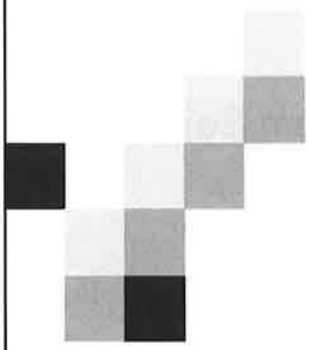


Automatisches Übersetzen von 6502-Spielen

Norbert Kehrer

Augsburg, 8. November 2014

Klassische Spiele kann man automatisiert portieren

- Ziel: Originalgetreue Umsetzung klassischer Videospiele z.B. im Browser → ein Beispiel: Asteroids
- Der traditionelle Weg: Emulatoren
- Eine interessante Alternative: Statische Binärcodeübersetzung (Static Binary Translation, Static Recompilation)

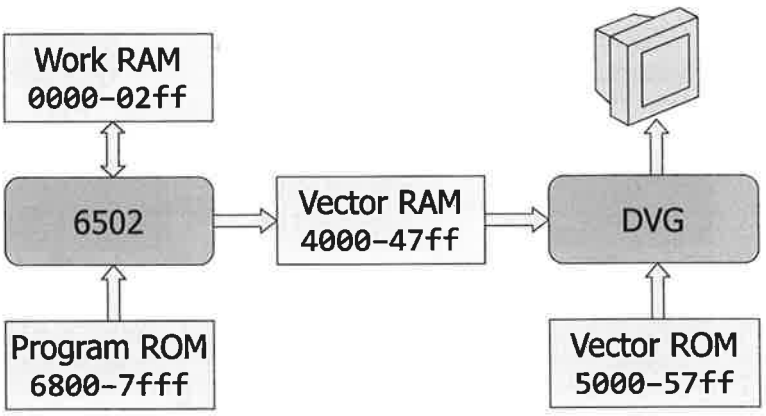
Was erwartet euch?

- Emulieren vs. Rekompilieren am Beispiel „Asteroids“
- Die Quelle: der 6502
- Das Ziel: JavaScript
- „Naive“ Übersetzungsmuster
- Code-Optimierung und der aufregende Weg bis tief in die Compiler-Theorie

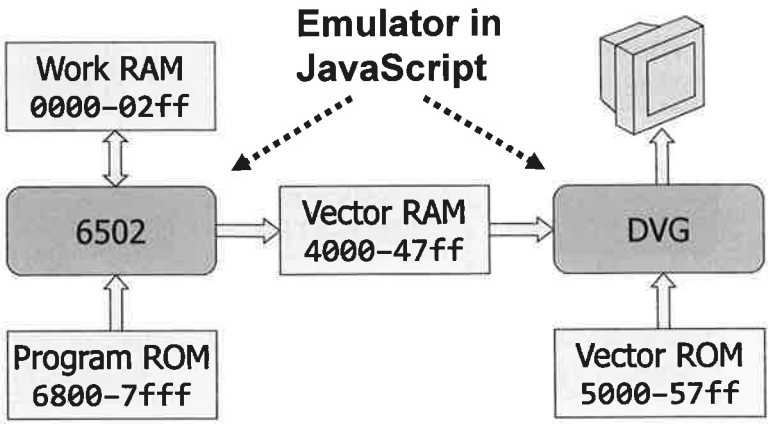
6502 in JavaScript übersetzen

- Emulieren vs. Rekompilieren am Beispiel „Asteroids“
- Die Quelle: der 6502
- Das Ziel: JavaScript
- „Naive“ Übersetzungsmuster
- Code-Optimierung und der aufregende Weg bis tief in die Compiler-Theorie

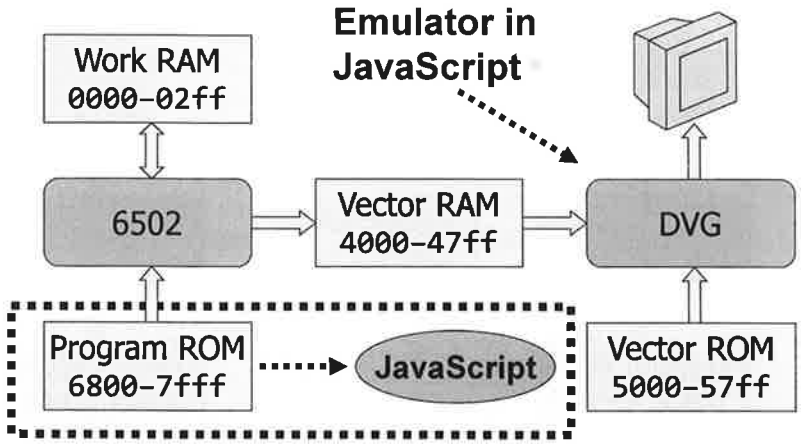
Der Asteroids-Arcade-Automat basiert auf dem 6502-Prozessor



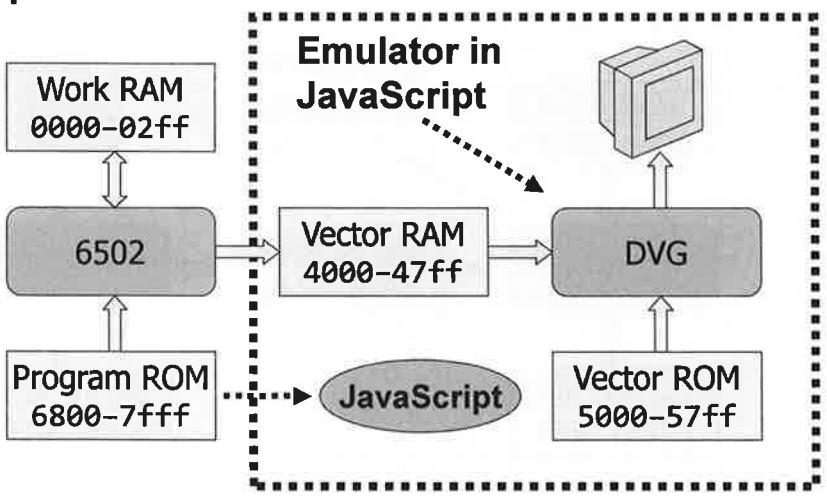
Klassisches Emulieren der Hardware ...



... oder das Spielprogramm in JavaScript übersetzen, …



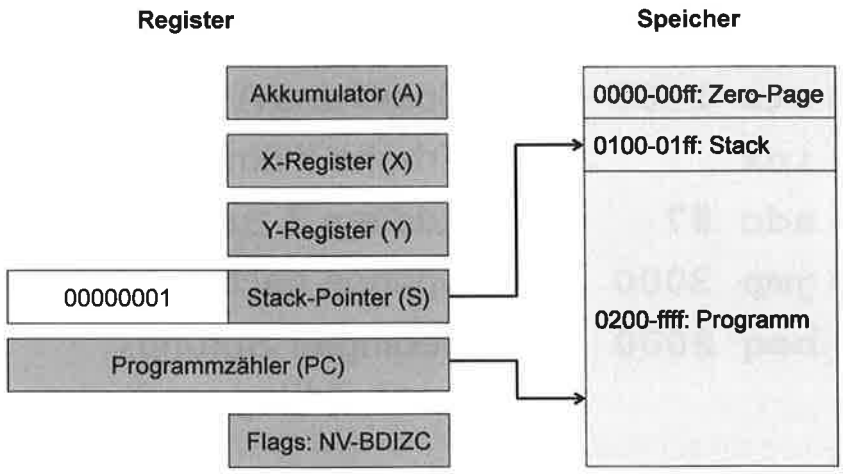
… und so eine Stand-Alone-Applikation schaffen



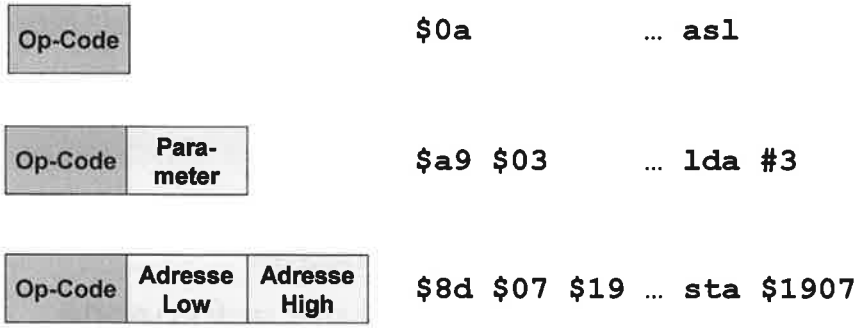
6502 in JavaScript übersetzen

- Emulieren vs. Rekompilieren am Beispiel „Asteroids“
- Die Quelle: der 6502
- Das Ziel: JavaScript
- „Naive“ Übersetzungsmuster
- Code-Optimierung und der aufregende Weg bis tief in die Compiler-Theorie

Der 6502 hat wenige Register



Der 6502 hat 3 Befehlsformate



Einige Befehlsbeispiele

`lda #1`	Lade Akku mit 1
`sta 1000`	Speichere A in 1000
`inx`	Erhöhe x um 1
`adc #7`	Addiere 7 zum Akku
`jmp 3000`	Springe nach 3000
`beq 2000`	Bedingter Sprung, wenn Z-Flag == 0

6502 in JavaScript übersetzen

- Emulieren vs. Rekompilieren am Beispiel „Asteroids“
- Die Quelle: der 6502
- Das Ziel: JavaScript
- „Naive“ Übersetzungsmuster
- Code-Optimierung und der aufregende Weg bis tief in die Compiler-Theorie

JavaScript ist ein C-Derivat

- Wichtigste Skriptsprache für Browser
- HTML-File + JS-File = Web-App
- Fast gleiche Syntax wie C und Java
- Variablen: `var x;`
- Arrays:
```
var a = new
    Array(100);
```
- Zuweisungen: `n = a[19]; k = 7;`
- Arithmetik: `x = x + 5 * a[4];`

Bedingte Anweisungen und Auswahlanweisung wie in C

- **`if`** (*bed*) { *anw1* } **`else`** { *anw2* };

- **`switch`** (*var*) {
  `case` *a*: *anw1*; **`break;`**
  `case` *b*: *anw2*; **`break;`**
  `case` *c*: *anw3*; **`break;`**
  ...
  };

6502 in JavaScript übersetzen

- Emulieren vs. Rekompilieren am Beispiel „Asteroids“
- Die Quelle: der 6502
- Das Ziel: JavaScript
- „Naive“ Übersetzungsmuster
- Code-Optimierung und der aufregende Weg bis tief in die Compiler-Theorie

Zuerst einen Disassembler

Programm-ROM
(aus Asteroids)

```
...
A2 44
A9 02
85 02
86 03
...
```

Disassembler-Listing

```
...
6834: A2 44    ldx #$44
6836: A9 02    lda #$02
6838: 85 02    sta $02
683A: 86 03    stx $03
...
```

Daraus den Code-Generator

Programm-ROM
(aus Asteroids)

```
...
A2 44
A9 02
85 02
86 03
...
```

JavaScript-Code statt Disassembler-Listing

```
...
... ~~ldx #$44~~   x=44;
... ~~lda #$02~~   a=2;
... ~~sta $02~~    mem[2]=a;
... ~~stx $03~~    mem[3]=x;
...
```

6502-Register und Speicher werden zu Variablen und Arrays

Akkumulator ⟶ `var a;`

X-Register ⟶ `var x;`

Y-Register ⟶ `var y;`

C-Flag ⟶ `var c;`

N-Flag ⟶ `var n;`

...

Speicher ⟶
```
var mem = new
          Array(65536);
```

„Normale“ Befehle sind einfach übertragbar

`lda 1000`	⟶	`a = mem[1000];`
`sta 1001`	⟶	`mem[1001] = a;`
`inc 1000`	⟶	`mem[1000] = (mem[1000]+1)&0xff;`
`ldx #10`	⟶	`x = 10;`
`sta 2000,x`	⟶	`mem[2000+x] = a;`
`inx`	⟶	`x = (x+1) & 0xff;`

„GOTO Considered Harmful“ Considered Harmful

```
...
1000 ldx #0        ; x = 0
1002 inx           ; erhöhe x
1003 stx $d020     ; schreibe x
1006 jmp 1002      ; springe nach 1002
...
```

Aber JavaScript hat kein (richtiges) GOTO !

Ein alter „Fortran-to-C“-Trick

```
pc = 1000;
while(true) {
  switch (pc) {
    case 1000: x = 0;              //ldx
    case 1002: x = (x+1) & 0xff; //inx
    case 1003: mem[0xd020] = x;  //stx
    case 1006: pc = 1002; break; //jmp
    ...
  };
};
```

Case-Labels braucht man nur für Sprungziele

```
pc = 1000;
while(true) {
  switch (pc) {
    case 1000: x = 0;              //ldx
    case 1002: x = (x+1) & 0xff; //inx
    case 1003: mem[0xd020] = x;  //stx
    case 1006: pc = 1002; break; //jmp
    …
  };
};
```

Bedingte “Branches” werden zu If-Anweisungen

```
    …
    if (z == 1) {     // beq 3000
        pc = 3000;
        break;
    };
    …
case 3000:
    …
```

Dr. Sheldon Cooper's „Fun with Flags"

- Als Nebeneffekt vieler 6502-Befehle werden Flags gesetzt
- Beispiel:

```
lda 1000  ⟶ wenn null → Z=1 sonst Z=0
          ↘ wenn neg. → N=1 sonst N=0
beq 4711
...
```

Viele Befehle brauchen daher auch Flag-Berechnungscode

```
lda 1000 ⟶ a = mem[1000];
```

```
lda 1000 ⟶ a = mem[1000];
           if (a==0) z=1; else z=0;
           if (a<0)  n=1; else n=0;
```

→ Korrekte, aber sehr große Kompilate

6502 in JavaScript übersetzen

- Emulieren vs. Rekompilieren am Beispiel „Asteroids“
- Die Quelle: der 6502
- Das Ziel: JavaScript
- „Naive“ Übersetzungsmuster
- Code-Optimierung und der aufregende Weg bis tief in die Compiler-Theorie

Flag-Berechnungen sind sehr oft überflüssig …

```
lda 1000 ⟶ a = mem[1000];
           if (a==0) z=1; else z=0;
           if (a<0)  n=1; else n=0;

ldx 1200 ⟶ x = mem[1200];
           if (x==0) z=1; else z=0;
           if (x<0)  n=1; else n=0;

beq 4711 ⟶ if (z==1) …
```

Flag-Berechnungen sind sehr oft überflüssig …

```
lda 1000 ⟶  a = mem[1000];
             if (a==0) z=1; else z=0;
             if (a<0)  n=1; else n=0;

ldx 1200 ⟶  x = mem[1200];
             if (x==0) z=1; else z=0;
             if (x<0)  n=1; else n=0;

beq 4711 ⟶  if (z==1) …
```

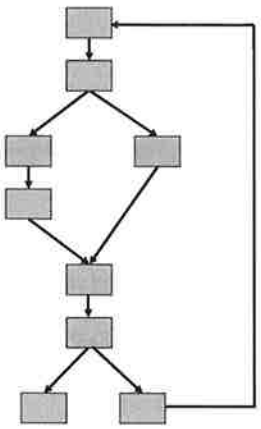
… aber nicht immer

```
lda 1000 ⟶  a = mem[1000];
             if (a==0) z=1; else z=0;
             if (a<0)  n=1; else n=0;

ldx 1200 ⟶  x = mem[1200];
        ja:  if (x==0) z=1; else z=0;
        ?:   if (x<0)  n=1; else n=0;

beq 4711 ⟶  if (z==1) …
```

Nur, wenn „später“ „gebraucht“ und nicht vorher neu berechnet



- „Später“ → Berechnung des Kontrollflussgraphen
- Zyklen im Graphen
- Use-Def-Mengen

Dead-Code-Elimination ist schwierig und interessant

- Liveness-Analysis-Problem
- Lösung über Fixpunktiteration
- Viele spannende Wege führen weiter: Weitere Optimierungen, Compilertheorie, gcc, LLVM, logische Programmierung, …
- Aber nicht mehr heute …

Zusammenfassung: Von Asteroids zur Liveness-Analysis

- Browser-„Asteroids“ als Beispiel
- 6502-Prozessor
- JavaScript
- Übersetzung von 6502 nach JavaScript
- Optimierung: Dead-Code-Elimination

Danke !

http://members.aon.at/nkehrer/