

# Web-basierte Systeme

## 07: WebAssembly

---

Wintersemester 2022

Rüdiger Kapitza



Lehrstuhl für Verteilte Systeme  
und Betriebssysteme



Friedrich-Alexander-Universität  
Technische Fakultät

# Vorlesungsplan

- 2. November** Einführung und Darstellung von Webseiten (HTML und CSS)
- 9. November** Hypertext Transfer Protocol
- 16. November** Browser Schnittstellen
- 23. November** Kommunikationsschnittstellen im Browser
- 30. November** WebAssembly
- 7. Dezember** **Architektur moderner Browser**
- 14. Dezember** Clientseitige Architekturmuster
- 21. Dezember** Vorbereitung Papieranalyse
- 11. Januar** Papieranalyse
- 18. Januar** Serverseitige Implementierung von Web-basierten Systemen
- 25. Januar** Caching bzw. Lastverteilung durch Zwischenspeicher
- 1. Februar** Thema noch offen – wahrscheinlich WebRTC
- 8. Februar** Zusammenfassung und Ausblick

# WebAssembly

---

## Zielsetzung der Lerneinheit

- Bestandsaufnahme der bisherigen Ansätze für clientseitigen Code und Ermittlung der Motivation für die Entwicklung von WebAssembly
- Basiskenntnis des grundsätzlichen Aufbaus und der Funktionsweise von WebAssembly
- Verständnis wie WebAssembly in Webanwendung eingebunden wird

## Motivation

- Webanwendungen werden immer komplexer und anspruchsvoller
  - Interaktive Visualisierung von 3D Objekten
  - Audio- und Videosoftware
  - Spiele!
- Obwohl JavaScript an Geschwindigkeit über die Jahre gewonnen hat ist die Ausführungsgeschwindigkeit oft deutlich langsamer im Vergleich zu nativen Programmcode!

## Status Quo clientseitiger Code

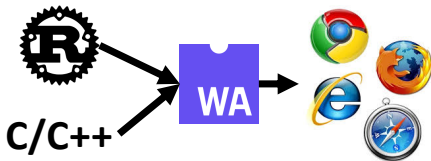
- JavaScript aktuell/bisher alternativlos
  - Ursprünglich schlechte Performanz wurde durch Just-in-Time-Compiler (JIT-Compiler) verbessert
  - Der Einsatz eines JIT-Compiler kostet aber auch Zeit und teilweise müssen die Laufzeitoptimierungen wieder verworfen werden
- Alternative Programmiersprachen bzw. Codeformate (mittels Plug-Ins) konnten sich nicht durchsetzen
  - Vgl. ActiveX, Java und Flash
- Letzter Trend: Transformation anderer Sprachen in JavaScript
  - Motivation sind besser Sprachkonzepte in der Ausgangssprache
    - Z.B. TypeScript - für bessere Typenprüfung und schnelleres Fehler identifizieren beim Programmieren
  - Höhere Ausführungsgeschwindigkeit
    - Transcompiler wandelt C/C++ in asm.js welches eine Teilmenge von JavaScript darstellt eine Voroptimierung ermöglicht

## Was ist WebAssembly?

- WebAssembly stellt ein **sicheres, schnell auszuführendes, portables Bytecode-Format für (aber nicht nur für) Webanwendungen** dar
- Sicherheit wurde bisher durch Laufzeitumgebungen für entsprechende Programmiersprachen erzielt, diese sind aber nicht auf portablen *low-level* Code ausgerichtet
  - Bsp.: C/C++ Binärcode ist schnell aber schwerer zu *verwalten/abzusichern*
- Die Performanz eines *ahead-of-time* kompilierten Codes steht oft in keinem Vergleich zu Programmiersprachen die mittels einer Laufzeitumgebungen (mit Sandbox) ausgeführt werden
- Portabilität & deterministisches Verhalten ist immens wichtig und folglich muss das Format hardware- und plattformunabhängig sein!
- Ein kompaktes Format ist für schnelles Laden von Code bei komplexen Anwendungen unabdingbar

## Was ist WebAssembly?

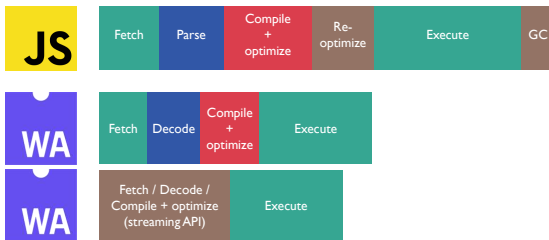
- WebAssembly realisiert eine virtuelle ISA (Instruction Set Architecture), diese kann das Übersetzungsziel verschiedener Programmiersprachen sein
- Aktuell: C/C++ und Rust
  - (Sprachen mit statischer Speicherverwaltung!)





## WebAssembly vs. JavaScript: Laden und ausführen

- Im Vergleich zu JavaScript <sup>1</sup> kann WebAssembly schneller geladen und ausgeführt werden
- Auch ein nebenläufiges Laden und Dekodieren ist möglich!



<sup>1</sup>Phasen laufen teilweise mehrfach und nebenläufig ab.

## Codeformat(e)

- Low-level Bytecode-Format mit **binär und textueller Darstellung**
- Verlustfrei ineinander überführbar
- Typen: 32-bit und 64-bit Integer werden unterstützt (i32, i64), sowie Fließkommazahlen (f32, f64)

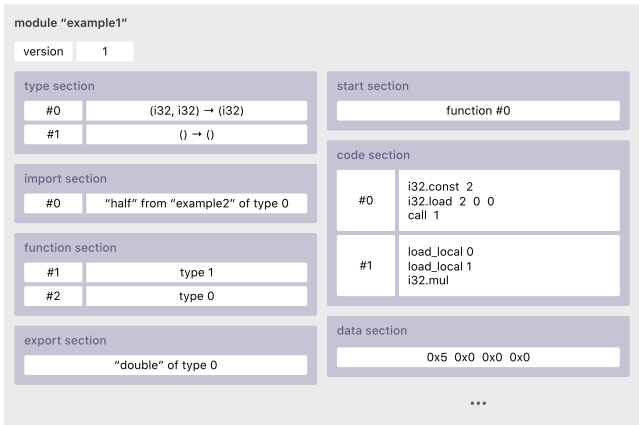
### Binärformat

```
0061 736d 0100 0000 0187 8080 8000 0160
027f 7f01 7f03 8280 8080 0001 0004 8480
8080 0001 7000 0005 8380 8080 0001 0001
0681 8080 8000 0007 9080 8080 0002 066d
656d 6f72 7902 0003 6164 6400 000a 8d80
8080 0001 8780 8080 0000 2001 2000 6a0b
```

### Textformat

```
1 (module
2   (table 0 anyfunc)
3   (memory $0 1)
4   (export "memory" (memory $))
5   (export "add" (func $add))
6   (func $add (; 0 ;) (param $0 i32) (param $1 i32) (
7     result i32)
8     (i32.add
9       (get_local $1)
10      (get_local $0)
11    )
12 )
```

## Anatomie eines WebAssembly Moduls (entspricht oft einem Programm)



## Anatomie eines WebAssembly Moduls

### 1. **Type** — Function signature declarations

- Enthält alle im Modul verwendeten Funktionen

```
1 (i32 i32 -> i32) // func_type #0
2 (i64 -> i64)     // func_type #1
3 ( -> )           // func_type #2
```

### 2. Import — Import declarations

- Führt alle externen Abhängigkeiten auf
- Hierfür wird der Modulname sowie der Feldname und -typ genannt

```
1 ("dumb-math", "quadruple", (func_type 1)) // func #0
2 ("dumb-math", "pi", (global_type i64 immutable))
```

- Es ist Aufgabe des Browsers (WASM Runtime) die Abhängigkeiten aufzulösen und ebnet den Weg für dynamisches Linken
- Dies ermöglicht ein Einbinden anderer Module aber auch eine Nutzung externer Funktionen (z.B. JavaScript)

## Anatomie eines WebAssembly Moduls

### 3. **Function** — Function declarations

- Definiert den Index der verwendeten Funktionen welche durch das Codesegment implementiert werden
- Index beginnt unter Aufschlag der Anzahl der importierten Funktionen
- Der Index (**func #N**) wird später verwendet um Funktion aufzurufen

```
1 (func_type 1) // func #1
2 (func_type 1) // func #2
3 (func_type 0) // func #3
```

### 4. **Table** — Indirect function table and other tables

- Enthält eine oder mehrere Tabellen
- Tabellen enthalten Element auf die aus WebAssembly nicht direkt zugegriffen werden können
  - Beispiele: JavaScript-Objekte und Dateideskriptoren
- Hier wird eine Bindung zwischen dem low-level (& nicht vertrauenswürdigen) linearen Speicher und den Hochsprachenfunktionen und -referenzen hergestellt

## Anatomie eines WebAssembly Moduls

### 5. Memory — Memory attributes

- Hier kann optional Speicher für das Modul alloziert werden
- Es wird die initiale Größe und die erwartete maximale Größe angegeben

### 6. Global — Global declarations

- Es werden alle globalen Variablen deklariert.
- Vgl. `static` im Falle von C/C++

## Anatomie eines WebAssembly Moduls

### 7. Export — Exports

- Es werden alle extern zugreifbaren Elemente des Module definiert

```
1 ("half" (func 1))
```

- In diesem Fall kann `half` in Kombination mit der `function`- und der `type`-Sektion kombiniert werden zu:

```
1 function half(arg0 :int64) :int64
```

- Neben Funktionen können auch noch Tabellen, Speichersegmente und globale Variablen exportiert werden

## Anatomie eines WebAssembly Moduls

### 8. Start — Start function declaration

- Legt eine Funktion fest welche im Rahmen des Ladens ausgeführt wird
- Diese kann als *main()*-Funktion dienen oder nur dazu verwendet werden die globale Variablen und die zuvor festgelegten Speicherregionen zu initialisieren

### 9. Element — Elements section

- Zuständig für die Initialisierung der von außen importierten oder zuvor definierten Tabellen



## Anatomie eines WebAssembly Moduls

### 10. Code — Function bodies

- Enthält den Code aller Funktionen des Moduls
- Funktionen sind der Reihenfolge nach angegeben wie in der Function-Sektion angegeben
- Beispielfunktion: `half`

```
1   get_local 0 // push parameter #0 on stack (our dividend)
2   i64.const 2 // push constant int64 "2" on stack (our divisor)
3   i64.div_u   // unsigned division; pushes result onto stack
4   end        // ends function, resulting in one i64 (top of stack)
```

- WebAssembly definiert eine abstrakte Stackmaschine, diese kann dann entsprechend optimiert auf eine Zielarchitektur abgebildet werden
- Es gibt eine Vielzahl von Standardoperatoren (z.B. `div` und `add`) aber auch spezifische wie bspw. `eqz` um zu testen ob ein Operand 0 ist

## Anatomie eines WebAssembly Moduls

### 11. Data – Data segments

- Die wird verwendet um importierte oder lokal definierte Speicherbereiche zu belegen

```
1 (data_segment
2   0 // linear memory index
3   (init_expr (i32.const 4)) // byte offset at which to place the data
4   (data 0x2a 0x0 0x0 0x0))
```

- Das Beispiel belegt im Speicherbereich mit dem Index 0 die Bytes [4–8] mit einem unsignierten i32 Wert (42)

## Kernkonzepte für die Verbindung zwischen WebAssembly und JavaScript

- **Module:** Repräsentiert ein im Browser kompiliertes binäres Blob. Er ist zustandslos und kann zwischengelagert werden (z.B. IndexedDB) oder auch zwischen Web Workern geteilt werden.
- **Memory:** Stellt einen variablen `ArrayBuffer` dar, welcher durch die Basisinstruktionen von WebAssembly gelesen und geschrieben werden kann.
- **Table:** Ein Array variabler Größe für Referenzen (z.B., um Funktionen zu verwalten). Zielsetzung ist Portabilität und Sicherheit.
- **Instance:** Eine Instanz eines Moduls mit initialisiertem Zustand und aufgelösten Importen.

## Übersetzen und Ausführen

- Verschiedene Wege möglich, hier mittels Emscripten<sup>2</sup>



1. Emscripten reicht die C/C++ Dateien intern an clang+LLVM weiter
2. clang+LLVM generiert eine Zwischenrepräsentation welche Emscripten in eine .wasm-Binärdatei umsetzt
3. WebAssembly kann bisher nicht direkt auf den DOM zugreifen, sondern nur Funktionen in JavaScript aufrufen und dabei Integer und Fließkommazahlen austauschen.
  - Es können nur über JavaScript Web APIs aufgerufen werden.
  - Emscripten erzeugt hierfür die nötigen Hilfsfunktionen

<sup>2</sup><http://kripken.github.io/emscripten-site/index.html>

## Übersetzen und Ausführen

- Emscripten stellt eine Reihe populäre C/C++ Bibliotheken bereit
  - Z.B. SDL, OpenGL, OpenAL und einen Teil der Posix-Schnittstelle
  - Teile dieser Bibliotheken müssen mit den Web APIs des Browser verbunden werden
- Hilfsfunktionen müssen auch zum Laden und Einbinden der Bibliotheken bereitgestellt werden
- Bspw. werden Ausgaben über `stdout` in ein `<textarea>` umgeleitet
- Im Falle von OpenGL wird entsprechend ein `<canvas>` genutzt

## Direkte Erstellung eigener WebAssembly-Module

- Wie schon angemerkt kann die Textform direkt in die Binärform umgesetzt werden
- Im Prinzip ist es also möglich direkt aus JavaScript WebAssembly-Module zu generieren, kompilieren und auszuführen

## Einfaches Beispielprogramm

### ■ `hello.c` als Klassiker

```
1 #include <stdio.h>
2
3 int main(int argc, char ** argv) {
4     printf("Hello World\n");
5 }
```

### ■ Verwendung der Emscripten-Werkzeugkette

```
1 emcc hello.c -s WASM=1 -o hello.html
```

- `-s WASM=1` legt die Erzeugung von WebAssembly fest
- `-o hello.html` erstellt eine einfach zu verwendende Vorlage mit Hilfsfunktionen
- Als Ergebnis werden drei Ausgabedateien erstellt:  
`hello.wasm/.js/.html`

## Aufruf einer WebAssembly-Funktion

- Um Funktionen aus JavaScript verfügbar zu machen müssen sie entsprechend markiert werden via `EMSCRIPTEN_KEEPALIVE`

```
1  #include <stdio.h>
2  #include <emscripten/emscripten.h>
3  int main(int argc, char ** argv) {
4      printf("Hello World\n");
5  }
6  #ifdef __cplusplus
7  extern "C" {
8  #endif
9  EMSCRIPTEN_KEEPALIVE void myFunction() {
10     printf("MyFunction Called\n");
11 }
12 #ifdef __cplusplus
13 }
14 #endif
```

- Damit das Modul langfristig verfügbar bleibt muss mit `NO_EXIT_RUNTIME=1` übersetzt werden, da sonst nach dem Ablauf der `main()` die Modulinstanz gelöscht wird



## Aufruf einer WebAssembly-Funktion

- Nun muss man noch die Funktion aus JavaScript aufrufen – hier als Beispiel falls ein Knopf ausgelöst wird:

```
1 document.querySelector('.mybutton').addEventListener('click', function(){
2   alert('check console');
3   var result = Module.ccall('myFunction', // name of C function
4                             null, // return type
5                             null, // argument types
6                             null); // arguments
7 });
```

## Laden eines WebAssembly-Modules

- Im Prinzip gibt es zwei Vorgehensweisen - entweder klassisch Laden und Übersetzen

```
1 fetch('simple.wasm').then(response =>
2   response.arrayBuffer()
3 ).then(bytes =>
4   WebAssembly.instantiate(bytes, importObject)
5 ).then(results => {
6   results.instance.exports.exported_func();
7 });
```

- ...oder Verwendung der neuen Streaming-API

```
1 WebAssembly.instantiateStreaming(fetch('simple.wasm'), importObject)
2 .then(obj => obj.instance.exports.exported_func());
```

- Der Rückgabewert von `WebAssembly.instantiate()` liefert

```
1 {
2   module : Module // The newly compiled WebAssembly.Module object,
3   instance : Instance // A new WebAssembly.Instance of the module object
4 }
```

## Verwendung von Speicher in WebAssembly

- Jedes Modulinstanz hat seinen *eigenen* Speicherbereich, dessen Adressierung bei der Adresse 0 beginnt
- Zielsetzung ist eine bessere Isolation der verschiedenen Modulinstanzen
- Aus Sicht von JavaScript kann der Speicher einer Modulinstanz als `ArrayBuffer` variabler Größe betrachtet werden
- Man kann den Speicher für WebAssembly auch in JavaScript erzeugen via `WebAssembly.Memory()`
  - Zwei Parameter können übergeben werden: initiale und maximale Größe
  - Es wird dabei in Vielfachen von 64KB allokiert
  - Beispiel:

```
1 var memory = new WebAssembly.Memory({initial:10, maximum:100});
```

## Verwendung von Speicher in WebAssembly

- Auf den Speicher einer Modulinstanz kann mittels einfacher getter/setter-Funktionen zugreifen

```
1 // Set 42 ...
2 new Uint32Array(memory.buffer)[0] = 42;
3 // and get it from ...
4 new Uint32Array(memory.buffer)[0]
```

- Der allokierte Speicher kann bis zur Maximalkapazität vergrößert werden – darüber hinaus wird eine `WebAssembly.RangeError` Ausnahme geworfen

```
1 memory.grow(1);
```

## Verwendung von Speicher in WebAssembly

- Im Prinzip kann Speicher in beiden Welten erzeugt und jeweils mit der anderen Seite ausgetauscht werden
- Beispiel in JavaScript:

```
1 <script>
2 var memory = new WebAssembly.Memory({initial:10, maximum:100});
3 WebAssembly.instantiateStreaming(fetch('memory.wasm'), { js: { mem: memory } })
4 .then(obj => {
5   var i32 = new Uint32Array(memory.buffer);
6   for (var i = 0; i < 10; i++) {
7     i32[i] = i;
8   }
9   var sum = obj.instance.exports.accumulate(0, 10);
10  console.log(sum);
11 });
12 </script>
```

## Verwendung von Speicher in WebAssembly

- Warum ist es sinnvoll den Speicher im Prinzip von beiden Seiten aus bereitstellen zu können?
  - Man kann wie im Beispiel den Speicher in JavaScript initialisieren während das Modul noch geladen und kompiliert wird
  - Man kann Speicher auch in verschiedene Modulinstanzen importieren (z.B. für dynamisches Linken)

## Verwendung von Tabellen

- Tabellen sind aus Sicherheitsgründen der einzige Mechanismus um Funktionen aufzurufen
- Ähnlich zur Manipulation von Speicher können auch Tabellen in JavaScript ausgelesen, verändert oder vergrößert werden
- Auch diese Funktionalität wird für das dynamische Linken benötigt

## Caching

- Komplexe Anwendungen verfügen über viel Programmcode – der auch noch auf der Zielplattform kompiliert werden muss
- Idee: WebAssembly Module in ihrer kompilierten Form in einer IndexedDB ablegen
- Versionierung ist hierbei wichtig, damit auch immer die neuste Version einer Anwendung verwendet wird
- Lösungsansatz kann hier gefunden werden:
  - [https://developer.mozilla.org/en-US/docs/WebAssembly/Caching\\_modules](https://developer.mozilla.org/en-US/docs/WebAssembly/Caching_modules)

## Zusammenfassung

- WebAssembly bildet ein portables, plattformunabhängiges Bytecodeformat welches speziell für den Einsatz im Web
  - Aktuell wird es von allen weiterverbreiteten Browsern unterstützt
  - Es eröffnet den Einsatz von neuen Programmiersprachen im Web und die Nutzung von existierendem Code
  - Durch Ahead-Of-Time Übersetzung und den Aufbau des Formats kann WebAssembly schnell geladen und ausgeführt werden
- Trotz breiter Verfügbarkeit steckt das Format noch im Anfang
  - Bspw. Multithreading?
- Weitere Anwendungszwecke ausserhalb des Webs sind sehr gut vorstellbar



## Referenzen

- `https://developer.mozilla.org/en-US/docs/WebAssembly`
- `https://webassembly.github.io/spec/`

## Literatur

---

- [1] Andreas Haas et al. “Bringing the Web Up to Speed with WebAssembly”. In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2017. Barcelona, Spain: ACM, 2017, S. 185–200.