3dnet

Innovative Clinical Collaboration, Not Software

www.3DnetMedical.com

# C++ On The Web

Liam Carter
Senior Software Engineer

Biotronics 3D
Analyze · Collaborate · Discover

https://github.com/liamcarter111/Lectures

# WebAssembly (WASM)

“WebAssembly is a new type of code that can be run in modern web browsers — it is a low-level assembly-like language with a compact binary format that runs with near-native performance and provides languages such as C/C++ and Rust with a compilation target so that they can run on the web. It is also designed to run alongside JavaScript, allowing both to work together.” (MDN, 2019)

# WebAssembly (WASM)

```c
// Filename: add.c
int add(int a, int b)
{
  return a + b;
}
```

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

clang --target=wasm32 -nostdlib -Wl,--no-entry -Wl,--export-all -o add.wasm add.c
```

# WebAssembly (WASM)

```c
// Filename: add.c
int add(int a, int b)
{
  return a + b;
}
```

```
// Filename: add.wasm
Offset(h) 00   02   04   06   08   0A   0C   0E
00000000  0061 736D 0100 0000 010A 0260 0000 6002
00000010  7F7F 017F 0303 0200 0104 0501 7001 0101
00000020  0503 0100 0206 2B07 7F01 4180 8804 0B7F
00000030  0041 8008 0B7F 0041 8008 0B7F 0041 8008
00000040  0B7F 0041 8088 040B 7F00 4100 0B7F 0041
00000050  010B 077D 0906 6D65 6D6F 7279 0200 115F
00000060  5F77 6173 6D5F 6361 6C6C 5F63 746F 7273
00000070  0000 0361 6464 0001 0C5F 5F64 736F 5F68
00000080  616E 646C 6503 010A 5F5F 6461 7461 5F65
00000090  6E64 0302 0D5F 5F67 6C6F 6261 6C5F 6261
000000A0  7365 0303 0B5F 5F68 6561 705F 6261 7365
000000B0  0304 0D5F 5F6D 656D 6F72 795F 6261 7365
000000C0  0305 0C5F 5F74 6162 6C65 5F62 6173 6503
000000D0  060A 0C02 0200 0B07 0020 0120 006A 0B00
000000E0  2004 6E61 6D65 0119 0200 115F 5F77 6173
000000F0  6D5F 6361 6C6C 5F63 746F 7273 0103 6164
00000100  6400 2609 7072 6F64 7563 6572 7301 0C70
00000110  726F 6365 7373 6564 2D62 7901 0563 6C61
00000120  6E67 0631 302E 302E 30
```

# WebAssembly (WASM)

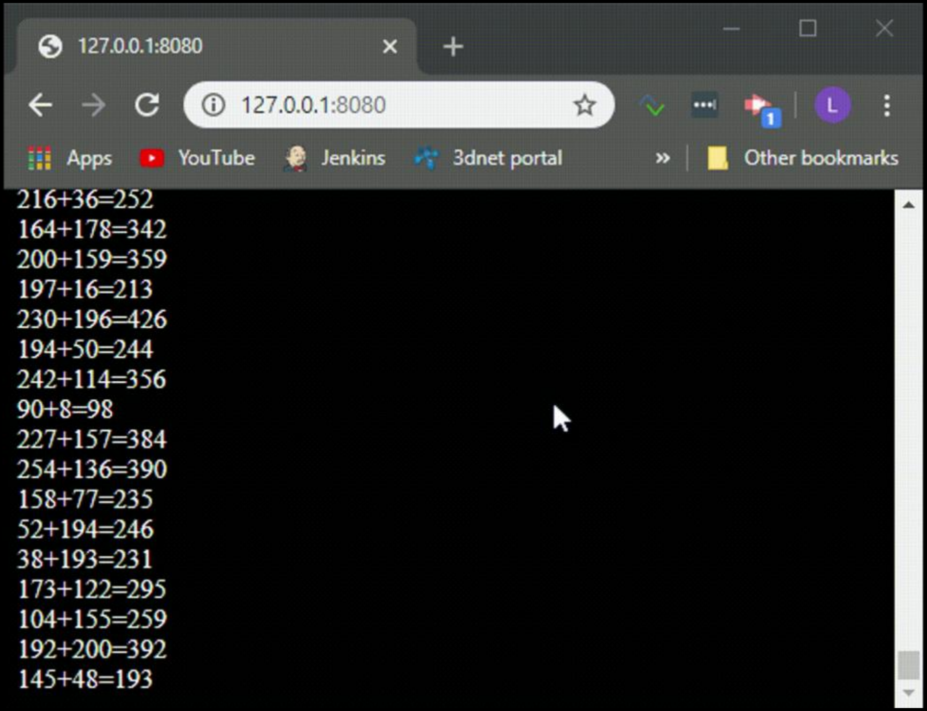
```c
// Filename: add.c
int add(int a, int b)
{
  return a + b;
}
```

```wat
;; Filename: add.wat
(module
 (table 0 anyfunc)
 (memory $0 1)
 (export "memory" (memory $0))
 (export "add" (func $add))
 (func $add (; 0 ;) (param $0 i32)
 (param $1 i32) (result i32)
  (i32.add
   (get_local $1)
   (get_local $0)
  )
 )
)
```

# WebAssembly (WASM)

```
// Filename: index.html
<!DOCTYPE html>
<script>
  async function init() {
    const { instance } = await WebAssembly.instantiateStreaming(
      fetch("./add.wasm")
    );
    const a = (Math.random() * 0xff) | 0;
    const b = (Math.random() * 0xff) | 0;
    const r = instance.exports.add(a, b);
    document.body.innerHTML += `<div>${a}+${b}=${r}<div>`;   }
  setInterval(init);
</script>
```

# WebAssembly (WASM)



```
216+36=252
164+178=342
200+159=359
197+16=213
230+196=426
194+50=244
242+114=356
90+8=98
227+157=384
254+136=390
158+77=235
52+194=246
38+193=231
173+122=295
104+155=259
192+200=392
145+48=193
```

# Why WebAssembly?

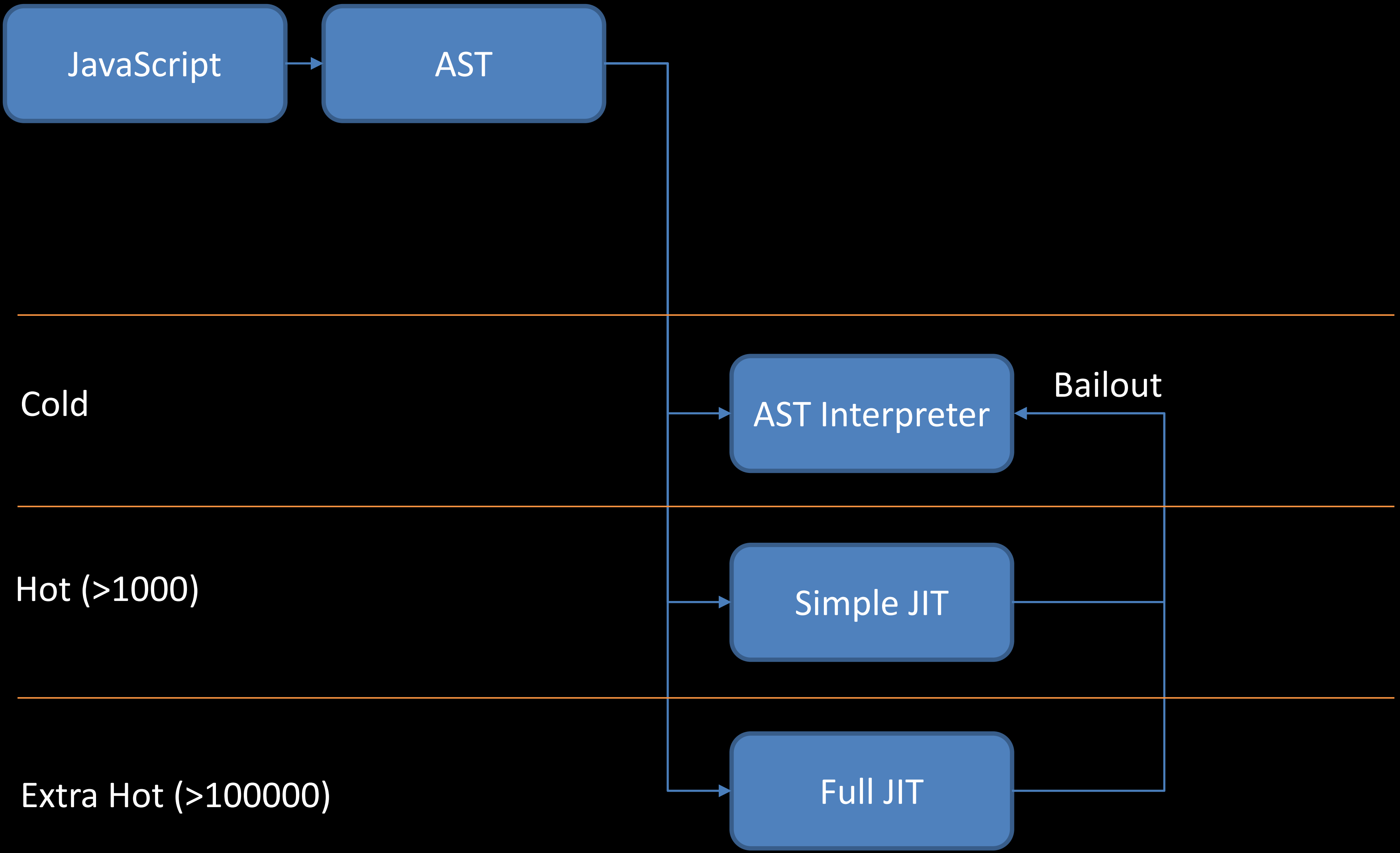
- Small binary format
- No Parsing/JIT overhead
- Performance
- Highly optimized STL algorithms
- Existing codebases

  (Image Processing, Encoding/Decoding, Compression)
- Type-safety
- Full control of assigned memory
- Complex memory structures
- No garbage collection

# Similar works

- x86 Emulators
- ASM.JS

# JavaScript under the hood



JavaScript

AST

Cold

AST Interpreter

Bailout

Hot (>1000)

Simple JIT

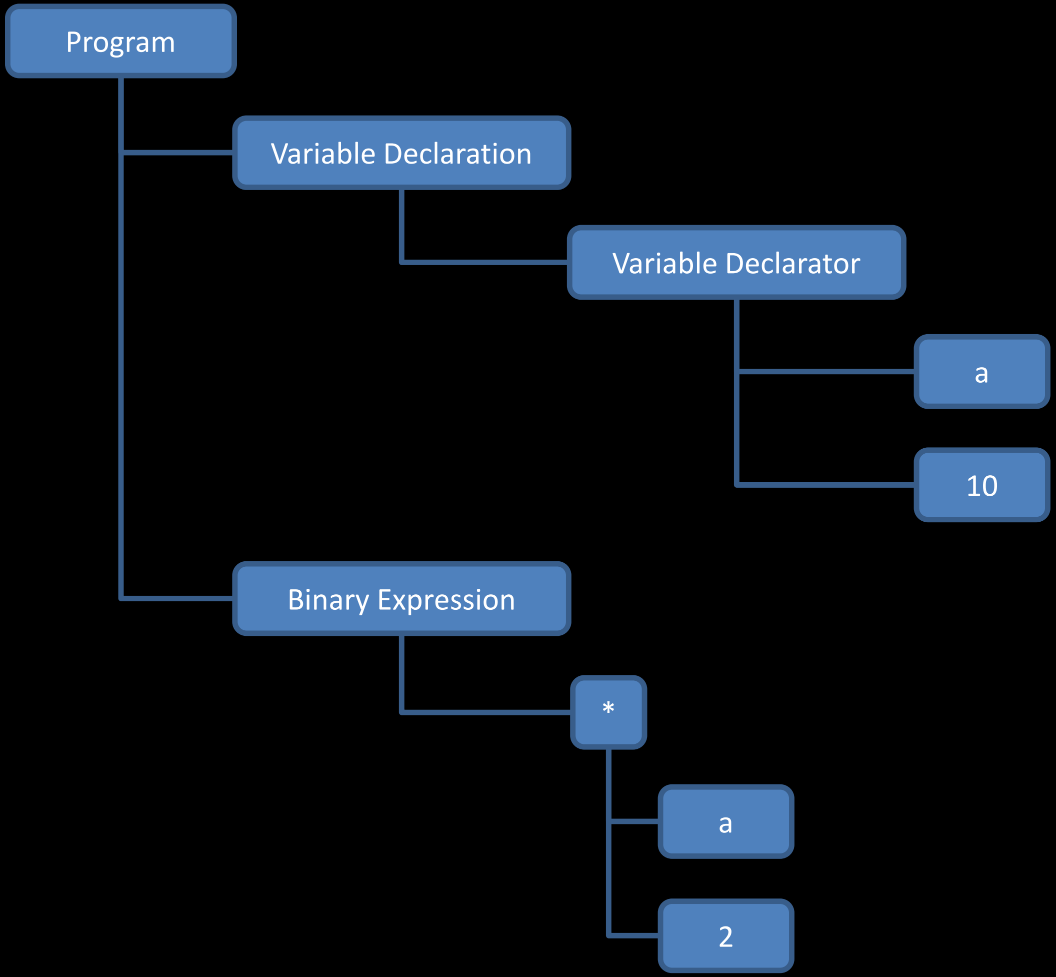
Extra Hot (>100000)

Full JIT

# JavaScript under the hood

Abstract syntax tree (AST)

```
// Source Code
var a = 10;
a * 2;
```



Program

Variable Declaration

Variable Declarator

a

10

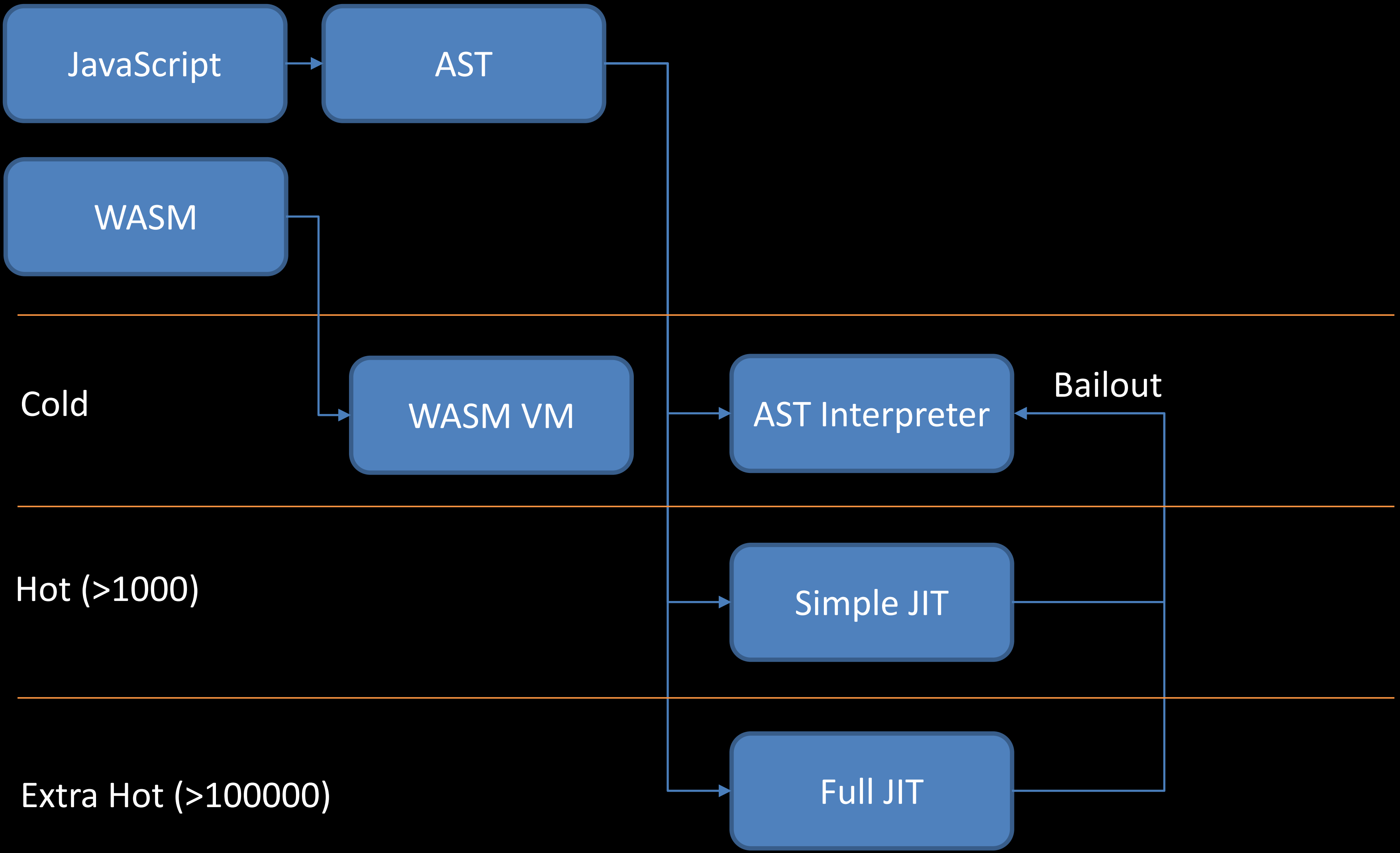
Binary Expression

*

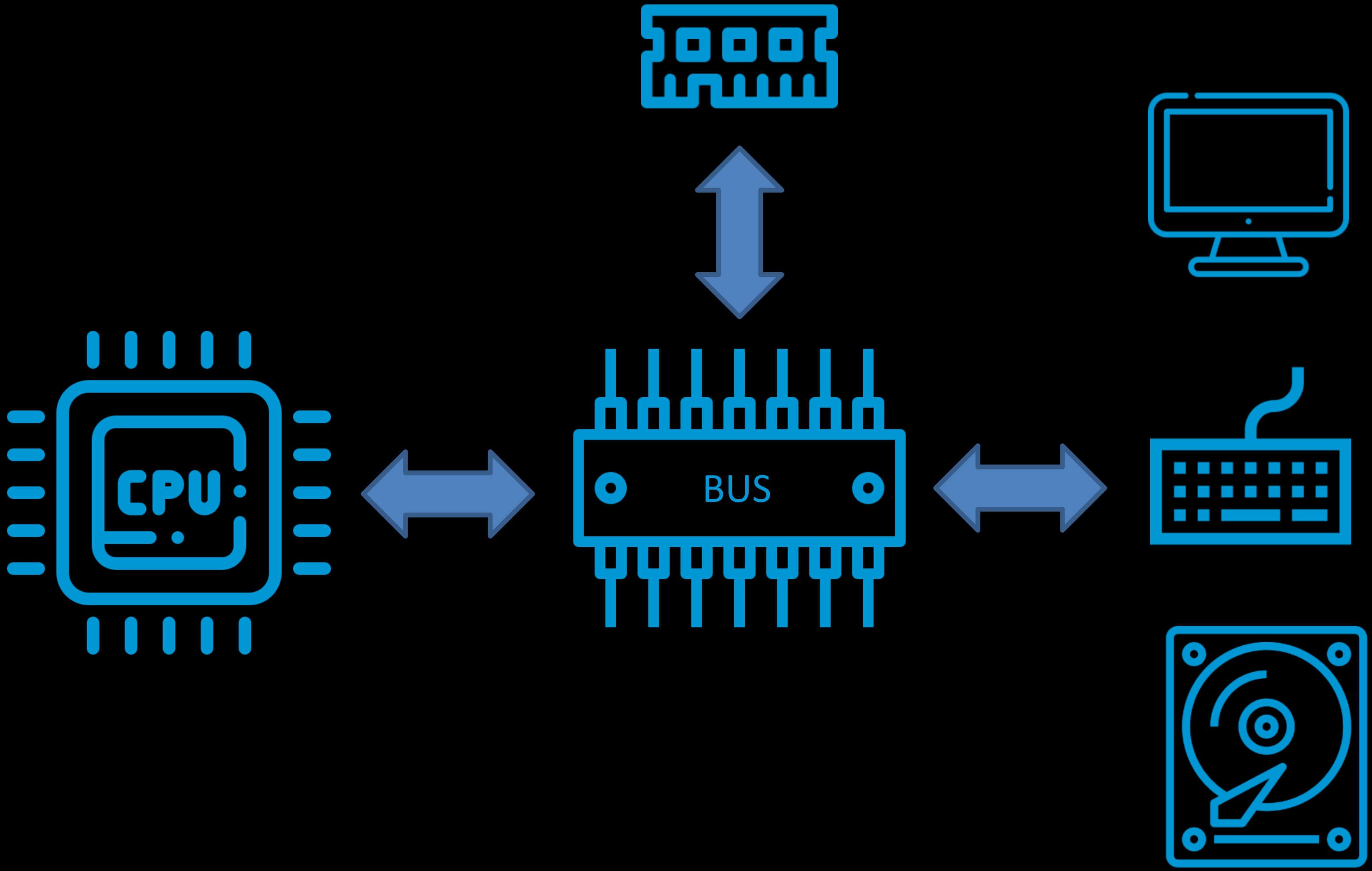
a

2

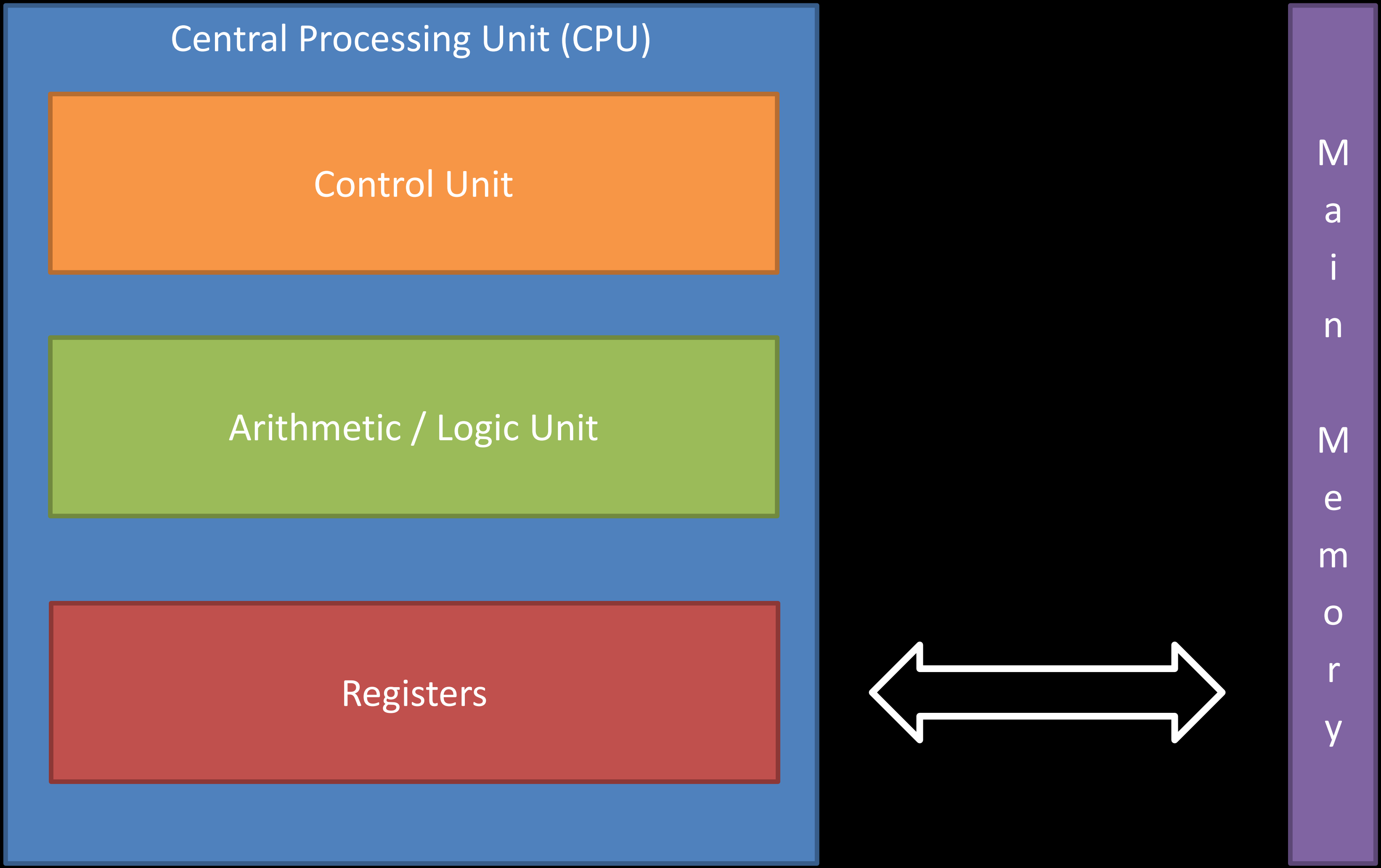
# JavaScript under the hood



# How A CPU Works



# How A CPU Works



Von Neumann Architecture

# Register Machine

- Consists of a finite set of registers, and a memory unit
- Data can be moved between registers and/or the memory unit
- Arithmetic operations performed on registers only
- Finite control of register assignment
- Arithmetic operations do not effect the machine state
- Branching becomes complex

# Register Machine

```
mov REG_A,0x00          ; move value at 0x00 into “Register A”
mov REG_B,0x01          ; move value at 0x01 into “Register B”
add REG_A,REG_B         ; perform an assignment addition (REG_A += REG_B)

mov REG_B,0x02          ; move value at 0x02 into “Register B”
add REG_A,REG_B         ; perform an assignment addition (REG_A += REG_B)

mov REG_B,0x03          ; move value at 0x03 into “Register B”
add REG_A,REG_B          ; perform an assignment addition (REG_A += REG_B)
```

# Register Machine

```
mov REG_A,0x00              ; move value at 0x00 into “Register A”
mov REG_B,0x01              ; move value at 0x01 into “Register B”
add REG_A,REG_B             ; perform an assignment addition (REG_A += REG_B)

mov REG_B,0x02              ; move value at 0x02 into “Register B”
add REG_A,REG_B             ; perform an assignment addition (REG_A += REG_B)

mov REG_B,0x03              ; move value at 0x03 into “Register B”
add REG_A,REG_B              ; perform an assignment addition (REG_A += REG_B)
```

Machine Code:

0x00 0x00 0x01 0x01 0x02 0x01 0x02 0x02 0x01 0x03 0x02

| Opcode | Mnemonic |
|---|---|
| 0x00 | mov REG_A |
| 0x01 | mov REG_B |
| 0x02 | add REG_A,REG_B |

# Register Machine

```
mov REG_A,0x00          ; move value at 0x00 into “Register A”
mov REG_B,0x01          ; move value at 0x01 into “Register B”
add REG_A,REG_B         ; perform an assignment addition (REG_A += REG_B)

mov REG_B,0x02          ; move value at 0x02 into “Register B”
add REG_A,REG_B         ; perform an assignment addition (REG_A += REG_B)

mov REG_B,0x03          ; move value at 0x03 into “Register B”
add REG_A,REG_B          ; perform an assignment addition (REG_A += REG_B)
```

Register A

0xFF

Register B

0xFF

| | | | |
|---|---|---|---|
| 0x01 | 0x02 | 0x03 | 0x04 |

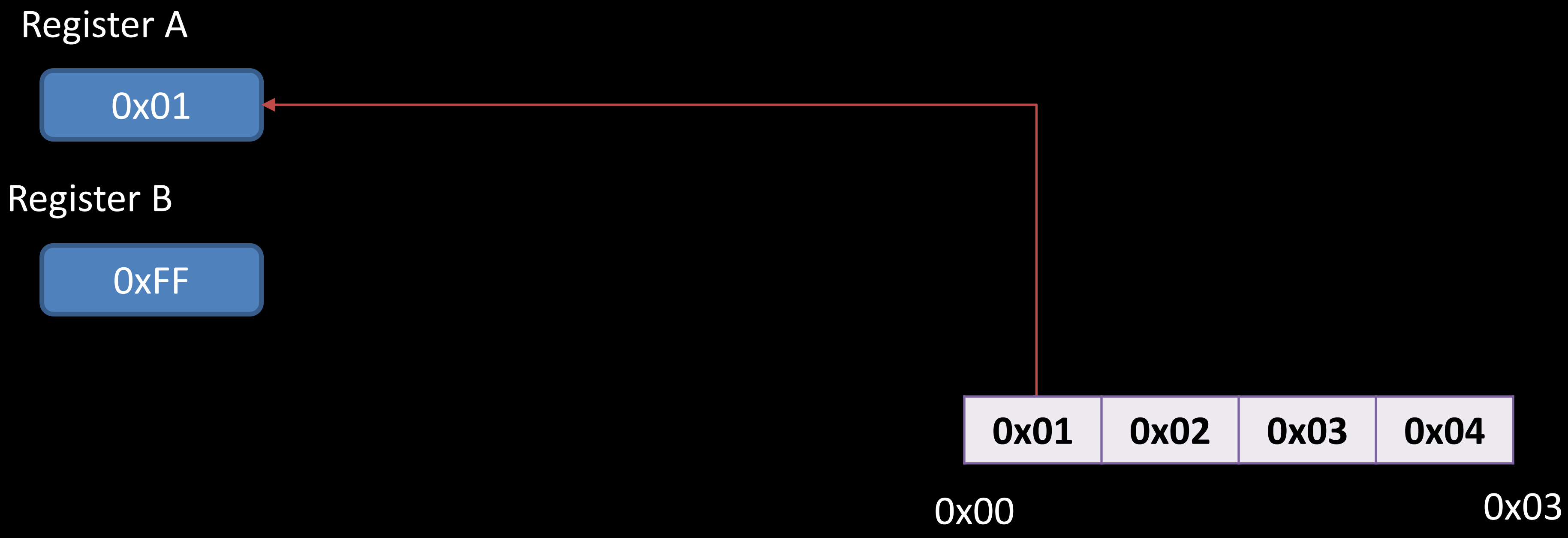
0x00 0x03

# Register Machine

```
mov REG_A,0x00            ; move value at 0x00 into “Register A”
mov REG_B,0x01            ; move value at 0x01 into “Register B”
add REG_A,REG_B           ; perform an assignment addition (REG_A += REG_B)

mov REG_B,0x02            ; move value at 0x02 into “Register B”
add REG_A,REG_B           ; perform an assignment addition (REG_A += REG_B)

mov REG_B,0x03            ; move value at 0x03 into “Register B”
add REG_A,REG_B            ; perform an assignment addition (REG_A += REG_B)
```



Register A

0x01

Register B

0xFF

| 0x01 | 0x02 | 0x03 | 0x04 |
|---|---|---|---|

0x00

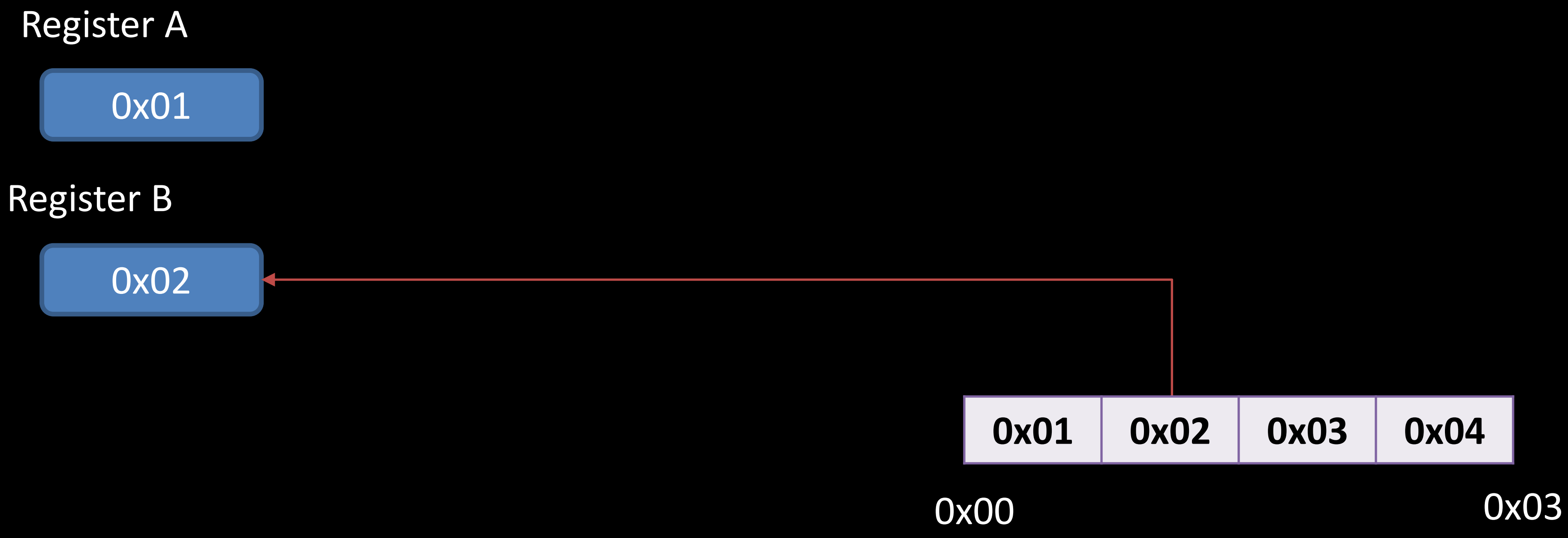
0x03

# Register Machine

```
mov REG_A,0x00          ; move value at 0x00 into “Register A”
mov REG_B,0x01          ; move value at 0x01 into “Register B”
add REG_A,REG_B         ; perform an assignment addition (REG_A += REG_B)

mov REG_B,0x02          ; move value at 0x02 into “Register B”
add REG_A,REG_B         ; perform an assignment addition (REG_A += REG_B)

mov REG_B,0x03          ; move value at 0x03 into “Register B”
add REG_A,REG_B          ; perform an assignment addition (REG_A += REG_B)
```



Register A

0x01

Register B

0x02

| 0x01 | 0x02 | 0x03 | 0x04 |
|---|---|---|---|

0x00

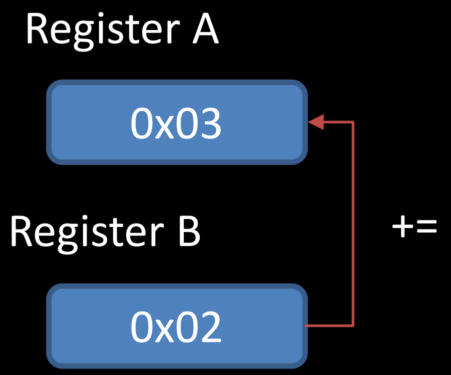
0x03

# Register Machine

```
mov REG_A,0x00        ; move value at 0x00 into “Register A”
mov REG_B,0x01        ; move value at 0x01 into “Register B”
add REG_A,REG_B       ; perform an assignment addition (REG_A += REG_B)

mov REG_B,0x02        ; move value at 0x02 into “Register B”
add REG_A,REG_B       ; perform an assignment addition (REG_A += REG_B)

mov REG_B,0x03        ; move value at 0x03 into “Register B”
add REG_A,REG_B        ; perform an assignment addition (REG_A += REG_B)
```



Register A

0x03

Register B

0x02

+=

| 0x01 | 0x02 | 0x03 | 0x04 |
|---|---|---|---|

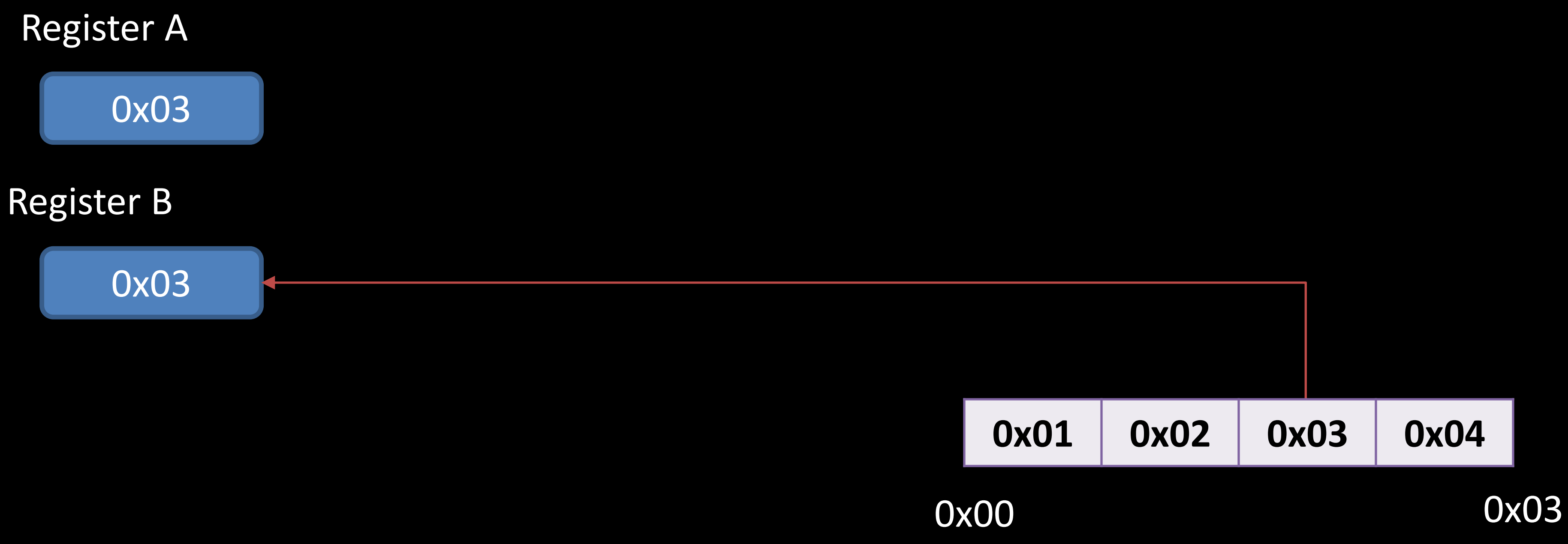
0x00 0x03

# Register Machine

```
mov REG_A,0x00          ; move value at 0x00 into “Register A”
mov REG_B,0x01          ; move value at 0x01 into “Register B”
add REG_A,REG_B         ; perform an assignment addition (REG_A += REG_B)

mov REG_B,0x02          ; move value at 0x02 into “Register B”
add REG_A,REG_B         ; perform an assignment addition (REG_A += REG_B)

mov REG_B,0x03          ; move value at 0x03 into “Register B”
add REG_A,REG_B          ; perform an assignment addition (REG_A += REG_B)
```



Register A

0x03

Register B

0x03

| 0x01 | 0x02 | 0x03 | 0x04 |
|---|---|---|---|

0x00

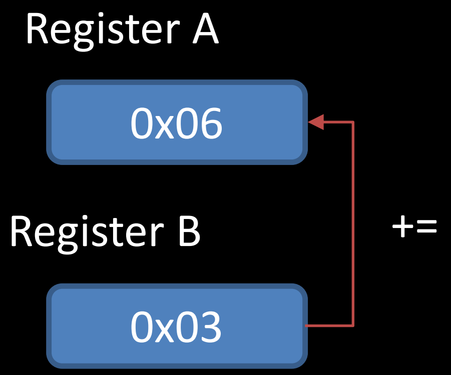
0x03

# Register Machine

```
  mov REG_A,0x00          ; move value at 0x00 into “Register A”
  mov REG_B,0x01          ; move value at 0x01 into “Register B”
  add REG_A,REG_B         ; perform an assignment addition (REG_A += REG_B)

  mov REG_B,0x02          ; move value at 0x02 into “Register B”
➡ add REG_A,REG_B         ; perform an assignment addition (REG_A += REG_B)

  mov REG_B,0x03          ; move value at 0x03 into “Register B”
  add REG_A,REG_B          ; perform an assignment addition (REG_A += REG_B)
```



| | | | |
|---|---|---|---|
| 0x01 | 0x02 | 0x03 | 0x04 |

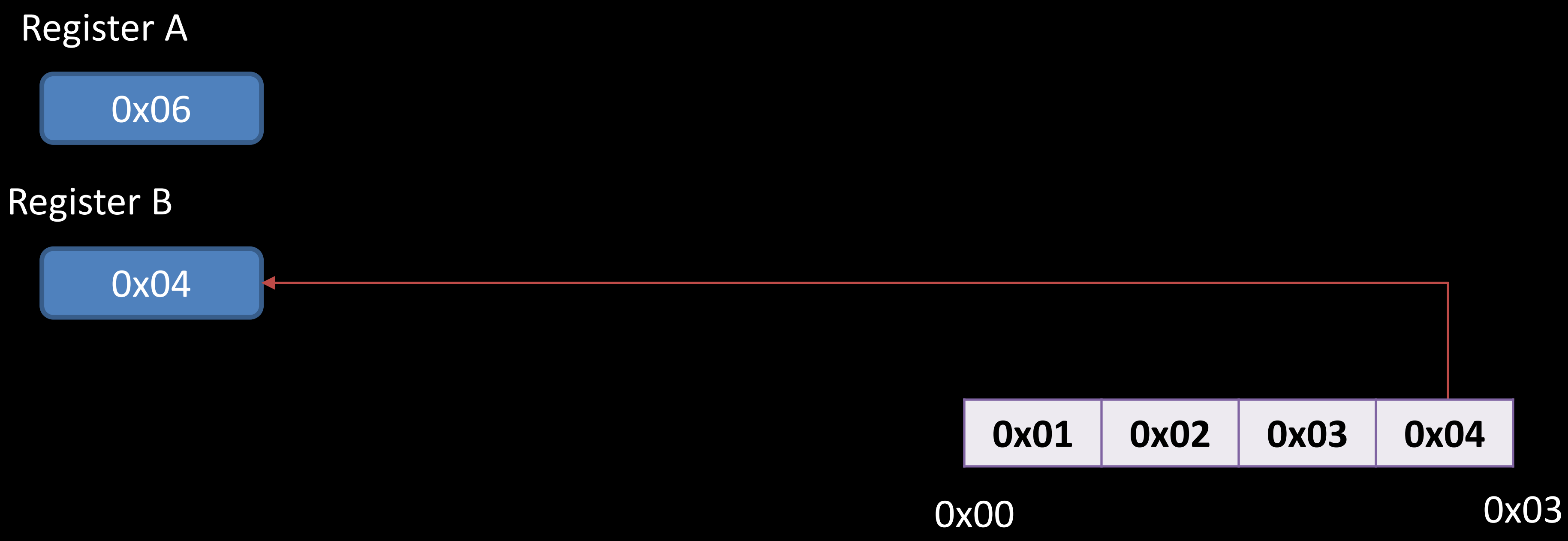
0x00 0x03

# Register Machine

```
mov REG_A,0x00          ; move value at 0x00 into “Register A”
mov REG_B,0x01          ; move value at 0x01 into “Register B”
add REG_A,REG_B         ; perform an assignment addition (REG_A += REG_B)

mov REG_B,0x02          ; move value at 0x02 into “Register B”
add REG_A,REG_B         ; perform an assignment addition (REG_A += REG_B)

mov REG_B,0x03          ; move value at 0x03 into “Register B”
add REG_A,REG_B          ; perform an assignment addition (REG_A += REG_B)
```



Register A

0x06

Register B

0x04

| 0x01 | 0x02 | 0x03 | 0x04 |
|---|---|---|---|

0x00

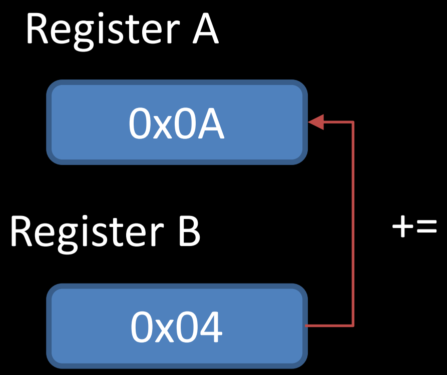
0x03

# Register Machine

```
mov REG_A,0x00              ; move value at 0x00 into “Register A”
mov REG_B,0x01              ; move value at 0x01 into “Register B”
add REG_A,REG_B             ; perform an assignment addition (REG_A += REG_B)

mov REG_B,0x02              ; move value at 0x02 into “Register B”
add REG_A,REG_B             ; perform an assignment addition (REG_A += REG_B)

mov REG_B,0x03              ; move value at 0x03 into “Register B”
add REG_A,REG_B              ; perform an assignment addition (REG_A += REG_B)
```



Register A

0x0A

Register B

0x04

+=

| 0x01 | 0x02 | 0x03 | 0x04 |
|---|---|---|---|

0x00 0x03

# Stack Machine

- Consists of a memory unit
- Two operations “push” and “pop” (LIFO)
- Arithmetic operations performed on the stack
- Arithmetic operations do affect the machine state
- Internal state management abstracted away
- Branching is simplified

# Stack Machine

```
push 0x00        ; push value at 0x00 on the stack
push 0x01        ; push value at 0x01 on the stack
push 0x02        ; push value at 0x02 on the stack
push 0x03        ; push value at 0x03 on the stack
add              ; pop two values, perform an addition, then push the result
add              ; pop two values, perform an addition, then push the result
add              ; pop two values, perform an addition, then push the result
```

# Stack Machine

```
push 0x00          ; push value at 0x00 on the stack
push 0x01          ; push value at 0x01 on the stack
push 0x02          ; push value at 0x02 on the stack
push 0x03          ; push value at 0x03 on the stack
add                ; pop two values, perform an addition, then push the result
add                ; pop two values, perform an addition, then push the result
add                ; pop two values, perform an addition, then push the result
```

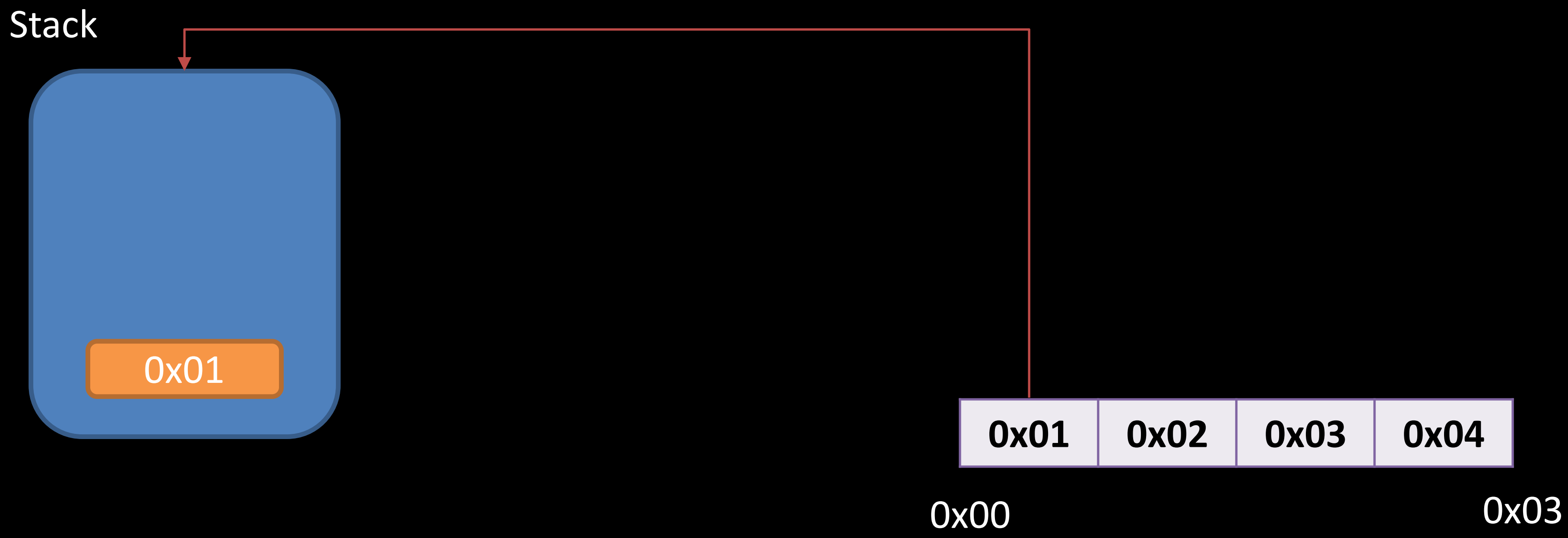
Machine Code:

```
0x00 0x00 0x00 0x01 0x00 0x02 0x00 0x03 0x01 0x01 0x01
```

| Opcode | Mnemonic |
|---|---|
| 0x00 | push |
| 0x01 | add |

# Stack Machine

```
push 0x00        ; push value at 0x00 on the stack
push 0x01        ; push value at 0x01 on the stack
push 0x02        ; push value at 0x02 on the stack
push 0x03        ; push value at 0x03 on the stack
add              ; pop two values, perform an addition, then push the result
add              ; pop two values, perform an addition, then push the result
add              ; pop two values, perform an addition, then push the result
```



Stack

0x01

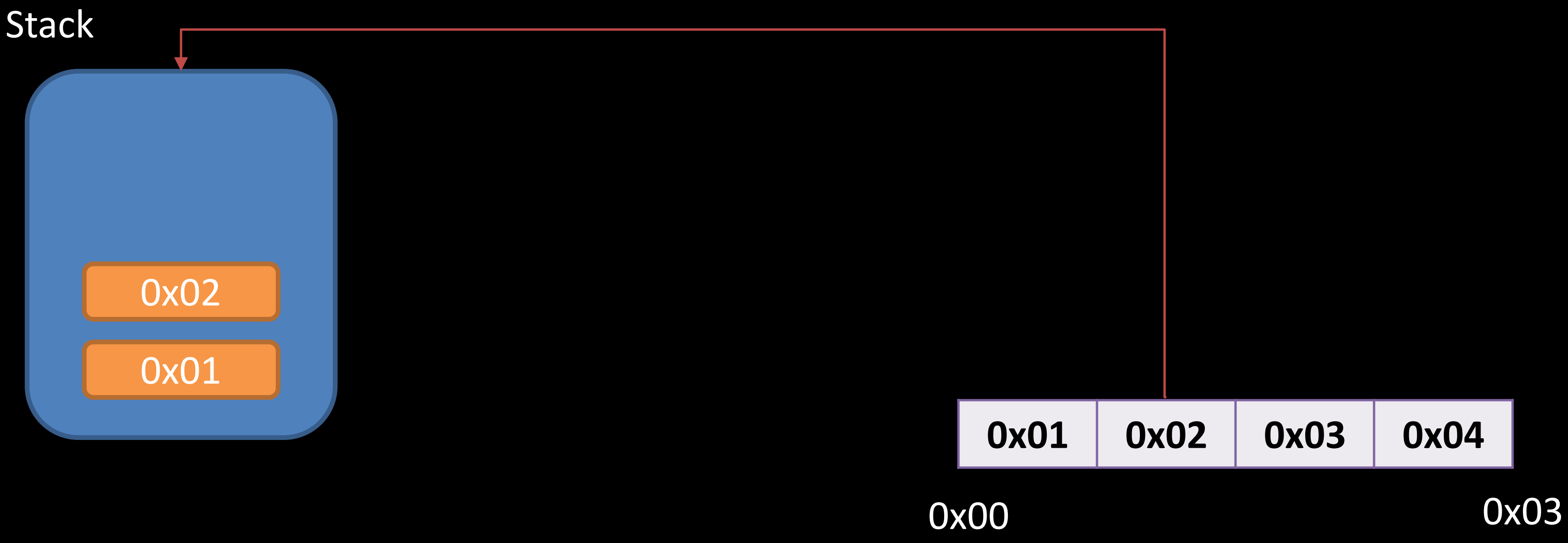
|  |  |  |  |
|---|---|---|---|
| 0x01 | 0x02 | 0x03 | 0x04 |

0x00

0x03

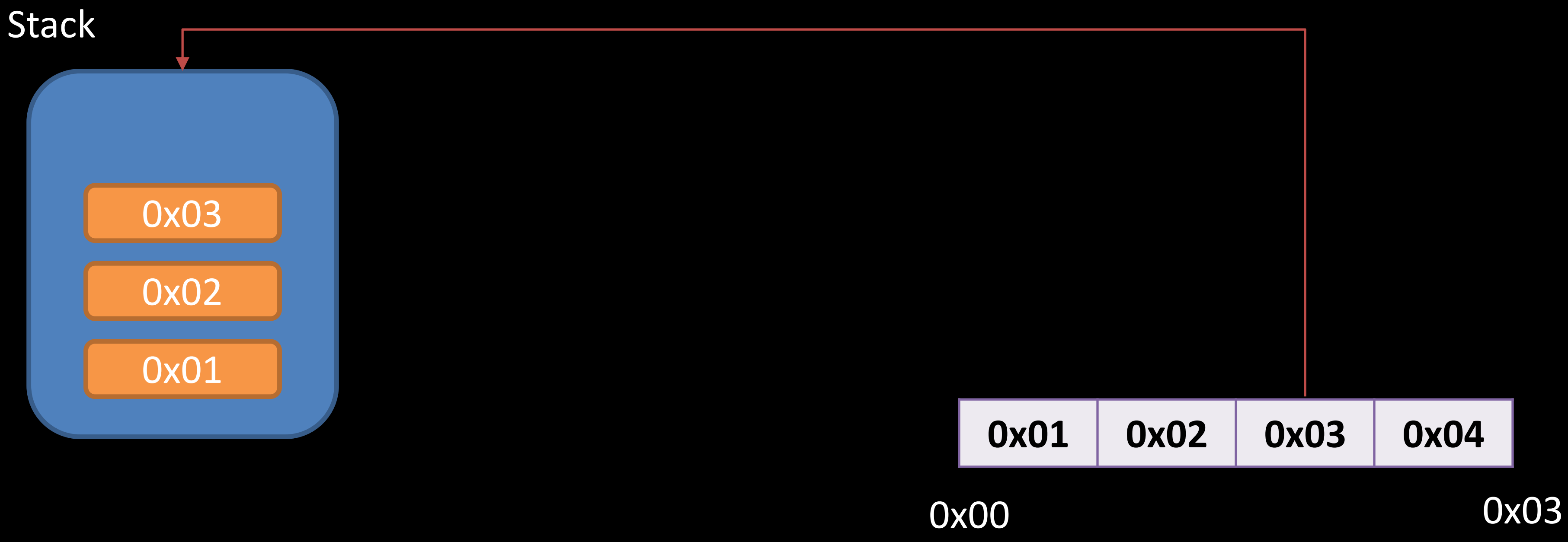
# Stack Machine

```
push 0x00        ; push value at 0x00 on the stack
push 0x01        ; push value at 0x01 on the stack
push 0x02        ; push value at 0x02 on the stack
push 0x03        ; push value at 0x03 on the stack
add              ; pop two values, perform an addition, then push the result
add              ; pop two values, perform an addition, then push the result
add              ; pop two values, perform an addition, then push the result
```



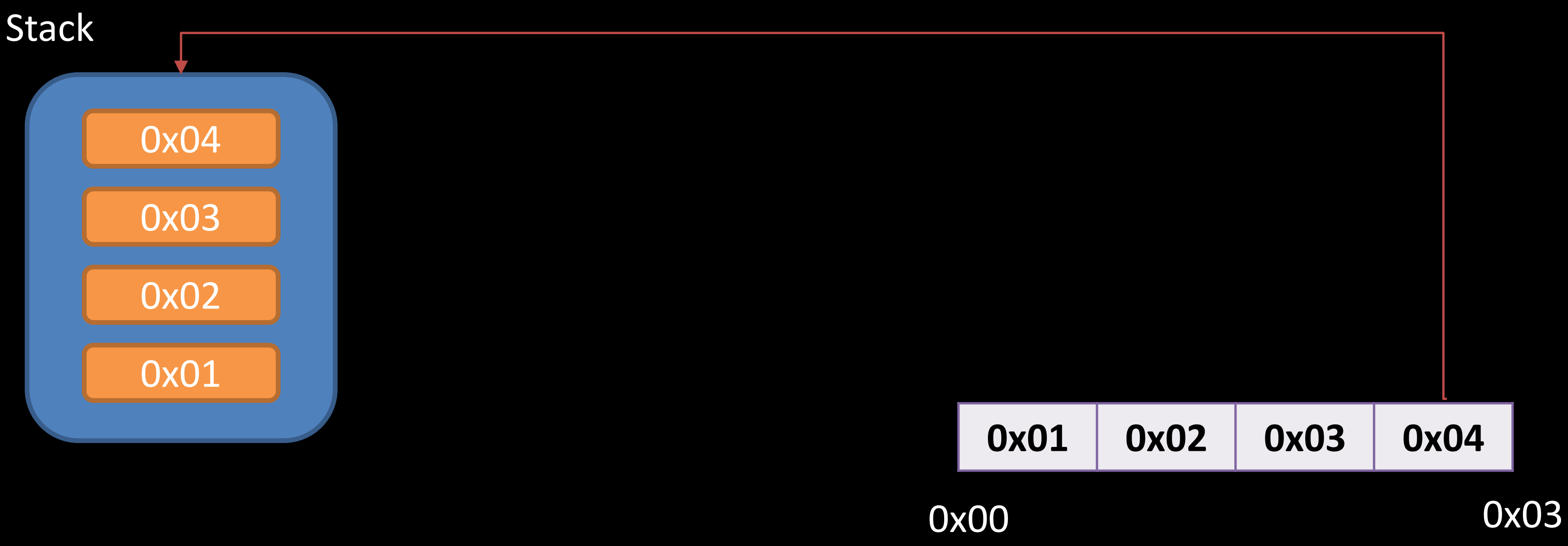
# Stack Machine

```
push 0x00          ; push value at 0x00 on the stack
push 0x01          ; push value at 0x01 on the stack
push 0x02          ; push value at 0x02 on the stack
push 0x03          ; push value at 0x03 on the stack
add                ; pop two values, perform an addition, then push the result
add                ; pop two values, perform an addition, then push the result
add                ; pop two values, perform an addition, then push the result
```



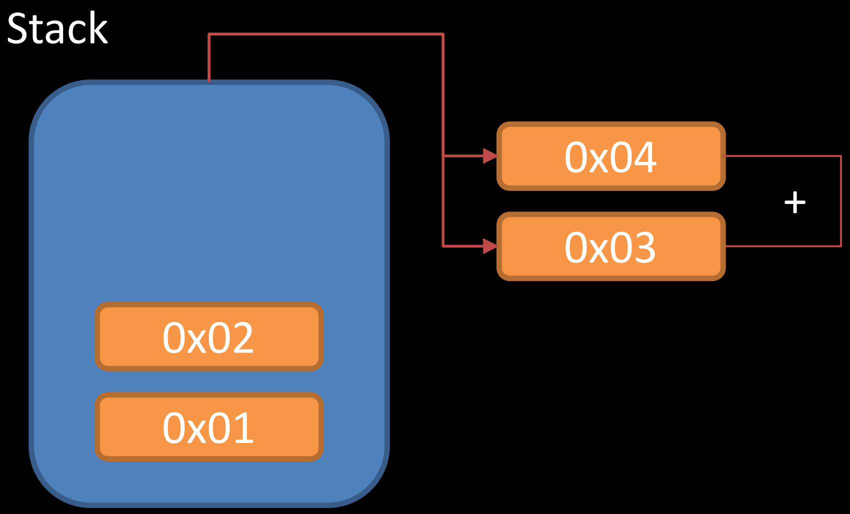
# Stack Machine

```
push 0x00        ; push value at 0x00 on the stack
push 0x01        ; push value at 0x01 on the stack
push 0x02        ; push value at 0x02 on the stack
push 0x03        ; push value at 0x03 on the stack
add              ; pop two values, perform an addition, then push the result
add              ; pop two values, perform an addition, then push the result
add              ; pop two values, perform an addition, then push the result
```



# Stack Machine

```
push 0x00          ; push value at 0x00 on the stack
push 0x01          ; push value at 0x01 on the stack
push 0x02          ; push value at 0x02 on the stack
push 0x03          ; push value at 0x03 on the stack
add                ; pop two values, perform an addition, then push the result
add                ; pop two values, perform an addition, then push the result
add                ; pop two values, perform an addition, then push the result
```

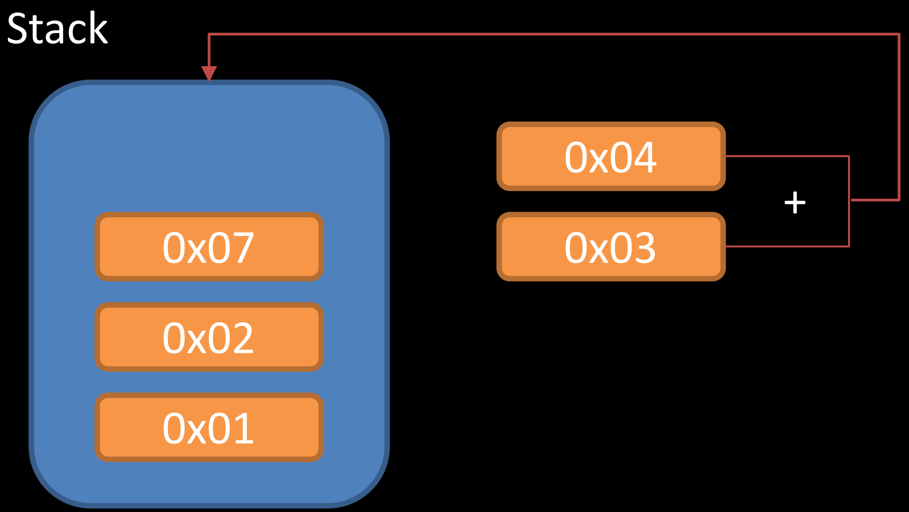


| 0x01 | 0x02 | 0x03 | 0x04 |
|---|---|---|---|

0x00 0x03

# Stack Machine

```
push 0x00        ; push value at 0x00 on the stack
push 0x01        ; push value at 0x01 on the stack
push 0x02        ; push value at 0x02 on the stack
push 0x03        ; push value at 0x03 on the stack
add              ; pop two values, perform an addition, then push the result
add              ; pop two values, perform an addition, then push the result
add              ; pop two values, perform an addition, then push the result
```



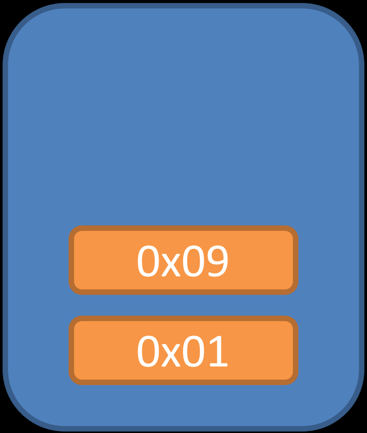
| | | | |
|---|---|---|---|
| 0x01 | 0x02 | 0x03 | 0x04 |

0x00 0x03

# Stack Machine

```
  push 0x00      ; push value at 0x00 on the stack
  push 0x01      ; push value at 0x01 on the stack
  push 0x02      ; push value at 0x02 on the stack
  push 0x03      ; push value at 0x03 on the stack
  add            ; pop two values, perform an addition, then push the result
➡ add            ; pop two values, perform an addition, then push the result
  add            ; pop two values, perform an addition, then push the result
```

Stack

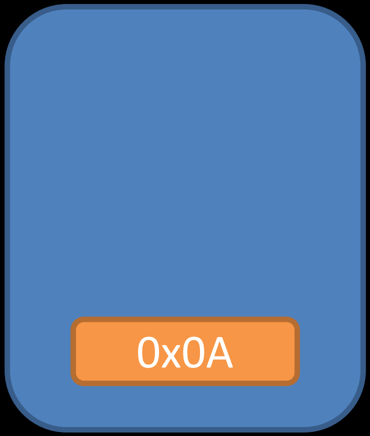


| | | | |
|---|---|---|---|
| 0x01 | 0x02 | 0x03 | 0x04 |
| 0x00 | | | 0x03 |

# Stack Machine

```
     push 0x00        ; push value at 0x00 on the stack
     push 0x01        ; push value at 0x01 on the stack
     push 0x02        ; push value at 0x02 on the stack
     push 0x03        ; push value at 0x03 on the stack
     add              ; pop two values, perform an addition, then push the result
     add              ; pop two values, perform an addition, then push the result
  →  add              ; pop two values, perform an addition, then push the result
```

Stack



|  |  |  |  |
|---|---|---|---|
| 0x01 | 0x02 | 0x03 | 0x04 |

0x00 0x03

# Hybrid Machine

- Modern CPU architectures are typically a hybrid of Stack and Register Machines.
- Allows efficient use of both paradigms (Languages such as C/C++ take advantage of this using both a stack and heap)
- Simplifies Branching
- Reduces the Machine Code size

# Virtual Machine

- Emulate a physical or theoretical machine
- Abstraction
- Portability
- Security

# Stack Machine

```
push 0x00       ; push value at 0x00 on the stack
push 0x01       ; push value at 0x01 on the stack
push 0x02       ; push value at 0x02 on the stack
push 0x03       ; push value at 0x03 on the stack
add             ; pop two values, perform an addition, then push the result
add             ; pop two values, perform an addition, then push the result
add             ; pop two values, perform an addition, then push the result
```

Machine Code:

```
0x00 0x00 0x00 0x01 0x00 0x02 0x00 0x03 0x01 0x01 0x01
```

| Opcode | Mnemonic |
|---|---|
| 0x00 | push |
| 0x01 | add |

# Stack Machine VM

```
enum Opcode
{
    PUSH = 0x0,
    ADD = 0x1
};
```

# Stack Machine VM

```
std::vector<uint8_t> program;
std::vector<uint8_t> heap;
std::stack<uint8_t> stack;
```

# Stack Machine VM

```
for (int i = 0; i < program.size(); ++i)
{
    const uint8_t opcode = program[i];
    switch (opcode)
    {
    case PUSH:
    {
        i++; // Align the operand
        const size_t address = (size_t)program[i];
        const uint8_t heapData = heap[address];
        stack.push(heap[address]);
        break;
    }

    case ADD:
    {
        const uint8_t rhs = stack.top();
        stack.pop();
        const uint8_t lhs = stack.top();
        stack.pop();
        stack.push(lhs + rhs);
        break;
    }

    default:
    {
                  return 1;
    }
    }
}
```

# Stack Machine VM

```
return stack.empty() ? 0 : stack.top();
```

# Stack Machine VM

```
// Filename: memory.bin
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000  01 02 03 04
```

```
// Filename: program.bin
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000  00 00 00 01 00 02 00 03 01 01 01
```

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

clang .\main.cpp -o Stack_VM.exe
.\Stack_VM.exe .\program.bin .\memory.bin
$LASTEXITCODE
10
```

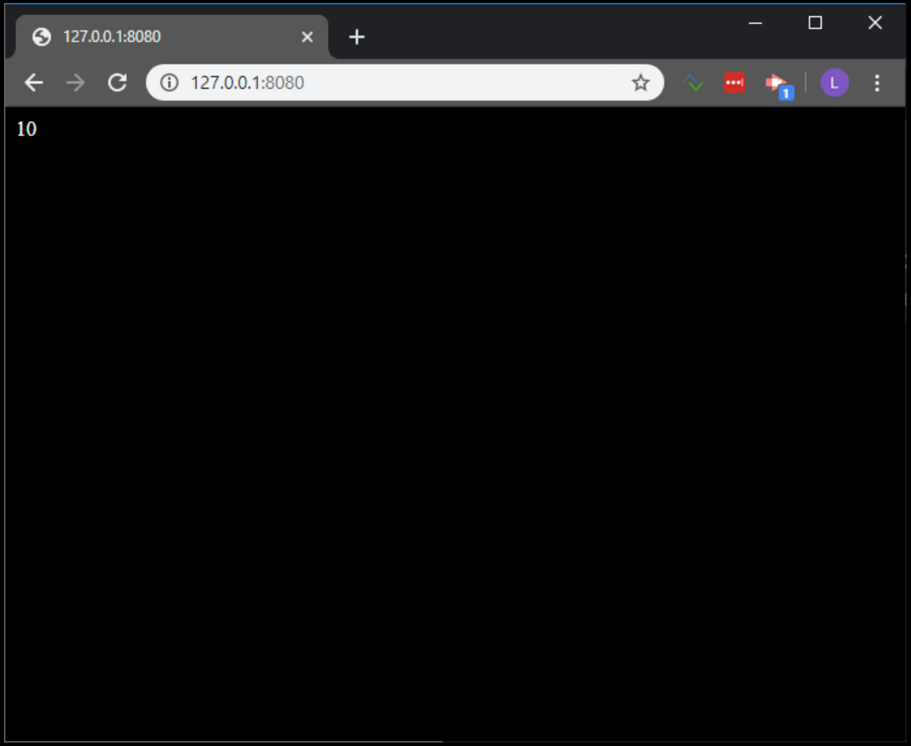
# Stack Machine VM (In WASM!)

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

emcc -o Stack_VM.js main.cpp -s "EXPORTED_FUNCTIONS=['_test’]”
--preload-file .\memory.bin --preload-file .\program.bin
```

```
// Filename: index.html
<!DOCTYPE html>
<script src="Stack_VM.js"></script>
<script>
  Module.onRuntimeInitialized = () => {
    const r = Module._test();
    document.body.innerHTML += `<div>${r}</div>`;
  };
</script>
```

# Stack Machine VM (In WASM!)



```
10
```

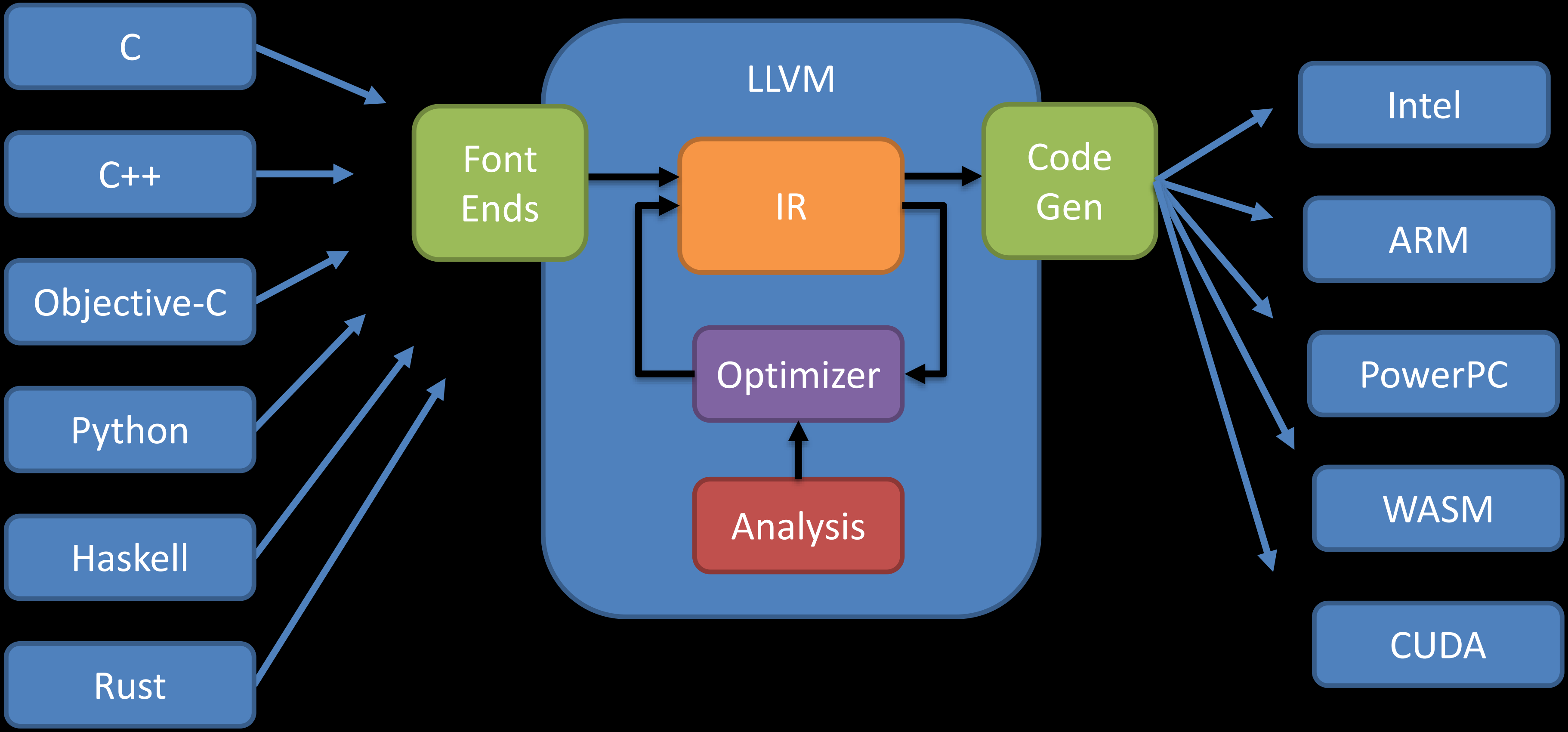
# Emscripten

- LLVM Toolchain
- Provides STL
- OpenGL (wraps WebGL)
- Thread Management (wraps webworkers)

https://emscripten.org/

# LLVM (Low Level Virtual Machine)

- Open Source Compiler Back-end
- Modular



# Limitations

- Debugging is tricky (although we can compile natively)
- API Limitations (Networking, File Systems)
- Zero-copy to JS
- Exception catching
- Memory limit

# Real world examples

- Google Earth
- Perspective
- D3
- PSPDFKit

# Do we still need JavaScript?

YES!

**THANK YOU!**

**Questions?**