Web Assembly

Exporting Lyra to the web

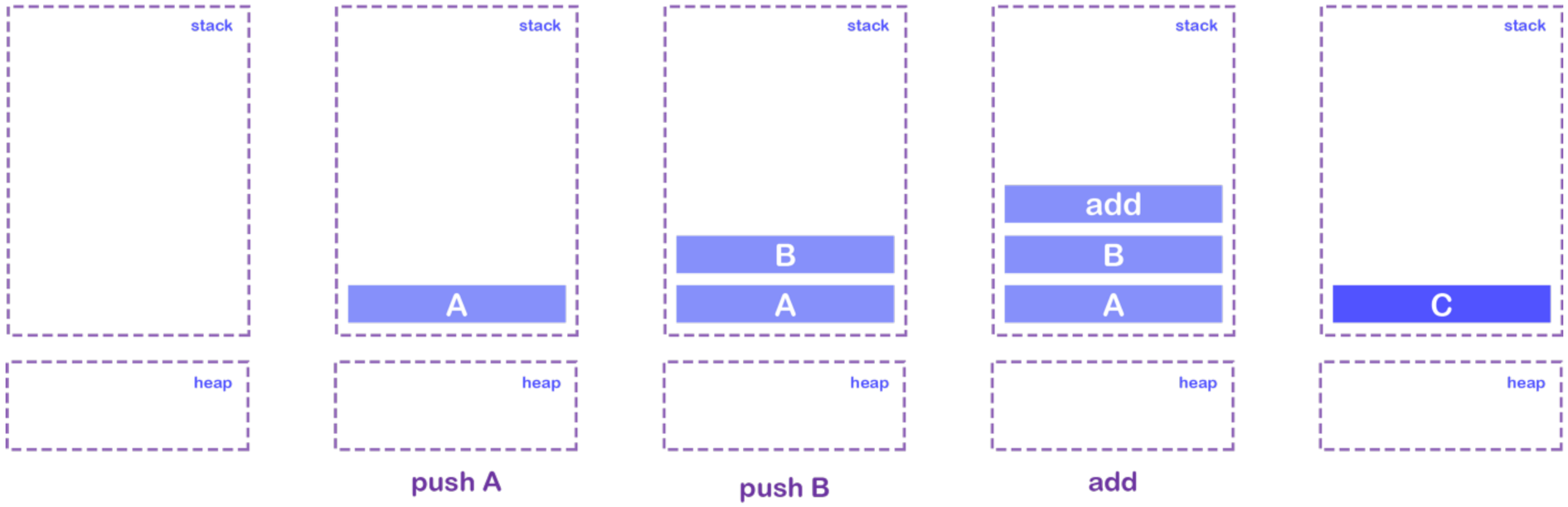
What is Web Assembly?

- A way to run programming languages other than JavaScript on web pages.

- A binary format of instructions just like a machine code but for a stack machine.

Stack Machine

- A virtual machine that takes one instruction at a time and pushes it on the stack.
- “add” knows that it needs 2 operands. => Instruction can be short, because it doesn’t need to specify registers.

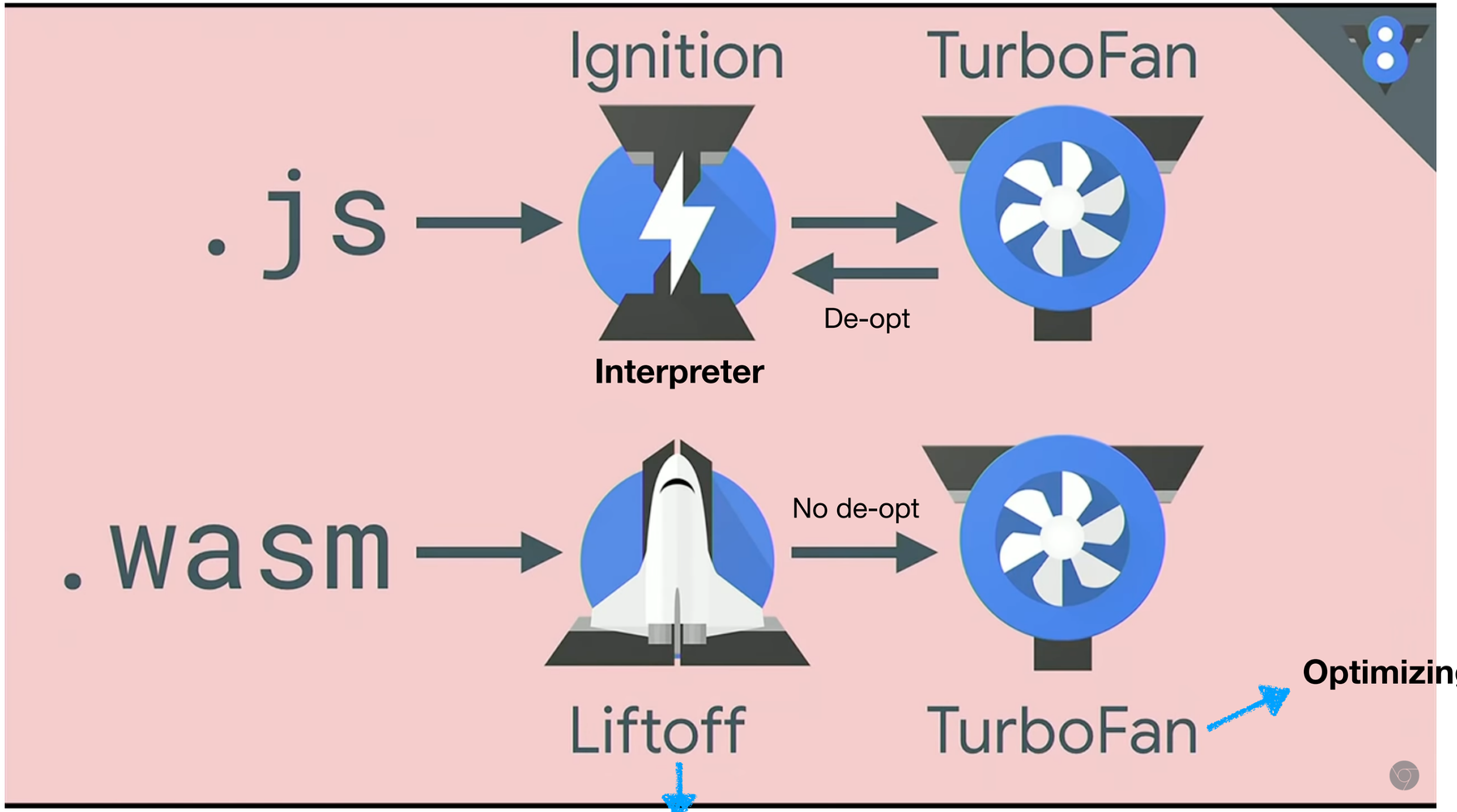


Stack Machine (rudimentary)

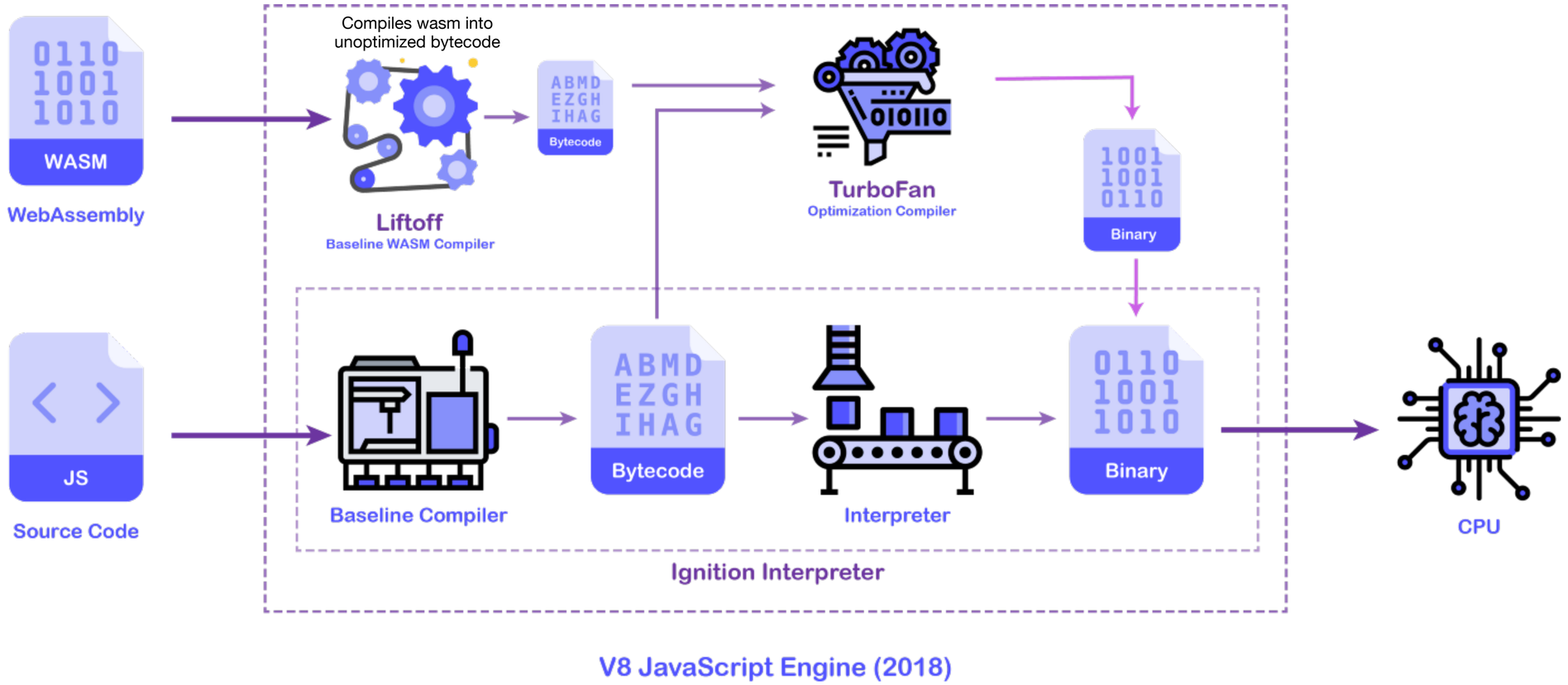
JS vs. WASM

- Their peak performance(speed) are the same.
- It is hard to reach peak performance using JS.
- Peak performance is easily reached with WASM. (Consistent & Predictable)

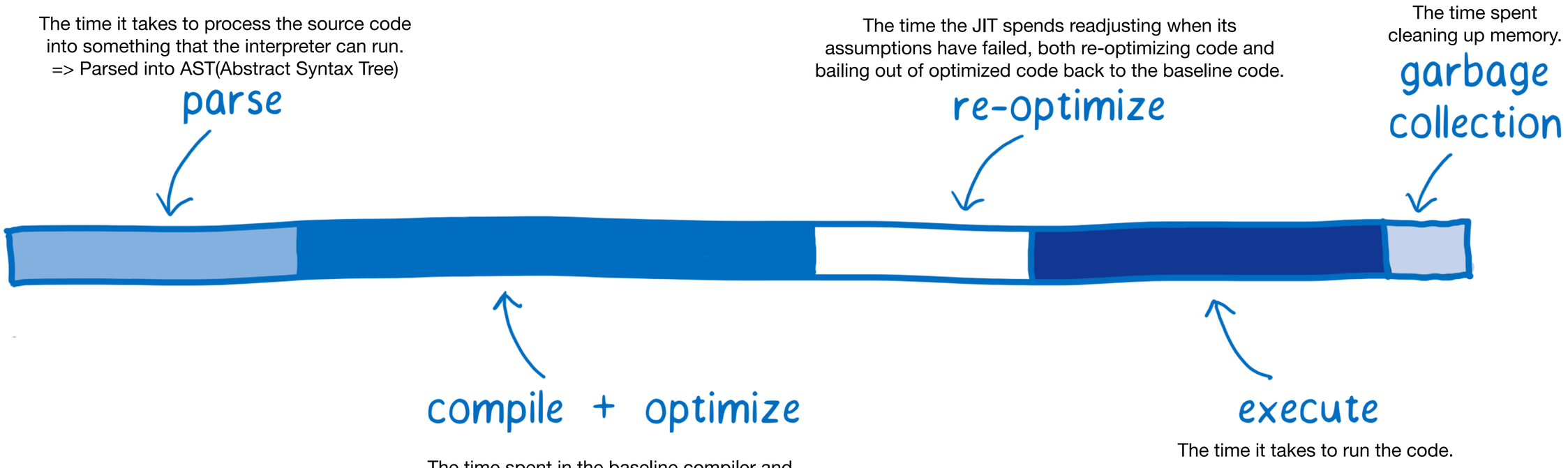
JS vs. WASM


Ignition
TurboFan
.js
De-opt
Interpreter
.wasm
No de-opt
Liftoff
TurboFan
Optimizing Compiler
Compiler that generates
machine codes

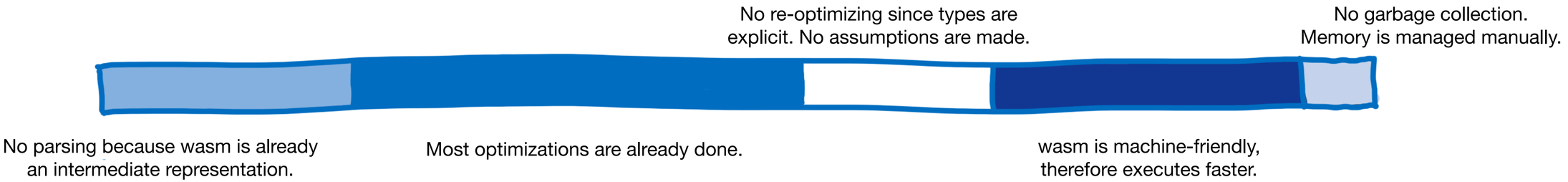
JS vs. WASM

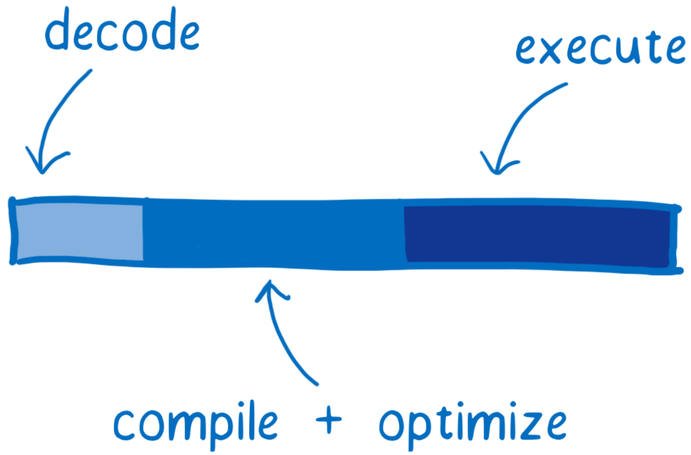

0110
1001
1010
WASM
WebAssembly
Compiles wasm into
unoptimized bytecode
ABMD
EZGH
IHAG
Bytecode
Liftoff
Baseline WASM Compiler
TurboFan
Optimization Compiler
1001
1001
0110
Binary
JS
Source Code
Baseline Compiler
ABMD
EZGH
IHAG
Bytecode
Interpreter
0110
1001
1010
Binary
CPU
Ignition Interpreter
V8 JavaScript Engine (2018)

JavaScript


The time it takes to process the source code into something that the interpreter can run.
=> Parsed into AST(Abstract Syntax Tree)
parse
The time the JIT spends readjusting when its assumptions have failed, both re-optimizing code and bailing out of optimized code back to the baseline code.
re-optimize
The time spent cleaning up memory.
garbage collection
compile + optimize
The time spent in the baseline compiler and optimizing compiler.
execute
The time it takes to run the code.

Web Assembly

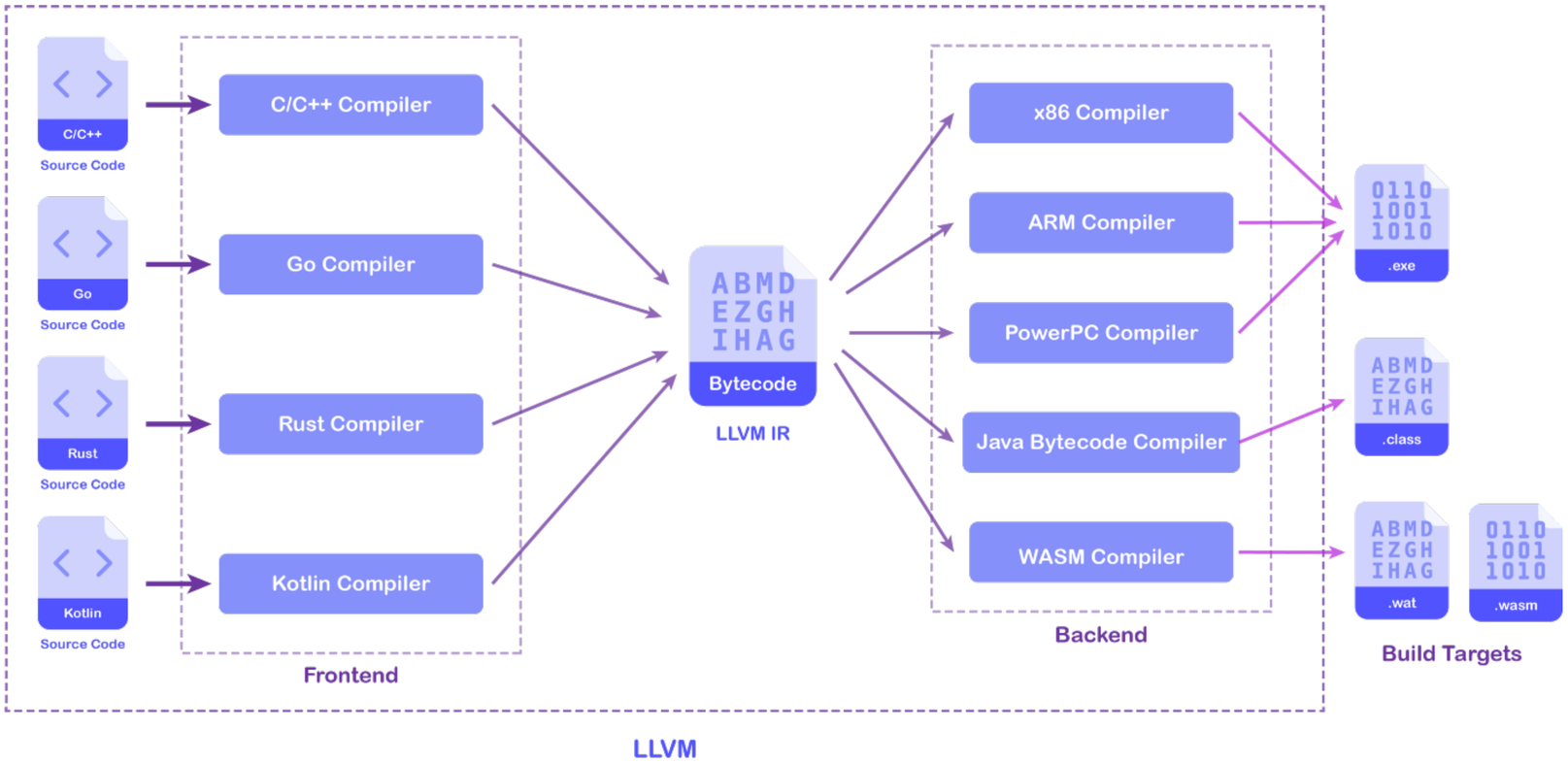




LLVM

*not an acronym.

- A set of compilers and a toolchain used to convert a program written in any programming language to a variety of build targets.
- **Main vision :** Compile programs written in any language into literally anything.



- Uses LLVM to compile C/C++ projects into web assembly.

hello.c

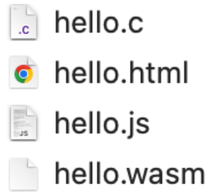
```c
#include <stdio.h>

int main() {
    printf("Hello, world\n");
    return 0;
}
```

Command

```
% emcc hello.c -o hello.html
```

Generated files



hello.html

Why AI Audio Codec?

- Reaching lowest possible bitrate while maintaining decent quality became the ultimate goal.
- Opus(not an AI codec) is not good enough. Demand for enhanced low-bitrate codec is increasing.
- More and more people are joining the internet, some of them with really slow network.
- Low bitrate is crucial to reduce the strain on networks.

Lyra vs. Satin

- AI based audio codec developed by Google.
- Very low-bitrate speech codec.
- Processing latency of 90 ms.
- Designed to operate at 3 kbps.
- Focuses on narrowband.

- AI based audio codec developed by Microsoft.
- Super wideband speech starting at a bitrate of 6 kbps.
- Full-band stereo music starting at a bitrate of 17 kbps.
- Progressively higher quality at higher bitrates.
- Supports super wideband.

Lyra supports lower bitrate than Satin, but Satin’s quality beats Lyra with a bitrate above 6 kbps.
Also, Lyra only supports speech, while Satin supports both speech and audio.

SoundStream

- A new AI based audio codec by Google.
- Enhanced version of Lyra.
- Supports audio other than speech.
- **SoundStream at 3 kbps outperforms Opus at 12 kbps.**

Compiling Lyra into Web Assembly

- First, we must find the key functions that are most relevant to encoding/ decoding.
- The official Lyra document states:

API

For integrating Lyra into any project only two APIs are relevant: LyraEncoder and LyraDecoder.

Compiling Lyra into Web Assembly

- However, to utilize these basic APIs, there were some compatibility problems.
    - Conversion of absl::Span and C++ constructor to JS.

```
class LyraEncoder : public LyraEncoderInterface {
 public:
  static std::unique_ptr<LyraEncoder> Create(
      int sample_rate_hz, int num_channels, int bitrate, bool enable_dtx,
      const ghc::filesystem::path& model_path);

  absl::optional<std::vector<uint8_t>> Encode(
      const absl::Span<const int16_t> audio) override;
```

```
class LyraDecoder : public LyraDecoderInterface {
 public:
  static std::unique_ptr<LyraDecoder> Create(
      int sample_rate_hz, int num_channels, int bitrate,
      const ghc::filesystem::path& model_path);

  bool SetEncodedPacket(absl::Span<const uint8_t> encoded) override;

  absl::optional<std::vector<int16_t>> DecodeSamples(int num_samples) override;

  absl::optional<std::vector<int16_t>> DecodePacketLoss(
      int num_samples) override;
```

Compiling Lyra into Web Assembly

- Instead, I found these two functions, where the conversion to JS are fairly easy.

```
// Encodes a vector of wav_data into encoded_features.
// Uses the quant files located under |model_path|.
bool EncodeWav(const std::vector<int16_t>& wav_data, int num_channels,
               int sample_rate_hz, bool enable_preprocessing, bool enable_dtx,
               const ghc::filesystem::path& model_path,
               std::vector<uint8_t>* encoded_features);
```

```
// Decodes a vector of bytes into wav data.
bool DecodeFeatures(const std::vector<uint8_t>& packet_stream,
                    float packet_loss_rate, float average_burst_length,
                    LyraDecoder* decoder, std::vector<int16_t>* decoded_audio);
```

Compiling Lyra into Web Assembly

- Slight modification to make it usable in JavaScript.

```
// Encodes a vector of wav_data into encoded_features.
// Uses the quant files located under |model_path|.
bool EncodeWav(const std::vector<int16_t>& wav_data, int num_channels,
               int sample_rate_hz, bool enable_preprocessing, bool enable_dtx,
               const ghc::filesystem::path& model_path,
               std::vector<uint8_t>* encoded_features);
```

```
std::vector<uint8_t> EncodeWav2(emscripten::val wav_data2, int num_channels,
                                int sample_rate_hz, bool enable_preprocessing, bool enable_dtx,
                                std::string model_path2) {
  std::vector<int16_t> wav_data = emscripten::convertJSArrayToNumberVector<int16_t>(wav_data2);
  ghc::filesystem::path model_path = GetCompleteArchitecturePath(model_path2);
  std::vector<uint8_t> encoded_features;
```

Compiling Lyra into Web Assembly

- Slight modification to make it usable in JavaScript.

```
// Decodes a vector of bytes into wav data.
bool DecodeFeatures(const std::vector<uint8_t>& packet_stream,
                    float packet_loss_rate, float average_burst_length,
                    LyraDecoder* decoder, std::vector<int16_t>* decoded_audio);
```

```
std::vector<int16_t> DecodeFeatures2(emscripten::val packet_stream2,
                                     float packet_loss_rate, float average_burst_length) {
  std::vector<uint8_t> packet_stream = emscripten::convertJSArrayToNumberVector<uint8_t>(packet_stream2);
  std::vector<int16_t> decoded_audio;
  auto decoder2 =
      LyraDecoder::Create(48000, 1, 3000, "./wavegru");
  LyraDecoder* decoder = decoder2.get();
```

Compiling Lyra into Web Assembly

- Created “binding.cc” file to bind C++ functions to JavaScript.

binding.cc

```cpp
#include <emscripten/bind.h>
#include "encoder_main_lib.h"
#include "decoder_main_lib.h"

using namespace emscripten;

EMSCRIPTEN_BINDINGS(Encoder) {
    function("EncodeWav2", &chromemedia::codec::EncodeWav2);
    function("DecodeFeatures2", &chromemedia::codec::DecodeFeatures2);

    register_vector<uint8_t>("vector<uint8_t>");
    register_vector<int16_t>("vector<int16_t>");
}
```

BUILD

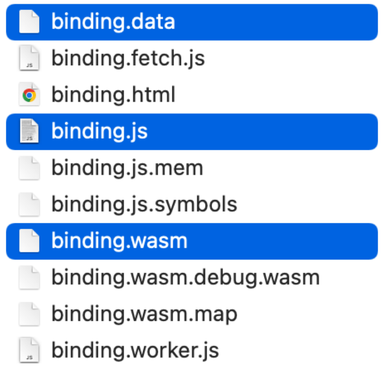
```
cc_binary(
    name = "binding",
    srcs = [
        "binding.cc",
    ],
    linkopts = ["-s USE_PTHREADS=0",
                "--bind",
                "--preload-file wavegru@wavegru",
                "--export-all",
                "--no-entry",
                "-s ALLOW_MEMORY_GROWTH=1",],
    deps = [
        ":architecture_utils",
        ":encoder_main_lib",
        ":decoder_main_lib",
        "@com_google_absl//absl/flags:flag",
        "@com_google_absl//absl/flags:parse",
        "@com_google_absl//absl/flags:usage",
        "@com_google_absl//absl/strings",
        "@com_google_glog//:glog",
        "@gulrak_filesystem//:filesystem",
    ],
)

wasm_cc_binary(
    name = "binding-wasm",
    cc_target = ":binding",
)
```

Compiling Lyra into Web Assembly

- Finally, JS can use C++ functions as if they were native JS functions.

Generated files by Emscripten



JS code in index.html

```
  //create the wav blob and pass it on to createAudio
  rec.exportWAV(createAudio);
}

function createAudio(blob) {
  blob.arrayBuffer()
    .then(buffer => {
      const data = new Int16Array(buffer);

      var encVector = Module.EncodeWav2(data, 1, 48000, false, false, "./wavegru");
      var uint8arr = new Uint8Array(encVector.size());
      for (var i = 0; i < encVector.size(); i++) {
        uint8arr[i] = encVector.get(i);
      }

      var decVector = Module.DecodeFeatures2(uint8arr, 0.0, 1.0);
      var int16arr = new Int16Array(decVector.size());
      for (var i = 0; i < decVector.size(); i++) {
        int16arr[i] = decVector.get(i);
      }
      var result = new Float32Array(int16arr.length);
      for(var i = 0; i < int16arr.length; i++) {
        result[i] = int16arr[i] / (int16arr[i] >= 0 ? 32767 : 32768);
      }
      var x = new Blob([exportWAV(result)], { type: 'audio/wav' });
```

Compiling Lyra into Web Assembly

- Lyra Wasm Demo

Web Assembly Games

- AngryBots
- Banana Bread