

UNIVERSIDADE DE SÃO PAULO
Escola Politécnica



Simulador Interativo Didático de ASM

PCS3732 – Laboratório de Processadores

Projeto Final

Professor Dr. Bruno Abrantes Basseto
Professor Dr. Marco Túlio Carvalho de Andrade

Grupo

Bruno Vieira de Alcântara
Henrique Fuga Duran
Matheus Junji Nakamura
Guilherme Rodrigues Bastos

Nº USP: 12551536
Nº USP: 12553570
Nº USP: 11795443
Nº USP: 10416851

SÃO PAULO, 04/07/2024

Introdução

A crescente demanda por profissionais capacitados em arquitetura de computadores e programação de baixo nível tem incentivado o desenvolvimento de ferramentas didáticas que auxiliem no ensino desses temas complexos. Entre essas ferramentas, os emuladores de instruções se destacam por proporcionar um ambiente interativo onde os alunos podem experimentar, depurar e entender o comportamento de um processador real sem a necessidade de hardware especializado.

A arquitetura ARM (Advanced RISC Machine) é amplamente utilizada em dispositivos móveis, sistemas embarcados e, mais recentemente, em servidores e computadores pessoais. Devido à sua popularidade e importância no mercado atual, compreender o funcionamento das instruções ARM é essencial para estudantes e profissionais da área de computação.

Este projeto tem como objetivo desenvolver um emulador didático para um subconjunto de instruções ARM, implementado em um framework web. A escolha de um framework web visa garantir uma ferramenta de fácil acesso e rápida utilização, permitindo que os usuários interajam com o emulador diretamente a partir de seus navegadores. O backend será desenvolvido utilizando FastAPI, um framework moderno e eficiente para criação de APIs em Python, enquanto o frontend será implementado com Angular, um framework robusto para a construção de interfaces de usuário dinâmicas.

Além de emular as instruções ARM, o projeto inclui a implementação de um tradutor de instruções para linguagem humana, facilitando a compreensão do código por parte dos usuários. Esse tradutor converterá as instruções ARM para descrições textuais mais intuitivas e adicionará rótulos aos registradores no front-end, tornando o código mais legível.

Através deste projeto, buscamos criar uma ferramenta educativa poderosa que possa ser utilizada tanto em ambientes acadêmicos quanto por entusiastas da programação, contribuindo para a formação de profissionais mais capacitados e preparados para enfrentar os desafios da computação moderna.

Revisão de Literatura

Histórico da Arquitetura ARM

A arquitetura ARM (Advanced RISC Machine) tem suas raízes na década de 1980, desenvolvida pela empresa britânica Acorn Computers. Inicialmente chamada de Acorn RISC Machine, a arquitetura foi criada como parte de um projeto para desenvolver um computador pessoal mais potente e eficiente. Em 1985, o primeiro processador ARM, o ARM1, foi fabricado. Este chip destacou-se por sua eficiência energética e desempenho, características que se tornariam marcas registradas da arquitetura ARM.

Em 1990, a ARM Limited foi fundada como uma joint venture entre Acorn, Apple e VLSI Technology. A empresa foi criada para desenvolver e licenciar a tecnologia ARM para outros fabricantes de chips. A partir daí, a arquitetura ARM começou a ganhar popularidade,

especialmente em dispositivos móveis e sistemas embarcados, devido à sua alta eficiência energética e desempenho satisfatório.

Com o passar dos anos, a arquitetura ARM evoluiu significativamente. A introdução das instruções Thumb, que permitiam um conjunto de instruções mais compacto, e o desenvolvimento do ARM Cortex, uma série de núcleos de processadores de alto desempenho, foram marcos importantes. Hoje, a ARM é uma das arquiteturas de processadores mais usadas no mundo, presente em smartphones, tablets, dispositivos IoT, e até mesmo em servidores e supercomputadores.

Descrição da Arquitetura ARM

A arquitetura ARM é baseada no conceito de RISC (Reduced Instruction Set Computer), que prioriza um conjunto de instruções simples e altamente otimizado, permitindo maior eficiência e desempenho. Entre suas principais características, encontram-se a simplicidade do conjunto de instruções, a utilização de *pipelining*, o baixo consumo de energia, a escalabilidade e a segurança. A combinação dessas características faz da arquitetura ARM uma escolha versátil e poderosa para uma ampla gama de aplicações, desde dispositivos de baixo custo e baixo consumo de energia até sistemas de alto desempenho.

Frameworks Web: FastAPI e Angular

A escolha dos frameworks web para o desenvolvimento de um emulador didático é crucial para garantir eficiência, performance e uma experiência de usuário agradável. Dois frameworks que se destacam nesse contexto são o FastAPI para o backend e o Angular para o frontend. A seguir, discutimos as características, vantagens e a adequação desses frameworks ao projeto.

FastAPI

FastAPI é um framework moderno para construção de APIs web com Python 3.7+ baseado em padrões como Python type hints. Ele é projetado para ser rápido (alto desempenho) e fácil de usar, oferecendo várias vantagens como facilidade de uso, desempenho e suporte a operações assíncronas, função essencial para aplicações que exigem alta performance e baixa latência, como emuladores interativos.

Angular

Angular é um framework de desenvolvimento front-end robusto, mantido pelo Google, que permite a criação de aplicações web dinâmicas e responsivas. Angular utiliza uma arquitetura baseada em componentes, permitindo a criação de interfaces modulares e reutilizáveis. Isso facilita a manutenção e a escalabilidade da aplicação. Desenvolvido em TypeScript, oferece recursos como tipagem estática, que ajuda na detecção precoce de erros e na melhoria da qualidade do código.

Além disso, Angular oferece várias otimizações de performance, incluindo a compilação ahead-of-time (AOT), que compila o código no momento da construção, resultando em tempos de carregamento mais rápidos e melhor performance. Essa ferramenta possui um ecossistema vasto com inúmeras bibliotecas e extensões disponíveis, devido a isso, se tornou uma escolha

popular criando uma comunidade ativa e extensa documentação, facilitando a resolução de problemas e a aprendizagem.

Metodologia

- **Arquitetura do sistema (frontend, backend, comunicação entre os componentes)**

Os seguintes comandos são necessários para se fazer o deploy do backend.

Instalação de dependências:

- `python3 -m pip install fastapi`
- `python3 -m pip install pygdbmi`
- `python3 -m pip install python-multipart`
- `python3 -m pip install fastapi[standard]`

Aplicações necessárias:

- `sudo apt install qemu-system-arm`
- `sudo apt install gdb-multiarch`

No diretório backend, roda-se o comando:

```
fastapi dev main.py
```

para rodar o servidor na porta 8000.

Para se executar o frontend, é preciso ter o npm instalado, e para se fazer o deploy de uma aplicação Angular usa-se os seguintes comandos:

```
npm install --save-dev @angular-devkit/build-angular
```

```
npm install @ctrl/ngx-codemirror codemirror@5
```

No diretório frontend/labproc-front é preciso fazer

```
ng serve
```

na linha de comando, e o frontend rodará na porta 4200.

- **Descrição detalhada do desenvolvimento do emulador**
 - **Implementação do backend com FastAPI**

O backend utiliza uma dependência que gerencia a interface do backend com o GDB rodando no sistema, o pygdbmi. Com isso, o QEMU-System-ARM e o GDB-Multiarch são executados “under-the-hood” e serão responsáveis pelo processamento do código.

Um problema enfrentado, típico de aplicações web, foi habilitar o CORS(Cross-Origin Resource Sharing) para permitir que o backend escutasse requisições do front.

Com isso, foram abertas diversas endpoints para atender às necessidades do emulador.

- compile: recebe o código assembly e repassa para o GDB
- next: salta para a próxima instrução e atualiza os registradores.
- back: reinicia a simulação.
- instruction: fornece a instrução atual para o tradutor.
- all: executa o programa atual até o fim.

Além disso, as endpoints foram modificadas para receber o “endereço-base” do front-end de modo a permitir que o gdb retorne o conteúdo de endereços de memória a partir desse endereço base. Assim, o usuário consegue monitorar o conteúdo, por exemplo, da pilha(\$sp) ou do fluxo de execução(\$pc) lendo os valores exibidos. Foi definido que poderiam ser vistos o conteúdo de 10 palavras após a base, inclusive, e que uma caixa de texto no frontend recebesse a entrada do usuário para o endereço base desejado.

- **Implementação do frontend com Angular**

No Angular, foi implementado um botão de submissão para que, com o envio de uma requisição post para o backend, um form-data contendo o conteúdo da caixa de texto com o código Assembly fosse submetido para o GDB, permitindo que o emulador receba o código e faça a compilação e o link.

Uma caixa de texto “a cada(ms)” e outro botão “passos periód.” permitem que o frontend faça requisições temporizadas sem intervenção do usuário, bastando que ele configure a duração dos passos e observe seu programa em funcionamento.

Finalmente, com o valor do registrador cpsr, dado extraído do backend, o front exibe o valor atual dos flags N(negativo), C(carry), Z(zero), e V(overflow).

- **Tradução de instruções para linguagem humana**

A tradução foi feita inteiramente no frontend, usando os recursos do Angular, permitindo que os “rótulos”, apelidos dados pelo usuário, se reflitam dinamicamente nas instruções ao longo da execução do programa. Por meio de uma lógica de segmentação das instruções, é possível converter as principais instruções do ARM em linguagem humana, a fim de facilitar a leitura do fluxo de execução.

- O mnemônico é a primeira palavra da instrução. Com base nele, são verificados os operandos envolvidos.
- Os operandos são substituídos por rótulos dos registradores, caso apropriado. Caso eles já sejam rótulos(para dados ou para instruções, por exemplo) ou constantes, permanecem inalterados.
- É verificado o caso especial de instruções com operações do barrel shifter, acrescentando-se sua descrição.
- Por fim, são analisadas as duas últimas letras do mnemônico para as instruções condicionais, adicionando-se suas condições à tradução.

Implementação

• Principais desafios e como foram superados

Após a definição do escopo do projeto, os principais obstáculos foram reconhecer as limitações do GDB, as quais inviabilizaram a implementação de diversas features anteriormente planejadas. Com isso, precisaram ser levantadas outras features úteis para uma interface, com foco em otimizar o conforto do usuário, descritas acima.

• Exemplos de código

Segue abaixo um exemplo de código utilizado para o vídeo de demonstração, que efetua o cálculo da sequência de Fibonacci. (Créditos: Lucas Veneziani Collevatti)

```
.global main
main:
    MOV R0, #10
    LDR R1, =fibonacci
    BL fill_fib_sequence
    B .
fill_fib_sequence:
    CMP R0, #0
    BEQ fill_fib_end
    MOV R2, #0
    STR R2, [R1], #4
    SUBS R0, R0, #1
    BEQ fill_fib_end
    MOV R2, #1
    STR R2, [R1], #4
    SUBS R0, R0, #1
    BEQ fill_fib_end
fill_fib_loop:
    LDR R3, [R1, #-8]
    LDR R4, [R1, #-4]
    ADD R2, R3, R4
    STR R2, [R1], #4
    SUBS R0, R0, #1
```

```
        BNE fill_fib_loop
fill_fib_end:
        BX LR
.data
fibonacci:
        .space 40
```

O outro exemplo de código faz a soma BCD e é de autoria do próprio grupo.

```
.global main
main:

mov r6, #0x90

mov r0, r6, lsl #24

mov r6, #0x69

mov r1, r6, lsl #24

mov r6, #0x88

add r0, r0, r6, lsl #16

mov r6, #0x71

add r1, r1, r6, lsl #16

mov r6, #0x43

add r0, r0, r6, lsl #8

mov r6, #0x46

add r1, r1, r6, lsl #8

add r0, r0, #0x00

add r1, r1, #0x99

bl soma_bcd

end:
b end

soma_bcd:

mov r5, #0
```

```
and r2, r0, #0x0000000f
and r3, r1, #0x0000000f
add r4, r2, r3
add r5, r5, r4
subs r4, r5, #0x0000000a
addpl r5, r5, #6
and r2, r0, #0x000000f0
and r3, r1, #0x000000f0
add r4, r2, r3
add r5, r5, r4
subs r4, r5, #0x000000a0
addpl r5, r5, #0x60
and r2, r0, #0x00000f00
and r3, r1, #0x00000f00
add r4, r2, r3
add r5, r5, r4
subs r4, r5, #0x00000a00
addpl r5, r5, #0x600
and r2, r0, #0x0000f000
and r3, r1, #0x0000f000
add r4, r2, r3
add r5, r5, r4
subs r4, r5, #0x0000a000
addpl r5, r5, #0x6000
and r2, r0, #0xf0000
and r3, r1, #0xf0000
```



```
add r4, r2, r3
add r5, r5, r4
subs r4, r5, #0xa0000
addpl r5, r5, #0x60000
and r2, r0, #0xf00000
and r3, r1, #0xf00000
add r4, r2, r3
add r5, r5, r4
subs r4, r5, #0xa00000
addpl r5, r5, #0x600000
and r2, r0, #0xf000000
and r3, r1, #0xf000000
add r4, r2, r3
add r5, r5, r4
subs r4, r5, #0xa000000
addpl r5, r5, #0x6000000
and r2, r0, #0xf0000000
and r3, r1, #0xf0000000
adds r4, r2, r3
addcs r4, #0x60000000
adds r5, r5, r4
addcs r4, #0x60000000
subs r4, r5, #0xa0000000
addpl r5, r5, #0x60000000
mov r15, r14
```

- Capturas de tela da interface

LabprocFront

localhost:4200

Código

```
.global main
main:

mov r6, #0x90

mov r0, r6, lsl #24

mov r6, #0x69

mov r1, r6, lsl #24

mov r6, #0x88

add r0, r0, r6, lsl #16

mov r6, #0x71

add r1, r1, r6, lsl #16

mov r6, #0x43

add r0, r0, r6, lsl #8

mov r6, #0x46

add r1, r1, r6, lsl #8
```

Compilar

→ Passo periód.

→ Passo

→ Executar

Registradores

Nome	Hex	Dec
r0	0x90884300	2424849152
r1	0x69710000	1769013248
r2	0x0	0
r3	0x0	0
r4	0x0	0
r5	0x0	0
r6	0x46	70
r7	0x0	0
r8	0x0	0
r9	0x0	0
r10	0x0	0
r11	0x0	0
r12	0x0	0
sp	0x40002148	1073750344
lr	0x0	0
pc	0x40000064	1073741924
cpsr	0x600001d3	1610613203

Instrução

add r1, r1, r6, lsl #8
Armazenar em x x mais y com 8 shift(s) à esquerda.
Negativo: 0
Zero: 1
Carry: 1
Overflow: 0
0x40002148: 0x00000000 0x00000000 0x00000000
0x00000000 0x00000000 0x00000000 0x00000000
0x00000000 0x00000000 0x00000000
bar
x
foo
r3
r4
r5
y
r7
r8
r9
r10
r11
r12
A cada(ms):1000
Ver memória a partir de: \$sp

LabprocFront

localhost:4200

Código

```
.global main
main:

mov r6, #0x90

mov r0, r6, lsl #24

mov r6, #0x69

mov r1, r6, lsl #24

mov r6, #0x88

add r0, r0, r6, lsl #16

mov r6, #0x71

add r1, r1, r6, lsl #16

mov r6, #0x43

add r0, r0, r6, lsl #8

mov r6, #0x46

add r1, r1, r6, lsl #8
```

Compilar

→ Passo periód.

→ Passo

→ Executar

Registradores

Nome	Hex	Dec
r0	0x90884300	2424849152
r1	0x69714699	1769031321
r2	0x90000000	2415919104
r3	0x60000000	1610612736
r4	0x60598999	1616480665
r5	0x60598999	1616480665
r6	0x46	70
r7	0x0	0
r8	0x0	0
r9	0x0	0
r10	0x0	0
r11	0x0	0
r12	0x0	0
sp	0x40002148	1073750344
lr	0x40000074	1073741940
pc	0x40000074	1073741940
cpsr	0x1d3	467

Instrução

mov r6, #0x90
Armazenar em r6 a constante #0x90
Negativo: 0
Zero: 1
Carry: 1
Overflow: 0
0x40002148: 0x00000000 0x00000000 0x00000000
0x00000000 0x00000000 0x00000000 0x00000000
0x00000000 0x00000000 0x00000000
r0
r1
r2
r3
r4
r5
r6
r7
r8
r9
r10
r11
r12
A cada(ms):1000
Ver memória a partir de: \$sp

Conclusão e encaminhamento

Encerramos esse trabalho comentando os principais ensinamentos que pudemos extrair ao longo do seu desenrolar. Em especial, gostaríamos de pontuar o potencial que os paradigmas de desenvolvimento web oferece para criar aplicações rápidas e responsivas sobre o GDB, um aplicativo extremamente poderoso que nasceu junto com o projeto GNU de software livre e que é essencial até hoje para programadores.

Além disso, seguem algumas propostas de encaminhamento que podem ser melhor desenvolvidas em um projeto futuro.

- Simulações envolvendo coprocessador;
- Simulações envolvendo interação com o Raspberry, incluindo comunicação serial e via LEDs, com displays das formas de onda;
- Tradutores de instruções ARM para outras linguagens assembly;
- Implementação de suporte a programas em C;
- Funcionalidade de retorno de instruções(go back), previsão de overflow e loop eterno. Isso exigiria uma lógica complexa de controle e armazenamento de registradores e posições de memória, e possivelmente exigiria uma grande quantidade de tempo para implementação com os recursos do GDB.

Referências

<https://www.arm.com>

ARM System-on-Chip Architecture, Furber, Stephen B. (2000)

CMOS VLSI Design: A Circuits and Systems Perspective, Weste, Neil H.E., e Harris, David (2010)

<https://medium.com/swlh/advances-in-arm-what-it-could-mean-to-the-future-of-computing-2e76417bbfe7>

<https://github.com/fastapi/fastapi>

<https://medium.com/coderhack-com/introduction-to-fastapi-c31f67f5a13>

<https://angular.dev/overview>