Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Um sistema computacional completo sobre uma máquina de instrução única implementada em FPGA

Alexandre Silva Dantas
Matheus Costa de Sousa Carvalho Pimenta

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Orientador
Prof. Dr. Marcus Vinicius Lamar

Coorientador
Prof. Dr. Diego de Freitas Aranha

Brasília
2014

Universidade de Brasília — UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Bacharelado em Ciência da Computação

Coordenador: Prof. Dr. Coordenador

Banca examinadora composta por:

Prof. Dr. Marcus Vinicius Lamar (Orientador) — CIC/UnB
Prof. Dr. Professor I — CIC/UnB
Prof. Dr. Professor II — CIC/UnB

CIP — Catalogação Internacional na Publicação

Dantas, Alexandre Silva.

Um sistema computacional completo sobre uma máquina de instrução única implementada em FPGA / Alexandre Silva Dantas, Matheus Costa de Sousa Carvalho Pimenta. Brasília : UnB, 2014.
55 p. : il. ; 29,5 cm.

Monografia (Graduação) — Universidade de Brasília, Brasília, 2014.

1. palvrachave1, 2. palvrachave2, 3. palvrachave3

CDU 004.4

Endereço: Universidade de Brasília
Campus Universitário Darcy Ribeiro — Asa Norte
CEP 70910-900
Brasília–DF — Brasil

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Um sistema computacional completo sobre uma máquina de instrução única implementada em FPGA

Alexandre Silva Dantas
Matheus Costa de Sousa Carvalho Pimenta

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Prof. Dr. Marcus Vinicius Lamar (Orientador)
CIC/UnB

Prof. Dr. Professor I
CIC/UnB

Prof. Dr. Professor II
CIC/UnB

Prof. Dr. Coordenador
Coordenador do Bacharelado em Ciência da Computação

Brasília, 30 de março de 2014

Dedicatória

Dedico a....**mamãe**

Agradecimentos

Agradeço a....*papai*

Abstract

A ciência...

Palavras-chave: palvrachave1, palvrachave2, palvrachave3

Abstract

The science...

Keywords: keyword1, keyword2, keyword3

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Com a necessidade humana de se comunicar à distância, a engenharia deu luz às Telecomunicações [10]. Com esta novidade, é possível tanto que pais e filhos se comuniquem estando em cidades distintas, quanto estratégias de guerra sejam elaboradas em conjunto por países de continentes diferentes. Comum em ambas as situações, é o fato de que as duas pontas da comunicação desejam privacidade. Isto é, pais e filhos não querem que seus vizinhos tomem conhecimento das mensagens que trocam. Tampouco, países aliados pretendem que suas estratégias falhem por vazamento de informação.

Para tornar possível o sigilo na troca de mensagens à distância, estudos são realizados na área que hoje chamamos de Segurança da Informação [2]. Diversas técnicas são desenvolvidas nesta área até hoje, para tentar garantir que um par de comunicação possa trocar informações sem que estas chegem ao conhecimento de adversários. Entre estas técnicas, as mais conhecidas e utilizadas nasceram da Criptografia [17].

A Criptografia estuda maneiras de criar uma versão ilegível de uma determinada mensagem, de modo que adversários com acesso ao canal inseguro pelo qual a mensagem será transmitida, por exemplo a Internet [9], não tenham acesso à informação contida na mensagem, e de modo que somente o destinatário seja capaz de reverter este processo, que chamamos de cifragem. A Criptografia estuda também maneiras de autenticar uma fonte, isto é, um destinatário que recebe uma mensagem deve poder estar seguro de que esta foi de fato enviada pelo remetente do qual este destinatário espera receber esta mensagem.

Atualmente, os sistemas criptográficos mais empregados são os sistemas assimétricos (uma ref. aqui). Nestes sistemas, cada ponta da comunicação possui um par do que chamamos de chaves criptográficas. Uma chave criptográfica pode ser, por exemplo, uma frase. Os pares de chaves criptográficas são utilizados para cifrar e decifrar mensagens através de algoritmos criptográficos. Um algoritmo de criptografia assimétrica é uma sequência de passos que utiliza uma mensagem e uma chave de um par de chaves criptográficas para produzir algo que chamamos de criptograma, uma versão ilegível da mensagem original. Para reconstruir a mensagem original, utiliza-se uma sequência de passos de volta do algoritmo criptográfico, que utiliza o criptograma gerado anteriormente e a outra chave do par de chaves criptográficas. Sistemas criptográficos assimétricos utilizam pares de chaves, para que uma das chaves de alguém que se comunica seja pública, ou seja, conhecida por todos os que se comunicam, enquanto a outra chave do par deve ser privada, ou seja, somente este alguém que se comunica conhece sua chave privada. Deste modo, é possível trocar mensagens de maneira segura e simultaneamente autêntica,

seguindo por exemplo a convenção de "assinar e colocar em um envelope"(cria-se um criptograma com a chave privada do remetente, une-se este criptograma com a mensagem original em uma única mensagem e transmite-se um criptograma da mensagem total, criado com a chave pública do destinatário. Deste modo, só o destinatário é capaz de abrir a mensagem total. Além disso, para verificar a autenticidade, basta verificar se a decifragem do criptograma interno utilizando a chave pública do remetente bate com a mensagem original).

É claro que entre os adversários interessados em obter informações sigilosas existem os mais astutos, praticantes de Criptanálise [5]. Diversas maneiras de se quebrar uma segurança são descobertas todos os dias. Uma maneira que vem sendo utilizada mais recentemente, devido ao aumento do poder computacional disponível, é a busca exaustiva por chaves [1]. É normal determinar que um sistema criptográfico é seguro se o melhor ataque conhecido não é mais eficiente do que a busca exaustiva no espaço de chaves.

Dos tipos de ataque existentes, o que é abordado neste trabalho chamamos de ataque de canal lateral [12]. Um ataque de canal lateral se baseia nas informações fornecidas pela parte física do sistema computacional utilizado para executar um algoritmo criptográfico, como por exemplo o consumo de energia em função do tempo.

Um computador funciona através de instruções. Uma instrução é um código que contém a informação de qual operação deve ser realiza pela máquina e quais dados devem ser utilizados como operandos. Historicamente, os primeiros computadores desenvolvidos são hoje chamados de computadores *CISC - Complex Instruction Set Computer*, ou Computador de Conjunto de Instruções Complexo [4]. O nome vem do fato de que os computadores oferenciam uma grande variedade de instruções, com diversas funcionalides complexas e por isso a estrutura interna da unidade central de processamento - *CPU* - era bastante irregular, ou desorganizada.

Passado um certo tempo após a invenção dos processadores digitais, um novo modelo de arquitetura foi proposto. O modelo *RISC - Reduced Instruction Set Computer*, ou Computador de Conjunto de Instruções Reduzido [16] - prega que o conjunto de instruções de um computador deve ser regular, de modo que é possível otimizar as operações mais frequentes na implementação da *CPU*.

Sabe-se que a intensidade do consumo de energia de um processador digital, em um determinado instante do tempo, depende diretamente da instrução que está sendo executada [8]. Em um computador *CISC* isto é mais evidente, dado que a irregularidade do conjunto de instruções se reflete na implementação física do processador. Em contrapartida, é de se esperar que computadores *RISC* reflitam consumos de energia por instrução mais ininteligíveis. No entanto, os consumos de energia por instrução em computadores *RISC* não são indiferenciáveis ao ponto de que um atacante experiente seja impedido de identificar um algoritmo criptográfico que está sendo executado em uma máquina deste tipo.

Mais recentemente, surgiu o modelo de computador *OISC - One Instruction Set Computer*, ou Computador de Instrução Única [15]. Computadores *OISC* possuem a vantagem de que, independente do consumo de energia em função do tempo, não é possível diferenciar quais instruções estão sendo executadas em um intervalo de tempo, porque só existe uma única instrução! A tendência do consumo de energia de uma *CPU OISC* em função do tempo é ser uma função periódica, isto é, uma função cujo valor em qualquer ponto inicial é exatamente o mesmo que o avaliado em qualquer ponto cuja distância ao

ponto inicial é um valor múltiplo de um determinado período (neste caso, um período de tempo). No entanto, por mais que hajam pequenas oscilações no consumo de energia, a dificuldade de se não poder identificar qual operação está sendo de fato executada em um determinado instante cria uma grande dificuldade para ataques de canal lateral.

Apesar de ser um modelo de computador mais seguro, computadores *OISC* não são muito atraentes, por conta do fato de que quanto mais reduzido é o conjunto de instruções de um computador, mais trabalho é colocado sobre os ombros dos programadores. O objetivo deste trabalho, no entanto, é mostrar que é possível construir um sistema computacional completo, de propósito geral, sobre uma máquina de instrução única. O sistema foi construído em um *FPGA - Field-programmable Gate Array*, ou Arranjo de Portas Programável em Campo [3] - utilizando a instrução Turing-completa *subleq - Subtract and branch if less or equal to zero*, ou subtrair e pular para outra instrução se o resultado for menor ou igual a zero [11].

O sistema computacional aqui proposto contempla todos os níveis de abstração de um sistema computacional. Indo do nível mais baixo ao mais alto, implementamos o *hardware* (incluindo *CPU* e controladores de dispositivos externos), o *software* básico (incluindo compilador, montador, ligador e sistema operacional) e *softwares* de aplicação (incluindo aplicações com algoritmos criptográficos).

Capítulo 2

Referencial Teórico

Computadores são sistemas incrivelmente complexos. Inúmeros componentes com papeis específicos necessitam de se intercomunicar para executar a mais simples das tarefas. Dessa forma, para compreender seu funcionamento, se faz o uso de camadas de abstrações.

Essas camadas exercem funções diferentes e são visíveis de acordo com seu uso — um usuário final não precisa saber programar para usar um processador de texto; da mesma forma, um programador não necessita saber da estrutura dos circuitos internos. Cada camada possui seu domínio, sendo as mais próximas do usuário final denominadas de "alto-nível" e as mais próximas dos transistores e fios, "baixo-nível". Observe a figura 2.1, especificada de acordo com Murdocca [14].

Iremos definir conceitos básicos, utilizados ao longo do trabalho. Nossa abordagem será *top-down*, significando que os conceitos serão explicados do nível mais abstrato até o mais concreto nível de *hardware*.

2.1 Redes de Computadores

CPU

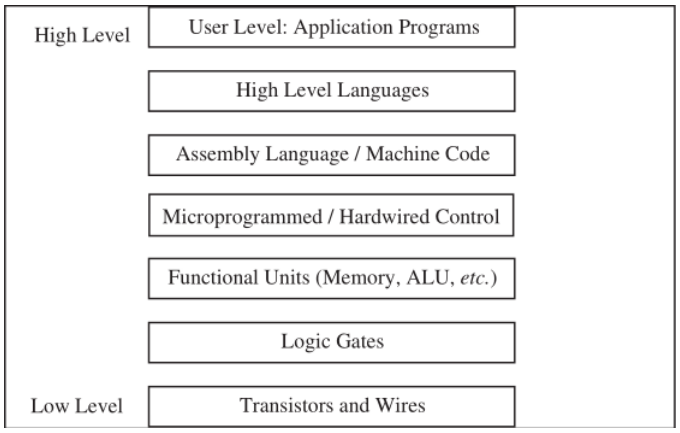

Figura 2.1: Camadas de abstração de um computador.

2.2 Teoria da Codificação e Criptografia

CPU

2.3 Tradutores

No âmbito da computação em geral, compiladores são uma das mais importantes aplicações. É impossível imaginar um sistema computacional sem alguma forma de compilação. Compiladores estão presentes em quase todas as camadas de abstração de um computador.

Porém, para compreender o conceito de compilador é necessário estudar sobre linguagens de programação e o processo de tradução.

2.3.1 Linguagens

Linguagem é um conceito abrangente. Enquanto que diferentes áreas do conhecimento definam *linguagem* por formas específicas, para nosso trabalho será suficiente uma definição geral.

Linguagem é um sistema de comunicação - um conjunto de regras e convenções que permitem a interação entre dois pares. Dialetos humanos (português, inglês, francês...) são exemplos de tais sistemas de comunicação. Cada um destes define acordos (gramáticas, sintaxe, morfologia) que permitem a troca de mensagens entre pessoas de mesmo grupo.

Dessa forma, pares que queiram interagir entre si necessitarão de trocar conhecimento através de mensagens na mesma linguagem. Caso haja comunicação por duas linguagens diferentes, ocorrerão erros de compreensão, desentendimentos e até completa inabilidade de se comunicar.

2.3.2 Linguagens de Programação

Partindo para o domínio computacional, há o conceito de linguagens de programação. Estas possuem objetivo de comunicar instruções de seres humanos para máquinas. Mais especificamente, possuem o objetivo de determinar o que computadores devem fazer - como estes são programados a agir.

Linguagens de programação são destinadas a expressar idéias concebidas por seres humanos. Dessa forma, precisam se aproximar da linguagem natural - forma de comunicação natural entre seres humanos. Devido à construção física de máquinas, computadores precisam de instruções em formato bastante diferente do oferecido pela linguagem natural - quanto mais de linguagens de programação.

2.3.3 Linguagem de Máquina

É nesse conceito que se define a linguagem de máquina, também conhecida como código de máquina. São conjuntos de instruções (ou ordens) que determinam como a máquina irá se comportar. Controla-se um computador ao agrupar várias dessas instruções em ordens específicas.

Linguagem de máquina se diferencia das linguagens de programação no sentido de que não são facilmente compreendidas por seres humanos. Enquanto que linguagens de programação se aproximam da linguagem natural, linguagem de máquina é definida pela construção física dos componentes de cada máquina.

Observaremos mais detalhes sobre linguagem de maquina na seçao 2.4.1.

2.3.4 Linguagens de Alto e Baixo nível

Com linguagens em pontos tão extremos, como alto e baixo nível, se torna impossível a comunicação entre pares que as utilizem. Necessita-se de um meio de conversão entre estas.

2.3.5 Tradução

2.3.6 Compiladores

Compiladores são programas que transformam código-fonte escrito em uma linguagem de programação em outra. Eles fazem o processo de tradução, em geral de linguagens de alto nível para linguagens de baixo-nível.

2.4 Software Básico

Na seção anterior, descrevemos os conceitos básicos necessários para compreender como se dá a criação de um programa de computador, que pode ser por exemplo um *software* de aplicação. Geralmente classificamos os programas utilizados por usuários finais como *software* de aplicação.

Programas tradutores como os descritos na seção anterior são classificados como *software* básico, ou *software* de sistema. Dizemos que *software* básico faz parte de um conjunto de *softwares*, ou de um sistema, capaz de gerenciar a utilização do *hardware* de modo a fornecer uma plataforma para *softwares* de aplicação (uma ref. aqui).

Entre os tipos de *software* classificados como *software* básico, podemos incluir tradutores (como compiladores e montadores), ligadores, sistemas operacionais (que incluem carregadores e *drivers* de dispositivos) e sistemas embarcados (uma ref. aqui). Discutiremos sistemas operacionais na próxima seção.

2.4.1 Linguagem de Máquina

Anteriormente vimos que uma linguagem de montagem, ou linguagem *assembly*, é formada por símbolos mnemônicos, ou palavras-chave, que identificam as instruções que um processador é capaz de executar.

A etapa seguinte à compilação é a montagem. Nesta etapa, é necessário traduzir os mnemônicos *assembly* para sequências de *bits*. Um *bit* é um dígito que pode assumir apenas o valor 0 (zero) ou o valor 1 (um). Durante a execução de um programa, estas sequências de *bits* serão diretamente interpretadas por uma *CPU*.

Costuma-se dizer que a etapa de montagem é onde está localizada a interface *software-hardware*. Uma *ISA - Instruction Set Architecture*, ou Arquitetura de Conjunto de

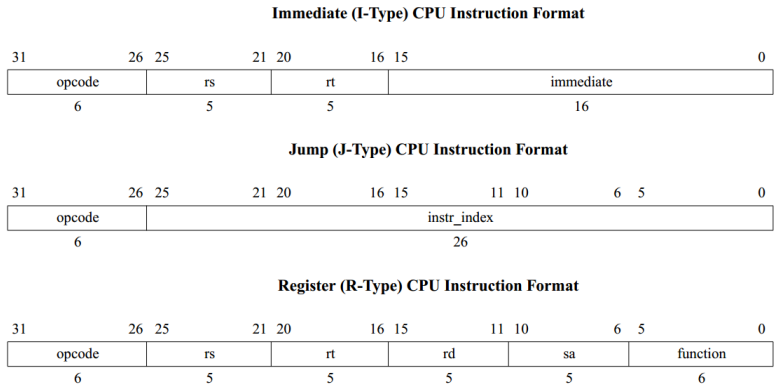

Figura 2.2: Os três formatos principais da *ISA MIPS32*.

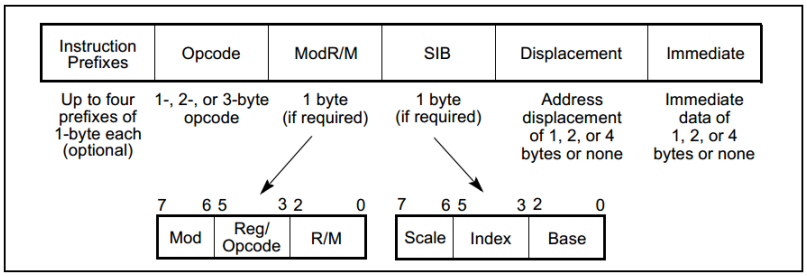

Figura 2.3: Formato geral de uma instrução da *ISA IA-32*.

Instruções - é uma especificação das instruções que uma implementação de processador digital deve fornecer. Esta especificação, entre outras coisas, estabelece os formatos que as sequências de *bits* geradas por montadores devem seguir. Este é o principal conhecimento necessário para construir um montador.

A figura 2.2 mostra os três formatos principais de instruções da arquitetura *RISC MIPS32* (uma ref. aqui). Grandes quantidades de instruções se encaixam em cada um dos três formatos. Nesta arquitetura, toda e qualquer instrução ocupa uma palavra de 32 *bits*.

A figura 2.3 mostra o formato geral para uma instrução qualquer da arquitetura *CISC IA-32* (uma ref. aqui), criada pela Intel. Nesta arquitetura, o tamanho das instruções vai de 1 até 17 *bytes* (um *byte* são exatamente 8 *bits*).

Com o conhecimento de como cada mnemônico deve ser convertido para código de máquina, podemos construir um montador para uma determinada arquitetura de processadores. À seguir descrevemos os algoritmos clássicos de montagem.

2.4.2 Montadores

Montadores são programas que traduzem um módulo de compilação em linguagem *assembly* para uma versão em linguagem de máquina. O armazenamento desta versão é feito em um arquivo objeto. Como veremos em uma subseção adiante, um arquivo objeto

é uma versão binária de um módulo de compilação, que geralmente fica a um passo de poder ser carregada para execução.

O problema da montagem pode ser colocado da seguinte forma: um módulo de compilação em linguagem *assembly* contém dados e instruções mnemônicas que devem ser convertidas para binário seguindo a ordem em que aparecem. Para termos a noção de localização de um dado ou instrução, temos o conceito de endereço de memória.

Antes de definirmos o conceito de endereço de memória, precisamos definir o conceito de palavra. Uma palavra é um agrupamento de uma quantidade fixa de *bits*. Uma *ISA* determina os diferentes tamanhos de palavras que podem ser utilizados para escrever um programa. Geralmente, o menor tamanho de palavra suportado é o *byte*, enquanto os outros tamanhos costumam ser múltiplos em potência de 2 de um *byte*.

Aqui iremos nos referir a um endereço de memória como um número cuja unidade é o menor tamanho de palavra definido por uma arquitetura. Para referenciar endereços, linguagens de montagem costumam utilizar o que chamamos de símbolo, ou rótulo. Um símbolo pode ser uma palavra ou uma frase de linguagem natural.

Para realizar a conversão de um arquivo *assembly* para binário, é necessário que um algoritmo de montagem realize passagens no arquivo de entrada, coletando informações de montagem. Com informação suficiente, o montador é capaz de colocar os dados em determinados endereços, colocar as instruções na ordem em que aparecem em outros endereços e substituir ocorrências de rótulos por seus respectivos endereços.

A próxima subseção descreve o algoritmo de duas passagens, o mais trivial.

2.4.3 Montadores de duas passagens

Na primeira passagem pelo código *assembly*, um algoritmo de duas passagem cria uma tabela de símbolos.

A tabela de símbolos é uma estrutura indexada por símbolos (ou rótulos). Para cada símbolo que indexar a tabela, temos uma entrada com o endereço ao qual o símbolo se refere e algumas informações adicionais. É criada uma entrada nesta tabela para cada rótulo que for encontrado no código-fonte. Rótulos podem ser criados tanto para dados, quanto para instruções.

Em um programa *assembly*, símbolos podem estar presentes tanto em definições de símbilos, ou seja, em locais onde o endereço de um símbolo é estabelecido, quanto podem estar contidos em instruções que referenciam endereços através de símbolos. Ao encontrar a definição de um símbolo, o montador pode gerar um erro, ou aviso, caso este símbolo já tenha sido definido anteriormente. Se a definição de um símbolo declara um dado, o montador pode colocar esta informação na entrada da tabela de símbolos.

Um montador tem a liberdade de implementar funcionalidades que facilitam a vida de um programador *assembly*. Uma funcionalidade comumente implementada é a disponibilização de diretivas de pré-processamento. Diretivas são instruções ao próprio montador, ou seja, diretivas não são instruções que geram código de máquina.

Outra funcionalidade comum, é a utilização de pseudo-instruções. Pseudo-instruções utilizam mnemônicos e formatos parecidos com os das instruções concretas da *ISA*, mas são instruções que não estão de fato implementadas em *hardware*. É comum montadores fornecerem pseudo-instruções que expandem para muitas instruções concretas, que em conjunto realizam uma tarefa mais complexa.

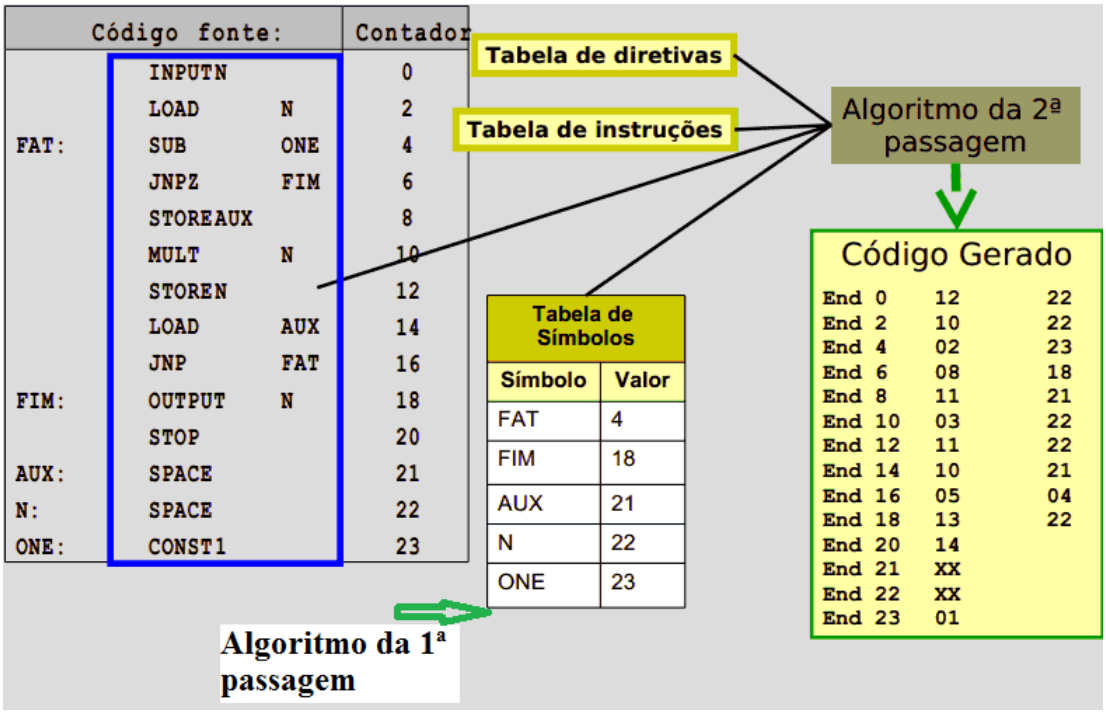

Figura 2.4: Ilustração do algoritmo de duas passagens sobre um *assembly* hipotético.

Ainda na primeira passagem, o montador pode avaliar mnemônicos de instruções. Ao avaliar uma instrução, o montador pode gerar um erro caso não identifique o mnemônico da operação como válido. Um mnemônico válido pode ser o de uma instrução concreta, o de uma pseudo-instrução, ou pode ser uma diretiva de pré-processamento.

Com a tabela de símbolos pronta, o montador pode realizar a segunda passagem. Esta é a etapa em que o montador de fato gera código de máquina.

O montador examina cada instrução à procura de símbolos. Símbolos que não estejam presentes na tabela de símbolos geram erro de montagem. Para instruções com símbolos definidos, o montador gera o código de máquina correspondente, substituindo cada símbolo por seu respectivo endereço.

Ao final da segunda passagem, o montador está pronto para gerar o arquivo objeto. Nesta etapa, coloca-se o código de máquina gerado junto com um espaço alocado para dados. Ao gerar de fato o arquivo de saída, o montador coloca nele informação de relocação, uma tabela de símbolos exportados e uma tabela de referências externas.

A figura 2.4 ilustra a execução de um algoritmo de duas passagens sobre um código *assembly* hipotético (uma ref. aqui).

Voltaremos agora nossa atenção ao algoritmo de passagem única, desenvolvido para ser mais eficiente do que o algoritmo que acabamos de descrever. Iremos deixar a discussão sobre arquivos objeto para a subseção depois da próxima.

2.4.4 Montadores de passagem única

O algoritmo de passagem única pode ser considerado uma evolução do algoritmo descrito acima.

Um montador de passagem única constrói a tabela de símbolos ao passo que gera código de máquina. Para trabalhar desta forma, são necessárias listas de referências pendentes.

Para cada símbolo encontrado em uma instrução que não estiver presente na tabela de símbolos, o algoritmo cria uma entrada na tabela, marca o símbolo como indefinido e cria uma lista de referências àquele símbolo.

Ao encontrar a definição de um símbolo, montadores de passagem única ainda geram erro, caso o símbolo já tenha sido definido anteriormente. No entanto, se o símbolo é encontrado na tabela, mas está marcado como indefinido, o algoritmo marca este símbolo como definido, estabelece seu endereço e itera sobre a lista de referências àquele símbolo atualizando os campos com o endereço válido. A implementação de um montador pode escolher atualizar as referências ao final da passagem, ao invés de realizar esta tarefa imediatamente.

Após a passagem ser realizada, se ainda existirem símbolos indefinidos o montador gera erro. O restante do processo é o mesmo que o descrito no algoritmo de duas passagens.

2.4.5 Arquivos objeto

Chamamos a saída de um montador de arquivo objeto. Juntamente com o código de máquina montado, montadores geram informações necessárias para etapas seguintes, até que o programa possa de fato ser executado em uma *CPU*.

Uma das estruturas colocadas em um arquivo objeto é uma lista de relocação. Anteriormente definimos o que são endereços. Agora vamos definir o que são endereços absolutos e o que são endereços relativos.

Uma lista de relocação armazena endereços absolutos. Endereços absolutos são endereços calculados utilizando um endereço base. Relocação é o processo de somar um endereço base à um endereço absoluto. Pode-se considerar que, durante a montagem, o endereço base é o endereço zero. Na subseção que vem à seguir, veremos mais sobre relocação.

Em contrapartida, endereços relativos são endereços que não precisam de relocação, pois são endereços cujo endereço base é a própria posição da instrução. Veremos mais sobre endereços relativos na próxima subseção.

Outra estrutura gerada por um montador é uma tabela de símbolos exportados. Ao definir um símbolo em um módulo de compilação, podemos querer referenciá-lo de outro módulo de compilação. Para que isto seja possível, é necessário declarar quais símbolos de um módulo serão exportados. A subseção à seguir trata sobre este assunto.

A terceira e última estrutura presente na saída de um montador é uma tabela de referências externas. Esta é a recíproca da tabela de símbolos exportados. Ao mesmo tempo que um módulo de compilação pode fornecer símbolos a serem referenciados em outros módulos, este mesmo módulo obviamente deve conter informações que dizem aonde símbolos externos foram utilizados. A tabela de referências externas contém listas de referências para cada símbolo importado.

Após criar uma seção inicial no arquivo binário que contém estas informações, seção que geralmente é denominada como cabeçalho, o montador coloca na saída o código de máquina montado. A etapa seguinte à montagem é ligação.

2.4.6 Ligadores

Após arquivos objeto de um ou vários módulos de compilação serem gerados por um montador, é preciso combiná-los em um único arquivo que pode ser carregado para a memória por um programa carregador do sistema operacional.

Ligadores utilizam as três informações colocadas nos cabeçalhos de arquivos objeto para gerar arquivos executáveis, ou bibliotecas de código.

Arquivos executáveis, ou módulos de carga, são arquivos que estão quase prontos para serem carregados na memória de um computador e serem executados. A etapa final que pode estar faltando é a relocação de endereços absolutos, caso o programa utilize instruções com este tipo de endereçamento. Discutiremos o processo de carga na próxima subseção.

Bibliotecas de código são arquivos com código de máquina que reúnem diversas funcionalidades e podem ser aproveitadas por outros programas. Bibliotecas dividem-se em bibliotecas estáticas e bibliotecas dinâmicas.

Bibliotecas estáticas são a combinação de vários arquivos objeto que utilizam instruções de máquina com endereços absolutos. Desta forma, para um programa aproveitar uma biblioteca estática, basta que o ligador considere esta biblioteca como um simples arquivo objeto. Portanto, é claro que o ligador também é responsável por gerar informações de relocação, símbolos e referências externas.

Bibliotecas dinâmicas, também consideradas módulos de carga, são a combinação de vários arquivos objeto que utilizam instruções de máquina com endereços relativos. A vantagem de se utilizar endereços relativos é que o módulo de carga pode ser carregado em qualquer posição da memória, sem que relocação seja necessária. Outra vantagem de utilizar bibliotecas dinâmicas, é a produção de arquivos executáveis mais leves. Em alguns sistemas operacionais, apenas uma cópia de uma biblioteca dinâmica na memória é necessária para a execução simultânea de diversos programas.

Neste ponto, deve estar fácil de concluir que ligadores aceitam como entrada não apenas arquivos objeto, como aceitam também bibliotecas.

O processo de ligação é extremamente simples. A primeira tarefa a ser realizada na ligação é o carregamento dos arquivos de entrada em uma determinada ordem. A cada arquivo carregado, o algoritmo de ligação associa um endereço base de posicionamento. Este endereço base é utilizado para relocar todas as instruções de endereçamento absoluto daquele arquivo.

Após carregar, colocar juntos os arquivos de entrada e efetuar as relocações, o algoritmo de ligação une as tabelas de símbolos exportados de cada arquivo em uma grade tabela global de símbolos.

Com a tabela global de símbolos, é possível passar pelas referências externas de cada arquivo, preenchendo cada campo com o respectivo endereço.

Na criação de uma biblioteca estática, o ligador deve gerar uma nova lista de relocação, utilizar a tabela global de símbolos como a nova tabela de símbolos exportados e pode criar uma nova tabela de referências pendentes, caso nem todas as referências tenham sido resolvidas durante aquela ligação. Na criação de uma biblioteca dinâmica, informação de relocação não é gerada.

Ao criar um arquivo executável, o ligador pode gerar informação de relocação e uma tabela de referências externas (para o caso de ligação com bibliotecas dinâmicas). Além disso, o ligador espera que um dos arquivos de entrada exporte um símbolo especial que

possa ser estabelecido com o ponto de entrada do programa. Esta é uma regra imposta por formatos de arquivos executáveis, adotados por sistemas operacionais.

Na prática, um ligador quase sempre gera uma tabela de referências externas em arquivos executáveis, pois diversos sistemas operacionais estabelecem que a própria biblioteca dinâmica que implementa operações de ligação deve ser ligada com um programa, obviamente para que seja possível realizar ligações com bibliotecas dinâmicas. A utilização de bibliotecas dinâmicas implica na necessidade de realizar relocação em tempo de execução, pois um sistema operacional pode determinar que uma biblioteca só deve ser carregada quando for de fato chamada (uma ref. aqui).

Com os dados de cabeçalho prontos, o ligador finalmente gera um arquivo binário com código de máquina. À seguir, descrevemos o processo de carregamento de módulos de carga para execução.

2.4.7 Carregadores

CPU

2.5 Sistemas Operacionais

CPU

2.6 Organização e Arquitetura de Computadores

CPU

2.6.1 Arquitetura de Processadores Digitais

CPU

2.6.2 Controladores de Dispositivos Externos

CPU

2.7 Circuitos Digitais

CPU

2.7.1 Lógica Booleana e Circuitos Combinacionais

CPU

2.7.2 Circuitos Sequenciais

CPU

2.7.3 Arranjo de Portas Programável em Campo

CPU

2.7.4 Linguagem de Descrição de Hardware

CPU

Capítulo 3

Revisão Bibliográfica

Capítulo 4

Metodologia

Capítulo 5

Protótipo do Quinto Capítulo

5.1 Conceitos Básicos

(Aqui estão coisas que não consegui colocar em outros lugares)

Uma definição muito importante para o programador de sistema é a Arquitetura do Conjunto de Instruções (*Instruction Set Arquitecture*) — de agora em diante referida apenas como arquitetura, ou *ISA*. Hennessy a define como "o limite entre *software* e *hardware*" [7].

A *ISA* descreve vários componentes essenciais para a criação de programas de sistema. Seu *design* define a memória interna do processador, o endereçamento de memória interna e externa, quais instruções/operações são suportadas, tipos e tamanhos de operandos, dentre muitos outros [16].

Existem tipos diferentes de *ISA*, sendo comuns as arquiteturas *RISC* e *CISC*.

Arquiteturas *RISC* (*Reduced Instruction Set Computer*) possuem uma quantidade reduzida de instruções. Em geral são instruções simples e rápidas que têm de ser combinadas para ações mais complexas. Já arquiteturas *CISC* (*Complex Instruction Set Computer*) provêem uma quantidade maior de instruções, que nativamente executam ações mais complicadas e abrangentes.

Instruções da arquitetura *RISC* são mais simples; elas partem da filosofia de otimizar os casos frequentes, visando tornar o comportamento geral mais rápido. Murdocca argumenta que um conjunto de instruções mais simples resulta numa central de processamento simples e menor, liberando espaço no processador para outros componentes, como registradores [14].

Porém Mostafa argumenta que isso traz a desvantagem de que uma grande quantidade de instruções são necessárias para executar uma função simples [13], possivelmente reduzindo o desempenho geral. Por fim, isso tambem causa um problema cognitivo, já que programas *RISC* tendem a ser mais verbosos e depositarem a complexidade do programa nos ombros do programador.

Além de ambas as *ISA*s citadas acima, existe a arquitetura *OISC* (*One Instruction Set Computer*). Ela define computadores com apenas uma única instrução. De acordo com Gilreath, *OISC* é como um *CISC* em um nível mais alto de abstração, já que precisa-se combinar essa única instrução de diversas formas para sintetizar o que seriam as instruções mais complexas [6].

Pela própria definição, arquiteturas *OISC* possuem as desvantagens de *RISC* em escala muito maior. Qualquer função simples necessitará de várias combinações da única instrução, dificultando tanto a velocidade quanto compreensão do programa final.

Entretanto, existem vantagens na previsibilidade de máquinas *OISC*.

Em máquinas não-*OISC*, existe um conjunto bem-definido de instruções que podem ser executadas. Cada uma exige demandas específicas do processador, resultando em gastos de energia possivelmente diferentes.

Dessa forma, ao se monitorar o gasto de energia por um período suficiente de tempo, pode-se observar os padrões de gasto de energia do processador. Então, um observador externo poderá deduzir quais instruções foram executadas na máquina sem necessariamente ter acesso à mesma.

Considerando que arquiteturas *OISC* possuem apenas uma instrução, cuja demanda ao processador é única, assume-se que o gasto de energia será constante. Logo, não seria possível determinar quais ações essa máquina executou independentemente da quantidade de tempo de monitoramento.

O ponto abordado nesse trabalho é exatamente esse — determinar se o gasto de energia em função do tempo é constante numa máquina *OISC*. Se for o caso, pode-se determinar aplicações interessantes para esse tipo de computador na área de segurança de informação e criptografia.

Primeiramente, devemos determinar que instrução será usada na nossa máquina *OISC*.

5.2 Linguagem de Montagem Subleq

Existem várias máquinas de arquitetura *OISC*. Uma delas é a máquina que possui apenas a instrução *SUBLEQ* (*Subtract and Branch on Less or Equal* [11].

Referências

[1] Giovanni Agosta, Alessandro Barenghi, and Gerardo Pelosi. Exploiting bit-level parallelism in gpgpus: A case study on keeloq exhaustive key search attack. In *ARCS Workshops (ARCS), 2012*, pages 1–7. IEEE, 2012. 2

[2] Schultz Eugene Siegel Carol A. Bernstein Terry, Bhimani Anish B. *Internet Security for Business*. Wiley, 1996. 1

[3] Alexander Brant and Guy GF Lemieux. Zuma: an open fpga overlay architecture. In *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*, pages 93–96. IEEE, 2012. 3

[4] You-Sung Chang, Bong-Il Park, In-Cheol Park, and Chong-Min Kyung. Customization of a cisc processor core for low-power applications. In *Computer Design, 1999.(ICCD'99) International Conference on*, pages 152–157. IEEE, 1999. 2

[5] Swenson Christopher. *Modern Cryptanalysis: Techniques for Advanced Code Breaking*. Wiley, 2008. 2

[6] Laplante Phillip A. Gilreath William F. *Computer Architecture: A Minimalist Perspective: Dynamics and Sustainability*. Springer, 2003. 16

[7] Patterson David Hennessy John L. *Computer Architecture: A Quantitative Approach*. Morgan Kaufman Publishers, 1989. 16

[8] Cheng-Ta Hsieh, Lung-sheng Chen, and Massoud Pedram. Microprocessor power analysis by labeled simulation. In *Proceedings of the conference on Design, automation and test in Europe*, pages 182–189. IEEE Press, 2001. 2

[9] Ross Keith W. Kurose James F. *Computer Networking: A Top-Down Approach*. Pearson, 2012. 1

[10] Ding Zhi Lathi B. P. *Modern Digital and Analog Communication Systems*. Oxford University Press, 2009. 1

[11] O. Mazonka and A. Kolodin. A Simple Multi-Processor Computer Based on Subleq. *ArXiv e-prints*, June 2011. 3, 17

[12] Mohamed Saied Emam Mohamed, Stanislav Bulygin, Michael Zohner, Annelie Heuser, Michael Walter, and Johannes Buchmann. Improved algebraic side-channel attack on aes. *Journal of Cryptographic Engineering*, 3(3):139–156, 2013. 2

[13] Hesham El-Rewini Mostafa Abd-El-Barr. *Fundamentals of Computer Organization and Architecture*. Wiley, December 2004. 16

[14] Heuring Vincent P. Murdocca Miles. *Principles of Computer Architecture*. Prentice Hall, 1999. 4, 16

[15] Jia Jan Ong, L-M Ang, and KP Seng. Implementation of (15, 9) reed solomon minimal instruction set computing on fpga using handel-c. In *Computer Applications and Industrial Electronics (ICCAIE), 2010 International Conference on*, pages 356–361. IEEE, 2010. 2

[16] Hennessy John L. Patterson David. *Computer Organization and Design: the Hardware/Software Interface*. The Morgan Kaufmann Series in Computer Architecture and Design. Morgan Kaufmann Publishers, 2011. 2, 16

[17] Stinson Douglas R. *Cryptography: Theory and Practice*. Chapman and Hall/CRC, 2005. 1