Poli Assembler

Igor Pontes Tresolavy
12553646
tresolavy@usp.br

Thiago Antici Rodrigues de Souza
12551411
thiago_souza@usp.br

Murillo Freitas da Silva
12554741
murillo@usp.br

***Abstract*— Nesse relatório, trataremos de explicar o que são os assemblers, pra que eles servem, e porquê decidimos desenvolver um para a disciplina de Laboratório de Processadores. Ao final, explicitamos o funcionamento do nosso assembler, com exemplos.**

I. Introdução e Contextualização

A. O que são Assemblers?

As linguagens de montagem (assembly languages) são linguagens de programação de baixo nível nas quais há uma correspondência de um para um entre as instruções escritas (usando mnemônicos e símbolos) e o código de máquina gerado. Esse processo de tradução é realizado por um assembler, que converte a linguagem assembly, uma forma legível por humanos das instruções de máquina, em código de máquina, o código binário que o processador de um computador pode executar diretamente.

A linguagem assembly está intimamente ligada à arquitetura do computador, refletindo a organização da memória, o conjunto de instruções e outras características do hardware. Por isso, um assembler deve ser especificamente projetado para a arquitetura que está sendo alvo. Por exemplo, um assembler para um processador ARM é diferente de um para um processador x86, porque as instruções de hardware subjacentes são diferentes.

No início da computação, quando os recursos de hardware eram limitados, a linguagem assembly e os assemblers eram essenciais, permitindo um controle fino sobre como os recursos eram utilizados para alcançar desempenho e eficiência do código. A primeira linguagem de montagem conhecida foi desenvolvida por Kathleen Booth em 1947, para o computador APEXC, na Universidade de Londres. Desde então, assemblers têm sido usados na programação de sistemas, em software embarcado e em situações onde o controle fino do hardware é necessário, como em sistemas operacionais.

Além de converter instruções simbólicas em instruções de máquina correspondentes, os assemblers também lidam com várias tarefas, como resolver endereços simbólicos, gerenciar diferentes formatos de instrução e manipular diretivas do assembler. As diretivas são instruções especiais fornecidas ao assembler para controlar o processo de montagem, como especificar o endereço inicial do programa ou incluir bibliotecas externas.

Os assemblers podem ser categorizados em diferentes tipos, dependendo de como processam o código-fonte. Um assembler de duas passagens, por exemplo, lê o arquivo fonte inteiro duas vezes. Na primeira passagem, ele coleta todas as definições de rótulos e constrói uma tabela de símbolos, que mapeia nomes simbólicos para endereços de memória. Na segunda passagem, ele traduz as instruções para código de máquina, usando a tabela de símbolos para resolver endereços. Já um assembler de uma passagem lê o arquivo fonte apenas uma vez e deve lidar com símbolos indefinidos de forma imediata, o que o torna mais rápido, mas menos flexível, pois frequentemente produz arquivos objeto absolutos, vinculados a endereços de memória específicos, em vez de arquivos relocáveis, que podem ser carregados em diferentes localizações de memória.

Embora o uso de assemblers tenha diminuído com o advento das linguagens de programação de alto nível, eles ainda são insubstituíveis em algumas áreas. Exemplos incluem drivers de dispositivos, firmware para sistemas embarcados e sistemas operacionais em geral. Além disso, os assemblers são utilizados em aplicações de desempenho crítico e, especialmente, em segurança, onde a engenharia reversa é uma atividade frequente.

Os assemblers evoluíram ao longo do tempo, desde sistemas simples de "assemble-and-go", que carregavam e executavam código diretamente, até assemblers modernos que produzem arquivos objeto relocáveis complexos, que podem ser vinculados a outros programas por um carregador. Essa evolução reflete a crescente complexidade dos sistemas computacionais e a necessidade de ferramentas mais sofisticadas para gerenciar essa complexidade.

II. Motivação

A. Porquê criar um próprio Assembler?

Desenvolver um assembler do zero é uma oportunidade de aprendizado, que proporciona uma compreensão profunda sobre como os computadores realmente funcionam. Além de entender como a linguagem de máquina é imple-

mentada na CPU, também é necessário entender profundamente os conceitos de como se monta uma linguagem de domínio, seu parseamento, análise léxica e semântica. Ao criar um assembler, interage diretamente com o processo de tradução das instruções simbólicas de uma linguagem de montagem para o código de máquina que a CPU pode executar. Esse processo não apenas desmistifica o funcionamento interno dos processadores, mas também oferece uma visão prática de como o software interage diretamente com o hardware.

Outra razão pela qual criar um assembler é o fator de diversão e desafio técnico envolvido. Para aqueles que têm paixão por programação e por entender os sistemas em um nível fundamental, construir um assembler é uma atividade incrivelmente recompensadora. É como montar um quebra-cabeça complexo, onde cada peça representa um componente fudamental do funcionamento de um computador. O processo de desenvolver uma ferramenta que pode converter símbolos legíveis por humanos em código executável é não apenas educativo, mas também extremamente legal.

Como dizia Steve Jobs: *"Everything was made up by people that were no smarter than you"*

B. Assembly na Segurança Cibernética e Engenharia Reversa

Na área de segurança cibernética, em específico, a linguagem assembly ainda é relevante. Muitas vulnerabilidades e *exploits* ocorrem em níveis baixos de abstração nos sistemas de computação. Além disso, *malwares*, no geral, são espalhados como códigos binários (sem informações de *debug*!). Como consequência, engenheiros que trabalhem nessa área devem analisar os códigos maliciosos diretos no nível de máquina. Essa tarefa é chamada de engenharia reversa.

A criação de um assembler permite um melhor entendimentos sobre os formatos de códigos de máquina gerados e como eles se relacionam com ferramentas existentes, como *linkers* e *loaders*. Desse modo, uma nova habilidade de engenharia reversa pode ser desenvolvida juntamente com o desenvolvimento do assembler.

III. Objetivos

O objetivo principal do nosso projeto foi desenvolver um assembler que pudesse receber um arquivo de texto escrito em linguagem assembly ARMv7 e convertê-lo em um arquivo no formato ELF (Executable and Linkable Format) relocável. Esse formato de arquivo é amplamente utilizado em sistemas baseados em UNIX e é importante para a interoperabilidade com outras ferramentas de desenvolvimento, como *linkers* e *loaders* da GNU, por exemplo. Garantir que o assembler pudesse gerar um arquivo ELF relocável foi um grande desafio, mas abriu a possibilidade da composição de programas, que desejávamos fazer.

O Poli Assembler foi projetado para interpretar corretamente algumas das instruções, rótulos e diretivas encontradas em um código assembly específico para a arquitetura ARM. Apesar de gostarmos da ideia de implementarmos um assembler completo, dado o limite de tempo e recursos que temos, limitamos o escopo de suporte à funcionalidades do Poli Assembler. Para que o assembler funcione corretamente, ele deve realizar as seguintes tarefas: a leitura e interpretação do arquivo de texto, a designação correta de registradores e endereços de memória, e a conversão das instruções no código de máquina correspondente. Um dos principais desafios foi garantir que as seções do arquivo ELF, como a tabela de símbolos e as seções de código, fossem estruturadas de acordo com as especificações do formato, para que pudessem ser manipuladas adequadamente pelas ferramentas de linkagem e carregamento. Não existe muita documentação moderna sobre como fazer isso, uma vez que, nos tempos modernos, os assemblers normalmente são desenvolvidos e adaptados pelas próprias empresas de processadores para arquiteturas específicas.

IV. Modelo do Problema

Um assembler, de forma geral, é uma ferramenta que converte o código escrito em linguagem assembly, que é mais legível para humanos, em código de máquina, que é executável pelo processador após modificações mínimas. O trabalho do assembler é ler o código assembly, interpretar cada instrução, e gerar o correspondente código binário que o processador pode executar.

A. Categorização de Assemblers

Existem duas categorias principais de implementação de assemblers: o **one-pass** (passagem única) e o **two-pass** (duas passagens).

1) *Assemblers One-Pass:*

Um assembler de passagem única (**one-pass assembler**) processa o código assembly em uma única leitura. Neste modelo, o assembler tenta resolver todas as instruções e rótulos à medida que os encontra. Isso significa que ele precisa determinar os endereços de memória e gerar o código de máquina durante essa primeira e única leitura do arquivo de entrada. Embora essa abordagem possa ser eficiente em termos de tempo, ela apresenta algumas limitações. A principal delas está relacionada no uso de referências a *labels* definidas posteriormente no código. Nesse caso, o assembler emite *erratas* para que o linker corrija as referências futuramente.

O assembler de uma passagem também possui uma limitação na geração de arquivos de objeto relocáveis. Assemblers de passagem única normalmente só são capazes de gerar arquivos de objeto absolutos, que possuem usos limitados, já que não permitem sua relocação.

2) *Assemblers Two-Pass:*

Por outro lado, um assembler de duas passagens (**two-pass assembler**) faz duas leituras completas do código assembly. Na primeira passagem, o assembler analisa o código para identificar todos os rótulos e símbolos, atribuindo endereços de memória sem gerar o código de máquina ainda. Durante essa primeira passagem, ele cria uma tabela de símbolos que mapeia os rótulos para seus respectivos endereços. Na segunda passagem, o assembler utiliza essa tabela de símbolos para gerar o código de máquina definitivo, substituindo os rótulos pelos endereços de memória apropriados. Essa abordagem é mais robusta, pois permite que o assembler resolva referências a rótulos que aparecem antes de sua definição no código.

Desse modo, o assembler de duas passagens permite a geração de arquivos de objeto relocáveis, que são mais flexíveis. Diante desses fatores, decidiu-se pela implementação de um assembler de duas passagens, no qual a primeira passagem exclusivamente procura os símbolos e monta a tabela de símbolos, que é usada posteriormente para montar o código.

B. Estrutura do ELF

Considera-se oportuno discutir o formato do arquivo ELF antes de discutir o projeto da solução do Poli Assembler. As seções a seguir explicitam com certo grau de detalhe as estruturas de dados dos arquivos ELF. Apesar do Poli Assembler ter sido escrito em Rust, os exemplos de código são dados em C, já que é a linguagem na qual esses conceitos têm sido abordados com maior frequência historicamente.

O Poli Assembler gera um arquivo ELF relocável como saída. Esse formato é o `ELF 32-bit LSB relocatable, ARM, EABI5 version 1 (SYSV), not stripped`. A seguir, uma explicação não exaustiva dos arquivos ELF e seus campos relevantes é apresentada.

1) *Cabeçalhos de Arquivos ELF Relocáveis (.o):*

Arquivos ELF (Executable and Linkable Format) são usados em sistemas do tipo Unix para representar programas executáveis, bibliotecas compartilhadas e arquivos de objeto relocáveis (.o). Os cabeçalhos dos arquivos ELF relocáveis fornecem informações sobre a organização e o conteúdo do arquivo.

Cabeçalho ELF:

O Cabeçalho ELF é o primeiro bloco de um arquivo ELF, contendo informações básicas sobre o arquivo, como o tipo, a arquitetura de destino e os deslocamentos para outros cabeçalhos importantes. Ele inclui as seguintes informações:

- **e_ident**: Contém números mágicos e bandeiras que identificam o arquivo como um ELF, especificando a classe do arquivo (32 ou 64 bits) e a ordem dos bytes (little ou big-endian).
- **e_type**: Especifica o tipo de arquivo. Para arquivos relocáveis, o valor é `ET_REL`.
- **e_machine**: Indica a arquitetura de destino, como `EM_ARM` para ARM.
- **e_version**: Versão do ELF, geralmente `EV_CURRENT`.
- **e_entry**: Endereço de entrada para executáveis, geralmente zero em arquivos relocáveis.
- **e_phoff**: Deslocamento da tabela de cabeçalhos do programa, geralmente zero em arquivos relocáveis.
- **e_shoff**: Deslocamento da tabela de cabeçalhos de seção, crucial para o processo de ligação.
- **e_flags**: Bandeiras específicas do processador.
- **e_ehsize**: Tamanho do próprio cabeçalho ELF.
- **e_phentsize**: Tamanho de uma entrada na tabela de cabeçalhos do programa, geralmente zero em arquivos relocáveis.
- **e_phnum**: Número de entradas na tabela de cabeçalhos do programa, geralmente zero em arquivos relocáveis.
- **e_shentsize**: Tamanho de uma entrada na tabela de cabeçalhos de seção.
- **e_shnum**: Número de entradas na tabela de cabeçalhos de seção.
- **e_shstrndx**: Índice da tabela de cabeçalhos de seção que contém os nomes das seções.

Estrutura de Exemplo do Cabeçalho ELF em C:

```c
#define EI_NIDENT 16

typedef struct {
    unsigned char e_ident[EI_NIDENT];
    Elf32_Half e_type;
    Elf32_Half e_machine;
    Elf32_Word e_version;
    Elf32_Addr e_entry;
    Elf32_Off e_phoff;
    Elf32_Off e_shoff;
    Elf32_Word e_flags;
    Elf32_Half e_ehsize;
    Elf32_Half e_phentsize;
    Elf32_Half e_phnum;
    Elf32_Half e_shentsize;
    Elf32_Half e_shnum;
    Elf32_Half e_shstrndx;
```

```
18 } Elf32_Ehdr;
```

Cabeçalhos de Seção:

A tabela de cabeçalhos de seção descreve a organização das seções dentro do arquivo ELF. Cada entrada nessa tabela fornece detalhes sobre uma seção específica:

- **sh_name**: Índice na tabela de strings contendo o nome da seção.
- **sh_type**: Tipo de seção, como `SHT_PROGBITS` para dados de programa ou `SHT_SYMTAB` para a tabela de símbolos.
- **sh_flags**: Bandeiras que descrevem os atributos da seção, como `SHF_WRITE` para dados graváveis e `SHF_ALLOC` para seções que ocupam memória durante a execução.
- **sh_addr**: Endereço da seção na memória, zero para seções não presentes na imagem de memória.
- **sh_offset**: Deslocamento da seção dentro do arquivo.
- **sh_size**: Tamanho da seção em bytes.
- **sh_link**: Índice de um cabeçalho de seção relacionado, dependendo do tipo de seção.
- **sh_info**: Informações adicionais, cuja interpretação depende do tipo de seção.
- **sh_addralign**: Alinhamento necessário da seção.
- **sh_entsize**: Tamanho de cada entrada para seções que contêm entradas de tamanho fixo, como a tabela de símbolos.

Estrutura de Exemplo do Cabeçalho de Seção em C:

```
1  typedef struct {
2      Elf32_Word sh_name;
3      Elf32_Word sh_type;
4      Elf32_Word sh_flags;
5      Elf32_Addr sh_addr;
6      Elf32_Off sh_offset;
7      Elf32_Word sh_size;
8      Elf32_Word sh_link;
9      Elf32_Word sh_info;
10     Elf32_Word sh_addralign;
11     Elf32_Word sh_entsize;
12 } Elf32_Shdr;
```

2) *Exemplos de Seções em um Arquivo Relocável:*

1. **`.text`**: Contém o código executável.
2. **`.data`**: Contém dados inicializados.
3. **`.bss`**: Contém dados não inicializados, que são inicializados para zero em tempo de execução.
4. **`.rodata`**: Contém dados de somente leitura.
5. **`.symtab`**: Contém a tabela de símbolos.
6. **`.strtab`**: Contém a tabela de strings para nomes de símbolos.
7. **`.shstrtab`**: Contém a tabela de strings para nomes de seções.
8. **`.rel.text`**: Contém entradas de relocação para a seção `.text`.

Essas informações permitem que o ligador e o carregador interpretem a estrutura e o conteúdo dos arquivos ELF relocáveis, facilitando o processo de construção e execução de programas.

3) *Tabela de Símbolos:*

A tabela de símbolos em um arquivo ELF contém entradas que fornecem as informações necessárias para localizar e relocar definições e referências simbólicas em um programa. Cada entrada na tabela de símbolos é descrita pela estrutura `Elf32_Sym`, que contém os seguintes campos:

1. **st_name**: Este campo contém um índice na tabela de strings que contém os nomes dos símbolos. Este índice referencia a string que nomeia o símbolo.

```
1  Elf32_Word st_name;
```

2. **st_value**: O campo `st_value` dá o valor associado ao símbolo. Dependendo do tipo de arquivo de objeto, isso pode ser um valor absoluto, um endereço ou um deslocamento.

```
1  Elf32_Addr st_value;
```

3. **st_size**: Especifica o tamanho associado ao símbolo. Para objetos de dados, esse é o número de bytes. Para funções, é o número de bytes no código da função.

```
1  Elf32_Word st_size;
```

4. **st_info**: Este campo especifica os atributos de tipo e vinculação do símbolo. A vinculação (linkage) está nos 4 bits superiores e o tipo nos 4 bits inferiores.

```
1  unsigned char st_info;
```

5. **st_other**: Atualmente não usado e contém 0. Pode ser usado em futuras extensões para informações adicionais.

```
1  st_other: Atualmente não usado e contém 0. Pode
   ser usado em futuras extensões para informações
   adicionais
```

6. **st_shndx**: Contém o índice da tabela de cabeçalhos de seção associado ao símbolo. Este índice vincula o símbolo à sua seção.

```
1   Elf32_Half st_shndx;
```

4) *Estrutura Completa da Entrada da Tabela de Símbolos:*

```
1   typedef struct {
2       Elf32_Word st_name;
3       Elf32_Addr st_value;
4       Elf32_Word st_size;
5       unsigned char st_info;
6       unsigned char st_other;
7       Elf32_Half st_shndx;
8   } Elf32_Sym;
```

5) *Entradas de Relocação:*

Relocação é o processo de conectar referências simbólicas com definições simbólicas. Arquivos relocáveis precisam ter informações que descrevem como modificar o conteúdo de suas seções para produzir um arquivo executável ou uma biblioteca compartilhada. As entradas de relocação são esses dados e são representadas pelas estruturas `Elf32_Rel` e `Elf32_Rela`.

6) *Estrutura `Elf32_Rel`:*

A estrutura `Elf32_Rel` representa uma entrada de relocação sem um addendo explícito. O addendo é armazenado implicitamente no local a ser modificado.

1. **r_offset**: Especifica o local onde a relocação deve ser aplicada. Para arquivos relocáveis, este é um deslocamento desde o início da seção até a unidade de armazenamento afetada pela relocação.

```
1   Elf32_Addr r_offset;
```

2. **r_info**: Codifica tanto o índice da tabela de símbolos quanto o tipo de relocação a ser aplicado. O índice da tabela de símbolos está nos bits superiores, e o tipo de relocação está nos bits inferiores.

```
1   Elf32_Word r_info;
```

Estrutura `Elf32_Rela`:

A estrutura `Elf32_Rela` é semelhante à `Elf32_Rel`, mas inclui um addendo explícito:

```
1   Elf32_Sword r_addend;
```

Estruturas Completas das Entradas de Relocação:

```
1   typedef struct {
2       Elf32_Addr r_offset;
3       Elf32_Word r_info;
4   } Elf32_Rel;
5
6   typedef struct {
7       Elf
8
9   32_Addr r_offset;
10      Elf32_Word r_info;
11      Elf32_Sword r_addend;
12  } Elf32_Rela;
```

7) *Exemplos de Tipos de Relocação para ARM:*

Alguns possíveis tipos de relocação são:

- **`R_ARM_ABS32`** (2): Relocação direta de 32 bits.
- **`R_ARM_THM_CALL`** (10): Deslocamento imediato para BL e BLX Thumb32.
- **`R_ARM_CALL`** (28): Deslocamento imediato para BL.

V. Modelo de Solução

A. O Poli Assembler

O Poli Assembler aqui descrito é um assembler de duas passagens, no sentido de que ele produz arquivos relocáveis. Na verdade, sua única saída possível são arquivos relocáveis no formato `ELF 32-bit LSB relocatable, ARM, EABI5 version 1 (SYSV), not stripped`. A operação do assembler pode ser dividida nos passos abaixo:

1. O `Tokenizer` pega as linhas de texto do Leitor e as divide em tokens individuais. Esses tokens representam as menores unidades de significado na linguagem assembly, como instruções, registradores ou rótulos;
2. O `Symbolizer` realiza a primeira passagem do processo de montagem em duas passagens. Durante esta fase, ele analisa os tokens e constrói uma tabela de símbolos, que mapeia os rótulos e outros nomes simbólicos para seus respectivos endereços de memória. Ele também lida com diretivas que mudam a seção atual do código (por exemplo, `.text`, `.data`) e atualiza o endereço atual com base no tamanho das instruções e dos dados;
3. Finalmente, o `Assembler` executa a segunda passagem. Durante esta passagem, ele usa a tabela de símbolos para resolver os endereços e traduz o código assembly em código de máquina, produzindo o arquivo ELF relocável final.

Esses passos serão explicados em mais detalhes nas seções subsequentes.

B. O Tokenizer

Um Tokenizer é um componente fundamental em muitos compiladores e assemblers. Seu papel principal é dividir o código-fonte de entrada em unidades menores chamadas tokens. Tokens são os elementos básicos da sintaxe de uma

linguagem de programação, como palavras-chave, operadores, identificadores e literais. Ao converter o texto bruto em uma sequência de tokens, o Tokenizer permite que as etapas subsequentes do assembler entendam e processem o código.

No Poli Assembler, o `Tokenizer` é responsável por converter cada linha do código assembly de entrada em uma série de tokens. Esses tokens são então usados por outros componentes do assembler para realizar tarefas como resolução de símbolos e geração de código.

O `Tokenizer` opera lendo cada linha de texto da entrada. Em seguida, ele processa essa linha dividindo-a em segmentos menores com base em um conjunto de padrões predefinidos, como registradores, números e diretivas do assembler. Esses segmentos, referidos como literais no código, são então convertidos em tokens correspondentes através de uma série de verificações e correspondências de padrões.

Por exemplo, se um literal corresponder ao padrão de um registrador, como "r0" ou "sp", o `Tokenizer` criará um `Token::REGISTER` para representá-lo. Se o literal começar com o caractere "#", ele pode ser reconhecido como um valor imediato, e um `Token::IMMEDIATE` correspondente será gerado. O `Tokenizer` também lida com construções mais complexas, como rótulos, instruções e diretivas, garantindo que cada uma seja corretamente identificada e tokenizada.

No final, os tokens gerados serão usados pelo assembler para processar o código assembly de uma maneira estruturada e sistemática, preparando o caminho para as fases subsequentes de simbolização e geração de código.

C. O Symbolizer

Em um assembler, o processo de simbolização resolve rótulos e gerencia endereços de memória dentro do código. O `Symbolizer` é o componente responsável por essa tarefa, operando durante a primeira passagem do processo de montagem. Sua função principal é identificar e registrar todas as referências simbólicas—como rótulos e diretivas—no código-fonte e mapeá-las para endereços de memória específicos.

No Poli Assembler, o `Symbolizer` trabalha iterando através dos tokens gerados pelo `Tokenizer`. À medida que processa cada linha de código, ele mantém o controle do endereço atual dentro do programa e registra as localizações de quaisquer rótulos que encontrar. Esses rótulos são armazenados em uma `SymbolTable`, que mapeia o nome de cada símbolo para seu respectivo endereço, escopo (global ou local), e a seção do código (por exemplo, `.text`, `.data`, ou `.bss`).

O `Symbolizer` também lida com diretivas do assembler, como `.global`, que indica que um símbolo deve ser acessível a partir de outros arquivos, ou diretivas de seção como `.text` e `.data`, que mudam a seção atual do código sendo processado. Essas diretivas influenciam como o `Symbolizer` atualiza o endereço atual e onde ele armazena os símbolos.

Quando o `Symbolizer` encontra um rótulo, ele cria um objeto `Symbol` representando o nome do rótulo. Em seguida, associa esse símbolo ao endereço atual e o armazena na `SymbolTable`. Esse processo garante que todo rótulo no código assembly seja atribuído a um endereço específico, o que é essencial para gerar o código de máquina correto na segunda passagem.

Além disso, o `Symbolizer` leva em conta o tamanho de cada instrução e diretiva, incrementando o endereço atual conforme necessário. Por exemplo, ao processar uma instrução, ele aumenta o endereço em quatro bytes (o tamanho típico de uma instrução ARM, excluindo a arquitetura de instrução Thumb).

O `Symbolizer` constrói um mapa das referências simbólicas do programa. Este mapa, armazenado na `SymbolTable`, é então usado na segunda passagem para resolver essas referências e produzir o arquivo ELF relocável.

D. O Poli Assembler

O assembler é a etapa final no processo de tradução da linguagem assembly para código de máquina. Ele pega os tokens produzidos pelo `Tokenizer` e a tabela de símbolos criada pelo `Symbolizer` e os transforma em um arquivo ELF relocável completamente formado. Este arquivo pode então ser alimentado a um linker que gerará um arquivo executável. O assembler faz uso do lexer, que é brevemente explicado abaixo. No final, a operação real do assembler é elucidada.

1) *O Lexer:*

O `Lexer` no Poli Assembler serve como um intermediário entre o `Tokenizer` e o `Assembler`. Sua função principal é processar as instruções de assembly produzidas pelo `Tokenizer`, substituindo as referências simbólicas por endereços de memória reais e convertendo instruções de assembly de alto nível em operações de máquina de baixo nível.

Quando o `Lexer` processa uma linha de tokens, ele primeiro substitui os rótulos simbólicos por seus respectivos endereços de memória, usando a tabela de símbolos construída pelo `Symbolizer`. Isso é crucial para converter o código assembly simbólico em operações concretas que o processador pode executar. O `Lexer` também lida com pseudo-operações—instruções de assembly de alto nível que não mapeiam diretamente para o código de máquina. Essas pseudo-ops são substituídas por uma ou mais instruções padrão que atingem o mesmo efeito. Uma limitação notável da maneira atual como essas pseudo-ops são tratadas é que elas não são expandidas em múltiplas instruções, o que poderia ser necessário para pseudo-ops mais complexas.

O `Lexer` então identifica o tipo de instrução na linha —se é uma operação aritmética lógica, uma operação de movimentação, uma operação de desvio, ou uma operação de carga/armazenamento—e faz o parsing dos operandos

de acordo. Cada instrução analisada é convertida em uma `CpuOperation`, que representa a operação de nível de máquina que será escrita no arquivo ELF de saída. Ela não só contém dados sobre o mnemonico, como a condição de código e se deve definir os flags, mas também sobre os operandos esperados, como os números dos registradores e os valores imediatos.

Além de fazer o parsing das instruções, o `Lexer` é responsável por acompanhar o endereço atual dentro do código. À medida que processa cada linha, ele incrementa o endereço para refletir o tamanho da instrução, garantindo que todas as referências de memória sejam calculadas corretamente. Isso porque o lexer lida apenas com instruções reais, não com a estrutura do próprio arquivo ELF.

2) *O Assembler:*

O `Assembler` é a etapa final no processo de tradução do Poli Assembler, responsável por gerar a saída final do código de máquina na forma de um arquivo ELF relocável. Ele combina os tokens produzidos pelo `Tokenizer`, as resoluções de endereços feitas pelo `Lexer` e a tabela de símbolos construída pelo `Symbolizer` para criar uma representação completa e funcional do código assembly de entrada em código de máquina. Ele também cria a estrutura do arquivo ELF, incluindo seções, símbolos e entradas de relocação.

O `Assembler` começa iterando através de cada linha de tokens. Para cada linha, ele determina o tipo de conteúdo—se é uma diretiva, uma instrução ou dado—e o processa de acordo. Se a linha contém uma diretiva de seção (por exemplo, `.text`, `.data` ou `.bss`), o `Assembler` alterna a seção atual, garantindo que instruções e dados sejam colocados na seção apropriada do arquivo ELF.

Ao encontrar uma instrução, o `Assembler` depende do `Lexer` para analisar a instrução e convertê-la em código de máquina. O `Lexer` processa a instrução e seus operandos, lidando com quaisquer pseudo-ops e resolvendo referências simbólicas. O código de máquina resultante é então adicionado a um buffer, que acumula os dados binários para a seção atual.

O `Assembler` também gerencia referências de símbolos não resolvidos. Durante a primeira passagem, alguns rótulos podem não ter sido resolvidos pelo `Symbolizer`. O `Assembler` rastreia essas referências não resolvidas e garante que elas sejam corretamente vinculadas ou ajustadas ao criar o arquivo ELF final. Esse passo possibilita que o arquivo de objeto relocável gerado seja vinculado a outros arquivos ou carregado em diferentes endereços de memória.

Uma vez que todas as linhas foram processadas, o `Assembler` finaliza as seções e os símbolos. Ele escreve o código de máquina armazenado em buffer no arquivo ELF, cria entradas de símbolos e gera entradas de relocação para quaisquer símbolos que precisem ser ajustados durante o linking. O resultado é um arquivo ELF completamente formado que segue as especificações da arquitetura ARMv7 e pode ser vinculado ou executado como parte de um programa maior.

O `Assembler` é o componente que sintetiza todas as etapas anteriores do processo de montagem—tokenização, simbolização e análise léxica—em uma saída final de código de máquina relocável.

E. Geração de Arquivos ELF no Poli Assembler

O Poli Assembler foi projetado para converter código assembly em um formato de arquivo ELF relocável para arquiteturas ARM, visto na seção Seção IV.B. O processo de geração desse arquivo ELF envolve dois componentes: o `ElfWriter` e o `SectionData`. Juntos, eles lidam com a criação, organização e gravação de seções ELF, tabelas de símbolos e entradas de relocação.

1) *O ElfWriter:*

O `ElfWriter` é o componente principal responsável por orquestrar a geração do arquivo ELF no Poli Assembler. Seu papel é gerenciar as várias seções do arquivo ELF, garantir que as dependências entre seções e símbolos sejam resolvidas e, finalmente, gravar os dados estruturados no arquivo de saída.

O `ElfWriter` lida com a criação e organização de seções dentro do arquivo ELF. Como visto na seção Seção IV.B, seções em arquivos ELF podem conter código executável, dados, símbolos ou entradas de relocação. Quando uma nova seção é adicionada, o `ElfWriter` determina o tipo apropriado e os atributos da seção com base no seu nome. Por exemplo, uma seção `.text` será marcada como executável e alocada na memória, enquanto uma seção `.bss` será marcada como gravável e alocada, mas não ocupará espaço no arquivo, pois contém dados não inicializados.

O `ElfWriter` também gerencia o alinhamento e o tamanho de cada seção. Além disso, ele lida com a ordenação correta das seções, particularmente garantindo que as seções de relocação apareçam imediatamente após as seções que elas referenciam.

Outra tarefa do `ElfWriter` é a de resolver as dependências entre seções e símbolos. Isso inclui vincular entradas de relocação aos seus respectivos símbolos e garantir que cada seção referencie corretamente as outras. O `ElfWriter` realiza isso por meio de um processo de duas passagens, onde primeiro reúne todas as informações necessárias sobre as relações entre seções e símbolos e, em seguida, atualiza as entradas relevantes com os índices e deslocamentos corretos.

Após organizar e resolver as dependências, o `ElfWriter` procede para gravar o arquivo ELF. Esse processo envolve várias etapas:

- Cabeçalho do Arquivo: O `ElfWriter` grava o cabeçalho do arquivo ELF, que inclui metadados sobre o tipo de

arquivo, a arquitetura de destino e o ponto de entrada (que geralmente é zero para arquivos relocáveis).
- Cabeçalhos de Seção: Em seguida, ele grava os cabeçalhos de seção, que descrevem o layout e os atributos de cada seção no arquivo.
- Dados da Seção: Finalmente, os dados reais de cada seção—como código executável, tabelas de símbolos e entradas de relocação—são gravados no arquivo.

2) *O SectionData:*

O componente `SectionData` encapsula o conteúdo de cada seção dentro do arquivo ELF. Ele é um `enum` que pode representar diferentes tipos de dados, incluindo bytes brutos, símbolos e entradas de relocação, explicados na seção Seção IV.B. Cada variante do `SectionData` corresponde a um tipo específico de seção no arquivo ELF, e fornece métodos para manipular e acessar os dados subjacentes.

1. **Bytes**

A variante `Bytes` do `SectionData` é usada para armazenar dados binários brutos, tipicamente representando o código executável ou os dados inicializados em uma seção como `.text` ou `.data`. O tamanho desses dados é gerenciado pelos métodos do `SectionData`, que permitem ao `ElfWriter` determinar o comprimento e os requisitos de alinhamento para a seção.

1. **Símbolos**

A variante `Symbols` armazena entradas para a tabela de símbolos, que é crucial para ligação e relocação. Cada entrada de símbolo inclui informações como o nome do símbolo, valor (endereço), tamanho e tipo. Os métodos do `SectionData` permitem adicionar novos símbolos, que o `ElfWriter` posteriormente resolve e grava no arquivo ELF.

1. **Entradas de Relocação**

A variante `RelocationEntries` é usada para armazenar dados sobre como certos endereços nas seções de código ou dados precisam ser ajustados quando o arquivo ELF é vinculado ou carregado. Essas entradas são críticas para garantir que as referências a símbolos, como chamadas de função ou acessos a dados, apontem para os locais corretos uma vez que o executável final seja produzido. Os métodos do `SectionData` suportam a adição de novas entradas de relocação, que incluem informações sobre o deslocamento, tipo e símbolo a ser ajustado.

VI. Implementação

Considerou-se que não seria proveitoso discutir minuciosamente o código desenvolvido. Portanto, para demonstrar o funcionamento do Poli Assembler, decidiu-se por mostrá-lo em ação. Utilizou-se o Poli Assembler para realizar a montagem de um programa de Fibonacci e, após isso, sua linkagem com o linker da GNU e execução no `qemu-system-arm`. Antes de mostrar os resultados, explica-se a estrutura do projeto a seguir.

A. Estrutura do projeto

O Poli Assembler está estruturado em diversos módulos e arquivos, cada um responsável por uma parte específica do processo de conversão do código assembly em código de máquina executável. A seguir, explica-se como esses componentes funcionam e interagem entre si para realizar o processo de montagem.

1. Componentes Principais

- `main.rs`: Este é o ponto de entrada do programa. Ele coordena o fluxo de execução do assembler e a entrada e saída, escolhendo os arquivos que vai operar e chamando as funções e módulos necessários para processar o arquivo de entrada e gerar o arquivo de saída. Aqui, é onde a leitura do código assembly e a subsequente chamada para as diversas etapas do processamento ocorre.
- `assembler.rs`: Esse arquivo implementa um assembler que transforma código assembly em um arquivo ELF executável. Ele lê e interpreta o código assembly utilizando um tokenizer e um lexer para dividir o código em partes menores e entender suas instruções. Além disso, o assembler gerencia as diferentes seções do código, como .text, .data, e .bss, organizando essas seções corretamente no arquivo ELF. Ele também é responsável por encontrar e resolver rótulos e variáveis usadas no código, garantindo que todas as referências estejam corretas antes de gerar o arquivo ELF final. Por fim, o assembler escreve o código de máquina no formato ELF, pronto para ser executado em um processador ARM.
- `reader.rs`: Este módulo é responsável pela leitura do arquivo de entrada contendo o código assembly. Ele lê o conteúdo do arquivo de texto e o prepara para ser processado pelas etapas subsequentes, como o lexer e o tokenizer.

2. Tokenização e Análise Léxica

- `tokenizer.rs`: O tokenizador (tokenizer) é responsável por converter o texto lido do arquivo assembly em tokens. Tokens são unidades básicas da linguagem, como palavras-chave, operadores, registradores, ou números. O tokenizador divide o código em componentes menores que podem ser mais facilmente processados nas etapas seguintes.
- `lexer/`: Este diretório contém o código responsável pela análise léxica do código assembly. A análise léxica

envolve a interpretação dos tokens para identificar as operações de CPU e outras expressões presentes no código.

- `cpu_op.rs`: Este arquivo define as operações a serem realizadas na CPU, juntando a instrução com seus operandos.
- `symbolizer.rs`: O simbolizador (symbolizer) efetivamente faz o primeiro passe sobre o arquivo de entrada, subsequentemente tratando dos símbolos e rótulos no código, e gerenciando como eles são mapeados para endereços de memória durante a montagem.
- `expression/`: Este subdiretório lida com expressões mais complexas, como deslocamento de barril (**barrel shifter**), índices de registradores, e outras construções específicas da linguagem assembly ARM.
- `operations/`: Aqui, são implementadas as operações específicas da CPU, como operações de branch e de load/store, que são importantes para o funcionamento do código gerado.

3. Tokens e Instruções

- `token/`: Este diretório contém módulos que definem os diferentes tipos de tokens utilizados pelo lexer e pelo assembler.
  - `instruction.rs e instruction_name.rs`: Estes arquivos definem as instruções e seus nomes, mapeando cada token de instrução para o que o processador executará.
  - `register.rs e immediate.rs`: Estes arquivos lidam com tokens que representam registradores e valores imediatos, respectivamente. Eles ajudam a identificar e processar esses componentes no código assembly.

4. Geração do ELF

- `elf/`: Este diretório é responsável por tudo relacionado ao formato ELF, que é o formato de saída do assembler.
  - `elf_writer.rs`: Este módulo escreve o código gerado no formato ELF, criando um arquivo binário que pode ser linkado e carregado para execução em um processador ARM.
  - `section_data.rs`: Provavelmente lida com as diferentes seções do arquivo ELF, como a seção de código, dados, e a tabela de símbolos.
  - `mod.rs`: Este arquivo geralmente coordena as interações entre os diferentes módulos dentro do diretório `elf/`.

5. Utilitários

- `utils.rs`: Este arquivo contém funções auxiliares que são utilizadas em várias partes do assembler. Essas funções podem incluir manipulação de strings, cálculos matemáticos, ou outras operações genéricas que suportam o funcionamento do assembler.

B. Exemplo de Fibonacci

Para demonstrar o assembler em funcionamento, utilizou-se o programa abaixo, que calcula o décimo elemento de uma sequência Fibonacci.

```
1  .text
2  fibonacci:
3      mov r0, #10
4      cmp r0, #0
5      beq zero_case
6
7      cmp r0, #1
8      beq one_case
9
10     mov r1, #0
11     mov r2, #1
12
13     mov r3, #2
14
15 loop:
16     cmp r3, r0
17     bgt end_loop
18
19     add r4, r1, r2
20     mov r1, r2
21     mov r2, r4
22
23     add r3, r3, #1
24
25     b loop
26
27 end_loop:
28     mov r0, r2
29     b end;
30
31 zero_case:
32     mov r0, #0
```

```
34
35 one_case:
36     mov r0, #1
37     b end
38
39 end:
40   b end
```

Utilizou-se o `Makefile` abaixo para o teste mostrado a seguir:

```
all:
  cargo run -- -i hello.txt -o hello.o
  arm-none-eabi-ld -T linker.ld  -o out.elf hello.o
  qemu-system-arm -s -M virt -kernel out.elf

debug:
    gdb -ex "set architecture arm" -ex \"target
extended-remote :1234" -ex "load" out.elf -ex "layout
asm" -ex "layout regs" -ex "b fibonacci" -ex "b end"
-ex "j fibonacci"
```

Após sua a sua montagem pelo Poli Assembler, o arquivo objeto resultante é um ELF relocável com a estrutura abaixo:

```
ipontes3olavy@localhost:[poli-as]$ arm-none-eabi-readelf -S hello.o
There are 5 section headers, starting at offset 0x34:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        00000000 0000fc 000058 00  AX  0   0  4
  [ 2] .symtab           SYMTAB          00000000 000154 000080 10      3   8  4
  [ 3] .strtab           STRTAB          00000000 0001d4 000031 00      0   0  1
  [ 4] .shstrtab         STRTAB          00000000 000205 000021 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  D (mbind), y (purecode), p (processor specific)
```

Figura 1: Estrutura do ELF relocável gerado pelo Poli Assembler

Abaixo se encontram alguns dos campos do ELF gerado:

Elf_Ehdr

0x0	e_ident	127,69,76,70,1,1,1,0,0,0,0,0,0,0,0,0
0x10	e_type	ET_REL
0x12	e_machine	EM_ARM
0x14	e_version	1
0x18	e_entry	0
0x1c	e_phoff	0
0x20	e_shoff	0x34
0x24	e_flags	0x5000000
0x28	e_ehsize	0x34
0x2a	e_phentsize	0
0x2c	e_phnum	0
0x2e	e_shentsize	0x28
0x30	e_shnum	5
0x32	e_shstrndx	4

Elf_Ehdr.e_ident

0x0	EI_MAG0	127
0x1	EI_MAG1	E
0x2	EI_MAG2	L
0x3	EI_MAG3	F
0x4	EI_CLASS	ELFCLASS32
0x5	EI_DATA	ELFDATA2LSB
0x6	EI_VERSION	1
0x7	EI_OSABI	ELFOSABI_SYSV
0x8	EI_ABIVERSION	0

Figura 2: Cabeçalho do ELF gerado

Elf_Shdr 0

0x0	sh_name	0 -> [empty string]
0x4	sh_type	SHT_NULL
0x8	sh_flags	-
0xc	sh_addr	0
0x10	sh_offset	0
0x14	sh_size	0
0x18	sh_link	0
0x1c	sh_info	0
0x20	sh_addralign	0
0x24	sh_entsize	0

Elf_Shdr 1

0x0	sh_name	1 -> .text
0x4	sh_type	SHT_PROGBITS
0x8	sh_flags	SHF_ALLOC | SHF_EXECINSTR
0xc	sh_addr	0
0x10	sh_offset	0xfc
0x14	sh_size	0x58
0x18	sh_link	0
0x1c	sh_info	0
0x20	sh_addralign	0x4
0x24	sh_entsize	0

Elf_Shdr 2

0x0	sh_name	25 -> .symtab
0x4	sh_type	SHT_SYMTAB
0x8	sh_flags	-
0xc	sh_addr	0
0x10	sh_offset	0x154
0x14	sh_size	0x80
0x18	sh_link	0x3
0x1c	sh_info	0x8
0x20	sh_addralign	0x4
0x24	sh_entsize	0x10

Figura 3: Alguns cabeçalhos de seção do arquivo ELF gerado

Elf_Sym 1

0x0	st_name	1 -> .text
0x4	st_value	0
0x8	st_size	0
0xc	st_info	ELF_ST_INFO(STB_LOCAL, STT_SECTION)
0xd	st_other	0
0xe	st_shndx	1 -> .text

Elf_Sym 2

0x0	st_name	7 -> end_loop
0x4	st_value	60
0x8	st_size	0
0xc	st_info	ELF_ST_INFO(STB_LOCAL, STT_NOTYPE)
0xd	st_other	0
0xe	st_shndx	1 -> .text

Elf_Sym 3

0x0	st_name	45 -> end
0x4	st_value	84
0x8	st_size	0
0xc	st_info	ELF_ST_INFO(STB_LOCAL, STT_NOTYPE)
0xd	st_other	0
0xe	st_shndx	1 -> .text

Elf_Sym 4

0x0	st_name	16 -> fibonacci
0x4	st_value	0
0x8	st_size	0
0xc	st_info	ELF_ST_INFO(STB_LOCAL, STT_NOTYPE)
0xd	st_other	0
0xe	st_shndx	1 -> .text

Figura 4: Algumas entradas da tabela de símbolos gerada

Nota-se que as referências à seções, offsets dentro de seções, assim como outras informações e metadados do arquivo ELF foram gerados com sucesso pelo assembler. Isso permitiu com que o ELF gerado fosse integrado com as ferramentas de linkagem da GNU. Consequentemente, foi possível executar o programa numa máquina ARM simulada através do simulador QEMU. Abaixo estão algumas imagens de execução.

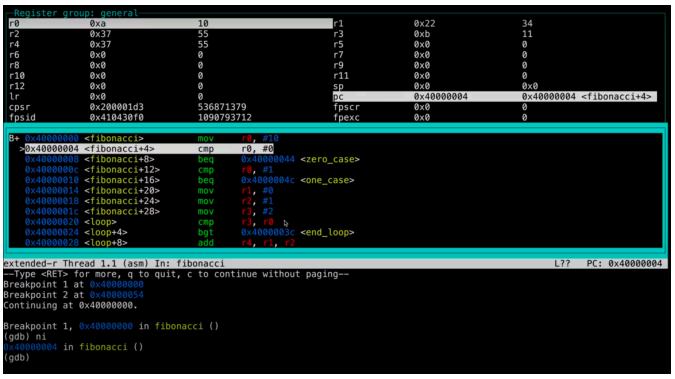

Figura 5: Simulação no QEMU antes da execução da função

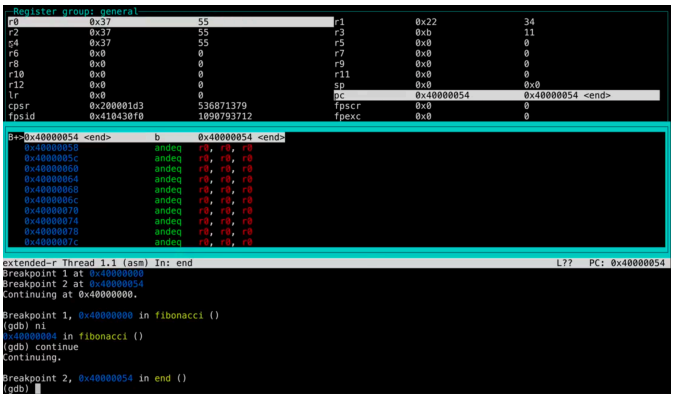

Figura 6: Simulação no QEMU após a execução da função

Nota-se que a função retorna, no registrador `r0`, o valor correto do décimo elemento da sequência de Fibonacci: 55.

VII. Agradecimentos

Gostaríamos de agradecer ao professor Bruno Basseto por todas as aulas sobre o conjunto de instruções ARM, arquiteturas de processadores e sistemas operacionais, assim como pelo suporte durante a realização do trabalho. Também gostaríamos de agradecer ao professor Marco Túlio pelo seu apoio durante o oferecimento da disciplina.

VIII. Considerações Finais e Conclusão

Espera-se que, ao longo do relatório, tenha ficado claro o funcionamento geral do Poli Assembler, assim como o contexto no qual o mesmo funciona e como funciona o formato ELF, que permite programas relocáveis.

Ficamos muitos satisfeitos com o resultado final do projeto. Apesar de seu escopo limitado (apenas suportando uma pequena parte das instruções do conjunto de instruções ARM), o nosso assembler consegue interagir com ferramentas externas, como o Linker da GNU, podendo assim, ser usado na construção de software legítimo. A experiência de criar um assembler do zero ao um foi muito enriquecedora.

Como considerações finais, salienta-se, em especial, que uma possível rota futura do projeto é a implementação de mais instruções ARM. Ademais, considerou-se que a lógica de parsing das instruções poderia ser mais flexível com os diferentes formatos de instruções do conjunto ARM. Finalmente, a geração de arquivos ELF também foi limitada, devido ao escopo e recursos do projeto. O assembler é capaz de gerar somente arquivos ELF relocáveis, resolvendo dependências entre alguns tipos de seções, símbolos e relocações. O formato ELF, em sua totalidade, é bem diverso e pode-se expandir o projeto para englobar mais de suas funcionalidades.

References

[1] D. Salomon, *Assemblers and Loaders*. Cambridge: Cambridge University Press, 1990.

[2] I. Standard, “ISA A64 A Profile”. 2022.

[3] Linux Foundation, “ELF Specification”.