o fluxo de execução de um programa utilizando as instruções `bne`*,* `beq` *e* `j`*. Existe, entretanto, uma limitação de implementação de estruturas condicionais de HLLs, com apenas estas instruções. Ora, e se a condição de comparação envolver uma condição de maior-que ou menor-que? Para suportar esta funcionalidade o conjunto de instruções do MIPS tem uma instrução que pode ser considerada aritmética, que opera informando se um determinado valor é menor que outro. Esta instrução é a* `slt` *(*set on less then*). Sua sintaxe exige a presença de dois operandos que*

terão seu valores avaliados e de um operando destino, que irá receber o valor 1 ou 0 dependendo se a condição de menor que *for verificada.*

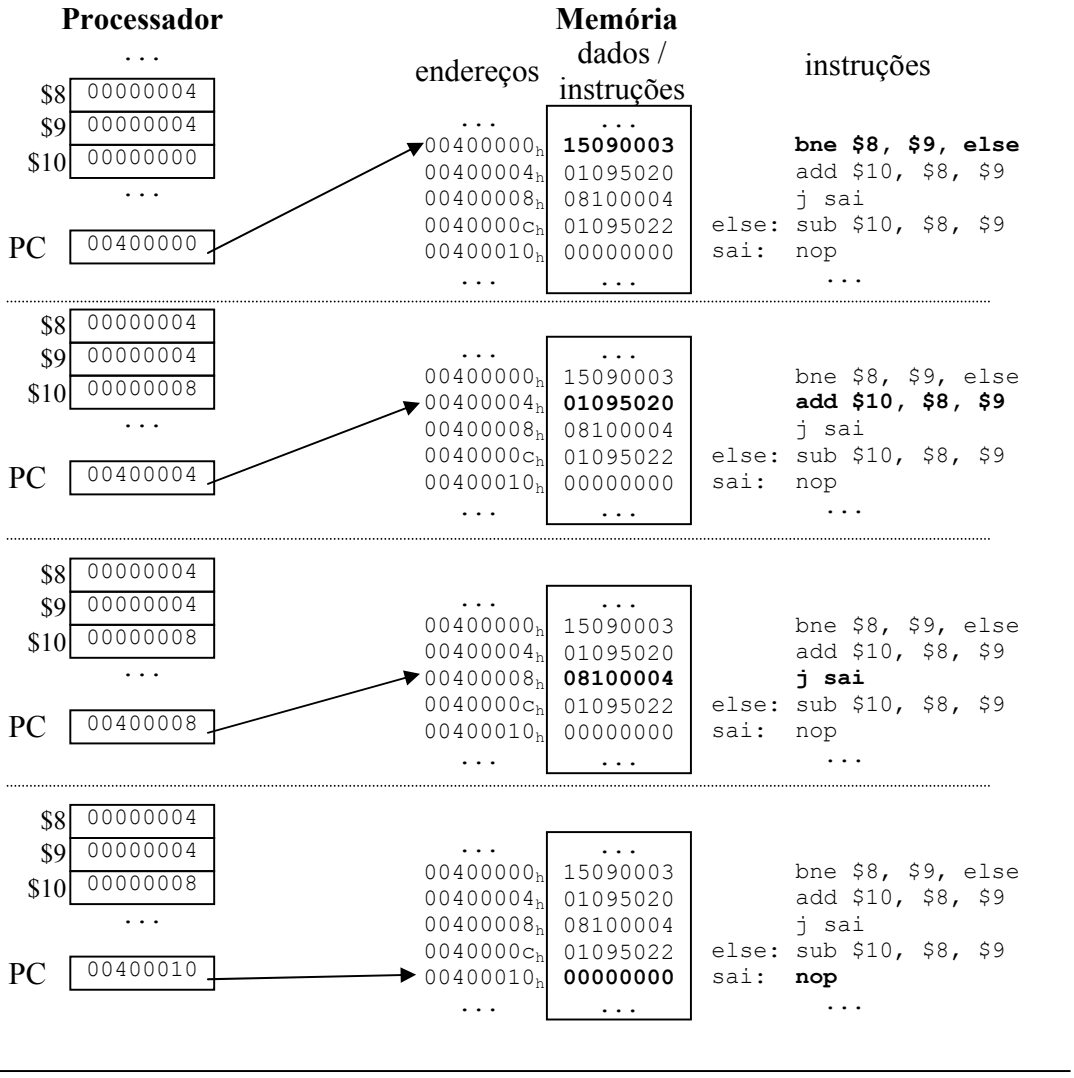


Figura 2.15 Execução de uma estrutura *if-then-else* no MIPS

Vamos a um exemplo:

```
slt $8, $9, $10 # se $9 < $10 então $8 = 1, senão $8 = 0
```

Neste caso, a instrução verifica se $9 é menor que $10, caso isto ocorra, o valor de $8 é setado (vira 1). Se $9 não for menor que $10 (ou seja, $9 é maior ou igual a $10), então $8 recebe o valor 0. Vejamos como podemos implementar uma estrutura if-then-else cuja condição exige uma comparação de grandeza. A Figura 2.16 mostra a combinação das instruções `slt` *e* `beq` *para implementar o* `if`*.*

Veja que se `i < j` *então a instrução* `slt` *vai colocar em $11 o valor 1. A* `beq` *que vem logo em seguida vai verificar se $11 contém um valor 0. Se tiver o 0 saltará para o rótulo* else*, senão executará a instrução seguinte, a soma. Neste caso $11 contém 1 e portanto a soma será executada. Este é exatamente o efeito procurado, se* `i < j` *a expressão a ser executada é a soma e não a subtração. A Figura 2.17 ilustra tal situação.*

```
if (i < j){                slt $11, $8, $9
    h = i + j;             beq $11, $0, else
} else {                   add $10, $8, $9
    h = i - j;             j sai
}                  else:   sub  $10, $8, $9
                   sai:    nop
```

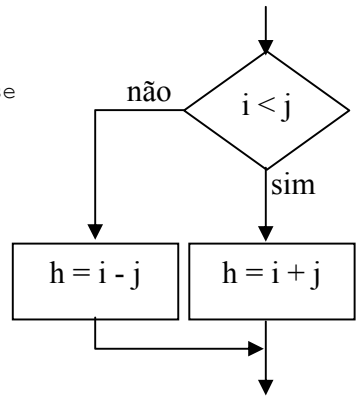


Figura 2.16 Implementação de uma estrutura *if–then–else* no MIPS

A Figura 2.18, por sua vez, mostra a situação contrária, onde $8 não é menor que $9. Isto irá fazer com que o bloco else (a subtração) seja executado.

A instrução `slt` *faz a comparação com números sinalizados, entretanto, existe uma outra instrução,* **`sltu`***, que opera com número não sinalizados. É preciso um cuidado especial para escolher qual das duas instruções deve compor nossos códigos. Uma escolha errada pode levar a resultados inesperados.*

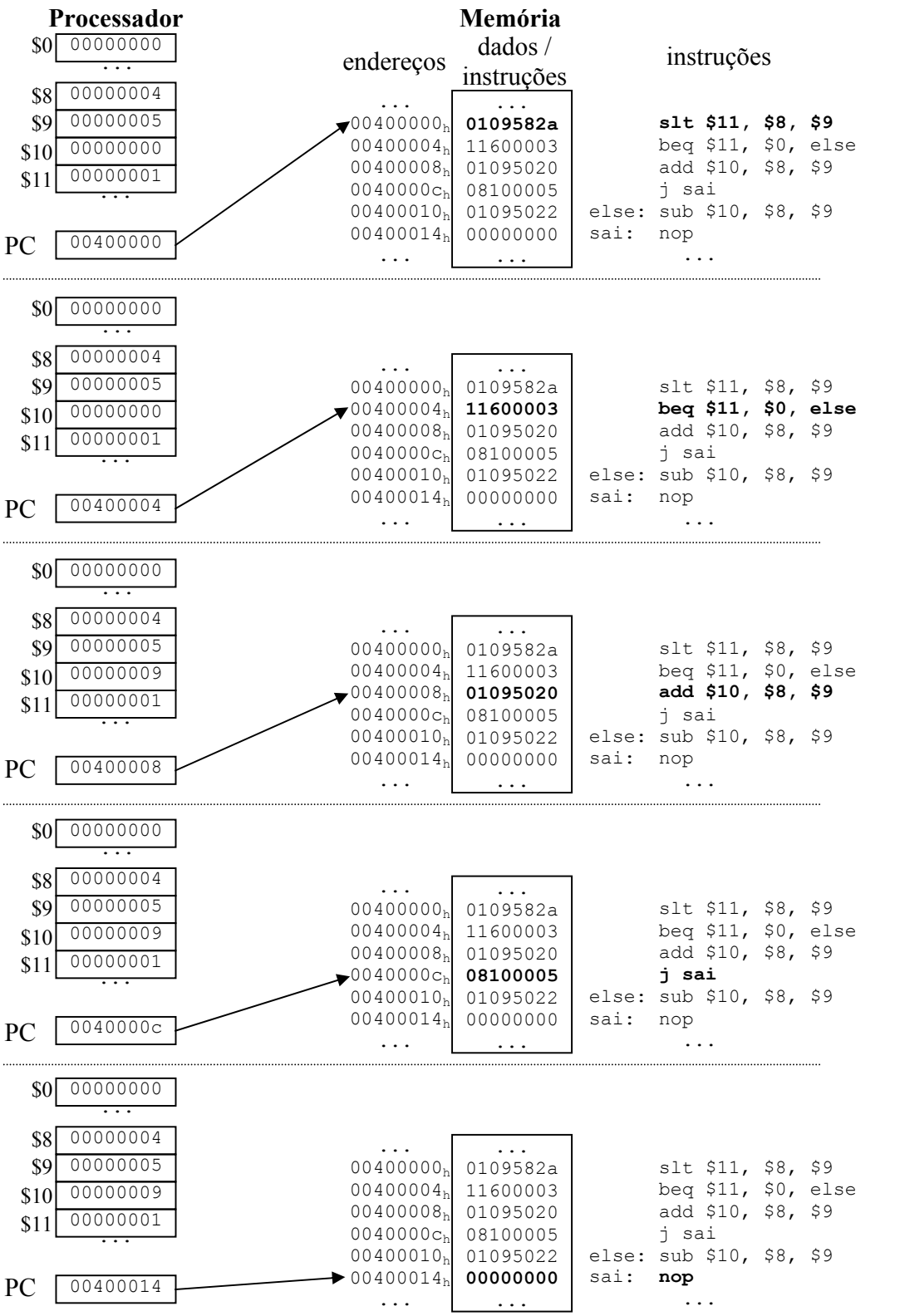


Figura 2.17 Execução de uma estrutura *if-then-else* no MIPS

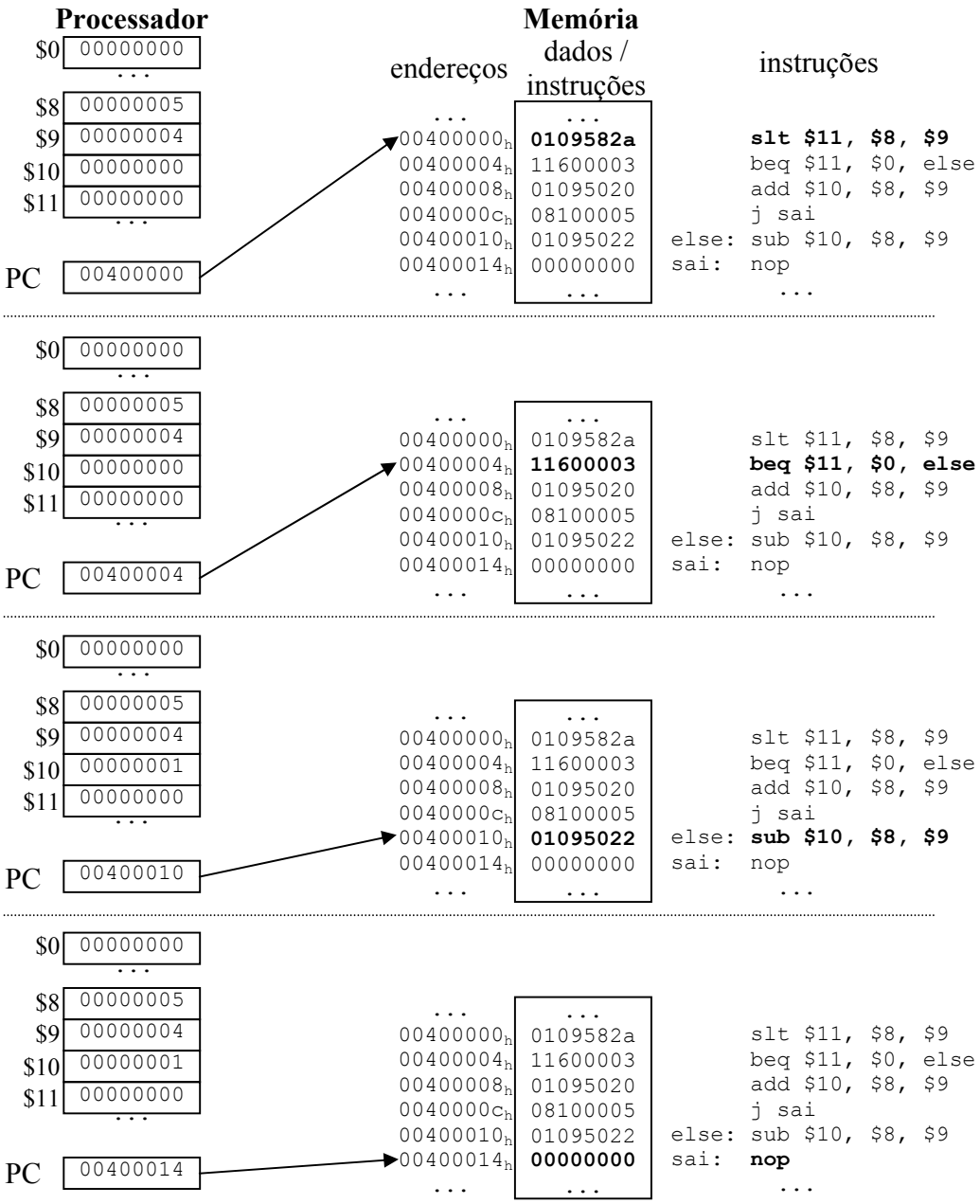


Figura 2.18 Execução de uma estrutura *if-then-else* no MIPS

Chegamos agora ao fim desta seção. As instruções de salto condicionais e incondicionais mostradas podem realizar qualquer estrutura de decisão de uma linguagem de alto nível. Com elas também é possível construir laços, sendo deixado para o leitor, como exercício, construir estruturas `for`*,* `while`*, e* `repeat`*.*

Categoria	Nome	Exemplo	Operação	Comentários
Suporte a decisão	bne	bne $8, $9, rotulo	se $8 ≠ $9 então PC ← endereço[rotulo]	
	beq	beq $8, $9, rotulo	se $8 = $9 então PC ← endereço[rotulo]	
	j	J rotulo	PC ← endereço[rotulo]	
	slt	slt $10, $8, $9	se $8 < $9 então $10 ← 1 senão $10 ← 0	Opera com números sinalizados
	sltu	sltu $10, $8, $9	se $8 < $9 então $10 ← 1 senão $10 ← 0	Opera com números não sinalizados

Tabela 2.5: Instruções de suporte a decisão

2.6 – A visão do software – suporte à procedimentos

Procedimento são porções de código que realizam uma tarefa bem específica dentro de um programa. Digamos que um programa exija a computação de um fatorial diversas vezes. Podemos escrever o código contendo os diversos trechos de computação do fatorial, como mostrado na Figura 2.19. Este trecho de código calcula o fatorial de um número armazenado em $4 duas vezes. O resultado fica em $2. O valor de $4 é setado para 5 e em seguida o fatorial de 5 é calculado, deixando o resultado, 120, em $2. Em seguida $4 recebe o valor 9 e um novo cálculo, agora para 9!, é realizado. Finalmente os dois fatoriais são somados. Veja que o trecho do código que calcula o fatorial está repetido no programa.

```
main:    addi $4,$0, 5
fat1:    addi $8, $0, 1
lab1:    mul $8, $8, $4
         addi $4, $4, -1
         bne $4, $0, lab1
fimFat1: add $2, $8, $0
         add $3, $2, $0
         addi $4, $0, 9
fat2:    addi $8, $0, 1
lab2:    mul $8, $8, $4
         addi $4, $4, -1
         bne $4, $0, lab2
fimFat2: add $2, $8, $0
         add $3, $3, $2
```

(fat1 … fimFat1: Cálculo do Fatorial; fat2 … fimFat2: Cálculo do Fatorial)

Figura 2.19: Trecho de código com cálculos de fatoriais

*A idéia de criarmos um único trecho que calcula o fatorial pode ser bastante proveitosa (e econômica do ponto de vista do tamanho do programa). Este trecho seria então invocado pelo programa principal (que se inicia no rótulo main) o quanto for necessário. O trecho do cálculo do Fatorial seria único em todo o programa e somente invocações (*call*) a este trecho ocorreriam dentro do programa principal. Ao Finalizar a execução do fatorial, o programa retornaria (*ret*) para o seu fluxo normal. A Figura 2.20 mostra o modelo de chamada e retorno do trecho que calcula o fatorial.*

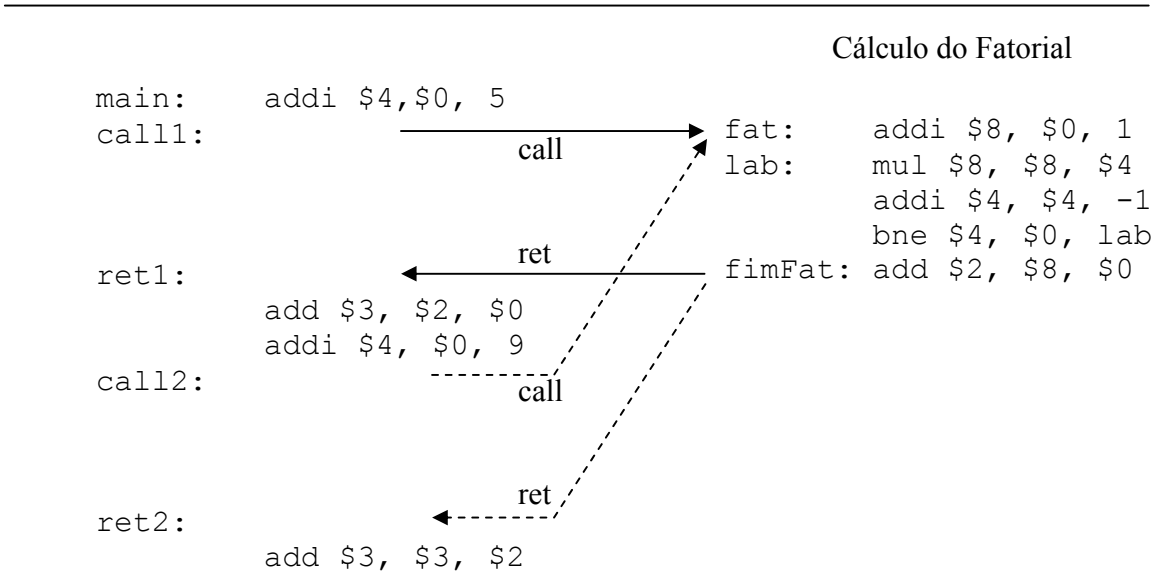


Figura 2.20: Trecho de código com cálculo de fatorial

Nós podemos então chamar este trecho de cálculo do fatorial, de um ***procedimento****,* ***função*** *ou* ***sub-rotina****. Em programação de alto nível, uma função difere de um procedimento pela presença ou não de um dado de saída. Como o cálculo de um fatorial retorna um valor, em HLL, o trecho seria considerado uma função. À parte desta discussão, para nós, vamos chamar de procedimento qualquer trecho de código que tenha uma funcionalidade específica e que possa ser fatorado de um programa.*

Como já sabemos como fazer desvios de fluxo com as instruções do assembly *do MIPS, poderíamos tentar nos arvorar em implementar um procedimento com os nossos conhecimentos atuais. Apesar de mirabulosamente possível, vamos nos deparar com um problema: quando o procedimento terminar, é preciso retornar para algum lugar no código principal e este lugar depende de onde o procedimento foi invocado. No nosso exemplo da Figura 2.19, ao fim da execução do trecho do fatorial*

poderíamos retornar para duas posições possíveis: ret1 ou ret2. Isto depende de onde o procedimento foi invocado: de call1 ou call2. Uma solução para o problema seria guardar em algum lugar a identidade de quem chamou e no momento do retorno, olhando para esta identidade, decidir para onde retornar.

Para implementar esta funcionalidade o MIPS dispõe de 2 instruções de desvio especialmente projetadas para dar suporte a procedimentos: `jal` *(*`jump-and-link`*) e* `jr` *(*`jump register`*).* `jal` *é uma instrução de desvio incondicional como* `j`*, mas além de alterar o valor do PC, ela também guarda em $31 o endereço de retorno do procedimento.* `jr`*, por sua vez, também é uma instrução de desvio incondicional e retorna para o endereço especificado no registrador usado na instrução. Se este registrador é o $31, então ele retorna exatamente para o endereço associado à posição de retorno da chamada específica do procedimento.*

A Figura 2.21 mostra o trecho do código alocado na memória. Observe que o procedimento fatorial (fat), será invocado pelo programa principal com a instrução `jal`*. O retorno do procediemento é implementado com a instrução* `jr` *especificando o seu operando como sendo o registrador $31. Observe também que quando o programa é alocado na memória o valor de PC passa a apontar para o endereço associado ao rótulo main. Por isto, qualquer de nossos códigos precisa ter um rótulo main no seu início.*

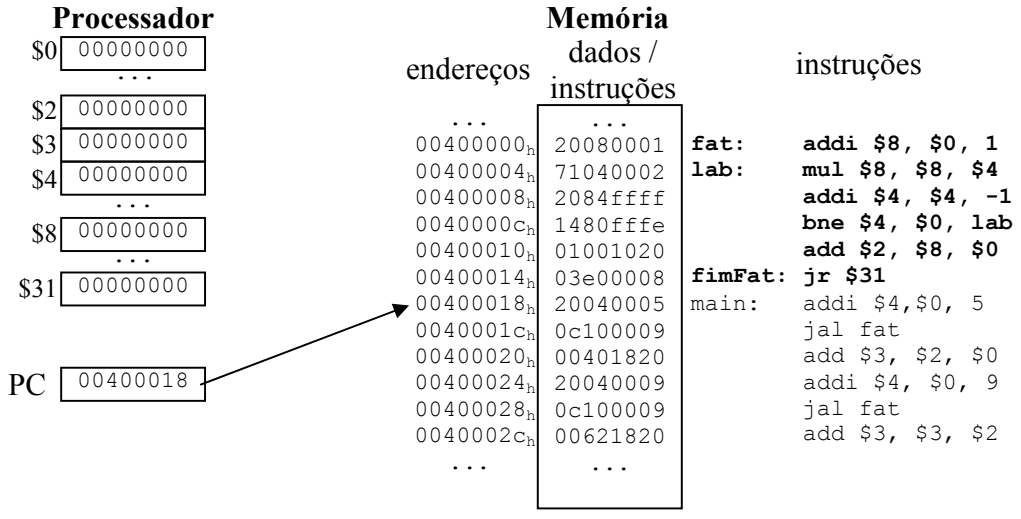


Figura 2.21: Implementação do procedimento fatorial

Agora vamos verificar como ocorre a execução deste trecho de código. A Figura 2.22 mostra o início da execução. Veja que a instrução `jal` *guarda em $31 o endereço de retorno do procedimento, ao mesmo tempo em que vai alterar o valor de PC para apontar para o endereço de início do procedimento.*

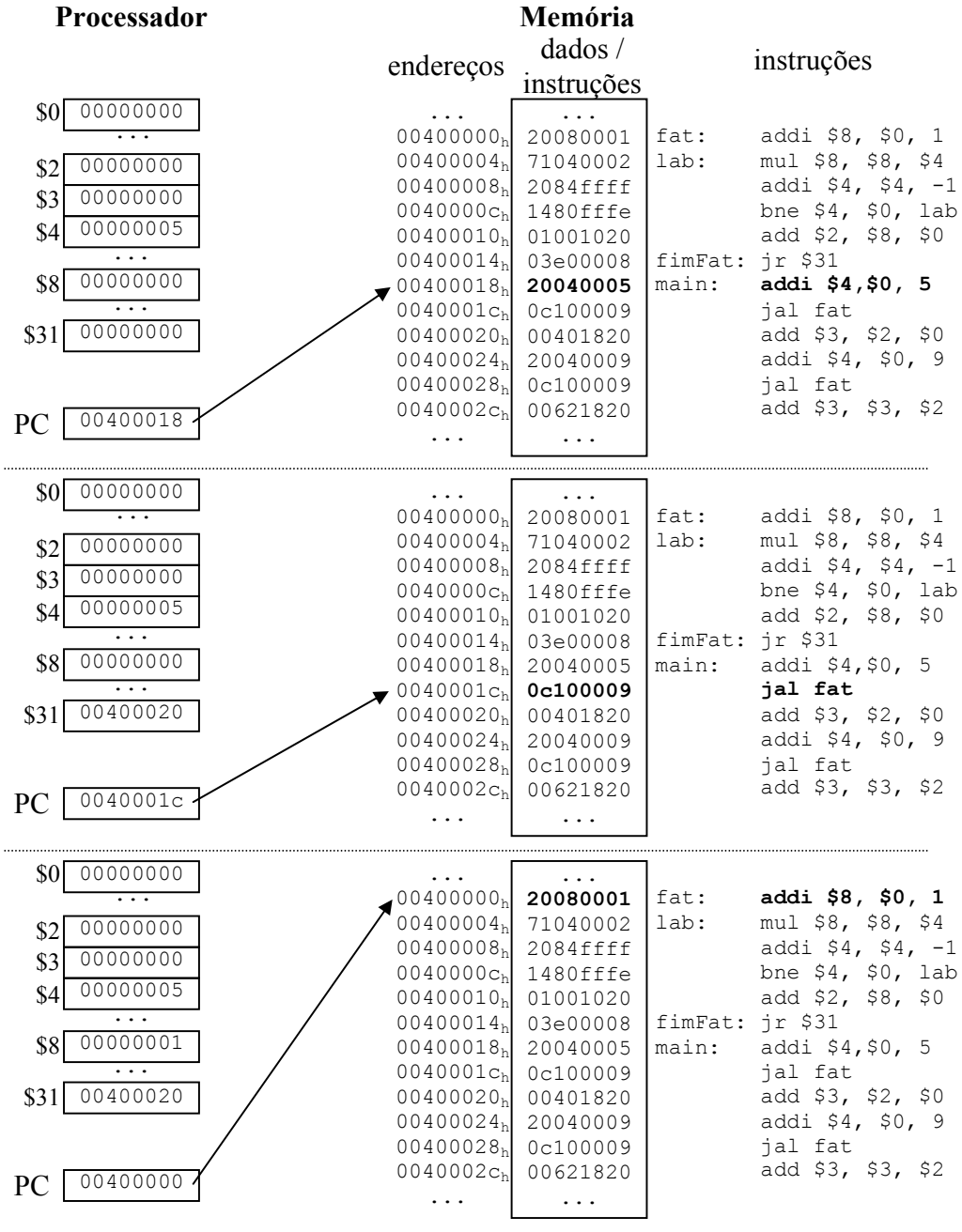


Figura 2.22: Execução do código com procedimento fatorial

Depois de executado todo o procedimento, o resultado é guardado em \$2 (78_h = 120). Daí acontece o retorno usando a instrução `jr`. Ao ser executada ela põe em PC o valor guardado em \$31. Isto faz com que a execução retorne para a instrução seguinte à chamada do procedimento. A Figura 2.23 ilustra a execução deste trecho.

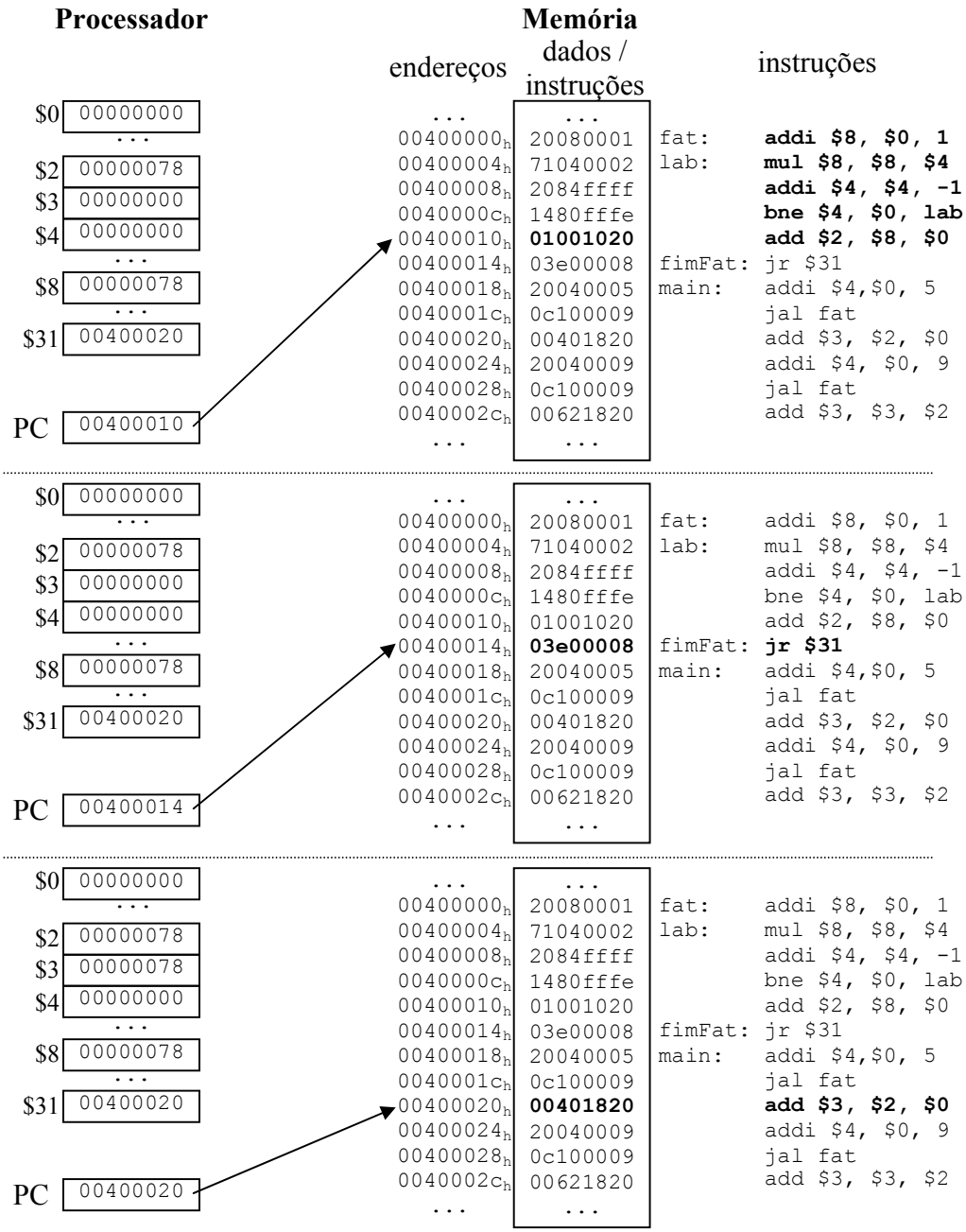


Figura 2.23: Execução do código com procedimento fatorial

Uma vez de volta ao programa principal o valor de \$4 é ajustado para 9. Em seguida a instrução `jal` *mais uma vez invoca o procedimento fatorial. Agora o valor guardado em \$31 é 0040002c$_h$, ou seja, o novo endereço de retorno do procedimento. Então o procedimento é novamente executado.*

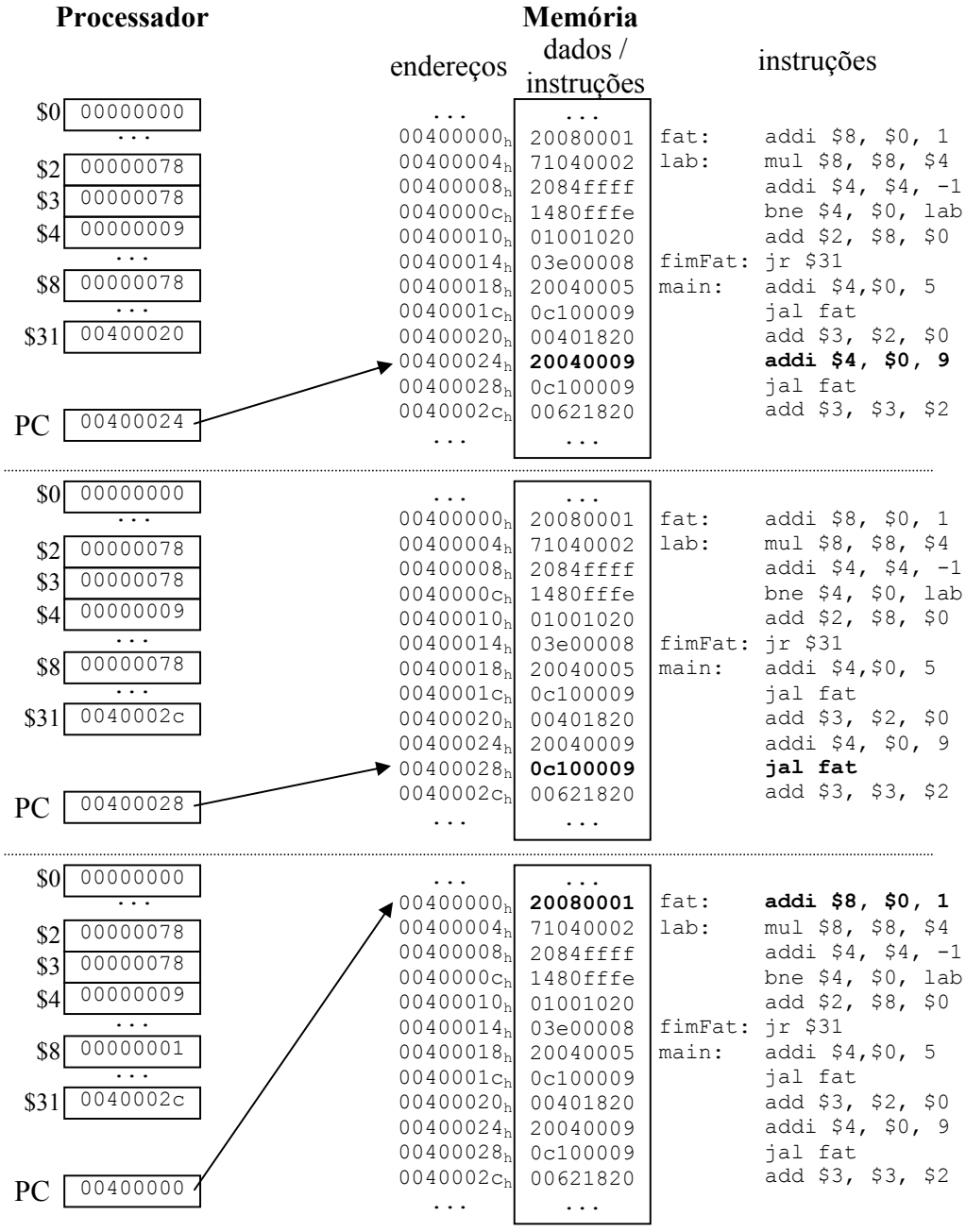


Figura 2.24: Execução do código com procedimento fatorial

Ao fim de mais uma execução do procedimento, a instrução `jr` *altera o valor de PC para o endereço de retorno, anteriormente guardado em $31. quando retorna ao programa principal o novo valor de $2 contém o cálculo do fatorial de 9. Este valor é então somado ao valor que existia anteriormente, propiciando o resultado final do programa: calcular 5! + 9!.*

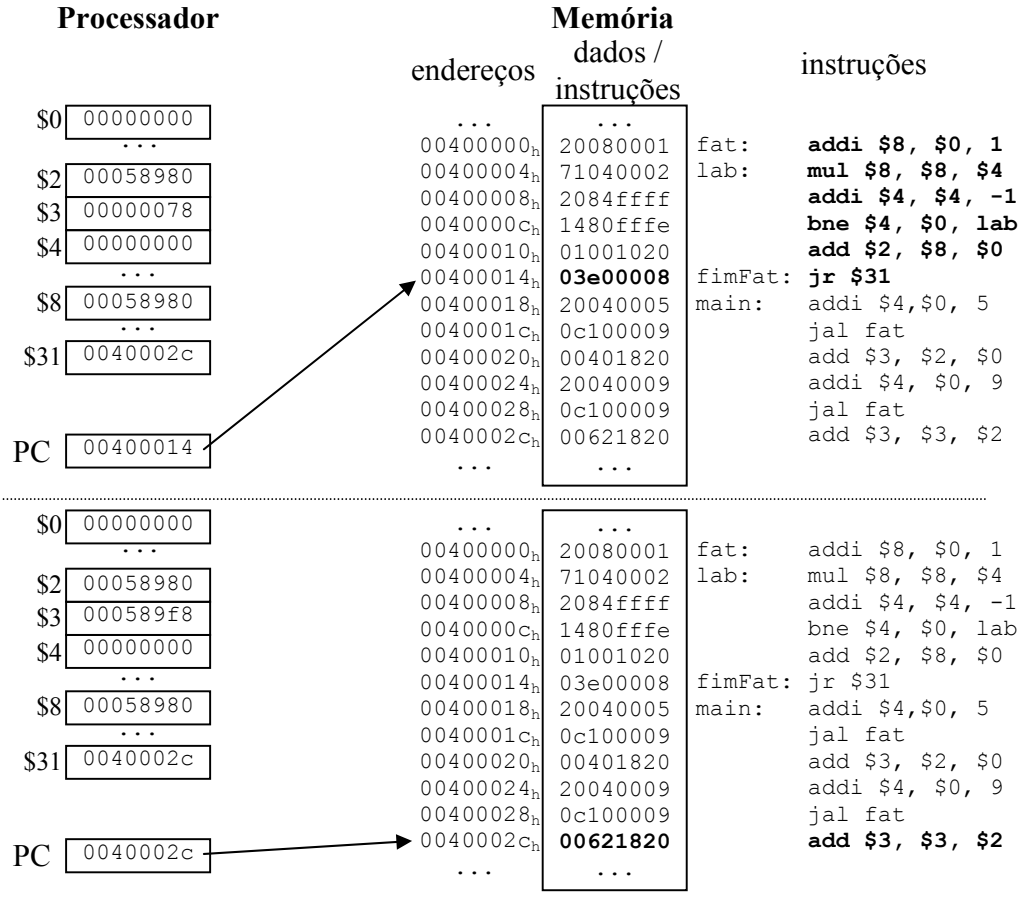


Figura 2.25: Execução do código com procedimento fatorial

Uma observação importante nesta execução é que o programador definiu exatamente em qual registrador deve ser passado para o procedimento o argumento de entrada, no nosso caso, o número sobre o qual será computado o fatorial. O registrador escolhido foi o $4. Também foi acordado entre o procedimento e o programa principal, qual o registrador que portaria o resultado da operação. O registrador escolhido foi o $2.

O MIPS convencionou que os registradores $4 a $7 devem ser usados para passagem de parâmetros para procedimentos. Também os registradores

$2 e $3 são usados, por convenção, para guardar resultados dos procedimentos.

Sumarizando agora, temos as instruções de suporte à implementação de procedimentos no MIPS. A Tabela 2.6 mostra a instruções estudadas.

Categoria	Nome	Exemplo	Operação	Comentários
Suporte a procedimentos	jal	jal rotulo	$31 ← endereço[retorno] PC ← endereço[rotulo]	
	jr	jr $31	PC ← $31	

Tabela 2.6: Instruções de suporte à procedimentos

2.7 – Conclusões

Conhecemos neste capítulo algumas das principais palavras da linguagem de montagem do MIPS. Como qualquer idioma, juntar as palavras para formar uma frase, no nosso caso um programa, requer treino. Muito treino. Decorá-las é de pouca valia por que o nosso interlocutor é uma máquina que só entende se as palavras estiverem corretas e as frases muito bem formadas. O esperado é que haja muito treino da parte do leitor que queira, de fato, se apropriar deste conhecimento.

A Tabela 2.7 serve de gabarito, para ser utilizado enquanto você não se apropria inteiramente das palavras. Além disto, ela também serve para você tirar dúvidas da sintaxe da instrução, agindo como um pequeno dicionário.

Aprendemos também que o cuidado para que um valor possa ser armazenado, sem problemas de *overflow*, nos registradores é de responsabilidade do programador.

Vamos agora ver alguns detalhes adicionais. No MIPS há também um conjunto de instruções, chamadas pseudo-instruções. Elas não existem de fato, mas um programador pode utilizá-las desde que o seu montador ofereça suporte à pseudo-instruções. Um exemplo é `mov $4, $8`. (*move*. Mova o conteúdo do registrador $8 para o registrador $4) O montador interpretaria esta pseudo-instrução e geraria o código de máquina correspondente a `add $4, $0, $8` ou `or $4, $0, $8`. Existem também pseudo-instruções que se desdobram em mais de uma, por exemplo, `bgt $8, $9, label` (*branch if grater then*, salta para *label* se $8 for maior que $9). O montador do MIPS geraria então, em código de máquina, a seguinte seqüência de instruções: `slt $1, $9, $8` e `bne $1, $0, label`. O efeito é o mesmo pretendido.

Categoria	Nome	Exemplo	Operação	Comentários
Aritmética	add	add $8, $9, $10	$8 = $9 + $10	*Overflow* gera exceção
	sub	sub $8, $9, $10	$8 = $9 – $10	*Overflow* gera exceção
	addi	addi $8, $9, 40	$8 = $9 + 40	*Overflow* gera exceção Valor do imediato na faixa entre –32.768 e +32.767
	addu	addu $8, $9, $10	$8 = $9 + $10	*Overflow* não gera exceção
	subu	subu $8, $9, $10	$8 = $9 – $10	*Overflow* não gera exceção
	addiu	addiu $8, $9, 40	$8 = $9 + 40	*Overflow* não gera exceção Valor do imediato na faixa entre 0 e 65.535
	mul	mul $8, $9, $10	$8 = $9 x $10	*Overflow* não gera exceção HI, LO imprevisíveis após a operação
	mult	mult $9, $10	HI,LO = $9 x $10	*Overflow* não gera exceção
	multu	multu $9, $10	HI,LO = $9 x $10	*Overflow* não gera exceção
	div	div $9, $10	HI = $9 mod $10 LO = $9 div $10	*Overflow* não gera exceção
	divu	divu $9, $10	HI = $9 mod $10 LO = $9 div $10	*Overflow* não gera exceção
lógicas	or	or $8, $9, $10	$8 = $9 or $10	
	and	and $8, $9, $10	$8 = $9 and $10	
	xor	xor $8, $9, 40	$8 = $9 xor 40	
	nor	nor $8, $9, $10	$8 = $9 nor $10	
	andi	andi $8, $9, 5	$8 = $9 and 5	Imediato em 16 bits
	ori	ori $8, $9, 40	$8 = $9 or 40	Imediato em 16 bits
	sll	sll $8, $9, 10	$8 = $9 << 10	Desloc. ≤ 32
	srl	srl $8, $9, 5	$8 = $9 >> 5	Desloc. ≤ 32
	sra	srl $8, $9, 5	$8 = $9 >> 5	Desloc. ≤ 32. Preserva sinal
Transferência de dados	mfhi	mfhi $8	$8 = HI	
	mflo	mflo $8	$8 = LO	
	lw	sw $8, 4($9)	$8 = MEM[4 + $9]	
	sw	sw $8, 4($9)	MEM[4 + $9] = $8	
	lui	lui $8, 100	$8 = 100 x 2^{16}	Carrega constante na porção alta do registrador de destino. Zera a parte baixa.
Suporte a decisão	bne	bne $8, $9, rotulo	se $8 ≠ $9 então PC ← endereço[rotulo]	
	beq	beq $8, $9, rotulo	se $8 = $9 então PC ← endereço[rotulo]	
	j	J rotulo	PC ← endereço[rotulo]	
	slt	slt $10, $8, $9	se $8 < $9 então $10 ← 1 senão $10 ← 0	Opera com números sinalizados
	sltu	sltu $10, $8, $9	se $8 < $9 então $10 ← 1 senão $10 ← 0	Opera com números não sinalizados
Suporte a procedimentos	jal	jal rotulo	$31 ← endereço[retorno] PC ← endereço[rotulo]	
	jr	jr $31	PC ← $31	

Tabela 2.7: Instruções do MIPS

O uso de pseudo-instruções é fortemente desaconselhável para aprendizagem, porque a apropriação do conhecimento sobre as instruções reais poderia ser prejudicada.

Ao longo deste capítulo vimos que alguns registradores são utilizados para funções específicas, por exemplo, $2 e $3 são usados para guardar os resultados de procedimentos. Vimos a pouco o $1 sendo usado para implementar uma pseudo-instrução. A fim de garantir uma certa inteligibilidade a respeito do propósito de cada registrador, foram convencionados nomes para cada um deles. $v0 é a mesma coisa que $2, ou seja, é usado para guardar o **v**alor de retorno de um procedimento. Como esta política de uso é universal é possível programar em *assembly* usando os números ou os nomes dos registradores. A Tabela 2.8 mostra os nomes e usos.

Nome do Registrador	Número do Registrador	Uso previsto
$zero	0	Constante ZERO
$at	1	Reservado para tarefas do montador (***a**ssembler **t**asks*)
$v0	2	**V**alores dos resultados dos procedimentos
$v1	3	
$a0	4	**A**rgumentos (parâmetros) a serem passados para procedimentos
$a1	5	
$a2	6	
$a3	7	
$t0	8	Registradores que guardam valores **t**emporários
$t1	9	
$t2	10	
$t3	11	
$t4	12	
$t5	13	
$t6	14	
$t7	15	
$s0	16	Registradores que guardam valores temporários que serão **s**alvos de manipulação por procedimentos.
$s1	17	
$s2	18	
$s3	19	
$s4	20	
$s5	21	
$s6	22	
$s7	23	
$t8	24	Mais **t**emporários
$t9	25	
$kt0	26	Reservados para tarefas do kernel do Sistema Operacional (**k**ernel **t**asks)
$kt1	27	
$gp	28	Apontador global (***g**lobal **p**ointer*)
$sp	29	Apontador de pilha (***s**tack **p**ointer*)
$fp	30	Apontador de quadros (***f**rame **p**ointer*)
$ra	31	Endereço de Retorno de procedimentos (***r**eturn **a**ddress*)

Tabela 2.8: Nomes e atribuições de cada registrador do MIPS

Bom, resta-nos desejar bom trabalho, bons exercícios e boa diversão na solução dos problemas. A programação é um mundo sem fronteiras, então se pode pensar nas mais variadas situações para produção de programas.

2.8 – Prática com Simuladores

A essência deste capítulo é a prática com simuladores. O SPIM é um simulador próprio para nós testarmos nossos programas. Ele tem um interpretador de código *assembly* do MIPS e um micro sistema operacional, para suporte a funcionalidades muito básicas.

Como já mencionamos, um programa é composto de duas partes, as variáveis e os códigos. Para fazer um programa inteligível pelo SPIM é preciso declarar a região que contem dados e a região que contem códigos. Para isto usamos uma **diretiva de montagem**, que vem a ser uma instrução do programador para o montador e não para sua ação de transformação do código. A diretiva `.data` especifica que deste ponto em diante, seguem-se os dados. A diretiva `.text` especifica que deste ponto em diante, estão as instruções propriamente ditas.

Um outro detalhe importante quando utilizamos o SPIM é saber onde ficam, por *default*, alocadas a região de dados e de códigos na memória. A convenção do MIPS é utilizada: códigos começam no endereço 00400000_h e dados em 10000000_h.

Entretanto, um pequeno trecho de instruções do sistema operacional é que de fato está alocado no endereço 00400000_h. Neste trecho existe uma instrução `jal main`. Isto significa que o código produzido pelo usuário vai ficar alocado do endereço 00400024_h em diante.

Também para os dados é preciso tomar cuidado. Embora a região comece no endereço 10000000_h, os dados especificados no seu programa vão, de fato, ficar alocados a partir de 10010000_h.

Para termos idéia de um programa em *assembly*, pronto para ser executado no MIPS, mostramos a seguir, na Figura 2.24, um código que promove a ordenação de um vetor. O Vetor está especificado na memória. Veja que foi utilizada uma diretiva `.word` na região de dados do programa. `.word` indica que cada elemento do vetor ocupará uma palavra de 32 bits. Em nossos exercícios vamos sempre utilizar dados de 32 bits. Um outro detalhe é que todos os elementos do vetor foram declarados em uma única linha, mas na prática, cada um ocupará uma palavra na memória.

```
#The default address for the beginning of
#users data is 0x10010000 = 268500992
#(High=0x1001=4097 and Low=0x0000=0)
#
# This program performs vector ordering
# by EBWN on Sept 17, 2004 11h00

      .data
a:    .word -2, 20, -270, 15, 9, 62, 1, 8, -100
n:    .word 9 # vector size (elements)


      .text
#Every code begins with "main" label
main:
      ori $8, $0, 0     # i in $8
      ori $9, $0, 0     # j in $9
      ori $10, $0, 0    # min in $10
      lui $18, 4097
      lw $17, 36($18)   # n in $s1 ($17)
      addi $19, $17, -1 # n-1 in $s3 ($19)

m1:   slt $18, $8, $19
      beq $18, $0, m3    # if i >= n-1 then quit
      addi $9, $8, 1     # j = i+1;

      ori $18, $0, 4
      mul $11, $8, $18
      lui $18, 4097
      add $15, $18, $11 # calculate address of a[i] (in $15)
      lw $10, 0($15)    # load a[i] in $10

m4:   slt $18, $9, $17
      beq $18, $0, m2    # if j >= n then exit loopint

      ori $18, $0, 4
      mul $11, $9, $18
      lui $18, 4097
      add $18, $18, $11 # calculate address of a[j] (in $18)
      lw $12, 0($18)    # load a[j] in $12

      slt $20, $12, $10 # if a[j] < a[i] swap a[j] with a[i]
      beq $20, $0, dontswap

swap: sw $12, 0($15)
      sw $10, 0($18)
      add $10, $12, $0

dontswap:
      addi $9, $9, 1    # j++
      j m4

m2:   addi $8, $8, 1     # i++
      j m1

m3:   nop                # this is the end
```

Agora, é utilizar o simulador!

2.9 – Exercícios

2.1 – Pesquise na web e produza um resumo sobre como transferir dados de tamanhos diferentes de uma palavra de e para o banco de registradores.
2.2 – Pesquise na web e produza um resumo sobre *big endian e little endian.*
2.3 – Pesquise na web e produza um resumo sobre *pseudo-instruções* do MIPS.
2.4 – Pesquise na web e produza um resumo sobre *instruções privilegiadas* do MIPS.
2.5 – Pesquise na web e produza um resumo sobre a *instrução* `jalr` do MIPS.
2.6 – Pesquise na web e produza um resumo sobre a *instrução* `madd` do MIPS.
2.7 – Pesquise na web e produza um resumo sobre a *instrução* `msub` do MIPS.
2.8 – Pesquise na web e produza um resumo sobre a *instrução* `rotr` do MIPS.
2.9 – Pesquise na web e produza um resumo sobre a *instrução* `seh` do MIPS.
2.10 – Escreva um código em linguagem de montagem para realizar a expressão
`a = b + c - (d - e)` sem precisar utilizar mais de 5 registradores.
2.11 – Por que não existe uma operação `subi` no MIPS?
2.12 – Por que a instrução `addiu $8, $9, -5` contém um erro de semântica?
2.13 – Crie uma seqüência de instruções MIPS para implementar a instrução `mod` do C/Java.
2.14 – Crie uma seqüência de instruções MIPS para implementar a instrução `div` do C/Java.
2.15 – Como implementar uma negação bit-a-bit no MIPS usando apenas uma instrução?
2.16 – Mostre como podemos fazer uma divisão de um número por outro sem utilizar a instrução `div` ou `divu`.

2.17 – Construir um código em *assembly* do MIPS para executar o seguinte código em alto-nível:

```
a = 1;
b = 2;
if (a<=b){
   a= a + b;
}
```

2.18 – Construir um código em *assembly* do MIPS para executar o seguinte código em alto-nível:

```
a = 1;
b = 2;
for (i = 0; i<10; i++){
   a= a + b;
}
```

2.19 – Construir um código em *assembly* do MIPS para executar o seguinte código em alto-nível:

```
a = 1;
b = 2;
for (i = 0; i<10; i++){
   if (a < b){
     a = a + 2;
   }else{
     b = b + 2;
   }
}
```

2.20 – Construir um código em *assembly* do MIPS para executar o seguinte código em alto-nível:

```
a = 1;
b = 2;
i = 0;
while (i<10){
   if (a < b){
     a++;
   }else{
     b++;
   }
   i++;
}
```

2.21 – Construir um código em *assembly* do MIPS para somar dois vetores;
2.22 – Construir um código em *assembly* do MIPS para ordenar um vetor
2.23 – Um produto interno de dois vetores é definido como:

$$\text{prod\_int} = \sum_{i=0}^{n-1} a_i \times b_i$$

onde a_i é o elemento de índice i do vetor a
onde b_i é o elemento de índice i do vetor b.

Fazer um programa em *assembly* do MIPS para calcular o produto interno de dois vetores com 30 elementos.

2.24 – Construir um código em *assembly* do MIPS para somar duas matrizes 3 por 2.
2.25 – Construir um código em *assembly* do MIPS para multiplicar duas matrizes quadradas de dimensões NxN
2.26 – Construir um código em *assembly* do MIPS para calcular uma matriz transposta.
2.27 – Construir um código em *assembly* do MIPS para resolver um sistema de equações de três variáveis.

Capítulo 3

Linguagem de Máquina

3.1 – Introdução

Neste capítulo vamos conhecer melhor a tarefa do montador. Vamos, de fato, conhecer o objeto que ele gera, a linguagem de máquina do MIPS. Já sabemos que existe uma correspondência de um para um entre uma instrução em linguagem de montagem e uma instrução em linguagem de máquina. Sabemos também que no MIPS todas as instruções têm o mesmo tamanho: 32 bits.

Bem, para começarmos a conhecer a linguagem de máquina, vamos abordar a conversão da instrução `add $8, $9, $10` em sua equivalente. Os bits que formam a instrução são divididos em grupos que indicam a operação a ser realizada e os operandos. No MIPS os 6 bits mais significativos (mais a esquerda) formam um campo chamado ***opcode***. Ele pode conter um código de uma instrução em particular, ou um código que indica uma família de instruções. No nosso exemplo da instrução `add`, o *opcode* vale 000000_2 indicando uma família de instruções. Os 6 bits menos significativos são responsáveis então por definir que instrução está codificada. Este campo nós chamamos de função (***function***). Os operandos são indicados em uma ordem particular. O campo `rd` significa registrador de destino, o campo `rs`, registrador de fonte (*source*) e o campo rt, registrador de alvo (*target*), que pode ser destino ou fonte.

A Figura 3.1 mostra os campos da instrução `add $8, $9, $10`. Como o *opcode* vale 000000_2, a instrução é definida pelo campo *function*. Este campo vale 32 (100000_2) para instrução `add`. O campo `rs` indica o registrador que compõe a primeira parcela da soma. O campo `rt`, a segunda. O campo `rd` indica o registrador onde o resultado da soma será colocado.

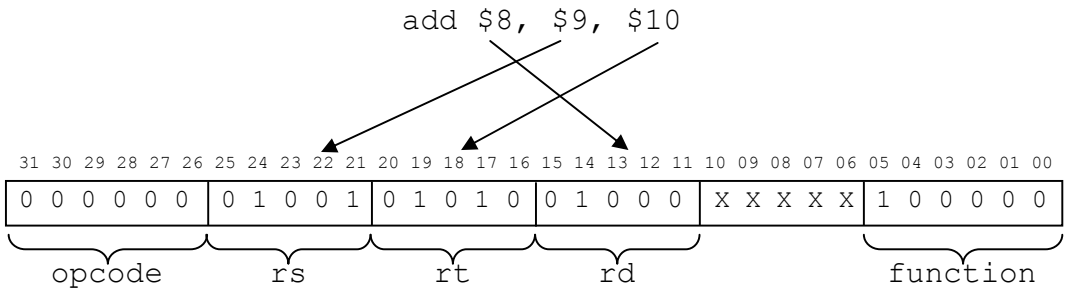


Figura 3.1: Instrução `add $8, $9, $10` em linguagem de máquina

Os campos `rs`, `rd` e `rt` guardam os números dos registradores operandos da instrução. Existe um campo não utilizado entre os bits 6 e 10, inclusive. Veremos que este campo será utilizado para outra funcionalidade, mas por enquanto vamos considerá-lo como sendo sempre 00000_2.

Agora vamos a um exemplo inverso: dado o código de máquina, desejamos encontrar o código *assembly* correspondente. A instrução é dada como sendo $016c820_h$. A primeira coisa a fazer é encontrar a representação binária correspondente. Em seguida vamos procurar os 6 bits mais significativos e descobrimos que eles valem 000000_2. Ora, já sabemos que este valor no campo `opcode` indica uma família de instruções e a instrução dentro desta família é indicada pelos 6 bits menos significativos. Então olhamos para os 6 bits menos significativos, eles indicam 100000_2. Este valor no campo function indica uma instrução de soma. Bem, então agora é só mostrar quais são os campos da instrução e distribuir os bits neles. Daí, encontraremos `rs` valendo 01011_2, `rt` valendo 01100_2 e `rd` valendo 01101_2. Agora reconstruímos a instrução `add rs, rt, rd` = `add $11, $12, $13`. A Figura 3.2 mostra o processo de descoberta da instrução correspondente.

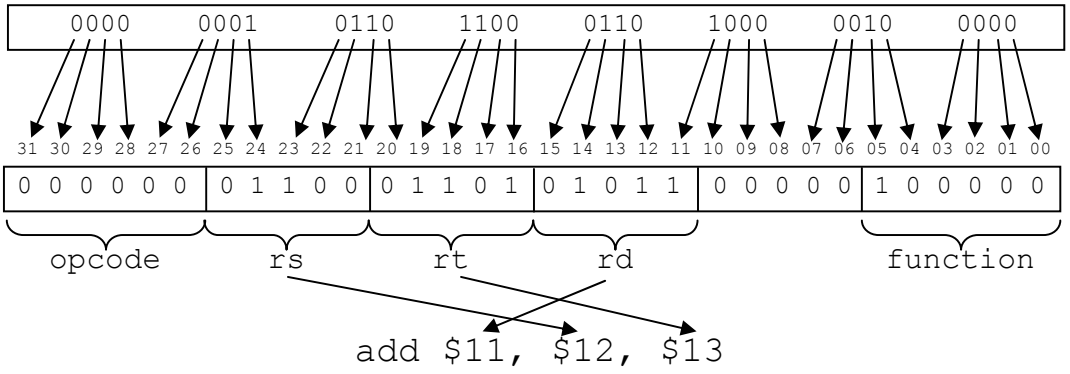


Figura 3.2: Instrução `add $11, $12, $13` em linguagem de máquina

Agora vamos à outra instrução: `sub $8, $10, $11`. Esta instrução também tem um *opcode* igual a 000000_2. Isto indica que ela pertence à mesma família da instrução `add`. O valor do campo `function` é: 100010_2. A codificação segue o mesmo padrão: 6 bits para `opcode`, 5 para `rs`, 5 para `rt`, 5 para `rd`, 5 desconsiderados e 6 para `function`, nesta seqüência necessariamente. A Figura 3.3 mostra a codificação da instrução.

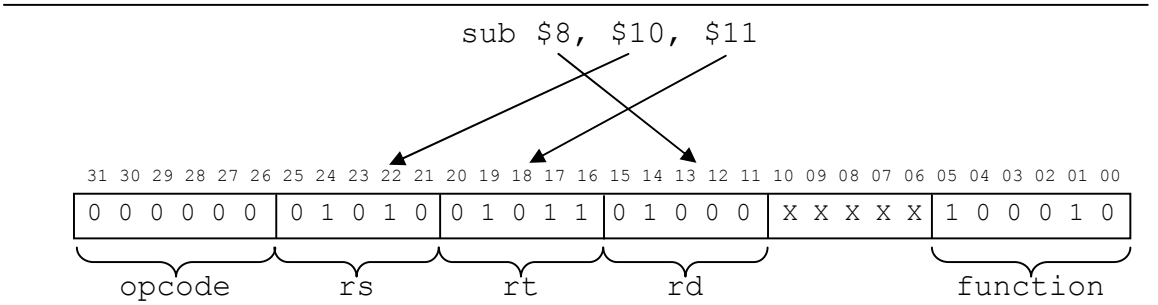


Figura 3.3: Instrução `sub $8, $10, $11` em linguagem de máquina

Para finalizarmos esta família vamos ver a codificação da instrução `sll $8, $9, 3`. Esta também é uma instrução que segue o mesmo padrão de codificação. Observe, entretanto que existe uma quantidade especificada na instrução, que é o montante de bits do dado no registrador `rt` que iremos deslocar para esquerda. `rd` deverá conter o resultado da operação e `rs` é desprezado (colocado em 00000_2). Bem, então fica faltando como especificar a quantidade 3 que representa o montante do deslocamento. Este montante é representado em um campo chamado montante de deslocamento, `sa` (*shift amount*). São usados os bits de 6 a 10 para ser o `sa`. A Figura 3.4 mostra a codificação da instrução `sll $8, $9, 3`, onde `function` vale 000000_2.

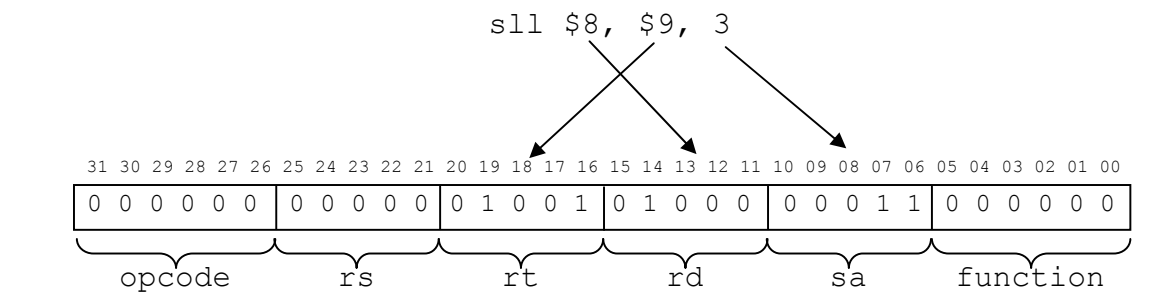


Figura 3.3: Instrução `sll $8, $9, 3` em linguagem de máquina

Este formato de codificação de instruções é compartilhado por muitas instruções e ele recebe um nome: formato R. A Tabela 3.1 mostra a codificação das principais instruções do tipo R. Observe que a instrução `mul`, embora compartilhe este formato, não tem como `opcode` o valor 000000_2.

Nome	Formato	Exemplo	Codificação					
			opcode	rs	rt	rd	sa	function
sll	R	sll $8, $9, 3	0	9	10	8	3	0
srl	R	srl $8, $9, 3	0	0	10	8	3	2
jr	R	jr $8	0	8	0	0	0	8
mfhi	R	mfhi $8	0	0	0	8	0	16
mflo	R	mflo $8	0	0	0	8	0	18
mult	R	mult $9, $10	0	9	10	0	0	24
multu	R	multu $9, $10	0	9	10	0	0	25
div	R	div $9, $10	0	9	10	0	0	26
divu	R	divu $9, $10	0	9	10	0	0	27
add	R	add $8, $9, $10	0	9	10	8	0	32
addu	R	addu $8, $9, $10	0	9	10	8	0	33
sub	R	sub $8, $9, $10	0	9	10	8	0	34
subu	R	subu $8, $9, $10	0	9	10	8	0	35
and	R	and $8, $9, $10	0	9	10	8	0	36
or	R	or $8, $9, $10	0	9	10	8	0	37
slt	R	slt $8, $9, $10	0	9	10	8	0	42
sltu	R	sltu $8, $9, $10	0	9	10	8	0	43
mul	R	mul $8, $9, $10	28	9	10	8	0	2

Tabela 3.1: Instruções da família R

Bem, agora já conhecemos a codificação de diversas instruções no MIPS. Vamos conhecer um outro formato possível. São as instruções do tipo I, onde um dado imediato vem na codificação da própria instrução.

Comecemos com a instrução `addi $8, $9, 3`. Esta instrução especifica uma parcela da soma em um operando e a outra na própria instrução. A Figura 3.4 mostra a codificação utilizada. Um campo

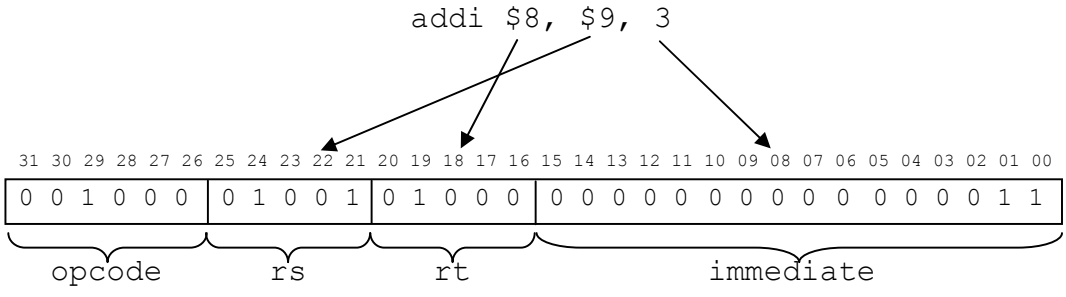


Figura 3.4: Instrução `addi $8, $9, 3` em linguagem de máquina

`immediate` é usado para indicar a quantidade que será somada ao registrador `rs`. O destino é o registrador `rt`. O campo `opcode` indica agora a instrução especificamente, e não uma família.

Observe que o campo `immediate` contém 16 bits, o que implica na limitação de um dos operandos à faixa entre –32768 e +32767. Isto é algo importante na nossa programação, pois uma instrução `addi $8, $9, 128256`, embora pareça válida, não é possível ser codificada.

Outra instrução do tipo I é a `lw`. `lw` usa o campo `immediate`, como sendo um deslocamento a partir de um endereço base especificado em um registrador, como vimos no Capítulo anterior. A Figura 3.5 mostra a codificação de `lw`, que também é seguida por `sw`.

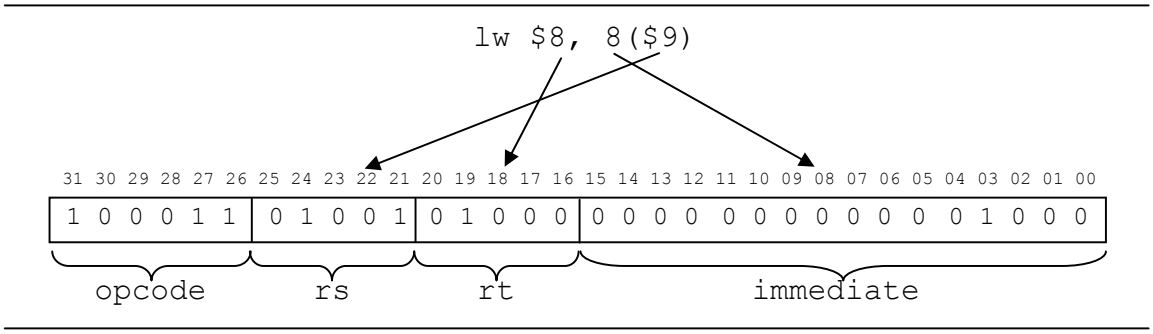


Figura 3.5: Instrução `lw $8, 8($9)` em linguagem de máquina

Finalmente, dentro deste formato existem as instruções `beq` e `bne`. Vamos precisar entender agora como é formado um endereço alvo de uma instrução condicional de salto. Assim como na instrução de carga de registradores existe um endereço base e um deslocamento, na instrução de desvio também existe um deslocamento. O registrador que guarda o endereço base, entretanto, está implícito, o que significa dizer que ele não será declarado na instrução. Bem, este tal registrador é justamente o PC. Então o deslocamento, estipulado no campo `immediate` da instrução de desvio condicional é somado ao valor de PC para informar ao processador qual o endereço da próxima instrução a ser executada.

A Figura 3.6 mostra a codificação da instrução `beq $8, $9, endereço_alvo`. Veja que neste caso um deslocamento, a partir do valor de PC, é aplicado, o que nos leva a refletir que a instrução `beq` é um **desvio condicional relativo a PC**. Este **modo de endereçamento**, é chamado relativo a PC, pois o próximo valor a ser endereçado depende do valor atual do PC.

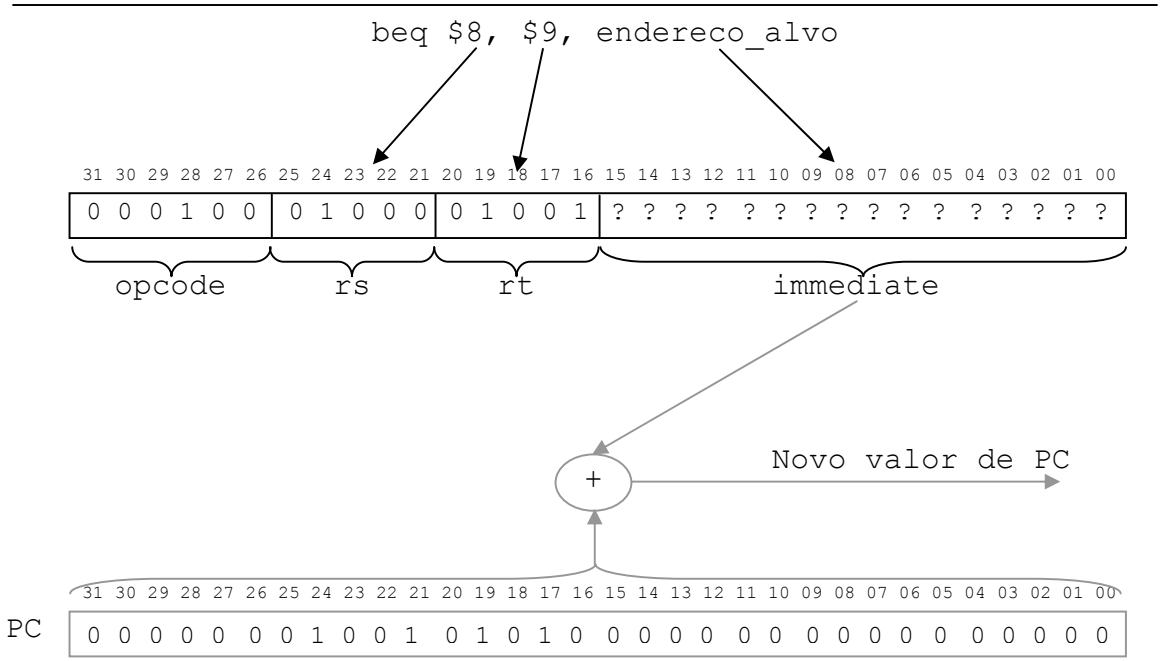


Figura 3.6: Instrução `beq $8, $9, alvo` em linguagem de máquina

Um aspecto muito importante nesta construção de desvios (instruções de saltos, como `beq` e `bne`) é que todas as instruções têm 32 bits, o que implica que a instrução antecessora da atual difere, em seu endereço de –4bytes. Seria um desperdício então utilizar este campo `immediate` para endereçar bytes, já que nunca os bits 0 e 1 seriam diferentes de 0. Por este motivo, o valor imediato especificado na instrução, antes de ser somado ao valor de PC é deslocado duas casas para esquerda, ou seja, o campo `immediate`, no final das contas, indica para quantas instruções antes ou depois da atual deve ser o salto, caso ocorra.

Vamos recorrer a uma adaptação do exemplo da Figura 2.11, mostrada novamente na Figura 3.7. A codificação do `opcode` e dos registradores formam os primeiros 16 bits da instrução: 1109_h. O valor do deslocamento é especificado como a distância entre a instrução atual e o seu alvo. Como o alvo da instrução de salto está duas instruções abaixo, precisamos especificar o campo `immediate` como sendo +2, isto é, 0002_h. Assim a codificação final da instrução é: 11090002_h, como visto na figura.

Quando estudarmos *pipeline* veremos que esta codificação é uma aproximação da realidade, mas vamos deixar esta discussão para o futuro. Por enquanto vamos assumir exatamente este modelo.

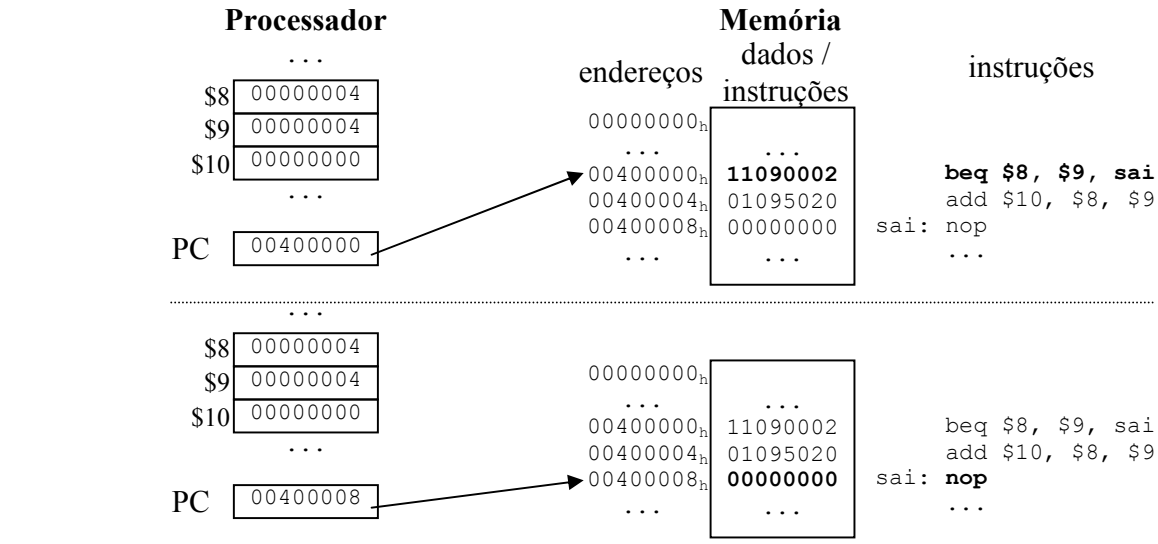


Figura 3.7: Execução e codificação, em linguagem de máquina, da instrução `beq $8, $9, sai`

Vamos fazer um exemplo inverso, dado a codificação em hexadecimal de uma instrução de salto vamos encontrar como seria sua equivalente em linguagem de montagem. A instrução é: $1509fffd_h$. Primeiro começamos encontrando a codificação equivalente em binário. O `opcode` é 000101_2. Se observarmos o exemplo original na Figura 2.11, vamos perceber que se trata do `opcode` da instrução `bne`. Ora, `bne` possui o mesmo formato de codificação de `beq`, então podemos separar os campos `rs` e `rt` para encontrar os registradores que serão comparados, neste caso $8 e $9 e vamos encontrar o alvo. Os 16 bits menos significativos indicam: $fffd_h$. Portanto, este é um número negativo, considerando que sua codificação está em 16bits. O seu equivalente em decimal vale –3. Agora chegamos à conclusão que se trata da instrução `bne $8, $9, -3`. Este –3 é a representação numérica de um rótulo do código. Um possível trecho de código que usa esta instrução é mostrado a seguir e a desmontagem está na Figura 3.8.

```
        ...
rotulo: lw ....
        lw ...
        addi ...
        bne $8, $9, rotulo #alvo à -3 instruções
```

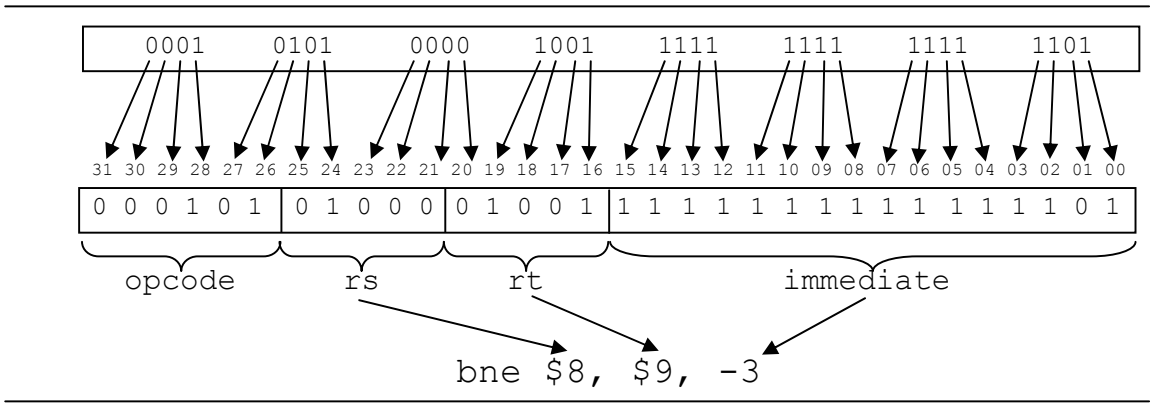


Figura 3.8: Desmontagem da instrução `bne $8, $9, -3` em linguagem de máquina

Uma observação final neste modelo é que podemos saltar para instruções que estão distantes até +32767 ou –32768 palavras da atual.

Vamos sumarizar as instruções do tipo I. A Tabela 3.2 mostra as principais instruções do MIPS que seguem este formato. Cuidado com a especificação dos registradores de `beq` e `bne`, pois eles são colocados em ordem direta na codificação e também, em `lui`, não existe o registrador `rs`.

Nome	Formato	Exemplo	Codificação			
			opcode	rs	rt	immediate
beq	I	beq $8, $9, 3	4	8	9	3
bne	I	bne $8, $9, 3	5	8	9	3
addi	I	addi $8, $9, 3	8	9	8	3
addiu	I	addiu $8, $9, 3	9	9	8	3
slti	I	slti $8, $9, 3	10	9	8	3
sltiu	I	sltiu $8, $9, 3	11	9	8	3
andi	I	andi $8, $9, 3	12	9	8	3
ori	I	ori $8, $9, 3	13	9	8	3
lui	I	lui $8, 3	15	0	8	3
lw	I	lw $8, 4($9)	35	9	8	4
sw	I	sw $8, 4($9)	43	9	8	4

Tabela 3.2: Instruções da família I

Finalmente chegamos ao último formato de instruções admitido pelo MIPS. Trata-se do formato J. Este formato é usado por instruções de desvio incondicional, como `j`, e `jal`. Ele é composto basicamente de um `opcode` e todos os demais bits são utilizados para endereçamento. A Figura 3.9 mostra o exemplo para instrução `j endereço`.

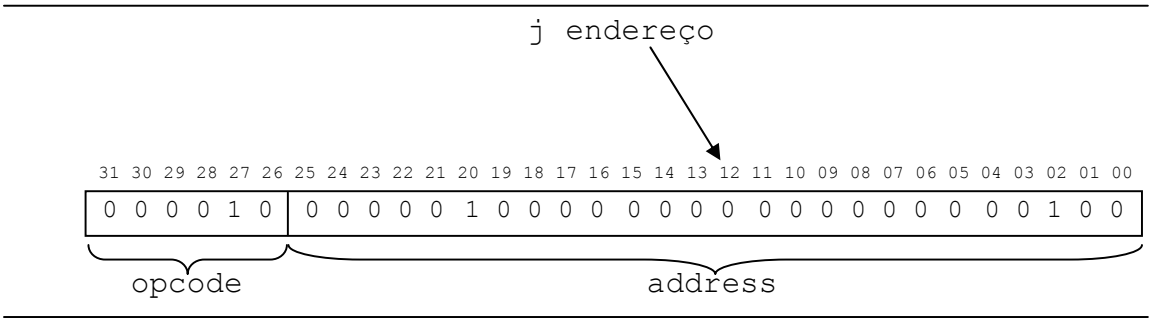


Figura 3.9: Instrução `j endereço` em linguagem de máquina

Assim como as instruções `bne` e `beq` usam um campo de 16 bits que é transformado em 18 bits para ser somado ao PC, os 26 bits do campo `address` deste formato também são deslocados à esquerda para serem transformados em 28 bits. Mas as similaridades param por aí. O cálculo do endereço de alvo do salto incondicional é feito de forma completamente diferente. A idéia é que haja uma independência do valor de PC. Este modo de endereçamento é chamado de **endereçamento direto**.

Infelizmente apenas 28 bits de endereço (26 explícitos e 2 implícitos) são carregados na instrução, o que não permite completar os 32 bits exigidos pela arquitetura. Assim, tomamos emprestado os 4 bits mais significativos do valor de PC para completar o endereço do alvo da instrução de salto. Formamos assim um modo de endereçamento chamado de **endereçamento pseudo-direto** pois ele tende a ser independente do PC, mas precisa utilizar alguns bits dele.

A Figura 3.10 mostra como é feito o cálculo do endereço alvo de uma instrução `j`. O campo *address* da instrução é escrito em PC, independentemente do que ali existia anteriormente. Entretanto, apenas os bits de número 2 a 27 são alterados. O *nibble* mais significativo do valor de PC fica mantido intacto. Veja que os nossos programas ocupam, no modelo de memória estabelecido, entre os endereços 00400000_h até $0fffffc_h$. Isto nos permite uma certa tranqüilidade, pois sabemos que o primeiro *nibble* do endereço contido em PC sempre será 0_h.

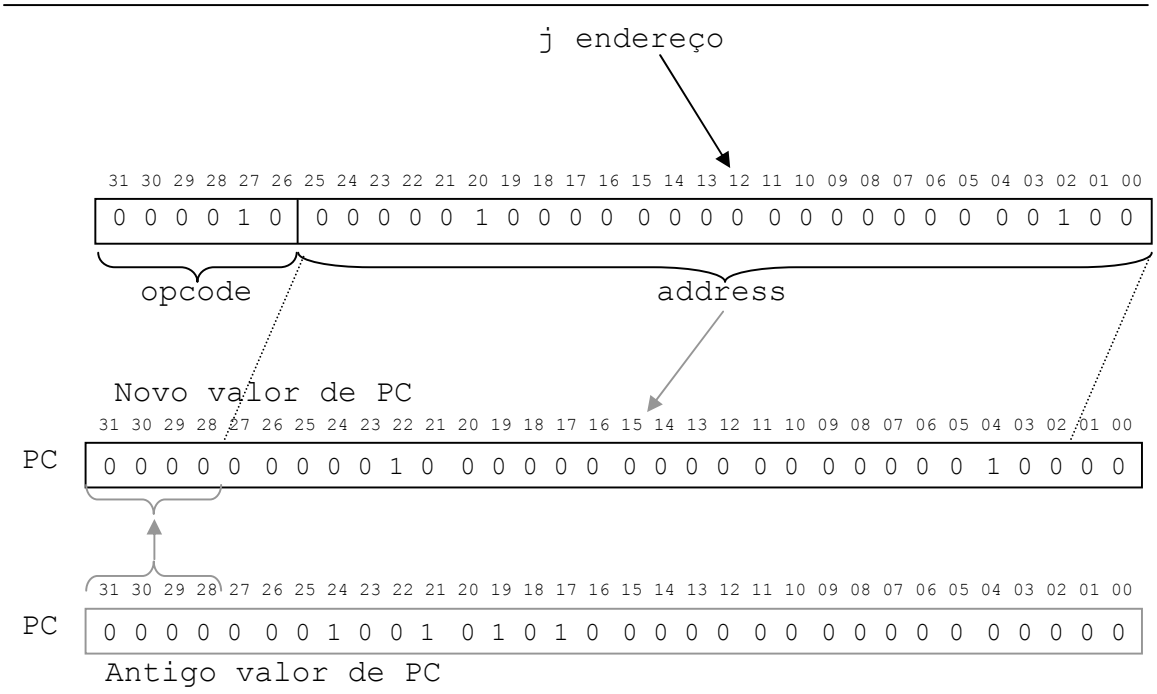


Figura 3.10: Instrução e cálculo de endereço para instrução `j endereço` em linguagem de máquina

Como deduzimos da Figura 3.9, o `opcode` associado à instrução `j` é 000010_2. Agora vamos ver um exemplo de como é feito o cálculo do campo `address` desta instrução. Vamos utilizar o exemplo da Figura 2.15 reproduzida a seguir, como Figura 3.11, para facilitar a leitura. Vamos prestar atenção na codificação da instrução `j sai`. Ela é exatamente a mesma mostrada na Figura 3.9. O Campo `address` contém o valor 0100004_h. Quando o processador vai executar esta instrução ele multiplica este valor por quatro (desloca dois bits à esquerda). Isto nos leva a 0400010_h. Este valor irá sobrepor os 28 bits mais baixos do PC. O primeiro *nibble* do PC fica como estava: 0_h. O endereço de destino passa então a ser 00400010_h. Justamente onde o rótulo `sai` está.

O campo de 26 bits não é operado (somado, subtraído, etc) a nenhum outro valor, portanto, não faz sentido em falarmos de valores sinalizados ou não.

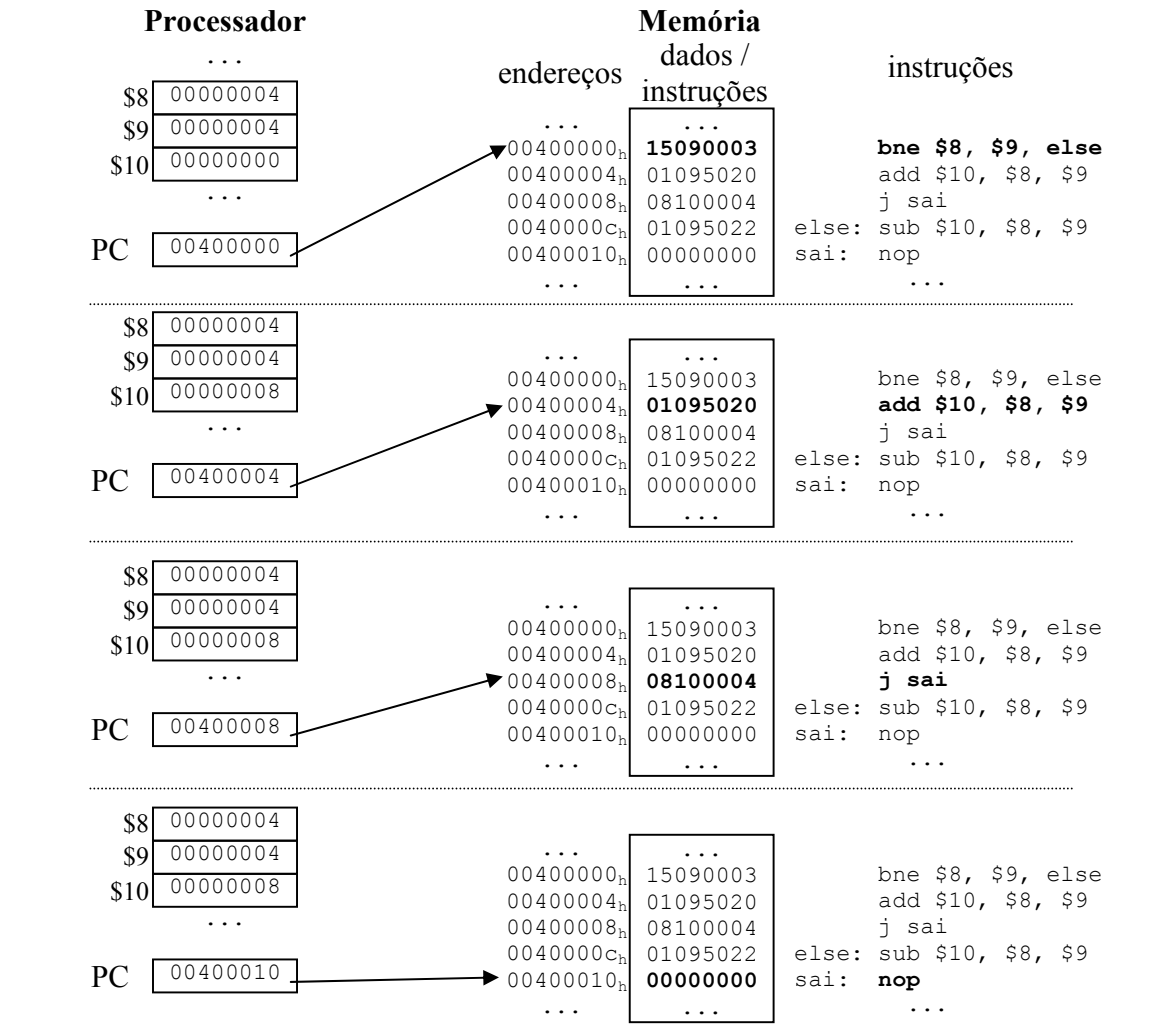


Figura 3.11: Execução de uma estrutura *if-then-else* no MIPS

A instrução `jal` opera exatamente da mesma forma. Apenas a funcionalidade de escrever o valor de PC+4 em `$ra` é acrescentada. A codificação segue a mesma lógica, mas o valor do `opcode` passa a ser 3. A Tabela 3.3 mostra a codificação das instruções do tipo J.

Nome	Formato	Exemplo	Codificação	
			opcode	endereço
j	J	j 1000	2	1000
jal	J	jal 1000	3	1000

Tabela 3.3: Instruções da família J

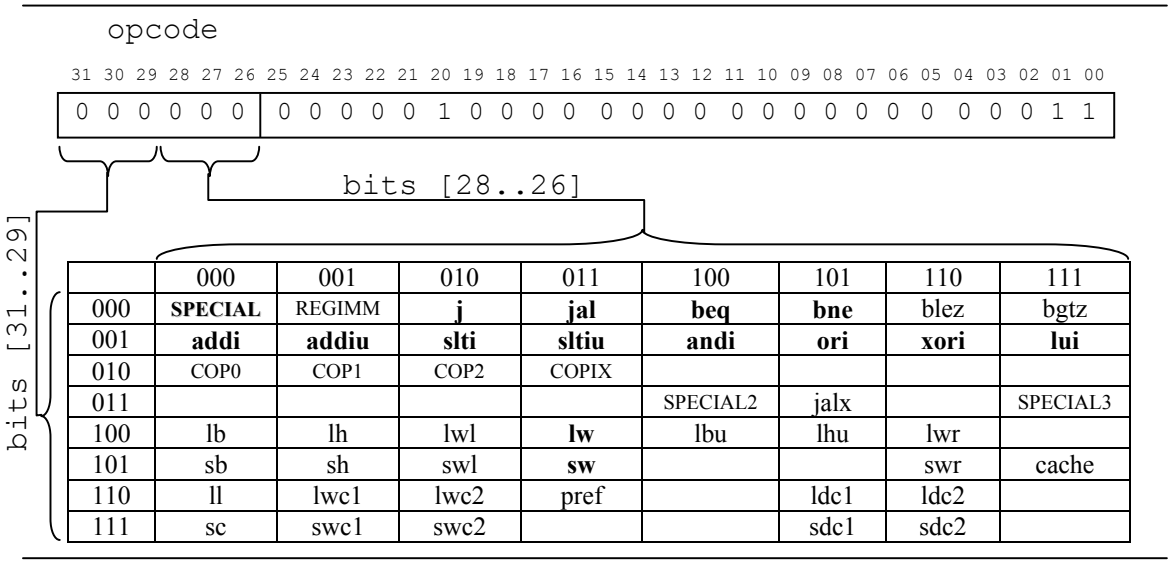
Agora vamos apresentar o quadro completo com os formatos utilizados pelas principais instruções do MIPS. Este quadro é muito didático, mas tende a ficar muito grande se o número de instruções também o for. Por isto apresentaremos também a forma mais convencional encontrada nos manuais das ISAs dos processadores.

Nome	Formato	Exemplo	Codificação					
			opcode	**rs**	**rt**	**rd**	**sa**	**function**
sll	R	sll $8, $9, 3	0	9	10	8	3	0
srl	R	srl $8, $9, 3	0	0	10	8	3	2
jr	R	jr $8	0	8	0	0	0	8
mfhi	R	mfhi $8	0	0	0	8	0	16
mflo	R	mflo $8	0	0	0	8	0	18
mult	R	mult $9, $10	0	9	10	0	0	24
multu	R	multu $9, $10	0	9	10	0	0	25
div	R	div $9, $10	0	9	10	0	0	26
divu	R	divu $9, $10	0	9	10	0	0	27
add	R	add $8, $9, $10	0	9	10	8	0	32
addu	R	addu $8, $9, $10	0	9	10	8	0	33
sub	R	sub $8, $9, $10	0	9	10	8	0	34
subu	R	subu $8, $9, $10	0	9	10	8	0	35
and	R	and $8, $9, $10	0	9	10	8	0	36
or	R	or $8, $9, $10	0	9	10	8	0	37
slt	R	slt $8, $9, $10	0	9	10	8	0	42
sltu	R	sltu $8, $9, $10	0	9	10	8	0	43
mul	R	mul $8, $9, $10	28	9	10	8	0	2
			opcode	**rs**	**rt**	**immediate**		
beq	I	beq $8, $9, 3	4	8	9	3		
bne	I	bne $8, $9, 3	5	8	9	3		
addi	I	addi $8, $9, 3	8	9	8	3		
addiu	I	addiu $8, $9, 3	9	9	8	3		
slti	I	slti $8, $9, 3	10	9	8	3		
sltiu	I	sltiu $8, $9, 3	11	9	8	3		
andi	I	andi $8, $9, 3	12	9	8	3		
ori	I	ori $8, $9, 3	13	9	8	3		
lui	I	lui $8, 3	15	0	8	3		
lw	I	lw $8, 4($9)	35	9	8	4		
sw	I	sw $8, 4($9)	43	9	8	4		
			opcode	**address**				
j	J	j 1000	2	1000				
jal	J	jal 1000	3	1000				

Tabela 3.4: Sumário da codificação das principais instruções do MIPS

A forma apresentada a seguir usa uma abordagem parecida com o que acontece de fato em um processador para que o mesmo possa **decodificar** a instrução. A decodificação significa encontrar a instrução que deverá ser executada para que todas os sinais elétricos sejam enviados à via de dados corretamente. A decodificação no MIPS ocorre em primeiro plano na observação dos 6 bits que formam o `opcode`. A Figura 3.12 mostra a decodificação inicial. Veja que em alguns as células da tabela estão grafadas em maiúsculo. Neste caso, não se trata de uma instrução específica, mas de uma ligação para outra tabela. Nós vamos mostrar apenas a tabela chamada SPECIAL, por isto ela está sublinhada. As instruções que nós estudamos estão destacadas em negrito. As demais instruções e tabelas ficam por conta da curiosidade do leitor.

Veja que neste tipo de tabela não existe informação sobre a codificação do restante dos campos. É preciso o auxílio de uma figura com os formatos e a listagem das instruções para cada formato. Os formatos estão na Figura 3.13.



	000	001	010	011	100	101	110	111
000	**SPECIAL**	REGIMM	**j**	**jal**	**beq**	**bne**	blez	bgtz
001	**addi**	**addiu**	**slti**	**sltiu**	**andi**	**ori**	**xori**	**lui**
010	COP0	COP1	COP2	COPIX				
011					SPECIAL2	jalx		SPECIAL3
100	lb	lh	lwl	**lw**	lbu	lhu	lwr	
101	sb	sh	swl	**sw**			swr	cache
110	ll	lwc1	lwc2	pref		ldc1	ldc2	
111	sc	swc1	swc2			sdc1	sdc2	

Figura 3.12: Tabela de codificação das instruções

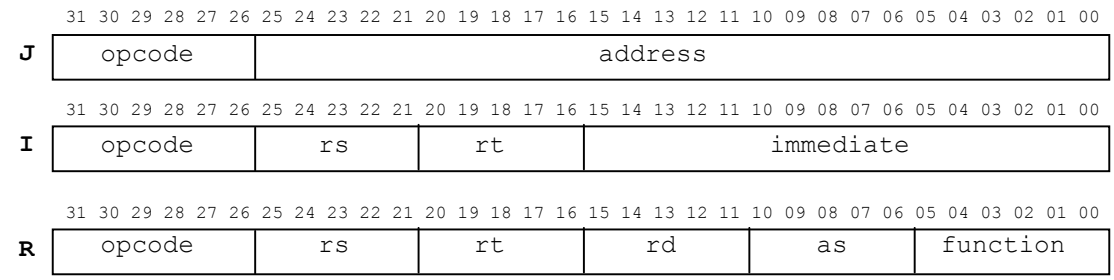
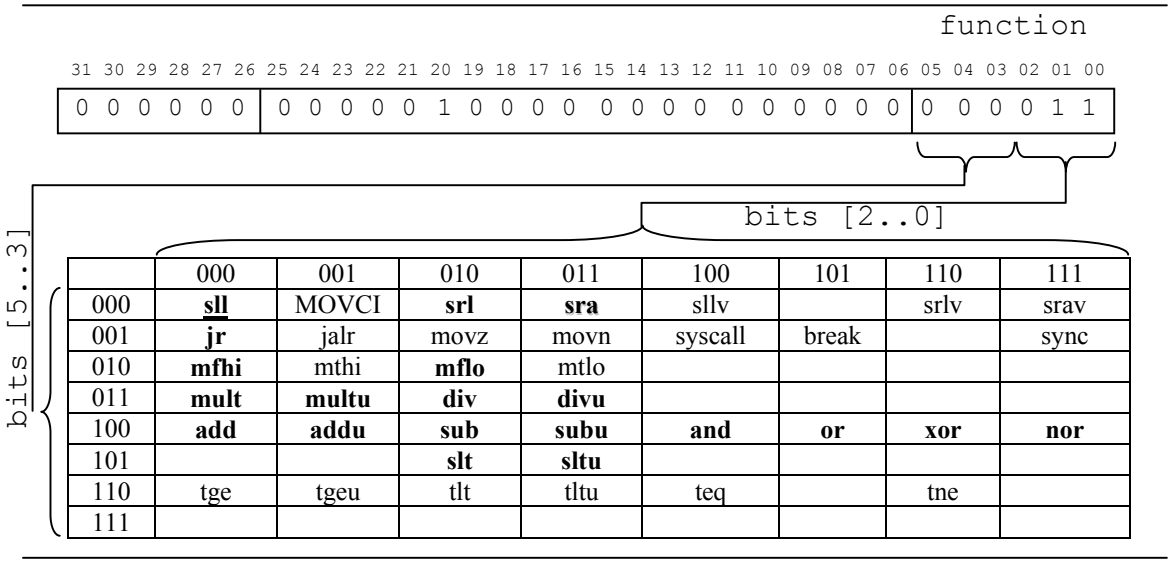


Figura 3.13: Codificação dos formatos



	000	001	010	011	100	101	110	111
000	**sll**	MOVCI	**srl**	**sra**	sllv		srlv	srav
001	**jr**	jalr	movz	movn	syscall	break		sync
010	**mfhi**	mthi	**mflo**	mtlo				
011	**mult**	**multu**	**div**	**divu**				
100	**add**	**addu**	**sub**	**subu**	**and**	**or**	**xor**	**nor**
101			**slt**	**sltu**				
110	tge	tgeu	tlt	tltu	teq		tne	
111								

Figura 3.14: Tabela de codificação das instruções

Uma vez feita a primeira decodificação é preciso saber se ainda é necessário a busca por mais uma tabela. É o caso das células que estão marcadas na tabela original como SPECIAL, entre outras. A Figura 3.14 mostra as instruções presentes nesta tabela. No exemplo da figura iríamos chegar a conclusão que se trata da instrução `sra`, cujo formato é R. Os restantes dos bits seriam decodificados nos campos `rs`, `rt`, `rd` e `as`.

Agora já chegamos a uma fase em que sabemos decodificar e montar qualquer instrução do MIPS (contida no texto). Vamos então abordar outros temas relevantes para este capítulo.

3.2 – Uso da linguagem de máquina

Nas seções seguintes iremos conhecer dois tópicos importantes na formação das linguagens de máquina. Como já mencionamos a complexidade dos sistemas eletrônicos nos levou a construir abstrações. Estas abstrações tampouco são manipuladas sem auxílio de ferramentas. Por isto, hoje ao projetarmos um processador vamos usar, em um primeiro estágio, uma linguagem de descrição de hardware, HDL (***Hardware Description Language***). Esta linguagem é um software capaz de descrever o comportamento de um hardware e de seus componentes. As mais famosas linguagens de descrição de hardware são VERILOG e VHDL. SystemC é recém-chegada ao grupo, mas está tendo um impacto muito importante na comunidade científica hoje.

Uma linguagem de descrição de hardware pode ser utilizada com dois propósitos: simular o comportamento da arquitetura e gerar um protótipo da mesma. Depois de simulada, a arquitetura passa por refinamentos constantes até atingir o ponto de satisfação desejado. Uma vez estável a descrição do hardware, um processo de síntese é iniciado. A síntese é a conversão do modelo descrito em uma HDL para um protótipo físico. Este processo também é feito com uma ferramenta, parecida com um compilador.

O processo de refinamento permite que partes do código sejam melhoradas e/ou detalhadas para que se tenha uma nova simulação e a corretude do modelo seja validada.

O processo de refinamento existe para prover uma melhor especificação do código, um melhor particionamento do projeto e a verificação de erros. Tudo isto, embora seja necessário, acaba consumindo muito tempo no projeto do sistema. Isto implica que o produto demora a chegar ao mercado. A Figura 3.15 Indica o processo básico de simulação e síntese de um projeto desenhado com uma HDL (SystemC).

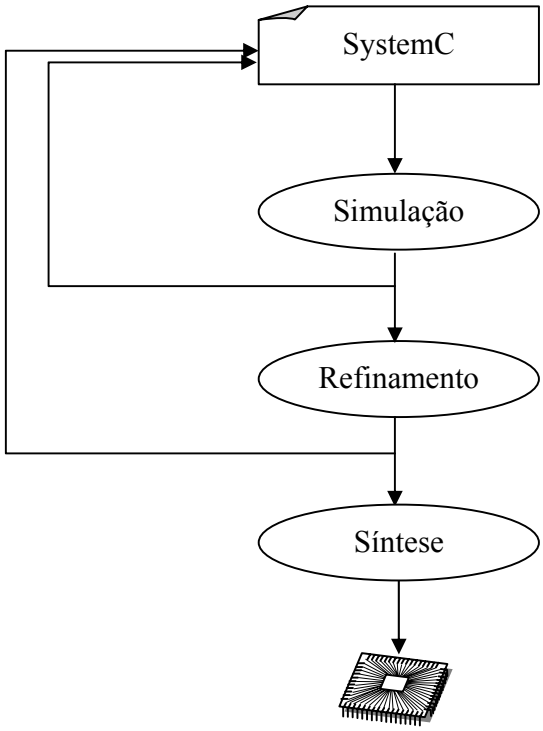


Figura 3.15: projeto de um sistema utilizando HDLs

Um produto para chegar ao mercado e fazer sucesso precisa ter associado a ele um conjunto de ferramentas básicas que permitam ao desenvolvedor de software utilizar os recursos da nova arquitetura. Este *toolkit* deve ser composto, no mínimo, de compiladores, montadores, ligadores e simuladores eficientes. Estas ferramentas, entretanto, são difíceis de serem geradas automaticamente a partir da descrição de um sistema em uma HDL porque as possibilidades de modelos de programação são muito variadas. Assim, estamos vendo hoje uma tendência no desenvolvimento de um projeto onde não mais utilizamos uma HDL, mas passamos a utilizar uma Linguagem de Descrição de Arquitetura, ADL (***Architecture Description Language***).

Um exemplo de ADL conhecida é LISA. Nós vamos estudar uma nova ADL, surgida recentemente, chamada ArchC, que gera código para SystemC. Baseados em ArchC, podemos criar automaticamente um montador e usar a mesma funcionalidade do SystemC para simulação da arquitetura. A geração automática do compilador para ArchC ainda está em desenvolvimento, mas existem técnicas de *re-target* do GCC no site da ferramenta.

Além de apresentar o ArchC na seção seguinte, vamos também explorar a anatomia de uma código executável. Quando compilamos um código, não somente informações de dados e instruções estão presentes no objeto gerado, mas também, e principalmente uma tabela de símbolos que permitem ao Sistema Operacional relocar o código para outras posições de memória. Vamos conhecer um pouco da especificação de um código executável no formato ELF (*Executable and Linking Format*), que é executado normalmente na maioria das distribuições linux.

Ambas as seções fazem parte da visão do software, que é de importância para o conhecimento de um aluno de um curso fortemente baseado em software.

3.3 – A visão do software – ArchC

O ArchC é uma ADL capaz de descrever o comportamento de uma arquitetura em diversos níveis de abstração. O modelo mais simples de descrição é o ***modelo funcional****, onde apenas as instruções são declaradas e suas funcionalidades sobre os recursos básicos da máquina, sem considerar, contudo, as inúmeras implicações que a temporização do modelo pode exercer sobre ele.*

A construção de um modelo funcional traz para o projetista a idéia de que as operações das quais ele precisa estão corretas e realizam as tarefas exatamente como proposto. Uma descrição neste nível de abstração requer muito pouco conhecimento da organização do computador e um modelo

simples de execução das instruções. A medida que refinamentos são feitos até chegarmos a um modelo mais complexo, acurado a ciclos, os recursos vão sendo redefinidos.

Neste capítulo iremos apresentar uma descrição funcional da ISA do MIPS no formato do ArchC.

O ArchC utiliza basicamente dois arquivos de entrada: o primeiro contendo a descrição da arquitetura e o segundo a descrição da ISA. A Figura 3.16 mostra o fluxo de projeto utilizando ArchC e a camada SystemC abaixo, até gerar uma especificação executável.

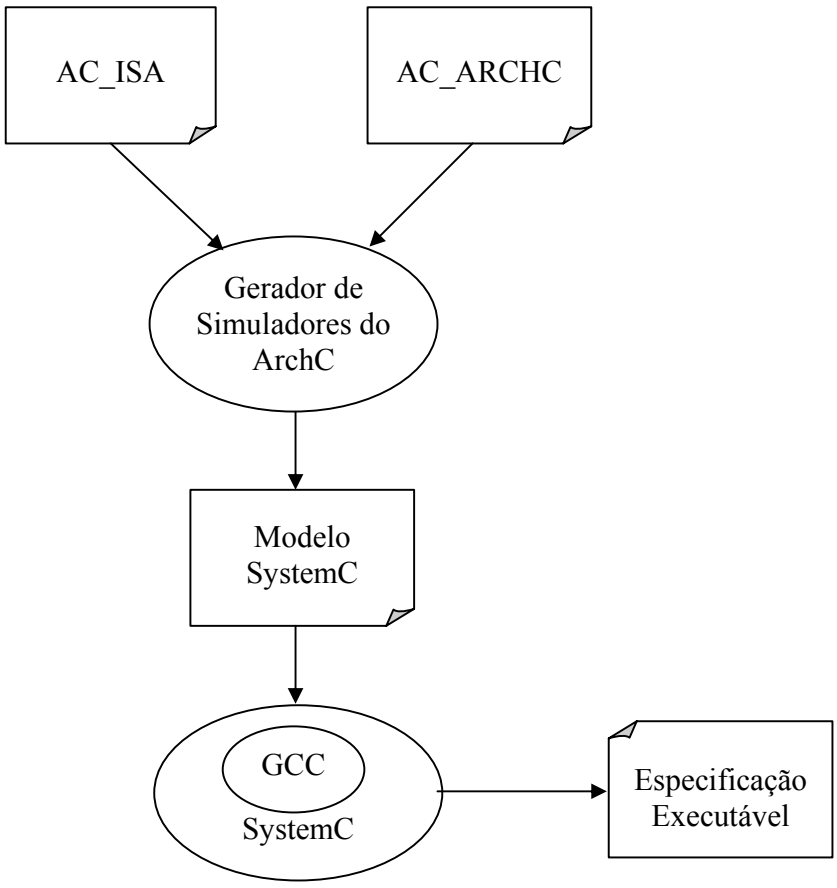


Figura 3.16: fluxo de projeto com ArchC

Dentro do arquivo de descrição do ISA (AC_ISA) o projetista especifica detalhes dos formatos das instruções, tamanhos e nomes e todas as informações necessárias para decodificação e para simulação do comportamento de cada instrução. O arquivo de Recursos de Hardware (AC_ARCH) contém informações acerca da memória, estrutura do pipeline etc.

A descrição de recursos que vamos utilizar é a mais simples possível, tipicamente para implementação de um modelo funcional. O arquivo `mips1.ac` *contém as linhas de código mostradas na Figura 3.17. O tamanho da palavra é 32 bits. Existe uma memória principal de 256kbytes e um banco de registradores com 34 registradores. O construtor do MIPS utiliza o arquivo "mips_isa.ac" para prover as informações sobre as instruções e o* endian *é setado para* big.

```
AC_ARCH(mips){
            ac_wordsize 32;
            ac_mem MEM:256k;
            ac_regbank RB:34;
            ARCH_CTOR(mips){
                  ac_isa("mips_isa.ac");
                  set_endian("big");
            };
};
```

Figura 3.17: Descrição do modelo funcional do MIPS em ArchC

A descrição do modelo funcional do ISA pode ser visto na Figura 3.18. Inicialmente são especificados os tipos e seus campos. Nesta especificação, `%op:6` *indica que existe um campo chamado* `op` *e cujo tamanho é de 6 bits. Os demais campos seguem o mesmo padrão.*

Em seguida são especificadas as instruções que utilizam cada um dos formatos descritos acima. `ac_instr<Type_J> j, jal;` *indica que as instruções* `j` *e* `jal` *possuem sua codificação binária seguindo o formato* `Type_J`.

Depois desta especificação é necessário indicar se existem apelidos para os registradores, o que é feito na seção `ac_asm_reg`.

Finalmente, o construtor do ISA especifica as instruções e a forma de decodificá-las. Por exemplo,

`lw.set_asm("lw %reg, %imm(%reg)", rt, imm, rs);`

indica que a instrução lw é formada por um registrador, um imediato e outro registrador. Os campos utilizados para estes três parâmetros são: `rt`, `imm` *e* `rs`. *Em seguida vem a decodificação que é feita exclusivamente pelo campo*

op, cujo valor deve ser 23_h (0x23 é outra forma de representar o valor em hexadecimal).

```
AC_ISA(mips){
  ac_format Type_R = "%op:6 %rs:5 %rt:5 %rd:5 0x00:5 %func:6";
  ac_format Type_I = "%op:6 %rs:5 %rt:5 %imm:16:s";
  ac_format Type_J = "%op:6 %addr:26";

  ac_instr<Type_R> add, addu, subu, multu, divu, sltu;
  ac_instr<Type_I> lw, sw, beq, bne, addi, andi, ori, lui, sltu;
  ac_instr<Type_J> j, jal;

  ac_asm_reg{
    "$"[0..31] = [0..31];
    "$zero" = 0;
    "$at" = 1;
    "$kt"[0..1] = [26..27];
    "$gp" = 28;
    "$sp" = 29;
    "$fp" = 30;
    "$ra" = 31;
  };

  ISA_CTOR(mips){
    lw.set_asm("lw %reg, %imm(%reg)", rt, imm, rs);
    lw.set_decoder(op=0x23);

    add.set_asm("add %reg, %reg, %reg", rd, rs, rt);
    add.set_decoder(op=0x00, func=0x20);
  };
}
```

Figura 3.18: Descrição do ISA do MIPS em ArchC

Uma vez descritos os dois arquivos, quando processados pelo gerador de simulador do ArchC é criado um arquivo de molde para a definição do comportamento de cada instrução. É também possível especificar um comportamento geral, para todas as instruções ou um comportamento específico para um tipo. Por exemplo, um trecho do molde do modelo do MIPS preenchido pode ser visto na Figura 3.19. A instrução lw por exemplo tem o seu comportamento descrito como,

```
RB[rt] = DM.read(RB[rs]+ imm);
```

Isto significa que o registrador apontado por `rt` *no banco de registradores irá receber o conteúdo da memória no endereço formado pela soma do valor* `imm` *da instrução e do valor guardado no registrador apontado por* `rs` *do banco de registradores.*

```
...
void ac_behavior( instruction ){
  ac_pc = ac_pc +4;
};

void ac_behavior( Type_R ){};

void ac_behavior( lw )
{
  RB[rt] = DM.read(RB[rs]+ imm);
};

void ac_behavior( slti )
{
  // Set the RD if RS< IMM
  if( (ac_Sword) RB[rs] < (ac_Sword) imm )
    RB[rt] = 1;
  // Else reset RD
  else
    RB[rt] = 0;
};
...
```

Figura 3.19: Comportamento do ISA do MIPS em ArchC

Veja também que existe um comportamento associado a todas as instruções: a soma do valor de PC (`ac_pc`). Esta variável pode ser reescrita por uma descrição de um comportamento em particular, como é o caso das instruções de salto.

Trechos das especificações do MIPS em ArchC foram apresentadas. Resta o leitor explorar a ADL para concluir a especificação de, ao menos, as instruções apresentadas até o presente momento.

3.4 – A visão do software – ELF

O formato de ligação e executável foi desenvolvido originalmente pela UNIX System Laboratories*. O Formato ELF foi um padrão projetado para ser executado em arquiteturas Intel de 32 bits numa variedade de sistemas operacionais. Hoje a plataforma que popularizou o ELF é a combinação da arquitetura Intel com o sistema operacional linux. Isto não significa que outras máquinas, de outras famílias não possam executar um arquivo neste formato.*

Existem três tipos de objetos que podem ser representados neste formato:

*- **um arquivo relocável**, que contém dados e instruções para serem ligados com outros arquivos objetos para criar um programa executável ou um arquivo objeto compartilhado;*

*- **um arquivo executável**, que contém informações suficientes para a criação de um processo pelo sistema operacional; e*

*- **um arquivo objeto compartilhado**, que contém informações suficientes para ligação em duas formas: o ligador pode ligá-lo com outro objeto compartilhado ou relocável; ou o ligador pode ligá-lo com um arquivo executável.*

Nós vamos nos ater ao formato de arquivo executável. Um arquivo executável normalmente contém algumas seções (trechos) de códigos e dados, que podem ser agrupados em segmentos. Tanto as seções como os segmentos estão, via de regra, dentro do próprio arquivo executável. Existe também um cabeçalho das seções, que é de fato uma tabela contendo informações sobre a localização, tipo e tamanho de cada seção; um cabeçalho de segmentos (chamado cabeçalho de programas), que indica localização, tipo e tamanho do segmento; e um cabeçalho geral que indica onde está cada cabeçalho no código.

A Figura 3.20 mostra a visão geral do formato. Interessante perceber que os segmentos possuem informações necessárias para o processo de carregamento do Sistema Operacional, portanto, um arquivo só será executável se possuir o cabeçalho de programas. Por outro lado, para ser ligável, é preciso utilizar informações particulares das seções, então um arquivo para ser ligado a outro pelo ligador precisa ter um cabeçalho de seções.

Os cabeçalhos geral, de seções e de programa possuem uma definição extremamente rígida e precisam seguir o padrão para que os conteúdos de cada parte do objeto sejam corretamente interpretados.

O cabeçalho geral é uma estrutura contendo 52 bytes que definem as informações contidas no arquivo ELF. Dentro deste cabeçalho estão definidos os deslocamentos, em bytes, desde o início do arquivo, do cabeçalho de programas e do cabeçalho de seções.

Figura 3.20: visão geral dos conteúdos de um arquivo executável e ligável no formato ELF

O cabeçalho de programas é uma tabela contendo em cada entrada informações sobre o segmento associado. O cabeçalho de seções segue o mesmo padrão, ou seja, contém uma tabela onde cada entrada contém informações sobre a seção associada. A Figura 3.21 ilustra esta hierarquia. Veja que no Cabeçalho Geral estão contidas as seguintes informações sobre o Cabeçalho de Programa: deslocamento desde o início do arquivo onde começa tal cabeçalho; tamanho de cada entrada no cabeçalho (todas as entradas têm o mesmo tamanho); e número de entradas na tabela.

Por sua vez, cada entrada do Cabeçalho de Programas tem, entre outras informações, o deslocamento (em bytes) desde o início do arquivo onde se encontra o Segmento associado e o tamanho deste Segmento. Isto ocorre também, de forma similar, com o Cabeçalho de Seções.

Vamos a seguir verificar o que existe de fato dentro de cada parte dos cabeçalhos.

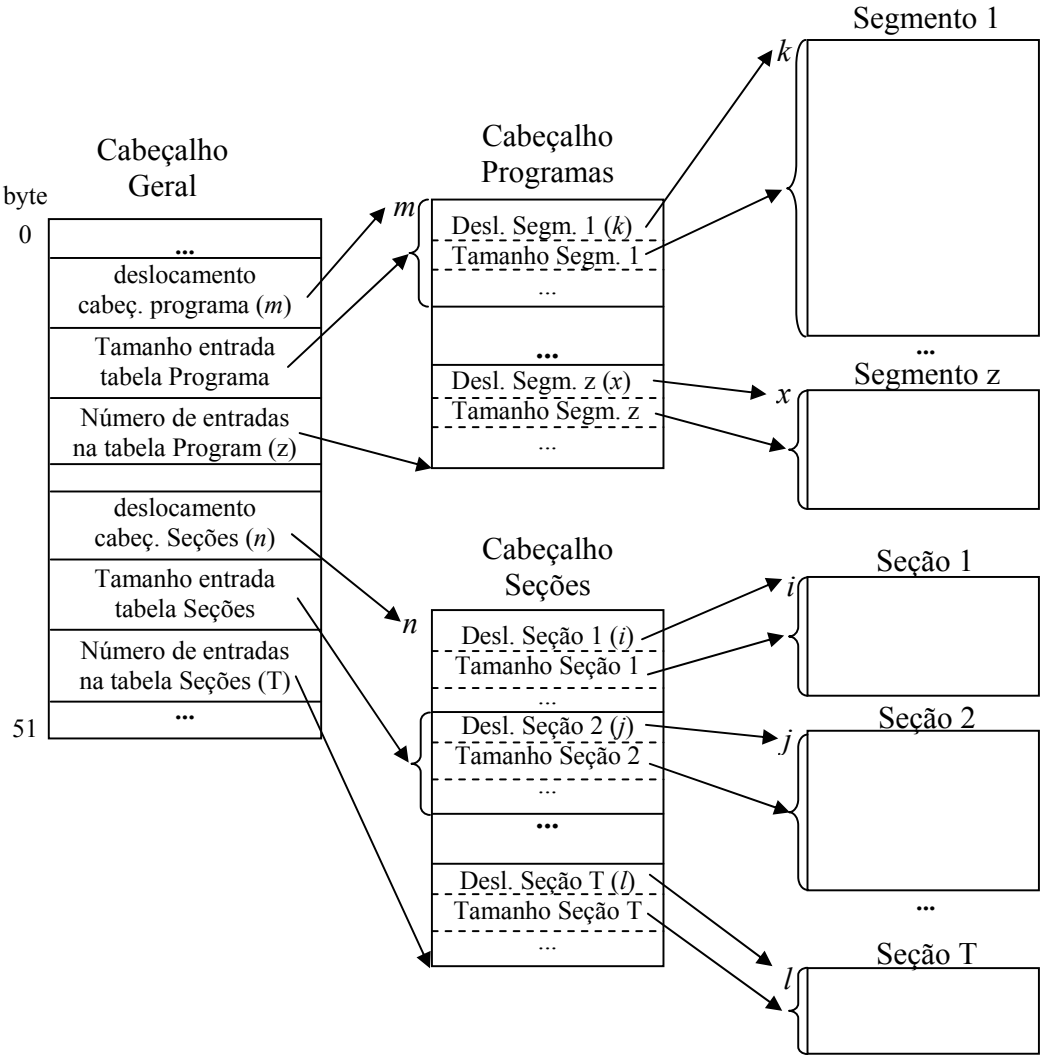


Figura 3.21: Cabeçalhos, Segmentos e Seções no formato ELF

O cabeçalho geral contém no primeiro byte um número específico: $7f_h$. Isto é obrigatório para identificação do tipo do arquivo, bem como a seqüência de bytes 45_h, $4c_h$, 46_h (que são os valores associados às letras E, L e F). Em seguida um byte especifica a `classe` *do arquivo: 0, classe inválida; 1, objeto de 32 bits; e 2 objeto de 64 bits. Atualmente a maioria das máquinas utilizam a classe 1. O próximo campo indica a forma como um número de 32 bits está organizado nos bytes. No caso de* **little endian** *o endereço de menor valor recebe o byte menos significativo, ou seja, mais à direita. No caso de* **big endian**, *o endereço menor recebe o byte mais*

significativo. Máquinas Intel normalmente utilizam little endian *e máquinas RISC como SPARC normalmente usam* big endian*. Há também máquinas que admitem os dois, como a MIPS. Finalmente chegamos à* `versão` *do cabeçalho usado. Hoje só utilizamos o valor 1. Os demais bits até o valor 15 podem ser usado para futuras especificações do ELF. No presente eles são postos todos em 0.*

Podemos já observar estes campos na Figura 3.22. O número apresentado sobre cada campo representa o byte do arquivo. Não há campos menores que um byte na especificação do ELF.

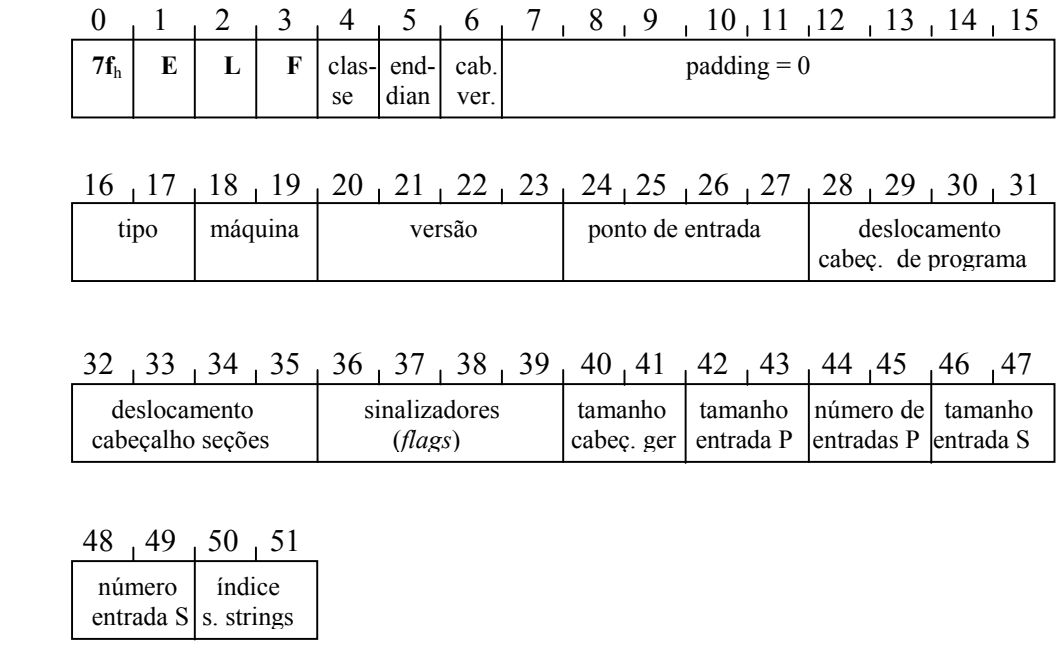


Figura 3.22: Cabeçalho Geral no formato ELF

O campo `tipo` *indica o tipo de objeto no formato ELF: 0, sem tipo; 1, arquivo de relocação; 2, arquivo executável; e 3, arquivo objeto compartilhado. Os nossos executáveis vão possuir então este campo assinalado como 1.*

O campo `máquina` *indica para qual máquina foi criado aquele objeto: 0, nenhuma; 1, AT&T WE 32100; 2, SPARC; 3, Intel 80386; 4, Motorola 68000; 5, Motorola 88000; 7, Intel 80860; e 8, MIPS RS3000. Outras máquinas podem ter valores associados no futuro.*

O campo `versão`*, indica a versão do objeto. A este campo é atribuído o valor 1 correntemente.*

O campo `ponto de entrada` *indica o endereço onde será iniciada a execução do programa. Um possível valor seria* 00400000_h *para o MIPS.*

Os próximos dois campos indicam em qual byte do arquivo começam as tabelas (cabeçalho) de programas e de seções respectivamente.

Os sinalizadores referem-se a bits de sinalização para máquina que irá executar o código. Este valor normalmente é deixado em 0.

Os bytes 40 e 41 informam o tamanho, em bytes, do cabeçalho geral e contém sempre o valor 34_h *(52).*

Os bytes 42 e 43 indicam o tamanho de cada entrada no cabeçalho de Programas, seguidos pelos bytes 44 e 45 que indicam quantas são estas entradas.

Os bytes 46 a 49 são os análogos à tabela de programas, mas agora associados com o cabeçalho de seções.

*Finalmente está indicado nos últimos bytes deste cabeçalho geral o número da seção que contém informações sobre os símbolos utilizados no programa, como por exemplo, o rótulo dos alvos de instruções de desvio e os nomes dos procedimentos. Esta seção especial contém cadeias de caracteres (*strings*) terminadas com o byte 0.*

Vamos agora analisar os dados presentes no cabeçalho de programas. A Figura 3.23 indica os campos presentes em cada entrada do cabeçalho. O campo `tipo` *indica que tipo de segmento está associado a esta entrada. O tipo mais comum é o 1, que indica um segmento que será carregado na memória para execução. O tamanho do segmento é especificado no campo* `tamanho segmento` *e a quantidade de memória necessária é especificada no campo* `tamanho memória`*. Em alguns casos o tamanho da memória requerida é maior que o tamanho do segmento, mas nunca ao contrário.*

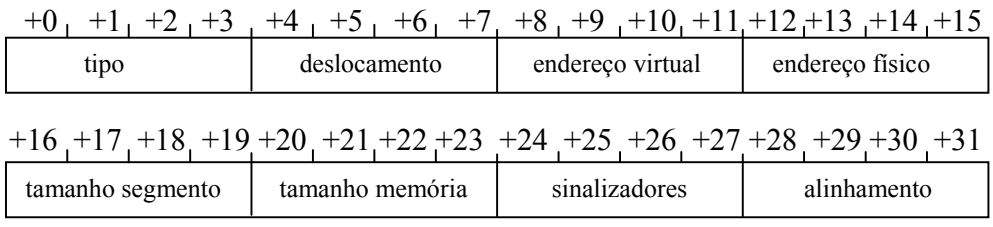


Figura 3.23: Cabeçalho de programas no formato ELF

O campo `endereço virtual` *diz o endereço virtual que deve ser utilizado para receber a primeira palavra do segmento (veremos o que é endereço virtual no capítulo 6).*

O campo `endereço físico` *diz o endereço físico que deve ser utilizado para receber a primeira palavra do segmento (veremos o que é endereço físico no capítulo 6).*

O campo `deslocamento` *indica o byte do arquivo onde se encontra o primeiro byte do segmento. Os demais campos ficam por conta da curiosidade do leitor em saber mais.*

Esta estrutura de dados se repete tantas vezes quantas forem as entradas do cabeçalho de programa.

Informações semelhantes são encontradas em cada entrada do cabeçalho de seções. Vamos analisar os principais campos de uma entrada como apontados na Figura 3.24.

O campo `nome` *indica um deslocamento dentro da seção especial de cadeias de caracteres onde está o nome da seção.*

O campo tipo indica o tipo de seção. Existem muitas definições de tipos possíveis. Os principais estão mostrados na Tabela 3.5. Certamente a de maior impacto é a de bits de programa, que contém as informações das instruções e dos dados do programa.

Os sinalizadores de cada seção são muito importantes. Existem 3 sinalizados específicos: WRITE, valor 1, ALLOC, valor 2 e EXECINSTR, valor 4. WRITE significa que a seção contém dados que podem ser escritos (além de lidos). ALLOC significa que a seção irá ocupar espaço na memória e EXECINSTR significa que a seção contém instruções em linguagem de máquina. Os sinalizadores podem ser usados em combinações. Por exemplo, o valor 6 (2+4) indica que a seção é EXECINSTR e ALLOC ao mesmo tempo.

O campo `endereço` *indica o endereço na memória onde esta seção deve aparecer, caso ela aloque espaço na memória.*

O campo `deslocamento` *indica o primeiro byte onde está, no arquivo, a seção associada.*

O campo `tamanho` *indica o tamanho da seção, em bytes. Os demais campos ficam como exercícios.*

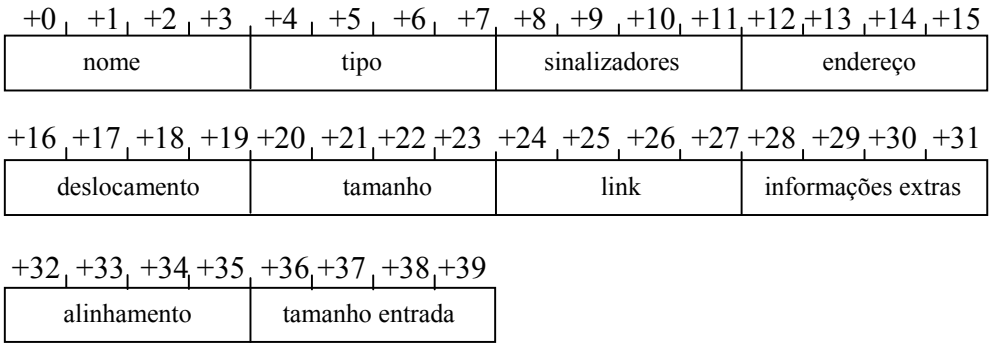


Figura 3.24: Cabeçalho de seções no formato ELF

Tipo	Valor	Descrição
NULL	0	Seção inexistente
PROGBITS	1	Bits do programa
SYMTAB	2	Tabela de Símbolos
STRTAB	3	Tabela de Strings
RELA	4	Informações de Relocação
HASH	5	Tabela de Espalhamento de Símbolos
DYNAMIC	6	Ligação dinâmica
NOTE	7	Anotações sobre o arquivo
NOBITS	8	Não ocupa espaço no arquivo
REL	9	Informações de Relocação
SHLIB	10	Sem semântica específica

Tabela 3.5: Tipos de Seções do ELF

Conhecer esta anatomia de um arquivo executável é importante para aqueles que querem aprender um pouco mais sobre programação. Aqui apresentamos um único formato, mas existem outros especificados, por exemplo, para o Sistema Operacional DOS e Windows. Ainda, existe outra anatomia para Java Bytecodes. Conhecer uma delas é importante para que as demais possam ser mais facilmente entendidas.

3.5 – Conclusões

Conhecemos neste capítulo como é a codificação binária de uma porção das instruções do MIPS. Não tratamos, entretanto de todas as instruções, como instruções de ponto flutuante ou de coprocessador. O que importa neste instante é que o leitor possa fazer uma tradução de código de máquina para código de montagem e vice-versa.

Vimos também que o número de formatos disponíveis no MIPS é de três. Isto pode parecer um número muito pequeno, mas existe um benefício muito importante associado. A decodificação de uma instrução é feita pela máquina e quanto mais formatos existirem, mais possibilidades de combinações de bits existirão. Isto tem implicação direta no desempenho do decodificador. Então, a escolha de um pequeno número de formatos melhora o desempenho da máquina.

Para finalizar, estudamos como é a anatomia de um arquivo executável no formato ELF. O conhecimento adquirido ao dividir os campos das instruções é diretamente aplicado para o reconhecimento do formato ELF. Não nos cabe, entretanto, discutir os detalhes do formato. A apresentação foi geral, mas com ela é possível extrair a maioria das informações no formato. Em particular uma seção muito especial precisa ser abordada: a seção de cadeia de caracteres (*strings*).

Cada seção também têm nome e eles podem indicar um tipo muito particular de seção. As seções mais conhecidas são: `.init` que contém um trecho de código que deve ser executado antes da chamada ao programa principal; `.text` que contém as instruções do programa propriamente dito; `.fini`, que é executada depois da finalização do código do programa em `.text`. `.init` e `.fini` executam instruções que contribuem para a criação e finalização do processo pelo Sistema Operacional. A seção `.data` contém os dados do programa. A seção `.rodata` (*read-ony data*) contém os dados que não serão passivos de escrita durante a execução do código. A seção `.strtab` contém *strings* usadas no programa e a seção `.symtab` contém informações sobre os símbolos utilizados.

Juntado os nossos conhecimentos, aprendemos a criar uma descrição de uma arquitetura em uma ADL, a ArchC. Fizemos um esboço de uma descrição do MIPS usando a linguagem. Usamos um modelo funcional que esconde os detalhes da organização do computador, mas explicita os detalhes da construção do conjunto de instruções. É um bom exercício completar a especificação.

3.6 – Prática com Simuladores

Usando o SPIM podemos observar como é a codificação binária de cada instrução que executamos, basta olhar a coluna que mostra o valor hexadecimal equivalente.

Um outro exercício com simuladores, envolve a prática com o ArchC. O leitor se sinta impelido a instalar o SystemC e o ArchC para poder fazer simulações de descrição de ISAs.

Por fim, o programa `objdump` pode ser útil na decodificação automática (*desassembler*) de arquivos do formato ELF. Mas para ir além seria interessante usar um programa que apresente o código hexadecimal de cada byte de um arquivo, como o `HexEdit`.

3.7 – Exercícios

3.1 – Pesquise na web e produza um resumo sobre a seção strtab.

3.2 – Pesquise na web e produza um resumo sobre como especificar *pseudo-instruções* do MIPS em ArchC.

3.3 – Para cada código criado nos exemplos do capítulo 2 encontre a codificação em linguagem de máquina do MIPS.

3.4 – Preencha a Tabela 3.6 como o exemplo.

3.5 – Dado o arquivo ELF a seguir, indique qual o programa que será carregado na memória para ser executado.

# byte	Conteúdo															
00000000	**7f**	**45**	**4c**	**46**	**01**	**02**	**01**	**00**	**00**	**00**	**00**	**00**	**00**	**00**	**00**	**00**
00000010	**00**	**02**	**00**	**08**	**00**	**00**	**00**	**01**	**00**	**40**	**00**	**00**	**00**	**00**	**00**	**34**
00000020	**00**	**00**	**00**	**a0**	**00**	**00**	**00**	**00**	**00**	**34**	**00**	**20**	**00**	**01**	**00**	**28**
00000030	**00**	**02**	**00**	**00**	**00**	**00**	**00**	**01**	**00**	**00**	**00**	**60**	**00**	**40**	**00**	**00**
00000040	**00**	**40**	**00**	**00**	**00**	**00**	**00**	**18**	**00**	**00**	**00**	**18**	**00**	**00**	**00**	**07**
00000050	**00**	**00**	**00**	**00**	**00**	**00**	**00**	**00**	**00**	**00**	**00**	**00**	**00**	**00**	**00**	**00**
00000060	**01**	**09**	**58**	**2a**	**11**	**60**	**00**	**03**	**01**	**09**	**50**	**20**	**08**	**10**	**00**	**05**
00000070	**01**	**09**	**50**	**22**	**00**	**00**	**00**	**00**	**00**	**00**	**00**	**00**	**00**	**00**	**00**	**00**
00000080	**01**	**09**	**58**	**2a**	**11**	**60**	**00**	**03**	**01**	**09**	**50**	**20**	**08**	**10**	**00**	**05**
00000090	**01**	**09**	**50**	**22**	**00**	**00**	**00**	**00**	**00**	**00**	**00**	**00**	**00**	**00**	**00**	**00**
000000a0	**00**	**00**	**00**	**00**	**00**	**00**	**00**	**00**	**00**	**00**	**00**	**00**	**00**	**00**	**00**	**00**
000000b0	**00**	**00**	**00**	**00**	**00**	**00**	**00**	**00**	**00**	**00**	**00**	**00**	**00**	**00**	**00**	**00**
000000c0	**00**	**00**	**00**	**00**	**00**	**00**	**00**	**00**	**00**	**00**	**00**	**01**	**00**	**00**	**00**	**01**
000000d0	**00**	**00**	**00**	**06**	**00**	**04**	**00**	**00**	**00**	**00**	**00**	**80**	**00**	**00**	**00**	**18**
000000e0	**00**	**00**	**00**	**00**	**00**	**00**	**00**	**00**	**00**	**00**	**00**	**04**	**00**	**00**	**00**	**00**

Categoria	Nome	Formato	Exemplo	Operação (ArchC)	Comentários
Aritmética	add	R	add rd, rs, rt	RB[rd] = RB[rs] + RB[rt];	*Overflow* gera exceção
	sub				*Overflow* gera exceção
	addi				*Overflow* gera exceção Valor do imediato na faixa entre –32.768 e +32.767
	addu				*Overflow* não gera exceção
	subu				*Overflow* não gera exceção
	addiu				*Overflow* não gera exceção Valor do imediato na faixa entre 0 e 65.535
	mul				*Overflow* não gera exceção HI, LO imprevisíveis após a operação
	mult				*Overflow* não gera exceção
	multu				*Overflow* não gera exceção
	div				*Overflow* não gera exceção
	divu				*Overflow* não gera exceção
lógicas	or				
	and				
	xor				
	nor				
	andi				Imediato em 16 bits
	ori				Imediato em 16 bits
	sll				Desloc. ≤ 32
	srl				Desloc. ≤ 32
	sra				Desloc. ≤ 32. Preserva sinal
Transferência de dados	mfhi				
	mflo				
	lw				
	sw				
	lui				Carrega constante na porção alta do registrador de destino. Zera a parte baixa.
Suporte a decisão	bne				
	beq				
	j				
	slt				Opera com números sinalizados
	sltu				Opera com números não sinalizados
Suporte a procedimentos	jal				
	jr				

Tabela 3.6: Instruções do MIPS

Capítulo 4

O Processador

4.1 – Introdução

Aprendemos até o presente a utilizar uma linguagem de montagem e como esta linguagem é convertida para linguagem de máquina, que é efetivamente inteligível ao processador. Com os conhecimentos adquiridos é possível descrever, no nível funcional, o comportamento de uma máquina que executa as instruções que nós projetamos. Este capítulo tem a intenção de mostrar como podemos projetar um hardware para tornar realidade um tal processador de instruções.

Também introduziremos a questão do desempenho, que será tratada de forma mais abrangente no capítulo 5. Aqui a abordagem será meramente a construção de um *pipeline* que permita que as instruções sejam executadas por partes, promovendo uma melhor utilização dos recursos e diminuindo o tempo necessário para execução de um programa.

Finalmente veremos, na visão do software, como podemos especificar uma organização de um computador de forma mais sofisticada utilizando ArchC.

4.2 – Componentes da via de dados

Para construirmos uma via de dados capaz de executar as instruções que nós projetamos é preciso conhecer primeiro os componentes fundamentais de uma organização de computador. Nós já nos deparamos com o modelo de memória como um grande repositório de informações endereçáveis. Como descreveremos em detalhes o sistema de memórias no capítulo 6, vamos apenas considerar que uma memória é uma caixa que possui duas informações

de entrada: se haverá uma leitura ou escrita; e qual o endereço será lido/escrito.

A Figura 4.1 mostra o modelo utilizado. Um sinal de controle W/R indica se haverá uma leitura (0_2) ou escrita (1_2). O endereço é posto pelo processador nos 32 bits que compõem o endereço. No caso de uma leitura de informação, o dado requerido é disponibilizado nos 32 bits que compõem a entrada/saída da memória. No caso de uma escrita de informações, o processador disponibiliza o dado e o endereço onde ele deve ser escrito e envia um sinal de escrita (W/R=1_2).

Mencionamos no capítulo 1 deste livro que na memória existem duas regiões distintas: a de código e a de dados. Esta distinção é meramente convencionada, mas existem técnicas que permitem a utilização simultânea das duas regiões. Para o software é como se existissem duas memórias separadas. Vamos seguir este modelo no desenvolvimento do restante de nossa construção da via de dados.

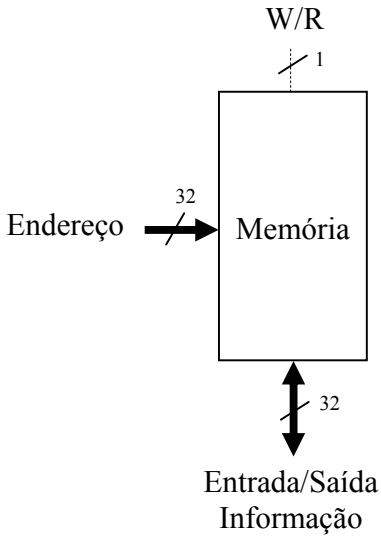


Figura 4.1: Representação de uma memória

O segundo componente da via de dados é o banco de registradores. Nós já sabemos que uma instrução, por exemplo, aritmética lê dois registradores e escreve o resultado em terceiro registrador. Nosso banco de registradores será capaz de ler dois registradores ao mesmo tempo e escrever em um terceiro (que pode inclusive ser um dos que acabou de ser lido).

A Figura 4.2 mostra o projeto de um banco de registradores. O sinal de controle W/R é análogo ao presente na memória. Ele vai controlar leitura e escrita no banco. As duas especificações de registradores a serem lidos são

feitas utilizando-se de 5 bits. Ora, com esta quantidade de bits é possível endereçar qualquer registrador em um banco de 32 registradores. Os dados que estão armazenados nos respectivos registradores serão disponibilizados em duas vias de 32 bits cada.

No caso de uma escrita, a informação do dado a ser escrito é disponibilizado e o sinal W/R é setado, indicando uma escrita.

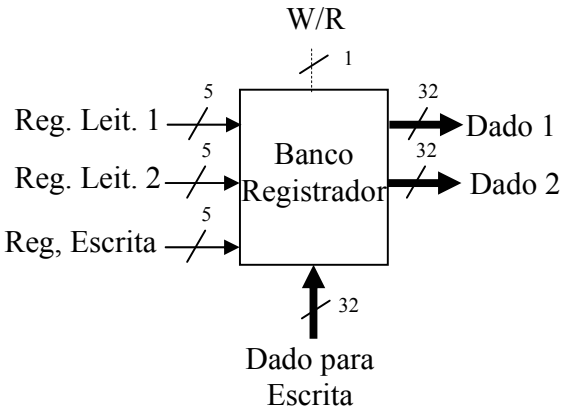


Figura 4.2: Representação de um banco de registradores

O terceiro e último componente da via de dados é a unidade lógica e aritmética. Este é, certamente, o componente onde as operações ocorrem. Ele é controlado por uma série de sinais que indicam qual operação deve ser realizada em um determinado instante. Soma, subtração, *and*, *or*, *slt* etc. Todas as operações são realizadas com operandos de 32 bits. Apenas dois operandos são utilizados por vez. O resultado da operação também é dado em 32 bits. Existe ainda um bit de saída, informando se o resultado da operação realizada é zero. Este sinal é chamado de `resultadozero`.

A Figura 4.3 mostra uma ALU típica para um conjunto de instruções bem restrito (mais ainda do que aquele que apresentamos nos capítulos anteriores). Existem 3 sinais de controle (`op`) para informar qual a operação a ser realizada. Dado 1 e Dado 2 serão os operandos e o resultado ficará em `resultado`. `ResultadoZero` valerá 1_2 se o resultado da operação, seja ela qual for, valer 0_2.

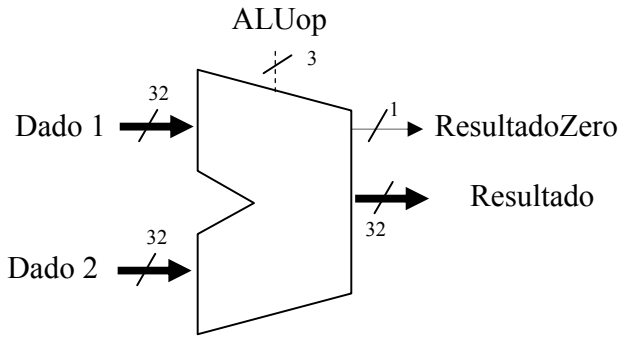


Figura 4.3: Representação de uma ALU

Existem ainda dois componentes muito importantes, mas que servem como interligação entre estas partes principais. O extensor de sinal é um destes componentes. Como já dissemos, a ALU opera sempre com dados de 32 bits, mas as instruções do tipo I portam em sua codificação, dados de 16 bits. É preciso convertê-los para 32 bits para que possam ser operados pela ALU. Agora, a título de recordação, vamos ver como um número negativo é representado em diversas quantidades de bits. Por exemplo, o número -2 em 16 bits vale 1111111111111110_2. Já em 32 bits este valor é representado como, $11111111111111111111111111111110_2$. Transformar um dado de 16 bits em 32 bits é simplesmente replicar o seu bit de sinal para as 16 casas binárias que restam nos 32 bits. Isto também funciona perfeitamente com números positivos. A Figura 4.4 mostra um extensor de sinal de 16 bits para 32 bits.

Finalmente temos o multiplexador, MUX. Este circuito funciona como um seletor de canais, controlado eletronicamente. Um MUX com 4 canais implica em um seletor de 2 bits. Um MUX com 8 canais, em um seletor de 3 bits. A Figura 4.5 mostra a operação de um MUX com 4 canais. Observe que os canais não necessariamente possuem apenas um bit. A propósito, na figura são mostrados canais de 32 bits.

Figura 4.4: Extensor de sinal (16 bits para 32 bits)

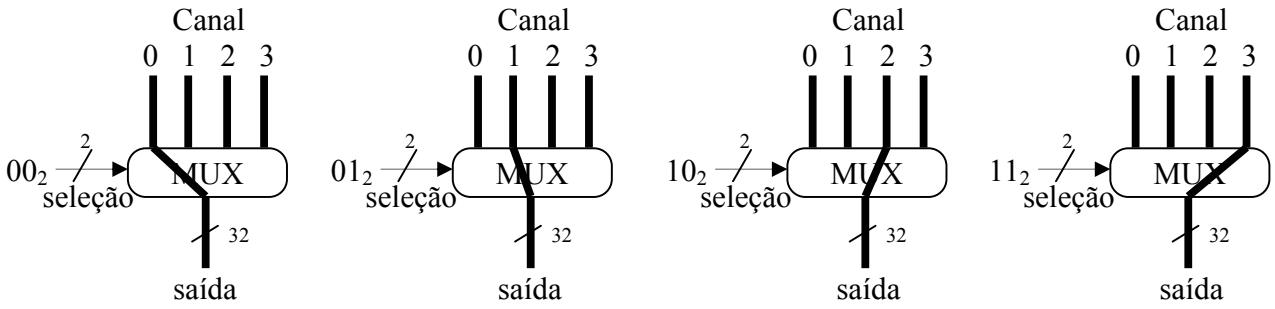


Figura 4.5: Operação de um MUX

4.3 – Interligação dos componentes da via de dados

Agora que já conhecemos os componentes que compõem a via de dados e os seus funcionamentos, vamos interligá-los de tal forma que possamos realizar as instruções do MIPS. Por simplicidade, vamos trabalhar com um pequeno subconjunto de instruções.

Vamos começar pela implementação da via de dados para realizar a instrução `add $8, $9, $10`. Bem, é preciso fazer duas leituras no banco de registradores, realizar a soma na ALU e escrever o resultado de volta no banco de registradores. A Figura 4.6 mostra a interligação dos componentes para formar a via de dados. A instrução a ser executada está em um registrador especial chamado IR. A partir dele os campos dos registradores são extraídos.

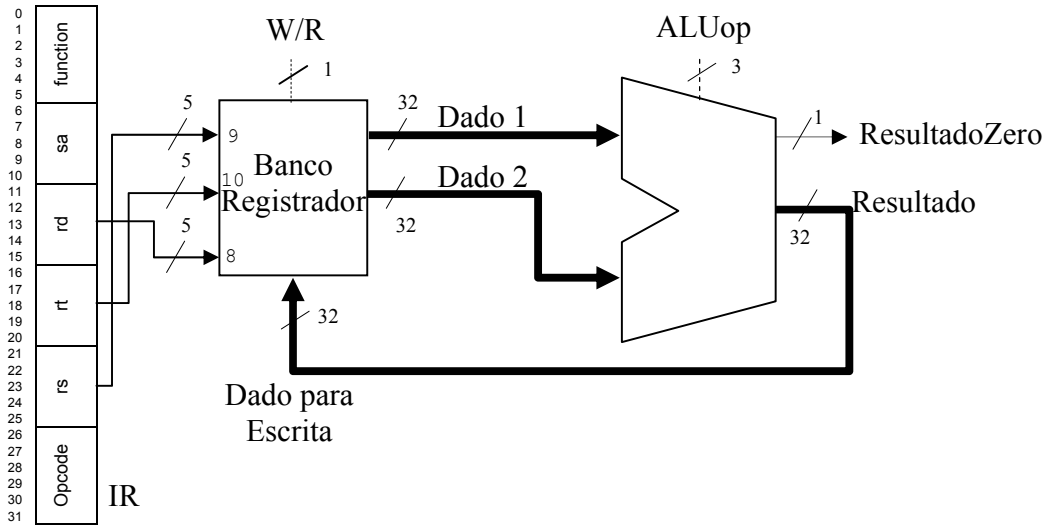


Figura 4.6: Via de dados para instrução `add`

Os registradores especificados em `rs` e `rt` são lidos do banco de registradores. O dados vão direto para as entradas da ALU, onde a operação de soma é selecionada e o resultado disponibilizado para escrita de volta no banco de registradores. Esta escrita será realizada no registrador especificado no campo `rd` da instrução. Os sinais de controle serão estudados oportunamente.

Agora vamos ver como seria a via de dados para execução da instrução `lw $8, 16($4)`. Nesta instrução, são utilizadas: a memória, para leitura do dado; o banco de registradores para ler $4 e escrever o dado retirado da memória em $8; e o extensor de sinais para converter o campo imediato da instrução em um valor de 32 bits. Este valor será somado na ALU com o conteúdo do registrador $4 para formar o endereço do dado a ser lido. A Figura 4.7 mostra esta nova via de dados.

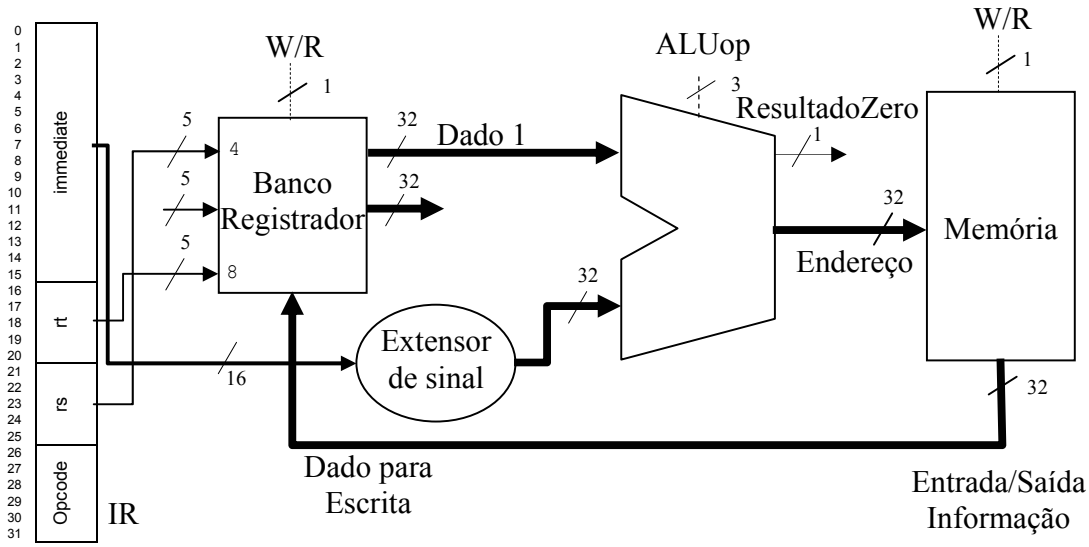


Figura 4.7: Via de dados para instrução `lw`

Agora vamos ver uma parte da via de dados que é comum a todas as instruções: o cálculo de PC e a busca de instruções na memória. Em verdade, o PC é sempre somado a 4, salvo em caso de instruções de salto. Vamos considerar inicialmente que não existem saltos e verificar como ficaria o circuito para calcular o PC. Ora, já sabemos que a ALU é capaz de realizar somas, subtrações, operações lógicas etc. Uma ALU mais simples, que realiza apenas somas com o valor 4, é então utilizada para cálculo do PC. O valor do PC é incrementado e a instrução seguinte é lida na memória. A Figura 4.8 mostra esta parte do circuito.

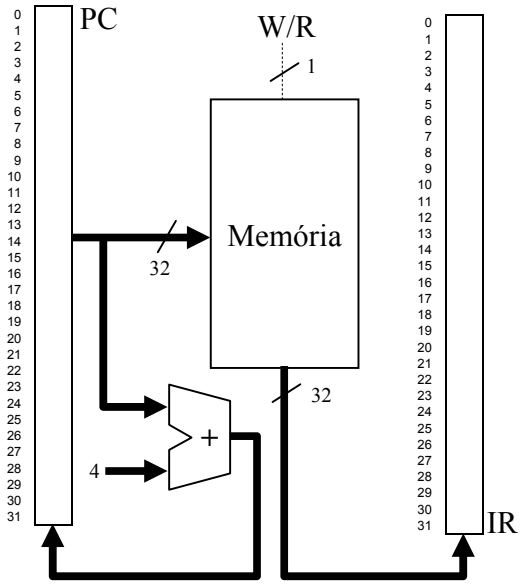


Figura 4.8: Cálculo do PC e busca de instruções na via de dados

Finalmente vamos verificar como ficaria a via de dados para podermos realizar uma instrução `beq $8, $9, alvo`. Sabemos que esta instrução compara $8 com $9 e se eles forem iguais salta para o endereço representado por alvo. O cálculo do endereço envolve somar o valor de PC com o valor do campo imediato da instrução (multiplicado por 4) para encontrarmos o novo valor de PC. Entretanto, este valor calculado de PC só será gravado no mesmo se a condição para o salto acontecer.

Então uma parte da via de dados precisa se certificar que $8 e $9 são iguais. Bem, para saber se dois números são iguais basta subtrair o primeiro do segundo. Se o resultado for 0, significa que os dois números são idênticos, caso contrário eles diferem. Uma boa opção seria utilizarmos a ALU principal para fazer esta operação e mais uma vez utilizarmos uma outra ALU menor, que realiza apenas somas, para cálculo do alvo do desvio.

A Figura 4.9 mostra como ficaria a via de dados para dar suporte a esta instrução. Veja que o valor de PC pode ser tanto PC+4 como o endereço alvo calculado usando o campo `immediate` da instrução e o próprio PC. Para selecionar uma das duas possibilidades usamos um MUX. O sinal de seleção do MUX vem exatamente da ALU principal. Se a comparação (subtração) resultar em um valor 1, ou seja, se os operandos forem idênticos, então é selecionado como novo valor de PC o endereço do alvo. Caso contrário, o PC é simplesmente incrementado de 4. Ainda, o cálculo do endereço envolve uma multiplicação por 4 que é implementada com um deslocador de 2 posições.

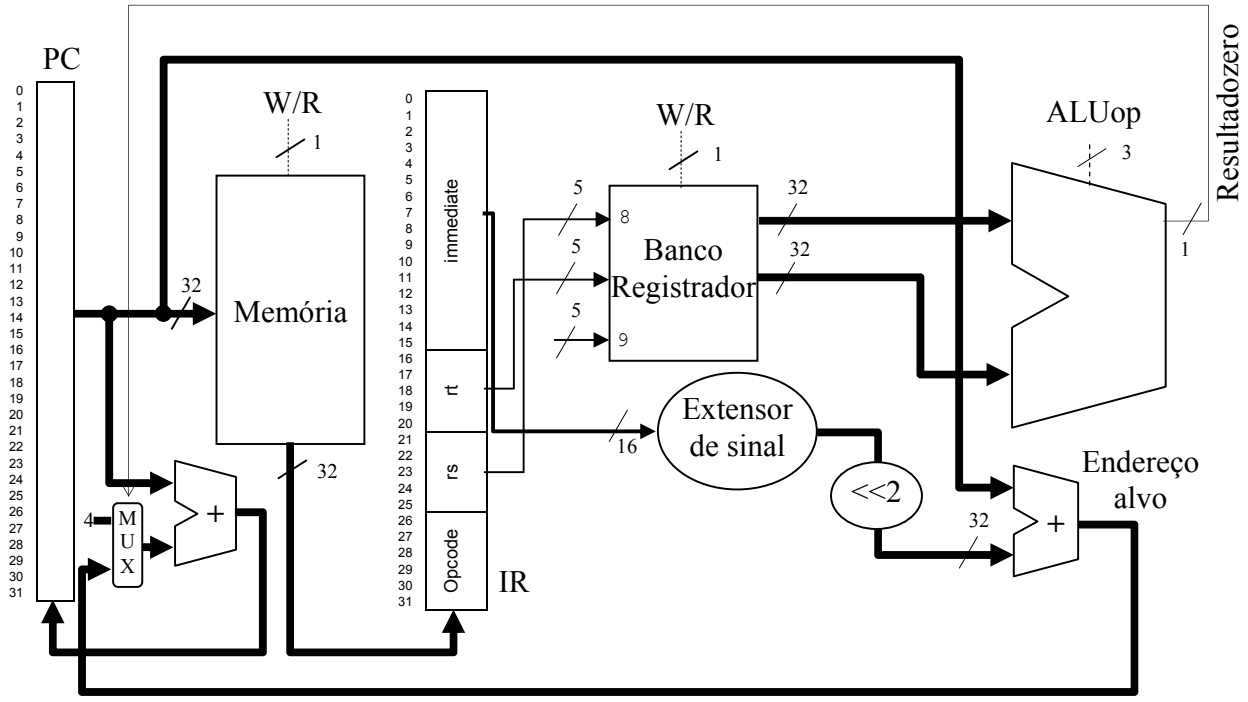


Figura 4.9: Via de dados para instrução `beq`

Até o presente projetamos vias de dados particulares para cada instrução. Observe, entretanto, que algumas ligações entre as partes são comuns a mais de uma instrução. Para evitar desperdiçar recursos de hardware vamos tentar reaproveitar os componentes e suas interligações. Vamos por exemplo projetar uma via de dados que execute tanto a instrução `add` como a instrução `lw`.

Observando as figuras 4.6 e 4.7 vamos perceber que a primeira diferença entre as duas refere-se à segunda entrada da ALU (doravante usaremos apenas ALU para designar ALU principal). Ela pode vir do Extensor de sinal ou do banco de registradores. Outra escolha que precisa ser feita é quanto ao dado a ser escrito no banco de registradores. Ou ele vem da memória, ou ele vem da ALU. Finalmente a terceira diferença refere-se a qual campo da instrução especifica o registrador onde, o resultado da soma ou a carga do dado, será escrito. Para fazer estas escolhas podemos utilizar MUXs controlados por sinais de seleção vindos da unidade de controle. A Figura 4.10 mostra como é feita a fusão das duas vias de dados em apenas uma.

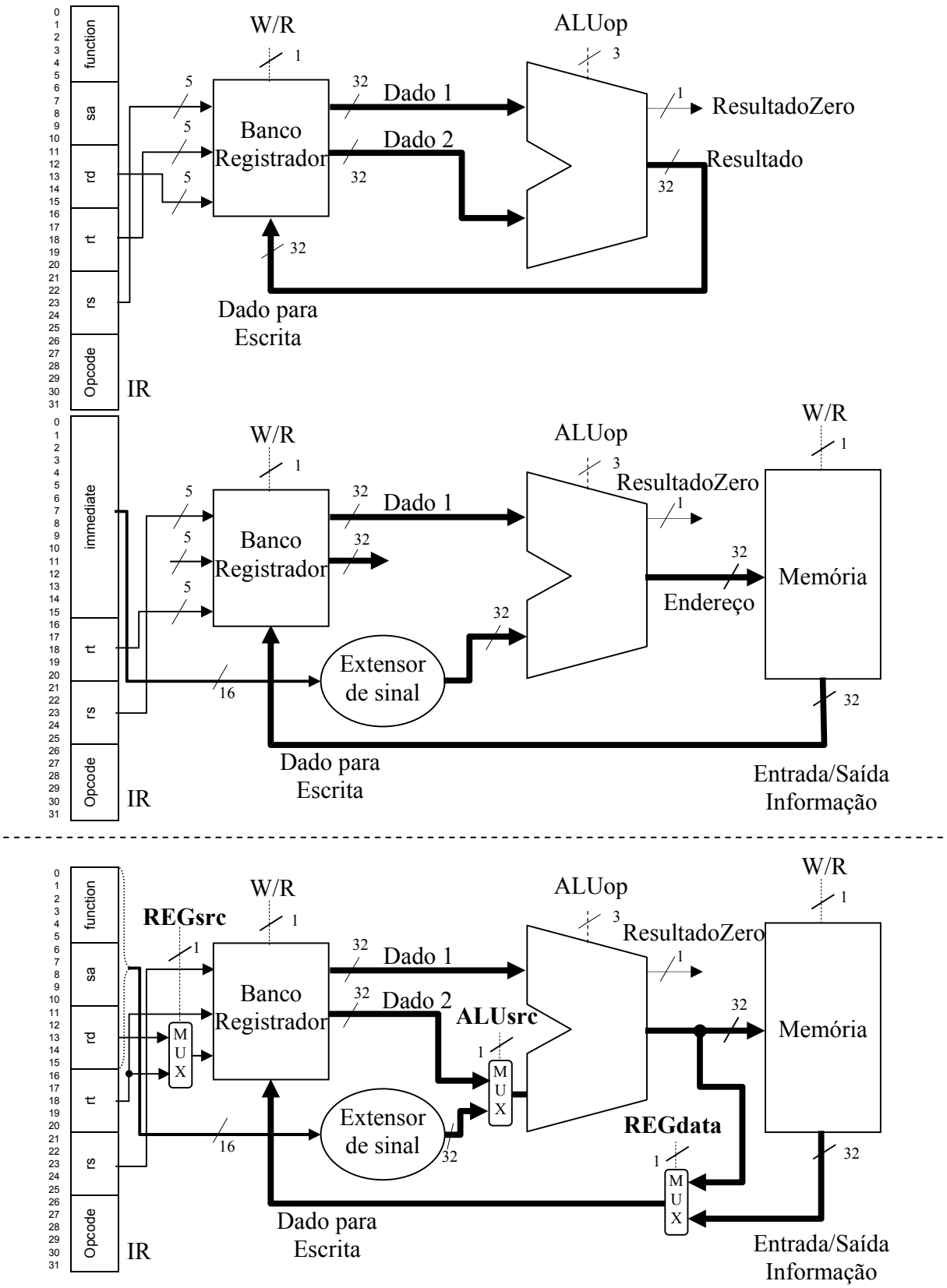


Figura 4.10: Via de dados para instruções `add` e `lw`

Agora, quando a instrução a ser executada for uma soma os sinais de controle devem refletir esta escolha, ou seja, `REGdst` = 0_2, `REGdata` = 0_2 e `ALUsrc` = 0_2. Ao contrário, quando a instrução for uma lw, os sinais de controle devem ser: `REGdst` = 1_2, `REGdata` = 1_2 e `ALUsrc` = 1_2.

A fusão desta nova via de dados com aquela projetada para instrução `beq` também é muito simples. Nenhum novo MUX é acrescentado, somente o cálculo de PC e a carga da instrução. A Figura 4.11 ilustra a nova via de dados. Perceba que a memória foi dividida em memória de instruções e memória de dados. Esta divisão lógica é implementada na prática através do uso de duas caches. Estudaremos a hierarquia de memórias mais tarde.

Um outro componente acrescentado na via de dados foi a porta AND. Ora, imagine que esta via de dados está realizando uma soma de dois valores iguais a zero. O resultado seria zero e a ALU setaria a saída `ResultadoZero`. Isto faria com que o PC fosse carregado com um valor qualquer. Esta saída da ALU só tem significado quando a instrução a ser executada for uma `beq`, por isto a porta AND e o sinal de controle `BEQ?` foram acrescentados.

Agora que já conhecemos a metodologia para acrescentar funcionalidades à via de dados, resta-nos pensar em como implementar as demais instruções que estudamos. O caminho é sempre montar uma via de dados para a instrução em particular e depois anexá-la com as demais.

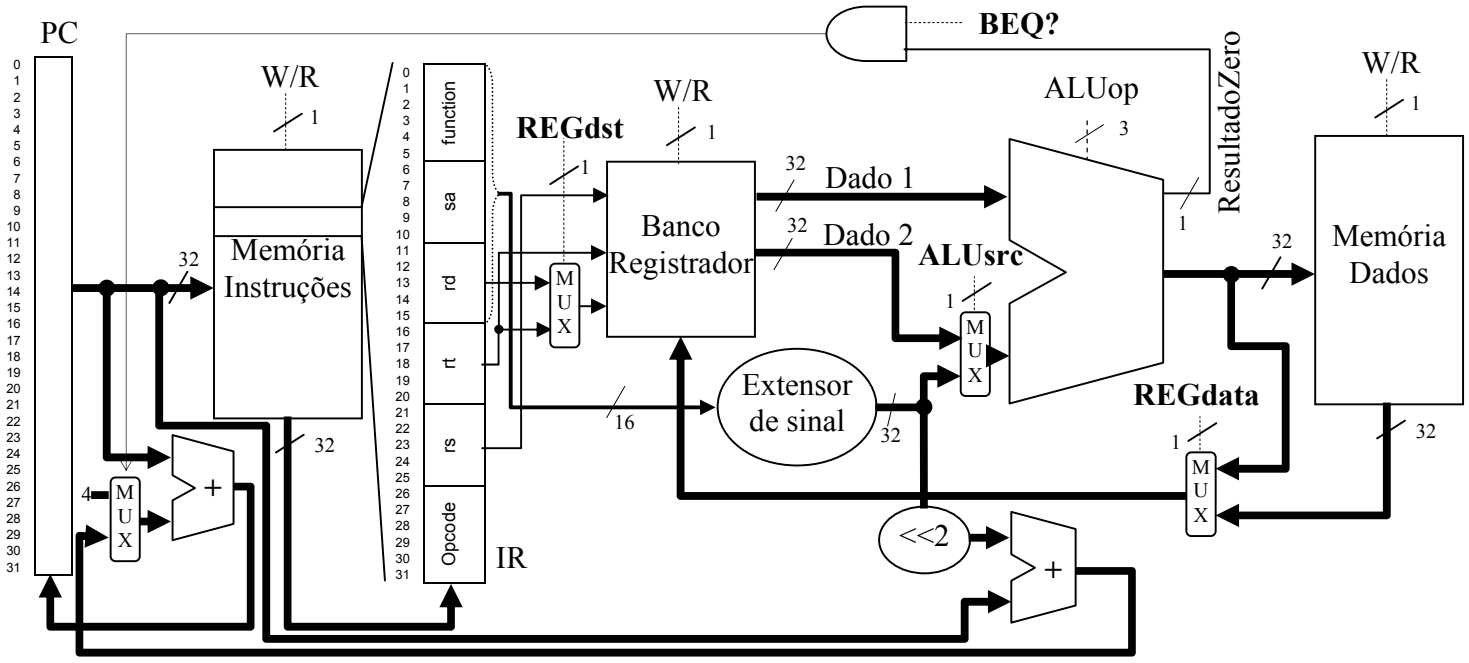


Figura 4.11: Via de dados para instruções `add`, `lw` e `beq`

4.4 – Unidade de controle

A Unidade de Controle é responsável por gerar os sinais de decisão para a via de dados, a fim de promover a correta execução das instruções. Além dos sinais de controle dos MUXs, ainda existem outros sinais de controle nas unidades que compõem a via de dados. O sinal de escrita nas memórias e no banco de registradores e o sinal `BEQ?`.

Para decidir quais sinais de controle devem ser setados e quais devem ser ressetados é preciso conhecer a instrução que se pretende executar. Para isto a unidade de controle lê o campo `opcode` e o campo `function` e a partir deles gera os sinais adequados.

Vamos exemplificar como seria construída a unidade de controle para as três instruções utilizadas acima na construção da via de dados. Por razões de simplicidade vamos considerar que as memórias e o banco de registradores estão sempre lendo e no momento em que precisam escrever um dado é que o sinal de escrita deve ser setado. Vamos também considerar que para a ALU realizar uma soma, os dados a serem inseridos em ALUop valem 010_2 e para que seja realizada uma subtração ALUop deve conter 110_2.

A forma mais simples de construir esta unidade de controle é gerar uma tabela verdade com todas as entradas e saídas. A unidade de controle seria modelada como um circuito combinacional simples. Na prática, as unidades de controle são **microprogramadas** ou **hardwired**. Para nossa abordagem poderíamos esquecer os detalhes de implementação e devotar o nosso tempo à escolha dos sinais de controle adequados para cada instrução.

A Tabela 4.1 mostra a tabela verdade implementada na unidade de controle para as nossas instruções. Os sinais de controle são então setados de acordo com a necessidade de cada instrução. `W/R REG`, por exemplo, vale 1_2 quando a instrução precisar escrever no banco de registradores. `W/R MD` vale 1_2 se for necessário escrever na memória de dados (caso da instrução `sw`, não exemplificada até o presente). Observe que o sinal `W/R MI` deve ser sempre 0_2, pois não há sentido, para um usuário, escrever na memória de instruções.

	Entradas		**Saídas**							
instrução	opcode	function	W/R MI	REGdst	W/R REG	ALUsrc	BEQ?	REGdata	ALUop	W/R MD
add	000000	100000	0	0	1	0	0	0	010	0
lw	100011	XXXXXX	0	1	1	1	0	1	010	0
beq	000100	XXXXXX	0	0	0	0	1	0	110	0

Tabela 4.1: tabela verdade da unidade de controle

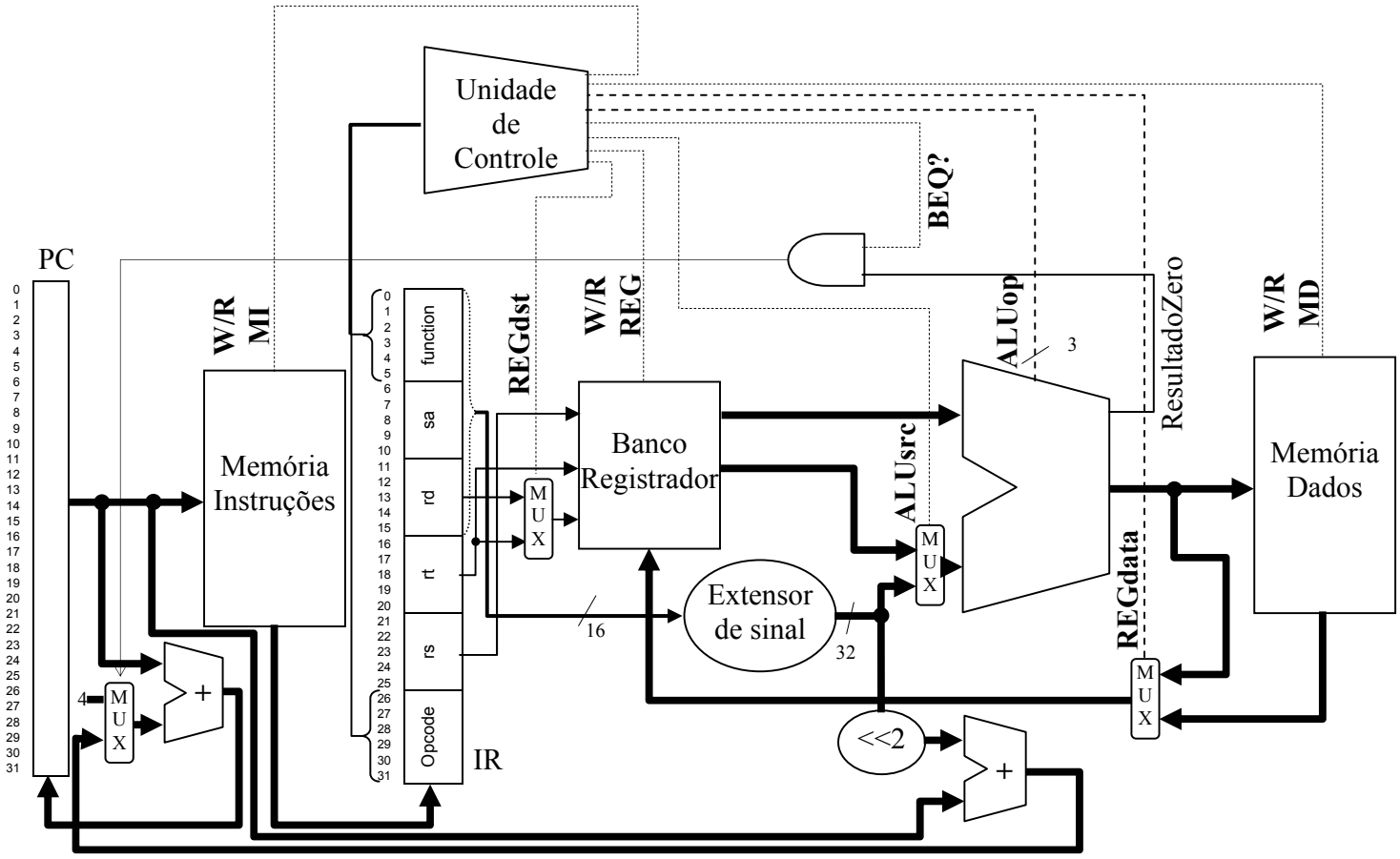


Figura 4.12: Via de dados com Unidade de Controle para instruções `add`, `lw` e `beq`

A Figura 4.12 mostra como a unidade de controle se acopla ao hardware da via de dados. À medida que novas instruções vão sendo implementadas novos MUXs vão surgindo e a complexidade aumentando. Isto significa que a unidade de controle, que realiza a decodificação (interpretação da instrução), vai ficando mais e mais difícil de ser implementada com lógica combinacional.

Esta nossa implementação, embora funcional, carece de uma importante variável: o tempo. Neste modelo todas as instruções gastam o mesmo tempo para serem executadas. Esta implementação é chamada **monociclo**. O ciclo a que se refere o nome é o **ciclo de clock**. O **clock** é uma onda quadrada que atua como uma espécie de maestro das tarefas de um processador. Ele dita em quanto tempo uma tarefa é executada. A freqüência do clock é o inverso do ciclo de clock. Isto significa que quanto mais alta a freqüência, mais instruções o processador será capaz de realizar em um determinado intervalo de tempo.

A busca da tecnologia por freqüências mais altas esbarra em um efeito colateral muito indesejado: a dissipação de potência em forma de calor. Hoje um processador opera com um ventilador (*cooler*) acoplado a ele para permitir que ele opere em freqüências mais altas.

4.5 – Via de dados mono e multi-ciclos

À parte destas implicações vamos ver como a nossa via de dados sofre a influência desta nova variável. Nossa implementação não se importou com os atrasos em cada componente do circuito, mas de fato, eles existem. Por exemplo, a leitura nas memórias pode custar 6ns, enquanto o acesso ao banco de registradores custa 2ns. A ALU realiza uma operação em 4 ns. O tempo de leitura e escrita nos registradores PC e IR são desprezíveis e também o tempo para as somas nas ALUs auxiliares.

Para estes números, por exemplo, o clico de clock seria: 6ns (acesso a MI) + 2ns (Leitura REG) + 4ns (ALU) + 6ns (acesso a MD) + 2ns (escrita no REG) = 20ns. Ou seja, mesmo que uma instrução não utilize todos os componentes da via de dados, é preciso garantir que o ciclo de clock acomode a execução de qualquer instrução. Nesta situação, o pior caso é quem dita a regra. A freqüência de operação desta máquina não poderia ser maior que 50 MHz, um valor longe de nossa realidade atual (de cerca de 3GHz).

Uma possibilidade para melhorar estas perdas seria utilizar uma abordagem **multiciclos**. Neste caso o ciclo de clock seria a menor unidade de tempo necessária para realizar uma tarefa em qualquer dos componentes da via de dados, ou seja, poderíamos, no exemplo acima, especificar o ciclo de clock como sendo 2ns. Seriam necessários então 3 ciclos de clock para acesso à memória e 2 ciclos de clock para operação da ALU. As operações de leitura e escrita no banco de registradores consumiria apenas 1 ciclo de clock cada. Agora, instruções que não utilizam todos os componentes da via de dados poderiam terminar sua execução mais rapidamente (em menos ciclos).

Infelizmente existem alguns requisitos para que esta idéia possa se tornar realidade. Precisamos dividir as tarefas do processador para executar uma instrução em etapas. A primeira seria buscar a instrução na memória (***Instruction Fetch***). A segunda seria decodificar esta instrução e ler os registradores adequados (***Instruction Decode***). Depois seria feito o envio dos dados para a ALU (*instruction issue*) e sua seqüente execução (***Instruction Execution***). Em alguns casos seria necessário ler ou escrever um dado na memória (***Memory Access***) e finalmente seria necessário escrever o dado lido no banco de registradores (***Write-Back***). Estas cinco etapas precisariam ser blindadas em hardware para que o clock pudesse comandar as atividades de

cada uma delas. Esta blindagem é feita utilizando-se de registradores entre estágios de execução. Deste ponto em diante já não vamos mais tecer detalhes sobre a implementação, bastando ao leitor o entendimento dos conceitos e das implicações em adotá-los. Naturalmente sintam-se encorajados a saber mais em uma literatura mais volvida para organização de computadores.

Vamos analisar as vantagens, em termos de tempo, de uma implementação multi-ciclo sobre uma mono-ciclo. Considerando uma seqüência de instruções, `add`, `lw` e `beq` e sabendo que na abordagem mono-ciclo cada instrução é executada em 20 ns, precisaríamos de 60ns para que esta seqüência fosse terminada. Numa abordagem multi-ciclo a instrução `add` utiliza 7 ciclos de 2ns cada, ou seja, ela consome 14ns. `lw` precisa passar por todas as etapas e portanto utiliza os mesmos 20ns como no original. `beq` utiliza 6 ciclos de 2ns, ou seja 12ns. A execução das 3 em um sistema multi-ciclos demanda apenas 46 ns.

A Figura 4.13 mostra a execução das instruções nas duas abordagens. Veja que a execução da instrução `add` passa pela memória de instruções, banco de registradores, ALU e banco de registradores. `lw` utiliza memória, banco de registradores, ALU, memória e banco de registradores. `beq` utiliza memória, banco de registradores e ALU.

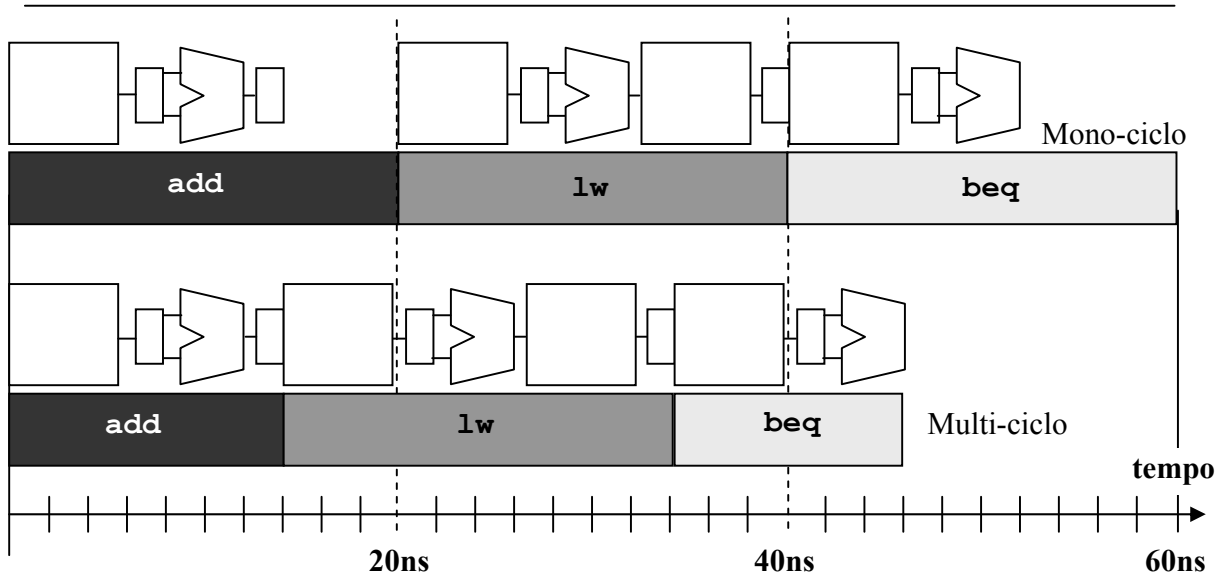


Figura 4.13: Execução de instruções em implementações mono e multi-ciclos

Poderia existir então algum método mais rápido para executar instruções que a implementação multi-ciclos? A resposta é sim, fazendo

reutilização dos componentes presentes em cada parte da execução da instrução.

4.6 – Pipeline

A idéia por trás do pipeline é uma linha de montagem de automóveis. Enquanto o uma equipe está trabalhando na pintura, outra está trabalhando no sistema elétrico e outra no motor. Quando as três equipes terminam sua tarefa, o primeiro carro está totalmente montado, o segundo, que estava na parte elétrica, vai para a parte de pintura, e o terceiro, que estava no motor vai para parte elétrica. Assim, podemos manter ocupadas todas as partes de um processador sem precisar esperar que a instrução fique pronta.

A divisão do trabalho é feita de forma semelhante ao mostrado anteriormente: um estágio de busca (F, *fetch*), um de decodificação (D, *decode*), um de execução (E, *execution*,), um de memória (M, *memory*) e o ultimo de escrita (W, *write*). Nesta abordagem, a equipe (o estágio do pipeline) mais lenta é quem dita a regra.

Seguindo a mesma temporização dos exemplos anteriores poderíamos definir um ciclo de clock de 6ns, pois a leitura da memória é a parte mais demorada do processo e gasta exatamente 6ns. Durante o primeiro ciclo de clock a instrução `add` é buscada na memória. No segundo ciclo de clock, a instrução `add` vai para o estágio de decodificação e a instrução `lw` é buscada na memória. No terceiro ciclo de clock, `add` vai para execução, `lw` vai para decodificação e `beq` é buscada na memória. E assim, sucessivamente. A Figura 4.14 mostra a execução das instruções.

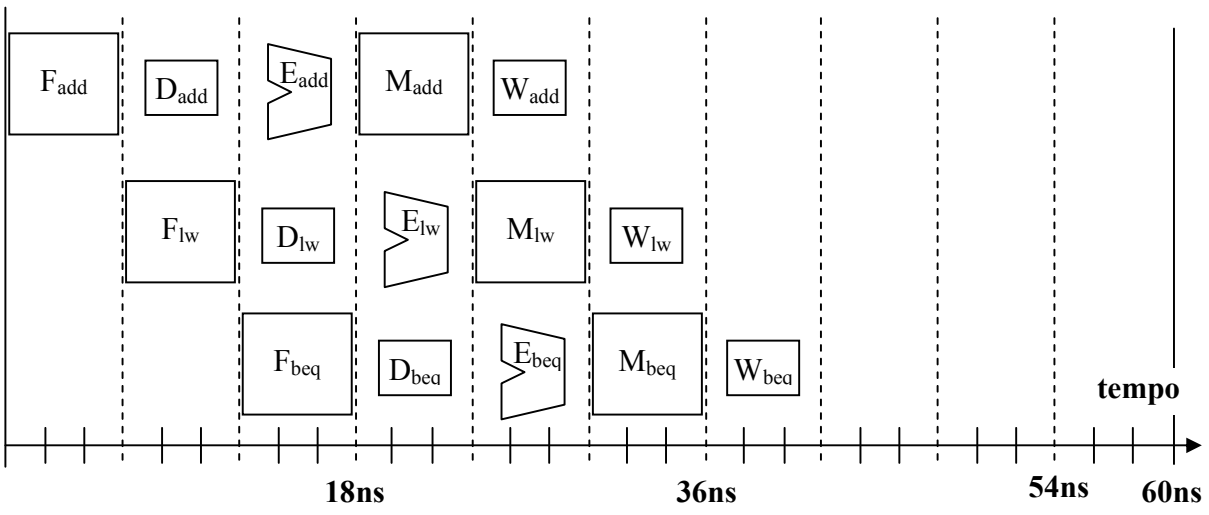


Figura 4.14: Execução de instruções em pipeline

Agora a execução foi concluída com 42ns. Melhor que a implementação multi-ciclo. Esta vantagem se amplia muito se considerarmos que outras instruções vão entrando no pipeline e a tendência é que tenhamos uma instrução terminada a cada 6ns, uma taxa próxima de 1 instrução por ciclo de clock (**IPC, *Intruction Per Cicle***).

Apesar destes avanços, vamos perceber na visão do software que o pipeline sofre um mal que exige muita atenção: a dependência de dados..

4.7 – A visão do software – O pipeline

O pipeline parece ser uma grande vantagem em termos de desempenho para os processadores. Entretanto existem algumas dificuldades que tornam a aplicação do pipeline um pouco menos eficiente que o limite teórico. Vamos ver um exemplo bastante simples. Depois de executar algumas instruções o pipeline mostrado na Figura 4.15 pretende ao mesmo tempo escrever no banco de registradores para instrução `add` *e ler o banco de registradores para instrução* `sub`*.*

Ora, isto gera um ***conflito de recursos****. O meu banco de registradores é capaz de ler duas instruções por vez, mas não está projetado para ler e escrever ao mesmo tempo. Um conflito de recursos é definido como sendo a tentativa de utilização de um mesmo componente por mais de uma instrução ao mesmo tempo.*

Neste caso específico, como a leitura no banco de registradores ocorre em 2ns e o ciclo de clock é de 6ns, conforme o exemplo que nós estamos trabalhando desde o início de nossa abordagem, é possível fazer uma escrita e uma leitura dos dados dentro do mesmo período do clock. Então, é utilizada a primeira metade do ciclo de clock para escrita no banco de registradores e a segunda metade para fazer as leituras. Poderia ser ao contrário, mas assim é mais eficiente.

Um segundo problema a ser enfrentado neste pipeline é a dependência de dados. Observe que o simples trecho de código abaixo gera um enorme problema para o pipeline.

```
add $8, $9, $10
sub $11, $12, $8
```

Aqui a instrução `sub` *utiliza o resultado da instrução* `add` *como um de seus operandos. Ora, pensando em pipeline, o resultado de* `add` *só vai estar pronto no quinto ciclo de clock, mas* `sub` *precisa dele já no terceiro ciclo de*

clock. Este tipo de conflito nós chamamos ***de conflito de dados****. Ele está ilustrado na Figura 4.16.*

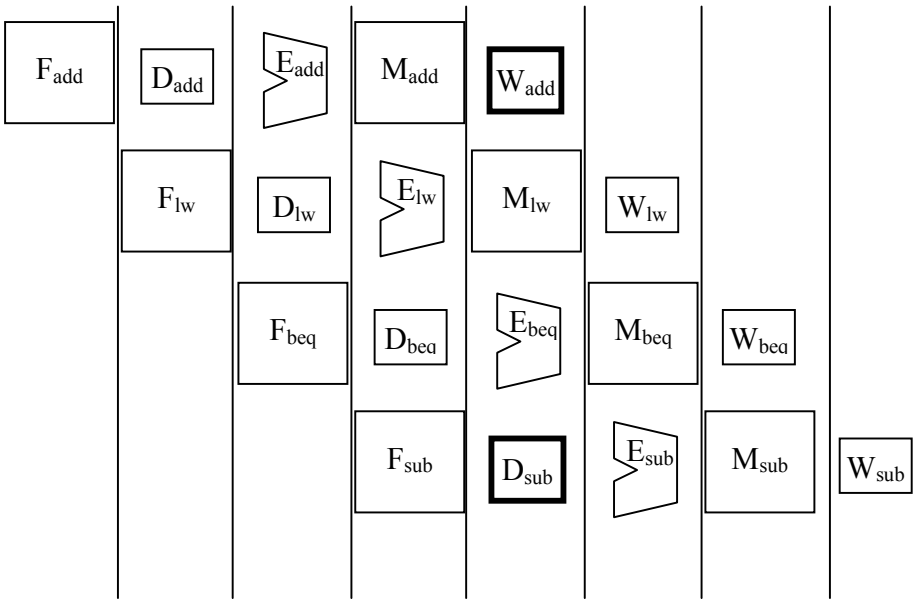


Figura 4.15: Dependência Estrutural no pipeline

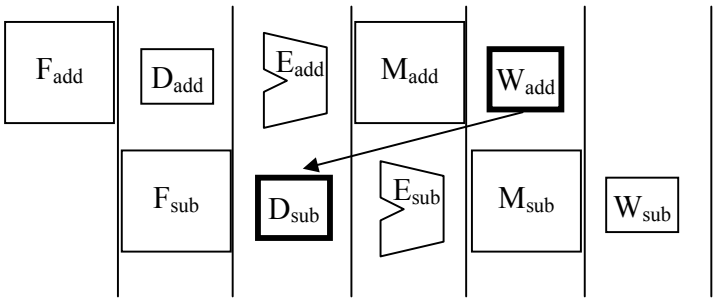


Figura 4.16: Dependência de dados no pipeline

Por fim, existe ainda um tipo de conflito muito comum: ***conflito de controle****. Imagine a execução de uma instrução de salto. Como o processador só vai saber se a condição de salto foi atingida ou não no estágio de execução, duas instruções que seguem no pipeline nos primeiros estágios podem correr o risco de serem descartadas. Se o salto for tomado, certamente haverá penalidades por isto. A Figura 4.17 mostra um* `beq` *que se fosse tomado deveria executar a próxima instrução* `sw` *e se não o fosse seguiria com* `lw`*. Como o teste só foi realizado no final do estágio de*

execução, as duas instruções que vinham após foram descartadas e a instrução correta passou a ser buscada.

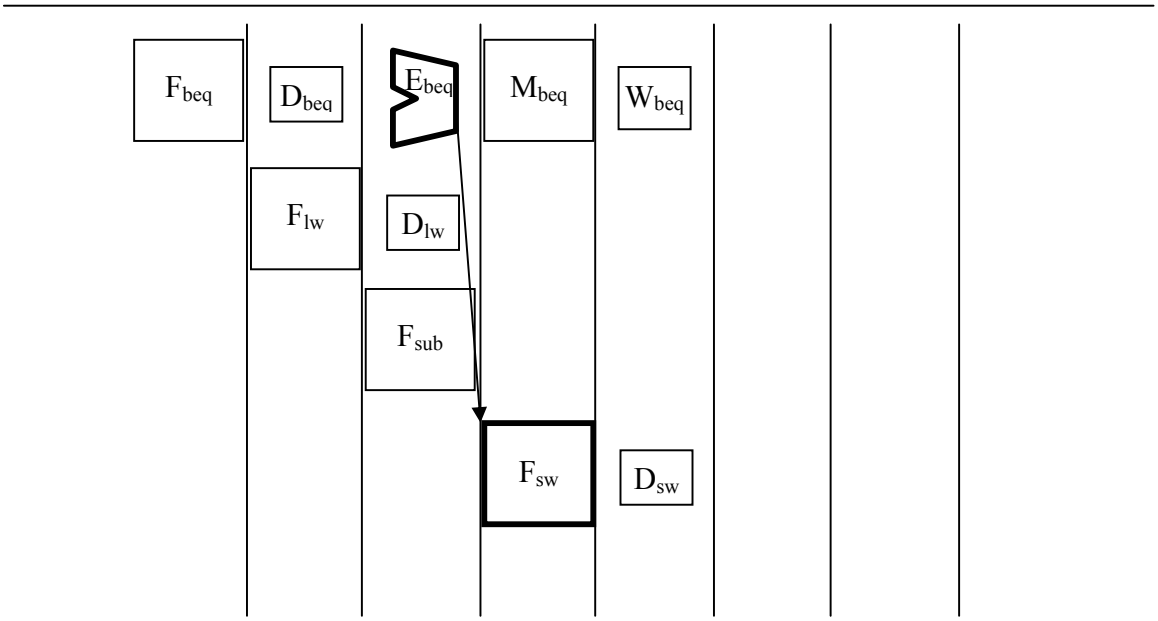


Figura 4.16: Dependência de controle no pipeline

*Apesar destas dificuldades existem técnicas que buscam resolver o problema e praticamente o eliminam. As técnicas de **forwarding** e de **delay slot** são muito eficazes para solucionar conflitos. Uma outra técnica para melhorar o desempenho é usar um circuito preditor de desvio. Este circuito memoriza se um salto foi tomado ou não e caso ele tenha sido tomado freqüentemente, a instrução que será retirada da memória após a instrução do salto será que se encontra em seu endereço alvo e não a sua subseqüente.*

Conhecer os conflitos gerados nos pipelines podem nos ajudar a programar melhor em HLL. A primeira dica é tentar evitar o uso de instruções que dependam imediatamente do resultado da anterior. A segunda, é construir estruturas de controle como laços de tal forma que eles sejam o máximo possível uniforme, ou seja, que sempre tome um caminho, variando muito pouco o seu comportamento. Construir estruturas de decisão que ficam alternando os caminhos de processamento das instruções é perder desempenho.

4.8 – Conclusões

Estudamos neste capítulo a construção de um processador simples capaz de executar as instruções em código de máquina vistas no capítulo anterior. Vimos também que existem algumas maneiras de interligar os componentes de um processador de tal forma que obtenhamos o melhor desempenho possível. A propósito, as métricas de desempenho serão abordadas em nosso próximo capítulo.

4.9 – Prática com Simuladores

O EWB é uma ferramenta interessante para construção de circuitos. A prática com este simulador leva o leitor a conhecer bem o material do curso. Nós propomos aqui dois projetos. A construção de uma ALU com portas lógicas e igualmente a construção de um banco de registradores. Depois tente interconectá-los.

4.10 – Exercícios

4.1 – Pesquise na web e produza um resumo sobre unidade de controle microprogramada.
4.2 – Pesquise na web e produza um resumo sobre unidade de controle hardwired.
4.3 – Pesquise na web e produza um resumo sobre *forwarding* em pipelines.
4.4 – Pesquise na web e produza um resumo sobre *delay slots* em pipelines.
4.5 – Crie um via de dados para executar a instrução `sw`.
4.6 – Anexe a via de dados criada no exercício 2.5 na via de dados mais completa apresentada neste capítulo.
4.7 – Crie um via de dados para executar a instrução `j`.
4.8 – Anexe a via de dados criada no exercício 2.7 na via de dados mais completa apresentada neste capítulo.
4.9 – Crie um via de dados para executar a instrução `jal`.
4.10 – Anexe a via de dados criada no exercício 2.9 na via de dados mais completa apresentada neste capítulo.
4.11 – Crie um via de dados para executar a instrução `addi`.
4.12 – Anexe a via de dados criada no exercício 2.11 na via de dados mais completa apresentada neste capítulo.

Capítulo 5

Desempenho do Computador

5.1 – Introdução

Retomando nossa trajetória de leitura, nós já descobrimos os componentes básicos de um computador, a sua linguagem de montagem e de máquina e uma implementação em formato simples. A propósito, discutimos três possibilidades de implementação (mono-ciclo, multi-ciclo e pipeline). Concluímos que a abordagem pipeline nos permite terminar um programa em menos tempo e assim inferimos que a mesma é a melhor.

A discussão sobre o desempenho apenas começou. Vamos, neste capítulo, trabalhar com algumas métricas de desempenho utilizadas para afirmar que uma máquina é melhor que outra. Um cuidado extremo é preciso para ter tal afirmação creditada além do imaginário promovido pela mídia. A propósito, vender produtos significa torná-los necessários aos olhos do comprador, mesmo que seja por uma ilusão de ótica. A ressalva é mesmo feroz em nosso mundo competitivo. Um vendedor ilibado provavelmente teria dificuldades de vender um produto sem fazer uso de apelos comerciais. Não nos cabe discutir a ética de comércio, então vamos parar com nossa analogia por aqui, sem atingir um ponto de discórdia com profissionais do ramo.

Para começar a discussão, vamos primeiro tentar enxergar o tamanho do problema que se nos apresenta. Imagine um sistema com diversos processadores lentos sendo comparado com um sistema com um único processador rápido. O segundo provavelmente termina uma tarefa bem antes que qualquer dos componentes do primeiro, mas este tem capacidade de processar mais informação (em paralelo) por intervalo de tempo.

E então, qual dos dois é melhor? A resposta para esta pergunta é frustrante. Ela é: ‘depende!’. Se muitas tarefas são usadas simultaneamente provavelmente o primeiro sistema é melhor, mas se poucas tarefas são usadas, o segundo sistema pode ser melhor. Dizemos que o primeiro sistema tem

maior ***throughput*** do que o segundo. Do ponto de vista do usuário, a métrica de melhor ou pior está fortemente associada ao fato de uma tarefa executar mais rápido. Por isto, neste capítulo estaremos avaliando como melhorar o **tempo de execução** de um programa.

O tempo total que um usuário gasta desde o comando para executar uma tarefa até a sua conclusão é chamado de **tempo decorrido** (***elapsed time***). Dentro deste tempo estão incluídas operações de entrada e saída de dados, acessos a serviços do sistema operacional, tempo de espera por respostas das memórias e o tempo efetivamente utilizado para o processamento pela CPU. A Figura 5.1 ilustra esta divisão. O **tempo de CPU** (***CPUtime***) é o tempo que a CPU utiliza efetivamente para executar um determinado processo. Este tempo pode ser muito pequeno se comparado com o tempo decorrido.

Além disto, em sistemas operacionais multi-tarefas o tempo de CPU é distribuído para diversos processos o que torna menor ainda o tempo de CPU dedicado a um processo. Este tempo de CPU pode variar guiado por três fatores: quantidade de instruções do programa; freqüência de operação do processador; e a quantidade média de **ciclos por instrução**, CPI.

Quando implementamos a via de dados e o controle para executar algumas instruções do MIPS, trabalhamos com um variável importante: o período do clock. O clock é o relógio do sistema responsável pela sincronia entre os componentes do processador. Ele é uma onde quadrada com ciclo de trabalho ligeiramente diferente de 50%. O ciclo do clock é o período desta onda medido em segundos. A freqüência de operação do processador é o inverso do ciclo de clock.

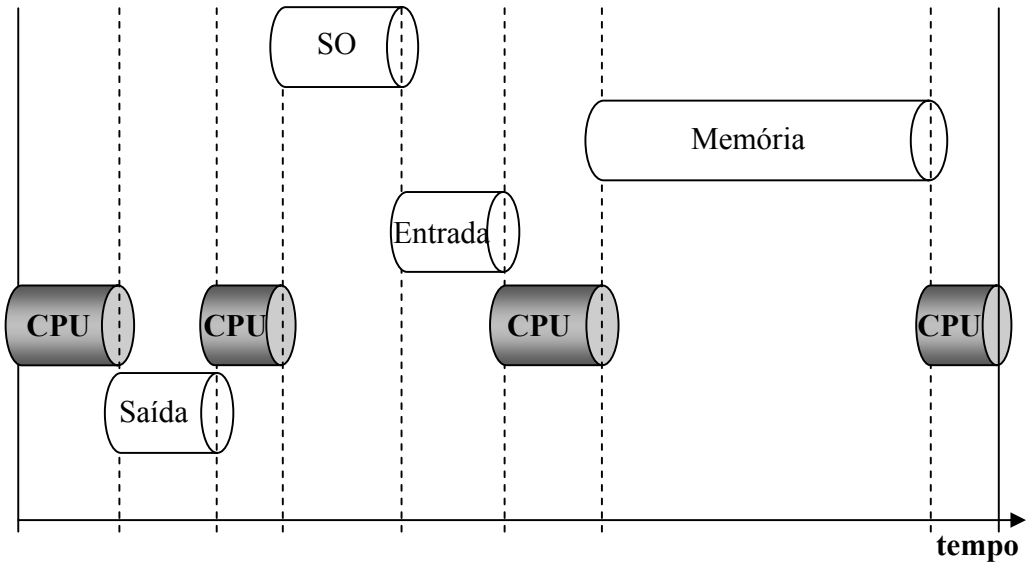


Figura 5.1: Tempo decorrido de uma tarefa

Quando estudamos pipeline percebemos que idealmente poderíamos ter uma instrução sendo terminada em cada ciclo de clock (após uma certa quantidade de instruções terem sido executadas). A tendência então é que tenhamos uma quantidade de ciclos por instrução muito próxima a 1. A realidade é muito mais cruel. Tipicamente para máquinas RISC este valor chega a 1,5 e para máquinas CISC este valor fica próximo a 4. Quanto menor este valor mais rápida será a execução do programa.

A quantidade de instruções necessárias para montar um determinado algoritmo é inversamente proporcional à complexidade de suas tarefas. Isto é, quanto mais complexo o conjunto de instruções, menos precisamos de instruções para confeccionar um programa. Só para darmos um exemplo básico, imagine que exista uma instrução `addm $8, $9, ($10)`, onde um dos operando da soma se encontra na posição de memória apontada por $10. Esta é uma instrução bem mais complexa que uma soma de valor armazenados em registradores (pense da implementação da via de dados). Ora, em nossa arquitetura MIPS seriam necessária duas instruções: `lw $11, 0($10)` e `add $8, $9, $11`. As instruções são mais simples, mas gastam mais espaço nos programas para sua implementação equivalente.

Bem, agora que conhecemos as partes que influenciam o tempo de CPU, vamos montar a equação fundamental da medida de desempenho do processador.

$$\text{Tempo de CPU} = \frac{\text{Tempo}}{\text{Ciclo}} \times \frac{\text{Ciclos}}{\text{Instruções}} \times \frac{\text{Instruções}}{\text{Programa}}$$

Esta equação norteia todo o cálculo de desempenho da CPU. Existem muitas técnicas para melhorar cada um destes fatores, ou seja, diminuí-los. Os modelos RISC investem pesadamente me diminuir Ciclos/Instruções, enquanto modelos CISC investem em reduzir o número de instruções por programa. A variável tempo/ciclo é um fator tipicamente guiado pela tecnologia de fabricação.

Outra coisa de fundamental importância é saber que estas variáveis não são tão independentes assim. Às vezes, modificar o hardware para alterar o número de ciclos por instrução também interfere no tamanho do ciclo do clock.

A variável ciclos por instrução é uma medida dinâmica, ou seja, não intrínseca ao processador e sua arquitetura. Para cada programa existe um valor que deve ser medido. Com várias medidas é possível calcular uma média e estimar o CPI, mas haverão códigos em que ele não ficará sequer próximo à

média. Naturalmente existem técnicas, como regularidade de decodificação, que ajudam a obtenção de CPI baixos.

Vamos ver um exemplo hipotético para que o leitor tenha uma melhor fixação do assunto. Para um determinado programa, uma máquina A possui um CPI de 1,2 ciclos/instrução. Uma máquina B possui, para o mesmo programa, um CPI de 2,4 ciclos/intrução. Este programa, para máquina A, necessita de 128 instruções, enquanto para máquina B ele precisa de 80 instruções. Queremos saber qual a relação entre as freqüências das máquinas para que as mesmas tenham o mesmo desempenho.

Considerando que o tempo de CPU deve ser igual temos:

$$\text{Tempo CPU}_{\text{maq.A}} = \text{Tempo CPU}_{\text{maq.B}}$$

$$\left(\frac{\text{Tempo}}{\text{Ciclo}} \times \frac{\text{Ciclos}}{\text{Instruções}} \times \frac{\text{Instruções}}{\text{Programa}} \right)_{\text{maqA}} = \left(\frac{\text{Tempo}}{\text{Ciclo}} \times \frac{\text{Ciclos}}{\text{Instruções}} \times \frac{\text{Instruções}}{\text{Programa}} \right)_{\text{maqB}}$$

$$\text{Tempo/Ciclo}_{\text{maqA}} . \ 1,2 \ .128 = \text{Tempo/Ciclo}_{\text{maqB}} . \ 2,4 \ . \ 80$$

$$\frac{\text{Tempo/Ciclo}_{\text{maqA}}}{\text{Tempo/Ciclo}_{\text{maqB}}} = \frac{2,4 \ . \ 80}{1,2 \ .128}$$

$$\frac{\text{Tempo/Ciclo}_{\text{maqA}}}{\text{Tempo/Ciclo}_{\text{maqB}}} = \ 1,25$$

$$\text{Tempo/Ciclo}_{\text{maqA}} = 1,25 \ . \ \text{Tempo/Ciclo}_{\text{maqB}}$$

$$\frac{1}{\text{Freqüência}_{\text{maqA}}} = 1,25 \ . \ \frac{1}{\text{Freqüência}_{\text{maqB}}}$$

$$\text{Freqüência}_{\text{maqB}} = 1,25 \ . \ \text{Freqüência}_{\text{maqA}}$$

Portanto concluímos que a freqüência da máquina B precisa ser 25% maior que a freqüência da máquina A.

A freqüência de uma máquina (com pipeline) depende do estágio do pipeline que tem o maior atraso. Recordemos o exemplo da Figura 4.14. Nele, afirmamos que o ciclo do clock deveria ser de 6ns. Isto significa que quanto maior a quantidade de dispositivos eletrônicos em um estágio do pipeline, menor será a freqüência da máquina. No exemplo da comparação de

freqüências entre as máquinas A e B, recém desenvolvido, temos a máquina A com valores típicos de RISCs e a máquina B com valores típicos de CISCs. Ora, o que se conclui do exemplo é que a freqüência de um CISC precisa ser maior que a freqüência de um RISC para que ambos tenham o mesmo desempenho. Acontece que, nos RISCs, os estágios de pipeline realizam menos tarefas que nos CISCs o que teoricamente beneficia o aumento de sua freqüência.

Infelizmente nossa conclusão é apenas parcial, uma vez que, como visto na Figura 5.1, o tempo de CPU não significa o tempo total que o usuário necessita para executar sua tarefa. A propósito, hoje, o tempo que a memória leva para responder ao processador é determinante no desempenho da máquina. Isto posto, e considerando que as máquinas CISCs normalmente apresentam maior densidade de código, em tese, elas são menos susceptíveis às baixas velocidades da memória. Isto poderia até mesmo inverter o quadro do desempenho das máquinas, tornando a CISC mais veloz que a RISC.

Enfim, a conclusão mais interessante é que deveríamos codificar e armazenar na memória programas tipo CISC e executar programas tipo RISC. Existem técnicas para isto que estão em desenvolvimento e algumas em uso, com tradutores de instruções CISCs em RISCs.

De volta à discussão matemática sobre as vantagens de um processador sobre outro definimos o ***speedup*** como sendo a razão entre o desempenho de uma máquina A e o desempenho de uma máquina B.

$$Speedup = \frac{\text{desempenho}_{\text{maqA}}}{\text{desempenho}_{\text{maqB}}}$$

Ora, o desempenho de uma máquina é composto de diversos fatores tempo de CPU, tempo de resposta da memória, entrada e saída etc. Nós costumamos simplificar a equação acima para apenas a componente do tempo da CPU.

$$Speedup = \frac{\text{Tempo CPU}_{\text{maqB}}}{\text{Tempo CPU}_{\text{maqA}}}$$

De fato poderíamos extrapolar o conceito de *speedup* para qualquer parte dos componentes de um desempenho, entretanto é preciso saber exatamente qual a influência deste componente no tempo final de execução (tempo decorrido) para um programa. Uma melhoria de 50% em alguma das

variáveis que compõem este tempo, não significa uma melhoria de 50% no desempenho da máquina.

Em 1967, George Amdahl quantificou suas observações sobre o fenômeno e derivou uma fórmula conhecida **Lei de Amdahl**. Em essência, a Lei de Amdahl define que o *speedup* de um sistema depende do *speedup* de um componente em particular, mas também da fração do tempo que este componente é usado pelo sistema. A fórmula é a seguinte:

$$Speedup = \frac{1}{(1-f) + f/k}$$

onde f é a fração do tempo utilizada pelo componente em particular; e
k é o *speedup* do componente em particular.

Vamos a um exemplo simples: Um sistema computacional passa 70% do seu tempo usando a CPU e os 30% restantes em disco. Uma nova CPU que executa os mesmos programas e que tenha 1,5 vezes a freqüência da anterior vai tornar o seu computador quantas vezes mais rápido?

Considerando que a CPU mudou apenas sua freqüência podemos dizer que o tempo de CPU da nova máquina é de 1,5 vezes menor que na configuração anterior. Usando então a Lei de Amdahl, temos:

$$Speedup = \frac{1}{(1-0{,}7) + 0{,}7/1{,}5} = 1{,}3$$

ou seja, a máquina passará a ser 1,3 vezes mais veloz. Se o disco fosse alterado por outro com a mesma melhoria de desempenho, 1,5 vezes, o desempenho final seria apenas:

$$Speedup = \frac{1}{(1-0{,}3) + 0{,}3/1{,}5} = 1{,}1$$

1,1 vezes mais rápido que o sistema anterior.

Isto deixa claro que temos de investir nas partes que mais influenciam na velocidade do sistema. Investir em um componente que raramente é utilizado significa pouca melhoria do sistema como um todo. Daí deriva a seguinte observação: faça mais simples (e conseqüentemente mais rápido) o

que se faz freqüentemente. Esta é uma grande máxima da Arquitetura de Computadores.

Um outro aspecto que é muito importante é a forma de medição do CPI de um processador. Um projetista precisa de simuladores da sua arquitetura para gerar estatísticas de tal forma que o CPI possa ser mensurado. Outra opção é esperar que o hardware fique pronto e daí medir *in loco*. Algumas vezes é possível mensurar o número de ciclos de CPU olhando para o CPI de uma determinada classe de instruções e sua freqüência de execução. Neste caso definimos:

$$\text{Ciclos de Clock da CPU} = \sum_{i=1}^{n} (CPI_i \times C_i)$$

onde CPI_i é o número de ciclos por segundo médio para uma determinada classe de instruções e C_i é o número de instruções desta classe que é efetivamente executado. *n* é o número de classes de instruções.

Por exemplo, no MIPS as instruções aritméticas são executadas em média a 1 ciclo por instrução. Instruções de transferência de dados entre a memória e o processador executam a uma taxa de 2 ciclos por instrução. Supondo que um determinado programa usa 60% de instruções aritméticas e 40% de instruções de carga, qual o número de ciclos de clock utilizados na CPU para um programa com 200 instruções executadas?

$$\text{Ciclos de Clock da CPU} = 1 \times 0{,}6 \times 200 + 2 \times 0{,}4 \times 200 = 280$$

portanto, o número de ciclos da CPU é estimado em 280.

5.2 – Métricas de desempenho

Não é incomum encontrarmos propagandas de processadores que prometem oferecer milhões de instruções por segundo, o que seria melhor que outros que oferecessem menos. De fato, milhões de instruções por segundo, ou **MIPS** (***Millions of Instructions Per Second***) é uma métrica popular para comparamos computadores. Não confunda a métrica MIPS com o processador MIPS.

Esta métrica é interessante porque um número de MIPS maior significa um processador melhor, o que é intuitivo, enquanto a medida de tempo significa que quanto menor o tempo de CPU, melhor o processador.

Infelizmente a métrica carece de alguns cuidados. Primeiro, nós não podemos comparar máquinas com conjuntos de instruções diferentes usando MIPS. Por que? Simplesmente porque os programas podem ser expressos de formas diferentes o que leva a uma contagem de instruções diferentes e a taxa de execução das instruções não significa que uma máquina é melhor que outra, já que o número de instruções sofre variação.

Segundo, MIPS varia de programa para programa, por isto não pode existir um único número mágico que exprima o valor do MIPS para o processador.

Terceiro e mais importante: o MIPS pode nos levar a condições adversas de desempenho, ou seja, uma máquina com menor MIPS pode ser mais rápida que outra com MIPS maior. O termo MIPS foi até elevado a uma chacota no seu significado, onde alguns afirmam ser *Meaningless Indicators of Performance for Salesmen* (indicador de desempenho sem sentido para vendedores). Tudo por causa da possível confusão com os resultados.

Vamos ver um exemplo. Um computador A executa 10 bilhões de instruções, usando uma freqüência de 4Ghz e com um CPI de 1,0. Um computador B executa 8 bilhões de instruções, usando a mesma freqüência de A, mas com um CPI de 1,1. Qual das duas apresenta maior MIPS e qual o mais rápido?

$$\text{Tempo de Execução} = \text{Ciclos de Clock} \times \text{Período do Clock} = \frac{\text{Ciclos de Clock}}{\text{Freqüência}}$$

$$\text{Ciclos de Clock} = \text{CPI} \times \text{Número de Instruções}$$

$$\text{Ciclos de Clock}_A = 1{,}0 \times 10 \times 10^9$$

$$\text{Tempo de Execução}_A = \frac{10 \times 10^9}{4 \times 10^9} = 2{,}5\text{s}$$

$$\text{Ciclos de Clock}_B = 1{,}1 \times 8 \times 10^9$$

$$\text{Tempo de Execução}_B = \frac{8{,}8 \times 10^9}{4 \times 10^9} = 2{,}2\text{s}$$

$$MIPS = \frac{\text{Instruções}}{\text{Tempo de Execução} \times 10^6}$$

$$MIPS_A = \frac{10 \times 10^9}{2{,}5 \times 10^6}$$

$$MIPS_A = 4.000 \text{ MIPS}$$

$$MIPS_B = \frac{8 \times 10^9}{2{,}2 \times 10^6}$$

$$MIPS_B = 3.636 \text{ MIPS}$$

Portanto, concluímos que o computador A possui maior MIPS, mas demora mais para executar os programas, em relação a B. Isto demonstra como a métrica pode atrapalhar.

Outra métrica que pode ser vexatória é MFLOPS (*millions of floating point operations per second*). A idéia por trás de MFLOPS é a mesma do MIPS, mas agora são consideradas apenas as instruções de ponto flutuante no cálculo. O problema com MIPS e MFLOPS é que eles não levam em consideração a organização do computador em relação à quantidade de ciclos gastos por instrução, o que pode alterar completamente os resultados como visto anteriormente.

5.3 – A visão do software – Benchmarking

A forma mais precisa de avaliar um computador em relação a outro é certamente executar um mesmo conjunto de programas em cada um deles e verificar qual apresenta os melhores resultados em termos de tempo. Este conjunto de aplicações é chamado de ***carga de trabalho****,* **workload**. *Infelizmente, este método nem sempre é possível. Primeiro porque cada usuário utiliza comumente um conjunto de aplicações particulares e a relação de melhor ou pior estaria restrita àquele usuário. Segundo porque quando estamos desenvolvendo um novo processador não temos todas as*

ferramentas, como compiladores e montadores e sistemas operacionais já prontos para gerar e pôr em execução os programas. Dentro deste contexto, escolher trechos de códigos ou rotinas que sejam executadas freqüentemente nos programas reais torna o trabalho de desenvolvimento e avaliação menos árduo. Além disto, os trechos são desenhados especificamente para medição de desempenho e por isto podem representar mesmo a situação real.

Estes trechos de código são chamados de **benchmarks**. *Um conjunto de* benchmarks *é chamado de uma suíte de* benchmarks. *Muitos* benchmarks *são desenhados especificamente para medir o desempenho da CPU, sendo excluídos os trechos que utilizam entrada e saída e os que requerem serviços do sistema operacional. Este tipo de* benchmark *é chamado de* **kernel** *(não confunda com os* kernels *do sistema operacional).*

Quanto mais específicas forem as aplicações de um processador, mais específicos podem ser os benchmarks *e mais precisos serão os resultados. Em nossos computadores de uso doméstico construir uma suíte de benchmarks é uma tarefa muito difícil. De qualquer forma, a mais famosa suíte é chamada de SPEC CPU. Esta suíte criada pela SPEC (*Standard Performance Evaluation Corporation*) é utilizada hoje com muito sucesso para medir o desempenho relativo entre máquinas e entre versões com diferentes componentes de uma determinada máquina. Seus 25 programas são divididos em dois conjuntos, SPECint e SPECfp, para testes de computação inteira e em ponto flutuante respectivamente.*

Uma crescente demanda de processadores hoje em dia é para os ditos sistemas embarcados que realizam processamentos específicos e entregam resultados para um sistema central. Muitos processadores são utilizados em sistemas embarcados e as principais suítes de benchmarks *para medidas de desempenho de tais sistemas são Mediabench e MiBench.*

Uma outra suíte muito conhecida é DSPstone. Esta suíte é desenvolvida para medir o desempenho de processadores de sinais digitais, DSPs. Os DSP normalmente são utilizados em sistemas embarcados, mas têm aplicação restrita ao tratamento de sinais digitais, como filtro FIR, IIR, transformada de Fourier e outros.

Escolher corretamente as suítes para fazer comparações entre máquinas é tarefa importante para qualquer trabalho científico.

Outra face de um benchmark *é a sua carga de dados particular. Um benchmark como cjpeg pode converter diversos formatos de arquivos de imagens para o formato jpeg. Escolher qual o formato de entrada pode interferir nos resultados. Assim também, os tamanhos dos dados de entrada devem estar padronizados para que uma comparação seja ilibada.*

5.4 – Conclusões

Aprendemos neste capítulo como calcular o desempenho de uma máquina em particular e vimos que o tempo de execução é a métrica mais acurada para apresentar vantagens de um computador sobre outro. Vimos que nem sempre outras métricas nos levam a resultados justos sobre o melhor computador.

Aprendemos também que alterar uma parte do sistema pode nos levar a melhorias pequenas de desempenho. Especificamente, aprendemos a lei de Amdahl, que pode ser extrapolada, com restrições, para qualquer assunto de nossas vidas. O grande colorário é: "faça o caso comum eficiente". Naturalmente que o estado da arte exige um pouco mais. Os detalhes não podem ser dispensados, mas sabemos que o esforço com eles não irá alterar substancialmente o desempenho geral.

5.5 – Prática com Simuladores

Este é um capítulo *sui-generis* em termos de simulação. O principal argumento que podemos extrair da prática com simuladores é a quantidade de ciclos por instrução para uma determinada aplicação. Mesmo assim, o fato de conhecer os *benchmarks* e operá-los pode ser muito útil.

Uma tarefa que comumente encontramos usando *benchmarks* é descobrir os *hot-spots* dos programas, ou seja, aquelas regiões que são utilizadas com muita freqüência. Isto é muito útil para que otimizações específicas possam ser realizadas nos códigos. Para isto utilizamos *traces* de referências à memória. Estes *traces* contém todos os endereços que foram utilizados pelo programa, indicando quais os caminhos percorridos. Assim podemos encontrar quais pontos foram mais acessados, onde se encontram os laços mais importantes do programa etc. Estes *traces* também servem para medir o desempenho do sistema de memórias. Veremos no capítulo seguinte como este sistema se comporta e seus benefícios.

5.6 – Exercícios

5.1 – Pesquise na web e produza um resumo sobre os programas que compõem a suíte CPU SPECint.

5.2 – Pesquise na web e produza um resumo sobre os programas que compõem a suíte CPU SPECfp.

5.3 – Pesquise na web e produza um resumo sobre os programas que compõem a suíte MiBench.

5.4 – Pesquise na web e produza um resumo sobre os programas que compõem a suíte Mediabench.

5.5 – Pesquise na web e produza um resumo sobre os programas que compõem a suíte DSPstone.

5.6 – Digamos que um sistema passe 70% de seu tempo executando instruções na CPU e 30% de seu tempo esperando pelo disco. Um vendedor oferece um novo processador que é 50% mais rápido e custa R$ 10.000,00. Um outro vendedor apresenta-lhe um novo sistema de discos, duas vezes e meio mais rápido com custo de R$7.000,00. Você dispõe de recursos para apenas um investimento, qual dos dois lhe proporcionará melhores resultados com menor custo?

5.7 – Agora você dispõe de recursos para realizar ambos os investimentos da questão anterior. Qual a melhoria de desempenho final do seu sistema?

5.8 – Considere um computador com três classes de instruções, A, B e C, cujos CPIs de cada classe são 1, 2 e 3 respectivamente. Agora vamos testar dois compiladores que geram código para um determinado programa em HLL e encontramos que o primeiro gera 5 bilhões de instruções da classe A, e 1 bilhão de instruções para cada classe restante. O segundo compilador gera 10 bilhões de instruções da classe A e 1 bilhão de instruções para cada classe restante. Assuma que todas as instruções geradas pelos compiladores são executadas nestas mesmas proporções e que a freqüência de operação do processador é de 4Ghz. Qual compilador tem o melhor desempenho segundo o tempo de execução? E segundo o MIPS?

5.9 – Em que situação a métrica MIPS pode ser empregada para comparar máquinas de forma inequívoca?

Capítulo 6

Sistema de Memórias

6.1 – Introdução

A maioria das máquinas atuais utiliza a arquitetura de Von Neumann como molde. Isto implica na adoção de um modelo fortemente baseado em memória, onde dados e instruções são alocados em um repositório único. De fato, a memória pode ser vista como um vetor onde cada posição (índice) contém uma instrução/dado, iniciando do endereço zero até o máximo valor de endereçamento do processador. Esta organização de uma memória é puramente lógica do ponto de vista de sua materialização. Desde os anos 60 já fora percebido que a materialização do conceito de memória, como descrita acima, exigiria uma enorme quantidade de recursos. Memórias grandes e rápidas, abstendo-se da discussão filosófica-temporal dos adjetivos, são caras e ocupam muito espaço físico para sua implementação. Por isto, usamos hoje um sistema de memórias formando uma hierarquia. Memórias mais rápidas, antes denominadas ***escravas*** [6.1], agora integram praticamente qualquer sistema de computação. Maurice Wilkes cunhou o termo em 1965, mas foi a IBM que em 1968 introduziu no 360/85 uma memória ***cache***, como são conhecidas hoje as memórias escravas.

Se voltarmos à nossa analogia inicial sobre o sistema de computação, vamos perceber que temos dentro da cozinha uma pequena despensa onde podemos guardar os ingredientes das nossas guloseimas. Certamente só faz sentido guardar nela os ingredientes mais comumente utilizados. Quando precisamos utilizar um ingrediente que não está em nossa prateleira, podemos verificar se está disponível no supermercado, mas isto requer mais tempo e mais recursos, já que temos de sair da cozinha, ir até o supermercado, encontrar a prateleira onde está o produto, pegá-lo, pagá-lo e trazê-lo para casa. Só então o ingrediente estará em nossa despensa pronto para ser utilizado. Se o produto não estiver no supermercado é preciso solicitar a um

distribuidor, que está ainda mais distante. Neste caso, é preciso esperar ainda mais e os custos envolvidos são ainda maiores. Ora, é impensável termos em nossas casas um grande silo para depositar todos os ingredientes possíveis. Isto seria caro e ocuparia uma enorme área. Também no supermercado, é impossível armazenar todo e qualquer tipo de produto. A solução é uma hierarquia! Agora podemos perceber que o sistema de memória é exatamente análogo.

Formalizando os conceitos, uma hierarquia típica de memória começa com um módulo rápido, pequeno e caro chamado ***cache***, seguido por uma memória mais lenta, maior e mais barata chamada ***memória principal***. Estas memórias são construídas de materiais semicondutores (tipicamente transistores CMOS). As memórias em semicondutores são seguidas, dentro da hierarquia, por memórias ainda maiores, mais lentas e baratas, que são implementadas em material magnético (os discos rígidos – *Hard Disk*, HD – são um exemplo deste tipo de memória). Este nível da hierarquia é chamado de memória secundária ou de massa. Existe ainda um outro nível classificado como memória terciária, também em material ferromagnético, que são as fitas. Estas são consideradas ***memória terciária***.

O objetivo de criar uma hierarquia de memória é criar uma ilusão para o programador de que ele pode acessar uma grande memória com uma enorme velocidade. A caracterização de uma hierarquia pode ser feita por inúmeros parâmetros, dentre eles: capacidade, tempo de ciclo, latência, penalidade por falha/falta (*miss penalty*), largura de banda (*bandwidth*) e custo. A capacidade de um módulo de memória é medida em bytes e seus múltiplos (Kilo, Mega, Giga, Tera etc.). O tempo de ciclo é o tempo decorrido desde o início de uma leitura até a memória estar pronta para uma nova leitura. A latência é o tempo desde o início de uma leitura até o dado estar disponível para o usuário. A penalidade por falha/falta é o tempo, normalmente expresso em ciclos de relógio, que leva para o dado/instrução que não está presente em um determinado nível da hierarquia ser recuperado do nível inferior. A largura de banda provê uma medida da quantidade de bits que podem ser acessados por segundo. Finalmente, o custo é o valor a ser pago por (giga) bytes de memória. A ilustração típica de uma hierarquia pode ser vista na Figura 6.1. Veja que o banco de registradores também pode ser considerado como parte desta hierarquia. Os tempos de acesso são sazonais, ou seja, dependendo da tecnologia eles podem ser outros, mas guardando uma certa proporção entre as camadas.

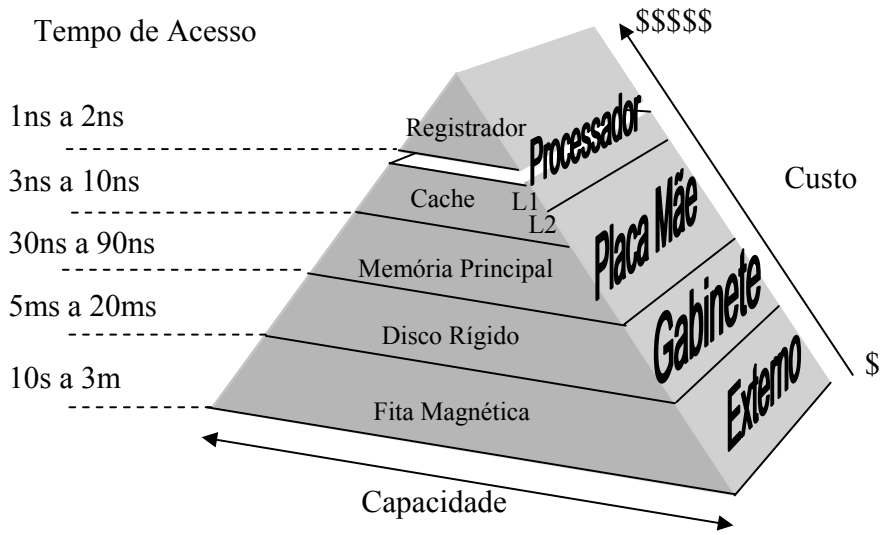


Figura 6.1: Típica hierarquia de memórias

Podemos ainda classificar as memórias como volátil ou não volátil. Uma memória é dita volátil quando cessada a sua alimentação ela perde todos os seus dados. As memórias cache e principal em sua maior parte são voláteis. Memórias não voláteis são aquelas que mesmo estando desligadas guardam os dados integralmente. Os discos rígidos e as fitas são considerados memórias não voláteis. Existe um tipo de memória em silício que também pode ser considerada não volátil: a memória ROM. ROM significa memória para leitura apenas (*Read Only Memory*). Este tipo de memória costuma guardar informações importantes sobre o sistema como a seqüência de inicialização dos dispositivos (*boot*). Existe uma flexibilização ao conceito de memória ROM que é uma memória que se pode regravar, chamada EPROM. De fato ela costuma ser gravada uma única vez, mas é possível apagar o seu conteúdo e reprogramá-la. Daí o seu nome: Memória ROM apagável e programável (*Erasable, Programmable* ROM).

Uma outra forma de classificar as memórias diz respeito à recuperação dos dados nelas armazenados. Uma memória pode ser RAM ou seqüencial. A memória RAM (*Radom Access Memory*) acessa qualquer dado em um tempo fixo, ou seja, se a latência da memória é de 30ns, qualquer que seja o endereço do dado solicitado ela vai levar 30ns para entrega do valor. As memória de acesso seqüencial, por outro lado, não possuem uma latência fixa, pois para acessar o dado no endereço 300 ela precisa percorrer desde o último endereço solicitado até o endereço 300. Por exemplo, para mudar de um endereço para outro adjacente, uma memória de acesso seqüencial precisa de 100ns e para

entregar o dado ao solicitante ela requer 15ns. Considerando que dois dados são solicitados a esta memória, o primeiro no endereço 10 e o segundo no endereço 25, qual o tempo desde o primeiro acesso para que o segundo dado esteja disponível? Bem, depois do primeiro acesso é preciso percorrer 15 endereços até chegar ao endereço solicitado o que leva 1500ns e mais 15ns para entregar o dado, totalizando 1515ns para o segundo dado ser entregue ao solicitante. As memórias principais e cache (e também os registradores) são memórias RAM, enquanto as fitas são memórias de acesso seqüencial.

A diferença de velocidades encontrada entre as memórias cache e principal é fruto da tecnologia empregada em sua construção. Memórias cache são implementadas com tecnologia SRAM (*Static* RAM) enquanto a tecnologia DRAM (*Dynamic* RAM) é usada para memórias principais. Memórias DRAM precisam de um circuito especial para manter os dados armazenados. Este circuito é chamado de circuito de refrescamento. Ele fica de tempos em tempos lendo os dados e re-gravando-os em suas posições corretas. Isto torna a memória DRAM mais lenta, mas o custo do armazenamento é bem menor que o custo de uma memória SRAM.

Voltando agora ao tema da hierarquia de memória, a questão que deve ser levada em consideração é a seguinte: Quais os dados/instruções que devem ser selecionados para os níveis mais altos da hierarquia? A resposta a esta questão depende do comportamento dos acessos a dados e códigos de um programa. Tipicamente usamos dados e/ou instruções que se encontram juntos ou próximos dentro da memória e também tendemos a usar dados e/ou instruções que usamos recentemente. Esta observação define o princípio da localidade de referências, sobre o qual as hierarquias de memórias são construídas com fins de otimizar o desempenho do sistema. Se a maior parte dos dados/instruções estão presentes na porção de memória mais rápida então, o acesso aos dados/códigos será mais rápido e por isto selecionar bem estes dados e instruções é fundamental. Os níveis mais altos da hierarquia, a saber: a memória principal e a cache (e, por conseguinte, o banco de registradores) armazenam dados e instruções temporárias para uso o mais imediato possível pelo processador, então, de fato, elas não armazenam dados perenes, mas fazem cópias dos dados/instruções contidos no sistema de discos/fitas para serem utilizados mais imediatamente.

Na prática, os processadores tendem a acessar a memória de forma muito repetitiva. Por exemplo: o valor do registrador PC é sempre incrementado de 4 unidades, assim a próxima instrução a ser executada é sempre a seguinte na memória. Isto implica que já sabemos qual o conjunto de instruções vai ser utilizado dentro de um certo intervalo de tempo. Naturalmente, instruções de desvio interferem nesta seqüência de execução

das instruções, mas considerando que tipicamente temos uma instrução de desvio a cada 5 instruções (que não saltam) é interessante percebermos que a probabilidade de virmos a utilizar uma instrução vizinha da sua antecessora na memória é muito alta. Este é o princípio da ***localidade espacial***, que diz que a probabilidade de acessarmos instruções vizinhas é muito alta. O princípio da localidade espacial é também usado para dados, por exemplo, quando fazemos algum processamento sobre um vetor.

Um outro tipo de localidade de referência é a ***localidade temporal***, que diz que itens acessados recentemente tendem a ser acessados novamente em um futuro muito próximo. Isto deriva do fato de que ao programarmos nós tendemos a realizar operações com um pequeno conjunto de variáveis por vez. Isto faz com que utilizemos as mesmas posições de memória repetidas vezes. Também, a presença de laços (*loops*) na programação faz com que instruções sejam executadas repetidamente.

O princípio da localidade provê a oportunidade para o sistema usar uma pequena porção de memória rápida para efetivamente acelerar a maioria dos acessos à memória. Tipicamente apenas uma pequena porção do espaço inteiro de memória é usada por vez e esta porção costuma ser acessada freqüentemente. Assim, podemos 'escolher' esta porção para ser transferida para uma região alta na hierarquia de memórias, onde o acesso é mais rápido. Isto resulta em um sistema de memória que pode armazenar uma grande porção de informações em uma memória de baixo custo e ainda provê quase a mesma velocidade de acesso a estas informações como se fosse utilizada uma enorme e cara memória rápida.

A seqüência de eventos que ocorre quando o processador pede um dado/instrução ao sistema de memórias é o seguinte: primeiro ele procura na cache. Se a informação estiver presente a cache devolve a informação para o processador. Isto é chamado de acerto na cache (*hit*). Se a informação não está na cache é buscada na memória principal e uma falha na cache é computada (*miss*). Se o dado estiver disponível na memória principal ele é entregue à cache que o entrega ao processador. Para melhor aproveitar o princípio da localidade espacial, não apenas a informação solicitada é copiada da memória principal para memória cache, mas um bloco inteiro de informações é repassado. O tamanho deste bloco será discutido brevemente. Se a informação não está presente na memória principal, então dizemos que ocorreu uma falta de página (*page fault*) e a informação deve ser buscada no disco, e assim sucessivamente.

A probabilidade de um acerto na cache (h_{cache}) é chamada de taxa de acerto da cache e o seu complemento é a taxa de falha na cache ($1- h_{cache}$). Tipicamente uma cache é projetada para ter uma taxa de acerto próxima a

95%, ou seja, em 95% dos acessos, as informações buscadas estão na cache. Naturalmente estes são números puramente estatísticos. Muitos programas 'mal comportados' podem apresentar estatísticas muito ruins para uma determinada cache e assim subutilizar os recursos computacionais.

Perceba que pusemos de fora desta abordagem a questão dos registradores. O banco de registradores, apesar de ser considerado parte integrante da hierarquia, é controlado pelo programa, então cabe ao programador escolher que informações devem ser transferidas para o banco de registradores em um determinado momento.

6.2 – Memórias Cache

Em 1965 Maurice Wilkes distinguiu dois tipos de memória principais: a primeira, convencional e a segunda, uma memória escrava, rápida e pequena, para armazenar apenas as informações mais freqüentemente utilizadas pelos programas. Assim nasceu a memória cache, que é uma região para guardar informações mais freqüentemente acessadas pelo processador. A escolha das informações a serem duplicadas (copiadas) da memória principal na cache é feita pelo hardware que controla a cache e se baseia nos princípios da localidade espacial e temporal. Atualmente encontramos caches em dois níveis, ou seja, duas memórias caches fazendo parte da hierarquia de memórias. Normalmente encontramos uma cache chamada L1 (nível 1, *Level* 1) e L2 (nível 2, *Level* 2). Conforme visto na Figura 6.1, a cache L1 está localizada dentro do processador e a cache L2 fora do processador, na placa mãe. L1 e L2 seguem os mesmos princípios da hierarquia: L1 mais rápida, menor e mais cara de implementar que L2.

A Figura 6.2 mostra o esquema básico de funcionamento de uma cache. Em um dado instante de tempo a cache e a memória principal estão com as informações representadas em (a). O processador então pede a informação Info9. A cache é verificada para saber se Info9 está presente. Caso afirmativo a cache entrega Info9 para o processador. Na ilustração, Info9 não está presente na cache, então é preciso buscá-la no nível inferior da hierarquia, a memória principal, e fazer uma cópia dela para dentro da cache. A cache então contém agora Info9 e pode entregá-la ao processador (b).

Neste ponto surgem as questões que precisamos resolver:

- Como saber se um dado em particular está na cache?
- Se estiver na cache, como saber em que posição da cache ele está?
- Caso a cache esteja cheia, que informação deve ser retirada para ceder espaço à nova?

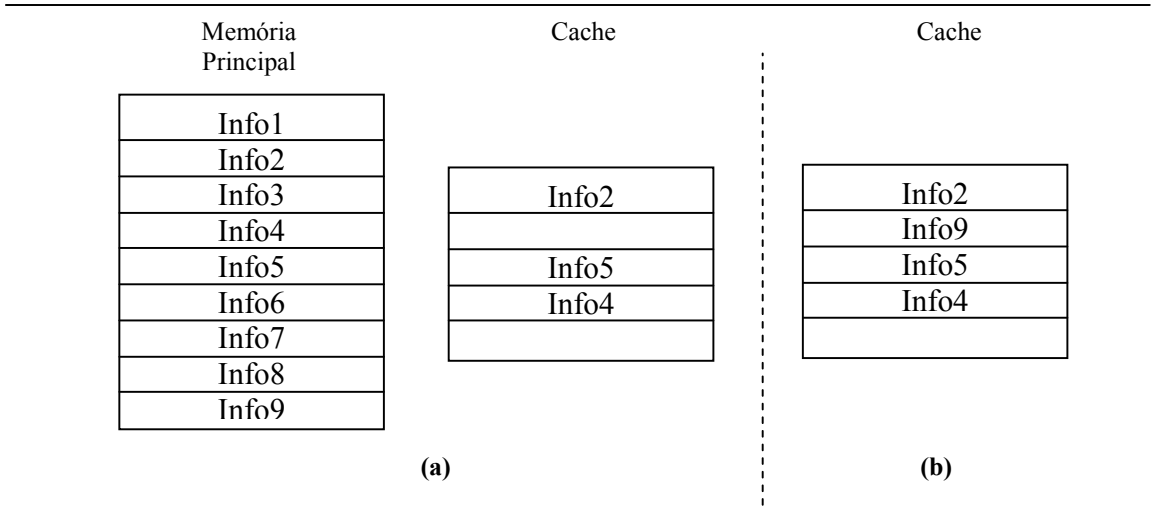


Figura 6.2: Cache antes (a) e depois do acesso à informação Info9 (b)

Vamos começar respondendo à segunda questão. A maneira mais prática para saber onde uma informação da memória principal vai parar na cache é atribuir à cada posição da cache um conjunto de possíveis posições da memória principal. Este mapeamento é feito com base no endereço da informação sendo solicitada. Alguns bits que compõem este endereço indicam em que posição da cache a informação deve ser buscada. No exemplo da Figura 6.3 podemos perceber que os endereços 0000_2, 0100_2, 1000_2 e 1100_2 partilham a mesma posição na cache, assim como os endereços 0011_2, 0111_2, 1011_2 e 1111_2. Este tipo de mapeamento é chamado de mapeamento direto e a cache que implementa esta estratégia é chamada de ***Cache Mapeada Diretamente***.

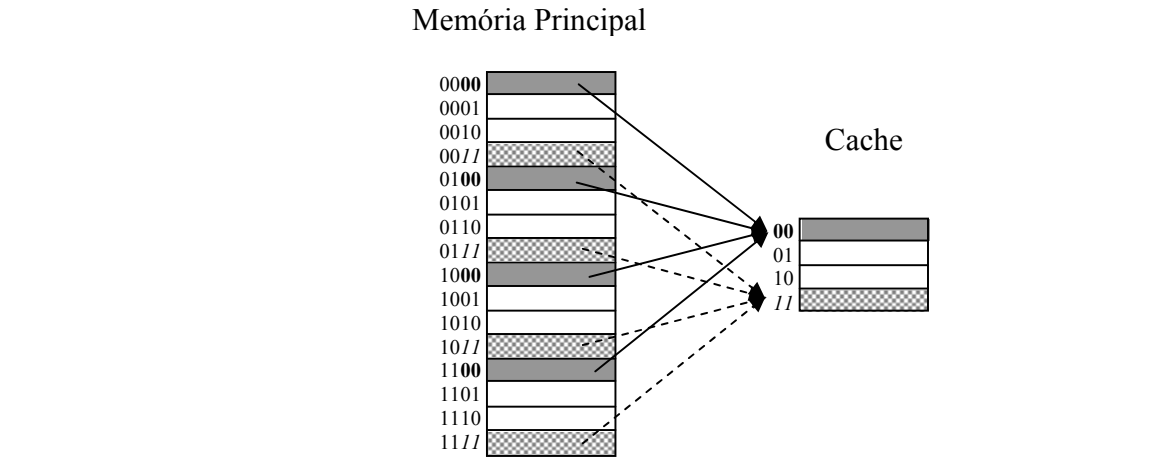


Figura 6.3: cache de mapeamento direto

O fato de *n* endereços serem mapeados para uma única posição da cache implica em sabermos qual dos *n* está em um determinado instante ocupando um lugar. Por exemplo, na posição 00 da cache poderiam estar as informações contidas nos endereços 0000_2, 0100_2, 1000_2 ou 1100_2. Isto significa que é preciso identificar qual dos endereços está contido na cache. Para resolver este problema as informações de endereço são guardadas em uma posição adjacente ao dado/instrução na cache. Esta posição adjacente é chamada de rótulo (*tag*). A Figura 6.4 mostra como fica uma memória cache de mapeamento direto com rótulo. Além do campo onde são armazenadas as informações, um campo de rótulo está mostrado. Veja que a concatenação do valor da *tag* com os endereços da cache (chamados de índices) forma o endereço original da informação na memória principal.

Quando o processador pede ao sistema de memórias uma informação ele simplesmente informa o endereço onde o dado se encontra na memória principal, então a cache desmembra este endereço, descobre qual índice ele contém e verifica se na posição correspondente na cache o *tag* é o mesmo do endereço solicitado. Se for, acontece um acerto, se não uma falha. Desta forma respondemos a nossa primeira pergunta feita acima.

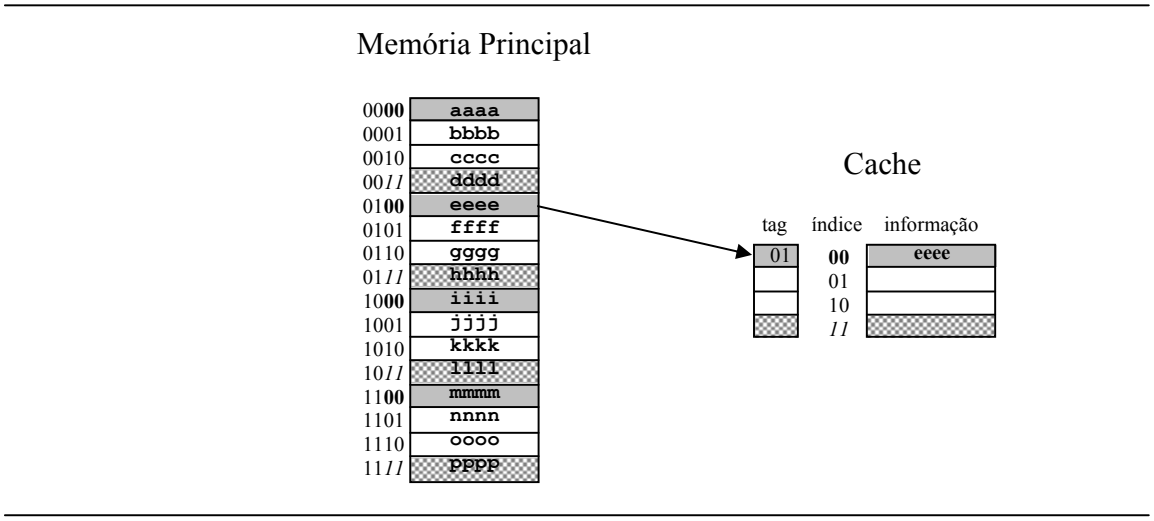


Figura 6.4: Cache de mapeamento direto com rótulo

Agora vamos fazer um exercício de fixação com uma cache mapeada diretamente. Suponha a cache mostrada na Figura 6.4 e uma seqüência de acessos de leitura mostrada na Figura 6.5 (a). O primeiro acesso é ao endereço 0000_2, conforme mostra a Figura 6.5 (b). O gerenciador da cache recebe o endereço e desmembra-o em índice e *tag*. O índice é formado pelos dois

últimos bits do endereço (os bits mais à direita) e a *tag* o restante dos bits do endereço. Ora, no índice 00 da cache não há informações na cache, portanto uma falha é gerada e a informação é buscada na memória principal. Neste caso a informação do endereço 0000_2 é aaaa. Junto com a informação é também armazenado o *tag* correspondente (neste caso os dois primeiros bits do endereço, a saber: 00).

O próximo endereço a ser solicitado é o 0100_2 (Figura 6.5 (c)). O gerenciador da cache separa os campos do endereço (*tag* = 01_2 e índice = 00_2). Ele então verifica que a posição da cache correspondente ao índice 00_2 contém uma informação. Ele checa, então se os *tags* conferem (presente na cache, *tag* = 00_2, solicitado pelo processador, *tag* = 01_2). Neste caso os *tags* diferem, portanto a informação que está presente na cache não é a mesma requerida pelo processador. Uma nova falha é anotada e o endereço correspondente é solicitado à memória principal. A informação na cache é, então, atualizada para o novo dado que veio da memória principal. Numa cache mapeada diretamente cada dado tem um lugar específico para alocar. Se houver outro dado, este precisa ser removido. Assim respondemos a nossa terceira pergunta.

O terceiro acesso é ao endereço 1110_2 (Figura 6.5(d)). O gerenciador da cache separa os campos do endereço (*tag* = 11_2 e índice = 10_2). O índice 10_2 na cache está vazio, portanto ele vai buscar a informação na memória principal e armazena o dado e a *tag* na posição correspondente ao índice 10_2.

O próximo acesso é ao endereço 0100_2. O gerenciador da cache faz a separação dos campos (*tag* = 01_2, índice = 00_2). No índice correspondente na cache à 00_2, existe uma informação. Os *tags* são então comparados (*tag* na cache = 01_2, *tag* solicitado = 01_2). Neste caso, os *tags* presente na cache e solicitado são iguais, então acontece um acerto na cache. O dado (eeee) é imediatamente repassado ao processador.

A Figura 6.5 (f) mostra o resultado final das operações de leitura na cache. Fica como exercício o preenchimento da cache em cada tempo (acesso) envolvido nesta seqüência de acessos.

(a)

Tempo	Leitura(L) Escrita (E)	Endereço (binário)	Hit	Miss	Cache (índice)			
					00	01	10	11
					tag/dado	*tag*/dado	*tag*/dado	*tag*/dado
1	L	0000						
2	L	0100						
3	L	1110						
4	L	0100						
5	L	0011						
6	L	0100						
7	L	1101						
8	L	0011						

Figura 6.5: Exemplo de acessos a uma cache

(b)

Tempo	Leitura(L) Escrita (E)	Endereço (binário)	Hit	Miss	Cache (índice)			
					00	01	10	11
					tag/dado	*tag*/dado	*tag*/dado	*tag*/dado
1	**L**	**0000**		X	00/aaaa			
2	L	0100						
3	L	1110						
4	L	0100						
5	L	0011						
6	L	0100						
7	L	1101						
8	L	0011						

(c)

Tempo	Leitura(L) Escrita (E)	Endereço (binário)	Hit	Miss	Cache (índice)			
					00	01	10	11
					tag/dado	*tag*/dado	*tag*/dado	*tag*/dado
1	L	0000		X	00/aaaa			
2	**L**	**0100**		X	01/eeee			
3	L	1110						
4	L	0100						
5	L	0011						
6	L	0100						
7	L	1101						
8	L	0011						

(d)

Tempo	Leitura(L) Escrita (E)	Endereço (binário)	Hit	Miss	Cache (índice)			
					00	01	10	11
					tag/dado	*tag*/dado	*tag*/dado	*tag*/dado
1	L	0000		X	00/aaaa			
2	L	0100		X	01/eeee			
3	**L**	**1110**		X	01/eeee		11/nnnn	
4	L	0100						
5	L	0011						
6	L	0100						
7	L	1101						
8	L	0011						

(e)

Tempo	Leitura(L) Escrita (E)	Endereço (binário)	Hit	Miss	Cache (índice)			
					00	01	10	11
					tag/dado	*tag*/dado	*tag*/dado	*tag*/dado
1	L	0000		X	00/aaaa			
2	L	0100		X	01/eeee			
3	L	1110		X	01/eeee		11/nnnn	
4	**L**	**0100**	X		01/eeee		11/nnnn	
5	L	0011						
6	L	0100						
7	L	1101						
8	L	0011						

Figura 6.5: Exemplo de acessos a uma cache

(f)

Tempo	Leitura(L) Escrita (E)	Endereço (binário)	Hit	Miss	Cache (índice)			
					00	01	10	11
					tag/dado	*tag*/dado	*tag*/dado	*tag*/dado
1	L	0000		X	00/aaaa			
2	L	0100		X	01/eeee			
3	L	1110		X	01/eeee		11/nnnn	
4	L	0100	X		01/eeee		11/nnnn	
5	L	0011		X	01/eeee		11/nnnn	00/dddd
6	L	0100	X		01/eeee		11/nnnn	00/dddd
7	L	1101	X		01/eeee		11/nnnn	00/dddd
8	L	0011	X		01/eeee		11/nnnn	00/dddd

Figura 6.5: Exemplo de acessos a uma cache

O segredo do acesso à cache é o correto particionamento dos bits de endereços nos respectivos campos. Nosso exemplo acima é puramente hipotético, mas didático. Vamos agora analisar um exemplo real de cache mapeada diretamente. Suponha o processador MIPS que estamos trabalhando. Os endereços de memória são representados com 32 bits, bem como os dados que estamos trabalhando, que também são de 32 bits. Neste caso, uma memória cache mapeada diretamente que contenha 1024 palavras de 32 bits (capacidade = 1024 x 4 bytes = 4kbytes) utiliza os seguintes campos: os dois últimos bits (mais à direita) são usados para selecionar um dos 4 bytes que compõem o conjunto de 32 bits dos dados. Como existem 1024 posições na cache são necessários 10 bits para acessar cada uma das posições ($2^{10} = 1024$). Os 20 bits restantes fazem parte do campo *tag*. Vamos temporariamente desprezar os dois últimos bits de nosso endereço, no sentido de que eles serão mantidos iguais a 00_2 indefinidamente.

A Figura 6.6 mostra a divisão dos campos do endereço e o diagrama da memória cache mapeada diretamente. Observe que os endereços que nós temos utilizado em programação *assembly* são sempre múltiplos de 4 portanto os dois últimos bits são sempre iguais a 00_2. Mencionamos também que a capacidade de armazenamento desta memória cache em particular é de 4Kbytes. Nesta conta não estamos incluindo os bits que fazem parte do campo de *tag*. Eles estão presentes e não são desprezíveis, mas, por convenção, a capacidade de uma cache é medida apenas pela quantidade de bytes de informação que ela pode armazenar.

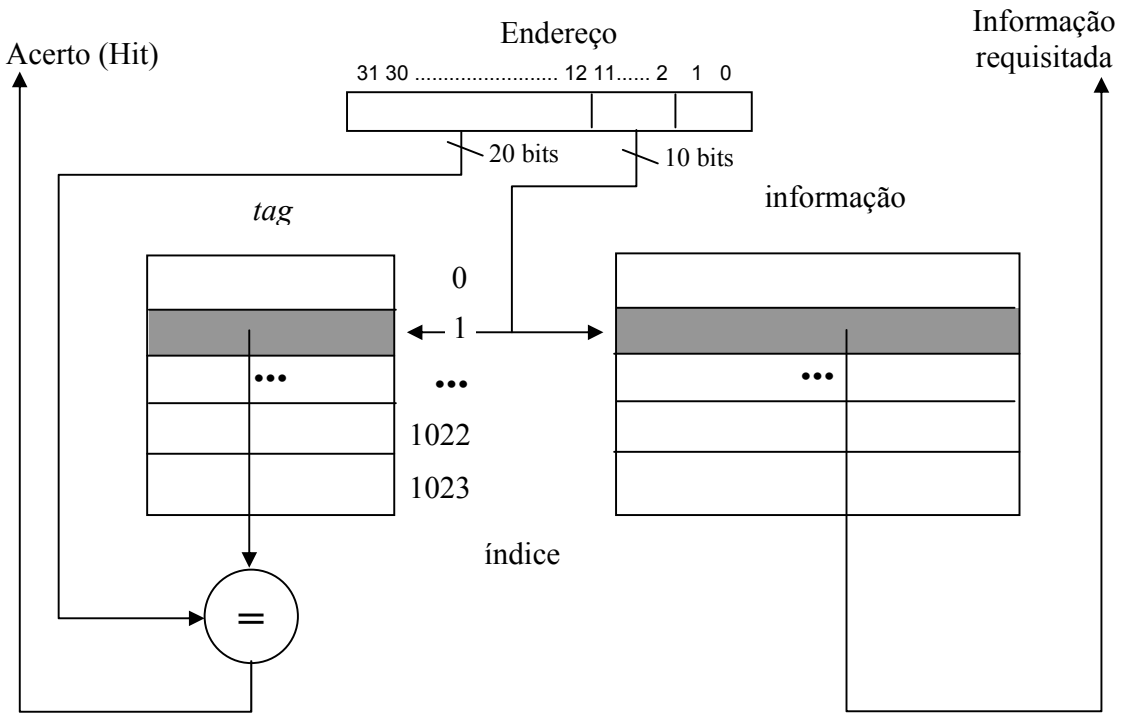


Figura 6.6: Memória cache para o MIPS

No exemplo da Figura 6.5 um pequeno detalhe foi omitido: como saber se uma informação é válida na cache. Naquele momento tratamos exclusivamente de dizer que a cache estava vazia em alguma de suas posições. De fato uma cache possui associado a cada linha um bit chamado de bit de validade (*v*). Ele sinaliza se uma informação na cache está coerente com o que existe na memória principal. A coerência na cache significa que a há uma cópia fiel da informação na memória principal armazenada na cache.

A presença deste bit de validade é muito importante na inicialização de uma cache. Inicialmente todas as informações na cache são consideradas inválidas, ou seja, os valores dos *tags* armazenados na cache não têm significância para o processamento das informações. Quando um bit de validade vale "1_2" significa que a informação que está na cache é coerente com o resto da hierarquia e quando este bit vale "0_2" a informação está incoerente. Todos os bits de validade são então inicializados com 0_2. À medida em que são transferidas informações da memória principal para cache, o bit de validade correspondente passa a valer 1_2.

A Figura 6.7 mostra a organização de uma cache mapeada diretamente contendo um bit de validade associado a cada endereço de cache. Agora, um

acerto na cache depende também do bit de validade estar setado ("1_2"). Assim, quando uma cache está vazia e, por conseguinte, todos os bits de validade ressetados ("0_2"), qualquer tentativa de transferência de dados da memória principal para cache irá resultar em uma falha. À medida que os dados são copiados da memória principal para a cache, o bit de validade vai sendo setado, o que possibilita um acerto na cache.

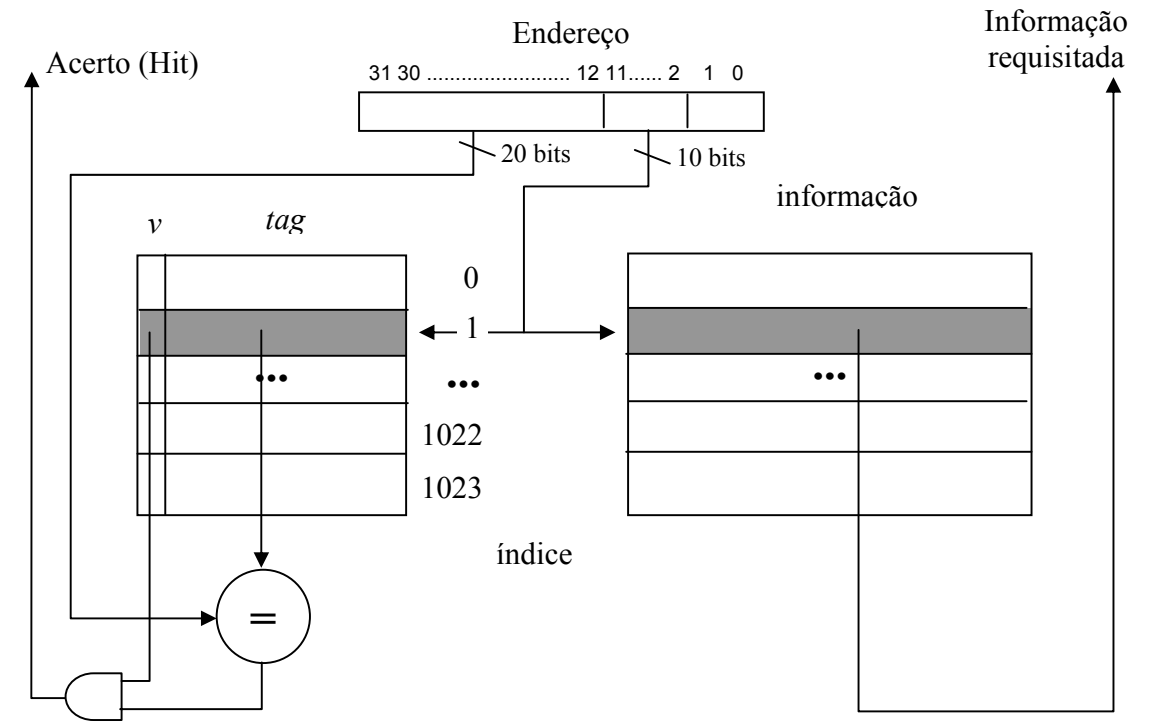


Figura 6.7: Memória cache para o MIPS com bit de validade

Vamos refazer o exemplo da Figura 6.5 agora contendo um bit de validade. A Figura 6.8 mostra a seqüência de acessos. Observe que inicialmente (Tempo = 0) todos os bits de validade estão ressetados e por isto não importa o que há em tag/dado. Quando o primeiro acesso é realizado o dado aaaa do endereço 0000_2 é trazido da memória principal para cache e por isto o dado no índice 00_2 da cache é uma cópia fiel do que existe na memória principal e, por conseguinte o bit de validade é setado.

Tempo	Leitura (L) Escrita (E)	Endereço (binário)	Hit	Miss	Cache (índice)			
					00	01	10	11
					v/tag/dado	*v/tag*/dado	*v/tag*/dado	*v/tag*/dado
0	-	-			0/--/----	0/--/----	0/--/----	0/--/----
1	L	0000		X	1/00/aaaa	0/--/----	0/--/----	0/--/----
2	L	0100		X	1/01/eeee	0/--/----	0/--/----	0/--/----
3	L	1110		X	1/01/eeee	0/--/----	1/11/nnnn	0/--/----
4	L	0100	X		1/01/eeee	0/--/----	1/11/nnnn	0/--/----
5	L	0011		X	1/01/eeee	0/--/----	1/11/nnnn	1/00/dddd
6	L	0100	X		1/01/eeee	0/--/----	1/11/nnnn	1/00/dddd
7	L	1101	X		1/01/eeee	0/--/----	1/11/nnnn	1/00/dddd
8	L	0011	X		1/01/eeee	0/--/----	1/11/nnnn	1/00/dddd

Figura 6.8: Exemplo de acessos a uma cache com bit de validade

Até o presente momento não tratamos a questão da escrita na cache. Perceba que todos os exemplos propostos apenas lidam com a leitura de informações. Para ler um dado o processador informa o endereço do mesmo, no barramento de endereços e emite um sinal de leitura de dados, no barramento de controle. O dado retirado do sistema de memórias é disponibilizado ao processador pelo barramento de dados. Quando o processador enseja escrever um dado na memória, ele disponibiliza o dado que deseja escrever no barramento de dados e envia um sinal de escrita para o sistema de memória, via barramento de controle, indicando uma escrita de dados.

O comportamento do sistema de memórias para uma escrita de dados é o seguinte: uma vez requisitada pelo processador a cache verifica se o endereço do dado a ser escrito está presente na cache. Se o dado está na cache (acerto) o gerenciador da cache precisa tomar uma decisão: escrever o novo dado na cache e na memória principal ou escrever o novo dado apenas na cache.

No primeiro caso o dado a ser escrito sempre manterá a consistência com o que existe na memória principal dado que a escrita se dá simultaneamente nas duas memórias. Esta política é chamada de ***write-through***. Uma desvantagem óbvia deste modelo é que todas as escritas precisam acessar a memória principal, minimizando assim o efeito benéfico (desempenho) promovido pela presença da cache. Se todos os acessos ao sistema de memória forem de escrita então a velocidade da hierarquia estará limitada pela velocidade da memória principal (desconsiderando os outros níveis da hierarquia).

Uma alternativa a este modelo é escrever o dado novo apenas na cache, na expectativa deste dado ser acessado novamente em breve. Esta política é conhecida como ***write-back*** ou ***copy-back***. Neste caso haverá uma inconsistência entre a informação na cache e na memória principal. O bit de

validade pode ajudar a resolver este problema. Ele seria ressetado sempre que houvesse uma escrita na cache. Numa cache *write-back*, os dados seriam escritos na memória principal apenas durante uma troca (substituição) de dados em uma determinada posição. Um dado inválido precisaria ser escrito na memória principal sempre que um novo dado fosse ocupar sua posição. Isto acabaria com os acessos à memória principal a cada acesso de escrita na cache, por outro lado o mecanismo de acerto e erro precisaria ser re-projetado, bem como o bit de validade sozinho não permitiria identificar se a posição da cache está vazia, ou se foi vítima de uma escrita.

Felizmente, em aplicações reais, o número de acessos à memória, para escrita de dados, é muito inferior que o número de acessos para leitura de dados, minimizando o impacto da adoção da política *write-through*, mais simples, mas que pode apresentar piores resultados em termos de desempenho.

No caso de uma tentativa de escrita na cache onde o dado não está na cache é possível escrevê-lo somente na memória principal, política ***write no-allocate***, ou escrevê-lo na memória principal e trazê-lo para cache, política ***write allocate***. Caches *write allocate* são normalmente associadas com a política *write-back*, enquanto *write no-allocate* é normalmente associada com a política *write-through*.

Vamos experimentar alterar o exemplo da Figura 6.8 usando agora uma cache com políticas *write no-allocate* e *write-through*. A Figura 6.9 mostra as alterações propostas. No instante Tempo = 2 o processador solicita uma escrita no endereço 0010_2 do dado zzzz. A cache informa uma falha. Como a política é *write no-allocate*, a informação é imediatamente repassada para memória principal. Nada é registrado na cache, mas a memória principal passa a aguardar o dado zzzz no endereço 0010_2. Perceba que no instante Tempo = 4 este dado é copiado da memória para a cache. No instante Tempo = 6 o processador escreve outro dado (yyyy) no mesmo endereço 0010_2. Neste momento existe um dado na cache que vale zzzz ocupando o endereço correspondente. Há um acerto na cache e como a política é de *write-through* o novo dado é escrito na cache e na memória principal ao mesmo tempo mantendo a coerência da cache.

Agora procure refazer este exemplo utilizando outras políticas, como exercício de fixação, e compare a quantidade de acertos e erros. Informe qual seria a melhor cache para esta seqüência de acessos.

Tempo	Leitura (L) Escrita (E) [dado]	Endereço (binário)	Hit	Miss	Cache (índice)			
					00	01	10	11
					v/tag/dado	*v/tag*/dado	*v/tag*/dado	*v/tag*/dado
0	-	-			0/--/----	0/--/----	0/--/----	0/--/----
1	L	0000		X	1/00/aaaa	0/--/----	0/--/----	0/--/----
2	E [zzzz]	0100		X	1/00/aaaa	0/--/----	0/--/----	0/--/----
3	L	1110		X	1/00/aaaa	0/--/----	1/11/nnnn	0/--/----
4	L	0100		X	1/01/zzzz	0/--/----	1/11/nnnn	0/--/----
5	L	0011		X	1/01/ zzzz	0/--/----	1/11/nnnn	1/00/dddd
6	E [yyyy]	0100	X		1/01/ yyyy	0/--/----	1/11/nnnn	1/00/dddd
7	L	1101	X		1/01/ yyyy	0/--/----	1/11/nnnn	1/00/dddd
8	L	0011	X		1/01/ yyyy	0/--/----	1/11/nnnn	1/00/dddd

Figura 6.9: Exemplo de acessos de leitura e escrita a uma cache

6.2.1 – Explorando a localidade espacial

As cache apresentadas até o momento não exploram a característica de localidades espacial de referências. Relembrando, a localidade espacial diz que é provável que um dado/instrução vizinha na memória a outro dado/instrução seja utilizado brevemente. Então, ao invés de trazer da memória principal apenas uma informação por vez, poderíamos trazer um conjunto de informações. A cache agora precisa ter espaço para alocar mais dados por linha. A Figura 6.10 mostra a organização de uma cache contendo 4 informações de 4 bytes em cada linha. Chamamos de **bloco** uma linha da cache. Cada bloco da cache proposta na figura possui 16 bytes. Para selecionar a informação que efetivamente será entregue ao processador utilizamos um MUX cujas linhas de seleção são retiradas do endereço solicitado. Para uma cache com 4 informações (também chamadas de palavras) por bloco precisamos de 2 bits para selecioná-las. A capacidade de armazenamento desta cache é de 1024x16 = 16Kbytes. Perceba que existe um único bit de validade para um bloco inteiro. Cada transferência da memória principal para cache aporta 4 palavras, bem como, no caso de uma política *write back*, um bloco inteiro é transferido da cache para memória principal no caso de uma reposição de algum bloco inválido.

Vamos fazer um exemplo simples para fixarmos as idéias sobre as caches com múltiplas palavras por bloco. Em uma cache mapeada diretamente com 64kbytes e 8bytes por bloco, quais os campos de endereço são usados para índice, *tag* e escolha da palavra do bloco? Considere a palavra sendo indivisível e de 32bits.

A resposta para esta questão é a seguinte: como temos blocos de 8 bytes, temos, de fato, duas palavras por bloco, então precisamos de apenas 1 bit para selecionar a palavra no bloco. Desprezando os bits 0 e 1 do endereço, o bit para selecionar a palavra será o bit de número 2. Como a memória é de

64kbytes e temos 8 bytes por bloco, existem 64x1024/8 blocos, ou seja, 8192 blocos. Portanto são necessários 13 bits para selecionar o bloco, pois $\log_2 8192 = 13$. Estes bits serão então de 3 a 15 no endereço. Tudo que resta vai para a comparação dos *tags*. Os bits de 16 a 31 formam o campo da *tag*.

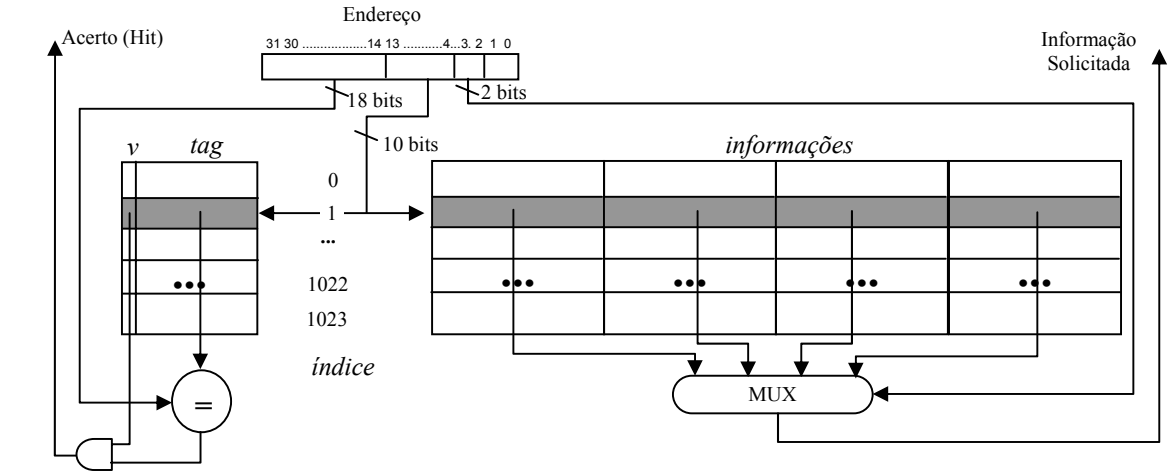


Figura 6.10: Memória cache para o MIPS com blocos de 16 bytes

Vamos ver um exemplo de como podemos localizar uma informação em uma cache mapeada diretamente. Para isto vamos analisar a seguinte configuração: capacidade de 128bytes e 4 palavras/linha. Neste caso, como existem 16 bytes por linha e a capacidade é de 128bytes, então existem 128/16=8 linhas. Para endereçar 8 linhas é preciso utilizar 3 bits e para endereçar 4 palavras é preciso utilizar 2 bits, portando o endereço enviado ao processador será dividido em 25 bits de tag, 3 de índice de linha e 2 de seleção da palavra. Os dois bits menos significativos serão descartados.

A Figura 6.11 mostra como um endereço $00400abc_h$ será enxergado pela cache. Os bits 3 e 2 formam o seletor de palavras, como valor 11_2. Os bits 6, 5 e 4 formam o índice da linha, com valor 011_2. Os bits restantes, 31 a 7, formam a *tag*, cujo valor será $00000000010000000000010101_2$ (0008015_h). Um fato importante para o mapeamento seguir um padrão é que um endereço como este, quando provoca uma falha na cache, faz com que os seus três antecessores na memória principal sejam trazidos para a cache. Se o endereço fosse $00400ab8_h$, os dois antecessores e o sucessor seriam trazidos juntos para a cache.

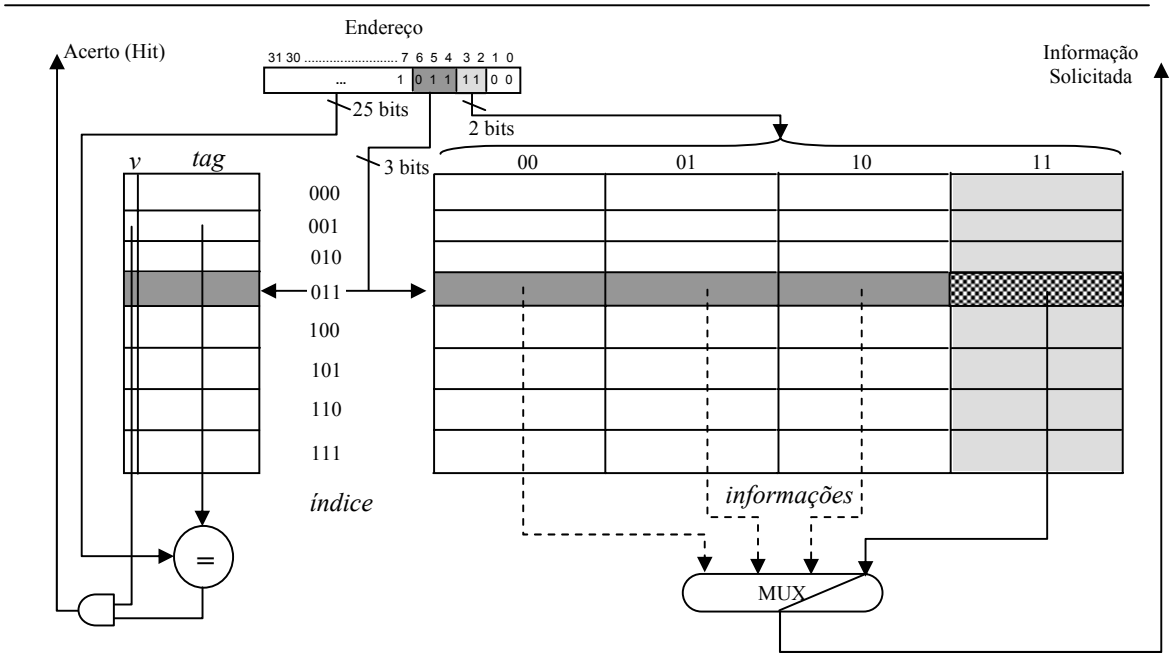


Figura 6.11: Localização de uma informação na cache

6.2.2 – Memórias cache associativas por conjuntos

Nestas cache que apresentamos até o presente momento existe um limite para exploração da localidade temporal. Isto porque cada endereço na memória principal ocupa um lugar pré-determinado na cache. Uma possibilidade mais amena seria permitir que uma posição da memória principal pudesse ocupar uma dentre *m* posições na cache. As cache associativas por conjunto (*set associative*) replicam seus campos de informações e *tags* e permitem que uma informação da memória principal possa ocupar mais de uma posição.

Um esquema de uma cache associativa por conjunto pode ser vista na Figura 6.12. Esta cache permite que cada informação da memória principal possa ficar entre o primeiro plano (*way* 0) ou o segundo plano (*way* 1) de dados da cache. Uma cache deste tipo é conhecida como cache associativa por conjuntos de 2 caminhos (*2-way set associative*). Isto significa que um determinado dado pode ocupar 1 entre 2 possíveis lugares. Uma cache *4-way set associative* permite que um dado ocupe 1 entre 4 possíveis posições. Na Figura podemos observar uma réplica dos circuitos que são usados para comparação, acerto (*hit*) e escolha da palavra do bloco. Um acerto na cache agora pode acontecer em qualquer dos planos. Isto implica que o acerto pode

vir do plano 0 ou do plano 1. É ainda preciso selecionar dentre as palavras disponíveis na saída dos *muxes* qual deve ser entregue ao processador. Para isto é usado um codificador $2^n \rightarrow n$, onde o número de planos vale $\log_2 n$. Os *hits* de cada plano são ligados ao codificador e dele sai um sinal de um novo *mux* para selecionar a palavra correta a ser entregue ao processador.

A cache mostrada na Figura 6.12 possui 1024 conjuntos, que são apontados pelo campo índice do endereço. Tal cache tem capacidade de 32Kbytes.

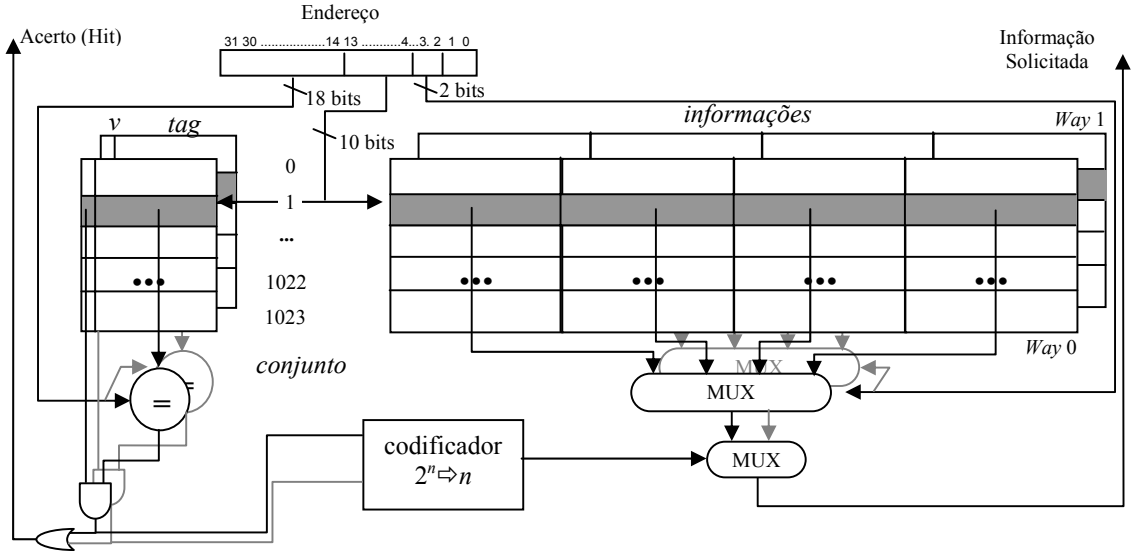


Figura 6.12: Memória cache associativa por conjuntos do MIPS

Uma cache mapeada diretamente pode ser vista como um caso particular de uma cache associativa por conjunto, a saber, uma cache *1-way set associative*.

Em caches associativas por conjunto surge uma questão muito particular. Uma vez preenchido um conjunto, quando for necessário substituir um bloco da cache, qual dos componentes do conjunto deve ceder lugar para o novo dado? O esquema mais comum de política de troca de dados na cache é a Menos Recentemente Usada – LRU (*Least Recently Used*). Nesta política, o bloco a ser desprezado será aquele que está sem uso há mais tempo. Isto requer um controle temporal do uso de cada bloco. Por exemplo, em nossa cache da Figura 6.13 supondo que o conjunto 1 do plano 0 foi usado no momento Tempo = 2 e que o conjunto 1 do plano 0 foi preenchido no instante Tempo = 5, e agora em Tempo = 9 houve uma falha na cache e um novo bloco deve ser colocado no conjunto 1. Qual dos dois conjuntos anteriores deve ser

expurgado da cache? Pela política LRU, o bloco pertencente ao plano 0 deve ser ceder seu lugar, pois ele foi utilizado há mais tempo.

Vamos ver um exemplo de preenchimento de uma cache associativa por conjuntos com a seguinte configuração: Capacidade: 512bytes, 32 Conjuntos e 2 palavras/bloco. Palavras de 32 bits. O trecho da memória principal com os respectivos dados está mostrado na Figura 6.13 (a). A seqüência de acessos é a seguinte: 40000028_h, 40000000_h, 40000004_h, $4000000c_h$, 40000008_h, 40000428_h, 40000128_h, $4000012c_h$, 40000128_h, $4000012c_h$, 40000028_h. Todos os acessos são de leitura de dados.

Iniciamos desenhando o esquema da cache. Ora, possuindo a cache 32 conjuntos e 2 palavras de 32 bits (4 bytes) por bloco, então cada plano contém 32x2x4 bytes = 256 bytes. Como a cache tem capacidade para 512bytes então existem 2 planos (*way* 0 e *way* 1). A Figura 6.13(b) mostra a organização da cache. Os planos foram colocados lado a lado para efeito de visualização dos dados.

Vamos agora determinar os campos que compõem o endereço, como visto pela cache. Os bits 0 e 1 são desconsiderados pois sempre fazemos acesso a palavras de 32 bits. O bit 2 serve para selecionar a palavra dentro do bloco que será enviada ao processador. Para selecionar os conjuntos precisamos de 5 bits, pois $2^5 = 32$. Os bits de 3 a 7 são então escolhidos. Os bits de 8 a 31 representam o campo de *tag*.

O primeiro acesso ocorre então ao endereço 40000028_h. Este valor em binário vale 0100 0000 0000 0000 0000 0000 **0010 1**$\underline{0}$$00_2$. Os bits 0 e 1 são desprezados. O bit 2 indica que a palavra a ser selecionada é a 0 e os bits de 3 a 7 formam 00101_2 o que indica que o conjunto a ser acessado é o 5. Os bits em negrito no endereço em binário representam o campo do conjunto. O bit sublinhado indica o campo da escolha da palavra. Ora, neste conjunto, o bit de validade de ambos os planos vale 0, portanto ambos indicam uma falha na cache. Assim sendo, é preciso fazer um acesso à memória principal e recuperar o dado.

Este acesso à memória principal deve trazer duas palavras adjacentes. Uma delas obrigatoriamente será a palavra que está no endereço 40000028_h, mas a segunda palavra a ser copiada da memória pode ser a sua antecessora (endereço 40000024_h) ou sua sucessora (endereço $4000002c_h$). Para descobrir isto observamos onde a informação requerida vai ser armazenada. Ora, o bit de escolha de palavra vale 0, portanto a informação vai ser armazenada na palavra 0. Então o campo que nos resta preencher é a palavra 1 que é a sucessora da palavra 0 na memória.

(a)

40000000_h	$0a00107a_h$
40000004_h	10202398_h
40000008_h	00000000_h
$4000000c_h$	$ffffffff_h$
...	....
40000020_h	$78ffffff_h$
40000024_h	$3affff80_h$
40000028_h	$8763482f_h$
$4000002c_h$	12650298_h
...	....
40000120_h	$6abcdef7_h$
40000124_h	$020013ee_h$
40000128_h	$5465387e_h$
$4000012c_h$	$ffeabc96_h$
...	....
40000420_h	98273457_h
40000424_h	$3487671a_h$
40000428_h	$46462ffe_h$
$4000042c_h$	$6453f0ff_h$

(b)

Way 0				*Way* 1			
v	*tag*	*informações*		*v*	*tag*	*informações*	
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			
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			

Figura 6.13: Exemplo de acessos a uma cache *set associative*

Uma regra prática utilizada pelo sistema de memórias é zerar o(s) bit(s) de seleção de palavra e solicitar a memória principal, a partir deste endereço modificado, tantas palavras quantas forem necessárias para preencher o bloco da cache.

Finalmente temos de escolher em qual dos planos o bloco será armazenado. Neste instante a escolha é indiferente, pois ambos os blocos que compõem o conjunto estão vazios. Vamos escolher ao acaso o plano 0. Os dados dos endereços 40000028_h e $4000002c_h$ são copiados para cache. A Figura 6.14 mostra como fica a cache depois de copiados os dados da memória principal.

Observe que o *tag* armazenado pertence ao bloco inteiro.

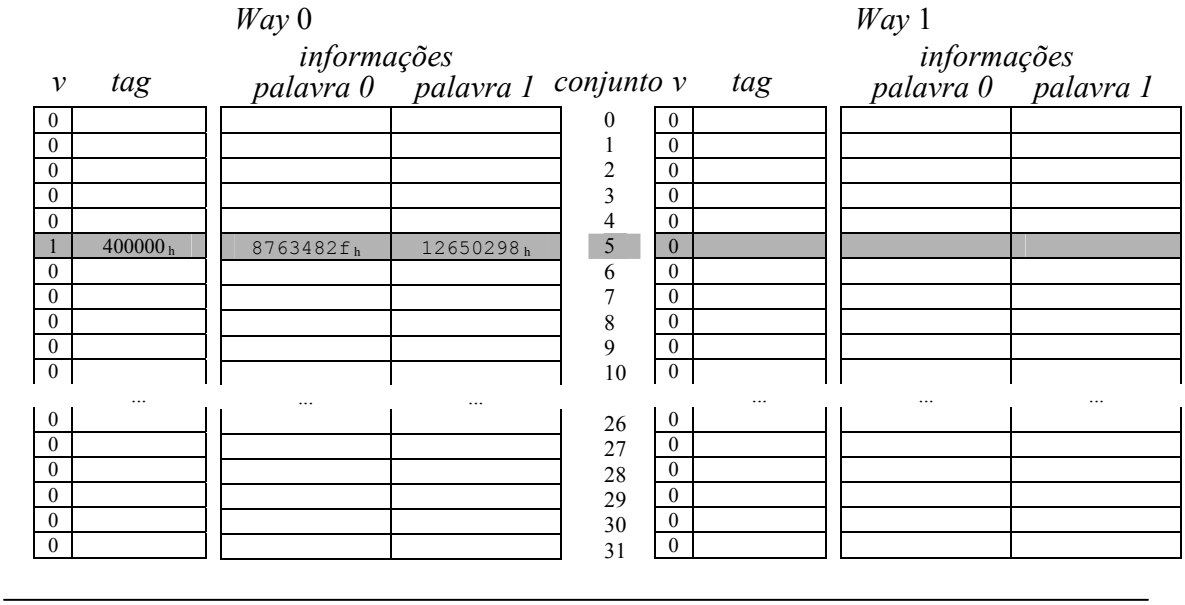


Figura 6.14: Exemplo de acessos a uma cache *set associative*

O segundo acesso ocorre à palavra no endereço 40000000_h. Extraindo os campos vamos perceber que a palavra escolhida é a 0 e o conjunto onde ela reside é também o conjunto 0. Ora, no conjunto 0 ambos os blocos são inválidos, então acontece uma nova falha. Como no passo anterior os dados dos endereços 40000000_h e 40000004_h serão copiados para cache. A Figura 6.15 mostra o resultado da cópia dos dados na cache.

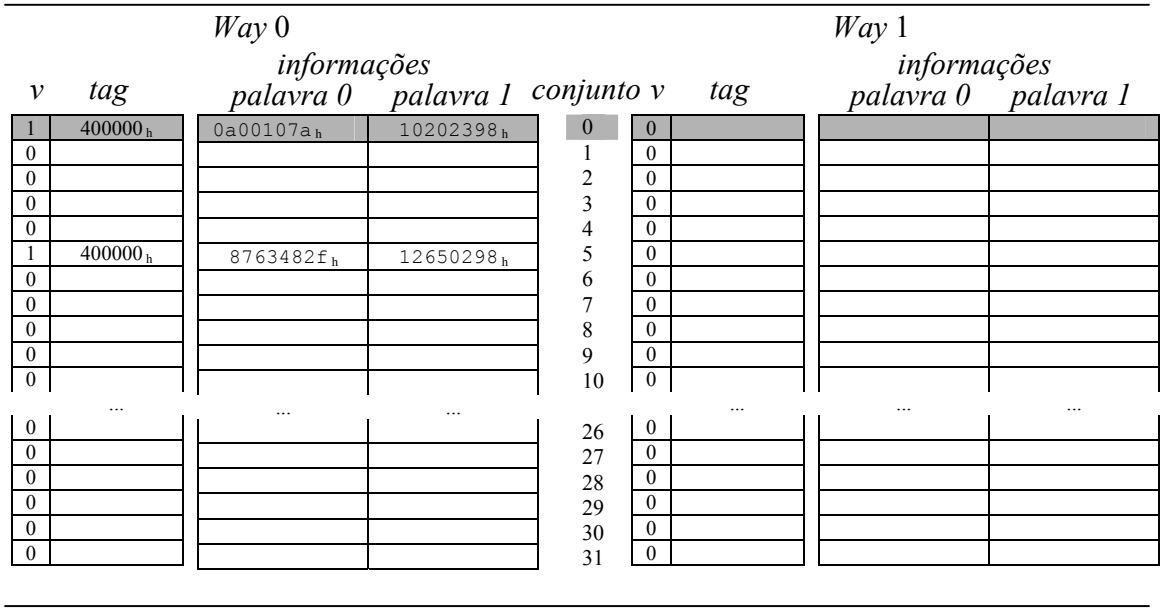


Figura 6.15: Exemplo de acessos a uma cache *set associative*

O terceiro acesso é feito ao endereço 40000004_h. Extraindo os campos do endereço temos: palavra selecionada = 1, conjunto = 0 e *tag* = 400000_h. Ora, no conjunto 0 existe um bloco válido cuja *tag* vale exatamente 400000_h. Assim sendo, se caracteriza um acerto na cache. A palavra 1 do plano 0 é então entregue ao processador, conforme mostrada na Figura 6.15.

O quarto acesso é feito à palavra de endereço $4000000c_h$. Neste caso, os campos são os seguintes: palavra selecionada = 1, conjunto = 1 e *tag* = 400000_h. O conjunto 1 está ainda vazio, em qualquer que seja os planos, então uma falha é assinalada. Os dados a serem retirados da memória são os de endereços 40000008_h e $4000000c_h$. Observe que o endereço requerido corresponde à palavra 1 na cache e portanto a cache é preenchida com seu antecessor e não o seu sucessor. A Figura 6.16(a) mostra como fica a cache depois deste acesso.

O quinto acesso é feito à palavra de endereço 40000008_h. Os campos extraídos de endereço são: palavra selecionada = 0, conjunto = 1 e tag = 400000_h. Neste instante existe no plano 0, conjunto 1 uma tag de valor 400000_h, portanto um acerto é sinalizado e a palavra 0 deste plano/conjunto é disponibilizada ao processador.

O sexto acesso é feito ao endereço 40000428_h. Os campos associados são: palavra selecionada = 0, conjunto = 5 e *tag* = 400004_h. Ora, no conjunto 5 do plano 0 existe um bloco válido, mas a *tag* nele armazenada vale 400000_h. Portanto um acerto não é caracterizado. Veja, entretanto, que este conjunto tem uma posição vazia no plano 1 e portanto, apesar de ser assinalada uma

falha na cache, não precisamos destruir o dado anteriormente armazenado. Simplesmente colocamos os novos dados no plano 1. A Figura 6.16(b) mostra a cache depois deste acesso.

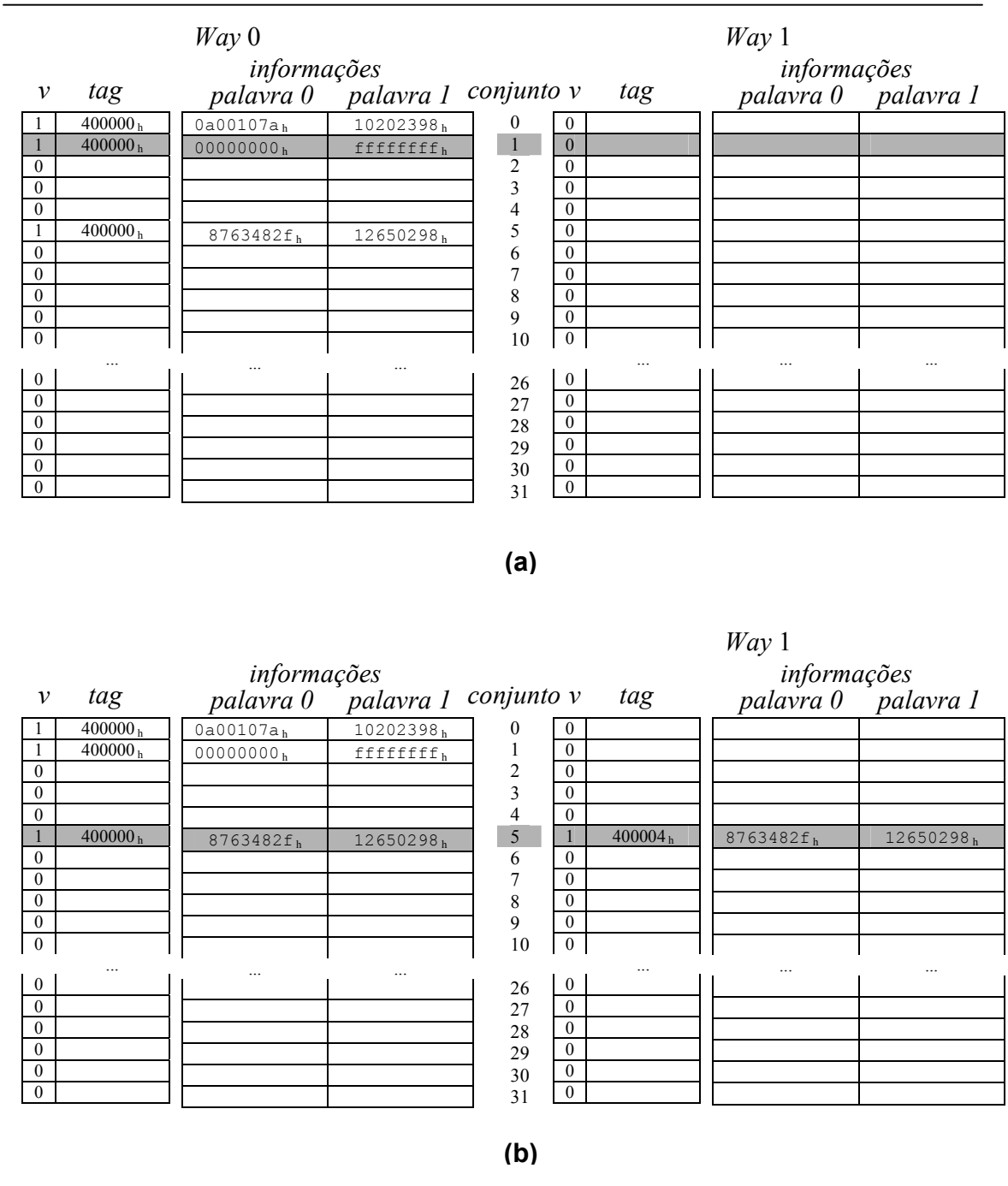


Figura 6.16: Exemplo de acessos a uma cache *set associative*

Chegamos ao sétimo acesso, onde a palavra solicitada está no endereço 40000128_h. Os campos extraídos deste endereço são: palavra selecionada = 0, conjunto = 5 e *tag* = 400001_h. Neste instante o conjunto 5 contém dois blocos válidos, entretanto, nenhum deles contém a *tag* 400001_h. Isto caracteriza uma falha na cache. Agora já não há espaços vazios para que um novo bloco possa ser copiado para cache sem sacrifícios de alguns dados. Precisamos, então, selecionar qual bloco será sacrificado. Aqui entra a política LRU. Ora o bloco que foi utilizado há mais tempo foi o bloco contido no plano 0. Então, este será escolhido para ceder seu lugar ao novo bloco. A Figura 6.17 mostra o novo bloco copiado da memória.

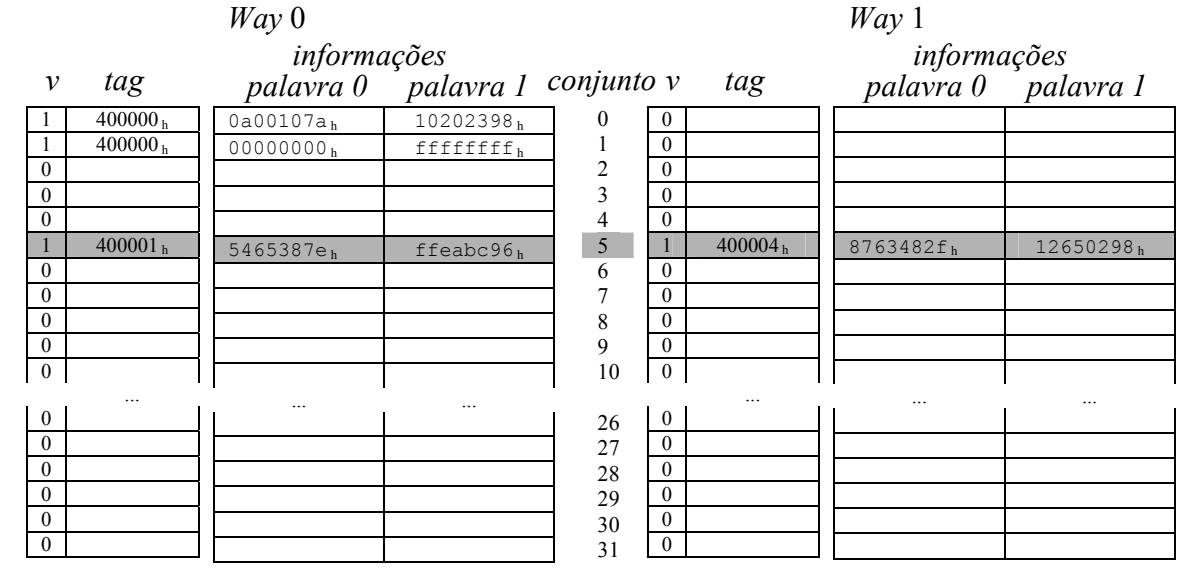


Figura 6.17: Exemplo de acessos a uma cache *set associative*

O oitavo acesso ocorre à palavra de endereço $4000012c_h$. Esta palavra faz parte do novo bloco recém-chegado à cache e, sem mais delongas, podemos informar o acerto na cache. O mesmo ocorre para o nono e décimo acesso.

Finalmente ocorre o décimo primeiro acesso. O endereço requisitado é 40000028_h. Neste caso os campos extraídos são: palavra selecionada = 0, conjunto = 5 e *tag* = 400000_h. No conjunto 5 não existe um bloco válido em qualquer de seus planos que contenha tal *tag*. Assim sendo uma falha é consumada. Mais uma vez é preciso escolher qual bloco pertencente a este conjunto será sacrificado. Ora o bloco do plano 1 foi o menos recentemente utilizado e por isto, utilizando a política LRU, ele deve ser descartado.

A Figura 6.18 mostra o conteúdo da cache após esta seqüência de acessos. Foram computados 5 acertos em 11 acessos, uma taxa de acerto de 45%, que é um valor muito baixo. Vale salientar, entretanto, que os primeiros acessos à cache costumam promover uma grande quantidade de falhas. À medida que os programas vão sendo executados, esta taxa de acerto tende a aumentar substancialmente.

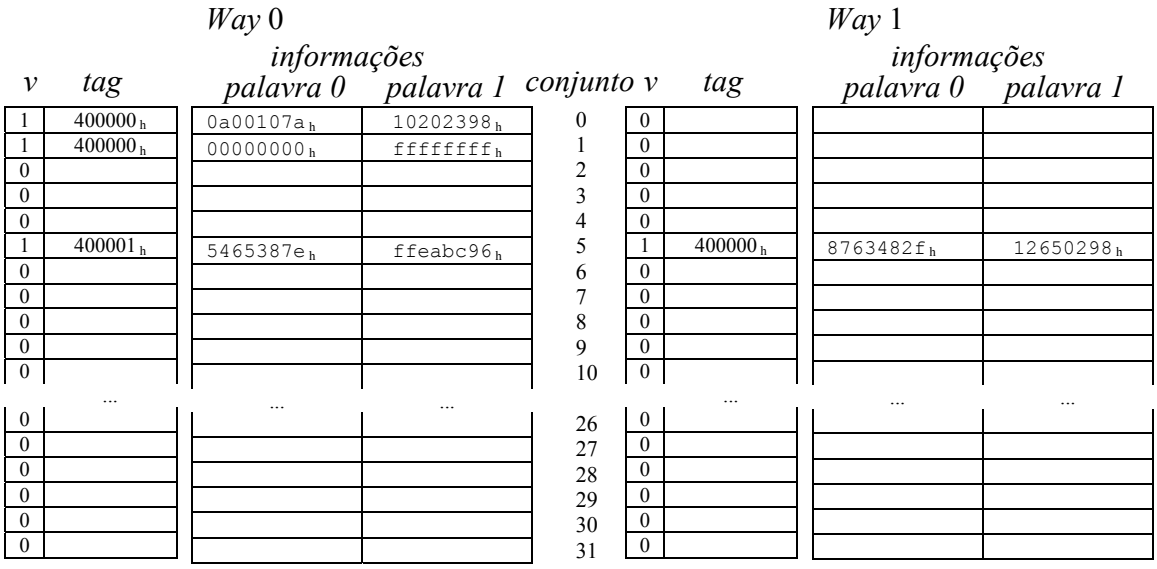


Figura 6.18: Exemplo de acessos a uma cache *set associative*

Agora que finalizamos nosso exemplo podemos lucubrar sobre alternativas de organização de cache. Por exemplo, uma cache que tenha apenas um conjunto de blocos. Neste caso o número de planos será o número de blocos da cache e qualquer dado pode ocupar qualquer posição na cache. Esta cache é conhecida como **totalmente associativa** (***fully-associative***). A maior dificuldade para implementar caches *fully-associatives* é o gerenciamento da política de substituição de cache. Quanto maior o número de planos, mais complexo fica determinar qual dos blocos deve ser substituído, a menos que nenhuma política seja implementada.

Para aumentar o desempenho de um sistema, usando caches é preciso projetá-las para a melhor situação possível. Veja que alguns programas podem apresentar um comportamento não convencional, com baixa localidade temporal e/ou espacial e isto compromete o desempenho geral do sistema.

Uma técnica normalmente utilizada para melhorar o desempenho de caches é utilizar duas caches em paralelo. Uma para dados e outra para códigos (programas). Isto permite que dois acessos possam ser feitos simultaneamente ao sistema de memórias. Outra abordagem, empregada largamente na indústria é um subsistema de caches, com vários níveis de memórias cache. Convencionalmente existem dois níveis L1 e L2, O primeiro nível está localizado dentro do próprio chip do processador e o segundo nível fica normalmente na placa mãe do sistema. A efetividade de uma cascata de caches muito longa, com 8, 16 níveis é bastante questionável, mas entende-se que entre 2 e 4 níveis as melhorias de desempenho podem valer o custo.

Agora que entendemos os princípios que regem a construção de memórias cache, vamos visitar um outro nível da hierarquia de memórias: a memória principal. Neste nível são implementados conceitos muito conhecidos como memória virtual, paginação e segmentação de memória. Então, à parte dos aspectos físicos da memória principal, vamos conhecer e aprender a explorar estes conceitos.

6.3 – Memória Principal

Já mencionamos que a memória principal é implementada em sua maior parte por módulos voláteis de memória. Isto significa que ao ligarmos um computador esta memória estará praticamente vazia. Nossos programas estão armazenados em memórias magnéticas, tipicamente no disco rígido do computador. A máquina não sabe quais programas serão utilizados dentro da miríade de possibilidades e *downloads* que se apinham em nossa memória de massa. Para um programa ser executado ele é trazido, pelo sistema operacional, para memória principal e então é iniciado.

A memória principal também tem um limite físico de módulos que podem ser utilizados na placa mãe. Uma arquitetura típica de 32 bits de endereçamento é capaz de endereçar até 2^{32} = 4Gbytes de informações, mas nem sempre, ou quase nunca, esta quantidade de memória é permitida em uma placa mãe convencional. Memórias principais na ordem de 1Gbytes hoje já são realidade, mas ainda estamos em um patamar típico de 256Mbytes. Apesar destes números serem uma realidade hoje eles podem mesmo ser motivo de chacota em breve, mas o fato de existir um limite físico não está próximo de seu fim. Bem, o programador tende a se abster destas restrições e fazer seus programas sem se preocupar como, nem em que lugar da memória ele será alocado. Vamos chamar de processo o programa que o usuário executa.

Em um sistema multi-processos (com multiprogramação) cria-se uma certa ilusão de que o processador é capaz de executar diversos processos por

vez, embora, o tempo do processador seja, de fato, compartilhado entre estes processos. À parte desta discussão, o que importa é que o usuário possui um espaço de memória em que diversos processos devem estar para serem executados. A memória principal tem uma região pré-determinada que é reservada para o Sistema Operacional e por isto o usuário precisa conviver com um espaço ainda menor da memória principal para seus processos (Figura 6.19(a)).

Agora vamos mostrar uma limitação típica que um sistema multi-processos está sujeito. Imagine um sistema com uma memória principal de 1Mbytes, apenas para efeito didático. Destes, 128kbytes são reservados para o sistema operacional. A Figura 6.19(a) mostra este espaço. O usuário então decide usar o processo A que é então carregado na memória principal. O processo A ocupa 350Kbytes e vai ser colocado no primeiro espaço livre de memória (Figura 6.19(b)). O mesmo ocorre com o processo B de 200kbytes e C de 256Kbytes (Figura 6.19(c)). Neste ponto ainda restam 90kbytes livres, mas o usuário resolve utilizar um outro processo (D) de 128Kbytes. O Sistema Operacional precisa abrir espaço na memória para este novo processo. Ele decide então expurgar o processo A da memória. O processo D é alocado no espaço de A (Figura 6.19(d)). Um processo E, de apenas 100kbytes, também chamado pelo usuário é alocado (Figura 6.19(e)). O usuário então resolve utilizar novamente o processo A. Neste instante o processo A não está mais na memória principal e então ele precisa ser recuperado. O Sistema Operacional precisa mais uma vez decidir quem retirar da memória. Ele decide retirar então os processos B e C que foram utilizados menos recentemente. O processo A é novamente alocado na memória (Figura 6.19(f)).

Quando um processo chamado pelo usuário é posto na memória e começa a ser executado ele guarda algumas informações, dados entrados pelo usuário, então para retirar um processo da memória, sem terminá-lo, é preciso guardá-lo em algum lugar. Um espaço no disco é reservado para guardar este processo. Chamamos de ***swap* de disco** o fato de retirarmos da memória um processo que estava sendo executado e colocarmos de volta um outro processo, que estava suspenso no disco, para a memória. Veja que neste modelo os tamanhos dos processos interferem enormemente no desempenho do sistema, bem como a 'inteligência' do Sistema Operacional em alocar os espaços vazios. Veja também que o processo A começou a ser executado em uma região da memória, mas devido ao *swap* de disco ele foi parar em outra região da memória. Isto significa que os processos não têm endereços fixos para serem executados. Finalmente, percebemos que um processo grande, maior que a memória presente no sistema, nunca poderia ser executado, pois não haveria como alocá-lo.

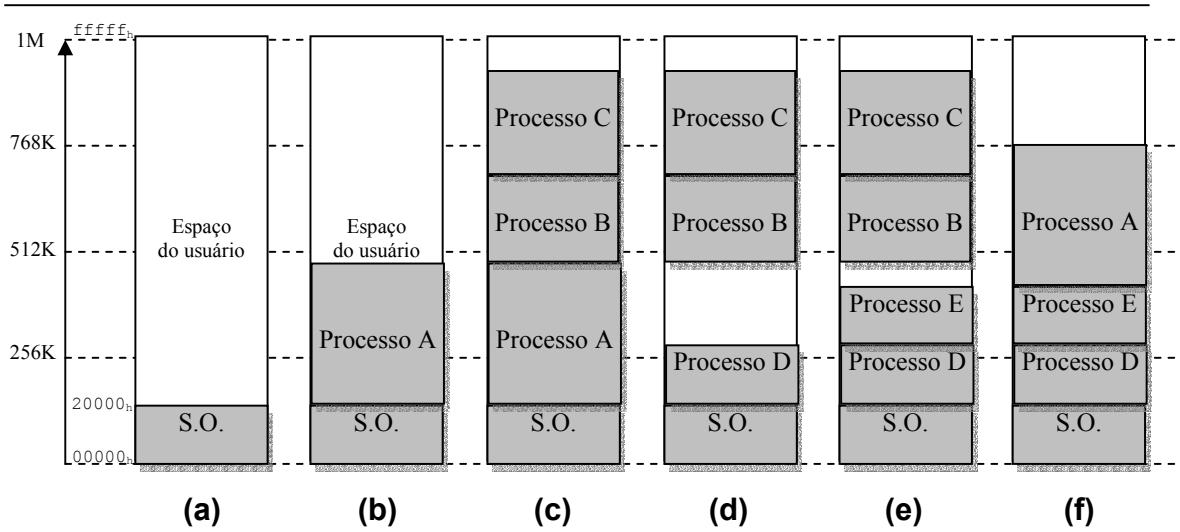


Figura 6.19: Alocação de memória

Todas estas limitações apontadas levaram a um conceito de paginação de código e de memória. Cada processo é então dividido em partes, de tamanhos fixos, chamadas de **páginas**. Estas partes, em geral, são pequenas e são alocadas em regiões da memória disponíveis chamadas **quadros** (*frames*). Os *frames* são freqüentemente chamados também de páginas. Um processo A pode agora ser dividido em diversas páginas e estas não necessitam ocupar *frames* consecutivos. O sistema operacional precisa manter uma tabela dizendo a quem pertence (a qual processo) cada *frame* e quais são os *frames* livres no sistema. Também é preciso saber onde foi alocado cada página do código.

Quanto aos endereços utilizados, já vimos que não podemos garantir que um processo vai ser executado em uma determinada região de memória, então, ao programarmos utilizamos endereços ditos **lógicos** (ou **virtuais**), mas quando um processo é carregado para memória, todos os saltos precisam ser verificados e os campos de deslocamentos das instruções precisam ser conferidos e possivelmente acertados (este procedimento é chamado de **relocação** ou **realocação**). Os endereços onde as instruções são efetivamente executadas são chamados de **endereços físicos**.

Um esquema parecido com o que ocorre com a cache é usado para particionar o endereço lógico e encontrar o endereço físico correspondente. A Figura 6.20 mostra o esquema convencional de tradução de endereços. Veja que partes dos bits do endereço virtual são usadas para indicar o deslocamento dentro de uma página e a porção restante é usada para indicar um número de

página virtual. Cada página virtual é alocada em uma página física (*frame*) e ela pode estar em qualquer lugar na memória principal. No modelo da figura, por exemplo, usamos 8 bits para o deslocamento dentro de uma página, portanto as páginas têm tamanhos de 2^8=256 bytes. Vemos também que o endereço físico tem apenas 20 bits, ou seja, a memória física disponível para uso é de apenas 2^{20}=1Mbytes. Portanto na memória inteira temos 4096 *frames* de 256 bytes cada.

Para encontrar um endereço físico correspondente a um endereço virtual uma tabela de páginas é utilizada. Esta tabela é indexada por uma parte do endereço virtual que é o Número da Página Virtual. Para cada número de página virtual possível uma entrada na tabela de páginas é utilizada. No exemplo da Figura 6.20, cada entrada guarda o início do endereço físico correspondente (os 12bits mais significativos). Um endereço virtual 00000104_h seria traduzido então como $45a04_h$.

Os processos têm a ilusão de dispor de 4Gbytes, mas de fato eles são executados em apenas 1Mbyte de memória real. Hoje em dia usamos páginas entre 4kbytes e 64kbytes. O tamanho da página é um dos principais fatores para definir o desempenho do sistema: páginas grandes requerem tempos grandes para sua carga na memória principal, por outro lado, mais informações por página implicam em menos necessidades de acesso ao disco.

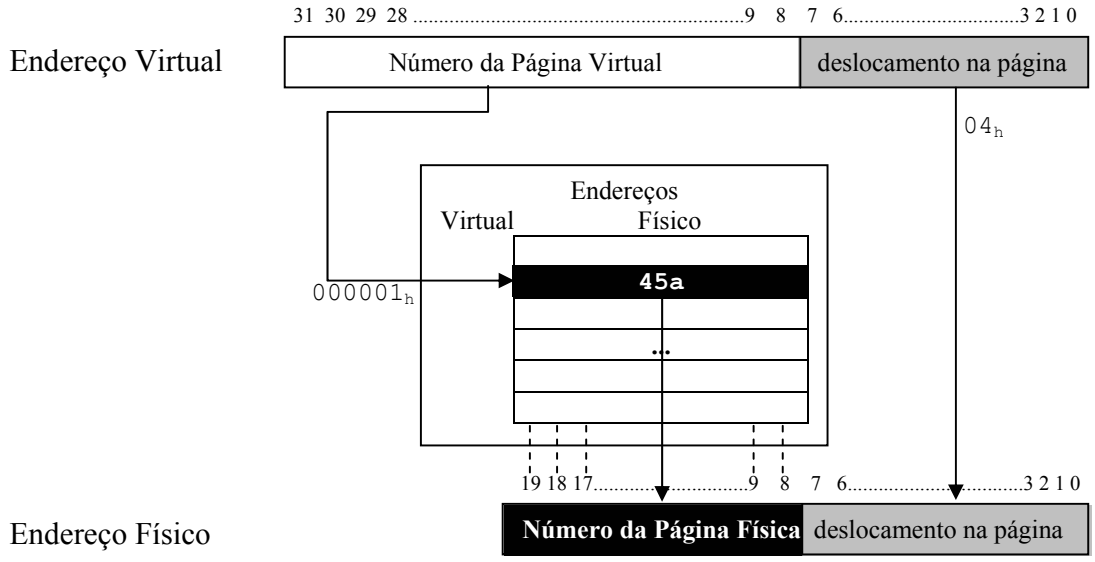


Figura 6.20: Mapeamento de endereços virtuais em físicos

6.3.1 – Memória Virtual

O principal conceito desta seção é a **memória virtual**. Ele pode ser enxergado como uma extensão do conceito de paginação. Um detalhe que deixamos para mencionar até então é que um processo que ocupa um espaço maior que a memória física disponível pode vir a ser executado. Segundo o princípio da localidade, os programas passam a maior parte de seu tempo executando um pequeno trecho do código. Então, de fato, não seria necessário carregar todas as páginas para execução de um processo, mas apenas uma porção delas, na esperança de serem as que mais freqüentemente são utilizadas. Uma memória organizada desta forma, paginada e com apenas um conjunto de páginas de um processo carregado por vez, é chamada de **memória virtual**.

Quando um endereço virtual é solicitado ao sistema de memória e este não tem um correspondente na memória principal, dizemos que ocorreu uma **falta de página** (***page fault***). Uma nova página deve ser então carregada para a memória principal.

Quando ocorre um falta de página, o controle passa para o Sistema Operacional (normalmente via interrupção) que deve então buscar a página correspondente no disco rígido e decidir onde alocá-la na memória física. Um endereço virtual não é suficiente para indicar onde está uma página no disco. Então, quando um processo é inicializado, uma região especial no disco, ***page file***, é criada para comportar todas as páginas do processo. Uma estrutura de dados é necessária para localizar uma página corretamente neste espaço. Esta estrutura de dados pode fazer parte da tabela de tradução de endereços (tabela de páginas) ou pode ser uma estrutura auxiliar.

Uma tabela de endereços compartilhada precisa de um sinalizador que indique se o endereço presente é um endereço físico ou um 'endereço de disco'. À medida que algumas páginas são carregadas para memória principal, elas vão ganhando endereços físicos associados aos endereços virtuais. As páginas que ficam no disco devem ter associadas a ela um endereço (no disco) que permita ao sistema operacional realizar um *swap*. A Figura 6.21 mostra um exemplo de tabela compartilhada. Para identificar se o endereço presente na tabela é um endereço de disco ou físico é usado um bit de validade. Quando o bit de validade for 0_2 a página está no disco e quando for 1_2 a página está na memória principal. Inicialmente todos os bits de validade são ressetados e à medida que as páginas associadas vão sendo alocadas na memória, eles vão sendo setados.

A tabela de páginas é construída em um espaço reservado da memória. Cada processo iniciado pelo usuário tem sua própria tabela de páginas. Estas

tabelas podem ocupar um espaço considerável na memória, por isto muitas técnicas são utilizadas para minimizar o problema. Tratar aqui destas técnicas não está no escopo deste livro, mas certamente um bom texto de Sistemas Operacionais contém informações mais aprofundadas.

O conceito de memória virtual é de suma importância para o entendimento de Sistemas Operacionais, entretanto, nossa abordagem não vai entrar em detalhes de funcionamento de uma implementação em particular do conceito. Existem técnicas muito apuradas e algoritmos muito eficientes para lidar com falta de página, proteção de processos e tradução eficiente de endereços, mas não trataremos destas soluções aqui.

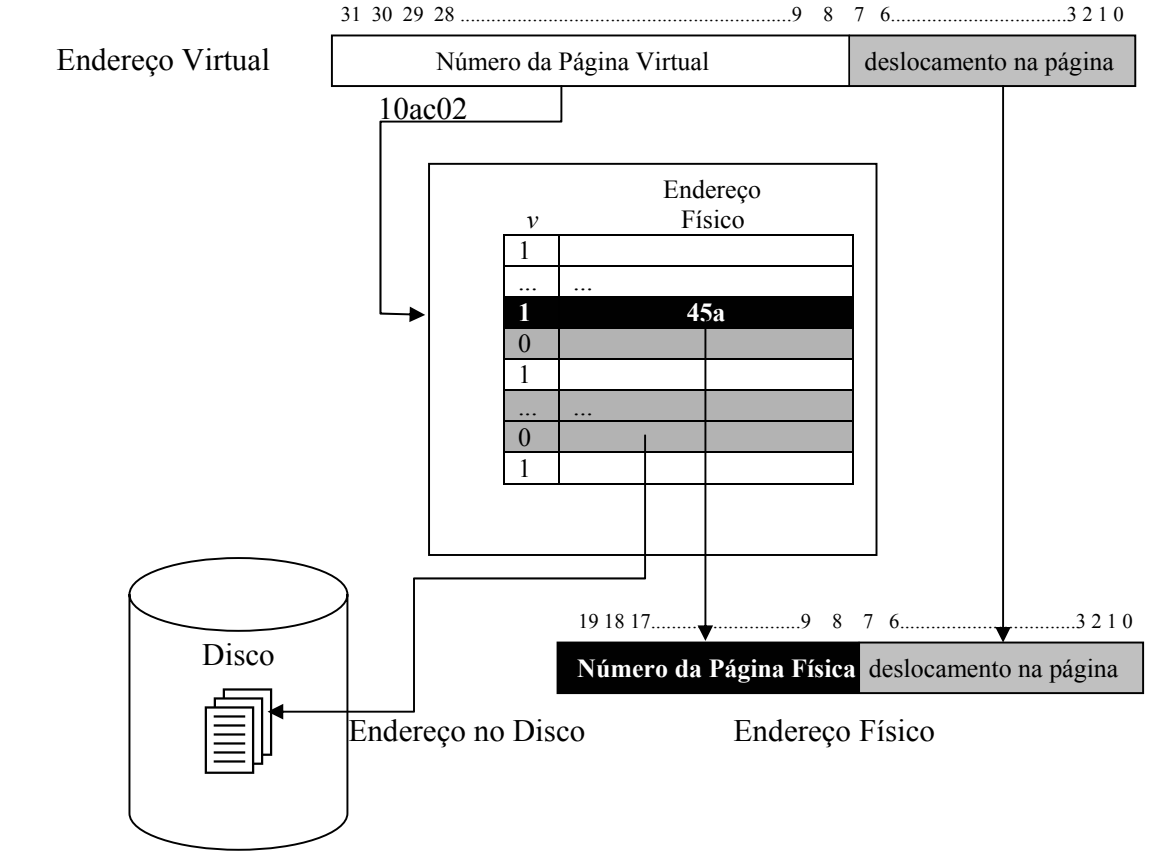


Figura 6.21: Mapeamento de endereços virtuais em físicos e de disco

6.4 – Memórias de massa

Antes da existência de discos de armazenamento magnético de dados, cartões perfurados ou fitas ferro-magnéticas eram as únicas formas de armazenamento durável de dados. Não precisa dizer que este tipo de

armazenamento levava a perdas de dados e à leitura seqüencial e lenta de cada informação desejada. Além disto, a degradação do material mediante as intempéries era evidente. Em 1956 a IBM lançou o RAMAC, um computador que usava um conjunto de discos magnéticos para armazenamento de dados. Nada muito portável cada disco ocupava quase um de diâmetro e o total do conjunto pesava quase uma tonelada, mas o que pesava mesmo era o preço: milhões de dólares.

Um disco rígido atual custa bem menos e guarda bem mais informações, entretanto a idéia básica se mantém. Um disco é basicamente um círculo de metal, vidro ou plástico recoberto com um material magnético capaz de orientar suas partículas e assim formar um campo magnético tal que quando lido pode ser interpretado como um dado binário.

Os dados em um disco magnético são armazenados de uma forma muito padronizada. Cada superfície do disco é dividida em circunferências concêntricas chamadas de **trilhas**. Por sua vez, cada trilha é dividida em arcos de tamanhos iguais chamados de **setores**. Em cada setor um conjunto de dados é armazenado. Junto com eles, alguns bytes de identificação. A Figura 6.22 mostra uma organização típica de um disco magnético. Existe um espaço entre trilhas e um espaço entre os setores. Dentro de um setor temos dois campos principais: um cabeçalho, contendo a identificação do setor (número da trilha, número da cabeça de leitura, número do setor) e um campo de dados contendo efetivamente os dados. Os tamanhos indicados em bytes na figura referem-se a um disco rígido comercial. Tanto o cabeçalho, quanto os dados são envolvidos em um par de campos: campo de sincronismo (SYN) e de checagem de erros (CRC).

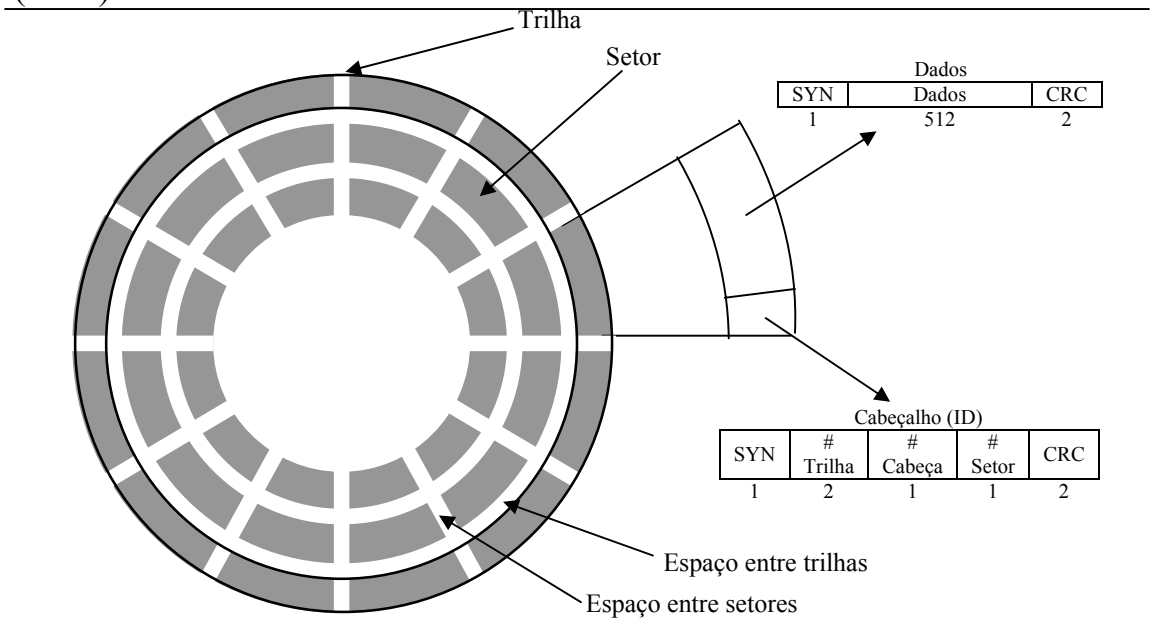


Figura 6.22: Organização típica dos dados em um disco magnético

Um disco magnético possui tipicamente entre 10.000 e 50.000 trilhas e de 100 a 500 setores por trilha. Um setor típico possui 512 bytes de dados. Inicialmente todas as trilhas possuíam o mesmo número de setores, mas recentemente (desde os anos 90) há uma flexibilização desta restrição.

6.4.1 – Discos rígidos

Um disco rígido é normalmente implementado com um conjunto de discos magnéticos chamados bandeja (*platter*), sobrepostos e com um eixo central comum. Cada bandeja possui duas superfícies magnetizáveis e independentes. Um pente de braços mecânicos é usado para posicionar as cabeças de leitura/escrita em um determinado setor. O conjunto de trilhas sobre as quais estão posicionadas as cabeças de escrita/leitura é chamado de **cilindro**. A Figura 6.23 mostra a organização de um disco rígido. Para proteger as bandejas de poeira, o conjunto é lacrado, o que possibilita também melhoria na aerodinâmica do movimento das cabeças de leitura e escrita.

Os discos rígidos giram a uma velocidade constante durante o processo de leitura ou escrita. Hoje tipicamente encontramos discos de 7.200 rpms. As bandejas têm um diâmetro típico de 3,5 polegadas.

Para acessar os dados armazenados em um disco rígido o sistema operacional emite uma seqüência de comandos para o hardware que controla o disco rígido. O primeiro passo é posicionar a cabeça de leitura e escrita sobre a trilha correta. O tempo que leva para este passo é chamado de tempo de busca (***seek time***). Depois de encontrada a trilha, é preciso esperar que o disco gire até o início do setor desejado. Este tempo de espera é chamado de atraso rotacional (***rotational delay*** ou ***rotational latency***). Finalmente, a leitura dos dados é feita. O tempo gasto para esta leitura é chamado de tempo de transferência (***transfer time***). O tempo de transferência é determinado pela velocidade de rotação, tamanho do setor e número de bytes da trilha. As taxas de transferência atuais estão entre 30 e 80 MB/seg.

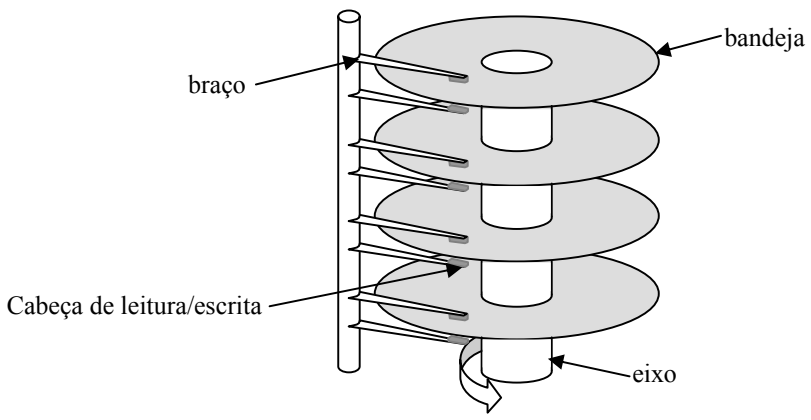


Figura 6.23: Disco rígido

6.4.2 – Discos Ópticos

Embora não tenhamos mencionado os discos ópticos na hierarquia de memória, os mesmos estão se tornando muito comuns hoje em dia, principalmente devido ao advento das tecnologias de regravação e também ao barateamento dos discos compactos (CD).

CD-ROMs são discos de policarbonato recobertos de um fino filme de alumínio e selado com uma camada de acrílico. O alumínio reflete o laser verde emitido pelo leitor, que é então detectado por um sensor de luz. Os CD-ROMs são escritos do centro para as bordas em uma trilha única espiralada. Protuberâncias no substrato do policarbonato indicam níveis lógicos. Estas protuberâncias são chamadas de buracos (*pits*), pois assim parecem quando olhados pela superfície do CD-ROM. Os *pits* medem entre 0,83 e 3,56 microns e têm uma largura de 0,5 microns. As protuberâncias interferem na reflexão do laser de tal forma a cancelar a luz emitida e assim provocar regiões de claros e escuros que são interpretadas como níveis lógicos 1 e 0.

O intervalo entre *pits* é chamado de *land*. Juntando *pits* e *lands*, se pudéssemos 'desenrolá-los', teríamos um comprimento de 8km de informações. A Figura 6.24 mostra um esquema de um disco óptico.

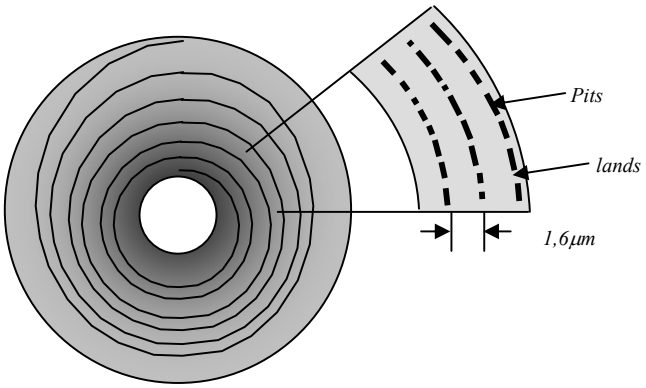


Figura 6.24: Esquema de armazenamento de dados em um CD-ROM

6.4.3 – Matrix de Discos Baratos e Redundantes (RAID)

Os primeiros discos magnéticos sofriam de males próprios de uma tecnologia nascente: confiabilidade baixa, tamanho grande e alto custo. Embora a capacidade dos discos tenha aumentado rapidamente, as melhorias

em termos de desempenho nem sempre acompanharam o aumento de capacidade. Uma matrix de discos baratos e redundantes foi então proposto em 1988 por pesquisadores de Berkeley. O termo RAID (*Redundant Arrays of Inexepensive Disks*) foi então cunhado para se contrapor a tecnologia de discos grande e caros. Os adjetivos são aplicáveis à época da introdução da RAID.

A idéia por trás desta matriz é distribuir os dados em diversos discos e usar alguns para redundância de dados. Esta redundância é importante para promover maior confiabilidade ao sistema: por exemplo, via de regra, em uma RAID se um dos discos quebrar os dados não são efetivamente perdidos. Muitas formas de redundância de dados podem ser exploradas para obter este efeito. Como usamos um conjunto de discos para armazenagem dos dados também é possível explorar como dividir nos diversos discos da matriz. Para classificar os tipos possível de redundância e distribuição dos dados nos discos foram utilizados originalmente 5 níveis distintos (RAID nível 1 a RAID nível 5). Estes níveis não formam uma hierarquia e servem basicamente para identificar que aplicações se beneficiam das características da RAID. Apesar de originalmente propostos 5 níveis, hoje são reconhecidos como padrão 7 níveis (adicionando RAID nível 0 e RAID nível 6) e existem mais uma gama de propostas que não são consensuais ou adotadas pela indústria. A propósito, mesmo dentro dos 5 primeiros níveis propostos alguns não se mostraram viáveis para indústria. Em nosso texto vamos apenas dar uma idéia de alguns dos níveis de RAID, mais freqüentemente utilizados.

6.4.3.1 –RAID nível 0

Uma das características de uma RAID é guardar os dados de forma distribuída, mas sem que o sistema operacional ou usuário perceba. Um conjunto de dados em seqüência é então distribuído pela matrix, em um processo chamado ***striping***. Veja que o Sistema Operacional, ao solicitar nos discos a seqüência de dados, provoca a movimentação de todas as cabeças de leitura (de cada disco) simultaneamente, favorecendo assim o desempenho do sistema. Um possível esquema de distribuição de dados nos discos é mostrado na Figura 6.25. Veja que a informação do título e data de publicação deste livro está distribuída em 4 discos com 3 bandejas cada.

RAID nível 0 é de fato um nome aproximado para este modelo, já que nenhuma redundância é provida. De qualquer forma o modelo está sedimentado com este nome. Exatamente pela falta de redundância, este é o nível do RAID que apresenta melhor desempenho. O problema com este modelo é que se existe uma probabilidade matemática de um dos discos

quebrar, como temos uma matriz, a probabilidade de algum dos discos do conjunto quebrar é ainda maior, o que faz com que a confiabilidade do armazenamento dos dados seja menor. Além disto, pela falta de redundância, é impossível recuperar uma seqüência de dados se algum dos discos apresentar defeito. Por isto, o RAID nível 0 é recomendado para dados não críticos e que precisem de alto desempenho de entrada e saída. Sistemas de backup poderiam utilizar muito bem o RAID nível 0.

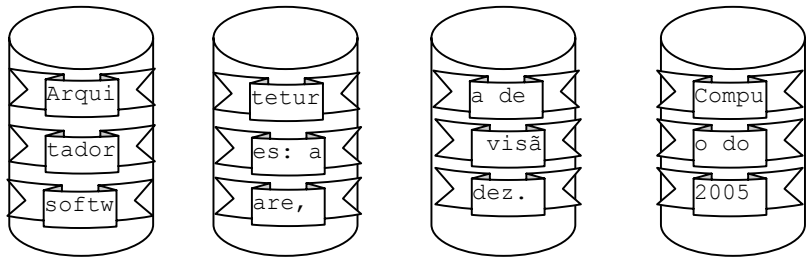


Figura 6.25: RAID nível 0

6.4.3.2 –RAID nível 1

No RAID nível 1 está implementada a mais antiga técnica de tolerância a falhas, chamada espelhamento (***mirroring***) ou sombra (***shadowing***). Simplesmente, os dados são replicado dentro da matriz em discos distintos. Se um disco falhar a informação pode ser obtida de seu disco espelho. Esta solução é a mais simples e confiável de todos os níveis de RAID, entretanto ela é também a mais cara. Seu desempenho é parecido com o RAID nível 0.

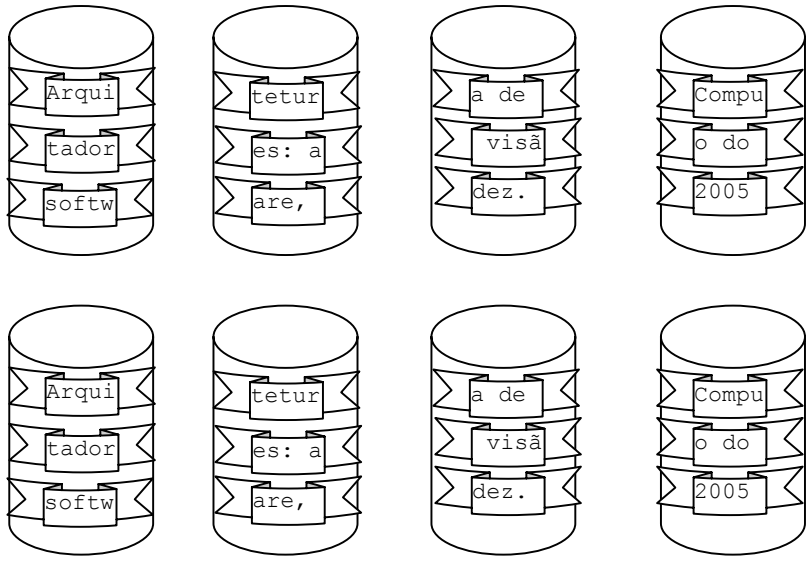


Figura 6.25: RAID nível 1

RAID nível 1 é apropriado para sistemas que precisam de alta tolerância a falhas, como sistemas bancários e de armamentos

6.4.3.3 –RAID nível 2

No RAID nível 2 os dados são distribuídos ao extremo nos discos. Apenas 1 bit de cada informação é guardado em um disco (este valor pode apresentar variações dependendo da fonte de pesquisa, mas o fato é que as tiras são muito pequenas). Apenas para formar um byte são necessários 8 superfícies de discos. Daí é usado um algoritmo de correção de erro capaz de detectar e corrigir erros. Os dados necessários para execução deste algoritmo são armazenados em discos extras. Normalmente a quantidade de discos para armazenar estes dados com informações de redundância é diretamente proporcional ao logaritmo do número de discos onde estão os dados. O algoritmo mais convencional para implementar RAID nível 2 é o código de Hamming.

Este nível de RAID está em desuso e por isto não vamos prover detalhes sobre sua utilização nem seus pontos fortes e fracos.

6.4.3.4 –RAID nível 3

Este nível de RAID é bastante interessante porque garante que usando apenas um disco extra seja possível reconstruir a informação original sem que os dados se percam, em qualquer que seja o disco que por ventura se danifique. A idéia é bastante simples, vamos começar com uma analogia para facilitar a compreensão: imagine que todos os discos de dados da matriz armazenem números. Um disco extra é usado para armazenar a soma de todos os dados de uma seqüência (cada número em um disco). Se algum destes discos quebrar é fácil identificar qual o valor que nele estava armazenado anteriormente à falha simplesmente pegando a soma total, no disco extra, e subtraindo da soma parcial no restante dos discos.

Na prática o RAID nível 3 é implementado com uma checagem de paridade. A paridade é simplesmente o resultado do ou-exclusivo de todos os bits que compõem uma determinada seqüência. Os dados do RAID nível 3 também são distribuídos bit a bit pelos discos. A Figura 6.26 mostra um exemplo. O disco cinza é usado para guardar bits de paridade. Para primeira tira, por exemplo, o bit de paridade é dado por:

$$p_0 = d0_0 \oplus d1_0 \oplus d2_0 \oplus d3_0 = 1 \oplus 0 \oplus 0 \oplus 0 = 1$$

agora suponha que o disco 2 apresente um defeito e precise ser retirado da matriz. O sistema entra então em um modo de operação chamado modo reduzido. O valor armazenado no disco 2 pode ser calculado da seguinte forma:

$$d2_0 = d0_0 \oplus d1_0 \oplus \mathbf{p_0} \oplus d3_0 = 1 \oplus 0 \oplus 1 \oplus 0 = 0$$

No modo de operação reduzida (sem um dos discos) sempre que um dado é solicitado ele precisa ser calculado para ser entregue ao solicitante. Quando um novo disco é introduzido no sistema, em substituição ao disco 2, todos os dados anteriormente em d2 são calculados e armazenados no novo disco. Então o sistema pode voltar a operar em modo normal.

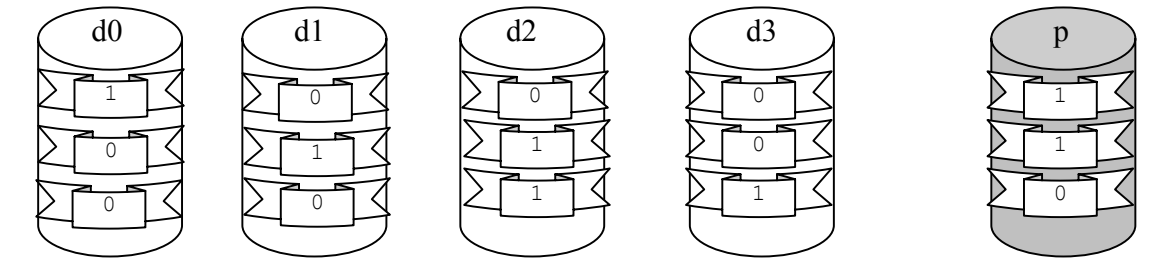


Figura 6.26: RAID nível 3

No modo RAID nível 3 sempre que um dado é gravado na matriz, o bit de paridade associado precisa ser recalculado, mas a leitura não é afetada. As benesses deste modelo se limitam a uma abordagem simples para provê a redundância e muito barata, já que exige apenas um disco extra para permitir a recuperação de informações. Entretanto, as falhas que ocorram nos discos são penalizadas em desempenho porque cada informação precisa ser reconstruída. RAID nível 3 é muito utilizada em sistemas de CAD e tratamento de imagens.

Todos os demais níveis de RAID usam o esquema de paridade para recuperação de erros, entretanto com algumas diferenças na forma como os dados são guardados. O leitor está convidado a buscar mais informações na web.

Como vimos no RAID nível 3, o período em que um disco apresenta defeito é muito crítico para o sistema. Se durante uma falha ou no período de correção da falha, ocorrer um outro defeito em outro disco, o sistema não será

mais capaz de se recuperar. O RAID nível 6 usa mais um disco extra para resolver falhas em mais de um bit, entretanto o preço por maior confiabilidade é pago com discos e mais discos de redundância.

6.5 – Sistema de memórias e desempenho da máquina

Um sistema de memórias apresenta um desempenho geral dependente dos parâmetros de tempo de acesso de seus níveis e quantidade de acertos, por exemplo: uma cache possui, para um determinado programa, uma taxa de acerto de 95% e seu tempo de acesso é 5ns. A memória principal, seu nível inferior, possui em tempo de acesso de 60ns. Qual o tempo de acesso efetivo na cache?

O tempo de acesso efetivo, EAT (***Effective Access Time***) é definido como:

$$EAT = H \times TempoAcesso_{Cache} + (1-H) \times TempoAcesso_{MemóriaPrincipal}$$

Onde H: taxa de acerto na cache.

Então, para nosso exemplo, EAT = 0,95 x 5ns + 0,05 x 60ns = 7,75ns.

A implicação deste tempo de acesso efetivo de 7,75ns é que temos a ilusão de estarmos usando o tempo todo uma memória do tamanho da principal, mas com a velocidade próxima à da cache. Ora, o parâmetro mais sensível nesta equação é a taxa de acerto, por isto é importante, ou mandatório, tentarmos elevá-la ao máximo.

Naturalmente que este cálculo pode ser aplicado, em sua semântica, para os outros níveis da hierarquia, até termos um tempo médio final computado.

Até o presente momento lidamos com aspectos de desempenho do sistema de memórias de forma isolada (dentro da própria hierarquia). Os aspetos de desempenho do sistema de memória, entretanto, interferem decisivamente no desempenho do sistema computacional como um todo. O problema é que a tecnologia que nos possibilita guardar grandes quantidades de dados com um custo aceitável é extremamente lenta se comparada com a velocidade de operação do processador. Alguns números apontam que a velocidade de processamento da CPU dobra praticamente a cada 18 meses, enquanto a velocidade das memórias principais dobra a cada 10 anos. Ora, isto cria uma barreira chamada de barreira de memória (***Memory Wall***).

A *memory wall* é minimizada com o uso ostensivo de caches. Via de regra, as caches funcionam tão bem que este problema é pouco sentido. Para se ter uma idéia, o Pentium 4 possui dois nível de cache internas, o que possibilita acesso a dados/instruções com velocidade compatível com a CPU.

A questão é saber se para todo tipo de aplicação uma determinada cache, com uma configuração particular, apresenta bom desempenho. Um computador de uso geral sofre com a miríade de aplicações e muitas delas mal comportadas do ponto de vista das caches. Não há como prever o desejo do usuário por algum software em particular, então o desempenho do sistema pode ficar aquém das expectativas.

Até então todas as nossas simulações envolveram a execução de instruções de forma que o processador nunca precisasse esperar por um dado, ou mesmo uma instrução subseqüente, estar disponível. Entretanto, isto está longe da realidade. Normalmente um software ao ser iniciado se torna mais lento que o normal, isto porque a cache do sistema de computação está apenas começando a buscar dados e instruções que são mais freqüentes e por isto, praticamente tudo precisa ser retirado do disco, posto em memória principal, levado a cache e só então chega ao processador.

O tempo de uso do processador pode ser então expresso como a parcela em que ele executa as instruções, somada a parcela em que ele fica aguardando dados e/ou instruções do sistema de memórias.

$$CPU_{tempo} = CPU_{executando} + CPU_{esperando}$$

O tempo que a CPU fica esperando um dado/instrução é determinado pela quantidade de falhas na cache e a penalidade por erro na cache (***miss penalty***). O tempo de acesso à cache é considerado o mesmo tempo de um ciclo de clock do processador, o que significa que quando um dado está na cache o processador simplesmente o requisita e o utiliza imediatamente no próximo ciclo de clock.

Então imagine a seguinte situação: uma cache de dados tem uma taxa de falha de 5% e um processador realiza 100 acessos à esta cache. A penalidade por erro na cache é de 10 ciclos por acesso e o código sendo executado usa 500 ciclos de clock, já contabilizadas as falhas na cache de instruções e *interlock* de dados. Qual o tempo, em números de ciclos, efetivamente gasto para executar este código?

De forma simplória podemos dizer que o tempo de CPU efetivamente executando o código é de 500 ciclos. Ora, em 100 acessos à cache, 5 deles resultou em falha, ou seja, 50 ciclos de espera (10x5). O tempo total, em números de ciclos, é então 500 + 50 = 550 ciclos.

Esta, de fato, é uma aproximação porque a penalidade por erro na cache não inclui falta de página, o que, por sua vez, tornaria a penalidade na cache variável. Se extrapolássemos a equação para atender também a memória principal, e com ela as faltas de páginas, poderíamos perceber que uma falta

de página é vertiginosamente mais prejudicial ao desempenho. Desta forma, muitas vezes é preferível aumentar substancialmente a quantidade, e se possível, a velocidade da memória principal que trocar o próprio processador.

6.6 – A visão do software

Quando estamos construindo um software enxergamos o sistema de memória como um repositório único e contínuo. Até o presente momento apenas mostramos como esta memória está hipoteticamente dividida em uma região de dados e outra de código. Entretanto, por convenção, a região de dados do MIPS é subdividida em diversas áreas. A Figura 6.27 mostra as áreas convencionadas. Existe uma região de memória reservada para dados dinâmicos, onde são alocados objetos na sua construção, estruturas de dados, etc. Esta região é comumente denominada ***heap****. Em C, por exemplo, é possível reservar uma parte do heap com a função* `malloc()` *e liberá-la com a função* `free()`*. Em Java, o* `new` *é quem aloca espaço nesta região.*

Os dados estáticos são aqueles comumente encontrados em forma de variáveis declaradas dentro de um programa. Elas ocupam espaço na inicialização do código e não são destruídas durante toda sua execução. Um registrador especial é o apontador global, $gp ($28). Ele é iniciado com o valor 10008000_h de tal forma que possa acessar, com um deslocamento de 16bits no endereçamento, de 10000000_h até $1000fffc_h$.

A região text é o repositório natural para os códigos. O apontador de programas PC é iniciado sempre com o valor 00400000_h.

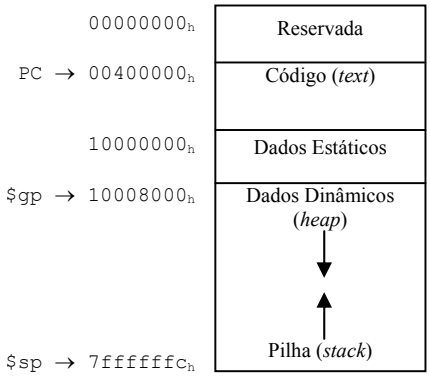


Figura 6.27: Modelo de Memória para o MIPS

A região de memória mais intrigante e mais importante para o domínio de nosso curso é a pilha. A pilha é uma região em que são guardados dados especiais e que tem um apontador em particular.

Nós já estudamos como dividir os nossos programas em unidades pequenas de códigos chamadas de procedimentos. Sabemos que os registradores $a0 até $a3 servem para passar parâmetros de entrada para os procedimentos e que $v0 e $v1 devem guardar os resultados das operações feitas neste procedimento. Além disto aprendemos que $ra é automaticamente carregado com o endereço seguinte ao da chamada do procedimento. Bem, a questão que está faltando responder é a seguinte: imagine que um procedimento A queira chamar outro procedimento B. Quando o procedimento A é invocado, o valor de $ra é imediatamente guardado. Se, dentro de A, houver uma chamada a B, então um outro valor de endereço será salvo em $ra, fazendo com que o endereço de retorno de A seja perdido.

A solução para este problema é o uso da pilha. Guardar o valor de $ra é uma importante tarefa para quem está codificando um procedimento e o lugar, por excelência, onde o $ra deve ser guardado é a pilha.

Um outro problema é que só existem 4 registradores para passagem de parâmetros para o procedimento. E se for preciso passar mais de 4 parâmetros? A solução mais uma vez é usar a pilha.

Finalmente, existem alguns registradores que devem ser preservados, com seus valores originais, após a execução de um procedimento. São eles $s0 até $s7. Cabe ao programador guardar os valores destes registradores na pilha logo no início da execução de um procedimento e reconstituí-los ao final ao seu estado original.

Vamos começar mostrando um exemplo simples de como guardar os valores dos registradores $s0 a $s7 na pilha do sistema, quando invocado o procedimento A.

```
ProcA:
     addi $sp, $sp, -32 # ajusta a pilha mais 8 espaços
     sw $s0, 28($sp)     # guarda $s0 na pilha
     sw $s1, 24($sp)     # guarda $s1 na pilha
     sw $s2, 20($sp)     # guarda $s2 na pilha
     sw $s3, 16($sp)     # guarda $s3 na pilha
     sw $s4, 12($sp)     # guarda $s4 na pilha
     sw $s5, 8($sp)      # guarda $s5 na pilha
     sw $s6, 4($sp)      # guarda $s6 na pilha
     sw $s7, 0($sp)      # guarda $s7 na pilha
..... # códigos do procedimento
.....# podem utilizar $s0 a $s7 localmente
```

```
     lw $s0, 28($sp)     # recupera $s0 na pilha
     lw $s1, 24($sp)     # recupera $s1 na pilha
     lw $s2, 20($sp)     # recupera $s2 na pilha
     lw $s3, 16($sp)     # recupera $s3 na pilha
     lw $s4, 12($sp)     # recupera $s4 na pilha
     lw $s5, 8($sp)      # recupera $s5 na pilha
     lw $s6, 4($sp)      # recupera $s6 na pilha
     lw $s7, 0($sp)      # recupera $s7 na pilha
     addi $sp, $sp, 32 # ajusta a pilha menos 8 espaços
     jr $ra
```

Para o caso descrito acima, onde o procedimento A chama o procedimento B, é necessário guardar também o valor de $ra, assim, teríamos:

```
ProcA:
     addi $sp, $sp, -36  # ajusta a pilha mais 9 espaços
     sw $ra, 32($sp)     # guarda $ra na pilha
     sw $s0, 28($sp)     # guarda $s0 na pilha
     sw $s1, 24($sp)     # guarda $s1 na pilha
     sw $s2, 20($sp)     # guarda $s2 na pilha
     sw $s3, 16($sp)     # guarda $s3 na pilha
     sw $s4, 12($sp)     # guarda $s4 na pilha
     sw $s5, 8($sp)      # guarda $s5 na pilha
     sw $s6, 4($sp)      # guarda $s6 na pilha
     sw $s7, 0($sp)      # guarda $s7 na pilha
..... # códigos do procedimento A
.....# podem utilizar $s0 a $s7 localmente

     jal ProcB # chama o procedimento B

..... #

     lw $ra, 32($sp)     # recupera $ra na pilha
     lw $s0, 28($sp)     # recupera $s0 na pilha
     lw $s1, 24($sp)     # recupera $s1 na pilha
     lw $s2, 20($sp)     # recupera $s2 na pilha
     lw $s3, 16($sp)     # recupera $s3 na pilha
     lw $s4, 12($sp)     # recupera $s4 na pilha
     lw $s5, 8($sp)      # recupera $s5 na pilha
     lw $s6, 4($sp)      # recupera $s6 na pilha
     lw $s7, 0($sp)      # recupera $s7 na pilha
     addi $sp, $sp, 36 # ajusta a pilha menos 9 espaços
     jr $ra
```

Uma vez sabendo como guardar valores na pilha, o importante é saber que os registradores que serão usados dentro do procedimento, que porventura sejam $s0 até $s7 devem ser guardados na pilha. Se apenas alguns deles serão usados, apenas alguns deles devem ser guardados.

Por convenção, os registradores $t0 a $t9 não precisam ser guardados na pilha, pois servem para armazenar dados temporários.

6.7 – Conclusões

Vimos que a hierarquia de memórias nos possibilita um bom desempenho para sistemas com muita localidade temporal e espacial. Hoje ainda não estão disponíveis sistemas com arquiteturas reconfiguráveis o suficiente para adaptar-se às idiossincrasias dos softwares sendo executados. Para um programador, resta a grande lição: já que nós não podemos interferir no hardware que executa nossos programas, devemos provê-los da maior quantidade possível de localidade.

A questão pode ser de engenharia reversa: se as caches e mesmo as memórias principais usam localidade para melhorar o desempenho, um bom programador vai também promover um alto nível de localidade a fim de que seu produto seja executado da forma mais rápida e eficaz possível.

E agora, nos resta pensar como melhorar a localidade. Não existem receitas prontas, mas a prática nos leva a considerar codificar usando pequenos laços que sejam muito utilizados; usar o menor número possível de variáveis; fazer mais leituras que escritas de dados; e tentar minimizar estruturas de decisão e/ou agrupá-las ao máximo. Sempre tentar usar variáveis na mesma seqüência em que elas são alocadas (normalmente na ordem em que foram declaradas, para a maioria das linguagens). Criar objetos e/ou procedimentos e funções pequenos.

Codificar laços pequenos implica em trazer para cache de instruções praticamente o laço inteiro. Se o laço for mesmo pequeno pode ser que apenas um acesso à memória principal seja capaz de alocá-lo completamente na cache. Dispensar laços que sejam executados pouco, ou seja, que tenham parâmetros próximos, como:

```
for (i=0; i<2; i++){ //.
```

Um laço deste tipo será executado apenas três vezes e embora o código de alto nível fique bastante elegante, nem sempre o compilador é capaz de

transformar esta elegância em eficiência de execução. Um programador mais atento aos aspectos do hardware vai procurar alternativas para não precisar usar com freqüência tais laços.

Muitas variáveis são sempre um transtorno para as caches de dados. Elas precisam trazer muitas informações e quase sempre bem aleatórias. Acessos seqüenciais a linhas de uma matriz normalmente causam na cache de dados muitas falhas. Procurar agrupar as variáveis usadas no código de tal forma que eles estejam na circunvizinhança ajuda a explorar a localidade espacial das caches.

Também é bastante complicado o processo de escrita nas caches. Principalmente porque costumamos declarar as variáveis de saída bem distantes das variáveis de entrada. Além disto, transferir dados para memória principal é sempre custoso. A idéia é usar variáveis temporárias, que normalmente são reconhecidas pelo compilador, e somente depois de um longo processamento escrever os dados finais na memória.

Estruturas de decisão, como *if-then-else* provocam desvio no fluxo do programa o que pode fazer com que a cache perca boa parte das instruções que foram carregadas em um bloco. Ao usar instruções de desvio vamos procurar ser tendenciosos por um caminho preferencial, o que pode fazer com que o algoritmo de previsão de desvios funcione bem e busque as instruções corretas para cache, desperdiçando minimamente buscas à memória principal. Estruturas de controle agrupadas permitem que as instruções de máquina de desvio de fluxo fiquem muito próximas umas das outras e isto potencializa o aproveitamento de um bloco.

Objetos pequenos, normalmente podem ser carregados do disco para memória principal em uma única página. Quando codificamos objetos grandes, apenas partes são carregadas por vez e isto pode degradar o desempenho.

Existe um caso especial que nos faz ser ainda mais precisos na construção de nossos softwares: quando conhecemos a arquitetura da hierarquia de memória onde nosso software vai ser executado. Neste caso, é de suma importância uma simulação da hierarquia de memórias de tal forma que se possa identificar quais trechos do código promovem o maior número de falhas e faltas. Reescrevê-los é uma boa idéia! Um caso prático demonstra que para aplicações críticas, envolvendo, por exemplo, um sistema de controle de respiração artificial de uma UTI, as partes que mantêm o sistema operativo devem ser construídas em *assembly* para garantir o desempenho esperado, sem envolver incertezas.

Estas dicas apresentadas são apenas lucubrações, mas não são limitantes. O importante é que um programador comece a se preocupar com a

qualidade do desempenho do seu software, usando para isto os conhecimentos sobre as máquinas onde eles são executados. Os compiladores otimizantes ajudam bastante, mas sua tarefa é árdua e complexa. Ajudá-los é uma bondade.

6.8 – Prática com Simuladores

Existe um gama de softwares capazes de simular o desempenho do sistema de memórias. O mais conhecido deles, certamente é o ***dinero***. Este software é capaz de simular a operação de uma cache com as mais variadas configurações, inclusive caches multi-níveis. Sua operação é baseada em comando de linha e os resultados apresentados em forma de quantidade de falhas. Caches de dados e instruções separadas podem ser simuladas simultaneamente. Os dados de entrada são uma seqüência de referências à memória. O dinero pode ser obtido em http://www.cs.wisc.edu/~markhill/DineroIV/, (acesso em dez, 2005), inclusive com explicações sobre seu uso e instalação.

Uma desvantagem do uso do dinero é sua interface com o usuário. De fato, nenhum dado na memória é mostrado. Por isto, na página web associada a este livro está disponível um software simulador de cache, LBGCache, com interface gráfica que se ajusta melhor à esta fase do aprendizado. Baixe o software e explore os exemplos propostos no site.

6.9 – Exercícios

6.1 – Pesquise na web e produza um resumo sobre *Scratchpad memories*.
6.2 – Pesquise na web e produza um resumo sobre *Memory segmentation*.
6.3 – Pesquise na web e produza um resumo sobre Memória Intercalada.
6.4 – Pesquise na web e produza um resumo sobre *Intelligent RAM*.
6.5 – Pesquise na web e produza um resumo sobre *compulsory*, *conflict* e *capacity misses*.
6.6 – Pesquise na web e produza um resumo sobre *nonblocking cache*.
6.7 – Pesquise na web e produza um resumo sobre *Loop Cache*.
6.8 – Pesquise na web e produza um resumo sobre *Victim Cache*.
6.9 – Pesquise na web e produza um resumo sobre RAID níveis 4, 5 e 6.
6.10 – Como são classificados os discos rígidos (RAM, seqüencial, ROM, volátil)?
6.11 – Mostre em pseudo-código um exemplo de programa que exibe baixa localidade espacial, com respeito a dados.

6.12 – Mostre em pseudo-código um exemplo de programa que exibe baixa localidade temporal, com respeito a dados.
6.13 – Mostre em pseudo-código um exemplo de programa que exibe baixa localidade espacial, com respeito a código.
6.14 – Mostre em pseudo-código um exemplo de programa que exibe baixa localidade temporal, com respeito a código.
6.15 – Em um código que exige muito acesso de escrita a dados, qual política de escrita você sugeriria para uma cache?
6.16 – Esboce o esquema de uma cache com capacidade de 4kbytes, 16 bytes por blocos, 2-*way set associative*.
6.17 – Esboce o esquema de uma cache com capacidade de 4kbytes, 16 bytes por blocos, 4-*way set associative*.
6.18 – Use a mesma seqüência de acessos do exercício da Figura 6.14, mas com uma cache de 512bytes, 16 conjuntos e 4 palavras/bloco. Indique qual o desempenho em termos de quantidade de falhas.
6.19 – Use a mesma seqüência de acessos do exercício da Figura 6.14, mas com uma cache de 512bytes, 32 conjuntos e 1 palavra/bloco. Indique qual o desempenho em termos de quantidade de falhas.
6.20 – Repita os exercícios 6.18 e 6.19 usando uma política de substituição de linhas, onde o último (mais recentemente) elemento acessado é o primeiro a ser retirado da cache.
6.21 – Em um sistema com endereço virtual de 32 bits, páginas de 4kbytes e 4 bytes por linha na tabela de páginas, calcule o tamanho da tabela de páginas para um programa.
6.22 – O que aconteceria em um sistema com 128Mbytes de memória principal que esteja executando simultaneamente 100 aplicações com as mesmas características do exercício 6.23 – O ***hot-swap*** é uma técnica que permite a troca de um equipamento computacional sem que o sistema seja desligado. Todos os nível de RAID permitem *hot-swap* de discos? Por que?
6.24 – Desenhe (esquematize) um sistema contendo cache e memória virtual operando juntas.
6.25 – Um sistema roda uma única aplicação com desempenho da cache de 90% de acertos e 80% de acertos na memória principal. Considerando os tempos de acesso mínimos explicitados na Figura 6.1, qual o tempo de acesso efetivo do sistema de memórias?
6.26 – Descreva como um procedimento recursivo deve utilizar a pilha do sistema. Exemplifique com uma implementação do fatorial.

6.10 – Referências Bibliográficas Específicas

[6.1] Maurice Wilkes. *Slave memories and dynamic storage allocation*. **IEEE Transactions on Electronic Computers, EC-14:2 (April), 1965, pp. 270-271.**

[6.2] David Patterson, Garth Gibson and Randy Katz. *A case for redundant arrays of inexpensive disks (RAID)*. In **Proceedings of the 1988 ACM SIGMOD international conference on Management of data, pp 109-116**

Capítulo 7

Entrada e Saída

7.1 – Introdução

Agora já conhecemos três componentes básicos de um computador: a unidade de controle; a via de dados; e o sistema de memórias. Conhecemos os principais aspectos das linguagens de programação de baixo nível, seja linguagem de montagem, seja linguagem de máquina e agora vamos terminar nossa exploração pelo sistema de entrada e saída (**E/S** ou **I/O**, *Input*/*Output*).

Um computador teria uma utilização mínima se não houvesse meios de alimentá-lo com dados e extrair informações do mesmo. A entrada e saída de dados é que permite a nós, seres humanos, interagir com a máquina. O processamento dos dados, como visto na Figura 5.1, depende de muitos fatores, inclusive da eficiência do sistema de entrada e saída.

Mesmo que tenhamos um sistema de memórias muito eficaz, uma CPU muito rápida e um sistema operacional que consumisse um mínimo de tempo, de nada adiantaria se a entrada de dados, via teclado, por exemplo, fosse muito lenta. Assim também, se ao digitarmos um caractere no teclado este levasse minutos até ecoar no monitor de vídeo, por ineficiência do sistema de saída de dados, a experiência homem-máquina seria frustrante. Portanto, neste capítulo, também lidamos com um fator essencial ao processamento de informações.

Antes de abordarmos o sistema de entrada e saída propriamente dito, vamos conhecer um elemento significativo no trânsito dos dados: o barramento. Um barramento é um conjunto de fios, dispostos logicamente lado a lado, com características elétricas próprias, que transfere uma informação de um ponto a outro de um sistema computacional. Um barramento pode ter uma função específica dentro de um computador, por exemplo, um **barramento de endereços** leva apenas endereços de um ponto a outro do sistema. Um **barramento de dados**, porta apenas dados. Um **barramento de controle** carrega sinais de controle.

De uma certa forma já conhecemos os barramentos de comunicação entre o processador e a hierarquia de memórias. Vamos recapitular a Figura 1.4, que mostra os componentes do computador adaptando-a para mostrar os barramentos de endereço e dados. A re-edição está apresentada na Figura 7.1. Sabemos que o processador, quando deseja um dado que está na memória, disponibiliza o seu endereço (do dado) no barramento de endereços e aguarda até que o mesmo esteja disponível no barramento de dados.

O caso da escrita é semelhante, ou seja, o processador disponibiliza o dado a ser escrito no barramento de dados e coloca o endereço no barramento de endereços. Veja que o processador sempre seta o endereço no barramento de endereços. Isto não é o que ocorre com os dados, que podem vir da memória ou do processador. Por isto, na figura, o barramento de endereços é unidirecional e o barramento de dados é bidirecional.

O que está faltando na Figura 7.1 é o barramento de controle por onde são enviados sinais da CPU para memória. Um sinal típico deste barramento é o sinal de escrita e leitura na memória. O processador precisa informar à hierarquia se deseja escrever ou ler um dado, então é preciso definir um sinal que quando setado signifique escrita e quando ressetado signifique leitura.

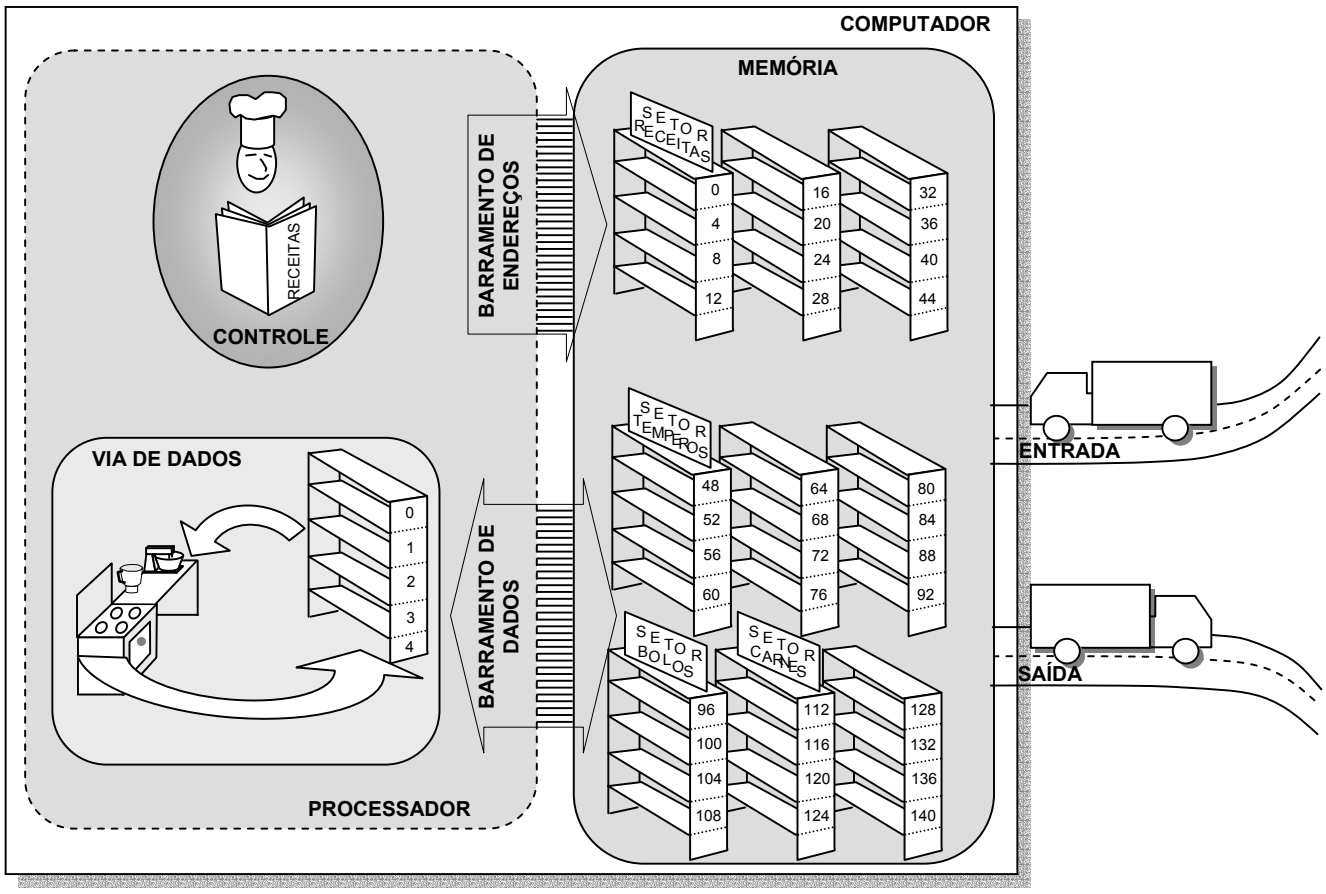


Figura 7.1: Componentes de um Computador

Um barramento é um canal de comunicação que pode ser compartilhado por mais de dois componentes de um computador. A **largura de um barramento** é a quantidade de fios, também chamadas de **linhas de comunicação**, que ele contém. Por exemplo, um barramento com largura de 32 bits, significa um barramento que pode transferir 32 bits simultaneamente de um ponto a outro em um determinado intervalo de tempo.

As características elétricas dos barramentos, como largura das linhas e proximidade entre linha, além do grau de interferência eletromagnética, definem características como a freqüência máxima de transferência de dados.

Um módulo de E/S também é conectado ao processador via barramentos. O módulo de E/S é responsável pelo funcionamento dos periféricos de entrada de dados e de saída de dados, como teclado e monitor, respectivamente. Um módulo de entrada e saída é um dispositivo físico, normalmente contendo alguns portos, onde são colocados dados e palavras de *status*. Um porto (ou porta) é endereçado como se fosse uma posição de memória e sinais de controle específicos são necessários para indicar quando se está fazendo uma leitura ou uma escrita em um determinado dispositivo de entrada ou saída.

A Figura 7.2 mostra o esquema de compartilhamento dos barramentos e diversos dispositivos de entrada e saída, junto com o processador e a memória. Para acessar um dispositivo de entrada ou saída o processador informa no barramento de endereços o número do dispositivo e envia ou recebe dados via barramento de dados. Sinais de controle específicos são usados para fazer leitura ou escrita nos dispositivos de entrada e saída.

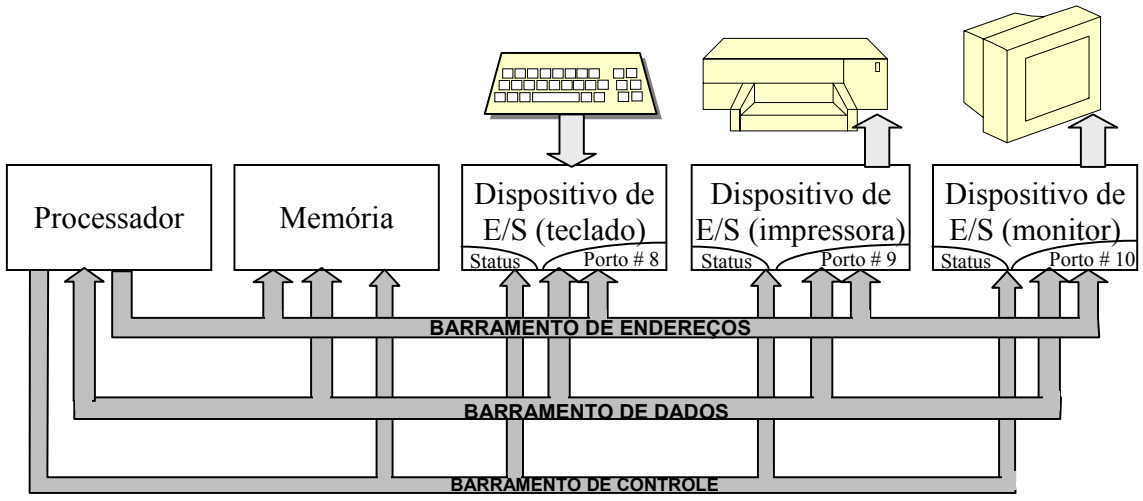


Figura 7.2: Barramentos de Interconexão dos componentes de um computador

Existem duas formas de endereçamento dos dispositivos de E/S: endereçamento específico e mapeamento em memória. No primeiro caso um número está associado a cada dispositivo e o processador precisa ter instruções específicas em sua ISA para fazer entrada e saída, tipicamente instruções `in $reg, dispositivo` e `out $reg, dispositivo`. No segundo caso, um endereço de memória (ou vários) está associado a cada dispositivo de E/S. Nesta situação, uma simples instrução de *load* ou *store* é capaz de ler ou escrever dados em dispositivos externos. O método de mapeamento de E/S em memória é mais simples, mas reserva algumas posições de memória para uso pelos dispositivos, o que diminui o espaço de endereçamento do processador. A Figura 7.3 mostra os dois métodos de endereçamento possíveis.

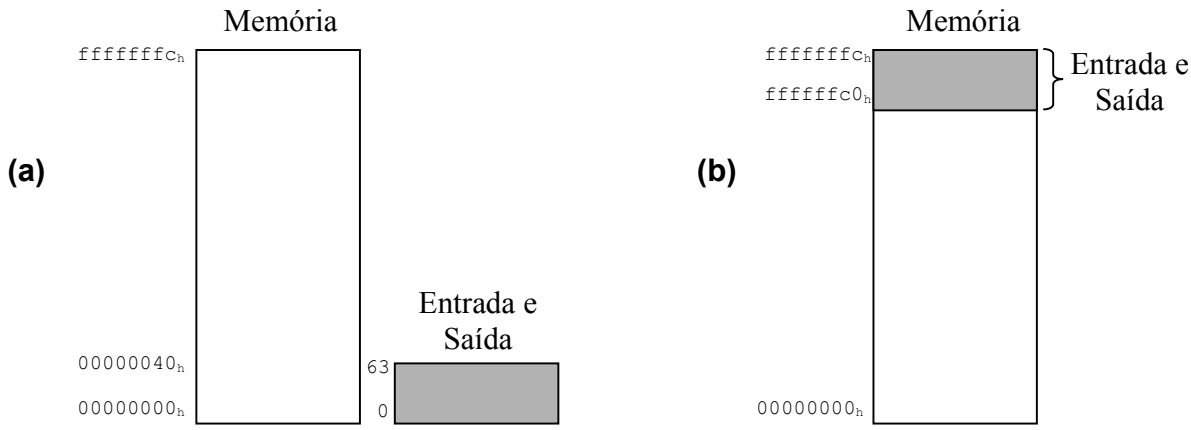


Figura 7.3: Espaços de endereçamanto de Memória e E/S (a) separados e (b)unificados

7.2 – Métodos de Controle de E/S

Existem basicamente três métodos de controle de entrada e saída: ***pooling*** (ou entrada/saída programada); E/S por interrupção; e acesso direto à memória, **DMA** (***Direct Memory Access***).

Na entrada e saída programada o processador fica em um estado de inquisição para os dispositivos de E/S. Em cada dispositivo é verificada a palavra de *status*. Se um dado foi enviado ao dispositivo, ou chegou de um dispositivo, o registrador de *status* do controlador é alterado o que permite ao processador saber se existem dados a serem tratados.

Este é um método simples de implementar, o problema é que alguns dispositivos têm uso em rajadas, como é o caso do teclado. Podemos perceber que enquanto estamos digitando algo, normalmente paramos para pensar nas

palavras, nas fórmulas ou mesmo no *design* do texto, figura ou o que quer que seja. Em outros instantes digitamos uma seqüência de caracteres de forma bem rápida (uma rajada). Nos intervalos entre rajadas o processador ficaria perguntando ao teclado se existe um novo dado e este responderia sempre que não.

Na segunda hipótese de transferência de dados, o processador não se preocupa em perguntar aos dispositivos se existem novos dados, mas o próprio dispositivo pede ao processador sua atenção usando um pedido de interrupção. Neste caso o processador deve parar a execução do processo corrente e atender ao dispositivo que o chama.

Finalmente o processo de DMA desafoga o processador do controle de entrada e saída dos dados entre a memória e o dispositivo periférico.No DMA o processador usa um dispositivo externo, chamado **controlador de DMA**, para programar as transferências de dados. Tipicamente, o processador informa os endereços da memória e do dispositivo de E/S envolvidos na transação e a quantidade de dados a ser transferida. Depois ele libera os barramentos para uso e controle do controlador de DMA. O processo de transferência de dados diretamente para memória implica no acesso aos **barramentos do sistema** (barramento de endereços, dados e controle que interliga o sistema de memórias, o processador e os dispositivos de E/S). É preciso um árbitro para os barramentos a fim de garantir que cada módulo os utiliza no momento adequado. Aqui, neste exemplo didático, o papel do árbitro é desempenhado pelo processador.

O fato de interligar o sistema de memória e os dispositivos de entrada e saída no mesmo conjunto de barramentos tem uma implicação imediata: por natureza os dispositivos de E/S são assíncronos, ou seja, não se pode prever uma certa quantidade de ciclos de clock para a resposta de tais dispositivos. Por outro lado, num sistema de memória, poderíamos indicar um funcionamento síncrono com o processador, definido pela velocidade das memórias e do próprio processador. Então, uma possibilidade de operação conjunta, embora exeqüível, pode não ser a melhor opção. Costumeiramente usamos barramentos síncronos separados para o acesso à memória e barramentos assíncronos dedicados aos dispositivos de entrada e saída.

Esta forma de separar os barramentos acaba promovendo uma hierarquização dos mesmos. Um conjunto de barramentos muito rápido poderia interligar o processador à cache, que por sua vez utilizaria um outro conjunto de barramentos para se comunicar com a memória principal e finalmente, a partir destes últimos barramentos, poderia haver uma interface para um conjunto de barramentos de expansão onde são interligados os dispositivos de entrada e saída. A Figura 7.4 mostra o esquema tradicional.

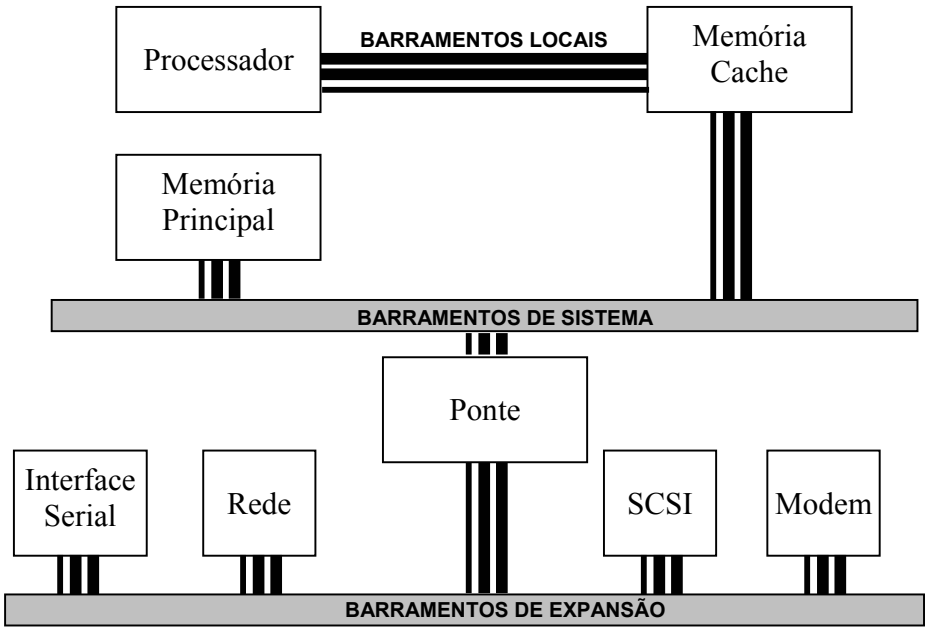


Figura 7.4: Hierarquia de barramentos de interconexão

Na prática esta hierarquia é definida com as velocidades de transmissão de dados suportadas por cada barramento. O exemplo da Figura 7.5 mostra uma abstração da hierarquia de barramentos do Pentium 4. A ponte norte interconecta a memória principal, a saída gráfica de alta velocidade, chamada AGP e uma porta de alta velocidade de rede, chamada de GigaBit Ethernet. A ponte sul interliga os dispositivos de entrada e saída à ponte norte e age como um controlador de DMA.

Esta organização hierarquizada está disposta dentro da placa mãe e tem características particulares, por exemplo, numa placa mãe com *chip set* Intel 845GL a velocidade dos barramentos de sistema é de apenas 400MHz, já com o chip set 875P esta velocidade é de 800 MHz. O primeiro *chip set* utiliza uma ponte norte que tem 760 pinos, enquanto o segundo utiliza uma ponte norte com 1005 pinos. O número de portos PCI, USB etc. também diferem de *chip set* para *chip set*. A propósito, um *chip set*, é o conjunto de chips próximos do processador que servem para interfacear a placa mãe com os diversos dispositivos do computador.

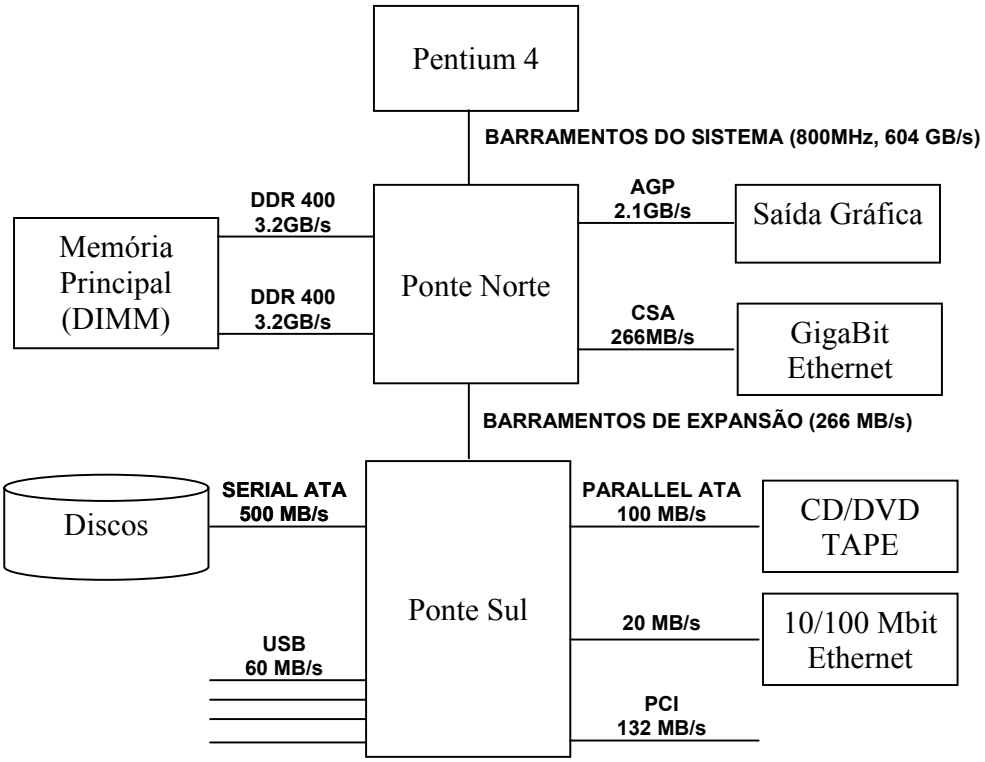


Figura 7.5: Hierarquia de barramentos de interconexão do Pentium 4

Quando utilizamos barramentos assíncronos é preciso definir um conjunto de passos através dos quais o processador irá se comunicar com o dispositivo de entrada e saída. Este conjunto de passos é implementado com a inserção de sinais de controle entre as partes comunicantes. Por exemplo, um dispositivo de entrada de dados requer a atenção do processador a partir de um pedido de interrupção. Ele então deve disponibilizar no barramento de endereços a sua identificação e setar o sinal INTR (***INTerrupt Request***) do barramento de controle. O processador indica que recebeu tal pedido e está pronto para processá-lo setando o sinal INTA (***INTerrupt Acknowledge***). Então o dispositivo disponibiliza os dados e assim sucessivamente.

Este protocolo de sinais é chamado de ***handshaking***, ou seja, um acordo de comunicação que deve ser fechado entre as partes comunicantes antes do início da transmissão/recepção dos dados. Existem protocolos padronizados e todo dispositivo que quiser se comunicar com o processador deve implementar o protocolo apropriado.

Estes protocolos também sugerem uma hierarquia. Considere o caso em que dois dispositivos distintos levantam o sinal INTR pedindo uma interrupção ao processador. Como o mesmo irá responder? Haveria alguma prioridade? A resposta é: sim. Normalmente os processadores possuem apenas um conjunto de linhas para indicar um pedido de interrupção, então é preciso utilizar um dispositivo externo, chamado **controlador de interrupções**, para controlar qual dispositivo deve ser atendido primeiro. O mais famoso controlador de interrupções é o chip 8259 da Intel. Ele processa até 8 pedidos de interrupção simultaneamente. É possível cascateá-lo potencializando uma expansão para até 64 dispositivos de E/S, como mostra a Figura 7.6.

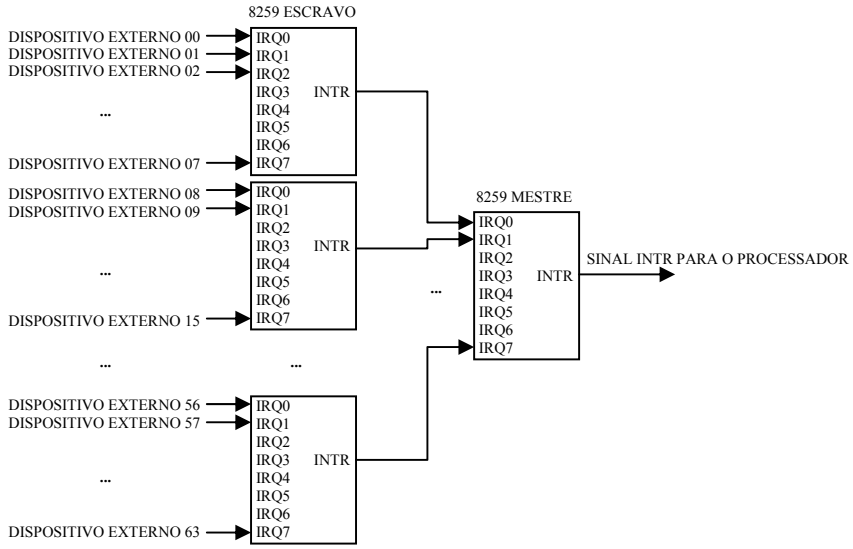


Figura 7.6: Controlador de Interrupções 8259 em cascata

O 8259 é programável e pode decidir prioridades de atendimento para cada pedido de interrupções. Mouse, teclado, discos e impressoras costumam utilizar interrupções para se comunicar com o processador.

Este tipo de interrupção é chamada de interrupção por hardware que nós iremos tratar a seguir na seção da visão do software, dando ênfase ao processo do atendimento da mesma.

7.3 – A visão do software – Interrupções

Mas afinal, o que acontece com um software que está sendo executado por uma máquina quando ocorre uma interrupção? Qualquer que seja a forma, o tratamento da interrupção resume-se a interromper o processamento para atender o pedido. É um caso semelhante a uma chamada

de procedimento, mas aqui, a interrupção pode ser gerada em qualquer instante e a instrução que foi interrompida (se não finalizada) irá ser executada novamente. Então, um fato importante é saber como uma interrupção gera o endereço do procedimento que irá tratá-la, normalmente chamado de ***rotina de tratamento de interrupção****.*

Em uma parte do sistema operacional, ficam guardadas as rotinas de tratamento de interrupções. Os endereços de cada rotina ficam guardados em uma tabela chamada ***Vetor de Interrupções****. Quando ocorre uma interrupção, o dispositivo que o gerou é responsável por identificá-lo dispondo no barramento de endereços o número associado. Este número é usado como índice do vetor de interrupções de tal forma que o endereço adequado da rotina de tratamento possa ser carregado no PC.*

A Figura 7.7 mostra um exemplo do que ocorre quando uma interrupção, por exemplo, uma tecla digitada no teclado, é gerada pelo dispositivo de entrada. O processador estava executando uma instrução `addi` *quando a interrupção foi pedida. O processador então verifica suas linhas de endereços para detectar que a interrupção número 1 foi gerada. O sistema Operacional toma conta da CPU, salvando todo contexto da aplicação que estava sendo executada e apontando para o local da memória onde fica o vetor de interrupções. O índice que será observado neste caso é o próprio número da interrupção. O valor contido nesta posição do vetor é então carregado em PC, o que causa um desvio para a rotina de tratamento da interrupção. Ao fim desta rotina, o fluxo de execução volta para instrução* `add` *com todo contexto do processo do usuário restaurado pelo SO.*

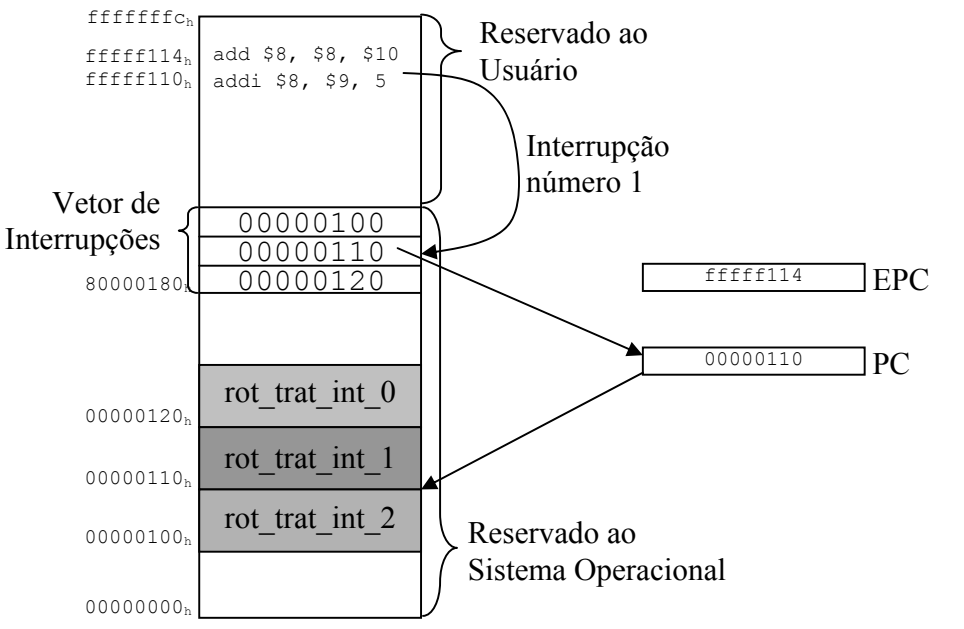


Figura 7.7: Tratamento de Interrupção

Especificamente no MIPS existe um registrador chamado EPC que guarda o valor do PC antes da ocorrência da interrupção, permitindo assim que ao fim da rotina de tratamento da interrupção o valor de PC seja restaurado corretamente.

O MIPS também tem um registrador chamado `cause`*, onde o número da interrupção é depositado e a partir daí o sistema operacional pode implementar sua tabela de interrupções e usar o valor de cause para ser o seu índice.*

Existe uma nomenclatura não universal sobre a definição de exceção. Em tese, uma exceção é um tipo de interrupção gerada dentro do próprio processador, como um overflow*. A exceção é tratada da mesma forma de uma interrupção, ou seja, dentro do mesmo vetor de interrupções há os endereços das rotinas de tratamento de exceções. Esta nomenclatura particular é usada no MIPS, mas a Intel não faz distinção entre interrupção e exceção.*

A propósito, o MIPS também considera uma chamada ao sistema operacional como uma exceção. Uma chamada ao SO é definida como um acesso a um serviço que o software do usuário não pode prestar. Por exemplo, um software do usuário não pode escrever diretamente na tela do computador, então ele utiliza uma chamada ao SO que permite ter os privilégios suficientes para escrever um dado na tela. A instrução `syscall` *é utilizada para prover esta chamada, no nível do software do usuário.*

7.4 – Conclusões

Neste capítulo aprendemos como as vias de comunicação de processador interligam seus componentes e como o mundo exterior se comunica com o computador.

Os barramentos do sistema são basicamente de dados, de endereços e de controle. Os barramentos podem ser hierarquizados de acordo com sua largura e freqüência de operações.

Os dispositivos de Entrada e Saída são módulos que se comunicam com os periféricos de um computador. Estes módulos são acessados pelo processador usando os barramentos de expansão e podem ser mapeados em memória ou podem ter sua numeração independente.

O controle de Entrada e Saída de dados é feito pelo processador de três formas distintas, programada, por interrupção e por DMA. As interrupções são

tratadas por módulos de softwares chamados de rotinas de tratamento de interrupções.

Muitas são as tecnologias que ajudam na velocidade de transmissão de dados de um ponto a outro de um computador. A maioria baseia-se no aumento da largura dos barramentos e/ou na freqüência de operação dos mesmos. As tecnológicas poderiam ser historicamente relembradas, como RS-232, IDE, ATA, SCSI, USB, *Fire wire* etc. Certamente este material já estará desatualizado em pouco tempo, pois estas tecnologias surgem efusivamente no dia-a-dia.

Outra importante lição é como o software do usuário pode pedir ao sistema operacional para usar seus serviços. Esta será abordada, no caso particular do SPIM na próxima seção.

7.5 – Prática com Simuladores

Nesta seção vamos mostrar como o SPIM pode se comunicar com o mundo externo. Quando iniciamos o SPIM surgem duas janelas: a primeira chamada PCSpim e a segunda chamada console. O console serve de meio de comunicação com o mundo exterior, sendo o ponto de entrada de dados, simulando o teclado, e o ponto de saída de dados, simulando o monitor de vídeo.

A interação do software do usuário com o console é feita através do uso da instrução `syscall`. Diversos serviços são implementados como escrita de um número inteiro, de um número real, de uma string, ou a leitura destes dados. Para que a instrução `syscall` execute o serviço apropriado é preciso antes identificá-lo, alimentando o registrador $2 com o número do serviço.

Quando trata-se de uma escrita de dados é preciso ainda dizer qual dado deve ser escrito. Este dado normalmente é especificado em $4. O resultado de uma leitura é disposto em $2, exceto no caso de uma leitura de uma string, onde $4 tem o endereço da string e $5 contém o seu tamanho.

A Tabela 7.1 mostra algumas das chamadas de sistema implementadas no SPIM e seus respectivos argumentos.

Serviço	Código ($2)	Argumentos	Resultados
Imprime inteiro	1	$4 = inteiro	
Imprime string	4	$4 = endereço da string	
Lê inteiro	5		$2 ← inteiro
Lê string	8	$4 = endereço $5 = tamanho	
Fim da execução	10		

Tabela 7.1: Códigos associados ao SYSCALL

Vamos então verificar como podemos criar um programa que leia dois dados do usuário, realize a soma e escreva o resultado.

```
.data
       .asciiz "soma = " # define uma string

.text



main:    addi $2, $0, 5 # serviço 5 (lê inteiro)
         syscall
         add $8, $2, $0 # guarda valor lido em $8

         addi $2, $0, 5 # serviço 5 (lê inteiro)
         syscall

         add $8, $2, $8 # soma os dois valores

         lui $4, 0x1001 # endereço da string
         addi $2, $0, 4 # serviço 4 (escreve string)
         syscall

         add $4, $0, $8 # soma em $4
         addi $2, $0, 1 # serviço 8 (escreve inteiro)
         syscall
```

7.6 – Exercícios

7.1 – Pesquise na web e produza um resumo sobre os tipos de dispositivos de entrada e saída.
7.2 – Pesquise na web e produza um resumo sobre os barramentos SCSI
7.3 – Pesquise na web e produza um resumo sobre os barramentos PCI.
7.4 – Pesquise na web e produza um resumo sobre os barramentos USB.
7.5 – Pesquise na web e produza um resumo sobre os barramentos FireWire.
7.6 – Pesquise na web e produza um resumo sobre arbitragem de barramentos.
7.7 – Pesquise na web e produza um resumo sobre roubo de ciclos pelo barramentos.
7.8 – Mostre com a transferência de dados de um disco é conduzida usando E/S programada, por interrupção e por DMA.
7.9 – Crie um programa para o SPIM que leia dois inteiros e realize a operação x^y.
7.10 – Crie um programa para o SPIM que leia as notas dos alunos de uma turma e calcule a média. Para parar de receber notas, o usuário deve informar um valor negativo.
7.11 – Crie um programa para o SPIM que leia os parâmetros de uma cache e desenhe, com caracteres, a divisão dos endereços.

Capítulo 8

Conclusões

8.1 – Introdução

Chegamos ao fim deste livro com um conhecimento básico da organização de um computador. O conjunto de assuntos estudado nos serve de alicerce para implementações reais. A indústria do hardware evolui rapidamente em nossos tempos, aproveitando os conhecimentos adquiridos em anos de experimentos. Entretanto, muitas são as tecnologias e nomes atribuídos a conceitos básicos e avançados desta área do conhecimento. A questão do *marketing* leva as empresas a usarem uma nomenclatura particular para suas novas implementações, muitas vezes descobertas dentro de um ambiente acadêmico ou em cooperação com centros universitários.

Um estudante que tenha usado este material deve estar apto a aprender novas linguagens de máquina, usar montadores e compreender como ocorre o processo de tradução de uma linguagem de alto nível em códigos binários inteligíveis para a máquina. O sistema de memórias ocupa uma grande região do projeto de um processador e melhorias nele podem promover grandes benefícios no desempenho real de uma máquina, por isto a hierarquia de memórias é bastante explorada no texto.

Estudamos neste texto uma linguagem de montagem (do MIPS) e sua correspondente linguagem de máquina. Conhecemos os principais componentes de uma organização, baseada na escola de *Von Neumman*. Vimos sua interligação com o uso de barramentos e conhecemos o canal de comunicação da máquina com outras máquinas e também com os seres humanos. Descobrimos ainda, como mensurar a eficiência de um computador.

A experiência é a verdadeira escola do processo ensino-aprendizagem. Sob este prisma, a leitura deste texto municia o aluno para o uso de diversos simuladores o que permite uma abordagem mais prática de um assunto tão complexo.

8.2 – Realidades

Assim como no mundo do software existem muitos avanços constantes, no mundo do hardware as técnicas e tecnologias também são ecléticas. Nós estamos chegando ao fim de uma era de evoluções de um modelo de processador que centraliza e controla todas as operações de um computador e que tem as tecnologias de fabricação, com freqüências de operações cada vez maiores, como arma para melhoria de desempenho de um produto sobre outro. Este processo, assim como o petróleo, tem um fim anunciado, com previsões ainda obscuras, mas o fato é que a nova ordem é o trabalho colaborativo, explorando a possibilidade de paralelismo entre as tarefas.

Existem diversos modelos de paralelismo que podem ser explorados. O primeiro deles é o de uma máquina com **superpipelines**. A observação que gerou este conceito é que muitas das tarefas em um estágio de um pipeline convencional pode ser quebrada em subtarefas o que nos permite aumentar a freqüência de operação e produzir mais resultados por ciclos de clock. Outra possibilidade é o pipeline **superescalar** onde muitas unidades de execução são dispostas em paralelo e mesmo a busca e a decodificação são feitas em pares, ou quadras. Para termos uma idéia de como estas implementações de nossos dias podem interferir no desempenho de uma máquina vamos analisar a Figura 8.1 que mostra a execução de seis instruções numa máquina pipeline convencional e numa máquina com superpipeline. O clock é o mesmo utilizado na Figura 4.14, com um período de 6ns.

Neste exemplo a máquina com superpipeline quebra a tarefa realizada em cada estágio em duas partes o que permite em um mesmo ciclo de clock realizar duas operações ao mesmo tempo. Esta vantagem permite que o programa como um todo seja concluído mais rapidamente.

A Figura 8.2 mostra, com a mesma base de tempo da Figura 8.1, a diferença de uma execução do mesmo exemplo em uma máquina superescalar de dois níveis, ou seja, que trata duas instruções por vez. A máquina superescalar apresenta melhor desempenho entre as três, produzindo mais instruções por ciclo de clock. Infelizmente, máquinas superescalares convivem com uma realidade menos animadoras: as dependências de dados não permitem que se explore ao máximo o paralelismo de instruções. A propósito, toda dependência de dados tenta ser tratada, pelo hardware, antes que as instruções sejam enviadas para o ciclo de execução. Este tratamento em hardware é bastante complexo, mas tem uma grande benesse: não é preciso a re-compilação ou re-escrita dos códigos executáveis.

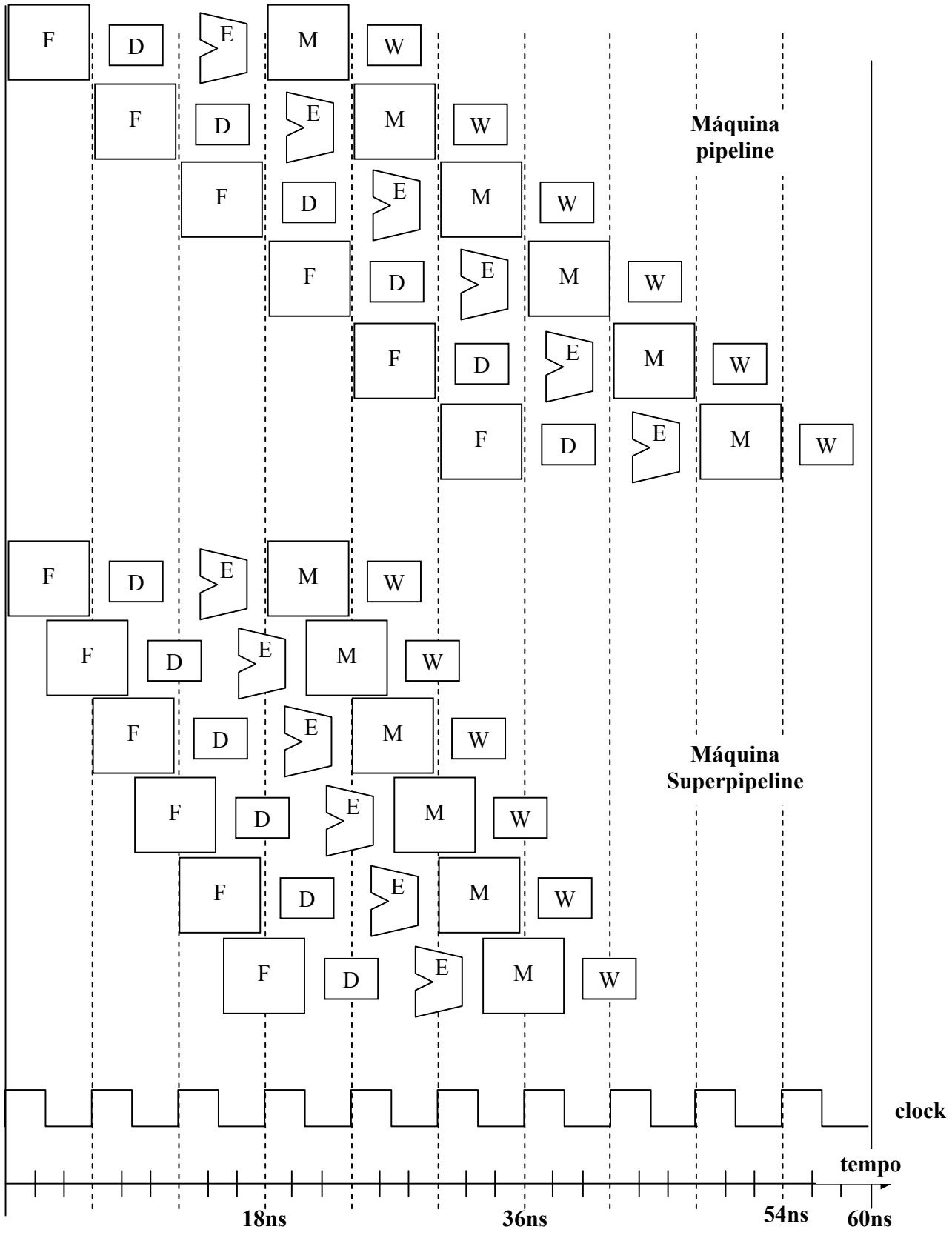


Figura 8.1: Comparação de uma máquina pipeline com uma superpipeline

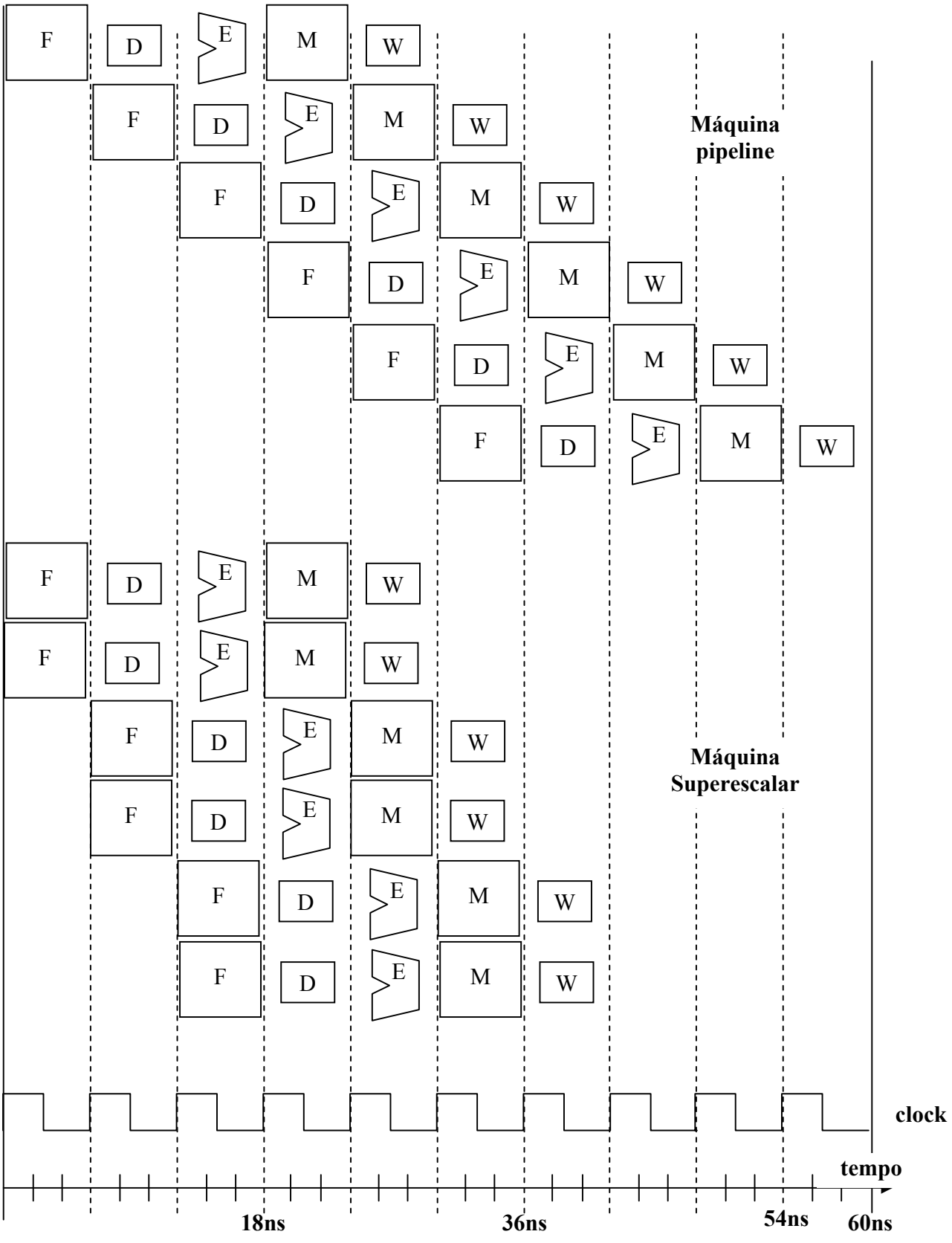


Figura 8.2: Comparação de uma máquina pipeline com uma superescalar

Uma outra classe de máquina que compartilha o mesmo paradigma de paralelismo no nível de instruções é a máquina VLIW (***Very Long Instruction Word***). Nesta máquina uma instrução tem mais de 32 bits de largura, chegando a algo em torno de 128 bits o que permite a realização de 4 instruções em paralelo. A diferença entre uma máquina VLIW e uma superescalar é basicamente o ator que encontra o paralelismo das instruções para serem exploradas. Numa máquina VLIW esta tarefa é do compilador que precisa alocar as operações corretas dentro das instruções. O compilador, por se tratar de um software, pode explorar melhor o paralelismo, mas por outro lado os programas existentes precisam ser recompilados para o uso em uma máquina VLIW.

Uma outra possibilidade de exploração de paralelismo é o uso de diversos processadores em paralelo, cada qual executando um programa, ou partes de um programa, que possam ao final se comunicar para produzir um resultado. Este tipo de organização é chamado de **multi-processador** e também tem enorme implicação na forma como construímos os nossos softwares.

8.3 – Além das Realidades

O futuro próximo das arquiteturas certamente passa por paralelismo. Já somos capazes de fabricar em um único chip um conjunto de processadores e interligá-los por meio de uma rede programável dentro do próprio chip, são as chamadas NoCs (*Network-on-Chip*). Alguns experimentos com NoCs estão sendo iniciados em diversas universidades e em algumas indústrias. Este modelo é interessante pois permite a interligação de diversos processadores com arquiteturas diferentes em uma única pastilha de silício. Atualmente estamos trabalhando na permissividade de operação dos diversos componentes da NoC em freqüências distintas.

Outra frente de pesquisa em andamento é a reconfiguração de circuitos. Imagine que um determinado programa precisa utilizar 4 somas simultaneamente. Numa máquina superescalar ou VLIW com 2 caminhos seriam necessários 2 ciclos de clock. Agora, poderíamos reconhecer a característica do software e, durante a execução do mesmo, construirmos dinamicamente 4 somadores em paralelo, o que nos permitiria em um único cliclo de clock realizar as 4 operações. A reconfiguração é uma nova abordagem que deseja trazer para o hardware a flexibilidade do software.

Bem, os campos de pesquisa em hardware são muitos e certamente o leitor pode se interessar por algum em particular. A Internet nos ajuda

sobremaneira a encontrar temas de pesquisa, principalmente depois do advento das bibliotecas digitais da ACM e IEEE.

Espero ter contribuído para a formação e informação do leitor com este texto. Bom trabalho a todos os leitores!

Apêndice A

Caches Revisitadas

A.1 – Introdução

Já estudamos os principais conceitos envolvidos na idealização das memórias cache, entretanto, não cobrimos por completo o assunto. Para ajudar num entendimento mais completo dispomos neste apêndice de uma abordagem mais avançada, principalmente sobre os aspectos de caches *write-back* e como se dá a escrita de dados.

Já sabemos que a escrita numa cache *write-back* pode trazer algumas dificuldades de inconsistência de dados com o nível inferior da hierarquia de memória (por simplificação adotamos a memória principal, MP). Para resolver o problema de inconsistência é necessário que sempre que formos substituir uma linha da cache por outra vinda da memória principal gravemos antes os dados da linha que será retirada da cache na memória principal. Isto faz com que qualquer falha de escrita na cache em uma linha válida gere um conjunto de escritas na memória principal.

Uma forma mais eficiente de solucionar o problema é acrescentar ao campo de *tag* mais um bit, chamado **bit de sujeira**, ou ***dirty bit***. Este bit será sempre iniciado com 0, mas quando for realizada uma escrita na cache, pelo processador, o valor do *dirty bit* deve ser posto em 1, significando que o conteúdo da cache é diferente do conteúdo da memória principal para aquele bloco. Isto faz com que sempre que for necessário substituir uma linha em que este bit esteja em 0, não seja necessário escrever a linha inteira na memória principal. Por outro lado, se este bit estiver em 1, o conteúdo da linha inteira precisa ser copiado na memória principal, antes de ser sobrescrito por outro bloco. A Figura A.1 mostra o esquema de uma cache com o bit de sujeira (*d*) anexado às *tags*.

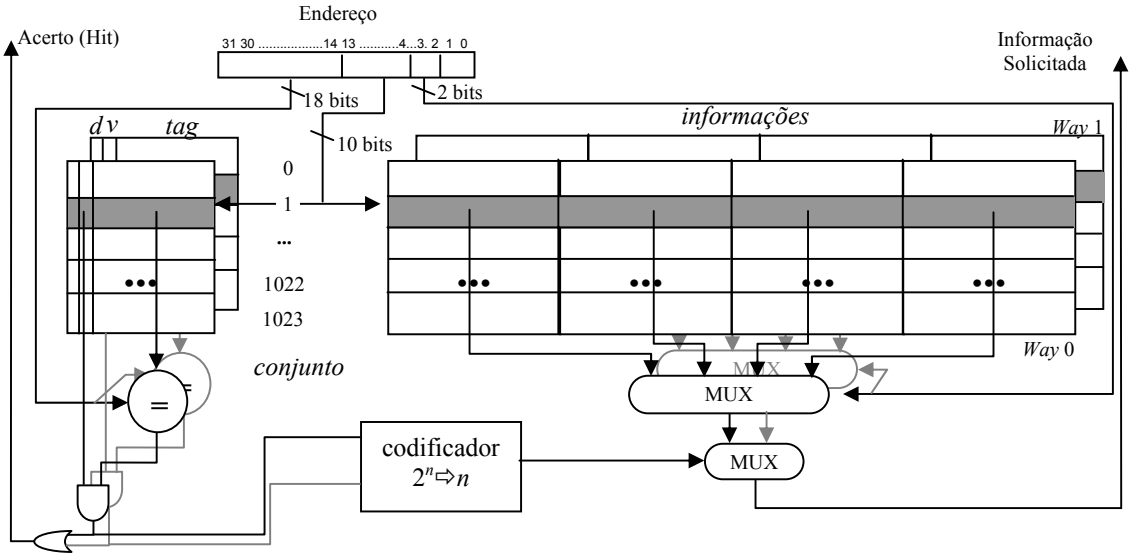


Figura A.1: Memória cache associativa por conjuntos do MIPS com bits de validade e sujeira

A.2 – Falha na Escrita

Outra possibilidade de ocorrências na cache é quando uma falha ocorre durante uma escrita. Neste caso existem duas políticas predominantes:

- ***write allocate***; e
- ***no write allocate***.

No caso de cache que implementam a política *write allocate*, quando ocorre falha numa escrita, o bloco é trazido da memória principal para cache e só então é trabalhado, como se houvesse acontecido um acerto (depende, a partir deste ponto, da cache ser *write-back* ou *write-through*).

No caso de *no write allocate*, o dado que falhou na cache é escrito unicamente na memória principal, sem que o bloco seja trazido para cache.

Assim sendo podemos montar um fluxograma mostrando as principais ações que ocorrem em cada possível situação de acesso à cache variando as políticas implementadas. A Figura A.2 mostra este fluxograma. Não estão mostrados os detalhes da política de troca de linhas, nem como os blocos são achados em uma cache, conhecimento presumido para quem lê este apêndice.

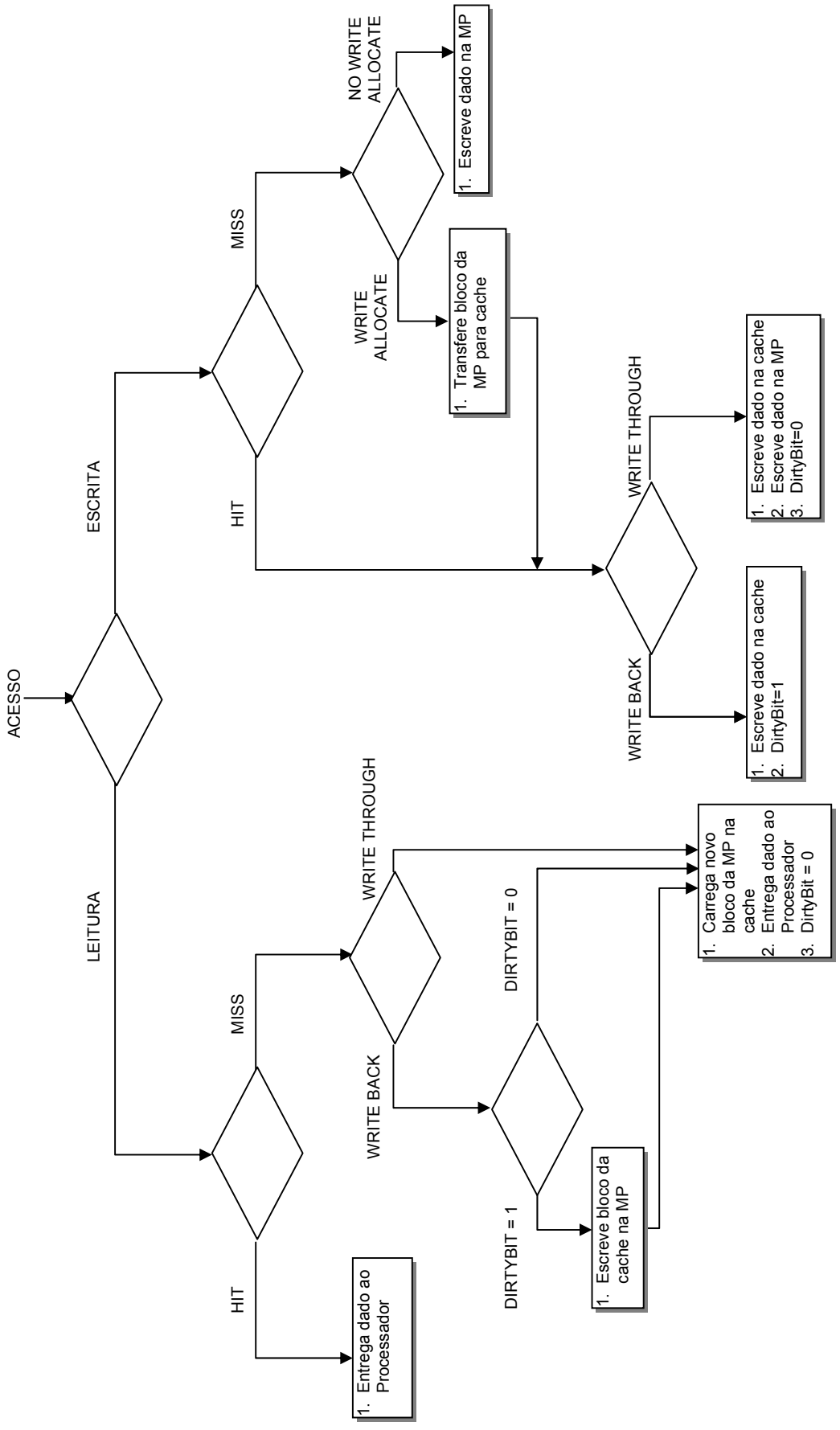


Figura A.2: Fluxograma simplificado de ações na cache

Eduardo Wanderley recebeu o seu doutorado em ciências da computação pela Universidade Estadual de Campinas em 2004. Sua carreira de professor perpassa uma década, sempre em cadeiras ligadas a Arquitetura e Organização de Computadores do Centro Federal de Educação Tecnológica do Rio Grande do Norte. Membro da Sociedade Brasileira de Computação e *Institution of Electrical Engineers*, atualmente desenvolve pesquisas em sistemas de ensino de Arquitetura de Computadores, compressão de códigos e softwares básicos, inclusive compiladores.

Arquitetura de Computadores
a visão do software

A Arquitetura é a arte de determinar as necessidades de um usuário e então projetar uma estrutura para satisfazer estas necessidades tão efetivamente quanto possível, dentro de restrições econômicas e tecnológicas. A arquitetura deve incluir considerações da engenharia, de tal sorte que o projeto seja econômico e realizável; mas a ênfase da arquitetura está nas necessidades do usuário, enquanto a ênfase da engenharia está nas necessidades do fabricante.

A Arquitetura de Computadores é entendida como a visão que um desenvolvedor tem de uma máquina quando programa em sua linguagem de montagem. É preciso, entretanto, conhecer as partes funcionais do computador e suas interligações de modo que se possa extrair o máximo de desempenho. Mas é a visão do usuário que prevalece.

Nesta obra a ênfase na visão do software traz à tona uma abordagem simplista para os meandros dos componentes e procura municiar o leitor com uma abstração da máquina suficiente para que o mesmo possa aprender a programar em linguagem de montagem.

A plataforma computacional envolvida compreende, dentro de um modelo de Von Neumann, um processador MIPS simplificado e um sistema de memórias hipotético tradicional. O modelo de entrada e saída também busca enfatizar a visão do software.

Este livro é direcionado para alunos de graduação em cursos de computação que têm forte ênfase em software, nos primeiros ciclos dos cursos, embora conhecimentos mínimos em programação de nível alto e circuitos lógicos básicos sejam necessários. Um subconjunto de instruções do MIPS é utilizado com simuladores específicos para dar maior portabilidade ao estudante que passa a não depender de uma plataforma física específica.

ISBN 85-89571-06-8
9 788589 571067

CEFET-RN Editora