SimuS

Um Simulador Didático para Arquitetura de Computadores

GABRIEL P. SILVA

JOSÉ ANTONIO BORGES

Gabriel P. Silva

José Antonio Borges

SimuS

Um Simulador Didático para Arquitetura de Computadores

Primeira Edição

Rio de Janeiro

Edição do Autor

2017

S586s Silva, Gabriel Pereira da

SimuS um simulador didático para arquitetura de computadores. / Gabriel Pereira da Silva, José Antonio dos Santos Borges. – Rio de Janeiro : Ed. do Autor, 2017.

xii, 130p. :il.

1. Arquitetura de Computadores - Estudo e Ensino. 2. Simulador SimuS.

I. BORGES, José Antonio dos Santos. II. Título.

CDD 004.22

Sumário

Lista de Figuras

Prefácio

”Conto os versos de um poema, calculo a altura de uma estrela, avalio o número de franjas, meço a área de um país, ou a força de uma torrente – aplico, enfim, fórmulas algébricas e princípios geométricos – sem me preocupar com os louros que possa tirar de meus cálculos e estudos!”

– Malba Tahan, *O Homem que Calculava*

Um dos problemas encontrados no ensino de arquitetura de computadores é fazer com que os alunos compreendam corretamente o funcionamento de um processador, proporcionando também uma visão comparativa sobre as diversas alternativas arquiteturais possíveis. O uso de simuladores de processadores, sejam eles comerciais ou puramente didáticos, é uma estratégia frequentemente utilizada para alcançar estes objetivos. A visão dos componentes, do funcionamento da arquitetura do processador, do formato das instruções e dos passos necessários para a execução de um programa, são elementos importantes na formação de um conhecimento sólido do funcionamento dos processadores por parte do estudante.

Com esse requisitos em mente lançamos, há mais de 10 anos, o simulador Neanderwin para o processador didático Neander-X (G. P. BORGES J. A. S. ; S., 2006), que têm sido utilizados, pelos autores e por diversos professores em universidades do país, nos cursos iniciais da disciplina de Arquitetura e Organização de Computadores. A partir da experiência acumulada ao longo desta década, iniciamos o desenvolvimento do simulador SimuS (SILVA, 2016), para o processador hipotético Sapiens, ambos sendo apresentados neste livro.

A filosofia principal do simulador SimuS é apresentar uma arquitetura que pode ser explorada em diversos níveis de complexidade, com o emprego de exemplos mais simples, para os cursos iniciais, até exemplos mais complexos, para os cursos mais avançados, sempre com o uso da mesma ferramenta. Pretendemos ainda que essa ferramenta tenha um uso ampliado,

tanto para os cursos de Ciência da Computação, como Engenharia de Computação, Engenharia de Software, Sistemas de Informação ou Licenciatura em Computação.

A arquitetura e o conjunto de instruções inicialmente propostos para o processador Neander-X foram estendidos no processador Sapiens, de modo a obtermos uma arquitetura de processador com maiores possibilidades. As mudanças introduzidas incluem um maior espaço de endereçamento de memória, agora com 64 Kbytes; instruções para a chamada e retorno de procedimentos; um apontador de pilha; um conjunto de instruções mais robusto; além da emulação de novos dispositivos de E/S e atualização da interface de usuário, para representar adequadamente essas novas modificações. Apesar dessas mudanças, não houve perda de compatibilidade, em nível de linguagem de montagem, com códigos produzidos para as arquiteturas legadas do Neander e Neander-X, ainda podem ser compilados e executados no simulador SimuS.

O simulador SimuS apresenta um ambiente integrado de desenvolvimento, onde o aluno pode realizar todo o ciclo de depuração de qualquer programa (em particular programas em linguagem de montagem ou linguagem de máquina), que exige diversas modificações no código, com idas e vindas entre as etapas de codificação, compilação e execução.

Na elaboração deste simulador nos mantivemos fiéis aos conceitos de um ambiente integrado de desenvolvimento, que inclui um editor do código com uma ferramenta de apoio ao desenvolvimento inicial do código, passando por um montador com informações sobre os erros de sintaxe e finalmente um simulador, onde o funcionamento final do programa pode ser verificado, com acesso a todos os elementos que compõem o estado do processador como acumulador, apontador de instruções, códigos de condição, memória, entre outros.

O simulador SimuS, por ser distribuído em código aberto (disponível em https://sourceforge.net/projects/SimuS/), viabiliza a sua expansão (por outros professores ou por alunos em projeto), possibilitando a exploração de variantes da arquitetura ou adição de novas ferramentas de ensino ou projeto. O código fonte deste simulador pode ser compilado para obtermos versões binárias para execução tanto para o sistema operacional Windows, como para

o sistema operacional Linux e suas variantes.

Gostaríamos de agradecer as valiosas contribuições, os comentários construtivos e toda a paciência de nossos alunos durante a fase de desenvolvimento desta ferramenta. Sem isso, o SimuS não poderia ter a qualidade e versatilidade que possui hoje em dia. Esperamos que este pequeno livro possa ser uma contribuição para a melhoria do ensino da disciplina de Arquitetura e Organização de Computadores e todas as suas variantes.

Capítulo 1

Breve Histórico

Em nossa pesquisa encontramos diversas ferramentas de simulação para processadores e microcontroladores comerciais. Os simuladores comerciais mais usados em projetos de equipamentos na atualidade (e que em princípio poderiam ser usados nos cursos básicos de arquitetura de computadores) são os de 8051, ATmega, PIC e variantes de processadores do tipo RISC, como o ARM. São quase sempre sistemas completos, contendo muitas ferramentas integradas, mas destinados ao uso profissional, tendo alguns deles licenças de uso pagas e de uso relativamente complexo, inviabilizando o uso amplo no ensino de graduação em muitos cursos no Brasil.

Na área dos simuladores didáticos existem vários sistemas desenvolvidos no exterior, muitos deles descritos em YURCIK; GEHRINGER, 2002. Como exemplos de simuladores didáticos de processadores comerciais podemos citar o Abacus (ZILLER, 2000), o GNUSim8085 (SRIDHAR, 2016), MARS (SANDERSON, 2016) e o WinMIPS64 (SCOTT, 2006). O Abacus (desenvolvido para Windows) e o GNUSim8085 (desenvolvido para plataformas Unix) simulam o microprocessador 8085 da Intel, que tem sido utilizado amplamente no ensino de arquitetura de computadores, devido à sua simplicidade e por não terem pipeline. Ambos apresentam os valores dos registradores no decorrer da execução das instruções, além de exibir o conteúdo presente na memória do processador. A arquitetura do processador MIPS, que utiliza o pipeline, tem disponíveis os simuladores MARS (32 bits) e WinMIPS64 (64 bits). Essas ferramentas possibilitam a visualização das instruções passando pelos diversos estágios do pipeline, além de examinar o conteúdo de todos os registradores, sejam inteiros ou de ponto flutuante, e também da memória, convenientemente dividida entre memória de dados e de instruções.

Há alguns exemplos de ferramentas didáticas para processadores hipotéticos desenvolvidas no país, entre as quais podemos citar o simulador R2DSim (MOREIRA, 2009), uma ferramenta didática para simulação de uma arquitetura RISC de 16 bits com pipeline de quatro estágios, cujo banco de

registradores possui tamanho e largura configurável. No conjunto de ferramentas apresentadas estão um montador e um simulador integrados. O SIMAEAC (VERONA, 2009) - Simulador Acadêmico para Ensino de Arquitetura de Computadores - implementa um subconjunto da arquitetura do processador 8085, mas com uma memória de apenas 256 *bytes*, com um montador e simulador integrados. Esses dois primeiros exemplos não possuem editor integrado e não é claro como eventuais mensagens de erro são apresentadas. Temos ainda o SEAC (E. V. C. L. ; e. a. BORGES, 2012), um simulador online para uma arquitetura hipotética com pipeline de quatro estágios, com foco nos passos de microprogramação necessários para executar cada instrução.

Outro simulador interessante é o W-Neander, produzido como *software* companheiro do livro Fundamentos de Arquitetura de Computadores (WEBER, 2012), mas que carece de um montador para linguagem de montagem e de um editor integrados.

Em nossas atividades de ensino básico de Arquitetura de Computadores no nível de graduação, a escolha de uma arquitetura e um simulador que pudesse ser de fácil aprendizado, utilizando um ambiente integrado de desenvolvimento, e com praticidade para uso em sala de aula, sempre foi um desafio. A arquitetura Neander foi uma das que apresentaram maior adequação, em particular nos cursos em turmas com grande desnível de formação e sem conhecimento prévio de conceitos de eletrônica.

A arquitetura Neander é muito simples e pode ser explicada em uma ou duas aulas. Avaliamos que o conjunto original de instruções do Neander apresenta algumas limitações que dificultam a criação de programas pouco mais do que triviais. Embora na obra desses autores sejam apresentadas outras arquiteturas mais sofisticadas, porem há necessidade de migração de uma ferramenta para outra, na medida em que a complexidade dos processadores aumenta, consumindo horas de aula com pouco ganho nos cursos.

Neste sentido, apresentamos em 2006 o simulador Neanderwin para uma versão estendida dessa arquitetura, que chamamos de Neander-X, ambos descritos em detalhes em (G. P. BORGES J. A. S. ; S., 2006) e (BORGES, 2016). Nossa ideia foi unir as estratégias de integração de ferramentas,

encontradas nas ferramentas profissionais com uma arquitetura simples como a do Neander, que seria expandida para diminuir algumas limitações e ampliar o seu potencial didático.

Consolidamos e avançamos nesses conceitos, apresentando aqui tanto o simulador SimuS quanto o processador Sapiens, que são uma evolução natural, mas sempre guardando compatibilidade em nível de linguagem de montagem, com os programas já desenvolvidos para o processador e simulador originais.

Acrescentamos diversas funcionalidades ao simulador Neanderwin, entre elas destacamos novos dispositivos virtuais de entrada e saída, como um visor (*banner*) de 16 caracteres e um teclado numérico de 12 teclas. Além disso, funcionalidades como rotinas de TRAP para emular operações de entrada e saída mais complexas como escrita e leitura em uma console, temporização e geração de números aleatórios.

Entendemos que hoje em dia ninguém programa em linguagem de montagem no seu dia-a-dia. Mas é muito importante que os alunos tenham esse tipo de atividade, para que possam aproveitar com maior segurança o que os computadores podem oferecer, usando de forma mais efetiva as linguagens de programação convencionais, e conseguindo entender melhor o processo de tradução dos programas em linguagem de alto nível para linguagem de máquina, de modo que esses programas possam ser executados com maior desempenho e eficiência.

Capítulo 2

Conceitos Básicos

2.1 Funcionamento de um Processador

De uma forma simplificada, o computador é um dispositivo eletrônico cuja principal finalidade é o processamento da informação (na forma de textos, números, imagens e sons) conforme especificado pelo usuário com a utilização de um programa.

Um programa é um conjunto de instruções e dados em formato binário que são carregados na memória do computador, que definem uma forma de interação com o usuário e especificam as tarefas que serão realizadas pelo computador.

Os programas são inicialmente definidos pelos programadores com o uso de uma linguagem de alto nível (como C, Python, Fortran e outras) e que são posteriormente traduzidos pelo compilador para a linguagem de máquina, que é a linguagem que o processador entende.

O processador é o componente eletrônico no computador responsável pela interpretação das instruções em linguagem de máquina e controle de todos os demais dispositivos do computador, com o objetivo de realizar as tarefas determinadas pelo usuário.

Embora existam diversos tipos de processadores, com diferentes conjuntos de instruções em linguagem de máquina e com variados níveis de sofisticação, podemos afirmar com segurança que o funcionamento básico de todos eles é o mesmo como definido por Von Neumann (Fig. 2.1) há mais de cinquenta anos.

Para executar um programa que está na memória, o processador realiza permanentemente a busca de instruções, no endereço de memória indicado pelo apontador de instruções (PC). Depois de lida da memória, a instrução é transferida para o registrador de instrução (RI) e o valor do apontador de

instruções é automaticamente incrementado para a posição de memória seguinte. Caso a instrução executada seja uma instrução de desvio, o valor do PC é alterado para o endereço especificado por essa instrução de desvio.

As instruções podem ler ou escrever dados (chamados de operandos), que podem ser acessados de diversas maneiras (chamados modos de endereçamento) em posições distintas na memória computador.

Figura 2.1: Arquitetura Convencional

Repare que fica tudo armazenado na memória: em princípio dados e instruções podem ficar na mesma memória, sem conflito, desde que estejam em endereços diferentes. Há algumas situações em que é conveniente colocar dados e instruções em memórias diferentes, esse tipo especial de organização recebe o nome especial de arquitetura *Harvard*(Fig. 2.2).

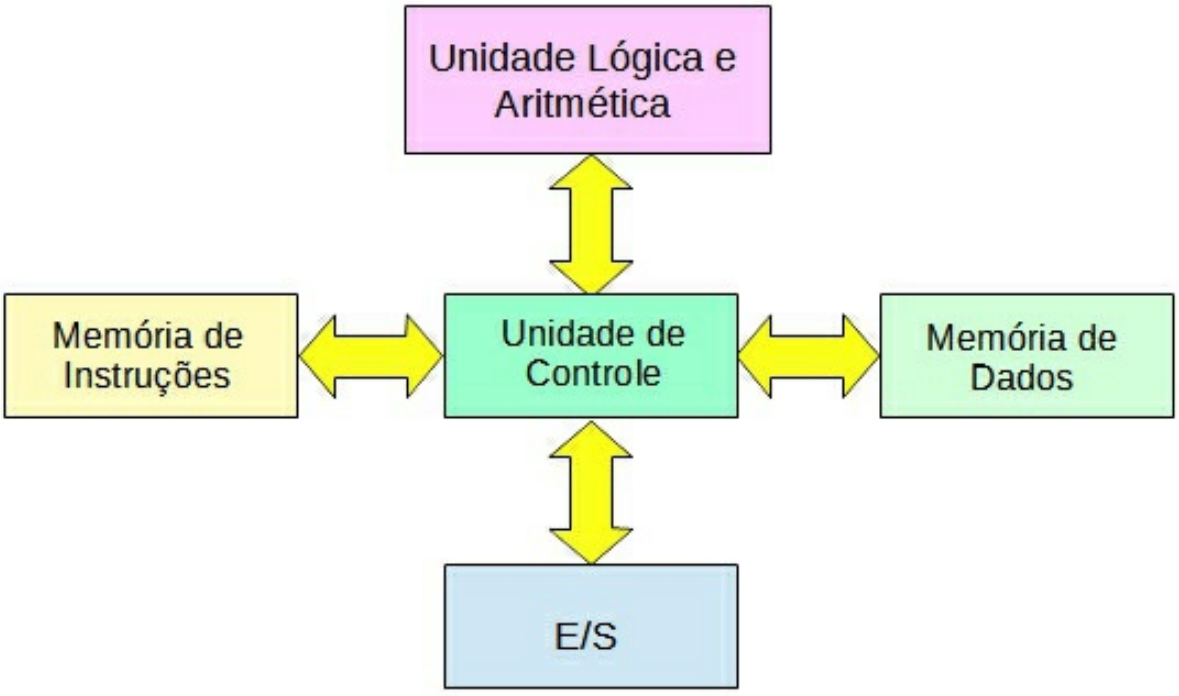

Figura 2.2: Arquitetura Harvard

Para executar as instruções o processador conta com um circuito especial, chamado de unidade aritmética e lógica (UAL), capaz de realizar operações aritméticas como soma e subtração com os operandos das instruções, sempre no formato binário. A fim de agilizar essas operações o processador possui internamente dispositivos especiais de armazenamento, chamados de registradores ou acumuladores, para guardar os resultados temporários dessas operações.

Os processadores mais simples possuem pelo menos um acumulador (AC), enquanto que os mais sofisticados possuem dezenas ou até mesmo centenas de registradores para armazenar esses resultados temporários das operações. Em qualquer caso, em algum momento, esses valores armazenados nos registradores serão transferidos para a memória do computador, nas posições correspondentes às variáveis do programa. Um modelo bem simples de processador pode ser visto na Figura 2.3.

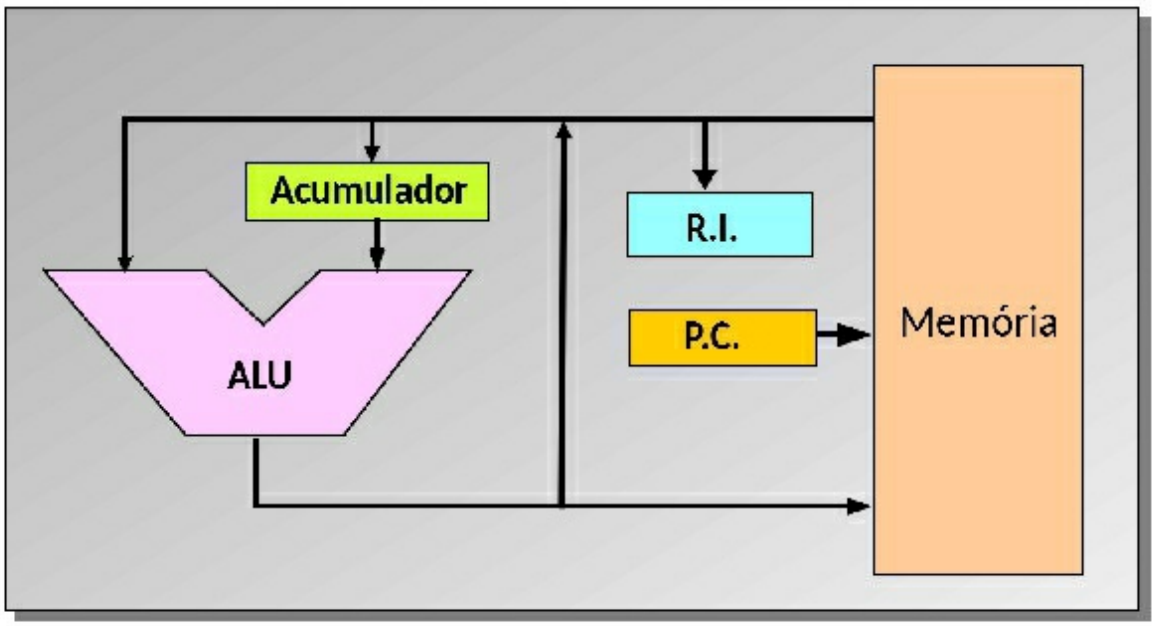

Figura 2.3: Processador com Acumulador

Uma facilidade que os processadores modernos oferecem é para a chamada de rotinas ou procedimentos, ou seja, trechos de programa que podem ser acessados a partir de vários pontos do programa principal. A chamada de procedimentos, ao contrário de um desvio normal, requer que um endereço de retorno seja armazenado em algum local, para que o processador possa retornar para o ponto do programa principal de onde a rotina foi chamada. Assim, esse endereço armazenado é utilizado para atualizar o valor do PC para retornar de uma rotina, voltando para o endereço da instrução imediatamente depois daquela que originalmente chamou a rotina.

Diversas formas podem ser utilizadas para armazenar este endereço de retorno, sendo que uma delas guarda este endereço em uma estrutura dados na memória, chamada de pilha, onde os elementos são retirados sempre na ordem inversa com que são colocados, ou seja, os últimos elementos colocados são os primeiros a serem retirados e vice-versa. Existe então um registrador especial, chamado de um apontador de pilha (SP), que aponta sempre para o último elemento que está no topo da pilha e que é automaticamente atualizado quando os elementos são inseridos ou retirados

da pilha. Uma característica especial da pilha é que ela é "cresce"no sentido inverso dos endereços de memória, do final para o começo da memória. Quando um valor é inserido na pilha, o apontador de pilha é decrementado, antes da movimentação dos dados, de um valor igual ao tamanho do dado a ser colocado na pilha. Quando um valor é retirado da pilha, o apontador é incrementado, depois da movimentação dos dados, de um valor igual ao tamanho do dado retirado na pilha.

2.2 Entrada e Saída

Depois que os resultados finais são calculados por um programa, eles são transferidos da memória para os dispositivos de entrada e saída, quando serão traduzidos para o ser humano na forma de caracteres, imagens ou sons; armazenados em dispositivos como discos ópticos ou magnéticos; ou transmitidos para outros computadores e dispositivos pela internet. A interação do processador com os dispositivos de entrada e saída se dá de duas formas básicas:

- por *loop de status*
- por interrupção

Nos processadores mais simples a única forma de comunicação disponível é por *loop de status*, onde o processador verifica periodicamente a situação dos dispositivos de entrada e saída, normalmente através de instruções especiais de E/S acessando registradores em um espaço de endereçamento reservado para os dispositivos de E/S. Esses registradores indicam quando o processador pode transferir os dados da memória para esses dispositivos de E/S e vice-versa, e eles são atualizados adequadamente pelos dispositivos de E/S conforme o desenrolar dessas operações de E/S.

Nos processadores mais modernos, linhas de interrupção conectam o processador aos dispositivos de E/S. Essas linhas são ativadas para informar ao processador o término das operações de E/S que foram solicitadas previamente. A sua ativação faz com que o processador interrompa a execução do programa atual, desviando automaticamente para rotinas especiais que vão realizar o tratamento adequado desta interrupção.

A seguir realizamos uma breve explanação de como os processadores podem ser programados para realizar as tarefas que desejamos.

2.3 Programando com Linguagem de Montagem

O processador entende apenas a linguagem de máquina, que está em formato binário, inacessível para o ser humano, composta com códigos formados pelos bits ’0’ e ’1’. Cada tipo de processador usa códigos diferentes dos demais, e a quantidade de operações disponíveis também varia imensamente. Em outras palavras, se escrevermos um programa na linguagem de máquina para um processador do tipo X, ele provavelmente não vai ser compatível com outro processador do tipo Y.

Para superar essa barreira no uso da linguagem de máquina, existe um programa tradutor de uma linguagem com complexidade bem menor, chamada de linguagem de montagem (*assembly language*), no formato de arquivos de texto. A linguagem de montagem, apesar de ainda específica para cada tipo de processador, tornam legíveis para aos humanos os programas de computador.

Nos arquivos em linguagem de montagem cada instrução que o processador consegue executar é representada por uma abreviatura, chamada de mnemônico. Em outras palavras, ao invés de escrever ’00110000’ para a indicar uma operação de soma, escrevemos algo bem mais simples como **ADD**. As posições de memória também podem ser associadas a nomes, ao invés de endereços binários.

Vejamos no exemplo a seguir um pequeno trecho em linguagem de montagem do processador Sapiens.

```
ORG 0
X:  DB  5
INICIO:
    LDA  X
    ADD  #1
```

```
    STA  X
    HLT
;
END INICIO
```

Exemplo 2.1: Primeiro Exemplo

As instruções vão sendo executadas linha após linha. Vejamos a seguir:

- Na primeira linha deste programa a instrução **LDA** (*load accumulator*) pede para executar uma operação de carga do conteúdo da variável X da memória para o acumulador;
- Depois a instrução **ADD** pede para somar o conteúdo do acumulador com o dado imediato ’1’, guardando o resultado no acumulador. Note que o acumulador é um operando IMPLÍCITO desta instrução, ou seja, não há nenhuma referência explícita ao acumulador no mnemônico da instrução;
- Finalmente a instrução **STA** (*store accumulator*) guarda o resultado desta soma, transferindo o conteúdo do acumulador para a variável X na memória. Novamente o acumulador é um operando implícito;
- Ao encontrar a instrução **HLT** (*halt*) o processador termina a execução do programa.

No exemplo acima definimos uma posição de memória (que apelidamos de X), que inicialmente contém o valor 5. Ao final deste processo a variável X terá o valor 6. Note que se antes de uma instrução encontramos uma palavra seguida por dois pontos, queremos dizer que sua posição na memória, que é um endereço binário, poderá ser representada por esta palavra.

O programa mostrado acima poderia ser facilmente descrito na linguagem de programação de alto nível como:

X = 5 ;

X = X + 1;

O compilador faz a leitura desse pequeno programa, traduzindo-o

primeiramente para a linguagem de montagem e depois para a linguagem de máquina desejada.

Maiores detalhes sobre a linguagem de montagem do processador Sapiens estão no Capítulo 3.4. A seguir apresentamos mais detalhes sobre arquitetura do processador Sapiens.

Capítulo 3

Processador Sapiens

3.1 Apresentação

A arquitetura do processador Sapiens pode ser facilmente associada a uma evolução do processador Neander, uma arquitetura baseada em acumulador (veja mais detalhes na Seção A.1) que, com sua simplicidade, é suficiente para introduzir de forma suave os aspectos centrais do funcionamento e programação de um processador. Ao longo dos anos, porém, observamos diversas limitações quando problemas um pouco mais ”reais” eram apresentados aos alunos. Uma análise do seu conjunto de instruções torna claro que muitas operações usuais (como chamada de rotinas, indexação, ponteiros, etc) são difíceis ou mesmo impossíveis de serem implementadas. Algumas características do processador original Neander eram:

- O apontador de instruções com apenas 8 bits (PC);
- Como consequência uma quantidade ínfima de memória (256 *bytes*), o que limitava o tamanho do programa ou dos dados processados;
- Grande dificuldade de realizar chamadas de rotinas pela ausência de instruções e mecanismos convenientes;
- Um registrador de código de condição com apenas 2 bits: negativo (N) e zero (Z).
- Ausência de *flags* para suporte a aritmética multi-byte;
- Apenas um modo de endereçamento: o modo direto (absoluto).

Desta forma, problemas interessantes acabavam não sendo abordados, seja pela falta de recursos ou pelo tempo necessário para desenvolvê-los. Para tornar possível atender aos requisitos de flexibilidade de programação desejados, porém mantendo a facilidade de ensino e sem fazer uso de outras arquiteturas, foi necessário estender o conjunto de instruções da máquina original Neander para incluir alguns detalhes na sua arquitetura. A arquitetura estendida, denominada Sapiens, mostrada na Figura 3.1, incluiu, entre outras, as seguintes melhorias:

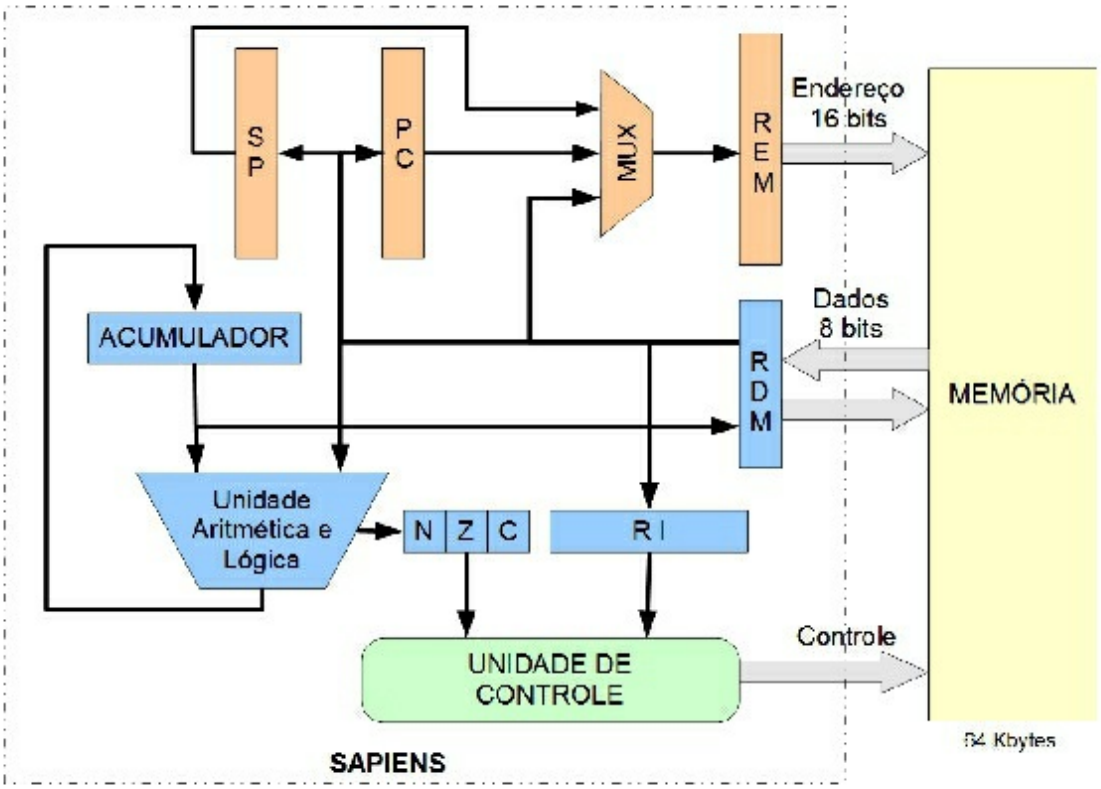

Figura 3.1: Processador Sapiens

- O modo imediato de acesso aos operandos, simplificando operações de atribuição de dados;
- Um modo indireto de endereçamento, possibilitando exercitar as noções de indexação e ponteiros – que são básicas para entendimento de qualquer estrutura básica de programação;
- Operações de entrada e saída de dados para dispositivos E/S, em espaço de endereçamento separado da memória;
- Incremento da largura do apontador de instruções (PC) para 16 bits, permitindo endereçar uma memória de até 64 Kbytes.
- Um apontador de pilha (SP, do inglês *stack pointer*), também de 16 bits, para possibilitar a chamada e o retorno de rotinas e procedimentos;
- Um código de condição (*flag*) **C** (do inglês *carry*) para o vai-um e também vem-um;

- Uma instrução de **TRAP** para chamada do simulador para realizar operações mais elaboradas de E/S.
- Um conjunto novo de instruções de movimentação de pilha, deslocamento do registrador, soma e subtração com *vai-um/vem-um*, entre outras.

Nas seções seguintes são apresentadas as instruções do processador Sapiens, seu formato e uma descrição do seu funcionamento.

3.2 Conjunto de Instruções

3.2.1 Formato e Modos de Endereçamentos

Figura 3.2: Formato das Instruções

As instruções em linguagem de máquina do processador Sapiens podem ter um, dois ou três *bytes* , conforme pode ser visto na Figura 3.2. Nas instruções, o primeiro *byte* (8 bits) sempre contém o código de operação nos 6 bits mais significativos e, quando for o caso, o modo de endereçamento nos 2 bits menos significativos. As instruções com apenas um *byte* não tem outro operando além do acumulador, que é um operando implícito para quase todas as instruções. As instruções com dois *bytes* são aquelas que, além do

acumulador, usam um dado imediato de 8 bits como operando no segundo *byte* da instrução.

Nas instruções com 3 *bytes* os dois últimos *bytes* podem conter o endereço de memória do operando (modo direto), ou o endereço de memória do ponteiro para o operando (modo indireto) ou ainda o próprio operando (modo imediato de 16 bits). Note que nos dois primeiros modos de endereçamento o operando final em memória possui apenas um *byte* de comprimento. A codificação para o modo de endereçamento é a seguinte:

- **00 - Direto:** o segundo e terceiro *bytes* da instrução contêm o endereço do operando na memória;

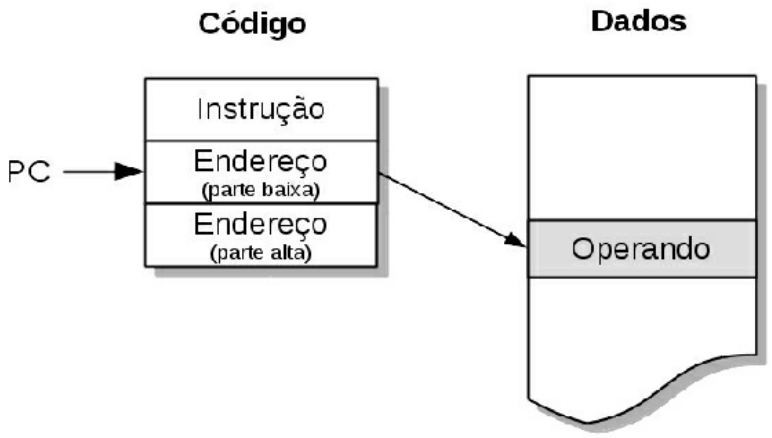

Figura 3.3: Modo Direto

- **01 - Indireto:** o segundo e terceiro *bytes* da instrução contêm o endereço da posição de memória com o endereço do operando (ou seja, é o endereço do ponteiro para o operando). Na linguagem de montagem, usou-se como convenção para indicar que um operando é indireto precedê-lo pela letra "@"(arrôba);

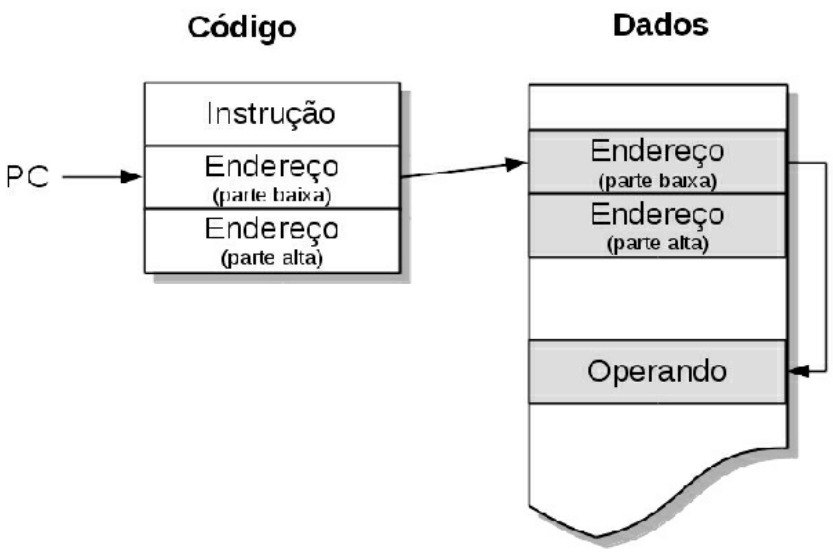

Figura 3.4: Modo Indireto

- **10- Imediato 8 bits:** o segundo *byte* da instrução é o próprio operando. Na linguagem de montagem, usou-se como convenção para indicar que um operando é indireto precedê-lo pela letra "#"(tralha).

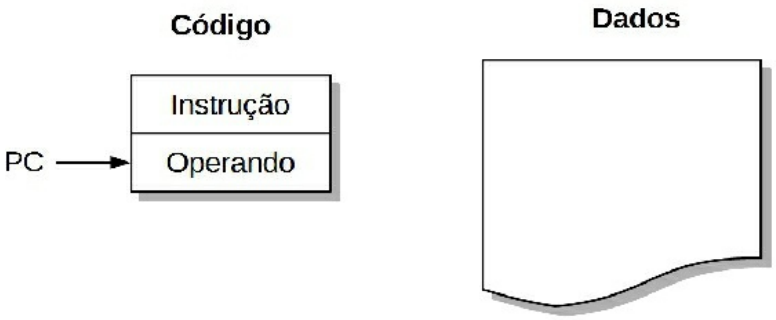

Figura 3.5: Modo Imediato

- **11 - Imediato 16 bits:** os dois *bytes* seguintes à instrução são utilizados como operando. Na linguagem de montagem, usou-se como convenção para indicar que um operando é indireto precedê-lo pela letra "#"(tralha). O compilador fica encarregado de gerar o operando no tamanho correto. A única instrução que utiliza este modo é a **LDS** (*load stack pointer*).

Maiores detalhes sobre modos de endereçamento que as instruções possuem nos processadores em geral podem ser vistos no Apêndice A.2.

3.2.2 Códigos de Condição

A seguir são apresentados os códigos de condição do processador Sapiens, ou seja, *flags* que indicam o resultado da última operação realizada pela UAL.

- **N** – (negativo): sinal do resultado
  - 1 – o resultado é negativo
  - 0 – o resultado não é negativo
- **Z** – (zero): indica resultado igual a zero
  - 1 – o resultado é igual a zero
  - 0 – o resultado diferente de zero
- **C** – (vai-um): indica que a última operação resultou em ”vai-um” (*carry*), no caso de soma, ou ”vem-um” (*borrow*) em caso de subtração.
  - 1 – o resultado deu ”vai-um” ou ”vem-um”.
  - 0 – o resultado não deu nem ”vai-um” ou ”vem-um”.

3.2.3 Descrição das Instruções

O conjunto original de instruções foi expandido para permitir uma maior capacidade de processamento. Todas as instruções originais do Neander-X foram mantidas, com exceção da instrução **LDI** (*load immediate*), que ainda é aceita pelo novo montador, mas na geração do programa em linguagem de máquina foi substituída pela instrução **LDA #imed**, que realiza a mesma função. Ou seja, um código escrito para os processadores Neander ou Neander-X é totalmente compatível e executado sem problemas no SimuS.

Como destaque para as novas instruções introduzidas na arquitetura temos:

- adição e subtração com carry (**ADC e SBC**);
- novas instruções lógicas como “ou” exclusivo, deslocamento para a direita e esquerda do acumulador (**XOR, SHL, SHR e SRA**);
- novas instruções de desvio condicional (**JP, JC, JNC**);
- instruções para chamada e retorno de procedimento (**JSR e RET**);
- instruções para a manipulação da pilha (**PUSH e POP**);
- a instrução especial **TRAP** para pedidos de serviço ao simulador;
- instruções de carga e armazenamento do apontador de pilha (SP) em

memória (**LDS e STS**)

A seguir apresentamos as instruções do Sapiens com os respectivos códigos de operação.

1. **NOP** (código 0000 00–)

   A instrução NOP não faz nada. Essa instrução tem um tamanho igual a um byte.

2. **STA ender ou @ender** (código 0001 00-X)

   A instrução **STA** armazena o conteúdo do acumulador (8 bits) na posição de memória definida pelo endereço *ender* de 16 bits, acessada no modo direto ou indireto. Essa instrução tem um tamanho igual a 3 bytes.

3. **STS ender ou @ender** (código 0001 01-X)

   A instrução **STS** armazena o conteúdo do apontador de pilha (SP), de 16 bits, na posição de memória definida pelo endereço *ender* de 16 bits, acessado no modo direto ou indireto. Essa instrução tem um tamanho igual a 3 bytes.

4. **LDA #imed, ender ou @ender** (código 0010 00XX)

   A instrução **LDA** carrega um byte no acumulador, que pode ser lido como operando imediato de 8 bits ou de uma posição de memória definida pelo endereço *ender* de 16 bits, acessada no modo direto ou indireto. Os *flags* **N e Z** são modificados de acordo com o valor carregado no acumulador. Essa instrução tem um tamanho igual a 2 ou 3 bytes.

5. **LDS #imed16, ender ou @ender** (código 0010 01XX)

   A instrução **LDS** carrega dois bytes no apontador de pilha (SP), que podem lidos como operando imediato de 16 bits ou de uma posição de memória definida pelo endereço *ender* de 16 bits, acessada no modo direto ou indireto. Essa instrução tem um tamanho igual a 3 bytes.

6. **ADD #imed, ender ou @ender** (código 0011 00XX)

A instrução **ADD** soma o acumulador com um byte, que pode ser lido como operando imediato de 8 bits ou de uma posição de memória definida pelo endereço *ender* de 16 bits, acessada no modo direto ou indireto. Os *flags* **N, Z e C** são modificados de acordo com o resultado da operação. Essa instrução tem um tamanho igual a 2 ou 3 bytes.

7. **ADC #imed, ender ou @ender** (código 0011 01XX)

A instrução **ADC** soma o acumulador com o *Carry* (*flag* C) e com um byte, que pode ser lido como operando imediato de 8 bits ou de uma posição de memória definida pelo endereço *ender* de 16 bits, acessada no modo direto ou indireto. Os *flags* **N, Z e C** são modificados de acordo com o resultado da operação. Essa instrução tem um tamanho igual a 2 ou 3 bytes.

8. **SUB #imed, ender ou @ender** (código 0011 10XX)

A instrução **SUB** subtrai o acumulador de um byte, que pode ser lido como operando imediato de 8 bits ou de uma posição de memória definida pelo endereço *ender* de 16 bits, acessada no modo direto ou indireto. Os *flags* **N, Z e C** são modificados de acordo com o resultado da operação. Essa instrução tem um tamanho igual a 2 ou 3 bytes.

9. **SBC #imed, ender ou @ender** (código 0011 11XX)

A instrução **SUB** subtrai o acumulador do *Carry* (*flag* C) e de um byte,que pode ser lido como operando imediato de 8 bits ou de uma posição de memória definida pelo endereço *ender* de 16 bits, acessada no modo direto ou indireto. Os *flags* **N, Z e C** são modificados de acordo com o resultado da operação. Essa instrução tem um tamanho igual a 2 ou 3 bytes.

10. **OR #imed, ender ou @ender** (código 0100 00XX)

A instrução **OR** realiza um ”ou” bit a bit entre o acumulador e um byte,

que pode ser um dado imediato de 8 bits ou lido de uma posição de memória definida pelo endereço *ender* de 16 bits, acessada no modo direto ou indireto. acessado no modo direto ou indireto. Os *flags* **N e Z** são modificados de acordo com o resultado da operação. Essa instrução tem um tamanho igual a 2 ou 3 bytes.

11. **XOR #imed, ender ou @ender** (código 0100 01XX)

A instrução **XOR** realiza um ”ou exclusivo” bit a bit entre o acumulador e um byte, que pode ser um dado imediato de 8 bits ou lido de uma posição de memória definida pelo endereço *ender* de 16 bits, acessada no modo direto ou indireto. acessado no modo direto ou indireto. Os *flags* **N e Z** são modificados de acordo com o resultado da operação. Essa instrução tem um tamanho igual a 2 ou 3 bytes.

12. **AND #imed, ender ou @ender** (código 0101 00XX)

A instrução **AND** realiza um ”e” bit a bit entre o acumulador e um byte, que pode ser um dado imediato de 8 bits ou lido de uma posição de memória definida pelo endereço *ender* de 16 bits, acessada no modo direto ou indireto. acessado no modo direto ou indireto. Os *flags* **N e Z** são modificados de acordo com o resultado da operação. Essa instrução tem um tamanho igual a 2 ou 3 bytes.

13. **NOT** (código 0110 00–)

A instrução **NOT** complementa (’0’ → ’1’ e ’1’ → ’0’) os bits do acumulador. Os *flags* **N e Z** são modificados de acordo com o resultado da operação. Essa instrução tem um tamanho igual a um byte.

14. **SHL** (código 0111 00–)

A instrução **SHL** (shift left) realiza o deslocamento do acumulador de um bit para a esquerda, através do *carry*. O bit mais significativo ao ser deslocado para fora do acumulador é colocado no *Carry* (*flag* C). São inseridos ’0’s no bit menos significativo. Os *flags* **N e Z** são modificados de acordo com o resultado da operação. Essa instrução tem um tamanho igual a um byte.

15. **SHR** (código 0111 01–)

A instrução **SHR** (shift right) realiza o deslocamento do acumulador de um bit para a direita através do *carry*. O bit menos significativo ao ser deslocado para fora do acumulador entra no *Carry* (*flag* C). São inseridos ’0’s no bit mais significativo. Os *flags* **N e Z** são modificados de acordo com o resultado da operação. Essa instrução tem um tamanho igual a um byte.

16. **SRA** (código 0111 10–)

A instrução **SRA** (shift right arithmetic) realiza o deslocamento do acumulador de um bit para a direita através do *carry*. O bit menos significativo que sai à direita do acumulador entra no *Carry* (*flag* C). O bit mais significativo (de sinal) é repetido à esquerda, de modo que um número negativo em complemento a 2 continua sempre negativo. Os *flags* **N e Z** são modificados de acordo com o resultado da operação. Essa instrução tem um tamanho igual a um byte.

17. **JMP ender ou @ender** (código 1000 00-X)

A instrução **JMP** (*jump*) altera o valor do PC e desvia a execução do programa para o endereço de 16 bits definido pelo seu operando, que pode ser acessado no modo direto ou indireto. Essa instrução tem um tamanho igual a 3 bytes.

18. **JN ender ou @ender** (código 1001 00-X)

A instrução **JN** (*jump if negative*) altera o valor do PC e desvia a execução do programa para o endereço de 16 bits definido pelo seu operando, que pode ser acessado no modo direto ou indireto, apenas se a última operação realizada produziu um valor negativo (*flag* N em ’1’). Essa instrução tem um tamanho igual a 3 bytes.

19. **JP ender ou @ender** (código 1001 01-X)

A instrução **JP** (*jump if positive*) altera o valor do PC e desvia a execução

do programa para o endereço de 16 bits definido pelo seu operando, que pode ser acessado no modo direto ou indireto, apenas se a última operação realizada produziu um valor positivo (*flags* N e Z em ’0’). Essa instrução tem um tamanho igual a 3 bytes.

20. **JZ ender ou @ender** (código 1010 00-X)

A instrução **JZ** (*jump if zero*) altera o valor do PC e desvia a execução do programa para o endereço de 16bits definido pelo seu operando, que pode ser acessado no modo direto ou indireto, apenas quando a última operação realizada produziu um valor igual a zero (*flag* Z em ’1’). Essa instrução tem um tamanho igual a 3 bytes.

21. **JNZ ender ou @ender** (código 1010 01-X)

A instrução **JNZ** (*jump if not zero*) altera o valor do PC e desvia a execução do programa para o endereço de 16 bits definido pelo seu operando, que pode ser acessado no modo direto ou indireto, apenas quando a última operação realizada produziu um valor diferente de zero (*flag* Z em ’0’). Essa instrução tem um tamanho igual a 3 bytes.

22. **JC ender ou @ender** (código 1011 00-X)

A instrução **JC** (*jump if carry*) altera o valor do PC e desvia a execução do programa para o endereço de 16 bits definido pelo seu operando, que pode ser acessado no modo direto ou indireto, apenas quando a última operação realizada produziu um ”vai-um” ou ”vem-um” (*flag* C em ’1’). Essa instrução tem um tamanho igual a 3 bytes.

23. **JNC ender ou @ender** (código 1011 01-X)

A instrução **JNC** (*jump if not carry*) altera o valor do PC e desvia a execução do programa para o endereço de 16 bits definido pelo seu operando, que pode ser acessado no modo direto ou indireto, apenas quando a última operação realizada não produziu um ”vai-um” ou ”vem-um” (*flag* C em ’0’). Essa instrução tem um tamanho igual a 3 bytes.

24. **IN ender8** (código 1100 00–)

A instrução **IN** (*input*) carrega no acumulador o valor lido de um dispositivo de E/S no endereço de 8 bits indicado pelo operando *ender8*, apenas no modo direto. Essa instrução tem um tamanho igual a 2 bytes.

25. **OUT ender8** (código 1100 01–)

A instrução **OUT** (*output*) descarrega o conteúdo do acumulador em um dispositivo externo com o endereço de E/S de 8 bits indicado pelo operando *ender8*, apenas no modo direto. Essa instrução tem um tamanho igual a 2 bytes.

26. **JSR ender ou @ender** (código 1101 00-X)

A instrução **JSR** (*jump to subroutine*) altera o valor do PC e desvia a execução do programa para o endereço *ender* de 16 bits, que pode ser acessado no modo direto ou indireto, salvando antes o endereço da próxima instrução (endereço de retorno) no topo da pilha. O valor do apontador de pilha é modificado (SP = SP - 2). Essa instrução tem um tamanho igual a 3 bytes.

27. **RET** (código 1101 01–)

A instrução **RET** (*return*) retorna de uma rotina, transferindo para o PC o endereço de retorno que está no topo da pilha e atualizando o apontador de pilha (SP = SP + 2). Essa instrução tem um tamanho igual a um byte.

28. **PUSH** (código 1110 00–)

A instrução **PUSH** coloca o conteúdo do acumulador no topo da pilha, atualizando antes o apontador de pilha (SP = SP - 1). Essa instrução tem um tamanho igual a um byte.

29. **POP** (código 1110 01–)

A instrução **POP** retira o valor que está no topo da pilha e coloca no acumulador, atualizando em seguida o apontador de pilha (SP = SP + 1). Os *flags* **N e Z** são modificados de acordo com o valor carregado no acumulador. Essa instrução tem um tamanho igual a um byte.

30. **TRAP ender ou @ender** (código 1111 00-X)

A pseudo instrução **TRAP** é utilizada para emulação de rotinas de E/S pelo simulador. O tipo de serviço solicitado é passado como parâmetro no acumulador. O operando *ender* de 16 bits define, no modo direto ou indireto, o endereço de memória para a passagem de parâmetros adicionais.

31. **HLT** (código 1111 11–)

A instrução HLT (*halt*) para a máquina. Essa instrução tem um tamanho igual a um byte.

As instruções lógicas e aritméticas (**ADD, ADC, SUB, SBC, NOT, AND, OR, XOR, SHL, SHR, SRA**) e as instruções de transferência **LDA, LDS e POP** afetam apenas os códigos de condição N e Z de acordo com o resultado produzido. As instruções lógicas e aritméticas (**ADD, ADC, SUB, SBC, SHL, SHR, SRA**) afetam também o código de condição Carry de acordo com o resultado produzido. As demais instruções (**STA, STS, JMP, JN, JP, JZ, JNZ, JC, JNC, JSR, RET, PUSH, IN, OUT, NOP, TRAP e HLT**) não alteram os códigos de condição. Um resumo pode ser visto na Figura 3.6.

Instrução	Códigos de Condição
LDA	N, Z
ADD	N, Z, C
SUB	N, Z, C
ADC	N, Z, C
SBC	N, Z, C
AND	N, Z
OR	N, Z
XOR	N, Z
NOT	N, Z
SHR, SRA	N, Z, C Obs. carry = bit menos significativo (deslocado para fora do acumulador)
SHL	N, Z, C Obs. carry = bit mais significativo (deslocado para fora do acumulador)
POP	N, Z

Figura 3.6: Instruções e o Código de Condição

O simulador SimuS conta com os seguintes dispositivos de entrada e saída, acessíveis através das instruções IN e OUT:

- Endereço 0 - Visor hexadecimal (**OUT**) e dado do painel de chaves (**IN**).
- Endereço 1 - O *status* de ”dado disponível” no painel de chaves (**IN**).
- Endereço 2 - Visor de uma linha com 16 caracteres (**OUT**) e dado do teclado de 12 teclas (**IN**).
- Endereço 3 - O *satus* de ”dado disponível” no teclado (**IN**) e limpa o visor de 16 caracteres (**OUT**).

3.3 Operações de E/S com a instrução TRAP

A necessidade de ensinar técnicas de entrada e saída em um momento preliminar do ensino sempre foi uma das mais problemáticas observadas em nossa experiência didática. Mesmo em qualquer problema menos trivial, havia a necessidade de realizar leituras e escritas de dados, porém isso exigia do aluno conhecimentos específicos ainda não aprendidos nas primeiras aulas (laços, testes, *polling*, *flags*, etc.). Desta forma se tornava quase mandatório que os primeiros programas fossem centrados em resolver problemas abstratos usando valores pré-carregados na memória.

Para resolver este problema de E/S, optamos por criar uma abstração, similar àquela apresentada em processadores reais para chamar funções de sistema operacional: a operação **TRAP**. Em nosso caso, porém, o sistema operacional é algo abstrato, sendo que na verdade essas funcionalidades são implementadas diretamente no simulador.

Neste mecanismo a instrução **TRAP** é interpretada pelo simulador como um pedido para que ele realize uma determinada operação de E/S. O número do *trap*, ou seja, a função que está sendo solicitada é passada no acumulador. Como o processador Sapiens é carente de registradores, a instrução **TRAP** tem um operando adicional que é o endereço de memória onde os parâmetros adicionais necessários são passados para o módulo do simulador responsável pela realização da operação. Temos previsão para suportar inicialmente as seguintes operações:

- 1 – Leitura de um caractere da console. O código ASCII correspondente é colocado no acumulador.
- 2 – Escrita de um caractere que está no endereço definido pelo operando

da instrução **TRAP** na console. O caractere escrito é retornado no acumulador.

- 3 – Leitura de uma linha inteira da console para o endereço definido pelo operando da instrução **TRAP**. O tamanho da cadeia de caracteres lida é retornado no acumulador, com no máximo 255 caracteres sem o fim de linha ou retorno de carro.
- 4 – Escrita de uma linha inteira na console. O operando da instrução **TRAP** contém o endereço para a cadeia de caracteres que deve ser terminada pelo caractere NULL (00H). O tamanho máximo da cadeia é de 255 caracteres. O número total de caracteres escritos é retornado no acumulador.
- 5 – Chama uma rotina de temporização. O operando da instrução **TRAP** contém o endereço da variável inteira com o tempo a ser esperado em milissegundos.
- 6 – Chama uma rotina para tocar um tom. A frequência e a duração do tom estão em duas variáveis inteiras de 16 bits no endereço definido pelo operando da instrução **TRAP**.
- 7 – Chama uma rotina para retornar um número pseudoaleatório entre 0 e 99 no acumulador.
- 8 – O operando da instrução **TRAP** tem o endereço com a variável com a semente inicial da rotina de número aleatórios.

3.4 Linguagem de Montagem do Sapiens

3.4.1 Aspectos Gerais

Foi definida uma linguagem de montagem (*assembly language*) para este processador obedecendo às regras usualmente encontradas nos sistemas comerciais e compatíveis com a sintaxe previamente utilizada no simulador Neanderwin. Na maioria são mnemônicos e comandos com sintaxe simplificada e de fácil utilização. A seguir um exemplo de programa em linguagem de montagem para o processador Sapiens:

a) **Formato geral das linhas:**

Uma linha pode conter alguns dos seguintes elementos: um rótulo, um operador ou uma pseudo-instrução, um operando opcional e comentários.

São permitidas linhas vazias.

b) **Comentários no programa**

Os comentários são começados por ponto e vírgula, e podem também ocorrer no final das linhas com instruções ou pseudo-instruções.

c) **Rótulos**

Um rótulo é um nome dado à próxima posição de memória. Os rótulos são construídos segundo as regras a seguir:

- Usam letras alfabéticas ou numéricas ou $ ou _ mas não espaços.
- A primeira letra não pode ser numérica.
- Não existe distinção entre maiúsculas e minúsculas (ou seja ’a’ é o mesmo que ’A’).
- Pode ser acrescido de um "+"e um valor decimal, para referenciar o segundo byte de uma dados de 16 bits ou um elemento de um vetor.
- O rótulo deve ser seguido por dois pontos. A única exceção é rótulo utilizado na pseudo-instrução **EQU**.

d) **Pseudo Instruções**

Pseudo instruções são orientações que o programador passa para o montador, com o intuito de organizar e posicionar o código e variáveis na memória do simulador.

- **ORG ender** – **ORG** (*origin*) indica ao montador que a próxima instrução ou dado será colocado na posição *ender* de memória. Por exemplo:

```
ORG 0

...

Instruções

...

ORG 100

...

Dados
```

```
...

END 0
```

- **var EQU imed** – **EQU** (*equate*) atribui um nome (rótulo) a um determinado valor. Entre muitos usos possíveis, esse comando pode ser usado para especificar variáveis que são posicionadas em um endereço específico de memória. Pode ser utilizado também para definir constantes, por exemplo:

```
INC    EQU    2

MAX    EQU    99

MIN    EQU    1000
```

- **END ender** – **END** indica que o programa fonte acabou. O operando *ender* é usado para pré-carregar o PC com o endereço inicial do programa, ou seja, quando o programa é carregado na memória para execução, esse valor é carregado no PC para indicar o endereço inicial de execução.
- **DS imed** – **DS** (*define storage*) reserva um número de *bytes* na memória definido pelo operando *imed*, sem nenhum valor inicial.

```
A:     DS 1

VETOR: DS 10
```

- **DB imed** – **DB** (*define byte*) carrega esta posição de memória com o valor definido pelo operando *imed*.

```
A:      DB 20

VETOR:  DB 1, 2, 3, 4, 5, 6, 7, 8, 9
```

- **DW imed16** – **DW** (*define word*) carrega esta posição de memória e a seguinte com o valor definido pelo operando *imed16*.

```
A:          DW 3200

PONT:       DW 1000

VETOR:      DW 1000, 1001, 1002, 1003, 1004, 1005

PONTEIRO:   DW VETOR
```

- **STR "cadeia"** – **STR** (*string*) Carrega na posicao de memória e nas seguintes o(s) valor(es) do código ASCII correspondente aos caracteres da cadeia entre aspas.

```
MSG:    STR  "Mensagem de aviso"
```

```
        DB 00
NOVA:   STR "Outra mensagem"
        DB 00
```

3.4.2 Representação de números

Os números positivos devem ser representados sem uso do sinal +. Por exdmplo, o número decimal **48** teria as seguintes representações possíveis:

- Decimal: **48**
- Hexadecimal: **30h**
- Binário: **00110000b**
- Obs: Números hexadecimais maiores que **7Fh** devem ser precedidos por um zero, p. ex. **0F3h**.

Os números negativos decimais são aceitos pelo montador, devendo ser precedidos de um sinal negativo e são representados em complemento a 2. Se estiver utilizando a diretiva DB são aceitos números entre 255 e -128. Caso a diretiva utilizada seja DW, os valores admitidos estão entre 65535 e -32768. Note que o bit mais significativo é utilizado para o sinal caso o número seja negativo. Os números negativos devem ser tratados com especial atenção pelo programador ao realizar operações aritméticas.

O número decimal **-48** teria as seguintes representações possíveis:

- Decimal: **-48**
- Hexadecimal: **0D0h**
- Binário: **11010000b**

No capítulo seguinte apresentamos uma série de exemplos de programação com a linguagem de montagem do Sapiens.

Capítulo 4

Primeiros Exemplos de Programação

Mostraremos neste capitulo uma série de situações de programação bem simples. Inicialmente os trechos de programa são descritos em uma pseudo-linguagem de programação de alto nível, que acreditamos ser de fácil entendimento, e são em seguida traduzidas para a linguagem de montagem do Sapiens.

4.1 Atribuições de Variáveis

As atribuições consistem em mover certo valor para uma variável. Este valor pode ser uma constante ou outra variável.

O processador Sapiens, como foi explicado anteriormente, realiza suas operações através de um mecanismo de *hardware* muito simples de implementar: faz uso de um acumulador, que é um registrador capaz de armazenar uma das parcelas das operações aritméticas, sendo que a outra parcela da operação é buscada direto na memória ou é um dado imediato, embutido na própria instrução.

Para carregar o acumulador com um determinado valor temos duas opções:

a) Buscar este valor de uma posição de memória usando a instrução **LDA** (*load from address*) com o modo de endereçamento direto ou indireto;
b) Carregar um valor imediato com o uso da instrução **LDA** (*load from address*) com o modo de endereçamento imediato, em que o valor está armazenado na própria instrução.

Mostremos a seguir alguns casos.

4.1.1 Atribuição de uma Constante

O valor de uma constante é atribuído a uma variável:

A = 10

Em uma primeira solução, vista no Exemplo 4.1, criamos uma variável em memória (chamada DEZ) com o valor inicial igual a 10 definido pelo montador. Ao executar esse programa, o conteúdo desta posição de memória é carregado no acumulador, e depois esse valor é transferido para a posição de memória correspondente à variável A.

```
ORG 0
    LDA DEZ ; Acumulador = 10
    STA A   ; A = Acumulador
    HLT     ; Termina a execução

DEZ: DB  10 ; Variável DEZ
            ; com valor inicial 10
A:   DS  1  ; Variável A sem valor
            ; inicial
END 0
```

Exemplo 4.1: Exemplo de Atribuição de Constante

Como dito, **DB** é uma pseudo-instrução usada para indicar ao montador que deve deixar um espaço na memória e atribuindo um valor inicial nesta posição. Já **DS** é uma pseudo-instrução usada para indicar ao montador que deve deixar um espaço na memória sem nenhum valor inicial. Cada posição de memória será associada a uma variável, com o mesmo nome do rótulo, que em tempo de execução pode permanecer intocada ou ser modificada.

Em uma segunda solução, vista no Exemplo 4.2, usamos a instrução **ADD #imed**, onde o valor a ser carregado no acumulador está embutido no segundo *byte* da própria instrução. Repare que esta solução tem o mesmo

número de instruções, mas usa uma variável a menos na memória.

```
ORG 0
    LDA  #10 ; Acumulador = 10
    STA  A   ; A = Acumulador
    HLT      ; Termina a execução

ORG 100
A:  DS  1    ; Variável A sem valor
             ; inicial
END 0
```

Exemplo 4.2: Atribuição com Dado Imediato

4.1.2 Atribuição Simples

Uma certa variável *B*, com valor inicial 7, que é atribuída à variável *A*:

B = 7

A = B

```
ORG 0
    LDA  B   ; Acumulador = B
    STA  A   ; A = Acumulador
    HLT      ; Termina a execução

ORG 100
A:  DS  1    ; A no endereço 100
B:  DB  7    ; B no endereço 101 (valor 7)
END 0
```

Exemplo 4.3: Exemplo de Atribuição Simples

O resultado da compilação pelo montador do Sapiens pode ser visto na Listagem 4.4. À esquerda aparece o número da linha do texto, seguida da posição de memória onde a instrução será colocada, seguida dos códigos da instrução e finalmente a instrução transcrita.

```
Compilação (assembly) do texto
Em 17/02/2016

  1   00     20 06     LDA B
  2   02     10 05     STA A
  3   04     F0        HLT
  4   05     00        A:   DS    1
  5   06     07        B:   DB    7
```

Exemplo 4.4: Resultado da Compilação

Vejamos a execução deste programa com maiores detalhes:

1. A memória, a partir de 00 é carregada com os valores 20 06 10 05 F0 00 07.
2. Feita a carga, o processador executa a instrução 20 06 que carrega o conteúdo da variável B no acumulador (ou seja, o valor 7).
3. Depois executa a instrução 10 05 que armazena o conteúdo do acumulador na variável A na memória.
4. Finalmente executa a instrução F0, que vai parar o processador.

4.2 Operações Aritméticas

Vejamos a seguir alguns exemplos bem simples com operações aritméticas no Sapiens.

4.2.1 Operação com uma Variável e uma Constante

Vamos considerar, como exemplo, um programa que realiza a soma de uma variável com uma constante, armazenando o resultado em uma segunda variável.

A = 3

B = A + 5

Até agora programa e dados estavam todos em endereços em sequência na memória, mas nem sempre isso pode ser interessante. Usualmente devem ser escolhidas uma área de dados e uma a área de programa, ou seja, a localização das instruções e dados na memória. Não existem critérios para essa escolha, mas deve ser observado que área de programa não pode invadir a área de dados e vice-versa. Ou seja, para esse programa, foi escolhida uma alocação de memória de tal forma que o programa ocupe a metade inferior da memória e os dados a metade superior, como segue:

- Área de programa:

  Início do programa posição 0 (0H)
- Área de dados:

  variável A na posição 128 (80H) variável B na posição 129 (81H)

A listagem 4.5 mostra este exemplo em linguagem de montagem do Sapiens.

```
ORG 0
    LDA A   ; Acumulador = A
    ADD #5  ; Acumulador = Acumulador + 5
    STA B   ; B = Acumulador
    HLT     ; Termina a execução

ORG 128
A:  DB 3    ; A no endereço 128 (valor 3)
```

```
B:  DS 1    ; B no endereço 129
END 0
```

Exemplo 4.5: Soma com uma Variável e uma Constante

Esse programa termina sua execução fazendo com que a posição de memória correspondente à variável B receba o valor 8.

4.2.2 Operação com Duas Variáveis

Vamos considerar, como exemplo, um programa que realiza a soma de 2 variáveis, armazenando o resultado em uma terceira variável.

X = 5

Y = 9

Z = X + Y

Os valores iniciais são atribuídos às variáveis X e Y com o uso das pseudo-instruções do montador, como já vimos antes. A soma é feita com a instrução de **ADD Y** após a carga do valor de X no acumulador. A atribuição do valor da soma à variável Z é feita com a instrução **STA Z**, que transfere o valor da soma para a posição de memória ocupada pela variáel Z. A listagem 4.6 mostra este exemplo em linguagem de montagem do Sapiens.

```
ORG 0
    LDA  X   ; Acumulador = X
    ADD  Y   ; Acumulador = Acumulador + Y
    STA  Z   ; Z = Acumulador
    HLT      ; Termina a execução

ORG 128
X:  DB  5    ; X no endereço 128 (valor 5)
```

```
Y:  DB  9    ; Y no endereço 129 (valor 9)
Z:  DS  1    ; Z no endereço 130
END 0
```

Exemplo 4.6: Soma com Duas Variáveis

4.2.3 Operação com Três Variáveis

Vejamos um exemplo, que faz uso de algumas variáveis: X, Y, Z, localizadas nas posições 100, 101 e 102 de memória, com valores iniciais 0, 70 e 91 previamente carregados pelo montador. As instruções do programa serão carregadas pelo montador na posição 0 de memória.

X = 0

Y = 70

Z = 91

X = Y

Z = Z + 1

X = X - 1

X = X + Y + Z

O código equivalente, com os comentários correspondentes, é mostrado a seguir.

```
ORG 100
X:  DB  0
Y:  DB  70
Z:  DB  91
```

```
ORG 0
; Faz X = Y
    LDA  Y
    STA  X
; Soma 1 à variável Z
    LDA  Z
    ADD  #1
    STA  Z
; Subtrai 1 da variável X
    LDA  X
    SUB  #1
    STA  X
; Faz X = X + Y + Z
    LDA  X
    ADD  Y
    ADD  Z
    STA  X
    HLT
END 0
```

Exemplo 4.7: Soma com Três Variáveis

4.3 Testes e Desvios

Grande parte das atividades de programação está relacionada a tomar decisões. A maior parte delas está relacionada a duas questões teóricas:

a) Verificar se uma variável é maior, menor ou igual a certo valor ou variável.
b) Verificar se alguma expressão lógica é verdadeira.

Quando realizamos estas verificações (que aqui chamaremos de testes), devemos ter um objetivo claro: quando o teste se concretizar como verdade,

iremos executar algumas coisas; quando não se concretizar (ou seja, se a situação for falsa), iremos executar outras coisas. Geralmente, depois de fazermos uma destas duas ações, o programa continuará executando outras instruções.

Em diversas linguagens de programação de alto nível, descrevemos essa situação parecida com algo assim:

```
IF  (certa condição)  THEN
   {
      Faz algumas coisas;
   }
ELSE
   {
      Faz outras coisas;
   }
Continua o programa ...
```

Uma visualização deste caso pode ser vista na Figura 4.1.

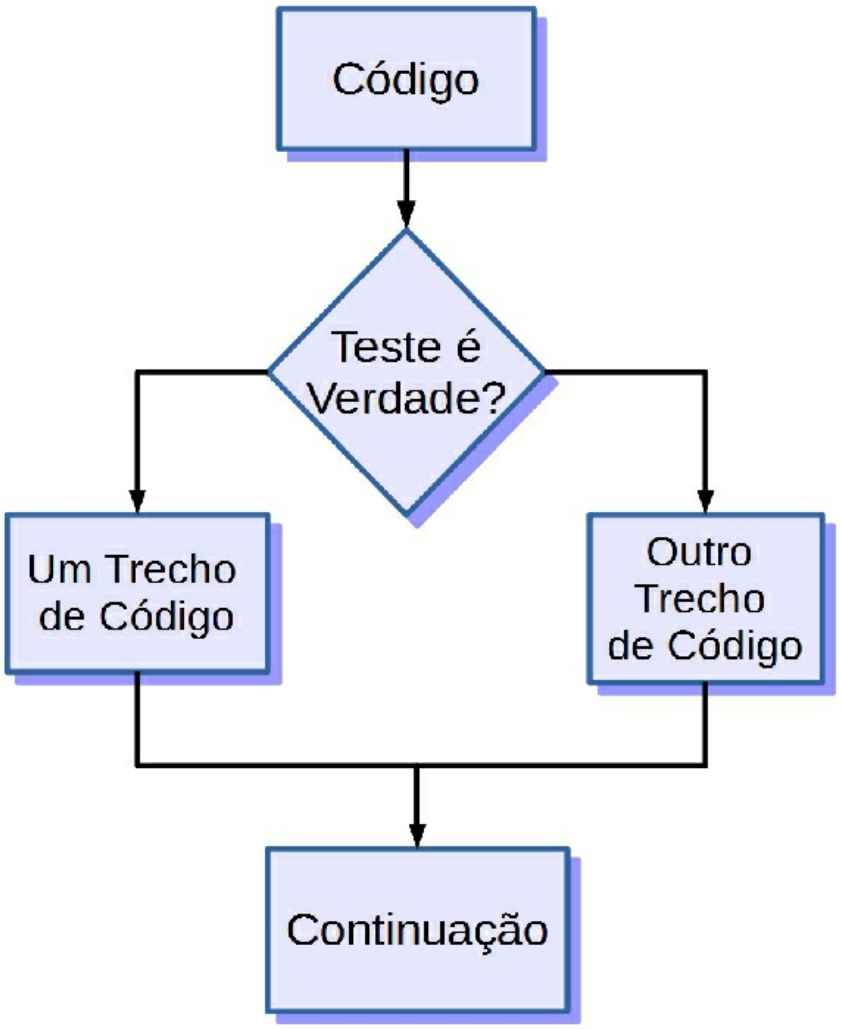

Figura 4.1: Teste e Desvios

A comparação é sempre algo que devemos prestar muita atenção, e não é tão fácil quanto pode parecer à primeira vista. É muito comum que nos enganemos neste teste.

No processador Sapiens, temos três indicadores: N, Z e C, que são associados às seguintes instruções de desvio condicional:

- JZ → desvia quando o indicador de zero estiver ativado;
- JNZ → desvia quando o indicador de zero não estiver ativado;

- JN → desvia quando o indicador de negativo estiver ativado.
- JP → desvia quando o indicador de negativo e o de zero não estiverem ativados.
- JC → desvia quando o indicador de vai-um estiver ativado.
- JNC → desvia quando o indicador de vai-um não estiver ativado.

O teste de igualdade é o mais simples de ser feito e há duas instruções que podem ser utilizadas neste caso: JZ (desvia se a subtração deu zero, indicando igualdade) e JNZ (desvia se a subtração não deu zero, indicando desigualdade).

Por exemplo, vamos desviar se as variáveis A e B forem iguais:

```
LDA  A
SUB  B
JZ  IGUAIS
DIFERENTES:
; ... Faz algumas coisas
JMP  SEGUE
IGUAIS:
; ... Faz outras coisas
SEGUE:
; ... Continua o programa
```

A instrução **JMP SEGUE** é muito importante, pois sem ela o processador iria executar os dois ramos do desvio, quando a comparação não fosse verdadeira, passando tanto por DIFERENTES como em seguida por IGUAIS. Agora vamos ver um exemplo para desviar se as variáveis A e B forem diferentes:

```
LDA  A
SUB  B
JNZ  DIFERENTES
IGUAIS:
```

```
; ... Faz algumas coisas
JMP  SEGUE
DIFERENTES:
; ... Faz outras coisas
SEGUE:
; ... Continua o programa
```

Em realidade é tudo uma questão de como se deseja organizar o código, podendo-se fazer uso de qualquer das duas formas indistintamente.

No Sapiens, para que possamos verificar se uma condição é verdadeira, devemos fazer uma operação de subtração, verificando se o resultado deu negativo usando a instrução JN (desvia se negativo). No exemplo a seguir vamos desviar se a variável A for maior do que 10:

```
ADD  #10
SUB  A
JN  MAIOR
MENOR_OU_IGUAL:
; ... Faz algumas coisas
JMP  SEGUE
MAIOR:
; ... Faz outras coisas
SEGUE:
; ... Continua o programa
```

Outro exemplo, vamos desviar para uma posição se a variável A for maior que 10, outra se for igual e uma terceira for menor que 10. Isso se torna um pouco mais complicado, já que não temos uma instrução que teste se um resultado é positivo. Podemos fazer isso de várias maneiras, como por exemplo:

```
LDA  #10
```

```
SUB  A
JZ   IGUAL
JN  MAIOR
MENOR:
; ... Faz algumas coisas
JMP  SEGUE
IGUAL:
; ... Faz outras coisas
JMP  SEGUE
MAIOR:
; ... Faz mais coisas
SEGUE:
; ... Continua o programa
```

Vejamos um exemplo para verificar se uma variável é par ou ímpar:

```
IF ((A MOD 2) == 0) THEN
  {
pares++
  }
ELSE
  {
impares++
  }
```

Como o Sapiens não possui a operação MOD (resto da divisão), o teste para verificar se um numero é par ou ímpar é feito analisando-se se o bit de ordem ’0’, para saber se o mesmo é igual a ’0’ (par) ou ’1’ (ímpar). Usamos a instrução **AND**, e caso o resultado seja ’1’ desviamos. No Exemplo 4.8 temos um programa completo, onde um valor a ser testado, que deve ser diferente de zero, é lido do painel de chaves:

```
ORG 0
INICIO:
; Faz a leitura do painel de chaves
    IN   0
    STA  A
; Se a entrada for 0 termina o programa
    AND  0
    JZ   FIM
; Testa se é par (bit 0=0)
    LDA  A
    AND  #1
    JNZ  SENAO
; Se for, faz pares++
    LDA  PARES
    ADD  #1
    STA  PARES
    JMP  INICIO
SENAO:
; Incrementa I
    LDA  IMPARES
    ADD  #1
    STA  IMPARES
    JMP  INICIO
FIM:
    HLT

ORG 100
A:      DS  1
PARES:  DS  1
```

```
IMPARES:DS  1
END 0
```

Exemplo 4.8: Par ou Ímpar?

Esse programa faz uso da instrução de E/S ”IN 0” para ler um valor do painel de chaves. Maiores detalhes sobre como realizar adequadamente as operações de E/S serão apresentados na Seção 4.5. Por enquanto execute esse programa no modo passo a passo no simulador, para que ele funcione corretamente.

4.4 Repetições

Nas atividades da vida diária há muitas coisas que temos que fazer muitas vezes. Como tudo na vida, há um fim para essas repetições. Por exemplo, no caso de mexer um bolo, o término ocorre quando a consistência é atingida. No caso da caminhada, o término das repetições ocorre quando chegamos ao lugar pretendido.

Talvez a maior habilidade dos computadores é poder realizar (sem reclamar...) a mesma coisa muitas vezes. Todas as linguagens de programação possuem comandos como WHILE, REPEAT ou FOR, que você já deve ter conhecido, e que comandam a realização dessas repetições. Vamos mostrar como essa situação pode ser tratada pela linguagem de montagem do Sapiens nas seções seguintes.

4.4.1 Repetições Simples

Em linguagem de máquina há uma forma peculiar de se tratar essas repetições: são os chamados comandos de desvio. Suponhamos que queremos repetidamente e indefinidamente repetir a sequência de instruções. No Sapiens utilizamos o comando JMP (abreviatura de *jump*, que significa saltar em inglês).

```
ORG 100
    LDA  X
```

```
    ADD  #1
    STA  X
; ...  faz mais coisas aqui
    JMP  100

X:  DB   0
```

Exemplo 4.9: Repetições Simples

Neste pequeno programa, armazenado na posição 100 de memória, o acumulador recebe o valor um, que é somado com o valor da variável X, e nela armazenado. Depois que isso é feito, o programa desvia para repetir o bloco iniciado pelas três instruções novamente, e novamente, infinitamente, sem parar.

Neste caso específico, o salto é para um endereço conhecido de memória (100). Na prática, o ponto para onde o processador irá desviar é um endereço qualquer, e neste caso, para não ter que definir o endereço real de desvio (o que pode ser muito chato, se o programa for grande e precisar ser alterado com frequência), podemos criar um rótulo (*label*) que associe aquela posição de memória a um dado nome. Por exemplo:

```
; ... algumas coisas antes
REPETE:
    LDA  X
    ADD  #1
    STA  X
; ...  faz mais coisas aqui
    JMP  REPETE
X:  DB   0
```

Exemplo 4.10: Repetições Simples - Outro Exemplo

Chamaremos essa posição de memória pelo nome de REPETE, e o

montador calcula que endereço é esse, facilitando a atividade de programação. Você pode pensar que esse pequeno programa, que simplesmente incrementa infinitamente o valor de uma variável, e depois faz algumas outras coisas, sem nunca terminar, não faz muito sentido. Realmente, na vida tudo termina, e isso aí, nunca termina. Surge portanto a necessidade de outros tipos de desvio, que são chamados de condicionais. Estes desvios só produzem o salto quando uma condição é satisfeita, portanto (a menos que esta condição nunca seja satisfeita) o programa vai terminar alguma hora.

Vamos agora produzir um outro exemplo, em que a repetição ocorre apenas quando o valor calculado não for 10. É simples: subtrairemos o valor de X de 10, e o desvio ocorrerá se o indicador de zero não estiver ativado.

```
ORG 0
REPETE:
    LDA  X
    ADD  #1
    STA  X
; ...  faz mais coisas aqui
    LDA  X
    SUB  #10
; desvia se positivo
    JNZ  REPETE

ORG 100
X:  DB   0
```

Exemplo 4.11: Desvio Condicional

Nota: o rótulo não precisa ser colocado na linha da instrução: pode ser escrito na linha anterior.

4.4.2 Repetições com contador

Uma situação extremamente comum é uma repetição contada: por exemplo, fazer algo 10 vezes. É usual, nestas situações, que uma variável seja utilizada para controlar esta contagem.

Nota: Algumas linguagens de programação tem comandos que fazem exatamente isso:

FOR CONTA = 1 TO 10

(faz alguma coisa)

Aqui o arranjo da programação é bem simples:

- colocamos o valor inicial na variável destino;
- introduzimos um rótulo para indicar que o trecho a seguir é o que repetiremos;
- depois vem o trecho a repetir;
- incrementamos a variável;
- testamos se a variável atingiu o valor final, desviando em caso negativo.

Veja a implementação:

```
ORG 0
; Inicia o contador com 1
    LDA  #1
    STA  CONTA
REPETE:
  ...  trecho que será repetido
; Incrementa o contador
    LDA  CONTA
    ADD  #1
```

```
    STA  CONTA
; Testa se passou do valor final
    SUB  #11
    JZ   REPETE
    HLT
ORG 100
CONTA:  DB  0
```

Exemplo 4.12: Repetições com Contador

É possível também começar o contador com o valor 10, e decrementá-lo, testando se chegou a zero. Isso fica como exercício para você.

4.5 Entrada e Saída de Dados

O SimuS tem diversos dispositivos de entrada e saída disponíveis para iteração com o usuário, tais como um painel de oito chaves binarias, um visor hexadecimal com dois dígitos, um visor (*banner*) com uma linha com 16 caracteres e um teclado numérico com 12 teclas. Esses dispositivos mais simples podem ser acessados através de operações simples de entrada e saída (**IN e OUT**), como descrito a seguir:

- **IN 0** : retorna no acumulador o valor binário do painel de 8 chaves;
- **IN 1**: retorna no acumulador o valor 01H quando houver um novo dado disponível no painel de chaves;
- **IN 2**: retorna do acumulador o valor ASCII correspondente a última tecla pressionada no teclado de 12 teclas;
- **IN 3**: retorna no acumulador o valor 01H quando uma nova tecla for pressionada no teclado de 12 teclas;
- **OUT 0**: o valor do acumulador é exibido no visor hexadecimal;
- **OUT 2**: o valor do acumulador, que deve estar na codificação ASCII, é exibido no visor de 16 caracteres.
- **OUT 3**: o visor de 16 caracteres é limpo.

Se desejarmos exibir no visor hexadecimal, por exemplo, o resultado final do Exemplo 4.2, basta executar a instrução **OUT 0**, que exibe o conteúdo do

acumulador no visor hexadecimal, como visto no Exemplo 4.13. Observe que o valor sempre será exibido, como o nome diz, no formato hexadecimal.

```
ORG 0
    LDA  #10    ; Acumulador = 10
    STA  A      ; A = Acumulador
    OUT  0      ; Mostra no visor hexadecimal
                ; o valor do acumulador
    HLT         ; Termina a execução

A:  DS  1       ; Variável A sem valor
                ; inicial
END 0
```

Exemplo 4.13: Escrevendo no Visor Hexadecimal

Fazer a entrada de dados é um pouco mais complicado, pois precisamos saber primeiro se algum dado novo foi colocado no painel de chaves, senão corremos o risco de ler um valor antigo, pois o processador, mesmo no simulador, é muito mais rápido na leitura das chaves do que o tempo que o ser humano leva alterar o seu conteúdo.

Para verificar se um valor novo foi colocado nas chaves, ficamos em *loop de status*, executando repetidamente a instrução **IN 1** até que o valor lido passe de ’0’ para ’1’. Depois disso, podemos fazer a leitura dos dados com a instrução **IN 0**. Um detalhe importante é a execução da instrução **AND #1** para alterar o valor da *flag* **Z**, já que a instrução **IN** não faz isso. Note que a execução da instrução **IN 0**, lendo o valor das chaves, faz com que o conteúdo do registrador de *status* no endereço ’01’ passe automaticamente para o valor ’0’.

O código do Exemplo 4.14 realiza permanentemente a leitura do conteúdo do painel de chaves e a exibição do valor lido no visor hexadecimal. Note que, no simulador, cada vez que o valor das chaves for alterado, é necessário

pressionar o botão ”Entrar” para validar a entrada de dados e alterar tanto o valor a ser lido no endereço ’00’ como o *status* no endereço ’01’, que passa automaticamente para ’1’.

```
ORG 0
; Fica em loop de status verificando
; se tem valor novo nas chaves
LACO:
    IN   1
    AND  #1
    JZ   LACO
; Faz a leitura das chaves
    IN   0
; Coloca o valor no visor hexadecimal
    OUT  0
    JMP  LACO
END 0
```

Exemplo 4.14: Lendo o Painel de Chaves

O exemplo 4.15 implementa um contador regressivo que faz a leitura do valor inicial do painel de chaves e exibe o resultado no visor hexadecimal do simulador decrementando de um em um até que chegue a zero, quando o programa termina.

```
ORG  0
; Fica em loop de status verificando
; se tem valor novo nas chaves
INICIO:
    IN   1
    SUB  #1
```

```
    JN   INICIO
; Faz a leitura das chaves
    IN   0
LACO:
; Coloca o valor no visor hexadecimal
    OUT  0
; Subtrai um do valor
    SUB  #1
; Armazena em CONT
    STA  CONT
    LDA  CONT
; Se já chegou a zero termina
    JN   FIM
    JMP  LACO
; Termina o programa
FIM:
    HLT
; Variáveis
CONT: DS 1    ;  CONTADOR
END 0
```

Exemplo 4.15: Contador Regressivo

Capítulo 5

Exemplos Avancados de Programação

Neste capítulo apresentamos exemplos mais sofisticados de programação, começando com o acesso a um vetor em memória, continuando com chamada de subrotinas e passagem de parâmetros na pilha, até exemplos com uso de E/S com a instrução **TRAP**.

5.1 Acessando um Vetor

5.1.1 Indexando os Elementos do Vetor

Um dos aspectos importantes da programação é como acessar os elementos de um vetor. Isso é realizado mais facilmente com uso do endereçamento indireto, ou seja, uma variável que contém o endereço de memória do elemento que pode ser incrementada ou decrementada de acordo. Na linguagem de montagem do Sapiens isso é expresso colocando-se um ”@” antes do operando da instrução.

Normalmente estamos de posse apenas do endereço de memória da primeira posição do vetor e precisamos realizar cálculos para determinar a exata localização do elemento desejado na memória. Normalmente a conta a ser realizada é a seguinte:

elemento[i] = elemento[0] + (i * tam. do elemento)

Na maior parte de nossos exemplos os vetores possuem elementos com um tamanho igual a um *byte* apenas, e a conta fica reduzida a:

elemento[i] = elemento[0] + i

Para declarar um vetor em linguagem de montagem, basta definir um rótulo apontando para a primeira posição do vetor, seguido da declaração de

todos os elementos. Veja a seguir:

```
; Calcula o endereço de X[I]
    LDA  #X      ; LDI X
    ADD  I
    STA  PT
; Faz acumulador = X[I]
    LDA  @PT
; .. Faz outras coisas
; Variáveis
PT: DS  1
I:  DB  0
X:  DB  10, 30, 25, 45
```

Exemplo 5.1: Declarando um Vetor

Mas no caso de, por exemplo, de um vetor com tipos inteiros na linguagem ”C”, o tamanho de cada elemento costuma ser de 4 *bytes*, e a conta seria;

elemento[i] = elemento[0] + i*4

5.1.2 Pares e Impares em um Vetor

No Exemplo 5.2, percorremos um vetor com seis elementos com um *byte* de tamanho, verificando quantos elementos são pares e totalizando em uma variável de nome **PARES**. Ao final o resultado é mostrado no visor hexadecimal do simulador.

```
ORG 0
; Repete até que I=6
    LDA I
```

```
LACO:
    SUB  #6
    JZ   FIM
; Calcula o endereço de X[I]
    LDA  PT       ; ACC = low(PT)
    ADD  I        ; ACC = ACC +1
    STA  PT       ; low(PT) = ACC
; Testa se é par (bit 0=0)
    LDA  @PT
    AND  #1
    JNZ  SENAO
; Se for, faz pares++
    LDA  PARES
    ADD  #1
    STA  PARES
SENAO:
; Incrementa I
    LDA  I
    ADD  #1
    STA  I
; Volta ao inicio
    JMP  LACO
FIM:
; Mostra no visor hexadecimal o total de pares
    LDA  PARES
    OUT  0
; Termina o programa
    HLT
; Variáveis
```

```
ORG 100
I:      DB  0
PARES:  DB  0
PT:     DW  X
X:      DB  6, 13, 8, 10, 9, 23
    END  0
```

Exemplo 5.2: Acessando um Vetor

Algumas observações são importantes: este programa funciona apenas com um tamanho máximo de vetor com 256 elementos. Como você o modificaria para funcionar com um vetor com maior número de elementos? E quando o vetor possuir elementos de 16 bits? Isso fica como exercício para você implementar.

5.1.3 Maior Elemento de um Vetor

Escreva um trecho de programa que determine qual o maior valor de um vetor de elementos com o tamanho de um *byte*, sem sinal. A variavel TAM, com tamanho de um *byte*, contém o número de elementos presentes na vetor. Ao final do programa, a variável MAX deve conter o maior valor e a variável MAXIND deve conter o índice do elemento com maior valor.

O exemplo a seguir foi feito com um vetor de 10 posições referenciado pela variável VETOR, que indica o seu endereço inicial na memória. Um variável auxiliar IND é utilizada para controlar o índice do elemento do vetor, assim como a variável PONTEIRO contém o endereço de memória do elemento do vetor que está sendo acessado.

```
;---------------------------------------
; Programa: DETERMINAR O VALOR MÁXIMO EM
; UM VETOR
; Autor: Gabriel P. Silva
; Data: 7.8.2017
```

```
;----------------------------------------
ORG 0
INICIO:
; Acumulador recebe vetor (0)
    LDA  @PONTEIRO
; MAX = VETOR(0)
    STA  MAX
LACO:
; Carrega o indice no acumulador
    LDA  IND
; Incrementa de um
    ADD  #1
; Salva na memória
    STA  IND
; Diminui do tamanho do vetor
    SUB  TAM
; Quando IND e TAM forem iguais termina
    JZ   FIM
; Carrega a parte baixa do ponteiro
    LDA  PONTEIRO
; Incrementa de um
    ADD  #1
; Salva na memória
    STA  PONTEIRO
; Se não de carry prossegue
    JNC  SEGUE
; Se deu carry carrega 1 no acumulador
    LDA  #1
; Soma com a parte alta do ponteiro
```

```
    ADD  PONTEIRO+1
; Salva na memória
    STA  PONTEIRO+1
SEGUE:
; Carrega VETOR(IND) no acumulador
    LDA  @PONTEIRO
; Diminui do valor máximo
    SUB  MAX
; Se for negativo continua
    JN   LACO
; Se deu "vem-um" continua
    JC   LACO
; Senão troca MAX por VETOR(IND)
    LDA  @PONTEIRO
; MAX = VETOR(I)
    STA  MAX
; Carrega IND no acumulador
    LDA  IND
; MAXIND = IND
    STA  MAXIND
; Continua
    JMP  LACO
FIM:
    LDA  MAX
; Mostra o valor máximo no visor hexadecimal
    OUT  0
    HLT
; Variáveis
    ORG 100
```

```
; Número de elementos do vetor
TAM:      DB    10
; Indice do vetor
IND:      DB    0
; Indice do maior elemento
MAXIND:   DB    0
; Valor do maior elemento
MAX:      DW    0
; Endereço do elemento atual
PONTEIRO: DW    VETOR
; Vetor
VETOR:    DB    11, 27, 31, 82, 23, 80, 180, 14, 47, 6
```

Exemplo 5.3: Determinar o maior valor de um vetor

5.2 Uso de Subrotinas

Quando da elaboração de um programa, com frequência necessitamos fazer uso de subrotinas (procedimentos ou rotinas são outros nomes usualmente empregados). Subrotinas são trechos de código que podem ser chamados de diversos pontos do programa principal. Ao contrário de um desvio convencional, que apenas segue em frente e se esquece do passado, ao chamarmos uma subrotina é necessário salvar do endereço de memória de onde ela foi chamada. Isso porque, ao final da subrotina, devemos retornar a execução do processador para o mesmo ponto do programa de onde ela foi chamada. Na Fgura 5.1 procuramos ilustrar esta situação.

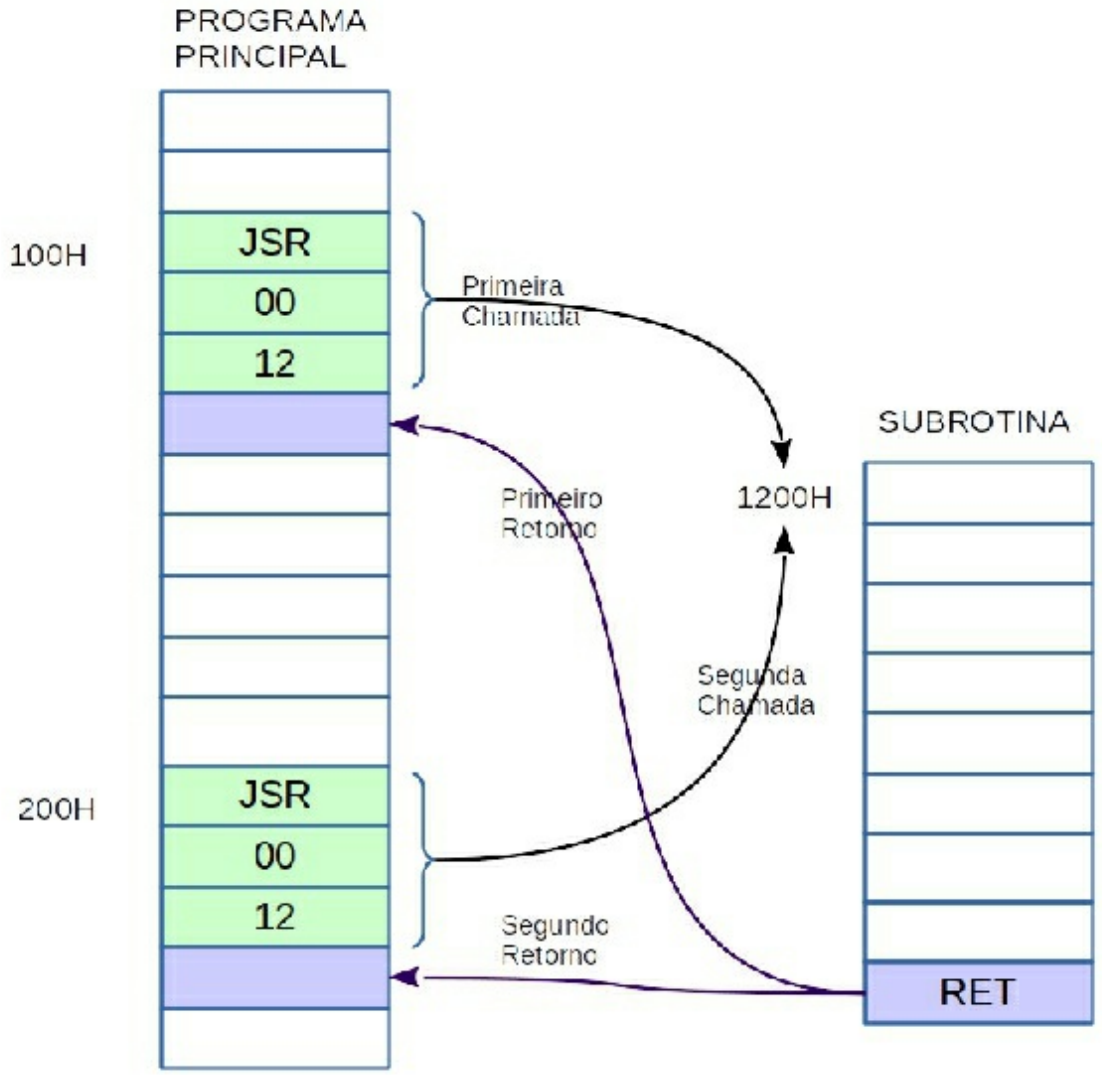

Figura 5.1: Chamada e Retorno de Subrotina

A chamada e o retorno de uma subrotina envolve uma série de passos que vamos detalhar a seguir:

1. Ao executarmos a instrução de chamada de subrotina (JSR),temos que salvar primeiro o endereço de retorno, que é o endereço da instrução logo após a instrução JSR.
2. Alguns processadores possuem registradores específicos para guardar este endereço, porém o mais comum é que este endereço seja guardado na memória, em uma estrutura de dados chamada de pilha.

3. A pilha, além de guardar o endereço de retorno, pode ser utilizada para a passagem e o recebimento de parâmetros para a subrotina.
4. Depois que o endereço de retorno foi salvo na pilha, o valor do apontador de instruções (PC) é alterado para o início da subrotina, onde a processador continua a execução das instruções.
5. Se houver parâmetros, os mesmos são retirados da pilha e utilizados pela subrotina. Da mesma forma que, ao final da subrotina, se houver resultados, eles podem ser passados também de volta na pilha.
6. Ao final da subrotina a ultima instrução que é executada (RET) faz a retirada do endereço de retorno da pilha, copiando o mesmo para o apontador de instruções (PC).
7. A execução do programa prossegue então a partir da instrução seguinte àquela que chamou a subrotina.

A seguir mostramos alguns exemplos práticos de chamada e retorno de procedimentos, com passagem de parâmetros.

```
ORG 100
    LDA  #END_BASE
    PUSH
    LDA  #STR_INICIAL
    PUSH
    JSR  ROTINA
; Le as chaves. Se for impar conta impares e vice-versa.
INPUT:
    IN   1
    AND  #1
    JZ   INPUT
    IN   0
    STA  MODO
; Faz a leitura de um elemento do vetor e incrementa o contador
INICIO:
    LDA  CONT
```

```
    ADD  #1
    STA  CONT
    LDA  @END_VETOR
; Testa se é par (bit 0=0)
    AND  #1
    JNZ  EHIMPAR
; Se for par, faz pares++
EHPAR:
    LDA  #1
    ADD  PARES
    STA  PARES
    JMP  TESTE
; Se for impar, faz impares++
EHIMPAR:
    LDA  #1
    ADD  IMPARES
    STA  IMPARES
; Incrementa o apontador e verifica se o loop acabou
TESTE:
    LDA  END_VETOR
    ADD  #1
    STA  END_VETOR
    LDA  TAM_VETOR
    SUB  CONT
    JP   INICIO
; Coloca o endereço da string na pilha
; Chama a rotina de impressão
IMPRIME:
    LDA  MODO
```

```
    AND  #1
    JNZ  SO_IMPAR
SO_PAR:
    LDA  #END_BASE
    PUSH
    LDA  #STR_PARES
    PUSH
    JSR  ROTINA
; Macete para imprimir o número em decimal (0-9)
    LDA  PARES
    ADD  #30H
    OUT  BANNER
    JMP  FIM
; Coloca o endereço da string na pilha
; Chama a rotina de impressão
SO_IMPAR:
    LDA  #END_BASE
    PUSH
    LDA  #STR_IMPARES
    PUSH
    JSR  ROTINA
; Macete para imprimir o número em decimal (0-9)
    LDA  IMPARES
    ADD  #30H
    OUT  BANNER
FIM:
    HLT
END 100
; Declaração das variáveis do programa principal
```

```
ORG 2000
TAM_VETOR:   DB  10
END_VETOR:   DW  VETOR
PARES:       DB  0
IMPARES:     DB  0
VETOR:       DB  1,2,3,4,5,6,7,8,9,10
CONT:        DB  0
MODO:        DB  0
STR_PARES:   STR "Pares:"
             DB  0
STR_IMPARES: STR "Impares:"
             DB  0
STR_INICIAL: STR "Entre par ou impar:"
             DB  0
END_BASE     EQU 0
;-----------------------------------------
; Rotina para impressão de uma string no banner
; Declaração das variáveis da rotina

ORG  1000
SP:     DW  0    ; Guarda o valor do stack pointer
PTR:    DW  0    ; Ponteiro com o endereço da string a ser impre
; Constantes de hardware
CLEARB  EQU 3
BANNER  EQU 2
;-----------------------------------------
ROTINA:
; Salva o valor atual do SP
    STS     SP
```

```
; Descarta as primeiras duas posições da pilha
    POP
    POP
; Tira a parte baixa do endereço da string da pilha
    POP
; Salva na parte baixa do ponteiro
    STA  PTR
; Tira a parte alta do endereço da string da pilha
    POP
; Salva na parte alta do ponteiro
    STA  PTR+1
    OUT  CLEARB  ; Limpa o Banner
LOOP:
    LDA  @PTR    ; Le o caractere
    OR   #0      ; É NULL?
    JZ   RETORNA ; Se for retorna
    OUT  BANNER  ; Imprime o caractere no banner
    LDA  PTR     ; Incrementa o ponteiro
    ADD  #1
    STA  PTR
    JMP  LOOP    ; Volta para o inicio
RETORNA:
    LDS  SP      ; Restaura o Stack Pointer
    RET          ; Retorna
```

Exemplo 5.4: Determinar o total de pares e ímpares em um vetor

No exemplo acima é possível observar o uso da pilha para a passagem de parâmetros para a subrotina. A pilha é uma estrutura de dados do tipo FILO ("Fist In Last Out"), ou seja, o primeiro elemento a ser colocado é o último a ser retirado. As operações possíveis são PUSH (colocar um elemento na

pilha) e POP (retirar um elemento da pilha).

O acesso à pilha é controlado por um registrador de nome ponteiro de pilha (SP - do inglês *stack pointer*). Esse registrador contém o endereço do último elemento colocado na pilha. Assim, o seu valor deve ser modificado toda vez que inserimos ou removemos um dado da pilha. O valor do SP será modificado de acordo com o tamanho do dado inserido ou removido: um byte, dois bytes e assim por diante.

Por razões históricas, a pilha no computador tem uma característica especial: ela cresce do final da memória para o início, no sentido inverso das outras estruturas de dados que o programa utiliza. Assim, quando um elemento é inserido na pilha, o SP é decrementado. Quando um elemento é retirado, o SP é incrementado. Em ambos os casos, do mesmo tamanho do dado inserido ou retirado.

Quando chamamos uma subrotina, o endereço de retorno é colocado na pilha, e o apontador de pilha é decrementado de dois bytes. Quando retornamos, este endereço é retirado da pilha, e o apontador de pilha é incrementado de duas posições. Deste modo é possível, enquanto houver espaço disponível para a pilha crescer, ir chamando subrotinas dentro de subrotinas e retornando sempre para o ponto correto no programa principal.

Enfim, as possibilidades são infinitas, mostramos aqui um pequeno número delas. Na próxima seção apresentamos algumas propostas para exercícios de programação.

5.3 Exercícios Propostos

A seguir propomos alguns exercícios complementares para serem realizados no simulador SimuS. Procuramos organizá-los em ordem crescente de dificuldade, na medida do possível.

1. Codifique um programa que leia um valor N do painel de chaves, faça a soma dos números entre 1 e N e apresente o resultado no visor hexadecimal.
2. Implemente um programa que leia dois valores em sequência do painel de chaves e de acordo com um terceiro valor faça:

- 0 - Mostre o maior valor no visor hexadecimal.
- 1 - Mostre o menor valor no visor hexadecimal
- 2 - Mostre a soma dos valores no visor hexadecimal.
- 3 - Mostre a diferença dos valores no visor hexadecimal.

3. Escreva uma rotina que receba o endereço de uma palavra de 32 bits na pilha e no acumulador um parâmetro adicional e faça o seguinte:

   - 0 - conta o número de ’0’s na palavra.
   - 1 - conta o número de ’1’s na palavra.

   O resultado é devolvido no acumulador. Apresente um programa de exemplo que mostre o resultado no visor hexadecimal.

4. Escrever um programa que leia um caractere, e apenas um, do teclado virtual, e calcule o valor correspondente da série de Fibonacci usando um procedimento recursivo. Escrever o resultado no visor de 16 caracteres. Note que o valor lido do teclado está codificado em ASCII.
5. Implemente uma rotina que receba um valor no acumulador e faça a impressão do seu valor em decimal no visor de 16 caracteres e na console. Elabore um programa de teste que leia o valor do painel de chaves. Uma função de conversão possível é seguinte: basta somar 6 se o número estiver entre 10H e 1FH, 12 se estiver entre 20H e 2FH, 18 se estive entre 30H e 3FH, e assim por diante. O algoritmo está limitado a um valor máximo igual a 99 em decimal. Você teria uma outra sugestão de algoritmo?
6. Escrever um procedimento com funcionalidade similar ao procedimento *memset*, da bilbioteca padrão de C. Ou seja, o procedimento deve preencher uma região de memória com um valor de byte dado como parâmetro. Os parâmetros são passados na pilha, ou seja, o endereço inicial, o número de bytes e o valor a ser preenchido. Apresente um programa completo de exemplo.
7. Escreva uma rotina para copiar uma cadeia de caracteres de uma posição da memória para outra. Os endereços de origem e destino são passados como parâmetros na pilha e a cadeia de caracteres é terminada com NULL (00H). Apresente um programa completo de exemplo.
8. Implemente uma rotina que receba o endereço de dois números de 8 bits na pilha e um parâmetro adicional passado no acumulador e de

acordo com esse valor passado faça:

- 0 - Multiplique os dois valores
- 1 - Faça a divisão inteira dos dois valores, determinado quociente e resto.

9. Variações em torno do tema de multiplicação/divisão incluem:

- Utilizar variáveis de 16 bits.
- Utilizar variáveis com sinal.

10. Implemente uma rotina que leia uma sequência de dígitos do teclado virtual terminadas por "#". Assumindo que o valor está em hexadecimal, calcule o seu valor em binário e armazene em uma variável de 16 bits cujo endereço foi passado como parâmetro na pilha.
11. Implemente o mesmo que foi solicitado no exercício anterior considerando que o valor é decimal.
12. Implemente um programa que leia dois valores decimais do teclado virtual, terminados por "#' e de acordo com um terceiro valor também lido do teclado faça:
    (a) - Mostre o maior valor no visor de 16 caracteres.
    (b) - Mostre o menor valor no visor de 16 caracteres.
    (c) - Mostre a soma dos valores no visor de 16 caracteres.
    (d) - Mostre a diferença dos valores no visor de 16 caracteres.
13. Escrever um programa, que implemente uma calculadora simples, que leia valores de até 4 dígitos do teclado de 12 posições e realize operações de soma e subtração. A tecla implementa a soma e a tecla # implementa a subtração. O resultado deverá ser exibido, junto com os números lidos, no visor de 16 caracteres do simulador.
14. Implemente uma rotina que receba o endereço de um vetor com elementos de 8 bits na pilha e um parâmetro adicional passado no acumulador e de acordo com esse valor faça:

- 0 - Calcula a soma de todos os elementos pares do vetor
- 1 - Calcula a soma de todos os elementos ímpares do vetor

O resultado é devolvido no acumulador. Elabore um programa de exemplo que leia o valor o segundo parâmetro do teclado virtual de 12

teclas e imprima o resultado no visor de 16 caracteres.

15. Variações em torno do tema de manipulação de vetores incluem:

- Contar o número elementos pares e ímpares do vetor
- Contar o número de elementos positivos, negativos e nulos de um vetor.
- Encontrar o maior ou menor elemento e sua posição no vetor.
- Utilizar elementos com 16 bits, com ou sem sinal.

16. Implemente uma rotina que receba o endereço de um vetor com elementos de 8 bits na pilha e um parâmetro adicional passado no acumulador e de acordo com esse valor passado faça:

- 0 - Ordene o vetor em ordem crescente
- 1 - Ordene o vetor em ordem decrescente

Elabore um programa de exemplo que leia o vetor da console, um valor em cada linha e imprima o resultado também na console.

17. Variações em torno do tema de ordenação de vetores incluem:

- Utilizar um algoritmo de ordenação recursivo (p. ex. "quick sort")
- Utilizar um algoritmo de contagem com "radix sort".
- Utilizar elementos com 16 bits, com ou sem sinal.

18. Escreva uma rotina para verificar se uma cadeia de caracteres é palíndrome, ou seja, se dá o mesmo resultado quando lida da direita para a esquerda ou da esquerda para a direita, ignorando brancos e pontuações (“a cara rajada da jararaca” é um exemplo). O endereço do início da cadeia é passado como parâmetro na pilha e a cadeia é terminada com NULL (00H). Se a cadeia for uma palíndrome, o valor ’0’ deve ser retornado no acumulador; caso contrário, deve retornar o valor ’1’. Apresente um programa exemplo que leia a cadeia da console e escreva o resultado também na console.
19. Escrever um programa que utiliza a rotina de TRAP e a função de temporizador, o visor de 16 caracteres e o teclado para implementar um contador de tempo regressivo rudimentar, para contar segundos. Suponha

que o valor inicial em segundos é entrado no teclado e cuja entrada é terminada com a tecla #.

20. Escrever um programa que mostre números em sequência entre 0 e 9 no visor hexadecimal começando com um segundo de intervalo entre eles, com uso das rotinas de **TRAP**. O botão de entrada das chaves deve ser apertado quando o valor 5 for exibido. Cada vez que o usuário acertar deve ser contabilizado um acerto e o tempo diminuído em cerca de 10%. Quando o usuário errar, apresentar o total de acertos no visor hexadecimal e tocar um bipe com uso da rotina de **TRAP**. Reiniciar o procedimento quando o valor 0 for entrado nas chaves.

Capítulo 6

O Simulador SimuS

6.1 Introdução

6.2 Componentes

Mantivemos a interface básica do simulador do Neanderwin, acrescentando mais algumas funcionalidades e agora destacando os módulos de E/S, que estão em janelas que são ativadas apenas quando for conveniente para o usuário. Desde o início do projeto do simulador SimuS nosso objetivo era facilitar ao máximo as atividades didáticas do professor e o apoio mais completo possível para as dificuldades comuns do aluno. Para isso foi criado um ambiente integrado para desenvolvimento, com versões para os sistemas operacionais Windows e Linux, e que inclui os seguintes módulos:

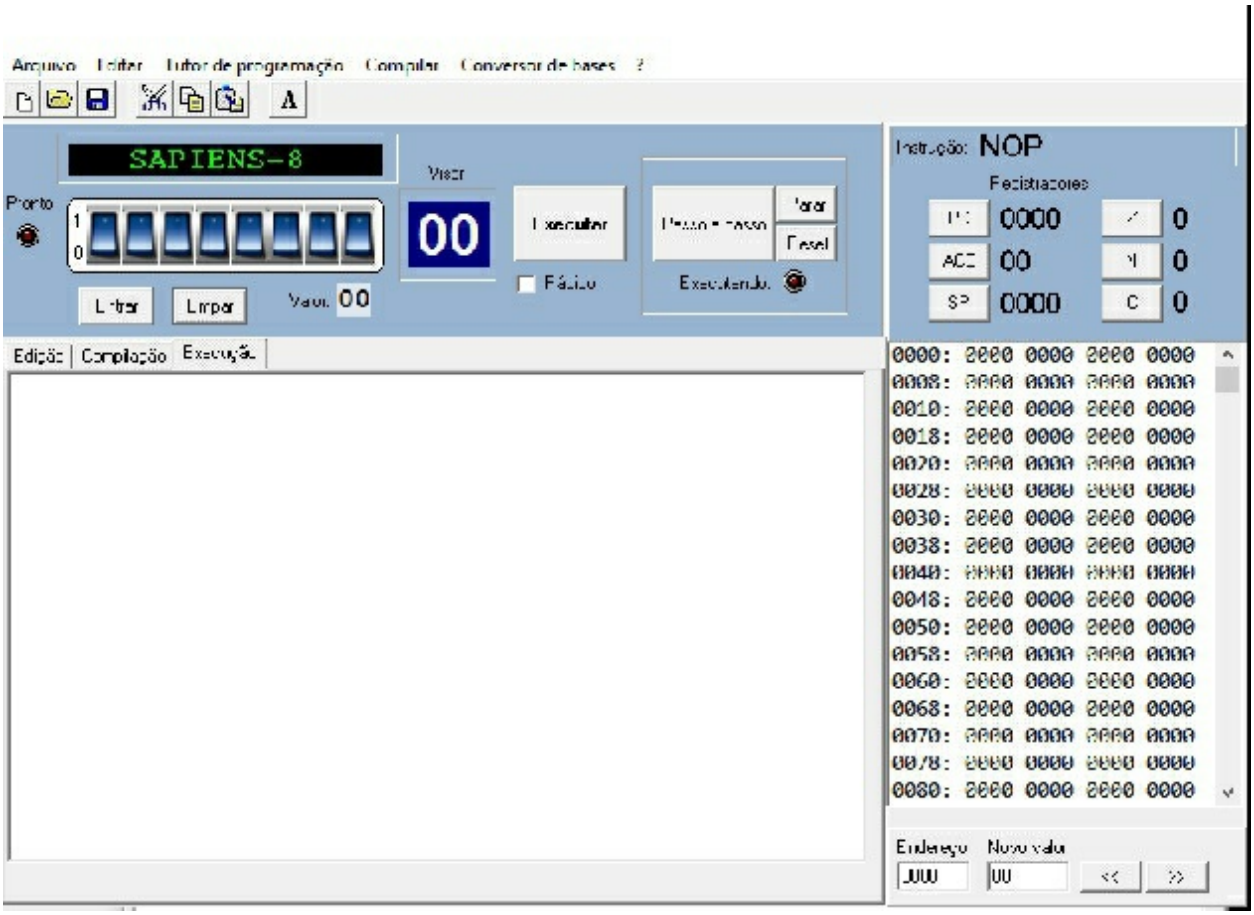

Figura 6.1: Simulador SimuS

- Editor de textos integrado, que possibilita a abertura, edição e salvamento de arquivo com o código em linguagem de montagem.
- Montador (assembler) também integrado, gerando o código objeto final em linguagem de montagem. Possui compatibilidade com programas escritos para o Neander ou Neander-X.
- Simulador da arquitetura, com visualização e modificação dos elementos arquiteturais do processador, tais como registrador de instrução, apontador de instrução, apontador de pilha, acumulador, flags N, Z e C; além da execução passo-a-passo ou direta.
- Módulo de depuração, definindo pontos de parada e variáveis em memória para serem monitoradas, em caso de mudança de valor a execução é interrompida.
- Visualização e modificação da memória simulada, no formato

hexadecimal os endereços e também para seu conteúdo, agora expandida para 64 Kbytes.

- Ferramenta de apoio ao aprendizado de instruções, com ajuda integrada ao editor de textos, para a geração de código em linguagem de montagem do processador Sapiens.
- Utilitário para conversão de bases binária, decimal e hexadecimal.
- Simulador de dispositivos de E/S, que inclui o tradicional painel com 8 chaves e visor hexadecimal, acrescidos de um teclado de 12 teclas e um painel visor (*banner*) de uma linha com 16 caracteres. Estes dispositivos são acessados no espaço de endereçamento de E/S convencional com instruções de **IN e OUT**.
- Dispositivos especiais como uma console virtual, acessada pelo mecanismo descrito anteriormente. Com esse mecanismo outros módulos mais complexos possam ser facilmente instalados, sem necessidade de expor esta complexidade para os usuários.
- Gerador/carregador de imagem da memória simulada, podendo ser salva em formato hexadecimal. Note que neste caso a compatibilidade entre o SimuS e o Neanderwin não é mantida, ou seja, a imagem salva em um simulador não pode ser carregada em outro.

A Figura 6.1 mostra a aparência da tela principal do simulador SimuS. Na parte superior estão o menu geral de operação (Arquivo, Editar, etc.) e diversos botões usados em conjunto com o editor de textos, que seguem o estilo usual de programas com interface gráfica.

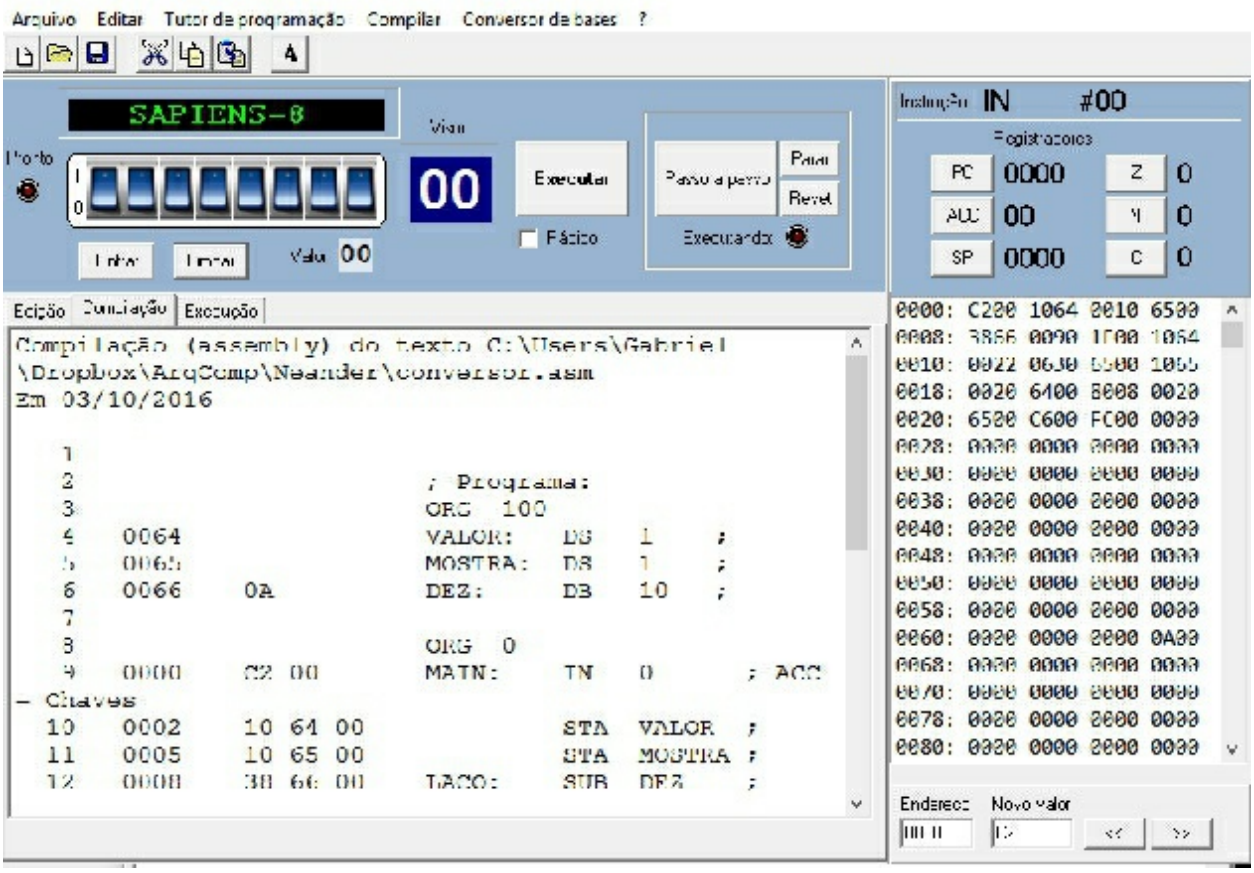

Figura 6.2: Janela de Compilação

Logo abaixo, está o painel do simulador, em que são mostrados os dispositivos virtuais de entrada e saída, e diversos botões para controle de execução.

Imediatamente abaixo, à esquerda, está o editor de textos, no qual o programa do aluno é digitado ou criado interativamente através de uma função para criação tutorada de programas, que será descrita com mais detalhes adiante.

À direita desta área podem ser visualizados a próxima instrução a ser executada, o conteúdo do apontador de instruções (PC), acumulador (AC) e das *flags* zero (Z), negativo (N) e vai-um (C). Logo abaixo o visualizador do conteúdo memória, com os endereços em hexadecimal, com duas janelas com endereço e dado, para alteração de conteúdo da memória.

Uma vez que o programa for compilado, o resultado aparecerá em uma aba com a listagem, como visto na Figura 6.2, cujo formato de saída é similar à maioria dos montadores profissionais, com indicação dos eventuais erros de compilação. O programa compilado é carregado imediatamente na memória, sendo o conteúdo atualizado e exibido no painel correspondente. Caso se deseje, é possível copiar o conteúdo desta janela para a área de transferência, para colar em algum editor de textos possibilitando eventuais embelezamentos e impressão a posteriori.

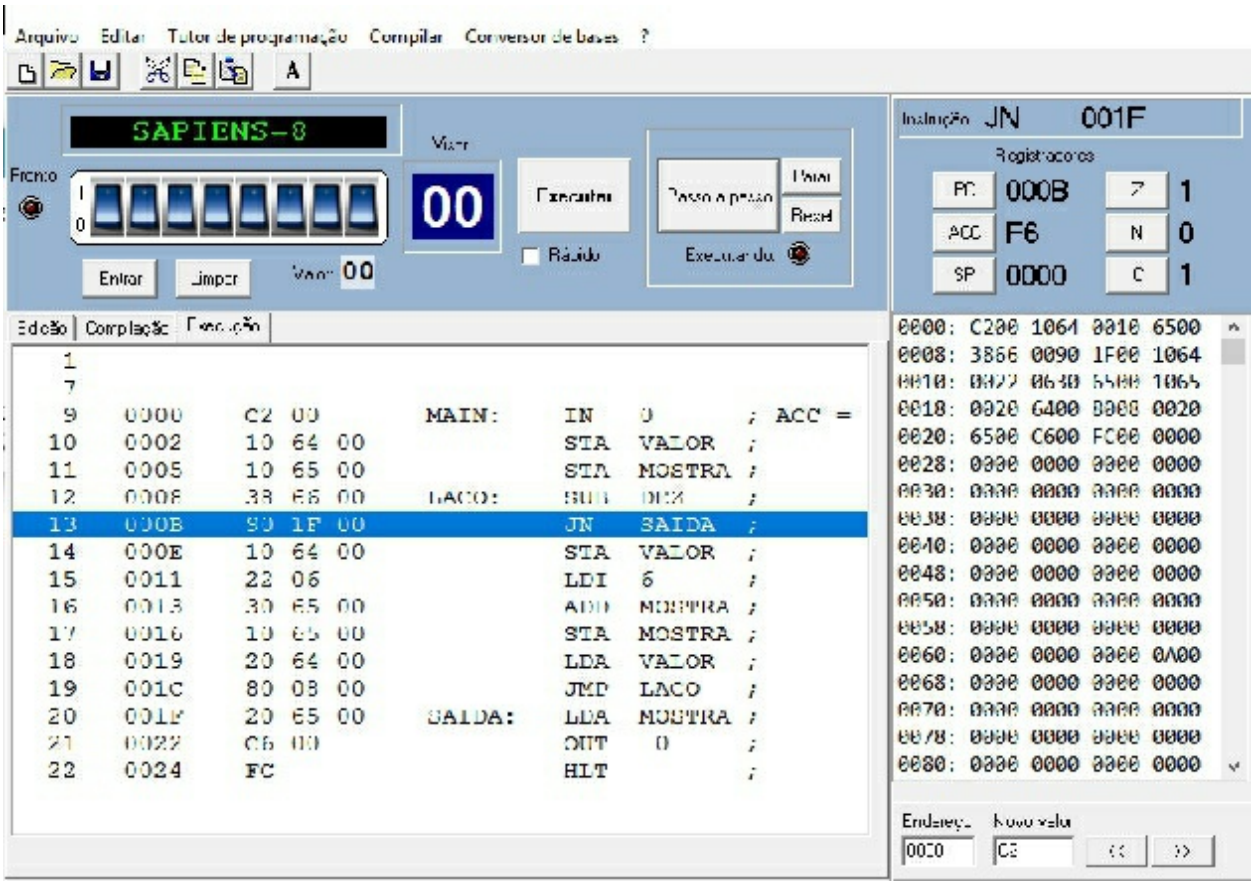

Figura 6.3: Janela de Execução

O número de instruções do processador Sapiens é pequeno, mas apesar disso notamos que é importante tornar disponível uma “ajuda online” interativa, que é ativada pelo menu do programa, um sistema de entrada interativa de instruções e pseudo-instruções. A função de criação tutorada de

programas, mostrado na Figura 6.4, faz com que o aluno cometa menos erros e a compreenda melhor o significado das instruções e produza um formato de instrução correto interativamente.

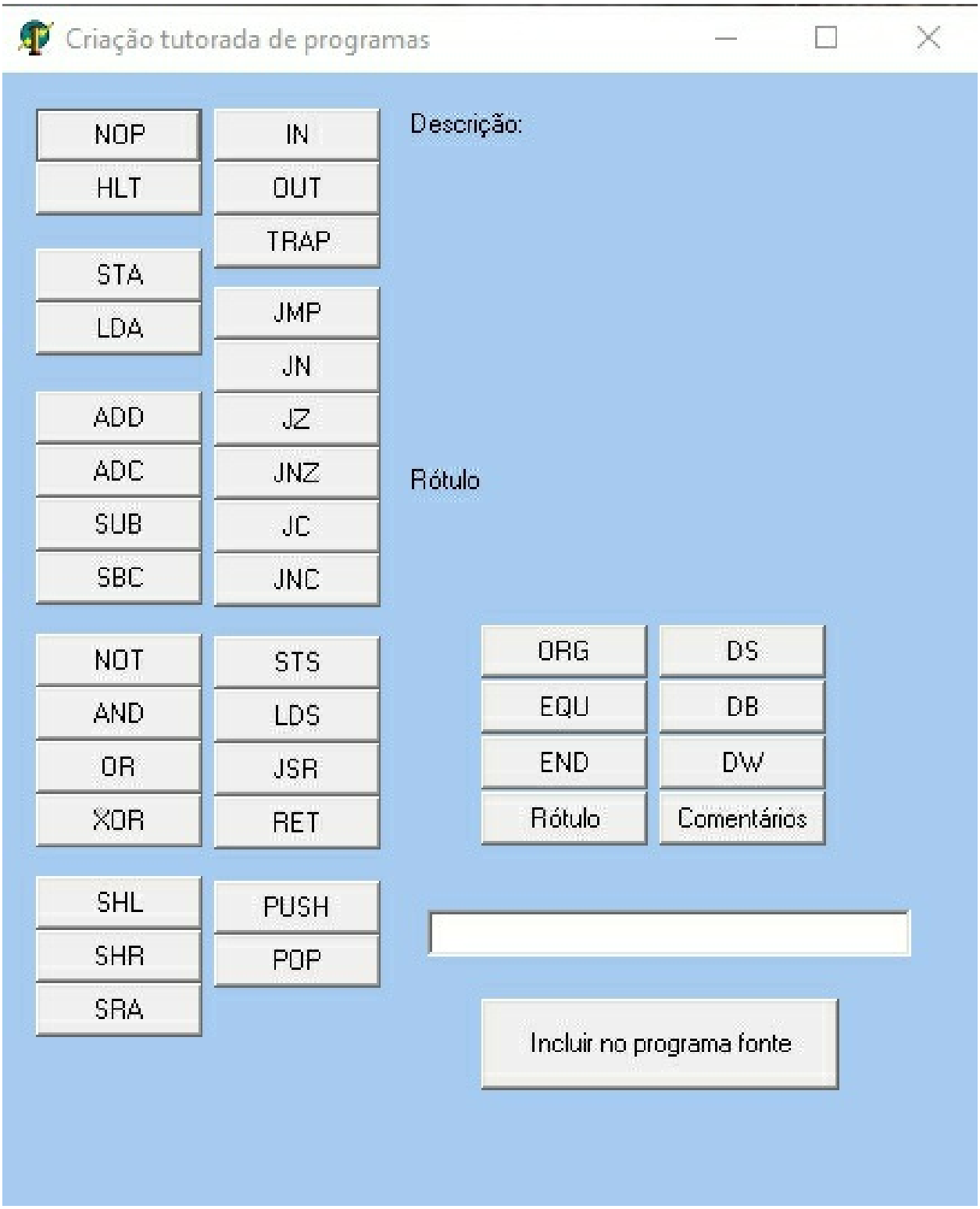

Figura 6.4: Edição Tutorada

Por último, notamos que para muitos estudantes o domínio das representações hexadecimal e binária, necessário para verificar e alterar a memória, só se dá depois de algum tempo. Mesmo sabendo que as calculadoras do Windows e do Linux exibem resultados em várias bases, incluímos um conversor de bases, que é muito mais simples.

6.3 Dicas de Uso do SimuS - **Colaboração Thales Magalhães**

6.3.1 Leia o manual

Antes de começar a escrever seu primeiro programa, descubra quais são e o que fazem as instruções da máquina, mesmo que superficialmente. Aprenda sobre as diretivas de compilador como **DB, DW, DS e STR** e como são usadas. A princípio esta dica pode parecer óbvia, mas acredite em mim, ela talvez seja dica mais importante dessa lista. O manual do Sapiens é razoavelmente curto, então comece com o pé direito ao invés de desperdiçar horas tentando solucionar problemas sem conhecer direito a ferramenta.

6.3.2 Comente seu código

Devido ao nível extremamente baixo da linguagem, programação em *assembly* é um processo minucioso, especialmente quando se trata de uma arquitetura limitada como o Sapiens. Para evitar a confusão, é importante manter uma ótima organização do seu código fonte. Nesse sentido, existem várias estratégias que podem ser abordadas. Um método simples, mas bastante efetivo de escrever um programa de forma organizada envolve separar o código em pequenos blocos logicamente distintos. Cada bloco realiza uma função específica e é acompanhado de um comentário descrevendo sua função. Idealmente, cada bloco deve executar sua função de forma autônoma, independentemente dos outros. Por exemplo:

```
LDA  I       ; ENQUANTO I != 4
SUB  #4
JZ  ENDWHILE
```

```
LDA  @ADDR   ; TEMP = *ADDR
STA  TEMP


LDA  #0      ; J = 0
STA  J
```

Assim, fica fácil seguir a semântica e o fluxo lógico do programa sem que seja necessário analisar cada instrução individualmente. Independente da estratégia que você decida adotar no seu próprio código, manter uma boa documentação do código fonte é indispensável.

6.3.3 Não se esqueça da diretiva END

A diretiva **END** é responsável por instruir ao compilador o endereço de início do seu programa. Seu uso é extremamente importante, mesmo que o compilador não efetivamente imponha sua presença no código. Até o momento, seus programas podem ter funcionado sem problemas aparentes. Por padrão, a execução se inicia a partir do endereço 0, porém este comportamento nem sempre é garantido. Ao não especificar o endereço inicial do programa, o simulador mantém o endereço inicial já estabelecido, seja este qual for. Isto é, caso algum programa anterior altere o endereço inicial para outro valor, seu código não será executado a partir do endereço inicial esperado. Por isso, lembre-se de sempre utilizar esta diretiva, garantindo que seu programa funcione independente do estado inicial do simulador.

Outra observação importante é pertinente ao uso da diretiva **ORG**. Esta diretiva define o endereço físico a partir do qual as instruções e dados que seguem são inseridos em memória. É importante perceber que o uso indevido desta diretiva pode fazer com que dados e instruções sejam sobrepostos na memória, causando comportamento inesperado. No Exemplo 6.1 a seguir, por exemplo, a execução não sai do laço e portanto nunca termina:

```
ORG 10
X: DB  0
```

```
ORG 0
WHILE:
    LDA  X       ; ENQUANTO X != 10
    SUB  #10
    JZ   ENDWHILE

    LDA  X       ; X++
    ADD  #1
    STA  X

    JMP  WHILE

ENDWHILE:
    HLT
END 0
```

Exemplo 6.1: Exemplo com Erro

Eu prefiro definir o início do meu programa com um rótulo qualquer, como por exemplo INICIO, START ou MAIN, desta forma as posições são definidas pelo compilador e não há risco de sobreposição:

```
X: DB 0
WHILE:
; Enquanto X != 10
    LDA  X
    SUB  #10
    JZ   ENDWHILE
; X++
    LDA  X
```

```
    ADD  #1
    STA  X
    JMP  WHILE
ENDWHILE:
    HLT
;
END WHILE
```

Exemplo 6.2: Exemplo usando rótulos

6.3.4 Use o depurador

Além de simular o funcionamento da arquitetura Sapiens, o ambiente SimuS também acompanha um depurador. Através dele, é possível alterar valores em memória durante a execução do programa, assim como interromper sua execução. Além disso, o programa também pode ser executado passo a passo, uma instrução por vez. Outra funcionalidade extremamente útil é a janela de variáveis, através da qual pode-se acompanhar ao longo da execução do programa os valores presentes nos rótulos definidos pelo programador. Juntas, estas ferramentas tornam bem mais prático o trabalho de depurar o funcionamento de programas e corrigir erros.

6.3.5 Janela de variáveis

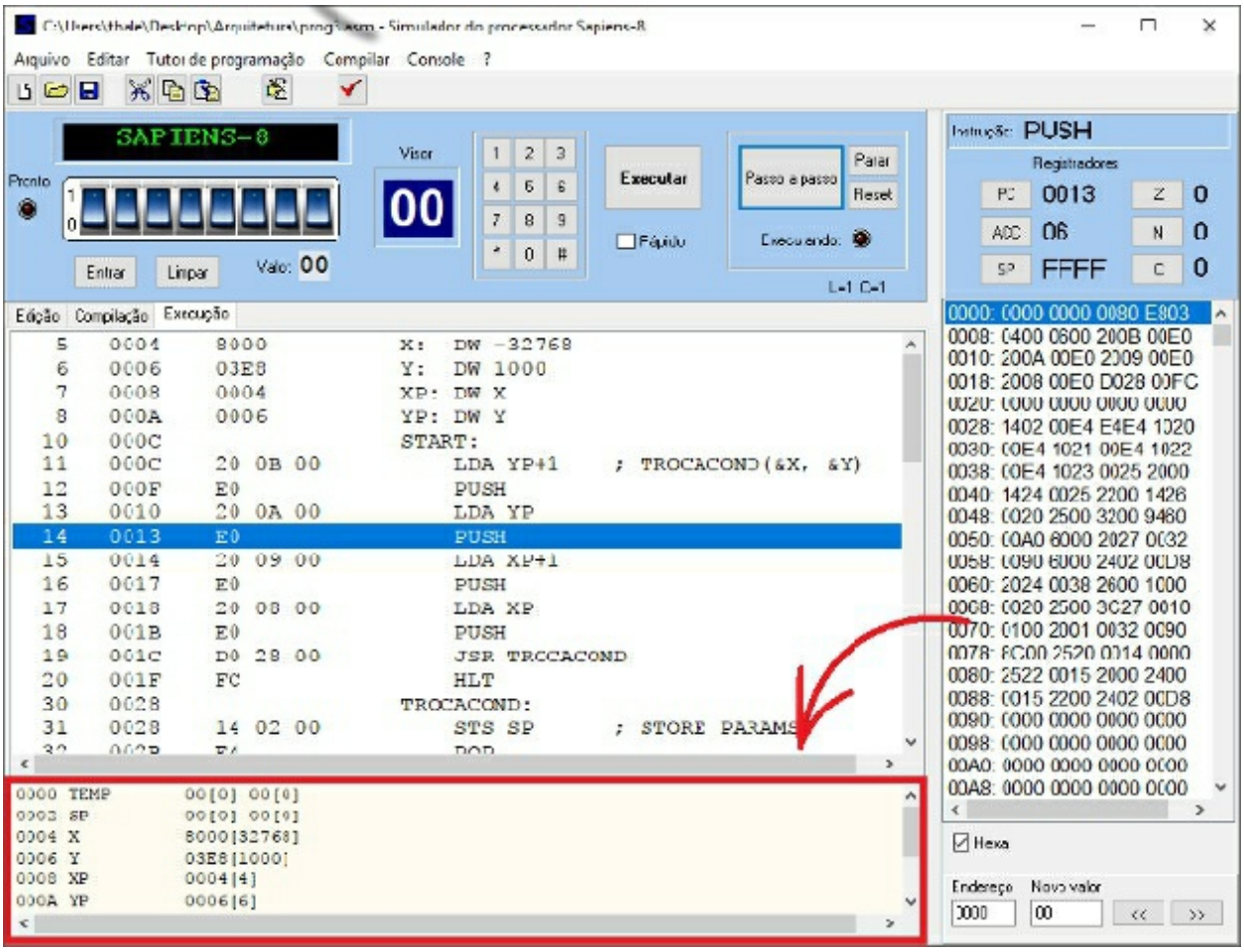

Figura 6.5: Janela de Variaveis

A janela de variáveis, destacada acima, apresenta os valores de todos os endereços declarados a partir das diretivas **DB, DW, DS e STR**. Quando a declaração acompanha um rótulo, como em **X: DB 1, 2, 3**, o rótulo é tambem exibido junto com seu valor.

6.3.6 Soma e subtração em 16 bits

Quando se trabalha com arquiteturas de 8 bits, é essencial saber como lidar com números em larguras maiores como 16, 32 ou até 64 bits. Isso se torna ainda mais importante no caso do Sapiens, que embora seja limitado a aritmética em 8 bits, também faz uso extensivo de dados em 16 bits para endereços de memória. Como a ALU da arquitetura é limitada a cálculos com

8 bits de largura, é necessário que haja uma separação dos cálculos com números grandes em múltiplas etapas de apenas um byte cada. No Sapiens, as instruções instruções ADC e SBC, que fazem uso da *flag* **C**, são a maneira mais prática de realizar essa separação. Através dessas instruções fica fácil encadear somas e subtrações com vários bytes, automaticamente levando em consideração os eventuais casos de "vai-um"e "vem-um"através da *flag* **C**. Desta forma, não é necessário o uso de desvios condicionais, o que torna o código mais conciso e fácil de entender. Exemplo:

```
LDA  X       ; Z = X + Y
ADD  Y
STA  Z
LDA  X+1
ADC  Y+1
STA  Z+1
```

6.3.7 Endereços de memória e a pilha

O uso correto da pilha é essencial para o funcionamento de uma sub-rotina. Por isso, é importante estabelecer um padrão que determine como os argumentos devem ser passados entre as subrotinas e quem faz a chamada. O padrão que eu uso para o meu código determina que os argumentos devem ser retirados da pilha na ordem normal pela subrotina. Você pode utilizar qualquer padrão em seu código, o mais importante é manter esse padrão de forma consistente. Veja o Exemplo 6.3 a seguir:

```
ARR:  DB 10, 11, 12
ARRP: DW NUM

START:
    LDA  ARRP+1
    PUSH
```

```
    LDA  ARRP
    PUSH
    JSR  FOO

[...]

PTR: DS 2
VAL: DS 1

; VOID FOO(CHAR *PTR)
FOO:
; Salva argumentos
    STS  SP
    POP
    POP
    POP
    STA  PTR
    POP
    STA  PTR+1
    LDA  @PTR    ; VAL = *PTR
    STA  VAL
```

Exemplo 6.3: Exemplo com Uso da Pilha

É importante observar que, pela própria natureza da pilha, os dados são inseridos e retirados de forma inversa. Não se esqueça disso, principalmente quando for transferir endereços de memória para suas rotinas! Também é bom lembrar que a instrução JSR sempre insere o endereço de retorno na pilha, portanto os primeiros dois bytes não fazem parte da lista de argumentos. Outra observação importante é quanto ao uso do atalho ENDEREÇO+OFFSET fornecido pelo compilador, que facilita bastante a transferência de dados grandes. Porém, fique atento ao fato de que este

cálculo é realizado em tempo de compilação. Por isso só é útil quando se trata de constantes (como um rótulo) e não serve para dados fornecidos em tempo de execução. Por exemplo, o seguinte trecho de código pode não fazer o que você espera:

```
LDA  @PTR+2
STA  VAL
```

6.3.8 Usos alternativos para o apontador de pilha

Atenção: o uso dos atalhos abaixo irá corromper o valor atual do apontador de pilha. Por isso, é importante sempre salvar o valor original do apontador e restaurá-lo antes de voltar a usar a pilha normalmente, com instruções como *PUSH e RET*.

O registrador SP na arquitetura do Sapiens é utilizado como o apontador de pilha. Isto é, ele guarda um ponteiro que indica o endereço atual do topo da pilha. No total, o Sapiens dispõe de quatro instruções que tratam diretamente com o valor desse registrador: **PUSH, POP, STS e LDS**. As duas primeiras lidam com a inserção e remoção de valores na pilha, já as duas últimas servem de mecanismo para salvar e recuperar o valor do apontador. Até o momento, temos usado este registrador única e exclusivamente para sua função projetada. Isto é, para manipular a pilha e seus valores. Porém, também podemos usar o registrador com propósito um pouco mais geral. Isso é importante porque, ao contrário do acumulador, o apontador de pilha trabalha com valores de 16 bits. Com isso, ele pode nos ajudar em certos casos a lidar com valores de 16 bits com maior facilidade. Por exemplo, na transferência de valores com essa largura:

```
LDS  A       ; B = A
STS  B
```

O trecho de código acima, em apenas dois comandos, copia dois bytes do endereço A para o B. A melhoria se torna ainda mais evidente quando usamos o modo de endereçamento indireto. Por exemplo, observe o seguinte trecho, que faz uso do registrador de pilha:

```
   LDS  @A      ; B = *A
   STS  B
```

Agora contraste este trecho com o seguinte, no qual nos limitamos apenas ao uso do acumulador, e perceba como ele se torna bem mais complexo:

```
; Copia byte de *A para B
   LDA  @A
   STA  B
; TEMP = A + 1
   LDA  A
   ADD  #1
   STA  TEMP
   LDA  A+1
   ADC  #0
   STA  TEMP+1
; Copia byte *TEMP para B+1
   LDA  @TEMP
   STA  B+1
```

O apontador de pilha também pode ser útil quando se quer incrementar uma variável de 16 bits de maneira concisa. Neste caso, a instrução POP pode ser utilizada:

```
  LDS  PTR      ; PTR++
  POP
  STS  PTR
```

Por outro lado, a instrução PUSH não deve ser utilizada, já que há o risco de corromper uma posição inesperada da memória.

Capítulo 7

SimuS no Raspberry Pi

Nesta seção apresentamos as modificações que foram realizadas no simulador SimuS, para o processador hipotético Sapiens, de modo a permitir o acesso aos pinos de GPIO do nanocomputador Raspberry Pi (FOUNDATION, 2017).

Além dos tradicionais dispositivos que são emulados no próprio simulador SimuS, como o conjunto de chaves e o visor hexadecimal, pretendemos ampliar o espectro de possibilidades ao alcance do professor, com o acesso aos pinos de GPIO diretamente a partir do código executado no simulador.

Entendemos que essa possibilidade de manipulação de dispositivos concretos, através do contato do aluno com componentes eletrônicos como relés, diodos, leds, displays, sensores de temperatura, pressão, entre outros, é extremamente benéfica e facilitadora da compreensão dos conceitos básicos de construção e funcionamento dos computadores.

7.1 Instalação do SimuS

O processo de transporte do SimuS para sistema operacional Raspbian se iniciou com a instalação do compilador Free Pascal e do ambiente de desenvolvimento Lazarus, de modo que possam ser gerados códigos-objeto compatíveis com a arquitetura ARM, de 64 bits, que é o processador utilizado no Raspberry Pi. Embora no site oficial do Lazarus (LAZARUS; TEAM, 2017) encontremos apenas versões disponíveis para a arquitetura Intel, é possível baixar versões compiladas para o processador ARM no endereço <https://www.getlazarus.org/setup/>. Para tanto deve ser baixado um “script” de instalação setup.sh e em seguida executados os seguintes comandos:

```
$ chmod +x setup.sh
$ ./setup.sh
```

Durante o processo de instalação são verificadas se as dependências com os pacotes necessários ao correto funcionamento do Lazarus e Free Pascal estão satisfeitas. Caso seja detectada algum falha de dependência, os pacotes devem ser instalados manualmente.

De posse dessas ferramentas configuradas no sistema operacional, a instalação do simulador SimuS no sistema operacional Raspbian foi direta, bastando apenas baixar o código fonte e compilar o mesmo no novo ambiente. O programa executou sem nenhum problema e com desempenho similar ao encontrado em ambientes do tipo Linux ou Windows com processadores x86 ou x86_64.

7.2 Acesso ao Pinos do GPIO no Raspbian

Um dos objetivos no desenvolvimento do Raspberry Pi foi facilitar o acesso sem esforço a dispositivos externos como sensores e atuadores. Os pinos de GPIO do Raspberry na realidade não são todos iguais, possuindo diversos tipos de conexão:

- Pinos verdadeiros de GPIO que podem ser usados para ativar e desativar LEDs, leitura de chaves, etc;
- Pinos seriais de RX e TX para comunicação com periféricos seriais.
- Pinos de interface I2C (SCL e SDA), que permitem conectar módulos de *hardware* com apenas dois pinos de controle;
- Pinos de Interface SPI (MOSI, MISO e SCKL), um conceito semelhante ao I2C, mas com um padrão diferente;

Além disso, alguns dos pinos podem ser usados para PWM (modulação de largura de pulso) para controle de potência (motores, intensidade de LEDs, etc.) e outro tipo de geração de pulsos para controle de servo motores denominado PPM (Modulação de Posição de Pulso).

Todos os pinos possuem níveis de lógica de 3,3V e não são seguros para 5V, portanto os níveis de saída são 0-3,3V e as entradas não devem ser superiores a 3,3V. Se você deseja conectar uma saída de 5V a uma entrada do Raspberry Pi deve usar um divisor resistivo ou circuito conversor de nível lógico. Os pinos I2C e SPI são compartilhados com pinos de GPIO

convencionais e estão desabilitados por padrão no Raspbian, já os pinos de TX/RX também são compartilhados, mas estão habilitados por padrão. Alterações nestas configurações podem ser feitas com uso do utilitário “raspi-config”.

De uma foma geral, esse pinos podem ser acessados principalmente dos seguintes modos a partir de um programa escrito em Object Pascal:

1. Através de registradores de E/S mapeados em memória:

    - Esta forma pode ser utilizada diretamente, com o uso de rotinas de mapeamento (mmap) da memória física no espaço de endereçamento virtual do processo, ou indiretamente através de bibliotecas como wiringPi e PXL.

2. Através do acesso ao sistema de arquivos dentro do diretório /sys/class/gpio:

    - Esta forma pode ser utilizada com chamadas embutidas ao shell, efetuadas com uso da rotina fpsystem() da unit Unix; ou com o uso das rotinas fpopen(), fpread() e fpwrite() da unit BaseUnix.

Em qualquer dos casos há a necessidade de os programas serem executados em uma conta com privilégios para ser possível o acesso ao espaço de endereçamento de E/S do sistema operacional. Para evitar isso, alguns artifícios podem ser utilizados, com uso do programa executável com bit “setuid” ligado, tendo o “root” como proprietário do arquivo. Nas versões mais recentes do Raspbian, o usuário “pi” pertence a um grupo especial “gpio”, que tem privilégios para acesso aos pinos do GPIO.

7.3 Acesso ao conector GPIO do Raspberry Pi

Ao considerar as possibilidades de acesso aos pinos do conector GPIO, gostaríamos de oferecer ao programador em linguagem de montagem uma forma fácil e rápida de realizar este acesso. Levamos porém em conta o fato de que o *hardware* do Raspberry Pi tem sofrido modificações e evoluções ao longo do tempo e que, certamente, novas mudanças surgirão e, portanto, procuramos tornar este acesso o menos dependente possível das

especificações do *hardware*.

A solução encontrada foi utilizar o mecanismo já existente no SimuS disponível com a instrução de TRAP, realizando uma “chamada de sistema”, com passagem de parâmetros relativos a operação de E/S desejada na memória do programa em execução.

A partir deste ponto a execução é transferida para simulador, que realiza as operações de E/S, reais neste caso, requisitadas. Como o desempenho não é uma questão fundamental, sendo esperado que o acesso aos pinos GPIO seja feito apenas para executar operações simples de E/S (por exemplo, piscar um LED, ler o valor de um pino), optamos pela maneira mais fácil e portátil de implementação, e o mais possível imune às eventuais mudanças que possam ocorrer no *hardware* para as próximas versões do Raspberry Pi.

Como observação adicional, deixamos claro que essas funções adicionais só estão disponíveis quando o SimuS é executado no Raspberry Pi, não tendo efeito quando utilizado em outros sistemas, seja com o sistema operacional Linux ou Windows. As rotinas de TRAP retornam silenciosamente nestes casos e nenhuma mensagem de erro é passada para o usuário.

As necessidades das operações de E/S envolvem basicamente a programação dos pinos como entrada, saída ou PWM; a escrita nos pinos de sinais digitais ou PWM; e a leitura de sinais digitais dos pinos. O suporte a interrupções não foi implementado. O acesso ao *hardware* encapsulado por “shell calls”, resolve satisfatoriamente os primeiros casos, mas não permite a leitura de valores digitais desses pinos. Para evitar o uso de operações de E/S mapeadas em memória, escolhemos utilizar a biblioteca wiringPi (HENDERSON, 2017), com um “wrapper” , de nome hwiringpi, para uso a partir dos programas escritos em Object Pascal.

Retomando a questão da numeração dos pinos, a biblioteca wiringpi oferece ainda um terceiro método de numeração “virtual” para acessos aos pinos do GPIO, como tentativa de estabelecer um padrão independente das mudanças de *hardware* no Raspberry Pi. Apesar disso, adotamos a numeração dos pinos conforme os mesmos são vistos pelo controlador de GPIO, por acreditarmos que seja a mais simples e com menor possibilidade de indução a erros por parte dos usuários.

Mostramos um exemplo básico de programação em Internet das Coisas, que é um programa em linguagem de alto nível, para fazer um LED piscar:

```
begin
wiringPiSetup();
pinMode(p22, OUTPUT);
While true do begin
  digitalWrite(p22, HIGH);  // Liga LED em P22
  delay(500);
  digitalWrite(p22, LOW);  // Desliga o LED
  delay(500);
  end;
end.
```

Exemplo 7.1: LED Piscando em Linguagem de Alto Nível

7.4 Rotinas de TRAP no Raspberry Pi

As seguintes rotinas de TRAP foram então implementadas no SimuS para acesso dos programas simulados aos pinos de GPIO do Raspberry Pi:

- 101 – rotina para configurar certo pino como entrada, saída comum, ou saída com modulação por largura de pulso (PWM), ou libera a alocação desta porta. Os parâmetros, que são passados no endereço de memória definido pelo operando da instrução TRAP, são o número do PINO (1 byte) e o MODO (1 byte), que poder ter os seguintes valores: 0 – ENTRADA, 1 – SAÍDA, 3 – PWM, 4 – libera o uso desta porta.
- 102 – escreve o valor lógico alto (1) ou baixo (0) no pino que deve ter sido previamente configurado como saída. Os parâmetros, que são passados no endereço de memória definido pelo operando da instrução TRAP, são o número do PINO (1 byte) e VALOR (1 byte), onde qualquer valor diferente de 0 é tratado como NÍVEL ALTO, e apenas 0 é NÍVEL BAIXO.
- 103 – retorna no acumulador o valor lido em um dado pino, podendo ser nível lógico baixo (0) ou alto (1). Os parâmetros, que são passados no

endereço de memória definido pelo operando da instrução TRAP, são o número do PINO (1 byte), apenas.

- 104 –Rotina para configurar o modo de resistência “pull-up” ou “pull-down” em um dado pino, que deve ser configurado necessariamente como uma entrada. O BCM2835 do Raspberry Pi tem resistores internos tanto de “pull-up” como de “pull-down”. Os parâmetros, que são passados no endereço de memória definido pelo operando da instrução TRAP, são o número do PINO (1 byte) e PUD (1 byte), que poder ter os seguintes valores: 0 – PUD_OFF, (sem nenhum resistor), 1 – PUD_DOWN (resistor de “pull-down” para terra) ou 2 – PUD_UP (resistor de “pull-up” para 3,3V). Os resistores internos de “pull up/down” têm um valor de aproximadamente 50 K*ohms* no Raspberry Pi.
- 105 – configura o “duty cycle” do registrador PWM do pino correspondente que deve ter sido previamente configurado como saída. Os parâmetros, que são passados no endereço de memória definido pelo operando da instrução TRAP, são o número do PINO (1 byte) e VALOR (2 bytes) do “duty cycle”. No Raspberry Pi o modo PWM possui uma frequência de 600 kHz e valores entre 0 e 1023 para o "duty cycle"da forma de onda.

7.5 Exemplos de Programas

A seguir apresentamos alguns exemplos de código em linguagem de montagem do Sapiens, começando com uma versão do tradicional programa “blink” para piscar um led ligado ao pino 13 do GPIO.

```
   ORG 0
; Pino desejado = 13
   LDA  #13
   STA  PIN_TRAP
; Modo de saída
   LDA  #1
   STA  PIN_TRAP +1
   LDA  #101
```

```
  TRAP PIN_TRAP
LOOP:
; Coloca ’1’  no pino 13
  LDA  #1
  STA  PIN_TRAP+1
  LDA  #102
  TRAP PIN_TRAP
; Espera (1000 ms)
  LDA  #5
  TRAP T1000
; Coloca ’0’  no pino 13
  LDA  #0
  STA  PIN_TRAP +1
  LDA  #102
  TRAP PIN_TRAP
; Espera (1000 ms)
  LDA  #5
  TRAP T1000
  JMP  LOOP
T1000:   DW 1000
PIN_TRAP:DS 3
  END  0
```

Exemplo 7.2: LED Piscando

A seguir mostramos um exemplo de um programa para fazer o controle do brilho de um LED ligado ao pino 18 do GPIO.

```
  ORG  0
; Pino desejado = 18
  LDA  #PWM_PIN
```

```
  STA  TRAP_PARAM
; Modo PWM
  LDA  #PWM_MODE
  STA  TRAP_PARAM +1
  LDA  #MODO_TRAP
  TRAP TRAP_PARAM
; Loop de fade in / fade out
LOOP:
  JSR  FADEIN
  JSR  FADEOUT
  JMP  LOOP
;-----------------------------------
FADEIN:
; Valor inicial PWM = 0
  LDA  #0
  STA  PWM_HIGH
  STA  PWM_LOW
LOOPFIN:
; Gera sinal PWM (0 - 1023)
  JSR  TRAP_PWM
; Lê a parte baixa do PWM
  LDA  PWM_LOW
; Incrementa de 16 (passo)
  ADD  #16
; Armazena 8 bits da parte baixa
  STA  PWM_LOW
  LDA  #0
; Se deu carry soma 1 na parte alta
  ADC  PWM_HIGH
```

```
; Armazena 8 bits da parte alta
  STA  PWM_HIGH
; Se chegou ao final (ou seja 1024)
  SUB  #04H
  JNZ  LOOPFIN
; Retorna
    RET
;-----------------------------------
FADEOUT:
; Valor inicial PWM = 1008
  LDA  #03
  STA  PWM_HIGH
  LDA  #0F0H
  STA  PWM_LOW
LOOPFOUT:
; Gera sinal PWM (0 - 1023)
  JSR  TRAP_PWM
; Lê da parte baixa do PWM
  LDA  PWM_LOW
; Diminui de 16
  SUB  #16
; Armazena 8 bits da parte baixa
  STA   PWM_LOW
; Carrega parte alta
  LDA   PWM_HIGH
; Se deu carry subtrai 1 da parte alta
  SBC   #0
; armazena 8 bits da parte alta
  STA   PWM_HIGH
```

```
  JP    LOOPFOUT
  JZ    LOOPFOUT
; Se for negativo retorna
  RET
;-----------------------------------
TRAP_PWM:
; prepara parametros
  LDA   PWM_LOW
  STA   TRAP_PARAM+1
  LDA   PWM_HIGH
  STA   TRAP_PARAM+2
  LDA   #PWM_TRAP
; executa o trap
  TRAP  TRAP_PARAM
; espera 200 Milissegundos
  LDA   #DELAY_TRAP
  TRAP  TIME_200
  RET
;-----------------------------------
TIME_200:   DW  200
  ; área para passar parâmetros do trap
TRAP_PARAM: DS  3
; Valor do PWM
PWM_LOW:    DB  0
PWM_HIGH:   DB  0
; pino PWM (18 é o único por hardware)
PWM_PIN     EQU   18
; parâmetro do modo do pino PWM
PWM_MODE    EQU 3
```

```
PWM_TRAP    EQU 105
MODO_TRAP   EQU 103
DELAY_TRAP  EQU 5
;
  END   0
```

Exemplo 7.3: Controle do Brilho do LED

Apêndices

Apêndice A

Arquitetura de Computadores

Neste apêndice apresentamos uma breve introdução sobre os conceitos básicos de arquitetura de computadores, tais como os principais tipos de arquitetura de processadores e os modos de endereçamento como complemento ao conteúdo apresentado neste livro.

A.1 Tipos de Arquitetura de Processadores

Os processadores podem ter diversos tipos de arquitetura que se diferenciam basicamente em relação à forma de acesso e de armazenamento interno dos operandos para a execução das instruções. Faremos nesta seção uma breve introdução aos seguintes tipos de arquitetura:

- **Pilha**
- **Acumulador**
- **Memória-Memória**
- **Registrador-Memória**
- **Registrador-Registrador**

Para cada uma dessas arquiteturas mostraremos como realizar a operação $C := A + B$ em detalhes.

A.1.1 Arquitetura de Pilha

Nos processadores com arquitetura de pilha todos os operandos implicitamente estão no topo da pilha. Com isso, as instruções gastam menos bits, quando comparados com as arquiteturas com registrador, para codificar a origem e destino das operações. Isso era uma propriedade importante nos computadores mais antigos, pois resultava em um tamanho menor para a codificação das instruções, permitindo uma significativa economia de memória.

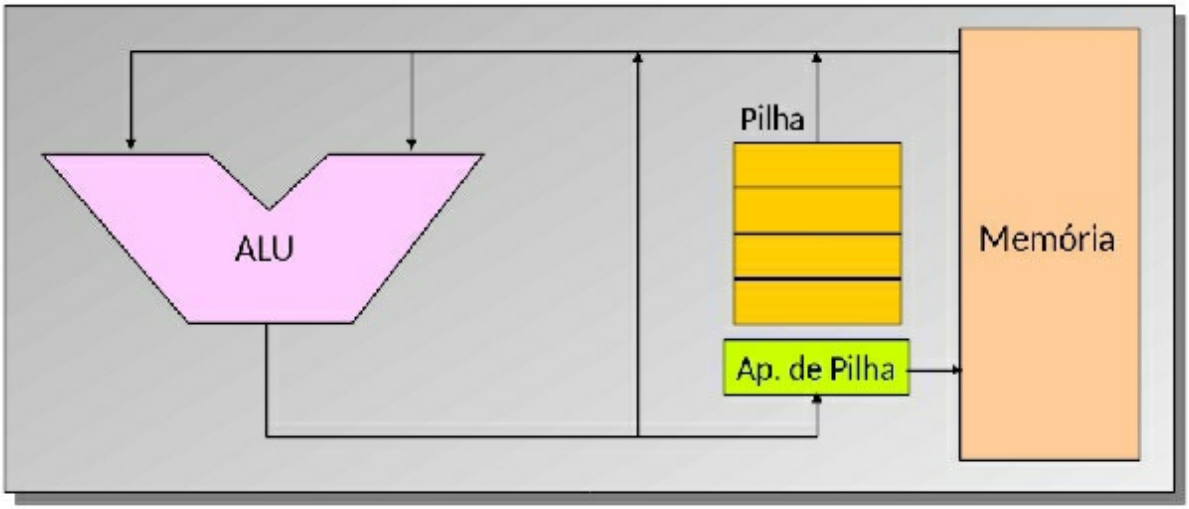

Figura A.1: Arquitetura de Pilha

As únicas operações de acesso à memória são ”POP” e ”PUSH”, para retirar e colocar um operando no topo da pilha. Normalmente os primeiros elementos no topo da pilha se encontram junto ao processador e o restante da pilha é distribuído na memória a partir do endereço dado pelo apontador de pilha. A Figura A.1 representa um modelo desta arquitetura. A máquina virtual Java pode ser considerado um exemplo moderno deste tipo de arquitetura.

```
PUSH  A
PUSH  B
ADD
POP   C
```

No exemplo mostrado acima, podemos verificar que a operação $C := A + B$ é feita colocando-se primeiro o valor da variável A no topo da pilha, com a a instrução **PUSH A**. Logo a seguir a variável B é colocado no topo da pilha com a instrução **PUSH B**. Agora, com os valores de A e B já colocados nas duas primeiras posições da pilha, a instrução **ADD** é executada, somando esses dois valores e o resultado é colocado por sua vez no topo da pilha. A seguir o resultado é retirado do topo da pilha e escrito na variável C em memória com a instrução **POP C**.

A.1.2 Arquitetura de Acumulador

A arquitetura de acumulador possui o acumulador com um dos operandos implícitos na maioria instruções. O acumulador é um registrador especial colocado junto à unidade aritmética e lógica (UAL) com o intuito de agilizar as operações que são realizadas pelo processador. Se todas as operações fossem realizadas diretamente com todos os operandos em memória provavelmente haveria um grande impacto negativo no desempenho desses processadores, já que tempo gasto para acessar os dados na memória é bem maior que o tempo para acessar o dado armazenado no acumulador.

É um modelo de arquitetura muito utilizado em processadores mais simples, tantos nos processadores mais antigos como nos modernos processadores embarcados, por ser de fácil implementação e ser uma solução de compromisso entre custo e desempenho. Exemplos modernos de arquitetura de acumulador são os processadores PIC e o 8051. A Figura A.2 representa um modelo desta arquitetura.

```
LOAD  A
ADD   B
STORE C
```

No exemplo mostrado acima, podemos verificar que a operação $C := A + B$ é feita colocando-e primeiro o valor da variável A no acumulador, com a instrução **LOAD A**. Logo a seguir o acumulador é somado com a variável B com a instrução **ADD B**, sendo que o resultado é colocado de volta no acumulador. A seguir o resultado é retirado do acumulador e escrito na variável C em memória com a instrução **STORE C**.

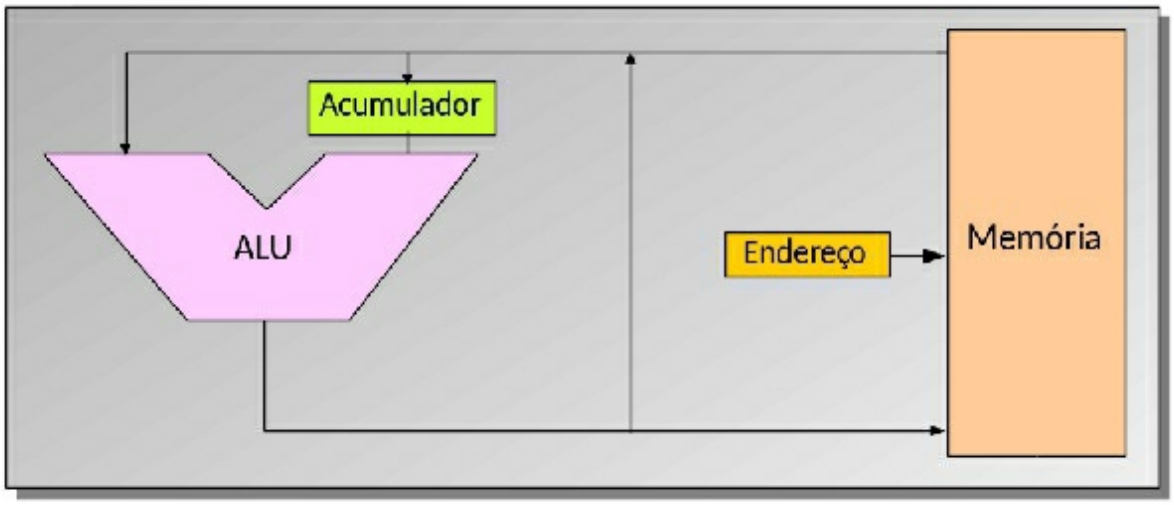

Figura A.2: Arquitetura de Acumulador

A.1.3 Arquitetura Memória-Memória

Os processadores com arquitetura memória-memória possuem instruções aritméticas e lógicas com todos os operandos colocados de forma explícita, ou seja, possuem um total de 3 operandos, podendo estar todos em memória. Embora em sua maioria não dispensem o uso também de registradores, as instruções aritméticas e lógicas podem referenciar todas as suas variáveis diretamente na memória. Normalmente o código gerado para essas máquinas possui um número menor de instruções mas que, por sua vez, resultam em arquiteturas com implementação bastante complexa. Um exemplo de arquitetura deste tipo é o VAX, um processador bastante comercializado nas décadas de 70 e 80 no século passado.

```
ADD C, B, A
```

No exemplo mostrado acima, podemos verificar que a operação $C := A + B$ é feita com apenas uma única instrução **ADD C, B, A** que lê as variáveis A e B da memória, realiza a soma e escreve o resultado de volta na memória na variável C.

A.1.4 Arquitetura Registrador-Memória

Os processadores com arquitetura registrador-memória normalmente possuem instruções aritméticas e lógicas com dois operandos, sendo que um deles está em memória e o outro em registrador. O resultado da operação é escrito automaticamente no mesmo registrador. Os exemplos clássicos deste tipo arquitetura são o IBM 360 e os processadores da linha Intel x86.

```
LOAD  R1, A
ADD   R1, B
STORE C, R1
```

No exemplo mostrado acima podemos verificar que a operação *C* := *A* + *B* é realizada colocando-se primeiro o valor da variável A no registrador R1, com a instrução **LOAD R1, A**. Logo a seguir o valor armazenado no registrador é somado com a variável B com a instrução **ADD R1, B**, sendo que o resultado é colocado de volta no registrador R1. A seguir o resultado é copiado do registrador R1 e escrito na variável C em memória com a instrução **STORE C, R1**.

A.1.5 Arquitetura Registrador-Registrador

Os processadores com arquitetura registrador-registrador possuem características particulares. A primeira delas é que as instruções aritméticas e lógicas possuem os três operandos em registrador, sendo dois de origem e um destino. As únicas instruções que fazem acesso à memória são ”LOAD”, para carregar os dados da memória para os registradores e ”STORE”, para fazer o caminho inverso.

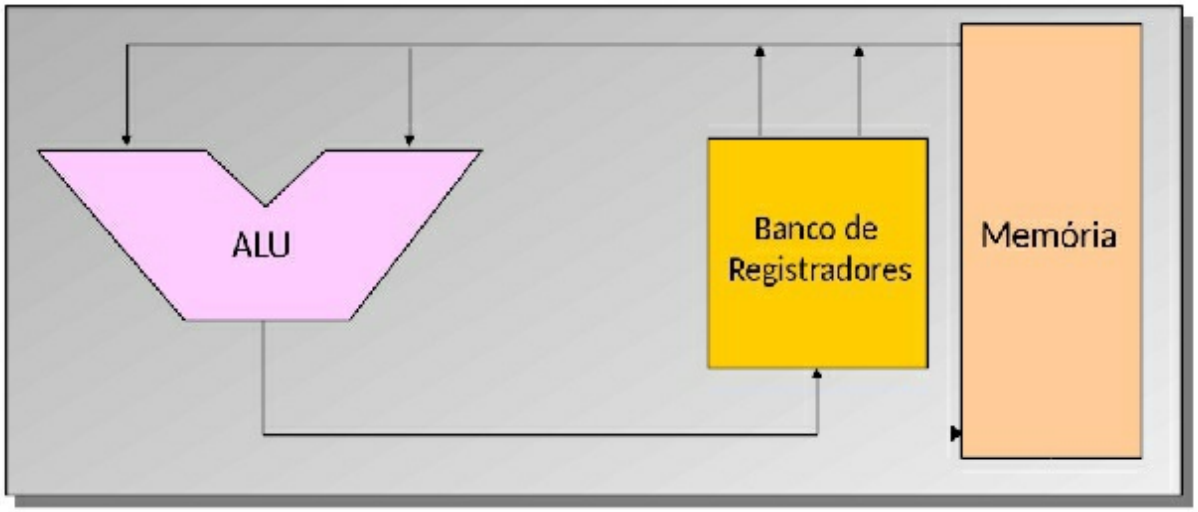

Figura A.3: Arquitetura de Registrador

A compilação dos programas em alto nível resultam um código com um maior número de instruções, porém mais simples. Essas características permitem o desenvolvimento de projetos de processadores mais simples e, em consequência, mais rápidos e com menor consumo de energia. Por isso é o modelo de arquitetura, característico das arquiteturas RISC, é o mais disseminado entre os processadores comerciais atualmente. A Figura A.3 apresenta um modelo simplificado desta arquitetura.

```
LOAD  R1, A
LOAD  R2, B
ADD   R3, R1, R2
STORE C, R3
```

No exemplo mostrado acima, podemos verificar que a operação *C* := *A* + *B* é feita colocando-se primeiro o valor da variável A no registrador R1 com a instrução **LOAD R1, A**. Logo a seguir o registrador R2 recebe o valor da variável B com a instrução **LOAD R1, B**. Depois conteúdo de R1 é somado com R2 e o resultado é colocado no registrador R3. A seguir o valor armazenado em R3 é escrito na variável C em memória com a instrução **STORE C, R3**.

A.2 Modos de Endereçamento

Em uma arquitetura de processador as instruções podem ter diversos modos de acessar os seus operandos. Os operandos podem ser constantes ou variáveis em registrador ou memória. Alguns modos de endereçamento existentes podem ser vistos No capítulo seguinte especificamos quais modos de endereçamento são suportados pelo processador Sapiens.

- **Registrador ou Acumulador**

  ADD R4,R3 ⇒ Regs[4] ← Regs[4]+Regs[3]

  Operando no registrador ou acumulador.

- **Imediato**

  ADD R4,#8 ⇒ Regs[4] ← Regs[4]+8

  Operando é constante na instrução.

- **Direto ou Absoluto**

  ADD R1,(1001) ⇒ Regs[1] ← Regs[1]+Mem[1001]

  Operando em memória com endereço na instrução.

- **Indireto (via Registrador)**

  ADD R4,(R1) ⇒ Regs[4] ← Regs[4]+Mem[Reg[1]]

  Operando em memória com endereço no registrador.

- **Indireto via Memória**

  ADD R2,@(1003) ⇒ Regs[2] ← Regs[2]+Mem[Mem[1003]]

  Operando em memória com endereço na memória.

- **Pilha**

  PUSH ⇒ SP = SP +1 ; MEM[SP] ← ACC

A pilha é um operando implícito da instrução

Note que uma instrução pode fazer uso de modos diferentes para buscar cada um dos seus operandos.

- No modo registrador ou acumulador, o operando da instrução está armazenado em um registrador ou no acumulador. Note que no caso do acumulador, ele será um operando implícito da instrução, e não aparecerá no código em linguagem de montagem.
- No modo imediato o valor do operando está codificado na própria instrução ou em *bytes* subsequentes a ela.
- No modo direto, o valor codificado na instrução (ou nos *bytes* subsequentes) não é operando em si, mas o endereço do operando na memória. Um novo acesso precisa ser realizado para acessar o dado propriamente dito.
- No modo indireto, alguma vezes referenciado como indireto via registrador, o registrador não possui o operando, mas o endereço do operando na memória. Aqui também é necessário um acesso à memória para buscar o operando.
- No modo indireto via memória, a instrução (ou os *bytes* subsequentes) contém o endereço de memória da posição de memória que tem o endereço do operando. São necessários mais dois acessos à memoria para buscar o operando da instrução.
- No modo pilha, o apontador de pilha (SP) é utilizado para indexar a memória e buscar o operando, de uma forma implícita .

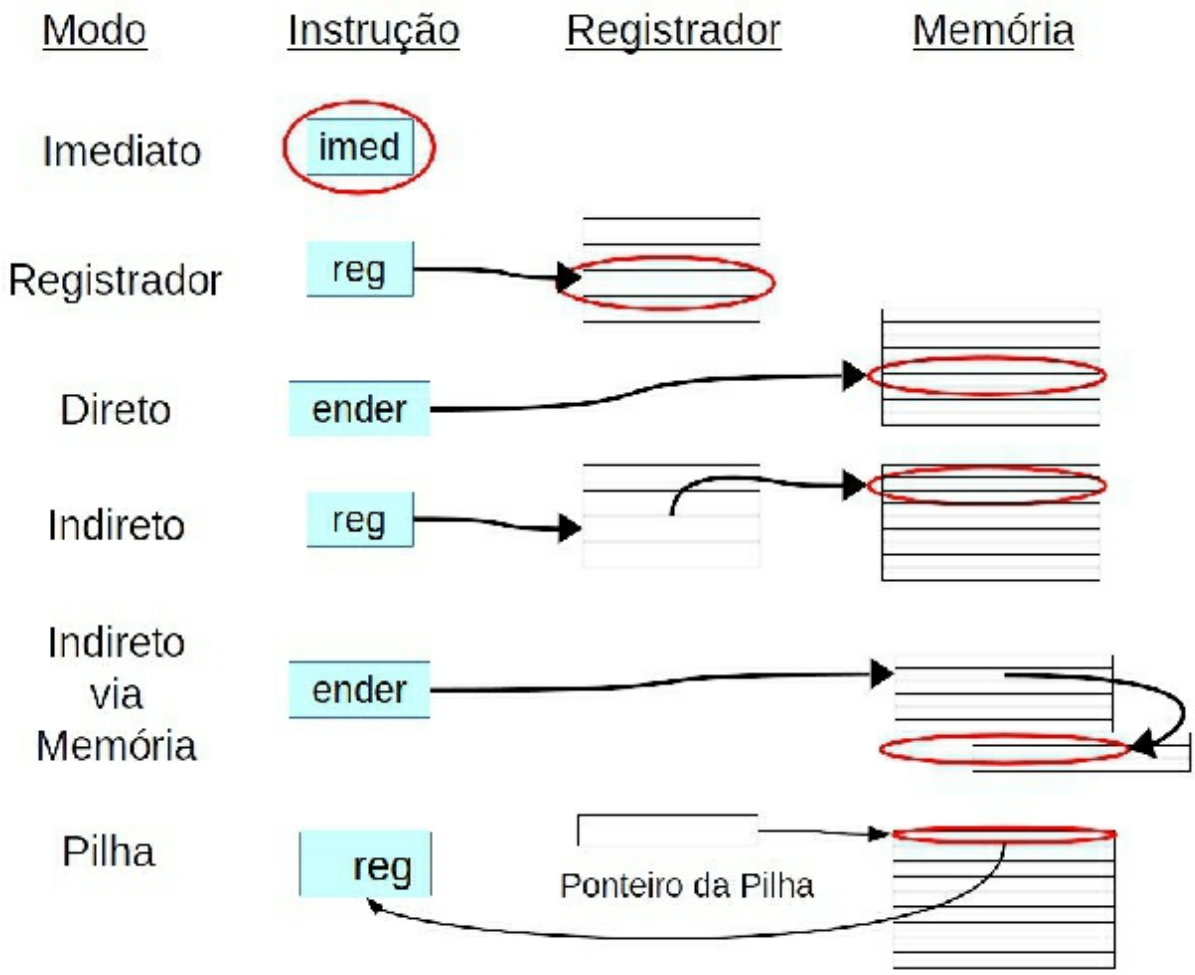

Figura A.4: Modos de Endereçamento

Existem ainda outros modos de endereçamento mais complexos utilizado por diversos processadores, mas cujo estudo foge ao escopo deste livro. Para maiores informações recomendamos a consulta ao livro de Hennessy e Patterson (HENNESSY, 2003).

Apêndice B

Detalhamento do Processador Sapiens

B.1 Microarquitetura

No diagrama em blocos da Figura B.1 podemos visualizar diversos elementos que compõem a microarquitetura do processador Sapiens. Procuramos explicar com mais detalhes o seu funcionamento a seguir.

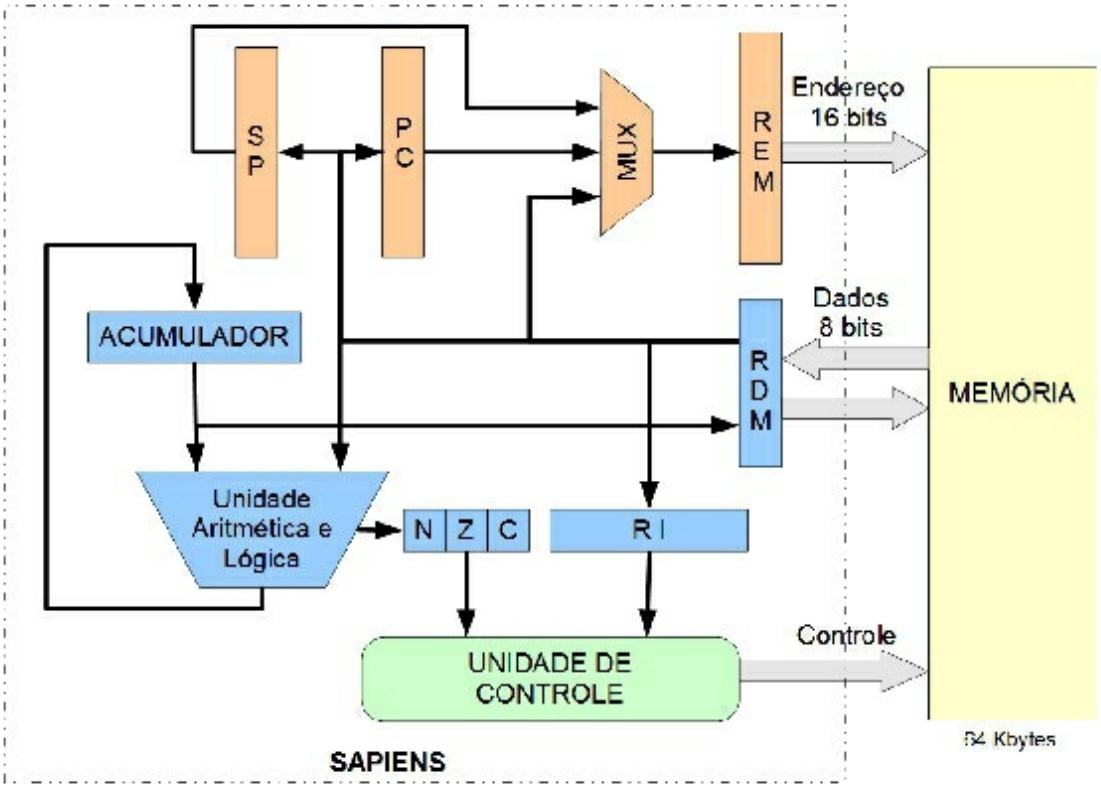

Figura B.1: Processador Sapiens

O programa, que é o conjunto de instruções e dados no formato binário, é

carregado pelo simulador na memória. A partir do endereço inicial carregado pelo simulador no apontador de instruções (PC, do inglês ”program counter”), o processador realiza permanentemente o ciclo de busca de instruções na memória, sendo que o valor do PC é automaticamente incrementado de 1 após cada leitura de um *byte* da memória. O PC armazena sempre o endereço da próxima instrução a ser executada no programa, podendo ser modificado pelas instruções de desvio.

Os acessos à memória são feitos a partir do endereço armazenado no registrador de endereço de memória (REM). O multiplexador (MUX) decide de onde virá o endereço a ser armazenado no REM:

i) Do apontador de instruções (PC), quando o valor do REM é utilizado para acessar as instruções, dados imediatos e endereços que estão no código do programa.
ii) Do valor armazenado no registrador de dados da memória (RDM), quando o endereço no REM serve para acessar os operandos na memória, quer no modo direto (um acesso apenas) ou indireto (dois acessos são necessários).
ii) Do valor armazenado no apontador de pilha (SP), quando o endereço no REM serve para acessar os operandos na pilha ou para salvar e restaurar o PC durante a chamada ou retorno de procedimento.

O ciclo de busca de instruções usa o endereço em REM (vindo do PC) para transferir as instruções da memória para o RDM e daí então para o registrador de instruções (RI). Ao chegar no RI a instrução é analisada pela unidade de controle e, de acordo com o seu código de operação, os caminhos de dados e registradores internos do processador são acionados adequadamente para execução da instrução.

No ciclo de execução da instrução, o valor em REM serve para a busca de operandos, no modo imediato, direto ou indireto de endereçamento ou para acesso à pilha. Os operandos são lidos da memória para o RDM, podendo então, em geral, ocorrer uma das seguintes situações:

i) Ser armazenado diretamente no acumulador (AC);
ii) Ser usado como o segundo operando de uma operação aritmética ou lógica a ser realizada com o acumulador(AC) pela UAL, com o resultado

sendo novamente escrito no acumulador (AC);

iii) Ser armazenado no apontador de instruções (PC), no caso de uma instrução de desvio;

iv) Ser movido para o REM, para definir o endereço de memória do operando. No modo indireto de endereçamento, essa operação ocorre ainda mais uma vez.

v) Ser movido para o apontador de pilha (SP), no caso da instrução LDS.

A unidade aritmética e lógica (UAL) é responsável pela realização das operações artiméticas e lógicas definidas pelas instruções. Recebe dois operandos na entrada, sendo um deles o acumulador (AC) e o resultado das operações é sempre escrito de volta no acumulador (AC). Em algumas instruções esse resultado afeta o valor dos *flags* **N, Z e C**. Se houver *carry* o flag **C** vai para ’1’. Se o resultado for negativo o *flag* **N** vai para ’1’, caso contrário fica em ’0’. Se o resultado for igual a zero, o *flag* Z vai para ’1’, caso contrário fica em ’0’. Como consequência, se os dois *flags* N e Z são ’0’ o resultado é positivo.

Todos os registradores possuem 8 bits de largura, com exceção dos que endereçam a memória, como PC e SP, que possuem 16 bits. Isso define o tamanho máximo de memória do simulador SimuS em 64 Kbytes. Não estão representados no diagrama em blocos os dispositivos de E/S, que são acessados por instruções especiais de entrada e saída (**IN** e **OUT**) em um espaço de endereçamento de E/S à parte.

No link a seguir você pode encontrar um vídeo exemplificando o funcionamento da arquitetura interna do processador Sapiens na execução de um pequeno programa:

https://goo.gl/zyIFuu

B.2 Detalhamento das Instruções do Sapiens

A Figura B.2 apresenta o ciclo de busca de instruções do processador Sapiens. Ou seja, ele está permanentemente executando a busca de uma instrução da memória para o registrador de instruções (RI) e incrementando o

valor do apontador de instruções (PC) para buscar a próxima instrução ou o próximo *byte* da instrução.

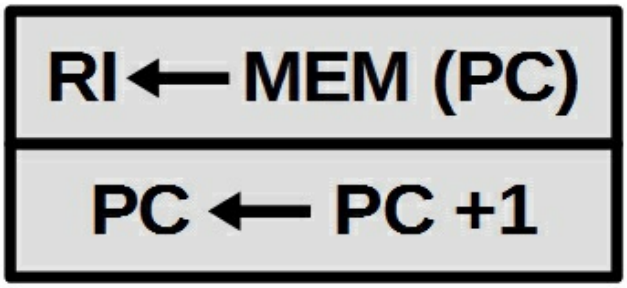

Figura B.2: Ciclo de Busca de Instruções

As Figuras B.3, B.4, B.5 e B.6 apresentam detalhadamente os passos executados pelo processador Sapiens para a execução de cada uma de suas instruções aritméticas, lógicas, de transferência de dados e de transferência de controle (algumas transferências requerem dois acessos à memória quando os registradores são de 16 bits).

Instruções Aritméticas	
Instrução	Ciclo de Execução
ADD ender ADD @ender	REM ← MEM(PC)[1] Se modo indireto REM ← MEM(REM)[1] AC ← AC + MEM(REM) PC ← PC + 2 Atualiza N, Z, C
ADD #imed	AC ← AC + MEM(PC) PC ← PC + 1 Atualiza N, Z, C
ADC ender ADC @ender	REM ← MEM(PC)[1] Se modo indireto REM ← MEM(REM)[1] AC ← AC + MEM(REM) + C PC ← PC + 2 Atualiza N, Z, C
ADC #imed	AC ← AC + MEM(PC) + C PC ← PC + 1 Atualiza N, Z e C
SUB ender SUB @ender	REM ← MEM(PC)[1] Se modo indireto REM ← MEM(REM)[1] AC ← AC - MEM(REM) PC ← PC + 2 Atualiza N, Z , C
SUB #imed	AC ← AC - MEM(PC) PC ← PC + 1 Atualiza N, Z, C
SBC ender SBC @ender	REM ← MEM(PC)[1] Se modo indireto REM ← MEM(REM)[1] AC ← AC - MEM(REM) - C PC ← PC + 2 Atualiza N, Z, C
SBC #imed	AC ← AC - MEM(PC) - C PC ← PC + 1 Atualiza N, Z, C

Figura B.3: Instruções Aritméticas

Instruções Lógicas	
Instrução	Ciclo de Execução
OR ender OR @ender	REM ← MEM(PC)[1] Se modo indireto REM ← MEM(REM)[1] AC ← AC or MEM(REM) PC ← PC + 2 Atualiza N, Z
OR #imed	AC ← AC or MEM(PC) PC ← PC + 1 Atualiza N, Z
XOR ender XOR @ender	REM ← MEM(PC)[1] Se modo indireto REM ← MEM(REM)[1] AC ← AC xor MEM(REM) PC ← PC + 2 Atualiza N, Z
XOR #imed	AC ← AC xor MEM(PC) PC ← PC + 1 Atualiza N, Z
AND ender AND @ender	REM ← MEM(PC)[1] Se modo indireto REM ← MEM(REM)[1] AC ← AC and MEM(REM) PC ← PC + 2 Atualiza N, Z
AND #imed	AC ← AC and MEM(PC) PC ← PC + 1 Atualiza N, Z
NOT	AC ← not AC Atualiza N, Z
SHL	AC ← AC << 1; C = AC(7); AC(0) = 0 Atualiza N, Z e C
SHR	AC ← AC >> 1; AC(7) = 0; C = AC(0) Atualiza N, Z , C
SRA	AC ← AC >> 1; AC(6) = AC(7) ; AC(7) = AC(7); C = AC(0) Atualiza N, Z , C

Figura B.4: Instruções Lógicas

Instruções de Transferência de Dados	
Instrução	**Ciclo de Execução**
STA ender STA @ender	REM ← MEM(PC)[1] Se modo indireto REM ← MEM(REM)[1] MEM(REM) ← AC PC ← PC + 2
STS ender STS @ender	REM ← MEM(PC)[1] Se modo indireto REM ← MEM(REM)[1] MEM(REM) ← SP[1] PC ← PC + 2
LDA ender LDA @ender	REM ← MEM(PC)[1] Se modo indireto REM ← MEM(REM)[1] AC ← MEM(REM) PC ← PC + 2 Atualiza N, Z
LDA #imed	AC ← MEM(PC)[1] PC ← PC + 2 Atualiza N, Z e C
LDS ender LDS @ender	REM ← MEM(PC)[1] Se modo indireto REM ← MEM(REM)[1] SP ← MEM(REM)[1] PC ← PC + 2
LDS #imed	SP ← MEM(PC)[1] PC ← PC + 2
IN ender	REM ← MEM(PC)[1] AC ← IO (REM) PC ← PC + 2
OUT ender	REM ← MEM(PC)[1] IO (REM) ← AC PC ← PC + 2
PUSH	SP ← SP - 1 MEM(SP) ← AC
POP	AC ← MEM(SP) SP ← SP + 1 Atualiza N, Z

Figura B.5: Instruções de Transferência de Dados

Instruções de Transferência de Controle	
Instrução	Ciclo de Execução
JMP ender JMP @ender	REM ← MEM(PC)[1] Se modo indireto PC ← MEM(REM)[1] Senão PC ← PC + 2
JN ender JN @ender	REM ← MEM(PC)[1] Se modo indireto REM ← MEM(REM)[1] if N = 1 then PC ← REM else PC ← PC + 2
JP ender JP @ender	REM ← MEM(PC)[1] Se modo indireto REM ← MEM(REM)[1] if (N = 0 and Z = 0) then PC ← REM else PC ← PC + 2
JZ ender JZ @ender	REM ← MEM(PC)[1] Se modo indireto REM ← MEM(REM)[1] if Z = 1 then PC ← REM else PC ← PC + 2
JNZ ender JNZ @ender	REM ← MEM(PC)[1] Se modo indireto REM ← MEM(REM)[1] if Z = 0 then PC ← REM else PC ← PC + 2
JC ender JC @ender	REM ← MEM(PC)[1] Se modo indireto REM ← MEM(REM)[1] if C = 1 then PC ← REM else PC ← PC + 2
JNC ender JNC @ender	REM ← MEM(PC)[1] Se modo indireto REM ← MEM(REM)[1] if C = 0 then PC ← REM else PC ← PC + 2
JSR ender JSR @ender	REM ← MEM(PC)[1] Se modo indireto REM ← MEM(REM)[1] SP ← SP – 2 MEM(SP) ← PC[1] PC ← REM
RET	PC ← MEM(SP)[1] SP ← SP + 2

Instruções Especiais	
Instrução	Ciclo de Execução
NOP	Nenhuma operação
HALT	Termina a execução

Figura B.6: Instruções de Transferência de Controle e Especiais

Na notação utilizada, PC é o apontador de instruções e tem o endereço da próxima instrução a ser executada; SP é apontador de pilha; AC é o acumulador; RI é o registrador de instruções; REM é o registrador de endereço de memória; MEM é um vetor que representa a memória; IO é um vetor que representa o espaço de endereçamento dos dispositivos de E/S.

Todas as movimentação de operandos de/para a memória, tais como REM ← MEM(PC), passam sempre pelo RDM, requerem dois acesso à memória, mas por simplificação isso foi omitido na tabela.

Quando a instrução está buscando um operando no modo imediato, o valor a ser recebido pelo registrador de dados RDM é: RDM ← MEM(PC)), ou seja o *byte* seguinte à instrução é o operando da instrução. Do RDM o dado é transferido para o acumulador (AC), o que é o caso mais frequente, ou para o apontador de pilha (SP) no caso da instrução **LDS**. Neste último caso, são necessários dois acessos à memória para carregar o SP. As instruções que admitem o modo imediato de endereçamento são **LDA, LDS, ADD, ADC, SUB, SBC, OR, XOR e AND**.

Quando a instrução está buscando um operando no modo direto, o valor a ser recebido por *REM* é: REM ← MEM(PC), ou seja os dois bytes seguintes à instrução são o endereço do ponteiro para o operando e é necessário mais um acesso à memória para buscar o operando. As instruções que admitem o modo direto são **STA, STS, LDA, LDS, ADD, ADC, SUB, SBC, OR, XOR, AND**. As intruções **JMP, JN, JP, JZ, JNZ, JC, JNC e JSR** usam o modo direto para alterar o valor do PC, mas não buscam um operando adicional como as instruções anteriores.

Quando a instrução está buscando um operando no modo indireto, o valor a ser recebido por *REM* é: REM ← MEM(MEM(PC)), ou seja os dois bytes seguintes à instrução são o endereço do ponteiro para o operando e são necessários mais três acessos à memória para buscar o operando, dois para o endereço armazenado no ponteiro e mais um para o operando propriamente dito. As instruções que admitem o modo indireto são **STA, STS, LDA, LDS,**

ADD, ADC, SUB, SBC, OR, XOR, AND. As instruções **JMP, JN, JP, JZ, JNZ, JC, JNC e JSR** usam o modo indireto para alterar o valor do PC, mas não buscam um operando adicional como as instruções anteriores.

Apêndice C

Arquitetura do Processador Raspberry Pi

C.1 Introdução

O Raspberry Pi é um nanocomputador de baixo custo e consumo de energia, no tamanho aproximado de um cartão de crédito, dotado de um sistema operacional do tipo Linux, desenvolvido primariamente para o ensino de computação. Nos últimos anos o Raspberry Pi teve o seu poder de processamento aumentado consideravelmente, permitindo o uso em diversas aplicações no domínio da Internet das Coisas.

Neste sentido, o Raspberry Pi apresenta um conjunto de pinos de E/S (GPIO – General Purpose Input Output), onde é possível a conexão de sensores e atuadores, de forma a controlar dispositivos externos com diversas funções. O seu uso possibilita ao educador enriquecer a experiência de ensino-aprendizagem, com a realização de diversos experimentos, desde os mais simples, como acender um led, até aqueles mais sofisticados, como a montagem de uma estação meteorológica acessível através da internet.

Entendemos que essa possibilidade de manipulação de dispositivos concretos, através do contato do aluno com componentes eletrônicos como relés, diodos, leds, displays, sensores de temperatura, pressão, entre outros, é extremamente benéfica e facilitadora da compreensão dos conceitos básicos de construção e funcionamento dos computadores.

Aliando-se a isso, temos a possibilidade de uso do simulador SimuS, onde todo o ciclo de edição, compilação, execução e depuração de um programa em linguagem de montagem é possível, potencializando assim a formação de um sólido conhecimento a respeito do funcionamento dos computadores e sua arquitetura por parte do estudante.

Apresentamos a seguir algumas de suas características de *hardware* e *software*.

C.2 Hardware

O *hardware* Raspberry Pi evoluiu através de várias versões que apresentam variações na capacidade de memória e no suporte a dispositivos e periféricos. Em nosso estudo utilizamos a versão 3 Modelo B do Raspberry Pi, que conta com a seguinte configuração:

- Processador ARM Cortex-A53 de 64 bits com 4 núcleos e cache compartilhado L2 de 512, do fabricante Broadcom, modelo BCM2837;
- Memória RAM de 1 Gbyte;
- Interface para Rede sem Fio e Bluetooth 4.1 (BLE) integrados na placa;
- 4 interfaces USB 2.0;
- Interface GPIO de 40 pinos;
- Saída de áudio externa e porta de vídeo composto;
- Saída HDMI padrão normal;
- Porta CSI para conexão de câmera Raspberry Pi;
- Porta DSI para conexão de tela de toque Raspberry Pi;
- Porta micro SD para carregar o sistema operacional e armazenamento persistente;
- Porta micro USB para fonte de alimentação.

O Raspberry Pi 3, com um processador quad-core Cortex-A53, é descrito como tendo 10 vezes o desempenho de um Raspberry Pi 1, assim como é aproximadamente 80% mais rápido do que o Raspberry Pi 2 em tarefas paralelizadas. [20]

C.3 Sistema Operacional

Diversos sistemas operacionais estão disponíveis para utilização com o Raspberry Pi. Entre eles destacamos o Raspbian, uma variante do Debian, que é a versão suportada oficialmente pela Fundação Raspberry; Ubuntu Mate e Windows 10 IoT Core. Há outros sistemas operacionais menos conhecidos, como aqueles com a função de Central de Mídia, como o OSMC (Open Source Media Centre) e LibreELEC (Kobi Media Centre) e ainda o distinto RISC OS, desenvolvido pela mesma equipe que desenvolveu o processador ARM, e que não é baseado em nenhuma distribuição Unix e roda com apenas um único usuário.

O sistema operacional utilizado para o durante os nossos testes foi o Raspbian. O Raspbian é uma versão de Linux feita especialmente para o Raspberry Pi. Ele vem com diversos pacotes como ambiente gráfico de trabalho PIXEL, a suíte de escritório LibreOffice, o navegador Chromium, cliente de correio eletrônicos e programas para o ensino de programação para crianças e adultos, como Heck e jogos como o Minecraft. É com certeza a escolha mais difundida de sistema operacional para o Raspberry Pi. Adicionalmente, o Raspbian oferece uma versão do compilador Free Pascal e do ambiente de desenvolvimento Lazarus, que foram utilizados para o transporte do simulador SimuS para o Raspberry Pi.

C.4 Pinos do GPIO

Uma característica importante do Raspberry Pi é o conector de pinos GPIO (entrada/saída de uso geral) que existe na sua placa. Esses pinos são uma interface física entre o Pi e o mundo exterior.

Os modelos originais A e B do Raspberry Pi possuíam um conector com 26 pinos, o que mais tarde foi expandido para 40 pinos nos modelos A+ e B+. Do total de pinos do conector, dezessete (originais) ou vinte e oito (A+ e B+) são pinos que podem ser configurados como entrada e saída digital, e os demais são pinos de alimentação ou terra.

Entre os pinos digitais, existem alguns que podem ser alternativamente configurados para outras funções: dois para saídas de modulação por largura de pulso (PWM); dois para comunicação serial assíncrona; dois para a interface I2C; e finalmente cinco pinos para a interface SPI (sendo um destes pinos para seleção de um escravo).

Uma limitação do Raspberry Pi é não possuir entradas que realizem conversão analógico digital. A Figura C.1 mostra um esquema com a pinagem do conector GPIO de 40 pinos.

Figura C.1: Pinos do GPIO

Os pinos de saída do conector GPIO podem gerar sinais com 3,3 volts e até 16 mA de capacidade de corrente individualmente, não excedendo 50 mA no conjunto, com configurações variadas, dependendo do modelo. Notem que esses pinos não tem proteção contra curto-circuito, necessitando de um cuidado adicional na sua manipulação para não serem danificados.

Como podemos observar na Figura Figura C.1 há dois tipos de numeração: a numeração física sequencial, iniciando no pino 1 até o pino 40, e a numeração do controlador GPIO, que se inicia em 0 e vai até 27. Isso é fator, infelizmente, de alguma confusão na utilização desses pinos, principalmente para os usuários menos experientes.

Mas antes da utilização desses pinos é necessário informar se os mesmos são entradas, saídas ou utilizados para algum protocolo de comunicação como I2C ou SPI. Isso pode ser feito de diversas maneiras, de acordo com o ambiente ou a linguagem de programação utilizada.

Esses pinos podem ser utilizados para interagir de diversos modos com o mundo real. As entradas não precisam vir apenas de uma chave ou interruptor físico; elas podem ser ligadas a um sensor ou conectadas a um sinal vindo de outro computador ou dispositivo. As saídas também podem ser utilizadas de diversas maneiras, com ligar um LED, acionar um motor ou enviar dados para outro dispositivo.

A conectividade e o controle de dispositivos físicos através da internet é

uma funcionalidade poderosa e de muito interessante para capturar o interesse e a atenção dos estudantes. Se o Raspberry Pi estiver conectado em uma rede, ele pode controlar dispositivos que estão conectados a ele a partir de qualquer lugar no mundo, em um conceito conhecido com Internet das Coisas.

Bibliografia

BORGES, E. V. C. L. ; e. a. SEAC: um simulador online para ensino de arquitetura de computadores. Em: *Anais do Workshop sobre Educação em Arquitetura de Computadores*. Petrópolis: Sociedade Brasileira de Computação, 2012, p. 34–38.

BORGES José Antonio S. ; SILVA, G. P. NeanderWin - Um Simulador Didático para uma Arquitetura do Tipo Acumulador. Em: *Anais do Workshop sobre Educação em Arquitetura de Computadores*. Ouro Preto: Sociedade Brasileira de Computação, 2006, p. 1–10.

BORGES José Antonio S.; Silva, G. P. *O Simulador Neander-X para o Ensino de Arquitetura de Computadores*. 1ª ed. Disponível em http://www.amazon.comm Acessi en Agosto de 2016. Rio de Janeiro, Brazil: Gabriel P. Silva, 2016.

FOUNDATION, R. P. *Raspberry Pi*. Disponível em: https://www.raspberrypi.org/. Acesso em Agosto, 2017. 2017.

HENDERSON, G. *Wiring Pi*. Disponível em: http://wiringpi.com. Acesso em Agosto, 2017. 2017.

HENNESSY J. L.; PATTERSON, D. *Arquitetura de Computadores: Uma Abordagem Quantitativa*. 3ª ed. Rio de Janeiro: Campus, 2003.

LAZARUS; TEAM, F. P. *Lazarus*. Disponível em: https://www.lazarus-ide.org/. Acesso em Agosto, 2017. 2017.

MOREIRA A. A. ; MARTINS, C. A. P. S. R2DSim: simulador didático do RISC reconfigurável. Em: *Anais do Workshop sobre Educação em Arquitetura de Computadores*. São Paulo: Sociedade Brasileira de Computação, 2009, p. 9–14.

SANDERSON P.; VollmarSridhar, Z. K. *MARS Simulator*. Disponível em:

http://courses.missouristate.edu/KenVollmar/mars/download.htm Acesso em Agosto, 2016. 2016.

SCOTT, M. *WinMips64*. Sep., 2006. Disponível em: http://indigo.ie/~mscott/. Acesso em Janeiro, 2016. 2006.

SILVA Gabriel P.; BORGES, J. A. S. SimuS - Um Simulador Para o Ensino de Arquitetura de Computadores. *International Journal of Computer Architecture Education (IJCAE)*, v. 5, n. 1, p. 7–12, dez. 2016. ISSN: 2316-9915. URL: http://www2.sbc.org.br/ceacpad/ijcae/v5_n1_dec_2016/IJCAE_v5_

SRIDHAR, Z. *GNUSim8085, versão 1.3.7*. Sep., 2006. Disponível em: https://launchpad.net/gnusim8085. Acesso em Agosto, 2016. 2016.

VERONA A. B.; MARTINI, J. A. SIMAEAC: Um simulador acadêmico para ensino de arquitetura de computadores. Em: *ENINED - Encontro Nacional de Informática e Educação*. Cascavel: UNIOESTE, 2009, p. 424–432.

WEBER, R. F. *Fundamentos de Arquitetura de Computadores*. 4ª ed. Porto Alegre: Bookman, 2012.

YURCIK, W.; GEHRINGER, E. F. A Survey of Web Resources for Teaching Computer Architecture. Em: *Proceedings of the 2002 Workshop on Computer Architecture Education: Held in Conjunction with the 29th International Symposium on Computer Architecture*. Anchorage, Alaska: ACM, 2002.

ZILLER, R. *Microprocessadores: Conceitos Importantes*. 2ª ed. Florianopolis, SC: EL - UFSC, 2000.