# Compiladores

## Aula 27: WebAssembly (Wasm)

Prof. Aléssio Miranda Júnior
alessio@cefetmg.br

CEFET-MG - Campus Timoteo
Dep. Engenharia de Computação

Fevereiro de 2026

# Sumário

# O Problema do Desempenho na Web

O JavaScript foi criado em 10 dias em 1995. Apesar das incríveis engines JIT modernas (V8, SpiderMonkey) terem revolucionado seu desempenho, a Web cresceu e passou a exigir muito mais.

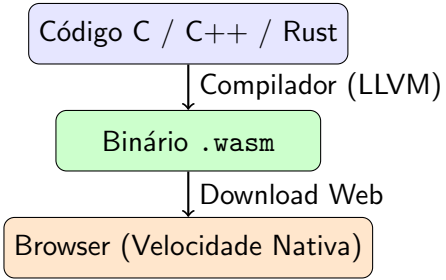
## As limitações da "Web Pesada"

- **Imprevisibilidade:** O *Garbage Collector* (Coletor de Lixo) do JS pausa a execução periodicamente e pode "engasgar" um jogo 3D de repente.
- **Custo de Parsing:** Baixar 5 MB de texto Javascript, transformá-lo em AST e passar por todos os níveis do JIT atrasa a abertura de grandes sites em vários segundos.
- **Código Legado:** Empresas precisam portar motores gráficos massivos em C/C++ inteiros para o navegador sem reescrever nada.

# A Chegada do WebAssembly (Wasm)

Anunciado em 2015 pelas grandes *Big Techs*, o **WebAssembly (Wasm)** não é uma linguagem para ser escrita por humanos, mas sim um **alvo universal de compilação** projetado para a Web.

- **Formato Binário:** Compacto e ultrarrápido de baixar.
- **Execução a $\approx$ 100% nativa:** Não tem *Garbage Collector* obrigatório, permite o gerenciamento manual e letal de memória (como em C).
- **Tempo de Abertura ("Parsing"):** Quase zero. O motor do navegador começa a compilar para nativo em *streaming* enquanto ainda está baixando o arquivo da internet!



# Arquitetura Conceitual: Uma Máquina de Pilha

Diferente das CPUs reais de *hardware* (como x86, ARM) que usam dezenas de registradores físicos isolados, o WebAssembly foi projetado para funcionar em cima de uma **Máquina de Pilha (Stack Machine)** puramente conceitual.

**Por que focar em Pilhas?**

- **Compressão máxima:** Instruções de pilha não precisam gastar bytes declarando de onde lerão e para onde gravarão. Tudo ocorre magicamente no topo da pilha atual. O binário fica enxuto.
- **Validação de Segurança:** É muito mais rápido para o *Browser* auditar e certificar uma pilha antes da execução do que auditar registradores arbitrários.

# Arquitetura Conceitual: Uma Máquina de Pilha

Diferente das CPUs reais de *hardware* (como x86, ARM) que usam dezenas de registradores físicos isolados, o WebAssembly foi projetado para funcionar em cima de uma **Máquina de Pilha (Stack Machine)** puramente conceitual.

**Por que focar em Pilhas?**

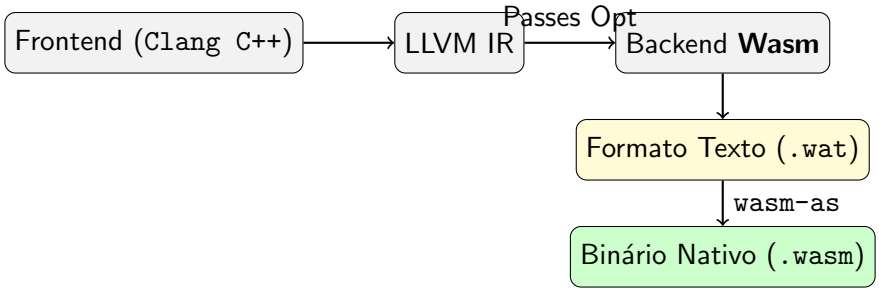
- **Compressão máxima:** Instruções de pilha não precisam gastar bytes declarando de onde lerão e para onde gravarão. Tudo ocorre magicamente no topo da pilha atual. O binário fica enxuto.
- **Validação de Segurança:** É muito mais rápido para o *Browser* auditar e certificar uma pilha antes da execução do que auditar registradores arbitrários.

**O Processo de Soma (a = 5 + 10)**

```
i32.const 5  // *Push* de 5 na pilha
i32.const 10 // *Push* de 10 na pilha
i32.add      // Pop(2x), soma, *Push* de 15
local.set 0  // *Pop* do 15 -¿ Grava em var 0
```

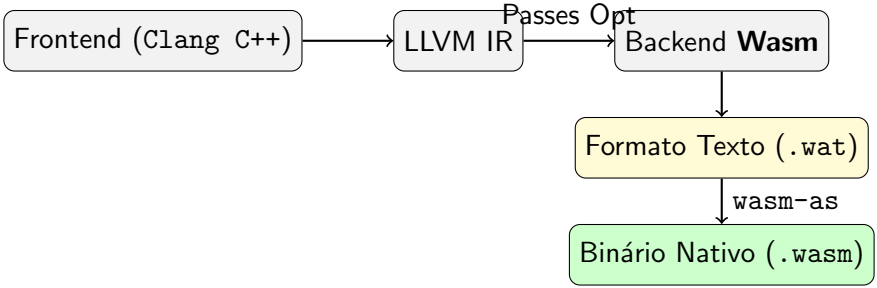
# O Pipeline de Compilação (O triunfo do LLVM)

A explosão do WebAssembly aconteceu porque o LLVM o adotou silenciosamente como um de seus *Back-ends* oficiais, conectando-o imediatamente ao C, C++, Rust e Swift de graça!



# O Pipeline de Compilação (O triunfo do LLVM)

A explosão do WebAssembly aconteceu porque o LLVM o adotou silenciosamente como um de seus *Back-ends* oficiais, conectando-o imediatamente ao C, C++, Rust e Swift de graça!




O formato **WAT** (*WebAssembly Text Format*) é uma representação em S-Expressions (como no Lisp) projetada exclusivamente para nós humanos conseguirmos ler, inspecionar e *debugar* código binário da Web (ao apertar F12).

# O Modelo de Isolamento de Memória (Linear Memory)

O Wasm é gerado por `C/C++`, linguagens notórias por usar Ponteiros invasivos. O navegador Chrome permitiria um ponteiro `C++` acessar memória livremente? Não, as falhas de segurança destruiriam a Web!

Para contornar isso, os navegadores criaram o modelo estrito de **Memória Linear**:

# O Modelo de Isolamento de Memória (Linear Memory)

O Wasm é gerado por `C/C++`, linguagens notórias por usar Ponteiros invasivos. O navegador Chrome permitiria um ponteiro `C++` acessar memória livremente? Não, as falhas de segurança destruiriam a Web!

Para contornar isso, os navegadores criaram o modelo estrito de **Memória Linear**:

- Consiste em um **Vetor Gigante (Array de Bytes contíguos)** que começa do índice 0 e vai até onde a RAM foi liberada.

# O Modelo de Isolamento de Memória (Linear Memory)

O Wasm é gerado por `C/C++`, linguagens notórias por usar Ponteiros invasivos. O navegador Chrome permitiria um ponteiro `C++` acessar memória livremente? Não, as falhas de segurança destruiriam a Web!

Para contornar isso, os navegadores criaram o modelo estrito de **Memória Linear**:

- Consiste em um **Vetor Gigante (Array de Bytes contíguos)** que começa do índice 0 e vai até onde a RAM foi liberada.
- Quando um programa C++ aponta e lê o endereço "NULL" ou o endereço 256, ele está acessando o **índice de vetor 256**. Este endereço é traduzido em segurança pela CPU e interpretado internamente.

# O Modelo de Isolamento de Memória (Linear Memory)

O Wasm é gerado por `C/C++`, linguagens notórias por usar Ponteiros invasivos. O navegador Chrome permitiria um ponteiro `C++` acessar memória livremente? Não, as falhas de segurança destruiriam a Web!

Para contornar isso, os navegadores criaram o modelo estrito de **Memória Linear**:

- Consiste em um **Vetor Gigante (Array de Bytes contíguos)** que começa do índice 0 e vai até onde a RAM foi liberada.
- Quando um programa C++ aponta e lê o endereço "NULL" ou o endereço 256, ele está acessando o **índice de vetor 256**. Este endereço é traduzido em segurança pela CPU e interpretado internamente.
- **Isolamento Total (Sandboxing):** Fica matematicamente impossível do código *hackeado* invadir abas vizinhas e roubar senhas do usuário. O limite superior de memória do aplicativo está sempre preso e contido dentro deste vetor.

# A Revolução Serverless: O Nascimento do WASI

Ao passarem anos construindo um executável ultra-seguro contra hackers, extremamente minúsculo em KiloBytes, e que atinge velocidades nativas, arquitetos e corporações notaram um paradoxo: *"Por que não usamos isso rodando fora dos navegadores, lá nos servidores em nuvem?"*

# A Revolução Serverless: O Nascimento do WASI

Ao passarem anos construindo um executável ultra-seguro contra hackers, extremamente minúsculo em KiloBytes, e que atinge velocidades nativas, arquitetos e corporações notaram um paradoxo: *"Por que não usamos isso rodando fora dos navegadores, lá nos servidores em nuvem?"*

## O nascimento do WASI (WebAssembly System Interface)

- O **WASI** é um novo padrão aberto mundial que fornece as permissões e funções restritas de Sistema Operacional (Escrever em Arquivo, Acessar IP de Rede) para programas isolados em WebAssembly.
- O Wasm migrou do Chrome e hoje substitui contêineres Docker robustos!
- **Escalabilidade Extrema:** Um aplicativo de Banco em Java/Docker pode pesar pesados 400 MB e levar 3 segundos para carregar. O correspondente em Wasm compila para incríveis 2 MB e "boota" nativamente em milissegundos num *Microserviço Serverless*.

# Conclusão da Disciplina de Compiladores

- O WebAssembly **não** vai matar o JavaScript. Eles trabalham juntos: o JS domina o *Front-end* (React, Eventos, UI, DOM) enquanto o Wasm devora pesados cálculos brutos, algoritmos massivos, IA local ou editores visuais potentes.
- Graças ao conceito espetacular do *Back-end* modular do **LLVM**, hoje transferimos toda a infraestrutura complexa (*Parser, Opt, Reg Alloc*) para qualquer dispositivo instantaneamente.
- O compilador tradicional de *desktop* (GCC/Clang) transformou-se no arquiteto mestre que exporta tecnologia para a Web livre e para a infraestrutura dos servidores nas Nuvens do futuro.

# Duvidas?

**Obrigado!**

**Email:** alessio@cefetmg.br