Programando o computador IAS

Edson Borin e Rafael Auler

8 de agosto de 2012

1 Introdução

O computador IAS foi um dos primeiros computadores a implementar o "conceito do programa armazenado". Neste paradigma, o programa é armazenado na memória do computador juntamente com os dados. Desta forma, a geração e a carga de programas pode ser feita de forma automatizada, pelo próprio computador. O IAS foi desenvolvido no Instituto de Estudos Avançados de Princeton e sua construção foi liderada por John von Neumann, um matemático que contribuiu bastante para o campo da ciência da computação.

Este texto apresenta uma breve descrição histórica do surgimento do IAS, a organização e o funcionamento do computador IAS assim como suas instruções e uma breve introdução à sua programação.

A Seção 2 apresenta o cenário histórico na época do surgimento do IAS e a motivação para construção do IAS. A Seção 3 apresenta a organização geral e a Seção 4 descreve o funcionamento do computador IAS. A Seção 5 descreve as instruções e apresenta uma breve introdução à programação do computador IAS e a Seção 6 apresenta uma linguagem de montagem para o computador IAS.

2 O Surgimento do IAS

Durante a segunda guerra mundial, o laboratório de pesquisa de balística do exército dos Estados Unidos da América financiou a construção do ENIAC, um computador eletrônico para auxiliar na computação de tabelas de disparo de artilharia. Terminado em 1946, o ENIAC não ficou pronto a tempo de ser utilizado na guerra, mas por ser um computador programável, foi utilizado por muitos anos para outros propósitos, até sua desativação em 1955. O ENIAC é considerado por muitos[1] o primeiro computador eletrônico de propósito geral.

Durante o projeto do ENIAC, seus desenvolvedores observaram que diversos items do projeto original poderiam ser melhorados. No entanto, para manter o cronograma do projeto, as novas ideias não foram incorporadas ao projeto do ENIAC. Com o intuito de desenvolver um computador melhor, que incorporasse as melhorias propostas pela equipe, os desenvolvedores do ENIAC começaram a discutir o projeto de um novo computador, o EDVAC.

A programação do ENIAC era feita através de cabos e interruptores, o que tornava a programação um processo lento e tedioso. Uma das melhoras que não puderam ser incorporadas ao projeto original do ENIAC foi o "conceito do programa armazenado", em que o programa é armazenado na memória, junto com os dados. A invenção deste conceito, presente em todos os computadores de propósito geral atuais, foi creditada ao matemático John von Neumann, membro do instituto de estudos avançados de princeton, o IAS, que trabalhava como consultor no projeto do EDVAC. Durante este tempo, ele escreveu um rascunho com anotações das reuniões do projeto, que foi distribuído por um dos membros da equipe para diversas universidades e órgãos do governo. Como o manuscrito tinha apenas o nome de John von Neumann, ele foi creditado como sendo o inventor do conceito do programa armazenado.

Após a sua participação no projeto do EDVAC, von Neumann voltou para o instituto de estudos avançados de Princeton e dirigiu o desenvolvimento de um computador que tinha como propósito servir de modelo para

[1] Não há um consenso sobre qual foi o primeiro computador eletrônico de propósito geral.

a construção de outros computadores no país. Este computador, o IAS, começou a ser desenvolvido em 1945 e ficou pronto em 1951.

De acordo com Willis Ware, em março de 1953, o computador IAS já havia executado um grande número de programas para calcular conjuntos de equações diferenciais parciais não lineares para metereologistas interessados em previsão do tempo. O IAS também foi utilizado para resolver problemas relacionados a teoria dos números e astronomia.

Diversos computadores tiveram seus projetos derivados ou inspirados no projeto do IAS. Dentre eles: ILLIAC I, JOHNIAC, MANIAC, SILLIAC e outros.

3 Organização do IAS

A estrutura do computador IAS possui quatro módulos principais: A memória principal, a unidade de controle, a unidade lógica e aritmética e o módulo de entrada e saída. A Figura 1 mostra a organização geral do IAS.

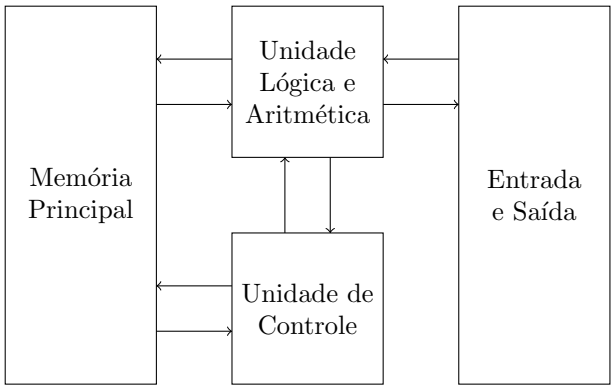


Figura 1: Estrutura geral do IAS

A memória principal é utilizada para armazenar os dados e as instruções ("conceito do programa armazenado"). A unidade lógica e aritmética realiza operações com dados binários. A unidade de controle é responsável por buscar e executar, uma a uma, as instruções armazenadas na memória principal e, por fim, o módulo de entrada e saída (E/S) permite a inserção e a recuperação de dados do computador.

3.1 Memória Principal

A memória principal do IAS possui 1024 palavras de 40 *bits*. Cada palavra está associada a um endereço distinto, um número que varia de 0 até 1023. A Figura 2 ilustra a organização da memória do IAS. Neste exemplo, a primeira palavra da memória, no endereço `0` (zero), contém o valor `01 06 90 50 67` e a última palavra da memória, no endereço `1023`, contém o valor `20 1A F9 10 AB`.

A **memória principal** do IAS realiza **duas operações básicas: escrita e leitura de valores nas palavras da memória**. Tanto a escrita quanto a leitura são controladas (iniciadas) pela unidade de controle (UC) do IAS. A UC pode solicitar à memória principal a escrita de um valor em um determinado endereço da memória. Nesta operação, a memória recebe como entrada o endereço e o valor a ser escrito. A UC também pode solicitar à memória a leitura de um dado armazenado em uma palavra. Neste caso, a UC envia o endereço da palavra que contém o dado a ser lido e a memória retorna o valor lido.

Cada palavra da memória principal do IAS pode armazenar um número de 40 *bits* ou duas instruções de 20 *bits*[2]. Os números são representados em complemento de dois. As instruções, apresentadas em detalhe

[2]Os números de 40 *bits* fazem parte do conjunto de dados do programa, em contraste às instruções, que compõem a lógica de um programa. O uso da memória para esses dois propósitos foi a novidade introduzida com o conceito do programa armazenado.

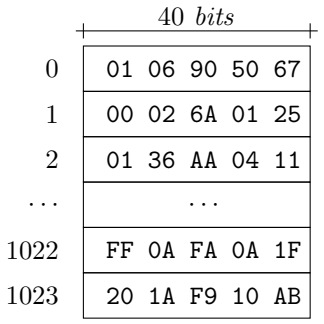
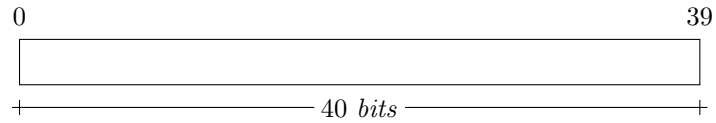
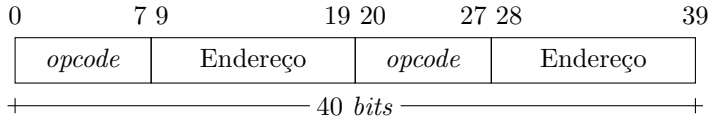


Figura 2: Organização da memória do computador IAS

na Seção 5, possuem dois campos, o "código da operação", de 8 *bits*, e o "endereço", de 12 *bits*. A Figura 3 ilustra a representação de números e instruções em uma palavra de memória.



(a) Um número de 40 *bits* em uma palavra da memória



(b) Duas instruções de 20 *bits* em uma palavra da memória

Figura 3: Representação de (a) números e (b) instruções em palavras da memória principal do IAS

3.2 Unidade de Controle

A unidade de controle (UC) do IAS é reponsável por coordenar cada um dos módulos para que o computador execute as instruções armazenadas na memória. A unidade de controle dispõe de um circuito de controle e quatro "registradores" internos. Registradores são pequenas unidades de memória que se situam tipicamente dentro do processador utilizadas para armazenar valores temporários. A operação de escrita ou leitura de dados em registradores é mais rápida do que a escrita ou leitura de dados na memória principal. Entretanto, registradores são fabricados com uma tecnologia que demanda mais área e de maior custo. Tipicamente, associamos palavras da memória a um endereço, entretanto registradores são frequentemente associados a nomes. A unidade de controle do IAS possui os seguintes registradores:

- **PC**: o *Program Counter*, ou contador do programa, armazena um valor que representa o endereço da memória que possui o próximo par de instruções a serem executadas. No início, quando o computador é ligado, o conteúdo deste registrador é zerado para que a execução de instruções se inicie a partir do endereço zero da memória.
- **MAR**: o *Memory Address Register*, ou registrador de endereço da memória, armazena um valor que representa um endereço de uma palavra da memória. Este endereço será lido pela memória durante a operação de leitura ou escrita de dados.

- **IR**: o *Instruction Register*, ou registrador de instrução, armazena a instrução que está sendo executada no momento. O circuito de controle da unidade de controle lê e interpreta os *bits* deste registrador e envia sinais de controle para o resto do computador para coordenar a execução da instrução.
- **IBR**: o *Instruction Buffer Register* serve para armazenar temporariamente uma instrução. O IAS busca instruções da memória em pares - lembre-se de que uma palavra da memória (de 40 *bits*) contém duas instruções (de 20 *bits*). Dessa forma, quando o IAS busca um par de instruções, a primeira instrução é armazenada diretamente em `IR` e a segunda em `IBR`. Ao término da execução da primeira instrução (em `IR`), o computador move a segunda instrução (armazenada em `IBR`) para `IR` e a executa.

A Figura 4 ilustra a organização interna da unidade de controle. As setas indicam caminhos onde os dados podem trafegar. Por exemplo, a seta entre o registrador `IR` e o circuito de controle indica que os *bits* armazenados no registrador `IR` podem ser transferidos para os circuitos de controle.

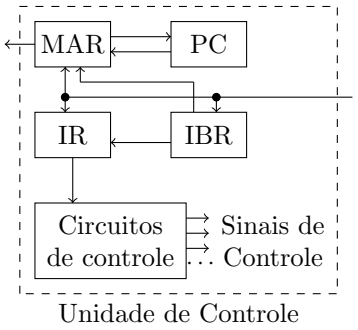


Figura 4: Organização da unidade de controle (UC) do IAS

3.3 Unidade Lógica e Aritmética

A unidade lógica e aritmética do IAS, ou ULA, é responsável por realizar operações aritméticas e lógicas nos dados armazenados no computador. No IAS, a ULA é composta por circuitos lógicos e aritméticos e três registradores, descritos a seguir:

- **MBR**: o *Memory Buffer Register*, ou registrador temporário da memória, é um registrador utilizado para armazenar temporariamente os dados que foram lidos da memória ou dados que serão escritos na memória. Para escrever um dado na memória, o computador deve colocar o dado no registrador `MBR`, o endereço da palavra na qual o dado deve ser armazenado no registrador `MAR` e, por fim, enviar sinais de controle para a memória realizar a operação de escrita. Assim sendo, os registradores `MAR` e `MBR`, juntamente com os sinais de controle enviados pela unidade de controle, formam a interface da memória com o restante do computador.
- **AC** e **MQ**: O *Accumulator*, ou acumulador, e o *Multiplier Quotient*, ou quociente de multiplicação, são registradores temporários utilizados para armazenar operandos e resultados de operações lógicas e aritméticas. Por exemplo, a instrução que realiza a soma de dois números (`ADD`) soma o valor armazenado no registrador `AC` com um valor armazenado na memória e grava o resultado da operação no registrador `AC`.

A Figura 5 mostra a organização detalhada da unidade lógica e aritmética e sua comunicação com o restante do computador.

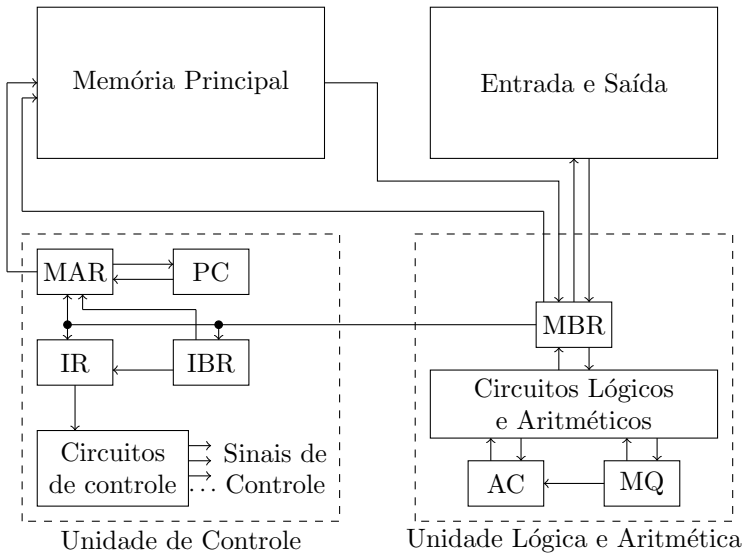


Figura 5: Organização detalhada da unidade lógica e aritmética e sua comunicação com o restante do computador

4 Operação do IAS

O computador IAS executa programas armazenados na memória principal do computador. Os programas são implementados como uma sequência de instruções e o IAS executa estas instruções uma a uma. A execução das instruções pode envolver o acesso à memória para leitura ou escrita de dados ou instruções[3]. Desta forma, antes de discutirmos o processo de execução de instruções, vamos estudar em mais detalhes como o computador escreve e lê dados da memória principal.

4.1 Operações de Escrita e Leitura na Memória

Para ler um dado armazenado em uma palavra da memória é necessário especificar o endereço da palavra na memória. No IAS, este endereço deve ser gravado no registrador `MAR` (registrador de endereço da memória) antes de solicitar a leitura do dado à memória. Ao final da operação de leitura, a memória grava o valor lido no registrador `MBR` (registrador temporário da memória). Os seguintes passos são executados para se ler um dado da memória:

1. O endereço da palavra a ser lida é escrito no registrador `MAR`;
2. Os circuitos de controle da unidade de controle (UC) enviam um sinal de controle através de um canal de comunicação de controle[4] à memória principal, solicitando a leitura do dado;
3. A memória principal lê o endereço do registrador `MAR` através do canal de comunicação de endereços e, de posse do endereço, lê o valor armazenado da palavra de memória associada a este endereço;
4. Por fim, a memória principal grava o valor lido no registrador `MBR` através do canal de comunicação de dados.

A operação de escrita é parecida mas, neste caso, além do endereço em `MAR`, devemos colocar o valor a ser armazenado em `MBR`. Os seguintes passos são executados para se escrever um dado da memória.

[3]Note que instruções também podem ser escritas na memória

[4]O canal de comunicação é implementados geralmente por meio de fios e os sinais ou dados trafegam em forma de pulsos elétricos.

1. O endereço da palavra que armazenará o dado é escrito no registrador `MAR`;
2. O dado a ser armazenado é gravado no registrador `MBR`;
3. Os circuitos de controle da unidade de controle (UC) enviam um sinal de controle à memória principal, solicitando a escrita do dado;
4. A memória principal lê o endereço do registrador `MAR` através de canal de comunicação de dados, lê o dado do registrador `MBR` e armazena este valor na palavra de memória associada ao endereço lido de `MAR`;

A Figura 6 ilustra os canais de comunicação de dados (azul), endereços (vermelho) e sinais de controle (verde) utilizados na comunicação com a memória.

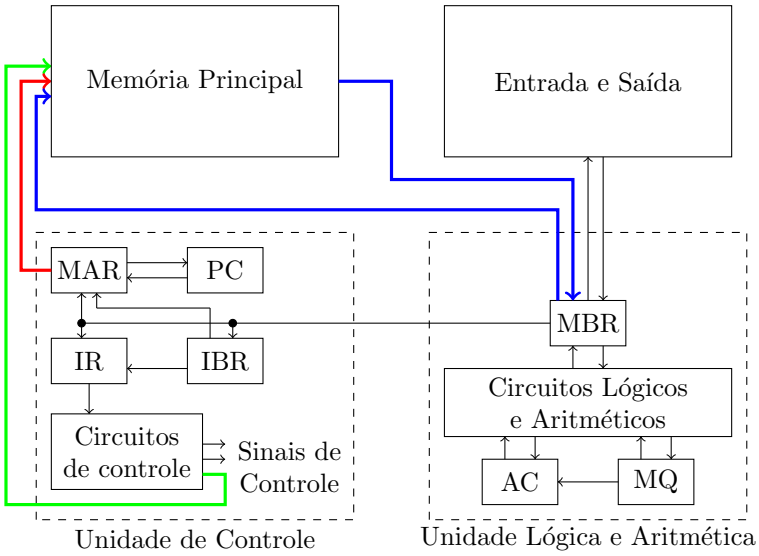


Figura 6: Canais de comunicação de dados (azul), endereços (vermelho) e sinais de controle (verde) na comunicação com a memória.

A próxima seção descreve o processo de execução de instruções no computador IAS.

4.2 Execução de Instruções

As instruções do IAS são executadas uma a uma através de uma sequência de operações orquestradas pela unidade de controle. Inicialmente, quando ligado, o computador executa a instrução à esquerda da primeira palavra da memória (endereço 0) e depois a instrução à direita da primeira palavra da memória. Em seguida, o computador prossegue executando a instrução à esquerda da segunda palavra da memória (endereço 1) e assim por diante, uma após a outra.

É válido lembrar que algumas instruções do computador podem "desviar o fluxo de execução" das instruções, fazendo com que a próxima instrução a ser executada não seja a instrução subsequente na memória. Mas antes de discutirmos instruções que podem desviar o fluxo de execução, vamos supor que as instruções são executadas em sequência, na ordem em que aparecem na memória.

A execução de uma instrução é realizada em dois ciclos: o "ciclo de busca" e o "ciclo de execução". O ciclo de busca consiste em buscar a instrução da memória (ou do registrador `IBR`) e armazenar no `IR`. O ciclo de execução, por sua vez, consiste em interpretar a instrução armazenada no registrador `IR` e realizar as operações necessárias para execução da mesma.

Ciclo de Busca No ciclo de busca, a instrução a ser executada é buscada e armazenada no registrador de instrução, ou `IR`. O registrador `PC` contém o endereço da próxima instrução que deve ser executada. Quando o computador é ligado, o registrador `PC` é iniciado com o valor zero, indicando que a execução deve ser iniciada a partir da instrução armazenada na primeira palavra da memória. Assim sendo, no início da execução, o computador utiliza o endereço armazenado em `PC` (zero neste caso) para buscar a instrução a ser executada na memória. Como a memória do IAS armazena palavras de 40 *bits*, a leitura do endereço zero da memória retornará duas instruções (cada instrução tem 20 *bits*). A fim de preservar a segunda instrução, o computador armazena a instrução à direita no registrador `IBR`, enquanto que a instrução à esquerda é transferida para o `IR`. Ao executar a próxima instrução, o computador transferirá o conteúdo do registrador `IBR` para o registrador `IR` e não precisará fazer a busca da instrução na memória. Dessa forma, o ciclo de busca pode ser dividido em duas partes: ciclo de busca de instruções à esquerda e ciclo de busca de instruções à direita. O ciclo de busca de instruções à esquerda consiste basicamente nos seguintes passos:

1. A UC move o endereço em `PC` para `MAR`;
2. A UC envia um sinal de controle para a memória fazer uma operação de leitura;
3. A memória lê a palavra de memória e transfere o conteúdo para o registrador `MBR`;
4. A UC copia a segunda metade (*bits* 20 a 39) do registrador `MBR` e salva no registrador `IBR`. Estes *bits* correspondem à instrução à direita da palavra de memória.
5. A UC copia os 8 *bits* à esquerda do registrador `MBR` para o registrador `IR`. Estes *bits* correspondem ao campo de operação da instrução à esquerda da palavra de memória.
6. A UC copia os 12 *bits* subsequentes ao campo de operação (*bits* 8 a 19) e os transfere para o registrador `MAR`. Estes *bits* correspondem ao campo endereço da instrução e devem estar no registrador `MAR` caso a instrução precise acessar a memória durante o ciclo de execução.
7. A UC incrementa o valor no registrador `PC`, indicando que o próximo par de instruções a ser lido da memória deve ser lido do endereço $\texttt{PC} + 1$.

A leitura da memória pode não ser necessária no ciclo de busca de instruções à direita, pois a instrução pode estar armazenada no registrador `IBR` caso a última instrução executada tenha sido a instrução à esquerda na mesma palavra de memória. Note, no entanto, que instruções que desviam o fluxo de controle podem fazer com que uma instrução à direita seja executada sem que a instrução à esquerda da mesma palavra de memória seja executada. Neste caso, o ciclo de execução da instrução à direita precisa buscar a instrução da memória, visto que a instrução presente no registrador `IBR` não corresponde à instrução a ser executada. Os seguintes passos descreve o ciclo de busca de instruções à direita:

1. Se a última instrução executada foi a instrução à esquerda (não houve desvio no fluxo de controle), então:
   (a) A UC copia os 8 *bits* à esquerda do registrador `IBR` para o registrador `IR`. Estes *bits* correspondem ao campo de operação da instrução armazenada em `IBR`.
   (b) A UC copia os 12 *bits* subsequentes ao campo de operação (*bits* 8 a 19) e os transfere para o registrador `MAR`. Estes *bits* correspondem ao campo endereço da instrução e devem estar no registrador `MAR` caso a instrução precise acessar a memória durante o ciclo de execução.
2. Senão:
   (a) A UC move o endereço em `PC` para `MAR`;
   (b) A UC envia um sinal de controle para a memória fazer uma operação de leitura;
   (c) A memória lê a palavra de memória e transfere o conteúdo para o registrador `MBR`;

(d) A UC copia os *bits* 20 a 27 do registrador `MBR` para o registrador `IR`. Estes *bits* correspondem ao campo de operação da instrução à direita da palavra de memória lida.

(e) A UC copia os 12 *bits* subsequentes ao campo de operação (*bits* 28 a 39) e os transfere para o registrador `MAR`. Estes *bits* correspondem ao campo endereço da instrução à direita da palavra de memória.

(f) A UC incrementa o valor no registrador `PC`, pois a instrução à esquerda não foi executada e o mesmo não foi incrementado anteriormente.

Ciclo de Execução Cada instrução executa passos distintos durante sua execução, caso contrário, elas teriam a mesma funcionalidade. Cabe aos circuitos de controle identificar a instrução armazenada no registrador `IR` e orquestrar a execução dos passos apropriados para a execução da instrução. Os seguintes passos mostram o ciclo de execução de uma instrução:

1. Interpretação dos *bits* do campo operação da instrução, armazenados em `IR`. Esta operação é também chamada de decodificação, pois a operação a ser realizada se encontra codificada, em forma de números, dentro do campo operação.

2. Após a identificação da instrução, a UC verifica se a instrução requer a busca de operandos da memória. Se a busca for necessária, então:

   (a) A UC envia um sinal para a memória realizar uma operação de leitura. Note que o endereço do operando já foi transferido para o registrador `MAR` durante o ciclo de busca.

   (b) A memória lê a palavra de memória e transfere o conteúdo para o registrador `MBR`;

3. Se a instrução envolve a realização de uma operação lógica ou aritmética:

   (a) A UC envia sinais de controle para a unidade lógica aritmética realizar a operação associada com a instrução. Note que neste ponto todos os operandos da operação já se encontram em registradores na unidade lógica e aritmética.

   (b) A ULA realiza a operação lógica ou aritmética de acordo com os sinais enviados pela UC. Estas operações incluem transferência de dados entre registradores da ULA, soma, subtração, multiplicação, divisão e outras.

   (c) A ULA grava o resultado da operação em seus registradores: `AC`, `MQ` ou `MBR`.

4. Se a instrução envolve a gravação do resultado na memória:

   (a) A UC envia um sinal para a memória realizar uma operação de escrita. Note que o endereço do operando já foi transferido para o registrador `MAR` durante o ciclo de busca e o dado já foi transferido de `AC` para `MBR` no passo anterior.

   (b) A memória lê o dado de `MBR` e o grava na palavra de memória associada ao endereço lido de `MAR`.

5. Se a execução da instrução implica no desvio do fluxo de controle, ou seja, se a instrução "salta" para uma outra instrução:

   (a) A UC move o conteúdo do registrador `MAR` para `PC`. Note que o registrador `MAR` contém o valor do campo endereço da instrução sendo executada. No caso de "instruções de salto", este campo contém o endereço da instrução para o qual o fluxo de execução deve ser desviado.

   (b) Caso a execução corresponda a um salto para a instrução à esquerda da palavra de memória selecionada, dá-se início ao ciclo de busca de instrução à esquerda. Caso o salto seja para a instrução à direita, o ciclo de busca de instrução à direita com desvio de fluxo é iniciado.

Como veremos na Seção 5, a instrução "`ADD M(X)`" é uma instrução que soma o conteúdo do registrador `AC` ao conteúdo da palavra de memória armazenada no endereço `X` e grava o resultado no registrador `AC`. A título de exemplo, os seguintes passos mostram o ciclo de execução da instrução:

1. A UC interpreta os *bits* armazenados em `IR` (`0000 0101` no caso da instrução `ADD M(X)`) e identifica a instrução como sendo uma soma.
2. Após a identificação da instrução, o UC sabe que a instrução requer a busca de operandos da memória. Dessa forma:
   (a) A UC envia um sinal para a memória realizar uma operação de leitura. Relembrando que o endereço do operando já foi transferido para o registrador `MAR` durante o ciclo de busca.
   (b) A memória lê a palavra de memória e transfere o conteúdo para o registrador `MBR`;
3. A UC sabe que a instrução `ADD` envolve a realização de uma operação de soma na ULA, então:
   (a) A UC envia sinais para a unidade lógica e aritmética (ULA) solicitando a realização da soma dos valores armazenados em `AC` e `MBR`. Note que neste ponto todos os operandos da operação já se encontram em `AC` e `MBR`.
   (b) A ULA realiza a operação de soma.
   (c) A ULA grava o resultado da soma no registrador `AC`. ULA: `AC`, `MQ` ou `MBR`.

Note que os passos 4 (armazenamento do resultado na memória) e 5 (desvio do fluxo de controle) não são necessários nesta instrução.

5 Instruções e Programação do IAS

O conjunto de instruções do computador IAS possui 20 instruções. As instruções podem ser classificadas em 4 tipos distintos:

- Transferência de dados: instruções para mover dados entre a memória e os registradores;
- Salto: instruções para desviar o fluxo da execução das instruções.
- Aritmética: instruções para realização de operações aritméticas.
- Modificação de endereço: instruções para alterar o campo endereço de outras instruções.

Todas as instruções do IAS possuem 20 *bits*. Além disso, todas as instruções são organizadas em um único formato, com dois campos: código da operação e endereço. O campo código da operação, ou *opcode*, possui 8 *bits*, enquanto que o campo endereço possui 12 *bits*. A Figura 3b ilustra o formato das instruções do computador IAS.

5.1 Instruções de Transferência de Dados

As instruções de transferência de dados são utilizadas para mover dados entre os registradores e a memória. Por exemplo, a instrução "`LOAD M(X)`" é uma instrução de transferência de dados que transfere o dado armazenado na palavra da memória associada ao endereço `X` para o registrador `AC`. A instrução "`LOAD MQ`", no entanto, é uma instrução de transferência de dados que transfere o dado armazenado no registrador `MQ` para o registrador `AC`.

Como veremos mais adiante, a unidade lógica e aritmética realiza operações com valores armazenados na memória e nos registradores `AC` e `MQ`. Para inicializar os valores de `AC` e `MQ` antes de executar uma instrução aritmética ou lógica, podemos executar um instrução de transferência de dados para transferir um dado da

Sintaxe	Operação	Codificação
`LOAD M(X)`	AC := Mem[X]	00000001 | X
Transfere o valor armazenado no endereço X da memória para o registrador AC.		

Figura 7: Instrução `LOAD M(X)`

memória para `AC` ou `MQ`. Para transferir um dado armazenado na memória para o registrador `AC`, podemos utilizar a instrução `LOAD M(X)` descrita na Figura 7.

Podemos utilizar a instrução `LOAD MQ,M(X)`, descrita na Figura 8, para transferir um dado armazenado na memória para o registrador `MQ`.

Sintaxe	Operação	Codificação
`LOAD MQ,M(X)`	MQ := Mem[X]	00001001 | X
Transfere o valor armazenado no endereço X da memória para o registrador MQ.		

Figura 8: Instrução `LOAD MQ,M(X)`

Os resultados de operações lógicas e aritméticas no computador IAS são armazenados nos registradores `AC` e `MQ`. Após a execução de uma operação lógica ou aritmética, pode ser interessante armazenar o resultado da operação na memória. Para transferir um dado armazenado no registrador `AC` para a memória podemos utilizar a instrução apresentada na Figura 9, o `STOR M(X)`.

Sintaxe	Operação	Codificação
`STOR M(X)`	Mem[X] := AC	00100001 | X
Transfere o conteúdo do registrador AC para a palavra da memória no endereço X.		

Figura 9: Instrução `STOR M(X)`

O computador IAS não possui uma instrução capaz de transferir diretamente um dado armazenado no registrador `MQ` para a memória. No entanto, a instrução "`LOAD MQ`", descrita na Figura 10, pode ser utilizada para transferir o dado armazenado em `MQ` para o registrador `AC`. Assim sendo, basta executarmos a instrução `LOAD MQ` e a instrução `STOR M(X)` na sequência para armazenarmos o dado do registrador `MQ` na memória.

Sintaxe	Operação	Codificação
`LOAD MQ`	AC := MQ	00001010 | –
Transfere o conteúdo do registrador MQ para o registrador AC.		

Figura 10: Instrução `LOAD MQ`

O computador IAS possui ainda duas instruções especiais para transferir dados da memória para o registrador `AC`: "`LOAD |M(X)|`" e "`LOAD -|M(X)|`". A instrução `LOAD |M(X)|` (Figura 11) carrega o valor absoluto do número armazenado no endereço `X` da memória, enquanto que a instrução `LOAD -M(X)` (Figura 12) carrega o negativo do número armazenado no endereço `X` da memória.

Sintaxe	Operação	Codificação
`LOAD |M(X)|`	AC := |Mem[X]|	00000011 | X
Transfere o absoluto do valor armazenado no endereço X da memória para o registrador AC.		

Figura 11: Instrução `LOAD |M(X)|`

Sintaxe	Operação	Codificação
`LOAD -M(X)`	AC := -(Mem[X])	00000010 | X
Transfere o negativo do valor armazenado no endereço X da memória para o registrador AC.		

Figura 12: Instrução `LOAD -M(X)`

5.2 Instruções Aritméticas

O computador IAS possui 8 instruções para realização de operações lógicas e aritméticas. Estas instruções são descritas a seguir.

A instrução `ADD M(X)`, descrita na Figura 13, soma dois números e armazena o resultado no registrador `AC`.

Sintaxe	Operação	Codificação
`ADD M(X)`	AC := AC + Mem[X]	00000101 | X
Soma o valor armazenado no endereço X da memória com o valor armazenado no registrador AC e armazena o resultado no registrador AC.		

Figura 13: Instrução `ADD M(X)`

As instruções aritméticas que precisam de 2 operandos, como a soma, buscam um dos operandos da memória e o outro do registrador `AC` ou `MQ`, internos à unidade lógica aritmética. No caso da soma, se quisermos adicionar dois números armazenados na memória, é necessário executar uma instrução de transferência de dados para transferir um dos operandos da memória para o registrador `AC`. O exemplo abaixo ilustra um programa em linguagem de montagem onde os valores armazenados nos endereços `100` e `101` são somados e o resultado é armazenado no endereço `102` da memória.

```
1  LOAD M(100)   # AC := Mem[100]
2  ADD  M(101)   # AC := AC + Mem[101]
3  STOR M(102)   # Mem[102] := AC
```

A instrução "`ADD |M(X)|`", descrita na Figura 14, é uma variação da instrução soma onde o valor do registrador `AC` é somado com o valor "absoluto" do operando carregado da memória.

Sintaxe	Operação	Codificação
`ADD |M(X)|`	AC := AC + |Mem[X]|	00000111 | X
Soma o absoluto do valor armazenado no endereço X da memória com o valor armazenado no registrador AC e armazena o resultado no registrador AC.		

Figura 14: Instrução `ADD |M(X)|`

O computador IAS também possui duas instruções para realizar a subtração de números: "`SUB M(X)`" e "`SUB |M(X)|`", descritas nas Figuras 15 e 16, respectivamente.

Sintaxe	Operação	Codificação
`SUB M(X)`	AC := AC - Mem[X]	00000110 | X
Subtrai o valor armazenado no endereço X da memória do valor armazenado no registrador AC e armazena o resultado no registrador AC.		

Figura 15: Instrução `SUB M(X)`

Sintaxe	Operação	Codificação
`SUB |M(X)|`	AC := AC - |Mem[X]|	00001000 | X
Subtrai o absoluto do valor armazenado no endereço X da memória do valor armazenado no registrador AC e armazena o resultado no registrador AC.		

Figura 16: Instrução `SUB |M(X)|`

A instrução "`MUL M(X)`" (Figura 17) é utilizada para realizar a multiplicação de dois números. Os operandos da multiplicação devem estar armazenados no registrador `MQ` e no endereço `X` da memória. A multiplicação de dois números de 40 *bits* pode resultar em um número de até 80 *bits*. Como o IAS não possui registradores de 80 *bits* para armazenar o resultado, o mesmo é armazenado em dois registradores: AC e MQ. A parte baixa do resultado, ou seja, os *bits* menos significativos, é armazenada no registrador MQ, enquanto que a parte alta é armazenada no registrador AC.

A operação de divisão no IAS é realizada pela instrução "`DIV M(X)`" (Figura 18). O dividendo deve estar armazenado no registrador `AC` e o divisor no endereço `X` da memória principal. Como resultado, a divisão produz dois números, o quociente e o resto. O quociente é armazenado no registrador `MQ` e o resto no registrador `AC`.

Além das operações aritméticas, o IAS possui duas operações lógicas: "`RSH`" e "`LSH`". A instrução `RSH` desloca todos os *bits* do registrador AC para a direita. Ou seja, o *bit* 39 recebe o valor do *bit* 38, o *bit* 38 recebe o valor do *bit* 37 e assim por diante. Por fim, o *bit* 0 recebe o valor 0.

Sintaxe	Operação	Codificação
`MUL M(X)`	AC:MQ := MQ * Mem[X]	00001011 | X
Multiplica o valor no endereço X da memória pelo valor em MQ e armazena o resultado em AC e MQ. O resultado é um número de 80 *bits*. Os 40 *bits* mais significativos são armazenados em AC e os 40 *bits* menos significativos são armazenados em MQ.		

Figura 17: Instrução `MUL M(X)`

Sintaxe	Operação	Codificação
`DIV M(X)`	MQ:=AC/Mem[X] AC:=AC%Mem[X]	00001100 | X
Divide o valor em AC pelo valor armazenado no endereço X da memória. Coloca o quociente em MQ e o resto em AC.		

Figura 18: Instrução `DIV M(X)`

Sintaxe	Operação	Codificação
`RSH`	AC := AC >> 1	00010101 | X
Desloca os *bits* do registrador AC para a direita.		

Figura 19: Instrução `RSH`

Além de ser uma operação lógica (deslocamento de *bits*), a instrução `RSH` pode ser utilizada para realizar a divisão de um número por 2. Note que, quando deslocamos os dígitos binários de um número na representação binária para a direita, eliminando o último dígito, o resultado é a metade do número. Como podemos ver no exemplo abaixo, se deslocarmos[5] os dígitos binários do número 5 ($0000\,0101_2$) para a direita, o resultado é o número 2. Note que o resultado final corresponde ao quociente da divisão. Como podemos ver no mesmo exemplo, o mesmo acontece para o número 8.

$$\begin{aligned} 0000\,0101_2(5_{10}) >> 1 &= 0000\,0010_2(2_{10}) \\ 0000\,1000_2(8_{10}) >> 1 &= 0000\,0100_2(4_{10}) \end{aligned}$$

A instrução "`LSH`" (Figura 20) realiza a operação inversa à instrução `RSH`, ou seja, ela desloca os *bits* do valor no registrador `AC` para a esquerda. Analogamente, o resultado aritmético é a multiplicação do número por 2.

[5]O operador >> é utilizado para representar o deslocamento de *bits* em algumas linguagens de programação de alto nível, como C.

Sintaxe	Operação	Codificação
`LSH`	AC := AC << 1	00010100 | X
Desloca os *bits* do registrador AC para a esquerda.		

Figura 20: Instrução `LSH`

5.3 Instruções de Salto

Uma das principais funcionalidades de um computador é a capacidade de executar trechos de código de um programa repetidas vezes ou sob determinadas condições. Esta funcionalidade permite ao programador expressar comandos condicionais como `if-then-else` ou comandos de repetição como `for` e `while`.

A implementação destes comandos é feita através de saltos no código da aplicação. No caso do IAS, a instrução "`JUMP M(X)`" (Figuras 21 e 22) pode ser utilizada para desviar o fluxo de execução, ou saltar, para a instrução no endereço `X` da memória. O programa em linguagem de montagem a seguir mostra como esta instrução pode ser utilizada para se implementar um laço que adiciona o valor armazenado no endereço `101` da memória ao valor armazenado no endereço `100` da memória.

```
1 Endereço  Instruções
2      000  LOAD M(100);      ADD  M(101)
3      001  STOR M(100);      JUMP M(000,0:19)
```

Neste exemplo, as instruções `LOAD`, `ADD` e `STOR` são executadas na sequência para somar os valores armazenados nos endereços `100` e `101` e armazenar o resultado no endereço `100`. Em seguida, a instrução `JUMP` desvia o fluxo de execução, informando ao processador que a próxima instrução a ser executada deve ser buscada do endereço `000` da memória. Dessa forma, as instruções `LOAD`, `ADD`, `STOR` e `JUMP` serão executadas repetidamente.

Note que o IAS armazena duas instruções em uma mesma palavra da memória: uma à esquerda e outra à direita da palavra. Assim sendo, o IAS possui duas instruções `JUMP`, uma que salta para a instrução à esquerda da palavra no endereço $M(X)$ (Figura 21), e outra que salta para a instrução à direita da palavra no endereço $M(X)$ (Figura 22).

Sintaxe	Operação	Codificação
`JUMP M(X,0:19)`	PC := M(X) e executa instrução à esquerda.	00001101 | X
Salta para a instrução à esquerda (*bits* 0 a 19) da palavra de memória armazenada no endereço M(X).		

Figura 21: Instrução `JUMP M(X,0:19)`

No exemplo anterior, as instruções `LOAD`, `ADD`, `STOR` e `JUMP` são executadas repetidamente em um ciclo que não tem fim, a não ser que o computador seja desligado. Este ciclo sem fim, ou "laço infinito", ocorre porque não há uma condição de parada para o laço. Para criarmos uma condição de parada no laço devemos utilizar uma instrução que desvia o fluxo de execução "condicionalmente", ou seja, uma instrução de salto condicional. No caso do IAS, esta instrução é a `JUMP+ M(X)` (Figuras 23 e 24).

A instrução `JUMP+ M(X)` salta para a instrução no endereço `X` da memória somente se o valor armazenado no registrador `AC` for maior ou igual a 0, ou seja, se `AC` for não negativo. Caso contrário (o valor em `AC` for

Sintaxe	Operação	Codificação
`JUMP M(X,20:39)`	PC := M(X) e executa instrução à direita.	00001110 | X
Salta para a instrução à direita (*bits* 20 a 39) da palavra de memória armazenada no endereço M(X).		

Figura 22: Instrução `JUMP M(X,20:39)`

negativo), o fluxo de execução segue normalmente, executando-se a instrução subsequente à instrução `JUMP+`. O exemplo a seguir mostra um programa que executa o mesmo trecho de código do exemplo anterior por 10 vezes, ou seja, um laço com 10 iterações.

```
1  Endereço  Instruções / Dados
2
3       000  LOAD  M(100);      ADD  M(0101)
4       001  STOR  M(100);      LOAD M(0102)
5       002  SUB   M(103);      STOR M(0102)
6       003  JUMP+ M(000,0:19); ...
7
8       102  00 00 00 00 09 # Contador
9       103  00 00 00 00 01 # Constante 1
```

Neste exemplo, o contador armazenado no endereço `102` (inicializado com o valor 9) é decrementado repetidamente até que o seu valor seja menor do que 0, ou seja, até que o valor seja negativo. Neste caso, a instrução JUMP+ não desvia o fluxo de execução para a instrução no endereço `000` da memória, fazendo com que a execução saia do laço.

Sintaxe	Operação	Codificação
`JUMP+ M(X,0:19)`	Se AC≥0 então PC:=M(X) e executa instrução à esquerda, senão, executa a próxima instrução.	00001111 | X
Salta para a instrução à esquerda (*bits* 0 a 19) da palavra de memória se o valor armazenado em AC for maior ou igual à zero.		

Figura 23: Instrução `JUMP+ M(X,0:19)`

Instruções de salto condicional também podem ser utilizadas para implementar construções `if-then-else`. O exemplo a seguir ilustra como estas instruções podem ser utilizadas em conjunto com instruções de salto incondicional (`JUMP`) para implementar construções `if-then-else`:

```
1  Endereço  Instruções / Dados
2
3       000  LOAD  M(100);      SUB  M(101)      # Se Y > X
4       001  JUMP+ M(003,0:19); LOAD M(101)      #  Z = Y
5       002  STOR  M(102);      JUMP M(004,0:19) # senão
```

Sintaxe	Operação	Codificação
`JUMP+ M(X,20:39)`	Se AC≥0 então PC:=M(X) e executa instrução à direita, senão, executa a próxima instrução.	00010000 | X
Salta para a instrução à direita (*bits* 20 a 39) da palavra de memória se o valor armazenado em AC for maior ou igual à zero.		

Figura 24: Instrução `JUMP+ M(X,20:39)`

```
6       003  LOAD  M(100);      STOR M(102)      #  Z = X
7       004  ...
8
9       100  00 00 00 00 01                     # Variável X
10      101  00 00 00 00 02                     # Variável Y
11      102  00 00 00 00 00                     # Variável Z
```

No exemplo anterior, as instruções `LOAD M(0100)`, `SUB M(0101)` e `JUMP+ M(0003,0:19)` são utilizadas para comparar se Y > X. Note que `AC` recebe o resultado da subtração `X - Y`. Se Y for maior que X, então o resultado em `AC` será um número negativo, e o salto condicional (`JUMP+`) não será tomado. Do contrário (X≥Y), o resultado em `AC` será maior ou igual à zero, e o salto condicional será tomado. Caso o salto não seja tomado (Y > X), as instruções `LOAD M(0101)`, `STOR M(0102)` e `JUMP M(0004,0:19)` serão executadas, movendo o conteúdo de Y para Z e desviando o fluxo de execução para a instrução no endereço `004`, após o `if-then-else`. Caso o salto condicional seja tomado (X≥Y), as instruções `LOAD M(0100)` e `STOR M(0102)` serão executadas, movendo o valor de X para Z. Note que, neste caso, não há a necessidade de inserir uma instrução para saltar para o endereço da instrução após o `if-then-else`, pois a instrução subsequente ao `STOR` (no endereço `004`) já é a instrução que deve ser executada.

5.4 Instruções de Modificação de Endereço

As instruções de transferência de dados e aritméticas que acessam dados da memória são anotadas com o endereço do dado na memória no campo "endereço" da instrução. Note que este endereço é fixo e não permite que uma mesma instrução seja usada para acessar dados em endereços distintos da memória. Esta deficiência fica clara quando trabalhamos com vetores. Vetores (*arrays*) são conjuntos ordenados de dados armazenados de forma consecutiva na memória. Tipicamente, programas que processam dados em vetores utilizam laços que iteram múltiplas vezes acessando um elemento distinto do vetor em cada iteração. Por exemplo, código da função `soma_elementos`, a seguir, soma todos os elementos do vetor A.

```
1  int A[1024];
2
3  int soma_elementos()
4  {
5    int i;
6    int soma=0;
7    for (i=0; i<1024; i++)
8      soma = soma + A[i];
9    return soma;
10 }
```

Já sabemos como implementar laços com o conjunto de instruções do IAS, entretanto, precisamos fazer com que uma mesma instrução acesse dados em endereços diferentes da memória à medida em que as iterações do laço são executadas. Para isso, o IAS dispõe de duas instruções que permitem a modificação do campo endereço de outra instrução: "`STOR M(X,8:19)`" (Figura 25) e "`STOR M(X,28:39)`" (Figura 26).

Sintaxe	Operação	Codificação
`STOR M(X,8:19)`	Mem[X](8:19) := AC(28:39)	00010010 | X
Move os 12 *bits* à direita do registrador AC para o campo endereço da instrução à esquerda da palavra de memória no endereço X.		

Figura 25: Instrução `STOR M(X,8:19)`

Sintaxe	Operação	Codificação
`STOR M(X,28:39)`	Mem[X](28:39) := AC(28:39)	00010011 | X
Move os 12 *bits* à direita do registrador AC para o campo endereço da instrução à direita da palavra de memória no endereço X.		

Figura 26: Instrução `STOR M(X,28:39)`

A instrução `STOR M(X,8:19)` modifica o campo endereço da instrução à esquerda da palavra no endereço `X` da memória, enquanto que a instrução `STOR M(X,28:39)` modifica o campo endereço da instrução à direita da palavra no endereço `X` da memória. Ambas transferem os 12 *bits* à direita do registrador `AC` (*bits* 28 a 39) para o campo endereço da instrução alvo. O programa a seguir exemplifica o uso destas instruções. O programa soma os elementos de um vetor de 20 números armazenados a partir do endereço `100`.

```
1  Endereço  Instruções / Dados
2
3       000  LOAD  M(0F2); STOR M(002,28:39) # Modifica o endereço da instrução ADD
4       001  ADD   M(0F1); STOR M(0F2)       # e atualiza o apontador.
5       002  LOAD  M(0F3); ADD  M(000)       # Carrega a variável e soma com o conteúdo
6                                            # do vetor apontado pelo apontador.
7       003  STOR  M(0F3); LOAD M(0F0)       # Salva a soma e carrega o contador de it.
8       004  SUB   M(0F1); STOR M(0F0)       # Atualiza o contador de iterações.
9       005  JUMP+ M(000,0:19); ...
10
11      0F0  00 00 00 00 19                  # Contador de iterações
12      0F1  00 00 00 00 01                  # Constante 1
13      0F2  00 00 00 01 00                  # Apontador
14      0F3  00 00 00 00 00                  # variável soma
15
16      100  00 00 00 00 00                  # Primeiro elemento do vetor
17      ...   ...
```

O programa armazena um apontador para o início do vetor no endereço `0F2` e a variável soma no endereço `0F3`. O primeiro passo é carregar o apontador no registrador `AC` e gravar o endereço no campo endereço da

instrução que realiza a soma, ou seja, a instrução `ADD` do endereço `002`. Note que, inicialmente, o campo endereço desta instrução possui o endereço `000`. No entanto, antes mesmo desta instrução ser executada, o seu campo endereço será modificado pela instrução `STOR M(0002,28:39)`. Após modificar a instrução, o apontador é incrementado e o novo valor (o endereço do próximo elemento no vetor) é armazenado na memória. Após este passo, o programa carrega o conteúdo da variável soma no registrador `AC` e utiliza a instrução `ADD` (com o endereço previamente modificado) para somar um elemento do arranjo. O valor resultante da soma é armazenado na memória (no endereço da variável soma) e o programa segue com o decremento e teste do contador de iterações, iterando o laço.

6 Linguagem de Montagem do IAS

Um montador é uma ferramenta que converte código em linguagem de montagem para código em linguagem de máquina. A Figura 27 mostra um trecho de programa representado na linguagem de montagem (a) e na linguagem de máquina (b) do computador IAS.

```
1           LOAD M(0x102)             01 10 20 B1 03
2           MUL  M(0x103)             0A 00 02 11 02
3           LOAD MQ
4           STOR M(0x102)
5                (a)                        (b)
```

Figura 27: Trecho de programa em linguagem de montagem (a) e em linguagem de máquina (b) do IAS.

O trecho de código da Figura 27 carrega o valor armazenado no endereço `0x102` da memória no registrador `AC`, multiplica pelo valor armazenado no endereço `0x103` da memória e armazena o resultado no endereço `0x102` da memória. Para converter o código em linguagem de montagem acima para o código em linguagem de máquina, o montador pode ler as linhas do programa em linguagem de montagem uma a uma e para cada linha:

- mapear o mnemônico da instrução (ex: `LOAD M(0x102)`) para o código da operação correspondente (ex: `01`);
- ler o parâmetro da instrução (ex: `0x102`).
- compor a instrução de 20 *bits* (ex: `0A 10 2`), sendo os 8 primeiros *bits* da instrução o código da operação e os 12 últimos *bits* da instrução o parâmetro lido.
- adicionar a instrução ao final do arquivo de saída.

6.1 A diretiva `.org`

No exemplo da Figura 27 não fica claro onde cada instrução montada será armazenada na memória. Por exemplo, onde deve ser colocada a instrução `LOAD M(0x102)`? Essa informação pode ser fornecida pelo programador (ou compilador) no programa em linguagem de montagem através da diretiva `.org`. Esta diretiva informa ao montador o endereço de memória onde o montador deve iniciar (ou continuar) a geração do código. O trecho de código a seguir mostra um exemplo de uso da diretiva `.org`. Neste trecho, o código após a diretiva `.org 0x000` será despejado na memória a partir do endereço `0x000`, enquanto que o código listado após a diretiva `.org 0x020` será despejado na memória a partir do endereço `0x020`. Observe que o mapa de memória[6] possui palavras nos endereços `0x000` e `0x001` e uma palavra no endereço `0x020`.

[6]Mapa de memória é uma lista de pares <endereço,valor> que especifica o valor a ser atribuido a cada palavra da memória na inicialização do simulador. Note que os valores podem representar instruções ou dados.

```
1           .org 0x000              000   01 10 20 B1 03
2             LOAD M(0x102)         001   0A 00 00 D0 20
3             MUL  M(0x103)
4             LOAD MQ               020   21 10 20 00 00
5             JUMP M(0x020,0:19)
6
7           .org 0x020
8             STOR M(0x102)
9
10        Ling. de Montagem         Mapa de memória
```

6.2 A diretiva `.word`

O código da Figura 27 carrega e multiplica os valores armazenados nas palavras de memória associadas aos endereços `0x102` e `0x103`. O resultado da multiplicação é armazenado no endereço `0x102` da memória. Note que o programa da Figura 27 não especifica o valor inicial dessas posições de memória. A diretiva `.word` é uma diretiva que auxilia o programador a adicionar dados à memória. Para adicionar um dado, basta inserir a diretiva `.word` e um valor de 40 *bits* no programa. O trecho de código a seguir mostra como esta diretiva pode ser utilizada para adicionar dados à memória.

```
1           .org 0x102              000   01 10 20 B1 03
2             .word 0x1             001   0A 00 00 D0 00
3             .word 10
4                                   102   00 00 00 00 01
5           .org 0x000              103   00 00 00 00 0A
6             LOAD M(0x102)
7             MUL  M(0x103)
8             LOAD MQ
9             JUMP M(0x000,0:19)
10
11          Ling. de Montagem         Mapa de memória
```

Como vimos antes, a diretiva `.org 0x102` é utilizada para informar ao montador que as instruções e dados provenientes de comandos subsequentes a esta diretiva devem ser despejados na memória a partir do endereço `0x102`. No caso do exemplo anterior, a diretiva `.word 0x1` é o próximo comando, portanto, o montador adicionará o dado `00 00 00 00 01` no endereço `0x102` da memória. A diretiva `.word` subsequente toma como argumento um número na representação decimal (note a ausência do "`0x`"). Nesse caso, o montador converte este valor para representação hexadecimal (`00 00 00 00 0A`) e adiciona o dado ao mapa de memória. Note que esta palavra é adicionada na próxima palavra da memória, ou seja, no endereço `0x103`.

6.3 Rótulos

Rótulos são anotações no código que serão convertidas em endereços pelo montador. A sintaxe de um rótulo é uma palavra terminada com o caractere ":" (dois pontos). As seguintes linhas de código mostram exemplos de rótulos:

```
1 laco:
2 var_x:
```

Rótulos podem ser utilizados para especificar um local no código para onde uma instrução de desvio deve saltar. O código abaixo utiliza um rótulo (`laco:`) para representar o alvo de uma instrução de salto.

```
laco:
  LOAD M(0x100)
  SUB  M(0x200)
  JUMP M(laco)
```

Durante a montagem do programa, o montador associará o rótulo `laco:` a um endereço de memória e o campo endereço da instrução `JUMP` será preenchido com este endereço.

Rótulos também podem ser utilizados para especificar um local na memória que contenha um dado. Por exemplo, podemos associar os endereços `0x100` e `0x200` aos rótulos `var_x` e `var_y` e referenciar os rótulos em vez dos endereços nas instruções do programa. O exemplo a seguir mostra um trecho de código onde o rótulo `var_x` é associado ao endereço `0x100` e o rótulo `var_y` é associado ao endereço `0x200`.

```
.org 0x000
laco:
  LOAD M(var_x)
  SUB  M(var_y)
  JUMP M(laco)
.org 0x100
var_x:
.org 0x200
var_y:
```

Observe que os rótulos podem ser utilizados em conjunto com a diretiva `.word` para declarar variáveis ou constantes. Em vez de associar um rótulo a um endereço fixo de memória, podemos declarar um rótulo e logo em seguida adicionar um dado neste endereço de memória com a diretiva `.word`, e o próximo rótulo, se usado, conterá o endereço da próxima palavra da memória. O trecho de código a seguir mostra exemplos de declaração de variáveis e constantes. Neste caso, o rótulo `var_x` será associado ao endereço de memória `0x100` e o rótulo `const1` será associado ao endereço de memória `0x101`.

```
    .org 0x000                      000   01 10 00 61 01
    laco:                           001   0D 00 00 00 00
      LOAD M(var_x)
      SUB  M(const1)                100   00 00 00 00 01
      JUMP M(laco)                  101   00 00 00 00 09

    .org 0x100
    var_x:
      .word 00 00 00 00 09
    const1:
      .word 00 00 00 00 01

    Linguagem de Montagem             Mapa de memória
```

Como mencionamos antes, rótulos são convertidos para endereços. Como endereços são números naturais, eles também podem ser utilizados em conjunto com a diretiva `.word` para inicializar posições de memória. O trecho de código a seguir mostra um exemplo onde o rótulo `vetor` é utilizado em conjunto com a diretiva

`.word` para adicionar o endereço base do vetor à palavra da memória associada com o endereço do rótulo `base`. Em outras palavras, declaramos a variável base e a inicializamos com o valor `0x100`, ou seja, o endereço inicial do vetor.

```
1     .org 0x100                        100   00 00 00 01 01
2     base:                             101   00 00 00 00 00
3       .word vetor                     102   00 00 00 00 01
4     vetor:                            103   00 00 00 00 02
5       .word 00 00 00 00 00
6       .word 00 00 00 00 01
7       .word 00 00 00 00 02
8     fim_vetor:
9
10    Linguagem de Montagem               Mapa de memória
```

Rótulos e o processamento da entrada em linguagem de montagem em dois passos

No exemplo anterior, nós utilizamos o rótulo `vetor` na linha 3 no programa antes mesmo de ele ser declarado (na linha 4). Durante a montagem, os comandos de montagem são lidos um a um. Ao processar a diretiva `.word vetor` o montador teria que adicionar uma entrada ao mapa de memória com o valor do endereço associado ao rótulo `vetor`. Entretanto, como o montador pode inferir o valor do rótulo `vetor` se ele ainda não foi declarado no programa? A resposta é: ele não pode! Para resolver este problema, os montadores realizam o processo de montagem em dois passos:

1. No primeiro passo, o montador realiza uma montagem parcial, onde os campos endereços e os dados das diretivas `.word` são ignorados, iniciando o mapa de memória com zero. Note que, durante a montagem, o montador não precisa colocar os valores corretos no mapa de memória. Basta saber o tamanho do dado ou da instrução na memória de forma que os rótulos sejam associados aos endereços corretos.

2. No segundo passo, o montador gera um novo mapa de memória, realizando a montagem completa, com os dados corretos. Neste caso, o montador já possui uma tabela informando para cada rótulo qual o endereço da memória.

6.4 A diretiva `.align`

As instruções do IAS possuem 20 *bits*, entretanto, as palavras de memória principal possuem 40 *bits*. Durante a montagem, o montador adiciona instruções uma a uma preenchendo as palavras com duas instruções cada. Todavia, a diretiva `.word` solicita ao montador para adicionar um dado de 40 *bits* a uma palavra da memória. O que o montador faria ao tentar montar o seguinte programa?

```
1 .org 0x000
2 laco:
3   LOAD M(var_x)
4   SUB  M(var_y)
5   JUMP M(laco)
6 var_x: .word 0x1
7 var_y: .word 0x2
```

Como vimos antes, o montador colocaria as instruções `LOAD` e `SUB` na primeira palavra da memória, no endereço `0x000` e colocaria a instrução `JUMP` (de 20 *bits*) à esquerda da segunda palavra de memória. Logo

em seguida, o montador tentaria adicionar o dado `00 00 00 00 01` à memória. Mas onde este dado seria colocado? Uma opção é adicionar a primeira metade do dado (20 *bits*) à direita da instrução `JUMP` e o restante na parte esquerda da próxima palavra. Além de ser confusa, esta abordagem divide o dado em duas palavras de memória, tornando impossível a leitura do dado pelas instruções do programa.

A outra opção seria escrever zero na parte direita da palavra que já possui a instrução `JUMP` e adicionar o dado de 40 *bits* na próxima palavra da memória. Neste caso, o mapa de memória ficaria como a seguir:

```
000 01 00 20 60 03
001 0D 00 00 00 00
002 00 00 00 00 01
003 00 00 00 00 02
```

Note que a segunda palavra da memória (endereço `0x001`) possui a instrução `JUMP` (operação `0D`) à esquerda e o valor zero à direita. Enquanto que os dados `00 00 00 00 01` e `00 00 00 00 02` foram adicionados nas palavras seguintes da memória. Esta abordagem funcionaria.

Uma terceira abordagem seria não fazer a montagem, ou seja, ao encontrar uma situação dessas, o montador emitiria um mensagem de erro informando que a diretiva `.word` está tentando emitir um dado em uma posição de memória "não alinhada" a uma palavra da memória. Esta é a abordagem que o montador do IAS deve tomar. Mas como fazemos para montar o programa do exemplo anterior? A resposta está na diretiva `.align N`. Esta diretiva informa ao montador para continuar a montagem a partir da próxima palavra com endereço múltiplo de `N`. O trecho de código a seguir mostra como a diretiva `.align` pode ser utilizada no programa anterior.

```
.org 0x000
laco:
  LOAD M(var_x)
  SUB  M(var_y)
  JUMP M(laco)
.align 1
var_x: .word 0x1
var_y: .word 0x2
```

6.5 A diretiva `.wfill`

Um vetor pode ser declarado e inicializado utilizando-se um rótulo e a diretiva `.word` para adicionar diversos valores em posições consecutivas da memória. O exemplo a seguir mostra como um vetor de 3 posições com os valores 4, 0 e 4 pode ser adicionado a um programa em linguagem de montagem:

```
.org 0x000
laco:
  JUMP M(laco)
.align 1
vetor:
  .word 0x4
  .word 0x0
  .word 0x4
```

Note que os valores foram adicionados à memória a partir do endereço `0x001`, pois o endereço `0x000` contém a instrução `JUMP`. Apesar da diretiva `.word` nos permitir inicializar o vetor, esta tarefa se torna

muito tediosa quando o tamanho do vetor é grande (por exemplo, 1000 elementos). Neste caso, para facilitar a adição de dados à memória, podemos utilizar a diretiva `.wfill N,D`. Esta diretiva preenche `N` palavras da memória com o dado `D`. O trecho de código a seguir mostra como podemos declarar um vetor com 1000 palavras inicializadas com o valor `00 00 00 00 05`.

```
1 .org 0x000
2 laco:
3   JUMP M(laco)
4 .align 1
5 vetor:
6   .wfill 1000, 0x5
```

6.6 A diretiva `.set`

Para tornar o código mais claro e mais portável, muitas linguagens de programação dispõem de diretivas que permitem ao programador associar valores a símbolos. Por exemplo, na linguagem C, a diretiva `#define NOME VALOR` pode ser utilizada para associar um valor a um determinado nome. O código a seguir mostra um exemplo deste uso:

```
1   #define TEMP_MAX 60
2   #define TEMP_MIN  0
3   ...
4   if (temperatura < TEMP_MIN)
5     error(‘‘Temperatura muito baixa\n’’);
6   else if (temperatura > TEMP_MAX)
7     error(‘‘Temperatura muito alta\n’’);
8   ...
```

Neste exemplo, o valor 60 foi associado ao símbolo `TEMP_MAX` e o valor 0 foi associado ao símbolo `TEMP_MIN`. Caso o valor de `TEMP_MAX` ou `TEMP_MIN` seja diferente em um sistema mais novo, o programador não precisa inspecionar todo o código modificando os valores, pois basta modificar o valor na diretiva que associa o valor ao símbolo.

A diretiva `.set NOME VALOR` é a diretiva utilizada na linguagem de montagem do IAS para associar valores a nomes. O trecho de código a seguir mostra um exemplo de uso desta diretiva.

```
1      .set CODIGO  0x000              000 0D 00 00 00 00
2      .set DADOS   0x100              100 00 00 00 00 05
3      .set TAMANHO 200                101 00 00 00 00 05
4                                      ...
5      .org CODIGO                     1C6 00 00 00 00 05
6      laco:                           1C7 00 00 00 00 05
7        JUMP M(laco)
8
9      .org DADOS
10     vetor:
11       .wfill TAMANHO, 0x5
12
13      Ling. de Montagem               Mapa de memória
```

Note que esta diretiva não emite dados ou instruções ao programa gerado, ela apenas associa valores a símbolos. De fato, este programa é equivalante ao programa:

```
1      .org 0x000                    000 0D 00 00 00 00
2      laco:                         100 00 00 00 00 05
3        JUMP M(laco)                101 00 00 00 00 05
4                                    ...
5      .org 0x100                    1C6 00 00 00 00 05
6      vetor:                        1C7 00 00 00 00 05
7        .wfill 200, 0x5
8
9       Ling. de Montagem             Mapa de memória
```

Apêndice A: Lista de instruções do Computador IAS

Tipo da Instrução	Código da operação	Representação Simbólica	Descrição
Transferência de Dados	00001010	LOAD MQ	Transfere o conteúdo do registrador MQ para o registrador AC
	00001001	LOAD MQ,M(X)	Transfere o conteúdo da memória no endereço X para o registrador MQ
	00100001	STOR M(X)	Transfere o conteúdo do registrador AC para a memória no endereço X
	00000001	LOAD M(X)	Transfere o conteúdo da memória no endereço X para o registrador AC
	00000010	LOAD -M(X)	Transfere o negativo do valor armazenado no endereço X da memória para o registrador AC
	00000011	LOAD |M(X)|	Transfere o absoluto do valor armazenado no endereço X da memória para o registrador AC
Salto incondicional	00001101	JUMP M(X,0:19)	Salta para a instrução da esquerda na palavra contida no endereço X da memória
	00001110	JUMP M(X,20:39)	Salta para a instrução da direita na palavra contida no endereço X da memória
Salto condicional	00001111	JUMP+M(X,0:19)	Se o número no registrador AC for não negativo então salta para a instrução à esquerda da palavra contida no endereço X da memória
	00010000	JUMP+M(X,20:39)	Se o número no registrador AC for não negativo então salta para a instrução à direita da palavra contida no endereço X da memória
Aritmética	00000101	ADD M(X)	Soma o valor contido no endereço X da memória com o valor em AC e coloca o resultado em AC
	00000111	ADD |M(X)|	Soma o absoluto do valor contido no endereço X da memória com o valor em AC e armazena o resultado em AC
	00000110	SUB M(X)	Subtrai o valor contido no endereço X da memória do valor em AC e coloca o resultado em AC
	00001000	SUB |M(X)|	Subtrai o absoluto do valor contido no endereço X da memória do valor em AC e armazena o resultado em AC
	00001011	MUL M(X)	Multiplica o valor no endereço X da memória pelo valor em MQ e armazena o resultado em AC e MQ. AC contém os *bits* mais significativos do resultado
	00001100	DIV M(X)	Divide o valor em AC pelo valor no endereço X da memória. Coloca o quociente em MQ e o resto em AC
	00010100	LSH	Desloca os *bits* do registrador AC para a esquerda. Equivale à multiplicar o valor em AC por 2
	00010101	RSH	Desloca os *bits* do registrador AC para a direita. Equivale à dividir o valor em AC por 2
Modificação de endereço	00010010	STOR M(X,8:19)	Move os 12 *bits* à direita de AC para o campo endereço da instrução à esquerda da palavra X na memória
	00010011	STOR M(X,28:39)	Move os 12 *bits* à direita de AC para o campo endereço da instrução à direita da palavra X na memória