Problema C: Implementação de um interpretador feito em C para a arquitetura PicoQuickProcessor

Grupo 1

11 de novembro de 2025

Sumário

1 Membros

Nome	Curso
Francisco Passos dos Santos Alves	Ciência da Computação
Gabriel Santos de Souza	Engenharia da Computação
Guilherme Ferreira Amâncio	Ciência da Computação
João Vinícius de Almeida Argolô	Engenharia da Computação

2 Introdução

Esta documentação refere-se ao "Problema C" da documentação disponibilizada pela equipe da CORETECH para o processo seletivo (PSEL) a qual os membros responsáveis por este projeto estão participando.

O objetivo do projeto a qual este arquivo irá documentar — assim como foi pedido na documentação do PSEL da CORETECH — é criar um simulador, em linguagem C/C++, capaz de reproduzir o comportamento de um processador 32 Bits como artifício para didática. Essa simulação será executada em um ambiente x86-64 e um arquivo contendo um código de máquina será oferecida como a entrada e, por intermédio do simulador projetado conforme as regras de arquitetura d o PicoQuick, gerará um relatório detalhado da execução e do estado final do sistema.

Esta documentação contará com informações técnicas valiosas de como cada átimo deste projeto funciona e assim servindo como material de apoio técnico-didático para disciplinas como arquitetura de computadores e sistemas operacionais.

3 Estruturas de Dados

O projeto está organizado de uma maneira semelhante ao paradigma orientado a objetos, utilizando de `structs` para isso. Como legenda, saiba que, para todo campo de um `struct` que foi citado, em parênteses estará seu tipo.

3.1 Memória

Para a memória, um `struct` contendo múltiplos campos foi utilizado:

1. `mem8` (`uint8_t`)

   Representa uma memória inteira com alinhamento de 8 bits

2. `size` (`uint32_t`)

Indica o tamanho atual da instância de memória. Usado para inicialização, com um valor personalizado.

3. `loaded` (`bool`)

   Indica se esta instância de memória foi carregada com código de máquina.

3.2 Logger

Essa duas estruturas foram criadas para o logging.

1. `content` (`char**`)

   Armezena strings salvas.

2. `index` (`int`)

   Controla o índice de strings armazenadas.

3. `size` (`int`)

   Tamanho do buffer (definido na criação.

1. `buffer` (`Buffer`)

   Estrutura de buffer.

2. `enabled` (`bool`)

   Flag de habilitação do logger.

3. `instruction_cnt[16]` (`int`)

   Coleta dados de instruções.

4. `reached_address` (`uint32_t`)

   Indica maior valor de PC já salvo para impressão.

5. `output` (`FILE*`)

   Referência para arquivo de saída.

3.2.1 Funções

Os tipos `uint` utilizados vem do header `<stdint.h>`, e portanto possuem larguras fixas, independentemente da implementação.

Funções que acompanham esse `struct` são:

- `mem_create` (`Memory*`)

Cria uma nova instância de memória. Recebe como argumento `size`, alocando quantidade de memória indicado por ele, com alinhamento de 1 byte.

- `mem_destroy` (`void`)

  Wrapper para a função `free`.

- `mem_load_program` (`void`)

  Dada uma instância de memória `mem` e um caminho de diretório `input_path`, essa função carrega o arquivo na memória.

- `mem_read8` (`uint16_t`)

  Lê um byte de memória.

- `mem_read32` (`uint32_t`)

  Lê quatro bytes de memória.

3.3 Registradores

Os registradores não estão organizados numa forma específica para eles. Todavia, fazem parte do `struct Cpu`, que guarda as seguintes informações:

1. `pc` (`uint16_t`)

   Um ponteiro que aponta para qual posição da memória está sendo lido pela CPU.

2. `r[REG_COUNT]` (`uint32_t`)

   Cria um `array` que servirá para simular os 16 registradores de propósito geral da arquitetura.

O campo `flags` foi omitido, mas ele será explicado logo abaixo.

3.4 Flags

As `flags`, assim como os registradores, fazem parte da definição da CPU.

Elas são manipuladas pela função `update_flags`. Como exigido pelo documento, as flags são `L` (*Less*), `E` (*Equal*) e `G` (*Greater*).

4 Lógica Principal

Um `while` loop é usado para fazer o programa rodar até o fim do arquivo. Nele, é executado a função `cpu_cycle`. Ela traduz código de máquina guardado na memória em instruções e as executa. A função incrementa o campo `pc` em 4 pois as instruções estão separadas em blocos de quatro sequências hexadecimais. Essa é a função principal do projeto, então vamos se aprofundar nela.

Prosseguindo com a função, o código de máquina é lido com a função `fetch`. A mesma é responsável por extrair o `upcode` e os operandos. As instruções são armazenados na memória em formato *Little Endian* para a sua devida leitura. O papel do `fetch` é reorganizar de modo que a CPU entenda e processe a instrução.

Outro componente importante é a função `decode`, usada para traduzir o código de máquina carregado pelo `fetch` em instruções claras para o interpretador. Após receber código de máquina, ela pega os 8 bits mais significativos e depois dá um shift pra direita por 24 para eliminar os zeros. Esse processo acontece para isolar o textttopcode.

Por fim, a `execute` recebe a instrução decodificada e a relaciona com uma tabela de instruções (um `array`). A ideia de Vinícius aqui era de implementar o `execute` de uma forma que ele tivesse tempo $O(1)$ (constante), mas o interessante é que o compilador já faz isso com `switches`, usando as flags de otimização. Segue o bloco de código do `cpu.c` que implementa esse processo:

add bloco de código add – CPU CYCLE

essa função é chamada em *main* dentro de *loop*

add bloco da print

4.1 Logging

Enquanto o programa executa, para cada instrução, uma linha da log é gerada e adicionada no buffer do *logger*. Quando o programa finaliza, a mensagem final de `0xF0F0->EXIT` no fim da execução é impressa e a função `cpu_finishing_simulation_log` é executada e a log completa é impressa com `log_print_all`.

5 Desafios

O principal desafio foi o tempo. O projeto demandou muito esforço e agilidade para o seu planejamento, execução, direção e documentação. Além deste, para tornar o projeto mais didático e mais auto-explicativo, foi necessário um amplo conhecimento e experiência na linguagem C e arquitetura de computadores. Aplicações como: arquitetura, design do código pra otimização e organização e a desenvoltura do módulo foram um dos principais desafios como mencionou Vinícius, principal desenvolvedor do simulador. Os contra-tempos não se voltaram apenas ao projeto, como também à liderança do grupo para que o mesmo fosse o mais produtivo possível, além da saída de um membro da equipe durante o desenvolvimento do projeto.

6 Decisões

A primeira decisão, introduzida por Francisco Passos e aceita pelo grupo, consistiu na subdivisão da liderança, de modo que aquele que se destacasse em determinada área do projeto seria o responsável por ditar as diretrizes de organização e planejamento dentro de seu setor de atuação. Com este princípio em mente, Gabriel ficou responsável pela liderança da documentação do projeto, visto que já possuía experiência com LaTeX. Vinícius, por sua vez, assumiu a liderança da execução prática do projeto, por ser o membro mais "mão na massa" e já possuir uma maior experiência acadêmica comparada aos demais. Por fim, Francisco ficou encarregado por gerenciar o repositório no GitHub, mantendo-se como chefe de equipe e responsável por acompanhar o desenvolvimento geral do projeto tanto da simulação, como da documentação. Por fim, o Guilherme ficou como ajudante do desenvolvimento do simulador, sendo orientado pelo Vinícius.

A segunda decisão foi a escolha pela resolução do Problema C. O principal motivo foi o fato de que todos os integrantes do grupo já possuíam experiência com a linguagem C, o que tornava o Problema C a escolha mais conveniente para o bom andamento do projeto. Mesmo que algum membro não atuasse diretamente no simulador, teria ainda assim a capacidade de compreender as etapas de sua construção.

A terceira e última decisão, tomada por Vinícius — líder da execução do projeto — dizia respeito à organização do código em C do pqp-interpreter. A estrutura do código foi elaborada de forma a ser o mais didática possível, com o desenvolvimento dividido em etapas bem definidas, o que contribuiu significativamente para a legibilidade e manutenção do projeto. Essa decisão representou um dos maiores desafios enfrentados, conforme mencionado na seção de desafios. Outras decisões incluem: Alinhamento de 8 bits da memória para tornar a leitura atômica, encapsulamento das funções da CPU de modo a criar uma interface mais limpa para quem chamar o `cpu.h`. Toda API pública para o usuário está em `/include`, dividir as instruções em tipos para facilitar a decodificação, além disso a extensão de sinal é feita em tempo de execução e a lógica de algumas funções como os jumps, foram centralizadas. Na etapa de execução, as funções foram dividas de modo a respeitar a estrutura física da CPU, separando-as em 3 arquivos:

- `alu.c`
- `lsu.c`
- `jump.c`

7 Aprendizado como uma unidade

O grupo uniu-se sob a filosofia de pensar como uma unidade. Cada integrante contribuiu com o que pôde e da forma que lhe era possível, utilizando os re-

cursos disponíveis naquele momento. Os participantes que aceitaram o desafio dedicaram-se com empenho e responsabilidade, respeitando as condições e o prazo estabelecidos para a pesquisa. Desde a gênese, o foco permaneceu na essência do projeto.

> "Adam Smith disse que a melhoria individual servia à conquista do grupo. Mas isso é incompleto. O melhor resultado ocorre quando todos no grupo fazem o que é melhor para si e também para o grupo."

John Nash (personagem), na cinebiografia Uma Mente Brilhante (2001). Inspirado no matemático homônimo, ganhador do Prêmio Nobel de Economia em 1994, ao apresentar o conceito de "equilíbrio de Nash".

8 Repositório do projeto no GitHub

O código-fonte e demais arquivos do projeto estão disponíveis publicamente no seguinte repositório:

https://github.com/FrankSteps/PSEL_CORETECH_2025

9 Referências

(JAKE BOX (ED.). **Perfect Emacs Org Mode Exports to LaTeX– Straightforward Emacs**. 7 Mar. 2021. Disponível em: <https://www.youtube.com/watch?v=0qHloGTT8XE>. Acesso em: 13 nov. 2024)