

ESCOLA POLITÉCNICA DA UNIVERSIDADE DE SÃO PAULO



Relatório de Projeto - ECVA1.2

PCS 3732 - Laboratório de Processadores

Antonio Nigro Zamur - 10696510

João Victor Araújo e Rocha de Carvalho- 10773102

São Paulo

2025

Relatório de Projeto - ECVA1.2	1
Introdução	3
converter.py	4
Main.s	6
Makefile	7
Bônus - player.py	8
Desenvolvimento	9
GIT	10
Resultados obtidos e conclusão	11

Introdução

Este relatório apresenta o projeto final desenvolvido para a disciplina de Laboratório de Processadores, ECVA1.2, que consiste na implementação de um reproduutor de áudio em um ambiente bare-metal para a plataforma Raspberry Pi 2B. O desafio central do projeto é orquestrar o hardware do microprocessador, sem o auxílio de um sistema operacional, para decodificar e reproduzir um sinal sonoro com fidelidade.

Para alcançar este objetivo, o sistema foi arquitetado em torno do mecanismo de interrupções de hardware. O periférico System Timer do Raspberry Pi é configurado para gerar interrupções (IRQ) a uma frequência precisa de 44.100 Hz, correspondente à taxa de amostragem padrão de áudio. A cada interrupção, o fluxo de execução do processador é desviado para uma Rotina de Serviço de Interrupção (ISR) customizada.

Dentro da ISR, a lógica de reprodução é executada: uma amostra de áudio de 8 bits, pré-processada e embutida no kernel, é lida da memória. Este valor, que representa a amplitude do som em um dado instante, é então utilizado para modular a largura de um pulso em um pino de saída GPIO. Esta técnica de Modulação por Largura de Pulso (PWM) é implementada via software através de um loop de espera ("busy-wait"), onde a duração do pulso em estado alto é diretamente proporcional ao valor da amostra lida. A sucessão rápida desses pulsos, a 44.100 Hz, cria a onda analógica que, ao ser amplificada, se torna o som audível. O projeto abrange desde o processamento do arquivo de áudio em um ambiente de desenvolvimento até a programação em baixo nível do hardware para controle de tempo e periféricos.

converter.py

O ponto de partida para a digitalização e preparação do áudio é o script `converter.py`, uma componente essencial do pipeline de compilação. Este script é responsável por uma série de transformações críticas que convertem um arquivo de áudio padrão (como `.mp3` ou `.wav`), localizado num diretório de entrada `input`, nos dados brutos que o kernel bare-metal irá consumir.

O processo de conversão é executado por esses estágios:

- **Carregamento e Normalização:** Utilizando a biblioteca `pydub`, o script primeiramente carrega o arquivo de áudio. Para garantir consistência, o áudio é padronizado: é convertido para um único canal (mono), ajustado para uma taxa de amostragem de 44.100 Hz e a sua amplitude é normalizada. A normalização assegura que o sinal de áudio utilize toda a gama dinâmica disponível, maximizando a qualidade do som.
- **Conversão para PWM:** O formato de áudio padrão, PCM (Modulação por Código de Pulso), representa a amplitude com valores positivos e negativos. Para o nosso uso, este formato é convertido para uma representação de Modulação por Largura de Pulso (PWM) de 8 bits. Neste processo, cada amostra de áudio é mapeada para um valor inteiro sem sinal no intervalo de 0 (silêncio) a 255 (amplitude máxima), que correspondem aos valores hexadecimais de `0x00` a `0xFF`.
- **Geração de pastas de Saída:** Ao contrário de gerar um único arquivo de texto gigante, o que se provou ineficiente para o compilador, o script adota uma abordagem mais robusta, criando dois arquivos distintos num diretório de saída `output`:
- **Um arquivo binário (`.bin`):** Este arquivo contém os dados PWM puros. É uma representação compacta e eficiente da música, consistindo numa sequência contínua de 44.100 bytes para cada segundo de áudio processado.
- **Um arquivo Assembly Wrapper (`.s`):** Este é um pequeno arquivo de texto que serve como um "manual de instruções" para o compilador. Ele contém a diretiva `.incbin` ("include binary"), que instrui o montador a ler o conteúdo completo do ficheiro `.bin` e a incorporá-lo diretamente no kernel final. Além disso, este ficheiro define dois labels globais essenciais: um que aponta para o início dos dados da música e outro (`_len`) que armazena o seu tamanho total em bytes.

Esta estratégia de dois arquivos é fundamental para o projeto, pois permite a inclusão de grandes volumes de dados no kernel de forma extremamente eficiente, contornando as limitações de memória e tempo do processo de compilação. O

resultado final é um conjunto de dados perfeitamente formatado e acessível pelo código Assembly que será executado no Raspberry Pi.

Main.s

O arquivo main.s é o núcleo do kernel bare-metal, responsável por toda a lógica de baixo nível do sistema. Sua estrutura começa com a Tabela de Vetores de Exceção, posicionada no endereço de memória 0x8000, que serve como ponto de entrada do processador. Esta tabela direciona o evento de Reset para a rotina de inicialização e, crucialmente, o vetor de Interrupção (IRQ) para o nosso código de tratamento de áudio.

A rotina de inicialização (start_code) é executada uma única vez para preparar o ambiente. Ela configura o pino GPIO 18 como saída, carrega os endereços de memória dos dados da música (gerados pelo converter.py) e ativa o sistema de interrupções, programando o System Timer para disparar o primeiro evento. Após a inicialização, o processador entra em um loop de espera, transferindo o controle do programa para o mecanismo de interrupções.

O componente central do sistema é a Rotina de Serviço de Interrupção (irq_handler). Para atingir a taxa de amostragem de 44.100 Hz, o System Timer é programado para gerar uma interrupção a cada 22 microssegundos – uma aproximação do intervalo ideal de 22,675 μ s, resultando em uma aceleração de 3,07% na reprodução, que é imperceptível. A cada ciclo de 22 μ s, o irq_handler executa as seguintes tarefas:

- Lê o próximo byte da tabela de dados PWM da música.
- Implementa um PWM via software ("busy-wait"), mantendo o pino GPIO 18 em estado alto por um período proporcional ao valor PWM lido.
- Avança o ponteiro para a próxima amostra de áudio, reiniciando a música ao seu final.
- Se atualiza, instruindo o System Timer a gerar a próxima interrupção, perpetuando assim o ciclo de reprodução de áudio.

Makefile

O Makefile é o gerenciador de todo o processo de compilação, automatizando a transformação dos arquivos de código-fonte no arquivo de imagem final `kernel7.img`. A sua operação é baseada em dependências, garantindo que apenas os componentes necessários sejam reconstruídos a cada alteração.

O processo inicia com o objetivo de construir o kernel. Para isso, o Makefile primeiro chama o script `converter.py` para processar o arquivo de áudio, resultando na geração de um arquivo binário (`.bin`) com os dados PWM e um arquivo Assembly (`.s`) que o referencia via o comando `.incbin`. Em paralelo, o Makefile aciona o montador (`arm-none-eabi-as`) para compilar tanto o `main.s` quanto o arquivo Assembly da música, gerando os respectivos arquivos objeto (`.o`).

Na etapa de ligação, o linker (`arm-none-eabi-ld`) é chamado, utilizando um script de linker (`linker.ld`) para organizar corretamente na memória o conteúdo de todos os arquivos objeto — incluindo os dados da música que já foram embutidos pelo montador. Este processo se finaliza com a criação de um arquivo no formato ELF (`kernel.elf`). Finalmente, a ferramenta `objcopy` ("dumper") extrai apenas o código de máquina puro do arquivo ELF, gerando o `kernel7.img` final, pronto para ser carregado na RAM do Raspberry Pi.

Bônus - player.py

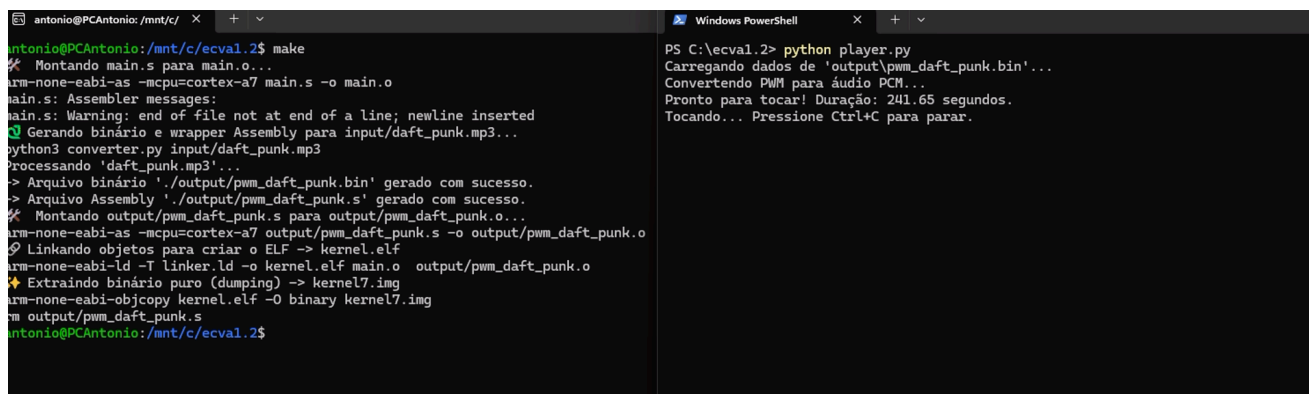
Durante o projeto, uma dúvida sempre aparecia: como saber se a conversão do som para dados binários está funcionando direito, antes de todo o trabalho de gravar o kernel no Raspberry Pi? A resposta foi criar uma ferramenta de verificação, o player.py.

A ideia é bem direta. O nosso converter.py pega uma música e, metaforicamente, a "mói", transformando-a numa sequência de números de amplitude (os dados PWM). Mas como saber se essa moagem não estragou o som? Como saber se aqueles números todos em binário realmente são um áudio? É aí que entra o player.py. Ele pega esses números e tenta "grudar" tudo de volta para ver se ainda parece com a música original.

O script funciona em três passos simples:

- Ele abre o arquivo .bin que o make gerou, que é só um monte de números de 0 a 255.
- Ele faz a matemática inversa, convertendo essa sequência de números de volta para um formato de áudio que a placa de som do computador entende.
- Finalmente, ele usa a biblioteca sounddevice para tocar o resultado diretamente nos alto-falantes.

Criar este player foi muito gratificante. Com ele, podíamos ouvir o resultado da conversão e ter certeza de que o "pó" de números gerado pelo converter.py ainda continha a essência da música. Foi o nosso controle de qualidade, permitindo ouvir e validar o converter.



```
antonio@PCAntonio: /mnt/c/ecval.2$ make
% Montando main.s para main.o...
arm-none-eabi-as -mcpu=cortex-a7 main.s -o main.o
main.s: Assembler messages:
main.s: Warning: end of file not at end of a line; newline inserted
% Gerando binário e wrapper Assembly para input/daft_punk.mp3...
python3 converter.py input/daft_punk.mp3
Processando 'daft_punk.mp3'...
-> Arquivo binário './output/pwm_daft_punk.bin' gerado com sucesso.
-> Arquivo Assembly './output/pwm_daft_punk.s' gerado com sucesso.
% Montando output/pwm_daft_punk.s para output/pwm_daft_punk.o...
arm-none-eabi-as -mcpu=cortex-a7 output/pwm_daft_punk.s -o output/pwm_daft_punk.o
% Linkando objetos para criar o ELF -> kernel.elf
% Linkando objetos para criar o ELF -> kernel.elf
arm-none-eabi-ld -T linker.ld -o kernel.elf main.o output/pwm_daft_punk.o
% Extraíndo binário puro (dumping) -> kernel7.img
arm-none-eabi-objcopy kernel.elf -O binary kernel7.img
rm output/pwm_daft_punk.s
antonio@PCAntonio: /mnt/c/ecval.2$
```

```
PS C:\ecval.2> python player.py
Carregando dados de 'output/pwm_daft_punk.bin'...
Convertendo PWM para áudio PCM...
Pronto para tocar! Duração: 241.65 segundos.
Tocando... Pressione Ctrl+C para parar.
```

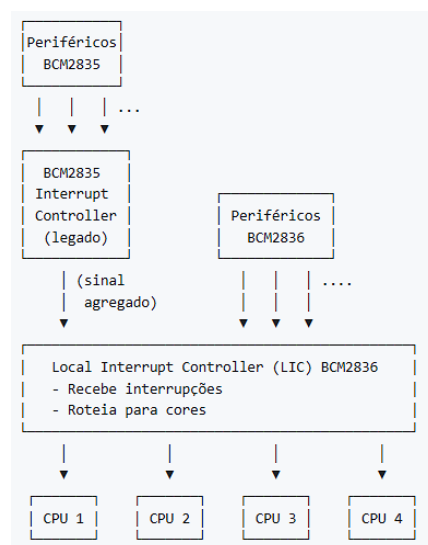

Desenvolvimento

Durante o desenvolvimento do projeto, o principal desafio técnico encontrado foi a interação com o sistema de interrupções, timers e clocks do Raspberry Pi 2B em um ambiente estritamente bare-metal. Pela documentação de hardware disponível na internet, foram investigados múltiplos periféricos de temporização como possíveis fontes de clock para o nosso sistema, incluindo o ARM Timer, o System Timer e o Local Timer de cada núcleo.

A dificuldade central residiu na complexidade, na ausência e, por vezes, na ambiguidade da documentação a respeito da arquitetura de interrupções do Raspberry Pi 2B. Este modelo de hardware combina periféricos do SoC BCM2835 (como o System Timer e o controlador de interrupções principal) com a arquitetura multi-core do BCM2836 (Cortex-A7), que possui seu próprio Local Interrupt Controller, além do GIC-400. Estabelecer a comunicação correta entre o periférico de temporização, o controlador de interrupções e o núcleo do processador provou ser um obstáculo significativo, exigindo um entendimento profundo do mapeamento de memória e dos registradores de controle específicos.

Como resultado, a cadeia de ferramentas de software do projeto foi desenvolvida e validada com completo sucesso. O script `converter.py`, o Makefile automatizado e a ferramenta de verificação `player.py` (que se tornou um bônus valioso ao projeto) estão totalmente funcionais, realizando o processamento do áudio e a compilação do kernel de forma robusta e eficiente.

No entanto, o arquivo `main.s` ficou com sua funcionalidade comprometida. Mesmo após um extenso processo de experimentação e depuração, não foi possível estabelecer uma rotina de interrupção funcional a partir dos clocks de hardware. Foram gastas dezenas de horas na busca por uma solução para esse problema, sem êxito.




GIT


Todo o processo de desenvolvimento e pesquisa do projeto foi documentado em nosso repositório Git. Ele serve como um registro da evolução do código, dos desafios encontrados e das soluções investigadas ao longo das semanas.


O repositório é particularmente detalhado no que diz respeito à questão central do contratempo do projeto: a busca por uma solução funcional para o sistema de clocks e interrupções. Estão documentadas as diversas alternativas pesquisadas para os temporizadores (ARM Timer, System Timer, Local Timer), as discussões sobre a arquitetura de interrupções e os múltiplos testes realizados.

Além de conter as versões finais dos códigos (Makefile, converter.py, player.py, linker.ld e a estrutura do main.s), assim como códigos de diversos testes incrementais. O repositório também armazena arquivos esquemáticos, notas de pesquisa e outros textos desenvolvidos que demonstram o trabalho realizado.

 registro_experimentos

 registro_teorico

 src

 testes_incrementais

Resultados obtidos e conclusão

O desenvolvimento deste projeto resultou na criação de uma cadeia de ferramentas de software capaz de realizar uma tarefa complexa de processamento de sinal. Foi alcançado com sucesso o objetivo de criar um sistema automatizado (Makefile) que processa um arquivo de áudio (converter.py), converte-o para um formato de dados PWM binário, e o compila e linca com um código Assembly, gerando um arquivo de imagem de kernel (kernel7.img) corretamente estruturado para a arquitetura do Raspberry Pi 2B. A validação acústica dos dados gerados, através da ferramenta player.py, confirmou a integridade e o sucesso de todo o pipeline de processamento.

No entanto, o objetivo final de reproduzir o áudio no hardware não foi alcançado. O obstáculo intransponível encontrado foi a incapacidade de configurar uma rotina de interrupções funcional a partir dos periféricos de temporização do Raspberry Pi. A complexidade da interação entre os componentes dos SoCs BCM2835 e BCM2836, a dualidade do LIC e do GIC aliada à documentação que se mostrou ambígua para a programação em nível de registrador, impediu a criação de um sinal de clock que disparasse a rotina de serviço de interrupção na frequência desejada.

Conclui-se que, embora a lógica de processamento de dados e a arquitetura do kernel tenham sido implementadas com sucesso, a complexidade da interação com o hardware de baixo nível da plataforma escolhida provou ser um desafio maior do que o previsto. A nossa experiência evidencia não só a dificuldade inerente à programação bare-metal sem o suporte de abstrações de um sistema operacional, mas também que o escopo de interagir diretamente com a arquitetura de interrupções do Raspberry Pi 2B pode ter sido excessivamente ambicioso para o tempo e os recursos humanos e de documentação disponíveis. O projeto, no entanto, foi um sucesso em demonstrar e validar uma complexa cadeia de compilação cruzada e processamento de dados para um sistema embarcado.