ARQUITETURA E ORGANIZAÇÃO DE COMPUTADORES

CAPÍTULO 2 - COMO O COMPUTADOR MANIPULA INSTRUÇÕES E INFORMAÇÕES?

Fernando Cortez Sica

Introdução

Com a quantidade de informações com que lidamos atualmente, sem o computador, não seria possível nem encontrar utilidade para tantos dados. Ao aprofundar nossos conhecimentos em arquitetura e organização de computadores, vamos entender como o computador se estrutura, segundo uma visão de alto nível, para que possamos mergulhar em seus elementos de interconexão, na estrutura e função do processador.

Diante disso, já nos deparamos com algumas reflexões: o processador precisa fornecer as instruções e os dados para os seus módulos internos a fim de que haja o processamento? Nesse sentido, como armazená-los internamente de modo que não tenhamos de acessar a memória em toda a etapa da execução da instrução? E, ainda, como uma instrução fica representada na memória?

Já que iremos tratar os aspectos inerentes às funcionalidades do processador e representação de instruções, por que não falarmos também sobre as etapas de execução da instrução dentro de um ambiente dotado de *pipeline*?

Mas, como você deve ter notado, toda forma de armazenamento e manipulação de dados e de instruções é realizada por intermédio de valores numéricos. E como representar e manipular os números em seu formato binário? Vamos entender isso também.

E, por fim, falaremos sobre a unidade de controle e os seus sinais de controle, – sinais estes indispensáveis à sincronização e ao gerenciamento dos módulos do processador.

Vamos entender melhor o funcionamento do computador, por meio da organização e das funcionalidades exportadas pelo processador. Além disso, com os pontos-chave aqui expostos, você poderá otimizar as aplicações a serem desenvolvidas por você em função da adequação ao modelo computacional adotado como plataforma.

Está pronto para começar esse percurso de descobertas? Bons estudos!

2.1 Introdução à arquitetura de computadores

Para entender como o computador funciona, precisamos, antes, conhecer a sua organização e as funcionalidades de seus módulos. Com isso, será possível não somente entendermos, mas, também, propormos soluções de desenvolvimento de sistemas mais otimizados – aproveitando-se as particularidades de um sistema computacional. A seguir, vamos compreender melhor de que maneira os computadores manipulam as instruções e informações por ele processadas, além de descobrir como o computador é estruturado, para que, em seguida, possamos detalhar os seus elementos.

2.1.1 Visão de alto nível do computador

Inicialmente, é conveniente lembrar que, em relação aos computadores ditos como pessoais, seguimos o modelo do matemático John von Neumann (TANENBAUM, 2013). Nesse modelo, a filosofia básica de trabalho consiste no programa armazenado em uma estrutura de memória para que possa ser executado pelo processador. Além da memória, como elementos de sua organização, podemos encontrar a unidade de controle e a unidade lógica e aritmética.

Diante da herança frente ao modelo de von Neumann (TANENBAUM, 2013), pode-se dizer que os elementos básicos de um sistema computacional são (STALLINGS, 2010):

- **sistema de memória**: dividido, ainda, em memória secundária (por exemplo, os HDs – *Hard Disks*), memória principal, memória *cache* e registradores (memória interna dos processadores);
- **unidade de processamento**: dentro do processador, encontramos a unidade de controle, a unidade lógica e aritmética, a rede de interconexão interna e os registradores (para o armazenamento de instruções, dados e informações de controle);

- **unidades de E/S (Entrada/Saída)**: objetiva o interfaceamento com os dispositivos de entrada e de saída;
- **rede de interconexão**: tem a função de interconectar os módulos pertencentes ao sistema computacional.

Além dos módulos em si, precisamos ter em mente que, para que eles atuem, é necessária toda uma sincronização de trabalho. Essa sincronização é atingida pela geração de sinais de controle, os quais estão presentes dentro do processador para que este possa executar as instruções, e, também, estão presentes nos demais módulos que integram o computador.

Nesse sentido, conhecer os módulos do computador e a sua forma de trabalho é fundamental para que você possa saber, entre outras questões, quais são os pontos de gargalos de processamento. Esse conhecimento permitirá que você otimize os sistemas a serem desenvolvidos ou, ainda, reconfigure o computador para que sua performance computacional se torne mais eficiente.

2.1.2 Interconexão do computador, estrutura e função do processador

Antes de entrarmos especificamente no processador, vamos detalhar um pouco mais os elementos do sistema computacional, destacando a sua interconexão. A figura a seguir ilustra os módulos e a sua interconexão.

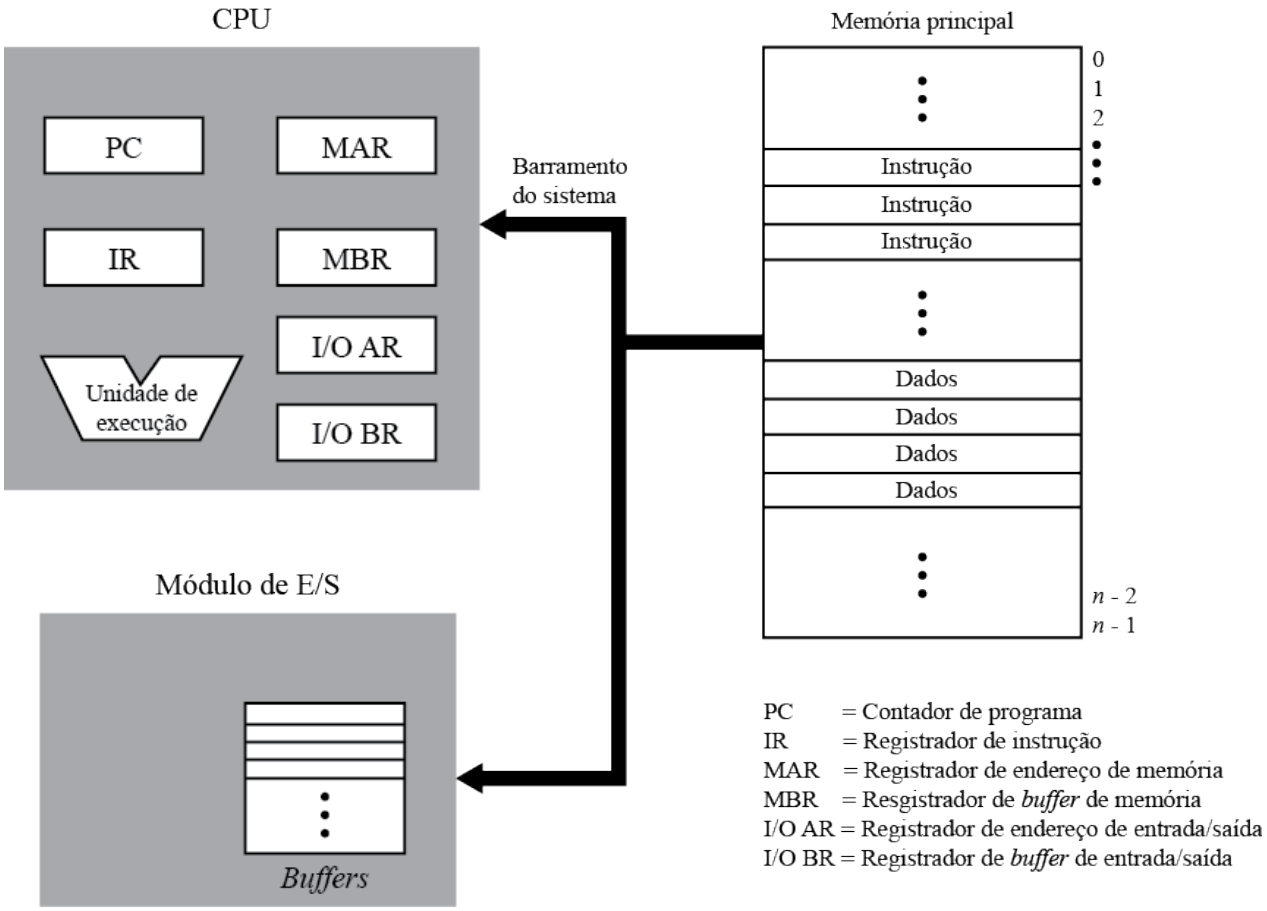

Figura 1 - Módulos básicos de um computador e sua interconexão, em que é possível notar a presença do barramento do sistema para prover a interconexão dos módulos.
Fonte: STALLINGS, 2010, p. 55.

Na figura acima, você pode notar a presença de canais de comunicação inter-relacionando os módulos do sistema computacional. Tais canais transportam informações envolvendo os dispositivos de E/S (interfaceados

via *buffers*) além de transportarem dados e instruções entre a memória e o processador. Por sua vez, internamente à CPU são apresentados, além da unidade de execução, registradores de controle – os quais você verá mais adiante.

A estrutura apresentada na figura anterior permite que o processador desempenhe a sua função básica: executar instruções. Para que uma instrução seja executada, é necessário que esta seja carregada da memória para o processador. Sendo assim, para executar um programa completo, o processador deverá, em suma, realizar a busca da instrução e executá-la – linha a linha de código.

A execução representa o envio de informação e a ativação de módulos específicos, correspondentes a cada classe de instrução. Geralmente, os processadores manipulam quatro classes de instruções, enumeradas a seguir (STALLINGS, 2010):

- **processador-memória**: representa as instruções de transferência de dados entre o processador e a memória;
- **processador-E/S**: ao invés da transferência de informações envolvendo a memória, as instruções do processador-E/S intercambiam informações entre o processador e os módulos de E/S;
- **processamento**: compreende as instruções que realizam a execução propriamente dita. As informações a serem manipuladas deverão estar, necessariamente, carregadas nos registradores do processador;
- **controle**: as instruções de controle estão relacionadas aos desvios condicionais e não condicionais, alterando, dessa forma, a sequência do fluxo de processamento das instruções.

Como mencionado anteriormente, as instruções precisam, para que sejam decodificadas e executadas, estar carregadas nos registradores do processador. No entanto, cabe a questão: como as instruções ficam alocadas na memória do computador? Para responder a essa pergunta, vamos adentrar em aspectos de organização de registradores para que possamos compreender o ciclo da instrução.

2.2 Organização dos registradores

Como mencionado, todas as instruções e informações a serem processadas deverão, necessariamente, ser carregadas previamente no processador. Para tanto, o processador possui um conjunto de registradores de uso geral (GPR – *General Purpose Registers*) e um conjunto de registradores para que sejam usados no controle do processamento.

2.2.1 Registradores visíveis ao usuário: uso geral e dados

Registradores do tipo GPR têm a função de armazenar as informações sob processamento. Por exemplo, caso seja necessário executar uma operação de soma do tipo C = A + B, os valores de A e B deverão ser carregados previamente da memória e armazenados em registradores. Após o processo de execução ser completado, o resultado deverá ser armazenado também em registradores.

Você pode estar se perguntando se o fato de ter que carregar sempre, antes, em registradores não consome mais tempo de processamento. Primeiramente, podemos afirmar que quando as informações estão presentes em registradores, a chance de serem reaproveitadas em outras instruções próximas é alta. Sendo assim, os compiladores possuem estratégias de otimização para que se possa aproveitar ao máximo os valores previamente carregados nos registradores.

Um segundo aspecto refere-se à organização do *hardware*. Por exemplo, em computadores do tipo RISC (*Reduced Instruction Set Computer*), as operações que manipulam informações são do tipo registrador-registrador, ou seja, as fontes são oriundas de registradores e o resultado é salvo também em registrador (STALLINGS, 2010). Dessa forma, os circuitos internos de interfaceamento tornam-se mais simples e, consequentemente, não é necessário um módulo de seleção da fonte (chaveando-se entre a memória e o banco de registradores).

VOCÊ SABIA?

Você sabia que alguns processadores, como o registrador zero, apresentam um valor fixo em 0 (zero)? Essa escolha é motivada pelo fato de que a frequência de aparição de instruções do tipo carga do valor 0 (instanciação de dados com o valor zero) é alta. Dessa forma, economiza-se o tempo de carga da constante já que o valor zero já se encontra internamente ao processador, bastando, para tal, realizar uma movimentação de dados do tipo registrador-registrador (por exemplo, $Reg_1 = Reg_0$).

Para finalizar essa seção, como exemplo prático, convém mencionarmos que o processador Intel Core i7, apresenta, em cada núcleo, 16 registradores GPR (*General Purpose Register*) em seu banco de registradores (TELES, 2018).

2.2.2 Registradores de estado e de controle

Além dos registradores do tipo GPR, o processador mantém um conjunto de registradores destinados ao armazenamento de códigos de estado e de valores de configuração.
Na figura anterior, é possível encontrar alguns exemplos de registradores desse tipo, tais como os listados a seguir (STALLINGS, 2010):

- **PC (*program counter*)**: o registrador PC, também chamado como IP (*instruction pointer*), armazena o valor de endereço que contém a próxima linha de código (instrução) a ser executada. A cada busca de instrução, o valor de PC é incrementado para que, no próximo ciclo, seja executada a instrução subsequente. Em instruções de desvio, o PC pode receber o valor do endereço que contempla a instrução após o salto (desvio);
- **IR (*instruction register*)**: como você já está sabendo, para que a instrução possa ser executada, ela precisa ser carregada para dentro do processador. O local para o armazenamento da palavra que contém a instrução é o registrador IR. Com base no valor armazenado, o processador realiza todas as etapas envolvidas na sua execução;
- **MAR (*memory address memory*)**: o registrador de endereço de memória contém o valor do endereço para o próximo acesso à memória. Com o uso desse registrador, o processador informa, ao sistema de memória, o endereço sobre o qual será realizado o acesso (para a leitura ou para a escrita);
- **MBR (*memory buffer register*)**: após o MAR configurado, ao efetivar o acesso à memória, o item manipulado (gravado ou lido da memória) é armazenado no MBR. Sendo assim, o MAR e o MBR fazem parte do interfaceamento entre o processador e a memória;
- **I/O AR (*input-output address register*)**: de forma análoga ao interfaceamento entre processador e memória, existe o registrador I/O AR, que contém o endereço do dispositivo de E/S a ser acessado;
- **I/O BR (*input-output buffer register*)**: o item manipulado entre o processador e o dispositivo de I/O é armazenado no registrador I/O BR.

Além dos registradores listados acima, não poderíamos deixar de citar o registrador de *flags*. O registrador de *flags* é instanciado ao término, por exemplo, das operações aritméticas a fim de armazenar o estado final da operação, bem como informar se a operação resultou em algum valor negativo, nulo ou, ainda, se houve estouro

de representação numérica (*overflow*). Mas por que armazenar esse estado final da operação? Vamos lhe dar um exemplo. Na maioria dos processadores, para se fazer uma operação de desvio, é necessário testar exatamente esse registrador de *flags*.

A figura a seguir ilustra a tradução de um desvio condicional a partir de uma linguagem de alto nível para um *assembly* (linguagem de máquina) hipotético.

Além dos registradores listados acima, não poderíamos deixar de citar o registrador de *flags*. O registrador de *flags* é instanciado ao término, por exemplo, das operações aritméticas a fim de armazenar o estado final da operação, bem como informar se a operação resultou em algum valor negativo, nulo ou, ainda, se houve estouro de representação numérica (*overflow*). Mas por que armazenar esse estado final da operação? Vamos lhe dar um exemplo. Na maioria dos processadores, para se fazer uma operação de desvio, é necessário testar exatamente esse registrador de *flags*.

A figura a seguir ilustra a tradução de um desvio condicional a partir de uma linguagem de alto nível para um *assembly* (linguagem de máquina) hipotético.

```
if (a >= b)
  {
     <sequência_if...>
  }
else
  {
     <sequência_else...>
  }
<sequência_após_desvio...>
. . .
```

(a)

```
sub reg3, reg1,reg2          (1)
js parte_else                (2)
   <sequência_if...>
jmp parte_após_desvio        (3)
parte_else:                  (4)
   <sequência_else>
parte_após_desvio:           (5)
   <sequência_após_desvio...>
. . .
```

(b)

Figura 2 - Exemplo de tradução de um trecho de código usando linguagem de alto nível (a) para um assembly hipotético (b) a fim de ilustrar o registrador de flags.
Fonte: Elaborada pelo autor, 2018.

Na parte (b) da figura acima, tem-se o código traduzido da parte (a) para o *assembly* de uma máquina hipotética. Suponha que os valores das variáveis ***a*** e ***b*** já estejam carregadas nos registradores reg_1 e reg_2, respectivamente.

A linha um realiza a subtração desses dois registradores e armazenando o resultado no registrador reg_3. Além do cálculo em si, a instrução de subtração também instancia os campos do registrador de *flags*. No nosso exemplo, daremos atenção ao campo ***S*** (sinal). Esse campo será instanciado em um caso a instrução de subtração retorne um valor negativo e, caso contrário, o campo será instanciado em zero.
O campo ***S*** será avaliado na instrução da linha dois: ***js*** (*jump signal*). Caso o campo ***S*** esteja sinalizado em um (resultado negativo), o fluxo de execução do código será desviado para o endereço apontado pelo rótulo ***parte_else*** (caso negativo, então, significa que ***a*** < ***b***). Caso contrário (***S*** = 0), assim, o trecho identificado como ***<sequência_if...>*** será executado.
Caso o fluxo tenha passado pela ***<sequência_if...>***, a linha três, contendo um desvio incondicional (***jmp*** = *jump*) será responsável por saltar o trecho ***<sequência_else...>*** para que o fluxo seja direcionado à primeira instrução fora do comando if... else (***<sequência_após_desvio...>***).
Por fim, as linhas quatro e cinco representam os rótulos para nortear os desvios junto às instruções de desvios condicionais e não condicionais.

VOCÊ SABIA?

Você sabia que é possível usar desvios não condicionais em linguagens como C/C++? Em disciplinas de programação, o desvio não condicional "goto" tem o seu uso proibido. Porém, em algumas situações, torna-se necessário utilizá-lo. Por exemplo: ao codificar programas para sistemas embarcados, em que se tem uma forte restrição de memória (não se pode realizar muitas chamadas de funções para evitar estouro de pilha – *stack*), às vezes, o goto é essencial. Não somente em sistemas embarcados podemos encontrar o "goto" – por exemplo, basta ver a codificação do *kernel* do Linux.

Cada família de processadores contém, em sua organização, o seu próprio mapeamento de registradores. A quantidade de registradores influenciará o formato com que as instruções são armazenadas na memória.

2.3 Ciclo de Instrução

Anteriormente, abordamos a arquitetura básica, as funcionalidades das instruções e dos registradores. Mas, cabe a pergunta, como as instruções são efetivamente executadas? Antes de detalharmos como as instruções são executadas, vamos ver, antes, como elas são armazenadas na memória.
Vamos supor a operação de subtração codificada em *assembly* contida na figura anterior: "**sub reg_3, reg_1,reg_2**".

Para começarmos a falar sobre a representação, lembre-se de que o computador armazena apenas números. Então, os dados são números (inclusive os caracteres são representados pelo valor contido na tabela ASCII) e as instruções não poderiam deixar de ser um valor numérico.

Cada instrução possui um código denominado *opcode*. Assim, é a partir do *opcode* que o processador reconhece quais os módulos deverão ser ativados e qual o caminho que aos dados deverão seguir dentro dele. A figura a seguir ilustra um possível formato para a instrução de subtração.

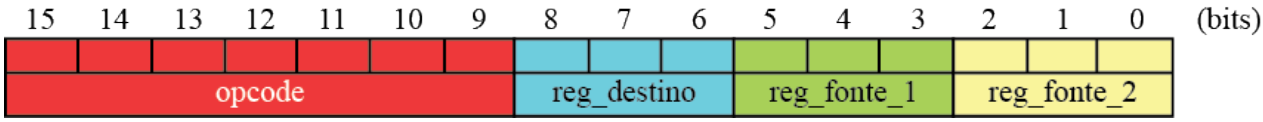

Figura 3 - Exemplo de formato de instrução de subtração de um processador hipotético de 16 bits.

Na figura acima, supõe-se que a máquina hipotética em questão tem uma largura de palavra de 16 *bits*, sendo que o seu banco de registradores engloba oito registradores e possui 125 instruções em sua arquitetura de conjunto de instruções (ISA – *Instruction Set Architecture*). Fazendo as contas rapidamente, chegamos à conclusão de que serão necessários três *bits* ($\log_2 8 = 3$) para endereçar cada registrador e mais sete *bits* para mapear as instruções ($\log_2 125 \approx 7$).

Então, para mapear "**sub reg_3, reg_1,reg_2**", atribuindo aleatoriamente (para fins de ilustração) o valor 0110111 para o *opcode* da instrução de subtração, tem-se a palavra: 0110111 011 001 010 (lembrando que $011_{(2)} = 3_{(10)}$; $001_{(2)} = 1_{(10)}$; $010_{(2)} = 2_{(10)}$).

Cada classe de instrução tem o seu formato de representação dentro da palavra de 16 *bits*. Sendo assim, os campos podem variar de tamanho e de representação, exceto o campo do *opcode* que, nesse caso, será mantido fixo (15 a nove *bits*).

O início do processo de execução da instrução consiste em sua busca a partir da memória. Como mencionado antes, o processador coleta toda palavra da memória (conjunto formado pelo *opcode* e seus operandos) e armazena no registrador IR, para que as informações sejam encaminhadas aos submódulos ao longo de sua organização. Nesse caso, existe um fluxo de informações e de sinais pelos caminhos do processador – caminhos estes chamados de *datapath*.

CASO

Em certas situações, ao nos depararmos com um problema para o qual temos de ter uma solução que envolva *hardware*, surgem algumas dúvidas para reflexão: que tipo de *hardware* usaremos? Qual microcontrolador/microprocessador utilizar para a questão? Porém, algumas vezes, a aplicação é muito específica e usar uma plataforma pronta não é muito adequado. Nesse caso, pode-se desenvolver o próprio processador – que manipula o seu próprio conjunto de instruções, possui o seu próprio mapeamento de registradores e o seu próprio fluxo de dados. Os pontos vistos neste capítulo ajudariam no projeto de esboçar o processador otimizado para o problema. Para tanto, a implementação seria baseada na utilização de *hardware* reconfigurável (ou lógica reconfigurável). E, para a implementação, seria usada alguma linguagem de descrição de *hardware* (HDL – *Hardware Description Language*), tal com Verilog ou VHDL. Nessa situação, poderá ser implementado, ainda, o sistema embarcado baseado na ideia de SoC (*system on chip*) onda há a integração do mundo do *software* e do mundo de *hardware* na mesma FPGA (*Field Programmable Gate Array*, em português "Arranjo de Portas Programáveis em Campo"). Em função da integração de *hardware* e *software*, as etapas de análise e projeto de um SoC também serão particionadas, ou seja, uma frente do projeto se envolverá com o *hardware* e uma outra frente se envolverá com o *software* – esse particionamento é chamado como *CoDesign*.

Os módulos e suas conexões, no *datapath*, são diagramados de acordo com as fases de execução das instruções. De acordo com (TANENBAUM, 2013), a execução de uma instrução poderá ser dividida nas seguintes etapas:

- busca, pelo processador, pela próxima instrução da memória e carrega no registrador RI;
- incremento do registrador PC, de modo a ser preparada a busca pela instrução subsequente (a modificação do valor de PC poderá ocorrer posteriormente caso a instrução seja de desvio);
- codificação do tipo da instrução carregada;
- carregamento dos operandos nos registradores (caso necessário);
- execução da instrução e registro do resultado;
- retorno ao início para recomeçar o ciclo em relação à próxima instrução.

Essa sequência de execução de uma instrução poderá ser manipulada de forma puramente sequencial ou, ainda, ser alvo de otimização de fluxo de processamento através da aplicação da técnica de *pipeline*.

2.3.1 Buscar, ler e interromper

Até o momento, comentamos as etapas para a execução das instruções sem a ocorrência de eventos externos. No entanto, você deve estar se perguntando: e as interrupções de *hardware* e de *software*? O que acontece no fluxo quando há a incidência de alguma interrupção? As interrupções causam um grande impacto na execução dos processos, pois elas devem ser atendidas quase que prontamente.

VOCÊ SABIA?

Você sabia que existe uma interrupção que é chamada quando o computador fica ocioso? E que existe outra que é ativada sempre que houver um pulso de *clock*? E que no Linux é possível visualizar uma lista de interrupções listando o conteúdo da pasta */proc/interrupts*?

O impacto é causado pelo fato de que quando o sistema operacional escalona os processos (interrupções são também tratadas como processos), deve haver a troca de contexto. Trocar um contexto significa que é necessário salvar todas as informações relevantes do processo (tais como o valor do registrador PC, as *flags* e algumas outras informações) para que, depois, quando o processo voltar à ativa, possa ser restaurado exatamente do mesmo ponto e com as mesmas configurações vigentes no momento de sua interrupção (PATTERSON, 1998).

VOCÊ QUER LER?

Em algumas ocasiões, é necessário implementar *drivers* (*softwares* que interfaceiam com o sistema operacional para controlar dispositivos de E/S – acessar registradores internos, interrupções etc.). No portal Labworks (2018) há material para leitura e exemplos de código para a construção de *drivers*. Seria um bom começo caso você esteja interessado no assunto: < https://e-labworks.com/treinamentos/drivers/source>.

A figura abaixo ilustra o tratamento das etapas de execução de uma instrução envolvendo a manipulação de interrupções. Em (a) tem-se o esquema básico de representação e em (b), um detalhamento das etapas propriamente ditas.

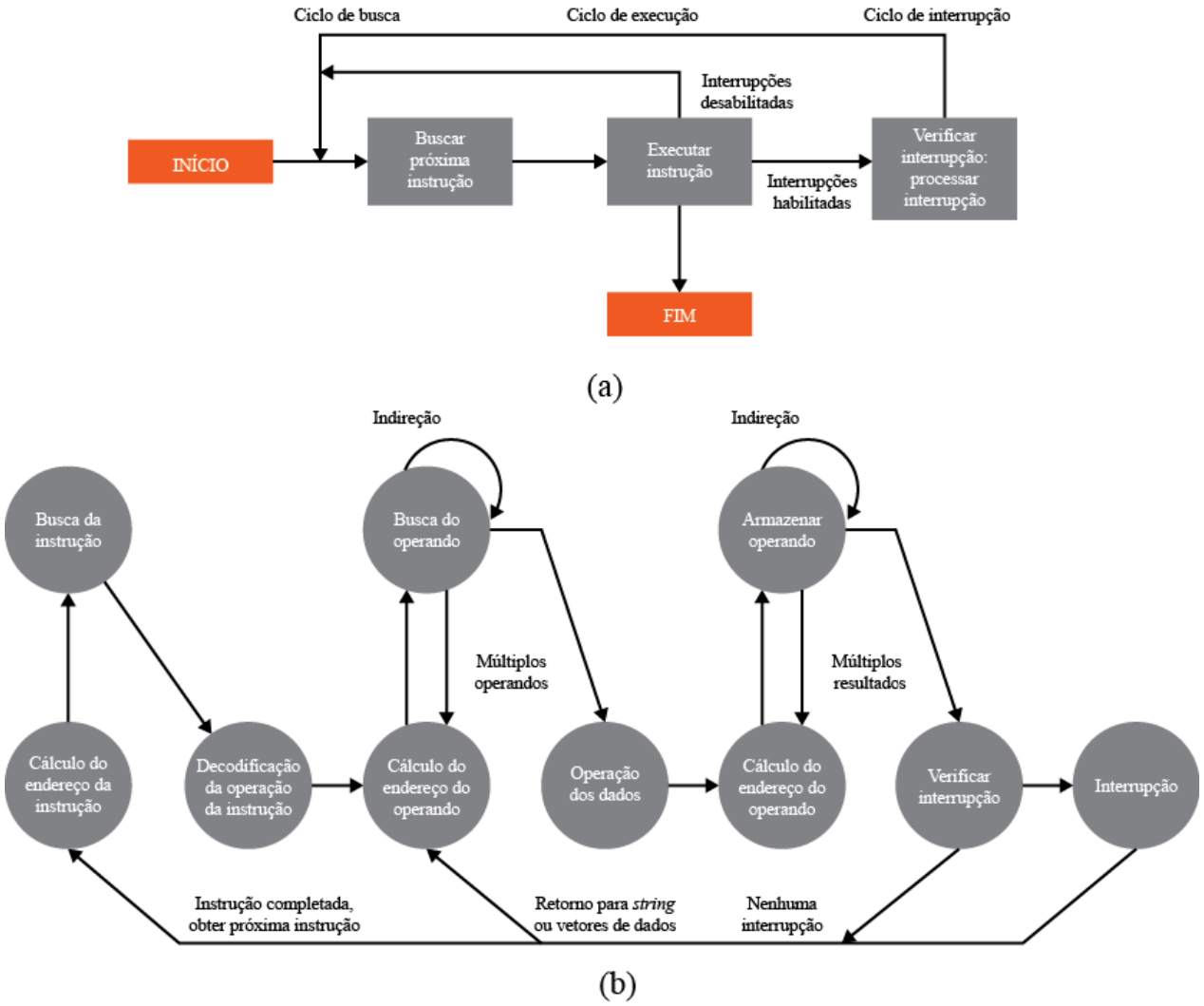

Figura 4 - Sequência de etapas para execução das instruções com a manipulação de interrupções: em (a), a representação básica, e em (b), o detalhamento das ações.
Fonte: STALLINGS, 2010, p. 62-363.

Na figura acima, nota-se que, após a execução de uma instrução ser completada, verifica-se se há alguma interrupção enfileirada pendente a ser executada. Nesse caso, ela entraria como um processo normal, ou seja, as instruções vinculadas ao tratamento da interrupção entrariam no fluxo do *pipeline*.
Quando um fluxo de execução é adicionado ao *pipeline*, o que acontece quando alguma outra interrupção ocorre? Quando esse cenário ocorre, temos duas maneiras de realizar o tratamento: na primeira forma, assim que o tratamento da interrupção for iniciado, poderemos desabilitar novas interrupções ou permitir que as próprias interrupções sejam interrompidas para o atendimento de outras. A figura abaixo mostra a diferença entre esses dois enfoques.

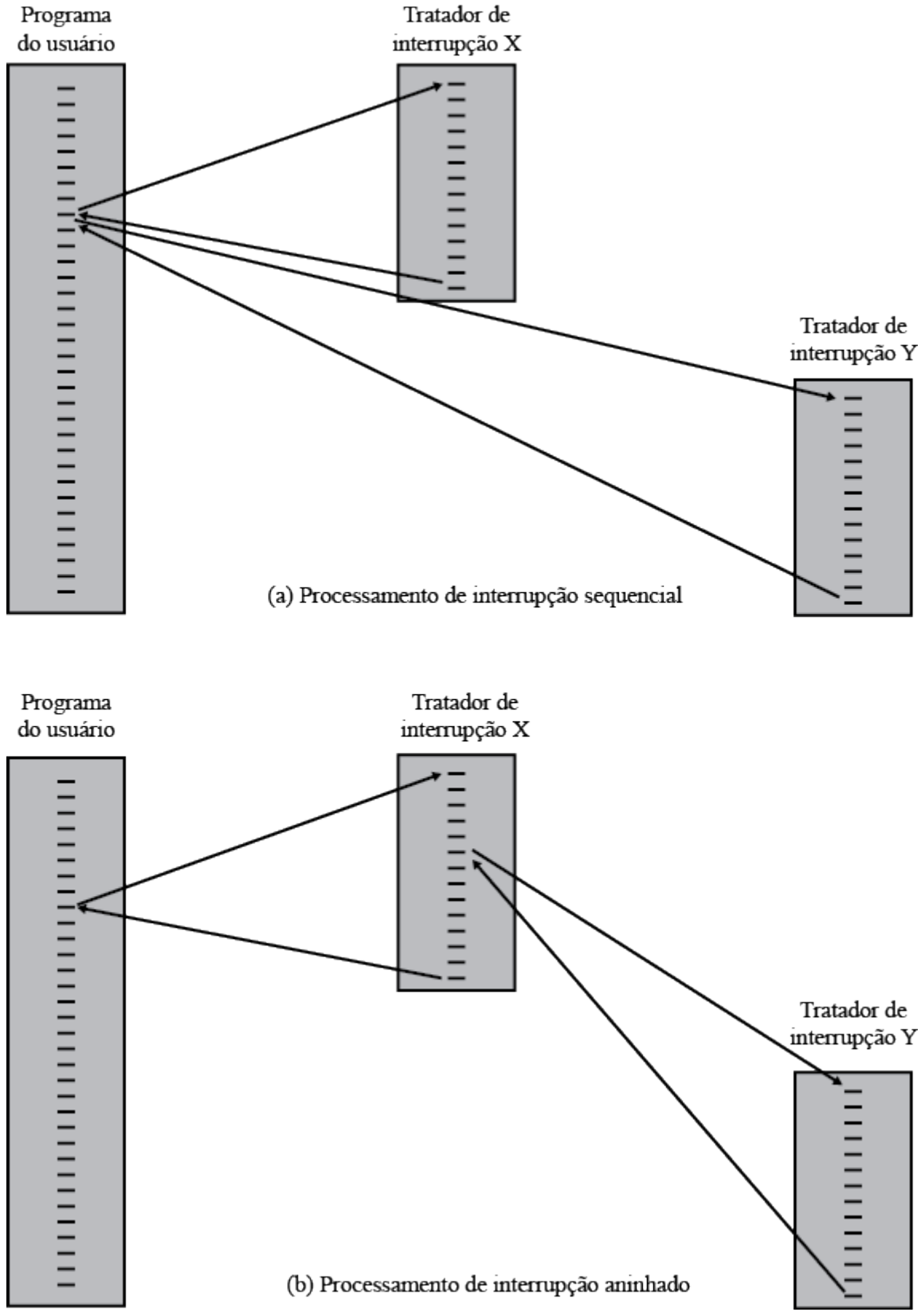

Figura 5 - Tratamento de interrupções com a desabilitação de novas interrupções (a) e tratamento de novas interrupções de forma aninhada (b).
Fonte: STALLINGS, 2010, p. 65.

Em relação à figura acima, temos em (a) o tratamento de interrupções sequencialmente. Nesse tipo de enfoque, ocorre a desabilitação de interrupções para que as novas que ocorrerem sejam apenas enfileiradas, de modo que o tratamento ocorra somente após o término da atual. Por sua vez, em (b), no tratamento aninhado, uma interrupção poderá ser interrompida para que seja tratada uma com maior prioridade.
Em ambos os casos (sequencial ou aninhado), ocorrerá a interrupção da execução dos processos do usuário para que as interrupções sejam tratadas pelo *pipeline*.

2.3.2 Fluxo de dados e estratégia de pipeline

Até o momento, por vários momentos, citamos o termo *pipeline,* porém sem nos aprofundarmos. Você deve estar se perguntando: como o *pipeline* consegue, efetivamente, agir em um ambiente com interrupções, desvios e demais fatos que podem degradar a eficiência computacional? Antes de respondermos a esse questionamento, vamos mencionar, antes, um fato relacionado com o fluxo de dados.
Para se executar uma instrução, tem-se que coletar as informações a serem manipuladas. Nem sempre uma informação é buscada de forma imediata, ou seja, às vezes é necessária mais de uma etapa para fazê-lo.
Pois bem, vamos relembrar alguns registradores de controle e estado do processador: PC, MBR e MAR. Você deve estar lembrando que o interfaceamento entre o processador e o sistema de memória é realizado pelos registradores MBR e MAR. Então, como acessar a memória para que seja buscada a instrução na posição apontada pelo registrador PC? A figura abaixo ilustra, em (a), esse processo.

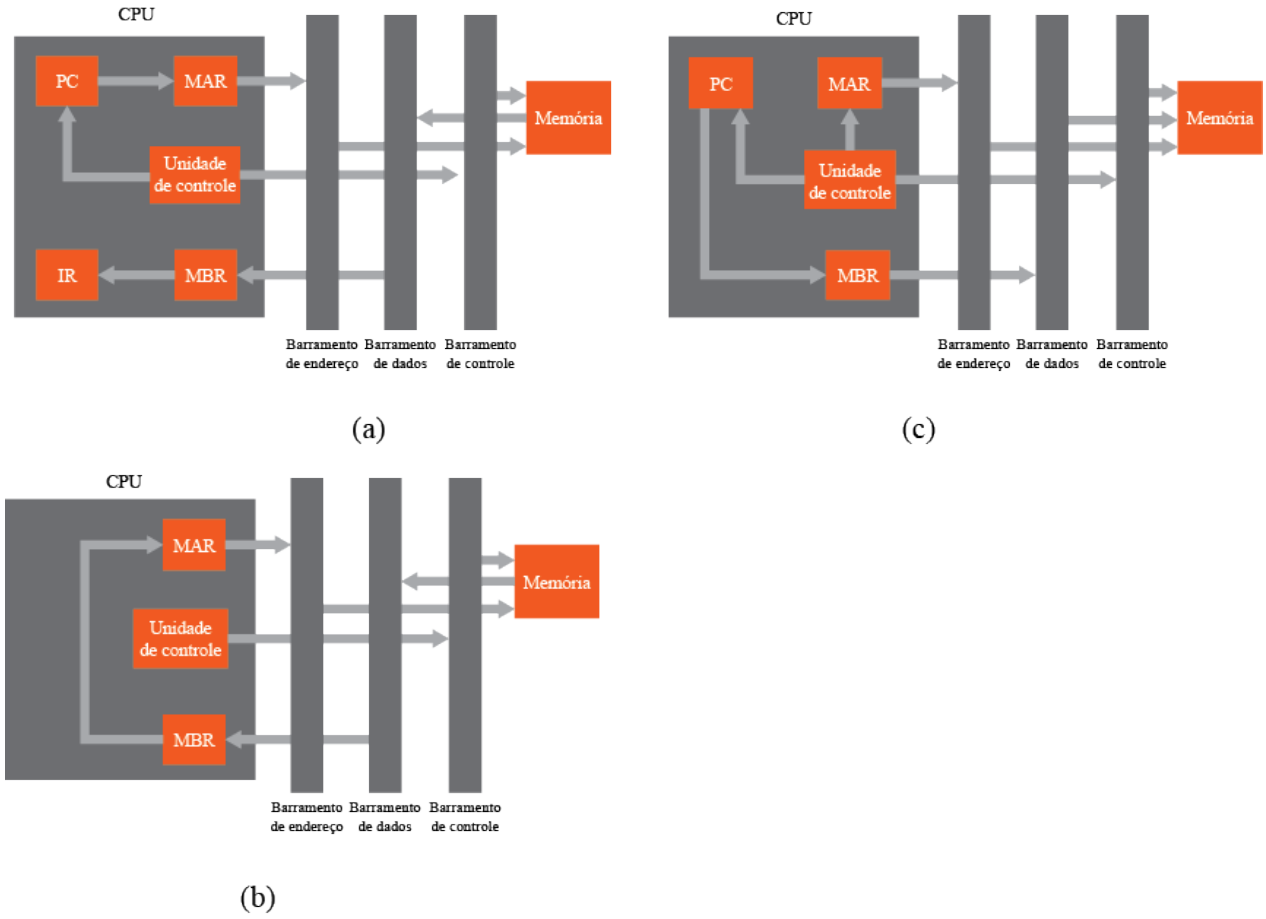

Figura 6 - Fluxo de dados na busca da instrução (a), na busca de operandos usando endereçamento indireto (b) e no ciclo de interrupção (c).
Fonte: STALLINGS, 2010, p. 363-364.

Na figura acima, temos, em (a), a representação do fluxo de dados durante o processo de busca da instrução. Nota-se que, para que a instrução seja buscada junto à memória, o registrador MAR (*memory address register*) deverá receber o valor contido em PC (*program counter*). O valor de retorno do sistema de memória é salvo no registrador MBR (*memory buffer register*), que, no caso, contém a palavra referente à instrução. A finalização do fluxo ocorre no momento em que o valor de MBR é passado ao registrador IR (*instruction register*) – mesmo instante em que ocorre o incremento do PC para que, no próximo ciclo, seja buscada, a princípio, a instrução subsequente.

Ainda em relação à figura acima, o item (b) ilustra o fluxo de dados no momento da etapa de busca de operandos por ocasião do endereçamento indireto de memória. O valor buscado no sistema de memória (alocado em MBR) representa, na verdade, o endereço no qual se deseja o valor a ser manipulado. Diante disso, um novo ciclo é necessário para providenciar essa busca, sendo, então, para isso, necessária a transferência do valor inserido no MBR para o MAR.

Por fim, temos, em (c), o fluxo de dados inerente ao tratamento de uma interrupção. A transferência do valor de PC para o MBR relaciona-se com o salvamento de contexto, para que o ponto de processamento seja restaurado após a finalização da execução do código da interrupção. Sendo assim, o valor de PC será salvo em um segmento de pilha na memória, cujo ponteiro é passado pela unidade de controle ao MAR.

VOCÊ QUER LER?

Quando se fala sobre fluxo de dados, correlacionando-o com o fluxo de instruções, vem à mente a Taxonomia de Flynn. Essa classificação abrange quatro tipos de modelos computacionais: SISD (*Single Instruction, Single Data*), SIMD (*Single Instruction Multiple Data*), MISD (*Multiple Instruction, Single Data*) e MIMD (*Multiple Instruction, Multiple Data*). Saiba mais no artigo (CARVALHO, 2017): <https://pt.linkedin.com/pulse/o-processamento-paralelo-e-taxonomia-de-flynn-carlos-eduardo>.

Mas, cabe a questão, qual a relação do fluxo de dados com o *pipeline*? A resposta reside no fato de que alterações no tempo de execução (havendo ou não a necessidade de ciclos extras de *clock* para a finalização de uma etapa) e de que alterações no chaveamento de comunicação entre os módulos alterará o modo de processamento e, consequentemente, impactará sobre o *pipeline*.

VOCÊ QUER LER?

Um dos principais algoritmos para a manipulação da execução de instruções envolvendo *pipeline* é o algoritmo de Tomasulo, criado por Robert Tomasulo (projetista da empresa americana IBM – International Business Machines), em 1967. O algoritmo serve de base para várias implementações utilizadas nos computadores atuais (FERREIRA, 2005). Para saber mais leia o artigo: <http://www.ic.unicamp.br/~rodolfo/Cursos/mo401/2s2005/Trabalho/049239-tomasulo.pdf>.

No modelo de sistema computacional superescalar, as instruções poderão ser despachadas fora de ordem caso haja disponibilidade de *hardware* e de dados para a sua execução. Porém, nesse caso, além do fluxo de dados, o *pipeline* também precisará se preocupar com outros detalhes que afetarão o seu desempenho computacional. Tais detalhes são chamados como conflitos ou riscos (*hazards*). Os *hazards* podem ser classificados como estruturais, de dados ou de controle (PATTERSON, 2014).

VOCÊ QUER VER?

No vídeo do professor Ricardo Santos (2015), da UFMS (Universidade Federal de Mato Grosso do Sul), você pode saber mais sobre o *pipeline* e seus *hazards*, acesse a aula disponível em < https://www.youtube.com/watch?v=9DOVl4aNYa0>.

Nos *hazards* estruturais, o *pipeline* é penalizado pela inexistência de recursos de *hardware* para que possa executar uma instrução. Como exemplo de *hazard* estrutural podemos citar o caso do *pipeline* que tenta buscar um certo operando, porém não é possível pois o sistema de memória está atendendo a gravação de resultado de uma instrução anterior.

Já os *hazards* de dados podem ser classificados em três categorias (PATTERSON, 2014):

- **leitura após escrita**: no exemplo, (i) $R_3 = R_1 + R_2$; (ii) $R_2 = R_4 + R_5$, temos um *hazard* do tipo leitura após escrita, pois, caso a linha (ii) seja finalizada antes da carga de R_2 na linha (i), teremos a manipulação de um valor errôneo na produção de R_3;
- **escrita após leitura**: ocorre quando a produção da informação pela instrução **i** ocorre após a ocorrência da leitura pela instrução **j** (sendo **i** < **j**);
- **escrita após escrita**: quando a instrução k modifica o valor do resultado da instrução **i**, porém existe uma instrução intermediária **j** que deveria ler o valor produzido por **i** (com **i** ocorrendo antes do **j**, que ocorre antes de **k**).

Para melhor gerenciar os *hazards* estruturais e de dados, pode-se adotar uma solução baseada em Scoreboard ou Algoritmo de Tomasulo e suas variações, e renomeamento de registradores (PATTERSON, 2014). Enfim, são inúmeros os projetos e estudos para que seja permissível a adoção de máquinas cada vez mais eficientes.

Por fim, os *hazards* de controle relacionam-se às instruções de desvio. Para evitar a paralização do *pipeline* enquanto a expressão lógica do desvio estiver em processamento, o processador tentará prever qual alternativa tomar. Para tanto, pode-se utilizar uma tabela ordenada pelos LSBs (*less signitificant bit – bits* menos significativos) contendo a resultado da expressão lógica da iteração anterior. Com esse valor, o processador antecipará a execução da próxima linha de instrução, efetivando o resultado apenas após a confirmação do caminho certo que deveria ser tomado (caso tenha tomado o caminho errado, descarta-se o resultado).

Assim como os *hazards* estruturais e de dados, existem, também, mecanismos para atenuar o impacto dos *hazards* de controle, dentre os quais podemos mencionar a previsão dinâmica de desvios.

2.4. Aritmética do computador

Tudo o que é armazenado e manipulado no computador consiste em informações numéricas. Mais especificamente, esses números seguem um modelo de representação **binária**. Como você deve estar acostumado em programação, existem os tipos de dados (por exemplo, em C/C++: *char, int, float* etc.) que

possuem uma faixa de valores derivada de seu tamanho (em *bits*). Por exemplo, caso estejamos manipulando uma variável do tipo char, significa que poderemos representar valores de -128 a 127, ou seja, metade de sua capacidade de representação consistirá de valores positivos e metade de valores negativos. Mas por que esse valor? Nesse caso, quando se manipula tipos cuja largura é de 8 *bits*, tem-se 256 valores possíveis ($2^8 = 256$). No entanto, cabe o questionamento, como representar um valor em binário?

VOCÊ O CONHECE?

Apesar do conceito de lógica binária nos acompanhar desde os primórdios dos tempos (somos seres dicotômicos), apenas no século XIX é que o pensamento binário foi formalizado através da Álgebra Booleana. O idealizador dela foi o matemático e filósofo britânico George Boole. Leia mais no artigo (PIROPO, 2012): <http://www.techtudo.com.br/artigos/noticia/2012/08/uma-algebra-diferente.html>.

Todo valor, em qualquer base numérica, poderá ser fatorado. Por exemplo, o valor $876_{(10)}$ poderá ser fatorado como:

$876_{(10)} = 800 + 70 + 6$

$= 8*10^2 + 7*10^1 + 6*10^0$.

No caso acima, o elemento que está sendo exponenciado, corresponde à própria base. Sendo assim, a fatoração de valores binários não é diferente, ou seja, o processo é o mesmo em relação à base decimal, porém, ao invés de trabalharmos com a base 10, trabalharemos com a base 2. Por exemplo, o valor:

$1011_{(2)} = 1*2^3 + 0*2^2+1*2^1 + 1*2^0$

$= 8 + 0 + 2 + 1 = 11_{(10)}$.

O processo apresentado acima corresponde à conversão de um valor na base binária para a base decimal.

Você pode notar que ao ser realizada a exponenciação da base pelo expoente que representa o grau de significância do termo, em função de sua localização, obtém-se: $[2^3; 2^2; 2^1;2^0 = 8; 4; 2; 1]$ (WEBER, 2012). Dessa combinação, originou-se o nome do modelo de representação binária que utilizamos: BCD 8421 (*binary coded decimal*).

Ainda assim como chegamos nos fatores que deverão ser multiplicados à base exponenciada? Simplesmente fazendo divisões sucessivas pela base e coletando os restos obtidos. As divisões pela base serão realizadas enquanto pudermos dividir pela base. Na última divisão, coleta-se o quociente. Esse processo de divisões sucessivas representa a forma de como convertemos um valor decimal em binário (realizando divisões sucessivas por 2).

Mas, voltando à programação em C/C++, o que acontece quando utilizamos o modificador de tipo *unsigned*? Esse modificador faz com que a variável represente apenas valores positivos. No caso do *unsigned char*, todos os 256 valores possíveis serão positivos (0 a 255). Isso ocorre porque, na verdade, na representação que permite trabalharmos com valores negativos, gasta-se 1 *bit* para representar o sinal. O *bit* mais significativo (MSB – *most significant bit*) representa o sinal. Caso o MSB seja zero, então o valor representado é positivo, caso contrário, um valor negativo é representado. Diante desse fato, tem-se 2^7 valores negativos (-128 a -1) e 2^7 valores positivos (0 a 127).

Essa ideia da possibilidade de se operar valores negativos e positivos remete a uma forma de representação denominada como complemento 2. Para se converter um valor de positivo para negativo ou negativo para positivo, usa-se o mesmo algoritmo constituído por duas fases: a) inverte-se todos os *bits* da palavra; b) soma-se o valor um ao valor com os *bits* invertidos.
Sobre os valores binários representados em BCD 8421 ou em complemento dois, poderão ser realizadas operações aritméticas de forma análoga ao processo empregado em valores com representação decimal.
Além dos valores inteiros, existe também a representação dos números reais, conhecidos, computacionalmente, por ponto flutuante (*float*, em C/C++). O complemento dois é padronizado pelo IEEE sob número 754, que permite armazenar um valor de maneira semelhante à notação científica.
Na nossa notação científica, temos, por exemplo, o número 532,41 representado na forma 5.3241E2, ou seja, $5.3241 * 10^2$. Nesse exemplo, temos, como mantissa o valor 5.3241 e, como expoente, o valor dois.
A responsabilidade de execução das instruções aritméticas, assim como as de operações lógicas, são de responsabilidade da Unidade Lógica e Aritmética.

2.4.1 Unidade Lógica e Aritmética (ULA)

A unidade lógica aritmética (ULA ou ALU – *Arithmetic-Logic Unit*) é o módulo de *hardware* integrante do computador, cuja função é realizar operações aritméticas e lógicas.
De forma simplista, podemos falar que a ULA realiza interfaceamento com o banco de registradores, de modo que ela possa coletar os valores a serem manipulados pelas operações aritméticas (como soma, multiplicação sobre números inteiros ou ponto flutuante) e pelas operações lógicas (como operação AND, deslocamentos etc.).

VOCÊ SABIA?

Você sabia que, em C/C++, você poderá fazer operações *bit*-a-*bit*? Por exemplo, faça um programa e teste a diferença no resultado entre as atribuições usando operações lógicas (exemplo: C = A && B; D = E || F) e operações *bit*-a-*bit* (exemplo: C = A & B; D = E | F). Você sabia, também, que poderá realizar divisões, por exemplo, por dois usando operação lógica? Experimente substituir: A = A / 2 pelo comando: A = A >> 1. O operador lógico >> denota deslocamento, no caso, deslocamento para a direita uma vez.

Como resultados, a ULA fornece o valor da operação propriamente dito e, também, é responsável por gerar os *bits* do registrador de *flags* (indicando, por exemplo, resultado negativo, nulo e *overflow*).
Para uma melhor compreensão da manipulação aritmética, você pode consultar o capítulo 9 de (STALLINGS, 2010).

2.4.2 Unidade de Controle (UC)

A unidade de controle do processador tem por objetivo gerenciar todo o fluxo de execução das instruções. Esse gerenciamento é conseguido pelo envio de sinais de controle aos módulos da CPU, ativando-os quando necessário e pelo roteamento das informações de forma a encaminhá-las de forma apropriada.
A ULA poderá executar instruções da forma reg op reg (quando os dois operandos estão armazenados em registradores) ou reg op constante (a instrução representa uma operação entre o valor do registrador e uma constante). Você pode já imaginar que a UC, além de ativar a ULA, também deverá selecionar os operandos que

incidirão no processamento. A figura abaixo, extraída de (PATTERSON, 1998), ilustra o interfaceamento entre a UC e a ULA.

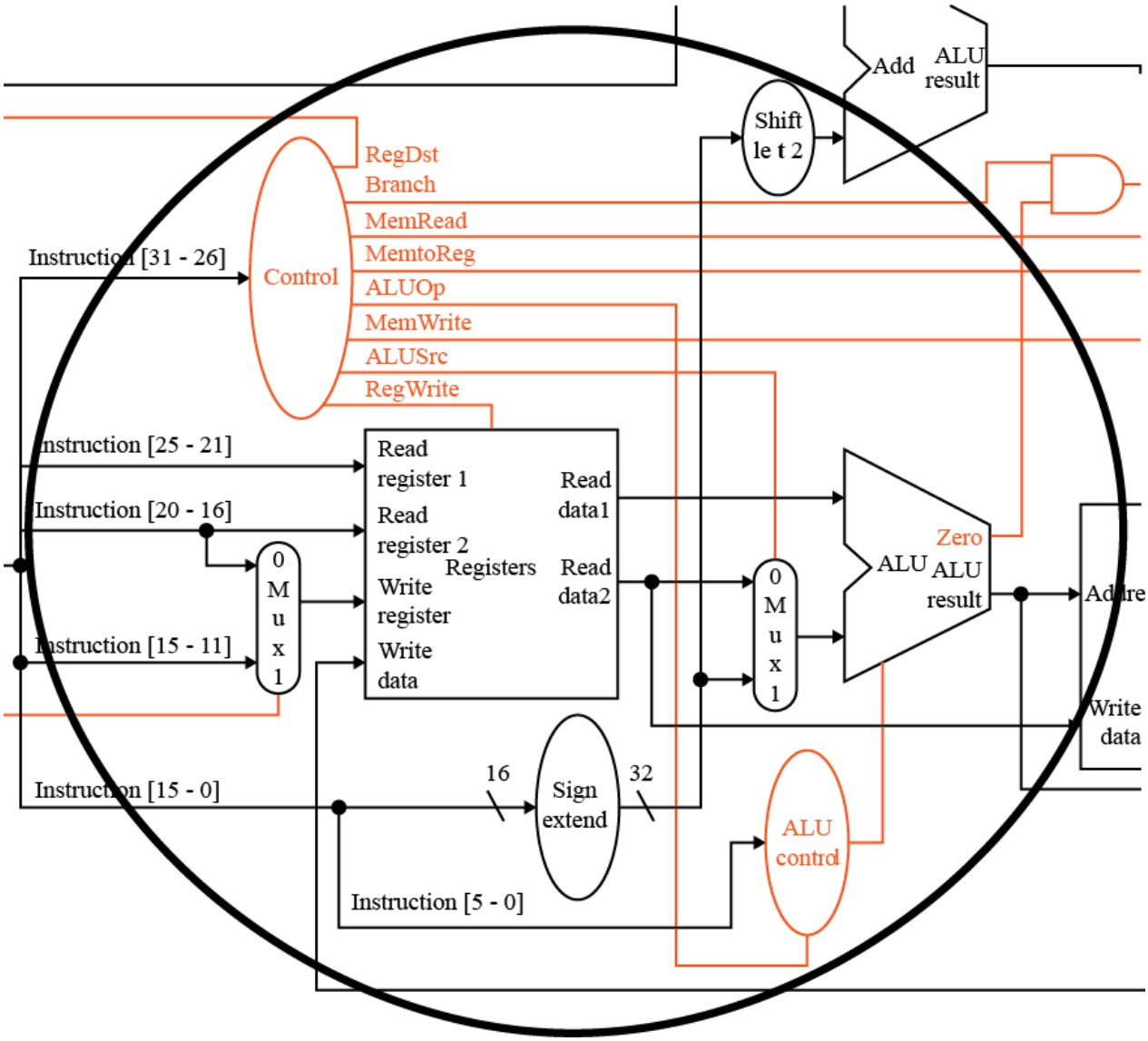

Figura 7 - Datapath do processador MIPS destacando-se a interface entre unidade de controle e a ALU e o chaveamento no fluxo de dados a ser encaminhado à ALU.
Fonte: Adaptada de PATTERSON, 1998, p. 209.

Na figura acima, nota-se que o módulo de controle, responsável pela decodificação das instruções, produz, para a ALU, dois sinais de controle: ALUop (operação de ALU) e ALUsrc (ALU *source* – ALU fonte). A seleção do dado é realizada por intermédio de um seletor denominado como MUX (também conhecido como multiplexador ou multiplex). O outro operando é coletado, sempre, do banco de registradores. O sinal "Read data 1" representa o conteúdo do registrador apontado por Read Register 1.

Por fim, tem-se, com saídas da ULA, o valor propriamente dito e, também, sinais do registrador de *flags* para indicar o status de seu processamento – no caso da figura, o *bit* zero. Ainda em relação à figura anterior, temos, para facilitar o processo de decodificação e demais ações, a quebra da palavra de instrução em vários pedaços. Por exemplo, o *opcode* das instruções corresponde aos *bits* 31 ao 26.

Você poderá notar, ainda na figura acima, que o *datapath* em questão é de um processador típico RISC. Mas como chegou-se à tal conclusão? Observando-se as entradas da ULA, percebe-se que operações do tipo soma (ADD)

manipulam três registradores (dois fontes e um destino), as operações de ULA são da forma reg_destino, reg_fonte_1, reg_fonte2.
Para finalizar, podemos falar que o funcionamento do computador, em sua essência, é algo básico – baseado em decodificações, envio de sinais de controle, roteamento de informações. O que o torna complexo são as técnicas cada dia mais agressivas para conseguir, continuamente, aumentar o poderio de processamento e de sua performance computacional. Essas técnicas são baseadas no *pipeline* e na superescalaridade. Um outro complicante no quesito de complexidade é representado pelo processamento paralelo cada vez mais atuante e presente. Temos vários exemplos e níveis de paralelismo: nível de programação, nível de *pipeline* (pseudoparalelismo), nível de superescalaridade e, por fim, nível de múltiplos núcleos (*cores* – físicos e virtuais) de paralelismo.

Síntese

Você chegou ao final do capítulo que aborda conceitos inerentes à forma da representação e do armazenamento de informações dentro do processador para que sejam executados. Novamente, foi necessário abordar os conceitos de arquitetura e organização – conceitos que ora se mesclam e ora são totalmente distintos.
A partir dos pontos que abordamos neste capítulo, esperamos que você se sinta confortável e motivado a prosseguir os estudos relacionados ao funcionamento do computador e a aplicar os conhecimentos adquiridos em seu dia a dia – seja em atividades de consultoria, na implementação de sistemas ou, ainda, em seu cotidiano pessoal. Esperamos, também, que você, a partir das informações até aqui apresentadas, vislumbre soluções, cada vez mais eficientes e "elegantes" para a implementação de sistemas computacionais.
Neste capítulo, você teve a oportunidade de:

- identificar os elementos constantes da arquitetura de computadores de modo a poder compreender as funcionalidades inerentes aos módulos pertencentes à máquina e, depois, de forma mais focada, pertencentes ao processador;
- compreender a forma de organização interna do processador, identificando a organização dos registradores, e reconhecer as funcionalidades dos registradores assim como aplicá-los no contexto de tradução de código para linguagem de máquina;
- inferir sobre o processo de execução de instruções, identificando os pontos de controle do sistema computacional. Compreender e aplicar os conceitos relacionados ao processo de tradução de uma linguagem de alto nível para o *assembly* (linguagem de máquina);
- entender a aritmética computacional assim como identificar os elementos de interfaceamento entre a ULA e a Unidade de Controle.

Bibliografia

CARVALHO, C. E. M. **O Processamento Paralelo e a Taxonomia de Flynn**. Portal LinkedIn, publicado em 13 de março de 2017. Disponível em <https://pt.linkedin.com/pulse/o-processamento-paralelo-e-taxonomia-de-flynn-carlos-eduardo>. Acesso em: 27/04/2018.
LABWORKS; EMBEDDED. **Linux Device Drivers**. Disponível em: <https://e-labworks.com/treinamentos>; <https://e-labworks.com/treinamentos/drivers/source>. Acesso em: 26/04/2018.
FERREIRA, C. D. **Algoritmo de Tomasulo**. Publicado em 03 de novembro de 2005. Disponível em: <http://www.ic.unicamp.br/~rodolfo/Cursos/mo401/2s2005/Trabalho/049239-tomasulo.pdf>. Acesso em: 25/04/2018.

PATTERSON, D. A.; HENNESSY, J. L. **Organização e Projeto de Computadores:** A interface *hardware*/*software*. 2. ed. Rio de Janeiro: Morgan Kaufmann Publishers, 1998.

PATTERSON, D. A.; HENNESSY, J. L. **Arquitetura de Computadores:** Uma abordagem quantitativa. 5. ed. Rio de Janeiro: Elsevier, 2014.

PIROPO, B. **Uma Álgebra Diferente**. Portal TechTudo, publicado em 02/08/2012. Disponível em: <http://www.techtudo.com.br/artigos/noticia/2012/08/uma-algebra-diferente.html>. Acesso em: 26/04/2018.

SANTOS, R. **Manipulação de Hazards de Dados**. Canal Ricardo Santos, YouTube, publicado em 09 de abril de 2015. Disponível em: <https://www.youtube.com/watch?v=9DOVl4aNYa0>. Acesso em: 30/04/2018.

STALLINGS, W. **Arquitetura e Organização de Computadores**. 8. ed. São Paulo. Pearson Pratice Hall, 2010. Disponível na Biblioteca Virtual: <https://laureatebrasil.blackboard.com/webapps/blackboard/content/listContent.jsp?course_id=_198689_1&content_id=_4122211_1&mode=reset>. Acesso em: 21/05/2018.

TANENBAUM, A. S. **Organização Estruturada de Computadores**. 6. ed. São Paulo: Pearson Pratice Hall, 2013. Disponível na Biblioteca Virtual: <https://laureatebrasil.blackboard.com/webapps/blackboard/content/listContent.jsp?course_id=_198689_1&content_id=_4122211_1&mode=reset>. Acesso em: 21/05/2018.

TELES, Bruno. **O Processador Intel Core i7**. Disponível em <http://www.ic.unicamp.br/~ducatte/mo401/1s2009/T2/042348-t2.pdf>. Acesso em: 13/06/2018.

WEBER, R. F.; **Fundamentos de arquitetura de computadores**. 4ª ed. Porto Alegre: Bookman, 2012. Disponível na Biblioteca Virtual: <https://integrada.minhabiblioteca.com.br/#/books/9788540701434>. Acesso em 12/06/2018.