

Entwicklung eines Gameboy-Emulators

Thomas Witte

5. Juli 2015

1 Motivation / Emulation

Der ab 1989 von Nintendo hergestellte Gameboy ist mit über 100 Millionen verkauften Einheiten eine der erfolgreichsten Spielekonsolen überhaupt.

Da die meisten für den Gameboy erschienenen Spiele nur in binärer Form als ROM-Abbild vorliegen, ist eine Portierung auf aktuelle Systeme ausgeschlossen. Eine Alternative bietet die Emulation der Gameboy-Architektur: es wird eine möglichst exakt dem Gameboy gleichende Laufzeitumgebung bereitgestellt, die in der Lage ist, das unmodifizierte Programm auszuführen. Dazu müssen neben der CPU auch sämtliche anderen Schnittstellen (Ein-/Ausgabe, Speicher, etc.) bereitgestellt werden.

Aufgrund der inkompatiblen Prozessorarchitekturen kann die Instruktionsfolge des emulierten Programms nicht direkt ausgeführt werden: entweder wird sie Instruktion für Instruktion interpretiert, also der Fetch-Execute-Cycle in Software durchgeführt, oder in kompatible Instruktionen übersetzt. Das zweite Verfahren – oft „dynarec“ oder „jit-compiler“ genannt – wird in vielen Emulatoren aufgrund der potentiell höheren Geschwindigkeit verwendet [3].

Die Übersetzung der Instruktionen geschieht meist dynamisch zur Laufzeit, da durch statische Analyse nur schwer – zB durch verfolgen aller möglichen Ausführungspfade ab einem bekannten Einsprungpunkt – zwischen Daten und Programmtext unterschieden werden kann. Selbstmodifizierender Code und das Anspringen von zur Laufzeit berechneten Adressen macht bei statischer Übersetzung zusätzlich häufig eine Rückfall auf Interpretation oder dynamische Übersetzung zur Laufzeit notwendig [9].



Abbildung 1: Gameboy der ersten Generation.

Der im Zuge des Projekts entstandene Emulator „dynasmgb“ führt daher eine dynamische Übersetzung der Prozessorinstruktionen aus. Sämtliche anderen Schnittstellen (Grafik, Sound, Speicher) werden zusätzlich durch Interpretation der Adressraums emuliert.

2 Gameboy Architektur

Die folgenden Abschnitte fassen Aufbau und Funktionsweise des klassischen Gameboys zusammen. Auf die Unterschiede zum Gameboy Color und Super Gameboy wird nicht eingegangen. Details finden sich in [4][1][10].

2.1 Hardware

Herzstück des Gameboy ist ein leicht modifizierter Z80 Prozessor, der mit 4,2MHz getaktet ist. Der Registersatz besteht – ähnlich auch einem Intel 8080 – aus 7 8bit Registern, einem Flagregister (Zero, Carry, Halfcarry, Subtract) sowie zweier 16bit Register (SP und PC). Die 8bit Register können durch bestimmte 16bit Instruktionen jeweils paarweise als 16bit Register genutzt werden.

Der Gameboy kann auf insgesamt 16kB internen Arbeitsspeicher zugreifen. Dabei sind 8kB dieses Speichers als Tile-RAM für Grafiken zweckgebunden. Der Memory Bank Controller (MBC) bestimmter Spiele kann bis zu 128kB zusätzlichen RAM innerhalb der Cartridge adressieren (16 Bänke à 8kB).

Das Spiel wird direkt aus dem ROM der Spiel-Cartridges gestartet. Über MBCs können dabei maximal 4MB (256 Bänke à 16kB) adressiert werden.

Als Display kommt ein Graustufen-LCD zum Einsatz. Es kann 4 Grautöne darstellen und besitzt eine Auflösung von 160×144 Pixel bei 60Hz. Die Pixel des Displays können dabei nicht einzeln angesprochen werden, sondern nur in 8×8 Pixel großen Kacheln (Tiles).

Dabei kann gleichzeitig eine Hintergrundkarte, eine Vordergrundkarte und bis zu 40 Sprites angezeigt werden.

Die Soundausgabe erfolgt über einen integrierten Lautsprecher oder den Kopfhörerausgang in Stereo. Der Gameboy bietet dazu vier Soundkanäle, die Rechteckschwingungen, Rauschen oder Wave-Samples erzeugen bzw. abspielen können.

Zur Eingabe stehen insgesamt acht Knöpfe zur Verfügung (vier Richtungstasten, A, B, Select und Start). Alternativ können Daten über eine serielle Schnittstelle gesendet oder empfangen werden.

2.2 Befehlssatz

Der Befehlssatz umfasst insgesamt 500 verschiedene Befehle variabler Instruktionslänge (1-3 Bytes). 244 Befehle nutzen das erste Byte für den Opcode und maximal zwei Bytes für Argumente. Die restlichen 256 Befehle werden durch das Prefixbyte 0xCB eingeleitet und besitzen somit einen 2Byte-Opcode und keine weiteren Argumente.

Die Ausführungszeit der Befehle beträgt jeweils zwischen 4 und 24 Taktzyklen. Damit beträgt der maximale Befehlsdurchsatz 1MOps/s.

Eine Übersicht sämtlicher Befehle befindet sich unter [5], sowie im Anhang.

2.3 Interrupts

Der Gameboy stellt insgesamt fünf verschiedene Interrupts zur Verfügung:

VBLANK Der VBLANK-Interrupt wird nach jedem dargestellten Bild dargestellt und markiert den Beginn der VBLANK-Phase in der für 4560 Taktzyklen frei auf den Videospeicher zugegriffen werden kann.

STAT Das STAT-Register (Speicheradresse 0xFF41) wechselt mit jeder dargestellten Bildzeile zwischen drei Zuständen und während der VBLANK-Phase auf einen vierten. Der STAT-Interrupt kann bei einem Wechsel dieser Zustände ausgelöst werden. Welche Zustandsübergänge betroffen sind, kann ausgewählt werden.

Timer Der Timer-Interrupt wird bei einem Überlauf des Timer-Registers (0xFF05) ausgelöst. Die Rate mit der das Timer-Register inkrementiert wird ist dabei auswählbar, sodass der Timer-Interrupt mit einer wählbaren Rate von 16Hz, 64Hz, 256Hz oder 1kHz auftritt.

Serial Der Serial Transfer-Interrupt wird beim Abschluss eines seriellen Transfers ausgelöst.

Joypad Bei jedem Tastendruck eines der acht Knöpfe wird der Joypad-Interrupt ausgelöst.

Tritt ein Interrupt auf, so wird er anhängig und ein Bit im Interrupt Flag-Register (0xFF0F) gesetzt. Über das Interrupt Enable Register (0xFFFF) kann ausgewählt werden, welche Interrupts aktiv sind. Das Interrupt Master Enable-Flag kann zusätzlich alle Interrupts abschalten. Es kann durch die Instruktionen DI (Disable Interrupts), EI (Enable Interrupts) oder RETI (Return from Interrupt) manipuliert werden.

Ist ein Interrupt anhängig, das entsprechende Bit im Interrupt Enable Register und das Interrupt Master Enable-Flag gesetzt, wird eine Handlerfunktion mit fester Startadresse zwischen 0x40 (VBLANK) und 0x60 (Joypad) aufgerufen und mittels des Interrupt Master Enable weitere Interrupts während der Behandlung unterbunden.

2.4 Adressraum

2.4.1 Memory Map

Der adressierbare Adressraum des Gameboys beträgt 64kB. In die unteren 32kB (0x0 – 0x7FFF) werden zwei Rombänke à 16kB gleichzeitig eingeblendet. Der Wechsel zwischen den Rombänken geschieht durch den MBC innerhalb der Cartridge. Alle verfügbaren MBCs lösen einen Wechsel der oberen Rombank (0x4000 – 0x7FFF) durch Schreibzugriffe auf bestimmte Adressen im ROM aus.

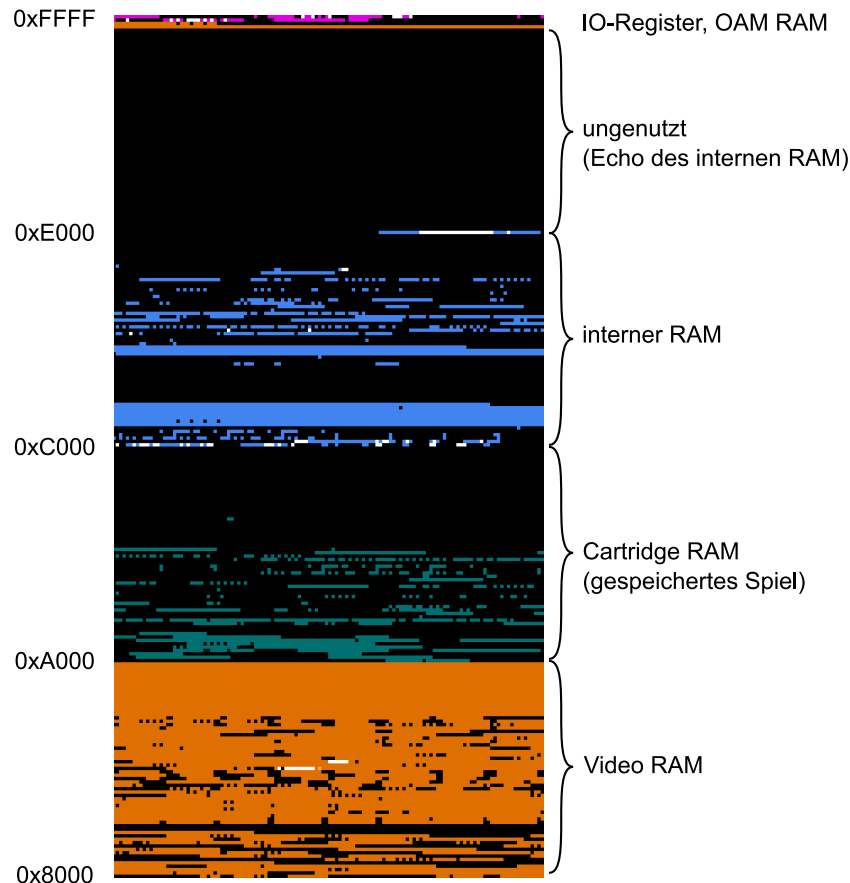


Abbildung 2: Visualisierung der oberen 32kB des Adressraums (Pokémon Blaue Edition)

Die Adressen zwischen 0x8000 und 0x9FFF bilden den Video-RAM. Er enthält 8×8 Pixel große Kacheln zu je 16 Byte, sowie Vordergrund und Hintergrund Tile Maps.

Zwischen 0xA000 und 0xBFFF wird der Cartridge RAM eingeblendet. Je nach MBC lassen sich eventuell mehrere Bänke tauschen. Dieser Speicher ist in einigen Spielcartridges durch eine Batterie versorgt und kann damit auch bei ausgeschaltetem Gameboy einen Spielstand halten.

Es folgen 8kB interner RAM (0xC000 – 0xDFFF), der fast vollständig ein zweites Mal im Adressbereich 0xE000 – 0xFDFD gespiegelt wird. Diese Adressen werden jedoch typischerweise nicht verwendet.

Die Adressen 0xFE00 bis 0xFE9F enthalten den OAM Speicher. Er enthält die Position, anzuzeigende Grafik, verwendete Graustufenpalette und Flags aller 40 Sprites. Per DMA-Transfer kann der OAM Speicher nebenläufig überschrieben werden.

Über den Adressbereich 0xFF00 bis 0xFF7F wird der Hardware IO gesteuert. Er enthält Register zur Kontrolle von Timern, Seriellen Übertragungen, DMA-Transfers, Soundausgabe und des anzuzeigenden Mapbereichs.

Im Anschluss befinden sich weitere 127 Byte Arbeitsspeicher (0xFF80 – 0xFFFFE), die jederzeit les- und schreibbar sind. Da während eines DMA-Transfers der gesamte

sonstige Speicher weder gelesen noch geschrieben werden kann, muss während eines solchen Transfers in diesen Speicherbereich gesprungen werden.

Das Interrupt Enable Register belegt die höchste Adresse 0xFFFF.

2.4.2 Startadressen

Die Ausführung des Programms beginnt an Adresse 0x100. Die Handlerfunktionen mit Startadressen zwischen 0x00 und 0x60 werden durch Restarts und Interrupts angesprungen.

2.4.3 ROM-Header

Die Rom-Adressen 0x104 bis 0x14F sind durch den ROM-Header belegt. Er enthält eine Grafik, die das Nintendo-Logo zeigt (0x104 bis 0x133). Diese wird auf realen Konsolen beim Start angezeigt und verglichen. Stimmt sie nicht exakt überein, startet der Gameboy das Spiel nicht. Die weiteren Bytes enthalten Spieltitel, Hersteller, Flags die anzeigen, ob das Spiel spezielle Funktionen für Gameboy Color oder Super Gameboy enthält, sowie Informationen über den verbauten MBC und die Anzahl der auf der Cartridge vorhandenen RAM- und ROM-Bänke.

2.5 Grafik

Die Pixel des Gameboydisplays können nicht einzeln angesprochen werden, sondern es werden immer ganze Kacheln von jeweils 8×8 Pixel Größe angezeigt. Neben einer Vordergrund- und Hintergrundkarte (WIN und BG genannt) die die Indizes der anzuzeigenden Kacheln enthalten, können bis zu 40 Sprites frei auf dem Display positioniert werden.

Das Bild wird zeilenweise von oben nach unten aufgebaut. Über das Register LY (0xFF44) kann die derzeit bearbeitete Zeile ausgelesen werden und über das STAT Register (0xFF41) ob derzeit ein Zugriff auf den Grafikspeicher möglich ist.

Die Größe der Vordergrund- und Hintergrundkarte beträgt 32 auf 32 Kacheln, sodass immer nur ein Ausschnitt auf dem Display sichtbar ist. Über die Register SCX (Scroll X, 0xFF43), SCY (Scroll Y, 0xFF42), WX (Window X, 0xFF4B) und WY (Window Y, 0xFF4A) kann der anzuzeigende Bereich gewählt werden. Durch Änderung des sichtbaren Bereichs während des Bildaufbaus können Welleneffekte auf dem Display erzeugt werden.

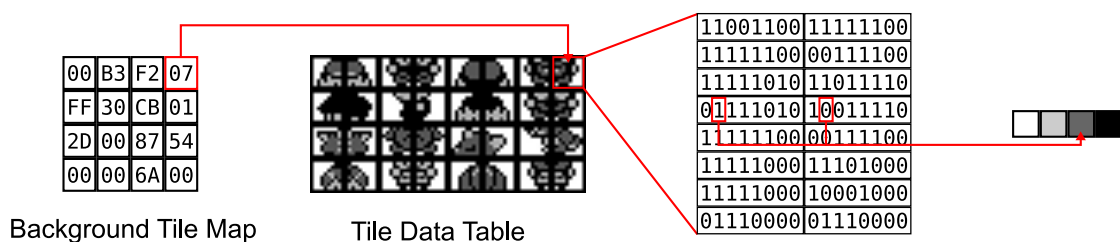


Abbildung 3: Bestimmung des Helligkeitswerts eines Hintergrundpixels

Abbildung 3 zeigt exemplarisch, wie die Farbe eines Hintergrundpixels zustande kommt: zunächst werden für die aktuell gezeichnete Bildzeile die Kachelindizes aus der Background Tile Map bestimmt; diese liegt wählbar entweder ab 0x9800 oder 0x9C00. Über diesen Index wird die Tile Data Table ab 0x8000 oder 0x8800 indiziert. Der Helligkeitswert des x -ten Pixels der y -ten Kachelzeile kann dann aus dem x -ten Bit des $2 \cdot y$ -ten und $2 \cdot y + 1$ -ten Bytes aufgebaut werden. Der Aufbau für ein Vordergrundpixel erfolgt analog.

Bei der Anzeige von Sprites kommt statt einer Tilemap, der OAM-Speicher zum Einsatz: Er enthält für jeden der 40 Sprites eine 4 Byte Struktur, die neben der Bildschirmposition den Kachelindex und einige Flags enthält. Über diese Flags kann der Sprite gespiegelt, hinter dem Hintergrund oder mit einer anderen Graustufenpalette angezeigt werden.

2.6 Audio

Der Gameboy verfügt über vier Kanäle zur Klangerzeugung, die beliebig in die Stereo-Ausgabekanäle gemischt werden können.

Kanal 1 und 2 erzeugen Rechteckschwingungen mit wählbarer Frequenz, Duty-Cycle und linearer Hüllkurve (Abbildung 4). Kanal 1 kann zusätzlich noch eine Frequenzänderung („Sweep“) eines Tons erzeugen.

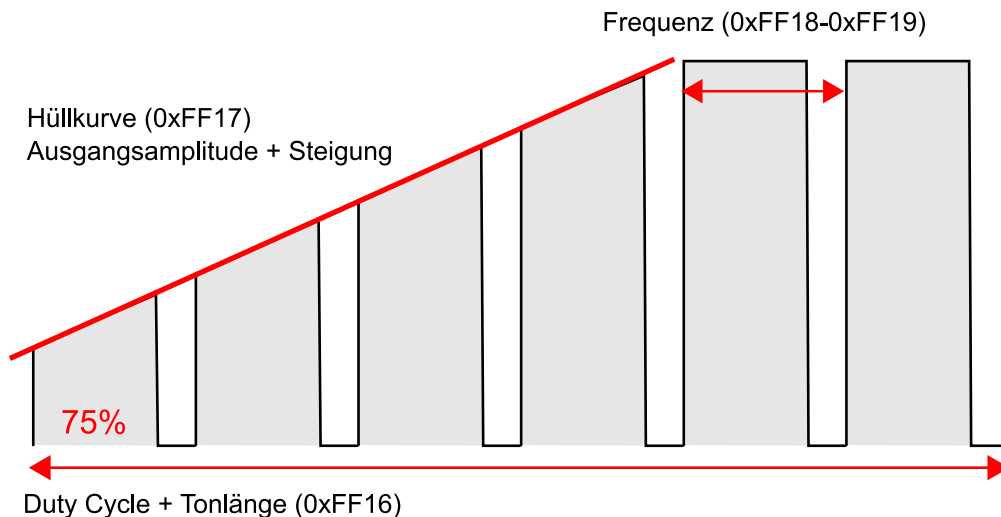


Abbildung 4: Parameter für die Tonerzeugung auf Kanal 2

Kanal 3 kann Wave-Samples abspielen. Der Wave-RAM aus dem diese gelesen werden ist auf 32 Samples à 4 Bit beschränkt.

Kanal 4 erzeugt weißes Rauschen. Dabei kann die Rauschcharakteristik beeinflusst werden um „weicherer“ oder „härterer“ Rauschen zu erzeugen.

3 Aufbau des jit-Übersetzers

Für die Emulation der Gameboy Hardware auf herkömmlichen PCs (x86-64 Architektur) wurde ein just-in-time-kompilierender Emulationskern implementiert. Anstatt einzelne Instruktionen in einer Schleife zu dekodieren und interpretieren, wie bei einem interpretierenden Emulator, wird versucht ganze Blöcke, die gewöhnlich mit einer Sprunginstruktion (JP, JR, CALL, RST, RET, RETI) enden, zusammenzufassen. Mittels des Laufzeitassemblers „dynasm“ des LuaJit Projekts [2][11] wird zum ersten Zeitpunkt, zu dem eine Speicheradresse angesprungen wird, dem Block entsprechende x86-Instruktionen erzeugt und ausgeführt.

Ein Ziel bei der Entwicklung war, möglichst die Statusflags (Carry, Halfcarry/Adjust und Zero) der Hostarchitektur für die emulierte Umgebung zu nutzen, anstatt diese zu emulieren. Dies ist in den meisten Fällen problemlos möglich, da sich die Z80-artige Gameboy CPU und die Intel 8080-Architektur, die größtenteils auch von modernen Prozessoren unterstützt wird sehr ähnlich sind. Da das Subtract-Flag des Gameboy keine direkte Entsprechung in der x86-64 Architektur besitzt muss es als einziges der Statusflags emuliert werden.

Sprünge werden nicht direkt ausgeführt, sondern stattdessen das Sprungziel gespeichert und die erzeugte Funktion mit RET verlassen. Dadurch kann die Laufzeitumgebung gegebenenfalls zunächst den Block am Sprungziel übersetzen und andere parallele Aufgaben ausführen. Diese umfassen die Interrupt, Grafik, Eingabe und DMA Emulation¹.

Während der Übersetzung eines Programmblocks wird für jedes mögliche Ende über das der Block verlassen werden kann², die Anzahl der bis zu diesem Punkt benötigten Gameboy-Taktzyklen berechnet und diese Summe bei der Ausführung auf einen Instruktionszähler addiert. Mittels dieses Zählers können auf dem Gameboy zu bestimmten Zeitpunkten auftretende Ereignisse wie Timer- oder VBLANK-Interrupts trotz der höheren Geschwindigkeit der Hostplattform zeitlich genau ausgeführt werden. Da eventuell Routinen in den emulierten Programmen existieren, die auf eine feste Anzahl ausgeführter Instruktionen in einem bestimmten Zeitraum angewiesen sind, können nicht ohne Kompatibilitätsprobleme Timer des Hostsystems benutzt werden. Aufgrund der blockweisen Ausführung besteht jedoch auch bei dem hier vorgestellten Emulator das Problem, dass Interrupts oder Timer erst um einige Takte verspätet – nach dem nächsten Sprung – ausgeführt beziehungsweise aktualisiert werden.

Während der Ausführung übersetzter Programmblöcke wird der Registersatz des Gameboy direkt auf Register der x86-64 Architektur abgebildet. Am Ende eines Blocks muss dann der gesamte Gameboy-Registersatz, Prozessorflags und die Anzahl emulierter Taktzyklen gesichert werden (struct gb_state). Tabelle 5 stellt die Registernutzung während der Ausführung übersetzter Blöcke dar. Die für die 16 Bit Instruktionen des Gameboy nötigen kombinierten Register AF, BC, DE und HL werden bei Bedarf zunächst in einem temporären Register zusammengesetzt und nach der Instruktion wieder zurückgeschrieben.

¹Die Soundemulation läuft komplett asynchron in einem eigenen Thread und benötigt daher keine Unterbrechung des Programmablaufs.

²Durch bedingte Sprünge können dies beliebig viele sein.

Gameboy	Host (x86-64)	Bemerkung
A	r0 (RAX)	Akkumulator
F	-	dynamisch aus FLAGS-Register erzeugt
B	r1 (RCX)	
C	r2 (RDX)	
D	r3 (RBX)	
E	r13	
H	r5 (RBP)	
L	r6 (RSI)	
SP	r7 (RDI)	
PC	-	nicht nötig
-	r8	Basisadresse des Gameboy-Adressraums
-	r9	Adresse des gb_state struct
-	r10	1. temporäres Register
-	r11	2. temporäres Register
-	r12	3. temporäres Register
-	r4 (RSP)	Host Stack Pointer

Abbildung 5: Registermapping zwischen Gameboy und Host

Ein zweites wichtiges Ziel bei der Implementierung war die Unterstützung direkter lesender Speicherzugriffe: um eine Adresse des Gameboy-Adressraums zu lesen, soll nur eine zusätzliche Addition eines Basiszeigers notwendig sein. Für schreibende Speicherzugriffe ist dies nicht möglich, da Schreibzugriffe auf Adressen im ROM zu Bankwechseln durch den MBC führen und manche IO-Register bei schreibenden Zugriffen bestimmte Aktionen wie DMA-Transfers oder das Lesen der Joypad-Buttons auslösen. Schreibende Zugriffe werden daher durch einen Funktionsaufruf ersetzt, der gegebenenfalls nötige Seiteneffekte emuliert.

Direkte lesende Zugriffe haben einige wichtige Implikationen:

- Kaum Leseoverhead: Im Vergleich zum Gameboy entsteht bei der Emulation kaum Overhead beim Lesen. Da lesende Speicherzugriffe oft zu den häufigsten Instruktionen gehören, bedeutet dies eine deutliche Effizienzsteigerung.
- Der emulierte Gameboy-Adressraum muss konsekutiv sein: der Wechsel von ROM- oder RAM-Bänken erfordert einen großen zusätzlichen Aufwand, da zunächst durch `munmap` und `mmap` die entsprechende Bank in den Adressraum eingeblendet werden muss.
- Statusregister müssen immer aktualisiert werden: der Programmablauf muss häufig unterbrochen werden, um spezielle Statusregister, wie den Timer TIMA (0xFF05), oder die aktuell gezeichnete Bildzeile LY (0xFF44) zu aktualisieren. Geschieht dies nicht, terminieren Warteschleifen eventuell nicht mehr.

3.1 Beispielhafte Übersetzung eines Blocks

Die einzelnen Schritte zur Übersetzung und Ausführung eines Programmblocks sollen durch ein Beispiel verdeutlicht werden: Das folgende Listing zeigt einen Block aus dem Spiel „Super Mario World“.

```
3E 02      LD A, 2
EA 00 20    LD (0x2000), A
E0 FD      LDH (0xFD), A
FA 1D DA    LD A, (0xDA1D)
FE 03      CP A, 3
20 0B      JR NZ, 0x0B
3E FF      LD A, 0xFF
EA 1D DA    LD (0xDA1D), A
CD E8 09    CALL 0x9E8
```

Im ersten Schritt werden Instruktionen bis zum Ende des Blocks gelesen. Jeder unbedingte Sprung (JP, CALL, RST, RET, RETI), sowie EI (Enable Interrupts) beenden einen Block. Die Instruktionen werden in einer verketteten Liste gespeichert und nach ihrem Typ gruppiert. Auf diese Instruktionsliste werden verschiedene Regeln zur Optimierung angewendet, sowie Instruktionen zur Speicherung und Wiederherstellung des Statusregisters eingefügt. Danach wird entsprechender x86-64 Assembler generiert – das Beispiel wird zu folgendem Code übersetzt (ohne Optimierung):

```
prologue
mov A, 2
write_byte 0x2000, A
write_byte 0xfffd, A
mov A, [aMem + 0xda1d]
cmp A, 3
save_cc
restore_cc
jz >1
add qword state->inst_count, 17
return 0x239
1: mov A, 0xff
write_byte 0xda1d, A
dec SP
dec SP
and SP, 0xffff
mov word [aMem + SP], 0x235
add qword state->inst_count, 28
```

```
mov byte state->return_reason, REASON_CALL
return 0x9e8
```

Zur Vereinfachung werden einige Makros verwendet:

prologue sichert alle nötigen Register und stellt die Gameboy-Registerinhalte wieder her.

return sichert alle Registerhalte in der `gb_state` struct, stellt die ursprünglichen Registerinhalte wieder her, schreibt das Argument in das Ergebnisregister und verlässt die Funktion mit `RET`.

write_byte ruft die Funktion `gb_memory_write`.

save_cc sichert das Statusregister auf den Stack.

restore_cc stellt das Statusregister vom Stack wieder her.

aMem bezeichnet das Register `r8`, das die Basisadresse des Gameboy-Adressraums enthält, *state* das Register `r9`, das die Adresse des `gb_state` enthält. *state->inst_count* zählt ausgeführten Gameboy Taktzyklen, *state->return_reason* gibt an, welche Instruktion den Block beendet, um den Backtrace im Debugger zu aktualisieren.

Im nächsten Schritt wird *dynasm* genutzt um den Code zu assemblieren und in einen vorallokierten Speicherbereich zu schreiben. Nachdem dieser mittels `mprotect` ausführbar gemacht wurde kann die Funktion ausgeführt werden. Um eine erneute Ausführung zu beschleunigen, wird der Funktionszeiger über die Startadresse indexiert gespeichert. Die durch die Funktion zurückgelieferte Speicheradresse ist die Startadresse des nächsten auszuführenden Blocks³.

Wird eine Speicheradresse innerhalb des RAM angesprungen, muss davon ausgegangen werden, dass sich die Instruktionfolge bei der nächsten Ausführung geändert hat. Blöcke innerhalb des RAM werden daher nach der Ausführung wieder verworfen. Eine Ausnahme bilden Blöcke innerhalb der Adressen `0xFF80` bis `0xFFFFE`: während DMA-Transfers muss kurzzeitig in diesen Bereich gesprungen werden. Die Routine, die auf das Ende des Transfers wartet ändert sich dabei gewöhnlich während der Ausführung des Programms nicht. Es lohnt sich deshalb die Blöcke bis zu einem Schreibzugriff in diesen Speicherbereich zwischenzuspeichern.

3.2 Optimierung

Nach dem Lesen eines Instruktionsblocks werden einige Regeln zur Optimierung angewendet. Schleifen unterbrechen durch eine große Anzahl an Sprüngen den Programmablauf sehr häufig und verursachen damit einen sehr hohen Overhead zur Sicherung und Wiederherstellung der Registerinhalte sowie zur Überprüfung auf Interrupts. Aus diesem Grund

³Tritt ein Interrupt auf, wird sie stattdessen auf den Gameboy-Stack gelegt und die Startadresse des Interrupthandlers angesprungen

dienen die meisten implementierten Optimierungen der Erkennung und Behandlung von Schleifen.

Schleifen können am einfachsten an einem Sprung auf die Startadresse des aktuellen Blocks erkannt werden, da spätestens nach der ersten Iteration und dem Rücksprung auf den Schleifenbeginn ein neuer Block ab dieser Startadresse übersetzt wird.

Falls im Schleifenrumpf kein lesender oder schreibender Speicherzugriff stattfindet und alle Interrupts mit RET oder RETI zurückkehren, kann die gesamte Schleife atomar ausgeführt werden. Es ist in diesem Fall unerheblich ob ein Interrupt vor, während oder nach der Schleife ausgeführt wird. Gerade einfache Warteschleifen, die eine feste Zahl von Taktzyklen warten, können so beschleunigt werden: der Rücksprung an den Blockanfang wird direkt ausgeführt, ohne zwischenzeitlich die Kontrolle an die Laufzeitumgebung abzugeben und auf Interrupts zu prüfen.

Auch schreibende Speicherzugriffe können meist gefahrlos durchgeführt werden. Ein Interrupthandler kann jedoch in diesem Fall eventuell durch die Schleife beeinflusst werden und sich durch die zusätzlich ausgeführten Schleifeniterationen fehlerhaft verhalten. Lesende Speicherzugriffe bergen dagegen ein erhebliches Risiko: eine Warteschleife, die auf einen Interrupt oder Timer wartet, terminiert eventuell nicht mehr. Da der lesende Zugriff auf Timer- und Statusregister meist mit speziellen Instruktionen erfolgt (LDH A, (a8) oder LDH A, (C)), werden in höheren Optimierungsstufen andere lesende Instruktionen in Schleifen erlaubt.

Die folgende Schleife führt ein memset auf einem Speicherbereich der Länge BC mit Endadresse HL durch und kann mit obigen Optimierungen unterbrechungsfrei ausgeführt werden:

32	LD (HL-), A	; Byte setzen und HL dekrementieren
05	DEC B	
20 FC	JR NZ, 0xFC	; an Anfang springen
0D	DEC C	
20 F9	JR NZ, 0xF9	; an Anfang springen

Andere Optimierungen suchen mittels pattern matching nach bekannten und häufigen Instruktionsfolgen, die sich vereinfachen lassen. Das folgende häufig verwendete Muster wartet bis eine bestimmte Displayzeile gezeichnet wird⁴:

F0 44	LDH A, (0x44)	; aktuelle Displayzeile lesen
FE ??	CP A, ??	; mit festem Wert vergleichen
20 FA	JR NZ, 0xFA	; an Anfang springen

In der Emulation kann stattdessen eine modifizierte HALT-Instruktion eingefügt werden, die statt auf einen Interrupt auf das Zeichnen der entsprechenden Displayzeile wartet.

⁴Es kann auch auf einen VBLANK gewartet werden, wenn die Zeile > 144 ist.

4 Grafik- / Audioausgabe

Da die Grafikausgabe des Gameboy – wie beschrieben – über spezielle Kontrollregister sowie festgelegte Speicherbereiche für Tilemaps erfolgt, muss der Emulator diese Speicherbereiche interpretieren und das entsprechende Bild pixelweise erzeugen. Es reicht nicht, den Speicher zu Beginn der VBLANK-Periode einmalig zu interpretieren und das Bild auszugeben, da viele Spiele das Displaytiming ausnutzen um Grafikeffekte zu erzeugen. Sollen diese korrekt dargestellt werden, muss die Erzeugung des Bildes im Emulator ebenso Zeilenweise erfolgen.

Nach jedem ausgeführten Instruktionsblock wird im Zuge der Interruptbehandlung auch das LY (ca. alle 450 Taktzyklen) und das STAT-Register (ca. alle 80, 180, 190 Taktzyklen⁵) aktualisiert. Wird das LY-Register inkrementiert, kann die nächste Bildzeile gezeichnet werden. Nach 144 gezeichneten Zeilen, zu Beginn der VBLANK-Periode, kann schließlich das erzeugte Bild an den Render-Thread zur Anzeige weitergegeben werden. Durch einen separaten Render-Thread wird der Haupt-Thread von der langsamen Aktualisierung der Bildtextur und deren Anzeige entlastet und die Laufzeit des Haupt-Threads pro Frame halbiert.

Der Beginn der VBLANK-Periode wird gleichzeitig zur Limitierung der Geschwindigkeit genutzt: Sind seit dem letzten VBLANK weniger als 1/60s vergangen, wird entsprechend lang gewartet, bevor die Ausführung fortgesetzt wird.

Für die Audioausgabe existieren zwei verschiedene Implementierungen: neben der standardmäßig verwendeten Bibliothek GB_Snd_Emu, die alle Soundregister und Funktionen vollständig interpretieren kann und störungsfreie Wave-Samples erzeugt, existiert eine eigene unvollständige Implementierung, die nicht alle Funktionen unterstützt.

SDL erwartet zur Audioausgabe eine Callbackfunktion, die einen Puffer von Wave-Samples fester Länge (ca. 6ms) füllt. Ein separater Audio-Thread spielt diese Wave-Samples ab und ruft bei Bedarf die Callbackfunktion auf um die nächsten Samples zu gewinnen. Mit jedem Aufruf der Callbackfunktion wird der zu diesem Zeitpunkt aktuelle Registersatz interpretiert und der Puffer entsprechend gefüllt. Dies ist hinreichend genau, da der kürzeste durch den Gameboy abspielbare Ton eine Länge von 4ms hat und damit eine ähnliche Auflösung wie der Audio-Thread.

5 Bedienung / Installation

Die vollständigen Quellen des vorgestellten Emulators sind unter [13] verfügbar und lassen sich mit git klonen. Um das Programm zu übersetzen und auszuführen wird ein x86-64 Linux⁶ und folgende Bibliotheken und Programme benötigt:

- gcc/g++ 4.8
- lua 5.2

⁵mit jeder bearbeiteten Zeile durchläuft das STAT-Register drei verschieden lang andauernde Modi.

⁶Das Programm lässt sich eventuell auch unter Apple OSX oder Windows mit cygwin übersetzen, dies wurde aber nie getestet.

- lua-bitop
- libreadline(-dev)
- libsdl2(-dev)
- libglib-2.0(-dev)

Die zusätzlichen Bibliotheken `dynasm` (Laufzeitassembler [11]) und `Gb_Snd_Emu` (Gameboy Sound Emulation [7]) liegen dem Repository in kompatiblen Versionen bei. Zum Übersetzen reicht ein Aufruf von `make`.

`dynasmgb` bietet derzeit keine graphischen Konfigurationsmöglichkeiten. Um ein Spiel zu starten reicht ein Aufruf von `./dynasmgb game.gb` im Hauptverzeichnis. Folgende zusätzliche Kommandozeilenoptionen werden unterstützt:

- d** Aktiviert direkt beim Start die Debuggerkonsole.
- O** Aktiviert die Optimierungen in mehreren Stufen (-O0 bis -O3).
- b** Setzt direkt beim Start einen Breakpoint, bei dessen Erreichen die Debuggerkonsole gestartet wird.
- ?** Gibt eine kurze Übersicht aller Kommandozeilenoptionen aus.

Während das Spielfenster den Fokus besitzt, werden die in Abbildung 6 dargestellten Tastenkombinationen unterstützt⁷. Die Tastenbelegungen lassen sich derzeit noch nicht konfigurieren.

Taste	Funktion
a	Gameboy Knopf „A“
b	Gameboy Knopf „B“
y	Gameboy Knopf „Start“
x	Gameboy Knopf „Select“
Pfeiltasten	Gameboy Richtung
d	Debuggerkonsole aktivieren
F2	Speicherabzug schreiben
F3	Speicherabzug lesen
ESC	<code>dynasmgb</code> beenden

Abbildung 6: `dynasmgb` Tastaturbelegung

Beim Spielstart werden im Terminal einige Informationen aus dem ROM-Header ausgegeben (Abbildung 7).

Beim Verlassen des Spiels werden einige Statistiken zur Emulation ausgegeben: Neben der Gesamtzahl übersetzter (und gespeicherter) Blöcke wird die Startadresse des am

⁷Während der Debugger gestartet ist werden Tastaturevents entgegengenommen, aber erst verarbeitet, wenn der Debugger wieder verlassen wird.

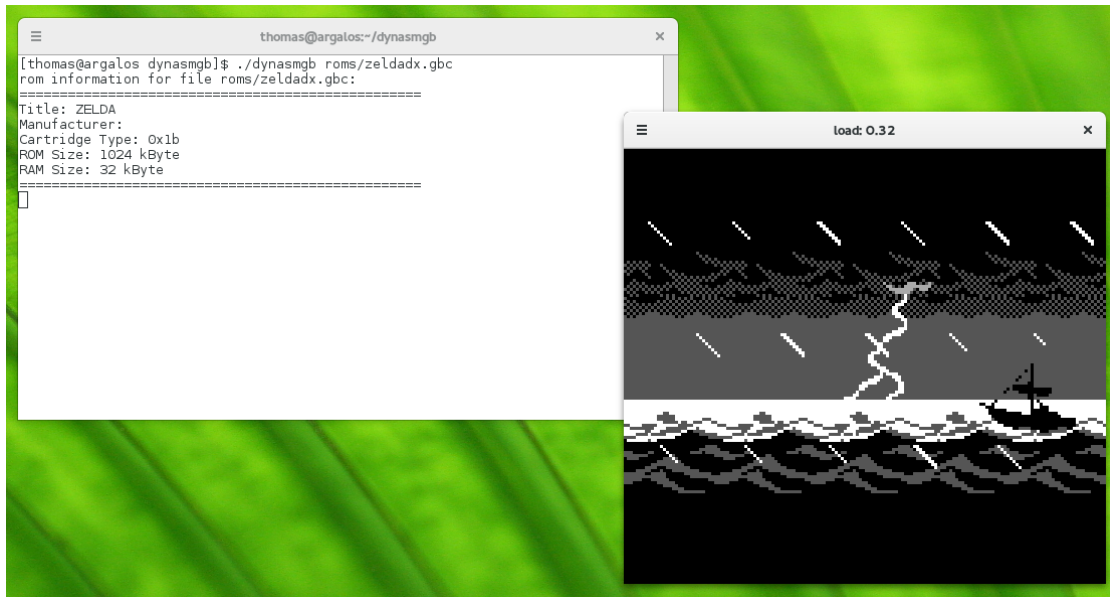


Abbildung 7: ROM-Header Informationen für das Spiel „The Legend of Zelda – Link’s Awakening DX“.

häufigsten ausgeführten Blocks ausgegeben. Dies hilft besonders kritische Schleifen zu erkennen und entsprechende Optimierungsregeln zu schreiben. Zuletzt wird noch die Gesamtzahl ausgeführter Blöcke, die durchschnittliche Anzahl ausgeführter Blöcke pro Frame und die Anzahl angezeigter Frames angezeigt.

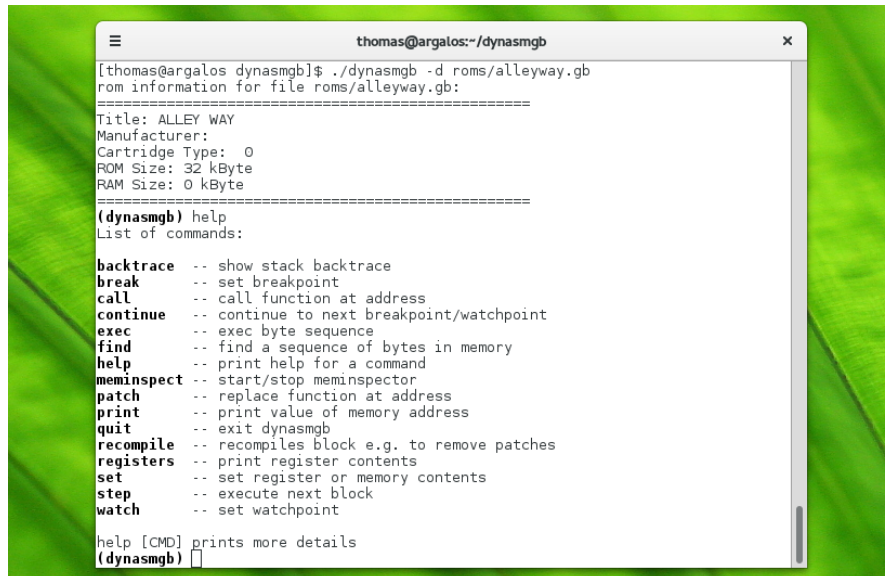
Falls das Spiel eine oder mehrere RAM-Bänke innerhalb der Cartridge besitzt, werden diese beim Beenden zusätzlich im Verzeichnis des Spiels als .sav-Datei gespeichert. Beim Spielstart wird der Inhalt einer solchen Datei – falls vorhanden – wieder in den Speicher geladen. Dies simuliert die batteriegepufferten RAM-Bänke, die in vielen Spielen zum Speichern von Spielständen verwendet werden.

6 Debugger

Um Emulationsfehler effektiv zu verfolgen und lokalisieren wurde eine Debuggerkonsole für das aktuell ausgeführte Spiel implementiert. Sie lässt sich durch Drücken der Taste 'd' zu einem beliebigen Zeitpunkt im Spiel, über die Startoption '-d' oder bei Erreichen eines Breakpoints aktivieren.

Mögliche Befehle sind:

backtrace Gibt den aktuellen Aufrufstack aus. Dabei wird für jeden Stackframe die Startadresse des ersten ausgeführten Blocks ausgegeben. Handelt es sich bei einem Aufruf um einen Interrupt, wird dieser rot dargestellt. Restarts werden dagegen grün dargestellt. Da sich die für den Backtrace nötigen Informationen, aufgrund eines möglicherweise von Spiel zu Spiel unterschiedlichen Stacklayouts, nicht oder



```
thomas@argalos:~/dynasmgb
[thomas@argalos dynasmgb]$ ./dynasmgb -d roms/alleyway.gb
rom information for file roms/alleyway.gb:
=====
Title: ALLEY WAY
Manufacturer:
Cartridge Type: 0
ROM Size: 32 kByte
RAM Size: 0 kByte
=====
(dynasmgb) help
List of commands:

backtrace -- show stack backtrace
break     -- set breakpoint
call      -- call function at address
continue  -- continue to next breakpoint/watchpoint
exec      -- exec byte sequence
find      -- find a sequence of bytes in memory
help      -- print help for a command
meminspect -- start/stop meminspector
patch     -- replace function at address
print     -- print value of memory address
quit      -- exit dynasmgb
recompile -- recompiles block e.g. to remove patches
registers -- print register contents
set       -- set register or memory contents
step      -- execute next block
watch     -- set watchpoint

help [CMD] prints more details
(dynasmgb) 
```

Abbildung 8: Debuggerkonsole.

nur schwer aus dem Speicher gewinnen lassen, wird parallel zur Ausführung des Spiels bei jeder CALL oder RET Instruktion und Interrupts der Debuggerstack aktualisiert. Einige Spiele manipulieren den Aufrufstack zusätzlich durch andere Instruktionen, oder kehren nicht mittels RET aus Funktionen oder Interrupts zurück – in diesen Fällen sind die angezeigten Informationen fehlerhaft.

break Setzt einen Breakpoint für eine Startadresse eines Blocks. Bei einem Sprung auf die angegebene Adresse wird die Debuggerkonsole gestartet. Die Adresse muss dabei genau die Startadresse eines Blocks sein, Adressen innerhalb eines Blocks werden ignoriert. Derzeit wird nur ein einziger Breakpoint unterstützt und bei jedem ausgeführten Block überprüft, ob der Breakpoint erreicht wurde. Eine alternative Implementierung, die den entsprechenden Block so verändert, dass der Debugger gestartet wird und damit eine beliebige Anzahl Breakpoints ohne zusätzlichen Overhead unterstützt, ist bisher noch nicht umgesetzt, aber denkbar.

call Ruft eine Funktion an einer bestimmten Speicheradresse auf. Im Unterschied zu einem „set PC *address*“ kann mit einer RET-Instruktion wieder zur aktuellen Adresse zurückgekehrt werden. Ein Aufruf von „call 0x1234“ hat den gleichen Effekt wie „exec CD 34 12“, ohne dass der `instruction_count` erhöht wird.

continue Verlässt die Debuggerkonsole und setzt das Spiel fort. Die Tastenkombination Strg+D hat den gleichen Effekt.

exec Übersetzt und führt eine beliebige Bytefolge aus. Die Bytes werden hexadezimal angegeben und können durch Leerzeichen oder Kommata getrennt werden. Die Bytefolge darf maximal 100 Byte lang sein. Falls die Bytefolge nicht mit einer Sprung-

instruktion endet, die den Block beendet, wird der Block stattdessen automatisch mit einer Sprunginstruktion abgeschlossen, die zur aktuellen Adresse zurückkehrt. Die Instruktionsfolge wird wie jeder andere Block, aber ohne Optimierung übersetzt. Die Ausführung beeinflusst insbesondere auch den `instruction_count` und damit Timer und Interruptzeitpunkte.

find Durchsucht den gesamten Adressraum nach der angegebenen Bytefolge. Die Bytes werden hexadezimal angegeben und können durch Leerzeichen oder Kommata getrennt werden. Die Bytefolge darf maximal 20 Byte lang sein. Um die Fundstellen wird ein Kontext von jeweils 10 Byte angezeigt.

help Gibt die Abbildung 8 dargestellte Hilfe aus.

meminspect Öffnet ein zusätzliches „Memory Inspector“ sowie ein „Address Space“-Fenster. Dieses zeigt in der linken Hälfte eine Liveansicht aller schreibenden Schreibzugriffe auf die oberen 32kB des Gameboy Adressraums. In der rechten Hälfte werden unten die Gesamtzahl, sowie die aktuell in den Adressraum eingeblendeten ROM (orange) und RAM-Bänke (türkis) angezeigt. In der rechten oberen Hälfte wird eine Detailansicht der IO-Register zwischen 0xFF00 und 0xFF4F dargestellt (pink). Da nur Schreibzugriffe registriert werden, die den Speicherinhalt verändern und die Ansicht mit jedem VBLANK-Interrupt aktualisiert wird, werden Änderungen an bestimmten IO-Registern (wie zB LY) nicht angezeigt. Das „Memory-Inspector“-Fenster visualisiert die Background Tile Map, sowie die Tile Data Table für beide möglichen Startadressen (0x8000 und 0x8800). Ein weiterer Aufruf von „meminspect“ schließt die Fenster wieder.

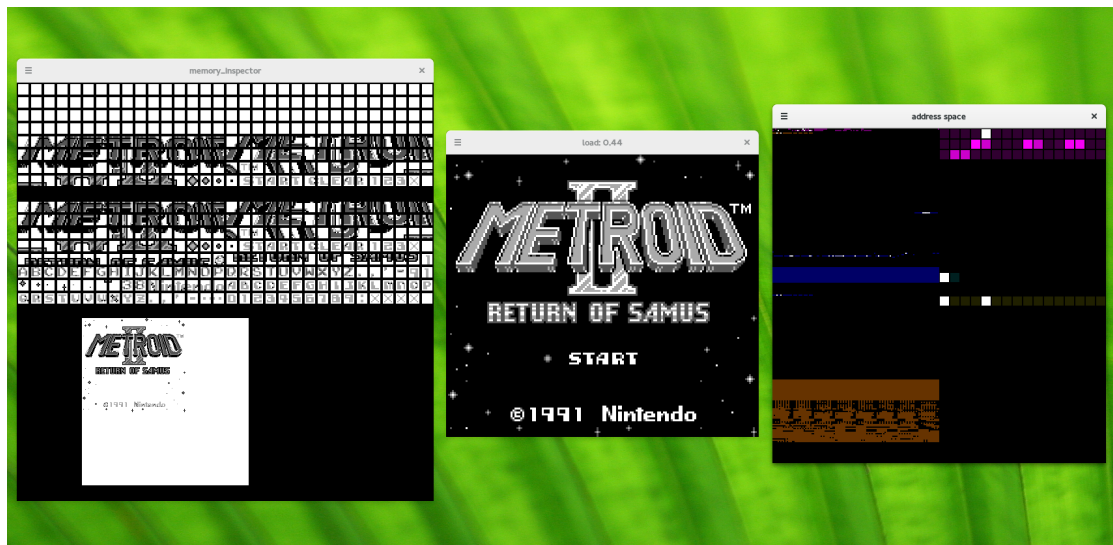


Abbildung 9: Das Spiel „Metroid II – Return of Samus“ mit geöffnetem memory inspector.

patch Ersetzt den Block mit der angegebenen Startadresse durch die angegebene Bytefolge. Es gelten die gleichen Einschränkungen wie für `exec`. Falls der angegebene Block

nicht mit einem Sprung oder einer anderen den Block beendenden Instruktion endet wird stattdessen die aktuelle Adresse angesprungen.

print Gibt den Speicherinhalt an der (hexadezimal) angegebenen Adresse aus. Um den Wert eines Registers auszugeben kann stattdessen „registers“ verwendet werden.

quit Beendet die Emulation. Gleicher Effekt wie ESC, während das Spiel läuft.

recompile Übersetzt den Block mit der angegebenen Startadresse neu. Dabei werden keine Optimierungen angewendet. Eventuell mit „patch“ gemachte Änderungen an diesem Block werden rückgängig gemacht.

registers Gibt die Werte aller Register aus.

set Setzt den Wert eines Registers oder einer Speicheradresse. Um einen Registerwert zu ändern muss als erstes Argument das Register angegeben werden (zB „set PC 0x100“ um das Spiel zu resettet). Ist das erste Argument eine Adresse wird der entsprechende Wert in den Speicher geschrieben (zB „set 0xFFFFE 0x42“). Zum Setzen der Speicheradresse wird – wie in übersetzten Blöcken – die Funktion `write_byte` verwendet, sodass ein Schreibzugriff auf den ROM eventuell keinen Effekt hat oder zu einem Bankwechsel führt.

step Führt den nächsten Block aus. Eine leere Zeile im Debugger hat den gleichen Effekt. Derzeit wird kein Stepping über einzelne Instruktionen unterstützt.

watch Setzt einen Watchpoint für eine Speicheradresse. Ändert sich der Wert der Speicheradresse, pausiert die Emulation am Ende des Blocks und die Debuggerkonsole wird angezeigt. Derzeit kann erst am Ende des Blocks in dem der Zugriff erfolgt gestoppt und maximal ein Watchpoint gesetzt werden, da nach jedem ausgeführten Block überprüft wird, ob sich die überwachte Adresse geändert hat.

7 Kompatibilität

Wie beschrieben implementiert `dynasmgb` einen Großteil der Gameboy-Architektur und Funktionalität. Der `jit`-Übersetzer unterstützt den kompletten Gameboy-Befehlssatz sowie wichtige Interrupts. Derzeit fehlt eine Implementierung serieller Transfers sowie des entsprechenden Interrupts.

Die Grafikemulation unterstützt alle drei Zeichenebenen (Background, Window und Sprites) nahezu vollständig. Die Reihenfolge in der die Sprites gezeichnet werden kann jedoch vom Gameboy abweichen.

Bekannte Hardwarefehler und Limitierungen des Gameboys – unter anderem der Sprite RAM Bug, oder das Limit von 10 Sprites pro Zeile – werden nicht emuliert. Der interne RAM wird im Unterschied zum Gameboy nicht in den Adressbereich zwischen 0xE000 und 0xFDFF gespiegelt.

Die Kompatibilität wird zusätzlich durch das möglicherweise ungenaue Timing der Interrupts und Timer-Register, sowie durch Unterschiede beim Setzen des Statusregisters eingeschränkt.

Der Emulator wurde anhand verschiedener Testroms, Beispielroms sowie kommerzieller Spiele getestet.

7.1 Blargg's GB Instruction Test

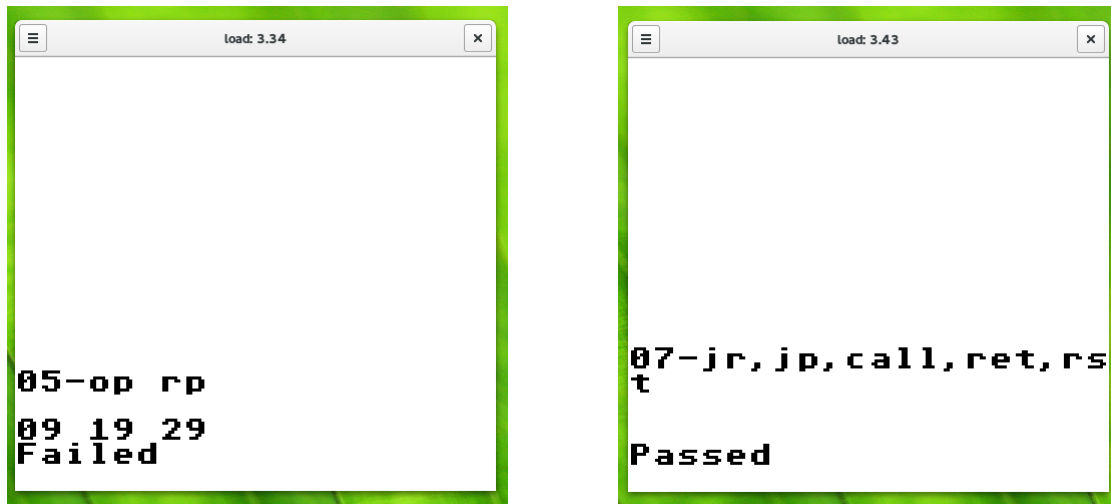


Abbildung 10: Blargg's GB Instruction Test (Einzelne Testfälle).

Blargg's GB Instruction Test [6] umfasst 11 Testfälle, die sämtliche Gameboyinstruktionen ausführen und die Registerinhalte nach der Ausführung mit einer CRC-Checksumme vergleichen. Dabei werden sämtliche Kombinationen von Eingabewerten und Statusflags überprüft.

Testfall	Status	Erfolgreich
01-special	failed	4/5
02-interrupts	failed	0/4
03-op sp,hl	failed	3/8
04-op r,imm	failed	15/16
05-op rp	failed	6/9
06-ld r,r	passed	64/64
07-jr,jp,call,ret,rst	passed	29/29
08-misc instrs	passed	19/19
09-op r,r	failed	73/134
10-bit ops	failed	112/168
11-op a,(hl)	failed	32/51

Abbildung 11: Blargg's GB Instruction Test – Ergebnisse.

7.2 GBDK Beispiel ROMs / PD ROMs

Das Gameboy Developer's Kit (GBDK, [8]) ist eine inoffizielle Toolchain zur Entwicklung von Gameboy-Spielen in C. Es enthält einige Beispielprogramme, die zum Testen des Emulators verwendet werden können, da sie meist nur eine bestimmte Funktionalität des Gameboy ausgiebig nutzen.

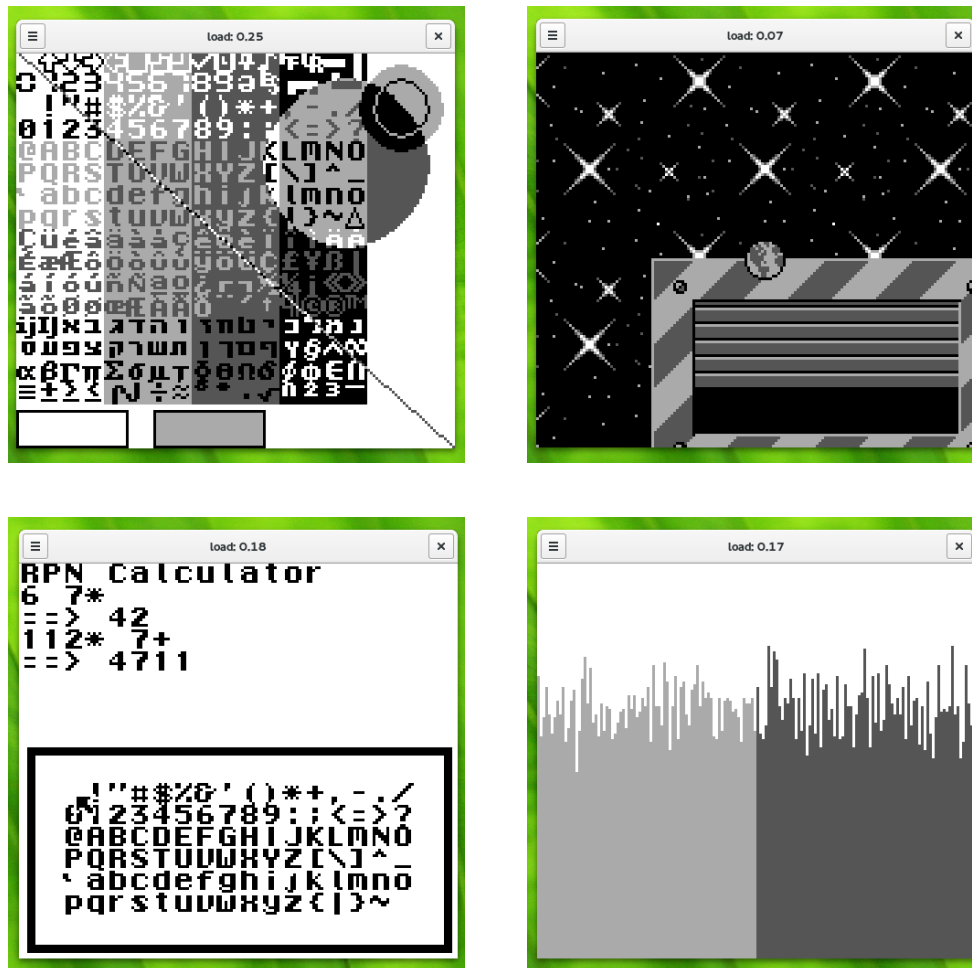
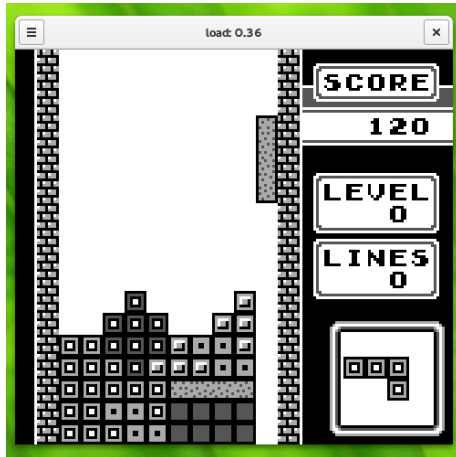


Abbildung 12: GBDK Test ROMs.

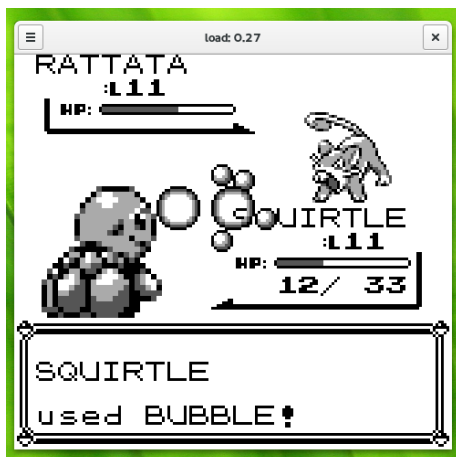
7.3 Kommerzielle Gameboy Spiele



Tetris Tetris wurde zusammen mit dem Gameboy Classic verkauft und ist daher das meistverkaufte Spiel für den Gameboy. Da das Spiel eine Größe von nur 32kB hat, kommt es ohne MBC aus. Es wird keine HALT-Instruktion verwendet, sodass es ein großes Optimierungspotential besitzt. Für die korrekte Emulation ist nur ein geringer Teil aller Features des Gameboy notwendig; selbst ohne Timeremulation ist das Spiel begrenzt lauffähig; der Zufallszahlengenerator um den nächsten Block auszuwählen funktioniert dann jedoch nicht.



Super Mario World Da Super Mario World eine ROM-Größe von 64kB besitzt wird ein MBC1 zur Verwaltung der vier Speicherbänke verwendet. Das Spiel nutzt fast alle graphischen Effekte bei der Spritedarstellung und benötigt ein funktionierendes Scrollen des Hintergrunds. Eine korrekte Timer-Emulation sowie eine funktionierende DAA-Instruktion sind notwendig um die Zeitanzeige korrekt darzustellen und das Level wechseln zu können. Es wird eine größere Palette an Instruktionen verwendet als bei Tetris.

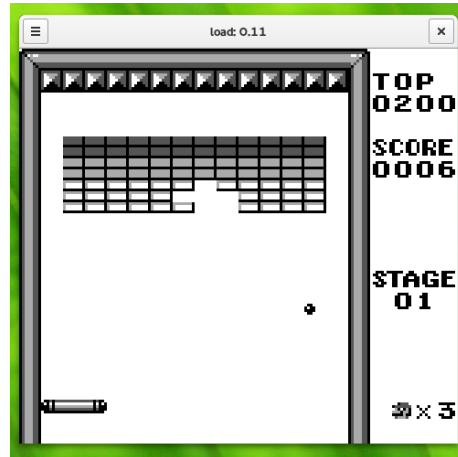


Pokémon Blaue Edition Pokémon nutzt nahezu alle Funktionen des Gameboy. Zur Verwaltung des 1MB (64 Bänke) großen ROMs und der RAM-Bänke wird ein MBC5 genutzt. Um die Speicherfunktion umsetzen zu können, müssen die RAM-Bänke gespeichert werden. Da das Spiel eine Vielzahl graphischer Effekte (Wellen, invertierte Farben, etc.) für Animationen nutzt, muss sowohl ein genaues Timing, als auch eine Unterstützung sämtlicher graphischer Features existieren.

Bei der Entwicklung wurde darauf geachtet, eine korrekte Emulation der oben genannten Spiele sicherzustellen. Es sind keine Fehler in diesen Spielen mehr bekannt, außer minimalen graphischen Fehlern in Pokémon (In manchen Scrollanimationen ist eine einzelne Zeile falsch platziert), die wahrscheinlich durch ungenauigkeiten im Timing verursacht werden. Fast alle anderen getesteten kommerziellen Spiele funktionieren zu großen Teilen, weisen aber kleinere, das Spiel beeinflussende Fehler auf.



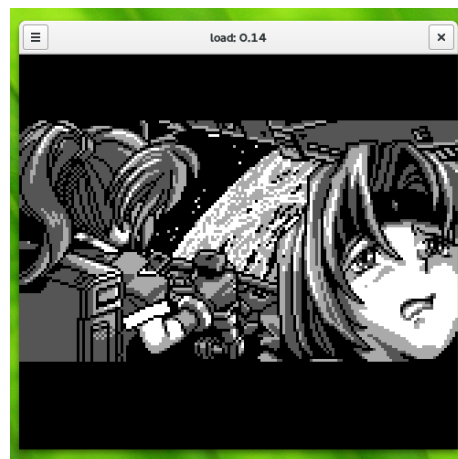
Zelda – Link's Awakening DX



Alleyway



Metroid II – Return of Samus



Star Ocean – Blue Sphere

Abbildung 13: Beispiele für andere, größtenteils funktionierende Spiele.

8 Fazit

Im Rahmen des Projekts wurde ein funktionierender Gameboy-Emulator implementiert. Die Kompatibilität ist derzeit stark eingeschränkt; um sie soweit zu verbessern, dass eine vollständige Funktion aller Spiele erreicht wird, wäre ein erheblicher Aufwand nötig. Zusätzlich müssten einige Optimierungen, wie die verspätete Ausführung von Interrupts oder die Nutzung des Host-Statusregisters fallen gelassen werden.

Das Potential der dynamischen Übersetzung kommt leider aufgrund der sehr hardware-nahen Programmierung der Gameboy-Plattform nur selten zum Tragen, da beispielsweise die Interruptbehandlung einigen Overhead erzeugt. Eine Architektur, die einen Hardware-zugriff nur über eine definierte Betriebssystemschnittstelle zulässt, kann deutlich stärker profitieren.

Das Projekt zeigt dennoch, dass eine dynamische Übersetzung in Verbindung mit dabei durchgeführten Optimierungen auch beim Emulieren der Gameboy-Plattform Potential hat und trotz der dabei kaum erreichbaren hundertprozentigen Genauigkeit, diese ausreicht um viele oder eventuell fast alle Spiele auszuführen.

Die Implementierung hat mir immer viel Spaß gemacht und ich habe einen tiefen Einblick in die Architektur des aus heutiger Sicht archaisch wirkenden Gameboy gewonnen. Ich hoffe, die entstandene Software, sowie diese Dokumentation, wird in Zukunft von Anderen genutzt, um *dynasmgb* entweder zu erweitern, oder ein anderes Projekt auf Basis der gewonnenen Erkenntnisse umzusetzen.

Literatur

- [1] Nintendo of America Inc. *GAME BOY Programming Manual*. Version 1.00.
- [2] Peter Cawley. *The Unofficial DynASM Documentation*. URL: <http://corsix.github.io/dynasm-doc/index.html>.
- [3] cottonvibes. *Introduction to Dynamic Recompilation*. 2010. URL: <http://forums.pcsx2.net/Thread-blog-Introduction-to-Dynamic-Recompilation>.
- [4] DP. *Game Boy CPU Manual*. Version 1.01. URL: marc.rawer.de/Gameboy/Docs/GBCPUman.pdf.
- [5] *Gameboy OPCODES*. URL: http://pastraiser.com/cpu/gameboy/gameboy_opcodes.html.
- [6] Shay Green. *Blargg's test ROMs*. URL: <http://blargg.8bitalley.com/parodius/gb-tests/>.
- [7] Shay Green. *GB_Snd_Emu library*. URL: http://blargg.8bitalley.com/libs/audio.html#Gb_Snd_Emu.
- [8] Michael Hope. *GameBoy Developer's Kit*. URL: <http://gbdk.sourceforge.net/>.
- [9] Andrew Kelley. *Statically Recompiling NES Games into Native Executables with LLVM and Go*. 2013. URL: <http://andrewkelley.me/post/jamulator.html>.
- [10] Martin Korth. *Pan Docs*. 2001. URL: <http://bgb.bircd.org/pandocs.htm>.
- [11] Mike Pall. *DynASM*. URL: <http://luajit.org/dynasm.html>.
- [12] Realboy Project. *Realboy – Emulating the Core*. 2013. URL: <https://realboyemulator.wordpress.com/2013/01/08/emulating-the-core-1/>.
- [13] Thomas Witte. *dynasmgb Repository*. URL: <https://mrm-11.e-technik.uni-ulm.de/thomas.witte/dynasmgb>.

Gameboy CPU (LR35902) instruction set

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	NOP 1 4 - - - -	LD BC,d16 3 12 - - - -	LD (BC),A 1 8 - - - -	INC BC 1 8 - - - -	INC B 1 4 Z 0 H -	DEC B 1 4 Z 1 H -	LD B,d8 2 8 - - - -	RLCA 1 4 0 0 0 C	LD (a16),SP 3 20 - - - -	ADD HL,BC 1 8 - 0 H C	LD A,(BC) 1 8 - - - -	DEC BC 1 8 - - - -	INC C 1 4 Z 0 H -	DEC C 1 4 Z 1 H -	LD C,d8 2 8 - - - -	RRCA 1 4 0 0 0 C
1x	STOP 0 2 4 - - - -	LD DE,d16 3 12 - - - -	LD (DE),A 1 8 - - - -	INC DE 1 8 - - - -	INC D 1 4 Z 0 H -	DEC D 1 4 Z 1 H -	LD D,d8 2 8 - - - -	RLA 1 4 0 0 0 C	JR r8 2 12 - - - -	ADD HL,DE 1 8 - 0 H C	LD A,(DE) 1 8 - - - -	DEC DE 1 8 - - - -	INC E 1 4 Z 0 H -	DEC E 1 4 Z 1 H -	LD E,d8 2 8 - - - -	RRA 1 4 0 0 0 C
2x	JR NZ,r8 2 12/8 - - - -	LD HL,d16 3 12 - - - -	LD (HL+),A 1 8 - - - -	INC HL 1 8 - - - -	INC H 1 4 Z 0 H -	DEC H 1 4 Z 1 H -	LD H,d8 2 8 - - - -	DAA 1 4 Z - 0 C	JR Z,r8 2 12/8 - - - -	ADD HL,HL 1 8 - 0 H C	LD A,(HL+) 1 8 - - - -	DEC HL 1 8 - - - -	INC L 1 4 Z 0 H -	DEC L 1 4 Z 1 H -	LD L,d8 2 8 - - - -	CPL 1 4 - 1 1 -
3x	JR NC,r8 2 12/8 - - - -	LD SP,d16 3 12 - - - -	LD (HL-),A 1 8 - - - -	INC SP 1 8 - - - -	INC (HL) 1 12 Z 0 H -	DEC (HL) 1 12 Z 1 H -	LD (HL),d8 2 12 - - - -	SCF 1 4 - 0 0 1	JR C,r8 2 12/8 - - - -	ADD HL,SP 1 8 - 0 H C	LD A,(HL-) 1 8 - - - -	DEC SP 1 8 - - - -	INC A 1 4 Z 0 H -	DEC A 1 4 Z 1 H -	LD A,d8 2 8 - - - -	CCF 1 4 - 0 0 C
4x	LD B,B 1 4 - - - -	LD B,C 1 4 - - - -	LD B,D 1 4 - - - -	LD B,E 1 4 - - - -	LD B,H 1 4 - - - -	LD B,L 1 4 - - - -	LD B,(HL) 1 8 - - - -	LD B,A 1 4 - - - -	LD C,B 1 4 - - - -	LD C,C 1 4 - - - -	LD C,D 1 4 - - - -	LD C,E 1 4 - - - -	LD C,H 1 4 - - - -	LD C,L 1 4 - - - -	LD C,(HL) 1 8 - - - -	LD C,A 1 4 - - - -
5x	LD D,B 1 4 - - - -	LD D,C 1 4 - - - -	LD D,D 1 4 - - - -	LD D,E 1 4 - - - -	LD D,H 1 4 - - - -	LD D,L 1 4 - - - -	LD D,(HL) 1 8 - - - -	LD D,A 1 4 - - - -	LD E,B 1 4 - - - -	LD E,C 1 4 - - - -	LD E,D 1 4 - - - -	LD E,E 1 4 - - - -	LD E,H 1 4 - - - -	LD E,L 1 4 - - - -	LD E,(HL) 1 8 - - - -	LD E,A 1 4 - - - -
6x	LD H,B 1 4 - - - -	LD H,C 1 4 - - - -	LD H,D 1 4 - - - -	LD H,E 1 4 - - - -	LD H,H 1 4 - - - -	LD H,L 1 4 - - - -	LD H,(HL) 1 8 - - - -	LD H,A 1 4 - - - -	LD L,B 1 4 - - - -	LD L,C 1 4 - - - -	LD L,D 1 4 - - - -	LD L,E 1 4 - - - -	LD L,H 1 4 - - - -	LD L,L 1 4 - - - -	LD L,(HL) 1 8 - - - -	LD L,A 1 4 - - - -
7x	LD (HL),B 1 8 - - - -	LD (HL),C 1 8 - - - -	LD (HL),D 1 8 - - - -	LD (HL),E 1 8 - - - -	LD (HL),H 1 8 - - - -	LD (HL),L 1 8 - - - -	HALT 1 4 - - - -	LD (HL),A 1 8 - - - -	LD A,B 1 4 - - - -	LD A,C 1 4 - - - -	LD A,D 1 4 - - - -	LD A,E 1 4 - - - -	LD A,H 1 4 - - - -	LD A,L 1 4 - - - -	LD A,(HL) 1 8 - - - -	LD A,A 1 4 - - - -
8x	ADD A,B 1 4 Z 0 H C	ADD A,C 1 4 Z 0 H C	ADD A,D 1 4 Z 0 H C	ADD A,E 1 4 Z 0 H C	ADD A,H 1 4 Z 0 H C	ADD A,L 1 4 Z 0 H C	ADD A,(HL) 1 8 Z 0 H C	ADD A,A 1 4 Z 0 H C	ADC A,B 1 4 Z 0 H C	ADC A,C 1 4 Z 0 H C	ADC A,D 1 4 Z 0 H C	ADC A,E 1 4 Z 0 H C	ADC A,H 1 4 Z 0 H C	ADC A,L 1 4 Z 0 H C	ADC A,(HL) 1 8 Z 0 H C	ADC A,A 1 4 Z 0 H C
9x	SUB B 1 4 Z 1 H C	SUB C 1 4 Z 1 H C	SUB D 1 4 Z 1 H C	SUB E 1 4 Z 1 H C	SUB H 1 4 Z 1 H C	SUB L 1 4 Z 1 H C	SUB (HL) 1 8 Z 1 H C	SUB A 1 4 Z 1 H C	SBC A,B 1 4 Z 1 H C	SBC A,C 1 4 Z 1 H C	SBC A,D 1 4 Z 1 H C	SBC A,E 1 4 Z 1 H C	SBC A,H 1 4 Z 1 H C	SBC A,L 1 4 Z 1 H C	SBC A,(HL) 1 8 Z 1 H C	SBC A,A 1 4 Z 1 H C
Ax	AND B 1 4 Z 0 1 0	AND C 1 4 Z 0 1 0	AND D 1 4 Z 0 1 0	AND E 1 4 Z 0 1 0	AND H 1 4 Z 0 1 0	AND L 1 4 Z 0 1 0	AND (HL) 1 8 Z 0 1 0	AND A 1 4 Z 0 1 0	XOR B 1 4 Z 0 0 0	XOR C 1 4 Z 0 0 0	XOR D 1 4 Z 0 0 0	XOR E 1 4 Z 0 0 0	XOR H 1 4 Z 0 0 0	XOR L 1 4 Z 0 0 0	XOR (HL) 1 8 Z 0 0 0	XOR A 1 4 Z 0 0 0
Bx	OR B 1 4 Z 0 0 0	OR C 1 4 Z 0 0 0	OR D 1 4 Z 0 0 0	OR E 1 4 Z 0 0 0	OR H 1 4 Z 0 0 0	OR L 1 4 Z 0 0 0	OR (HL) 1 8 Z 0 0 0	OR A 1 4 Z 0 0 0	CP B 1 4 Z 1 H C	CP C 1 4 Z 1 H C	CP D 1 4 Z 1 H C	CP E 1 4 Z 1 H C	CP H 1 4 Z 1 H C	CP L 1 4 Z 1 H C	CP (HL) 1 8 Z 1 H C	CP A 1 4 Z 1 H C
Cx	RET NZ 1 20/8 - - - -	POP BC 1 12 - - - -	JP NZ,a16 3 16/12 - - - -	JP a16 3 16 - - - -	CALL NZ,a16 3 24/12 - - - -	PUSH BC 1 16 - - - -	ADD A,d8 2 8 Z 0 H C	RST 00H 1 16 - - - -	RET Z 1 20/8 - - - -	RET 1 16 - - - -	JP Z,a16 3 16/12 - - - -	PREFIX CB 1 4 - - - -	CALL Z,a16 3 24/12 - - - -	CALL a16 3 24 - - - -	ADC A,d8 2 8 Z 0 H C	RST 08H 1 16 - - - -
Dx	RET NC 1 20/8 - - - -	POP DE 1 12 - - - -	JP NC,a16 3 16/12 - - - -		CALL NC,a16 3 24/12 - - - -	PUSH DE 1 16 - - - -	SUB d8 2 8 Z 1 H C	RST 10H 1 16 - - - -	RET C 1 20/8 - - - -	RETI 1 16 - - - -	JP C,a16 3 16/12 - - - -		CALL C,a16 3 24/12 - - - -		SBC A,d8 2 8 Z 1 H C	RST 18H 1 16 - - - -
Ex	LDH (a8),A 2 12 - - - -	POP HL 1 12 - - - -	LD (C),A 2 8 - - - -			PUSH HL 1 16 - - - -	AND d8 2 8 Z 0 1 0	RST 20H 1 16 - - - -	ADD SP,r8 2 16 0 0 H C	JP (HL) 1 4 - - - -	LD (a16),A 3 16 - - - -				XOR d8 2 8 Z 0 0 0	RST 28H 1 16 - - - -
Fx	LDH A,(a8) 2 12 - - - -	POP AF 1 12 Z N H C	LD A,(C) 2 8 - - - -	DI 1 4 - - - -		PUSH AF 1 16 - - - -	OR d8 2 8 Z 0 0 0	RST 30H 1 16 - - - -	LD HL,SP+r8 2 12 0 0 H C	LD SP,HL 1 8 - - - -	LD A,(a16) 3 16 - - - -	EI 1 4 - - - -			CP d8 2 8 Z 1 H C	RST 38H 1 16 - - - -

Prefix CB

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	RLC B 2 8 Z 0 0 C	RLC C 2 8 Z 0 0 C	RLC D 2 8 Z 0 0 C	RLC E 2 8 Z 0 0 C	RLC H 2 8 Z 0 0 C	RLC L 2 8 Z 0 0 C	RLC (HL) 2 16 Z 0 0 C	RLC A 2 8 Z 0 0 C	RRC B 2 8 Z 0 0 C	RRC C 2 8 Z 0 0 C	RRC D 2 8 Z 0 0 C	RRC E 2 8 Z 0 0 C	RRC H 2 8 Z 0 0 C	RRC L 2 8 Z 0 0 C	RRC (HL) 2 16 Z 0 0 C	RRC A 2 8 Z 0 0 C
1x	RL B 2 8 Z 0 0 C	RL C 2 8 Z 0 0 C	RL D 2 8 Z 0 0 C	RL E 2 8 Z 0 0 C	RL H 2 8 Z 0 0 C	RL L 2 8 Z 0 0 C	RL (HL) 2 16 Z 0 0 C	RL A 2 8 Z 0 0 C	RR B 2 8 Z 0 0 C	RR C 2 8 Z 0 0 C	RR D 2 8 Z 0 0 C	RR E 2 8 Z 0 0 C	RR H 2 8 Z 0 0 C	RR L 2 8 Z 0 0 C	RR (HL) 2 16 Z 0 0 C	RR A 2 8 Z 0 0 C
2x	SLA B 2 8 Z 0 0 C	SLA C 2 8 Z 0 0 C	SLA D 2 8 Z 0 0 C	SLA E 2 8 Z 0 0 C	SLA H 2 8 Z 0 0 C	SLA L 2 8 Z 0 0 C	SLA (HL) 2 16 Z 0 0 C	SLA A 2 8 Z 0 0 C	SRA B 2 8 Z 0 0 0	SRA C 2 8 Z 0 0 0	SRA D 2 8 Z 0 0 0	SRA E 2 8 Z 0 0 0	SRA H 2 8 Z 0 0 0	SRA L 2 8 Z 0 0 0	SRA (HL) 2 16 Z 0 0 0	SRA A 2 8 Z 0 0 0
3x	SWAP B 2 8 Z 0 0 0	SWAP C 2 8 Z 0 0 0	SWAP D 2 8 Z 0 0 0	SWAP E 2 8 Z 0 0 0	SWAP H 2 8 Z 0 0 0	SWAP L 2 8 Z 0 0 0	SWAP (HL) 2 16 Z 0 0 0	SWAP A 2 8 Z 0 0 0	SRL B 2 8 Z 0 0 C	SRL C 2 8 Z 0 0 C	SRL D 2 8 Z 0 0 C	SRL E 2 8 Z 0 0 C	SRL H 2 8 Z 0 0 C	SRL L 2 8 Z 0 0 C	SRL (HL) 2 16 Z 0 0 C	SRL A 2 8 Z 0 0 C
4x	BIT 0,B 2 8 Z 0 1 -	BIT 0,C 2 8 Z 0 1 -	BIT 0,D 2 8 Z 0 1 -	BIT 0,E 2 8 Z 0 1 -	BIT 0,H 2 8 Z 0 1 -	BIT 0,L 2 8 Z 0 1 -	BIT 0,(HL) 2 16 Z 0 1 -	BIT 0,A 2 8 Z 0 1 -	BIT 1,B 2 8 Z 0 1 -	BIT 1,C 2 8 Z 0 1 -	BIT 1,D 2 8 Z 0 1 -	BIT 1,E 2 8 Z 0 1 -	BIT 1,H 2 8 Z 0 1 -	BIT 1,L 2 8 Z 0 1 -	BIT 1,(HL) 2 16 Z 0 1 -	BIT 1,A 2 8 Z 0 1 -
5x	BIT 2,B 2 8 Z 0 1 -	BIT 2,C 2 8 Z 0 1 -	BIT 2,D 2 8 Z 0 1 -	BIT 2,E 2 8 Z 0 1 -	BIT 2,H 2 8 Z 0 1 -	BIT 2,L 2 8 Z 0 1 -	BIT 2,(HL) 2 16 Z 0 1 -	BIT 2,A 2 8 Z 0 1 -	BIT 3,B 2 8 Z 0 1 -	BIT 3,C 2 8 Z 0 1 -	BIT 3,D 2 8 Z 0 1 -	BIT 3,E 2 8 Z 0 1 -	BIT 3,H 2 8 Z 0 1 -	BIT 3,L 2 8 Z 0 1 -	BIT 3,(HL) 2 16 Z 0 1 -	BIT 3,A 2 8 Z 0 1 -
6x	BIT 4,B 2 8 Z 0 1 -	BIT 4,C 2 8 Z 0 1 -	BIT 4,D 2 8 Z 0 1 -	BIT 4,E 2 8 Z 0 1 -	BIT 4,H 2 8 Z 0 1 -	BIT 4,L 2 8 Z 0 1 -	BIT 4,(HL) 2 16 Z 0 1 -	BIT 4,A 2 8 Z 0 1 -	BIT 5,B 2 8 Z 0 1 -	BIT 5,C 2 8 Z 0 1 -	BIT 5,D 2 8 Z 0 1 -	BIT 5,E 2 8 Z 0 1 -	BIT 5,H 2 8 Z 0 1 -	BIT 5,L 2 8 Z 0 1 -	BIT 5,(HL) 2 16 Z 0 1 -	BIT 5,A 2 8 Z 0 1 -
7x	BIT 6,B 2 8 Z 0 1 -	BIT 6,C 2 8 Z 0 1 -	BIT 6,D 2 8 Z 0 1 -	BIT 6,E 2 8 Z 0 1 -	BIT 6,H 2 8 Z 0 1 -	BIT 6,L 2 8 Z 0 1 -	BIT 6,(HL) 2 16 Z 0 1 -	BIT 6,A 2 8 Z 0 1 -	BIT 7,B 2 8 Z 0 1 -	BIT 7,C 2 8 Z 0 1 -	BIT 7,D 2 8 Z 0 1 -	BIT 7,E 2 8 Z 0 1 -	BIT 7,H 2 8 Z 0 1 -	BIT 7,L 2 8 Z 0 1 -	BIT 7,(HL) 2 16 Z 0 1 -	BIT 7,A 2 8 Z 0 1 -
8x	RES 0,B 2 8 - - - -	RES 0,C 2 8 - - - -	RES 0,D 2 8 - - - -	RES 0,E 2 8 - - - -	RES 0,H 2 8 - - - -	RES 0,L 2 8 - - - -	RES 0,(HL) 2 16 - - - -	RES 0,A 2 8 - - - -	RES 1,B 2 8 - - - -	RES 1,C 2 8 - - - -	RES 1,D 2 8 - - - -	RES 1,E 2 8 - - - -	RES 1,H 2 8 - - - -	RES 1,L 2 8 - - - -	RES 1,(HL) 2 16 - - - -	RES 1,A 2 8 - - - -
9x	RES 2,B 2 8 - - - -	RES 2,C 2 8 - - - -	RES 2,D 2 8 - - - -	RES 2,E 2 8 - - - -	RES 2,H 2 8 - - - -	RES 2,L 2 8 - - - -	RES 2,(HL) 2 16 - - - -	RES 2,A 2 8 - - - -	RES 3,B 2 8 - - - -	RES 3,C 2 8 - - - -	RES 3,D 2 8 - - - -	RES 3,E 2 8 - - - -	RES 3,H 2 8 - - - -	RES 3,L 2 8 - - - -	RES 3,(HL) 2 16 - - - -	RES 3,A 2 8 - - - -
Ax	RES 4,B 2 8 - - - -	RES 4,C 2 8 - - - -	RES 4,D 2 8 - - - -	RES 4,E 2 8 - - - -	RES 4,H 2 8 - - - -	RES 4,L 2 8 - - - -	RES 4,(HL) 2 16 - - - -	RES 4,A 2 8 - - - -	RES 5,B 2 8 - - - -	RES 5,C 2 8 - - - -	RES 5,D 2 8 - - - -	RES 5,E 2 8 - - - -	RES 5,H 2 8 - - - -	RES 5,L 2 8 - - - -	RES 5,(HL) 2 16 - - - -	RES 5,A 2 8 - - - -
Bx	RES 6,B 2 8 - - - -	RES 6,C 2 8 - - - -	RES 6,D 2 8 - - - -	RES 6,E 2 8 - - - -	RES 6,H 2 8 - - - -	RES 6,L 2 8 - - - -	RES 6,(HL) 2 16 - - - -	RES 6,A 2 8 - - - -	RES 7,B 2 8 - - - -	RES 7,C 2 8 - - - -	RES 7,D 2 8 - - - -	RES 7,E 2 8 - - - -	RES 7,H 2 8 - - - -	RES 7,L 2 8 - - - -	RES 7,(HL) 2 16 - - - -	RES 7,A 2 8 - - - -
Cx	SET 0,B 2 8 - - - -	SET 0,C 2 8 - - - -	SET 0,D 2 8 - - - -	SET 0,E 2 8 - - - -	SET 0,H 2 8 - - - -	SET 0,L 2 8 - - - -	SET 0,(HL) 2 16 - - - -	SET 0,A 2 8 - - - -	SET 1,B 2 8 - - - -	SET 1,C 2 8 - - - -	SET 1,D 2 8 - - - -	SET 1,E 2 8 - - - -	SET 1,H 2 8 - - - -	SET 1,L 2 8 - - - -	SET 1,(HL) 2 16 - - - -	SET 1,A 2 8 - - - -
Dx	SET 2,B 2 8 - - - -	SET 2,C 2 8 - - - -	SET 2,D 2 8 - - - -	SET 2,E 2 8 - - - -	SET 2,H 2 8 - - - -	SET 2,L 2 8 - - - -	SET 2,(HL) 2 16 - - - -	SET 2,A 2 8 - - - -	SET 3,B 2 8 - - - -	SET 3,C 2 8 - - - -	SET 3,D 2 8 - - - -	SET 3,E 2 8 - - - -	SET 3,H 2 8 - - - -	SET 3,L 2 8 - - - -	SET 3,(HL) 2 16 - - - -	SET 3,A 2 8 - - - -
Ex	SET 4,B 2 8 - - - -	SET 4,C 2 8 - - - -	SET 4,D 2 8 - - - -	SET 4,E 2 8 - - - -	SET 4,H 2 8 - - - -	SET 4,L 2 8 - - - -	SET 4,(HL) 2 16 - - - -	SET 4,A 2 8 - - - -	SET 5,B 2 8 - - - -	SET 5,C 2 8 - - - -	SET 5,D 2 8 - - - -	SET 5,E 2 8 - - - -	SET 5,H 2 8 - - - -	SET 5,L 2 8 - - - -	SET 5,(HL) 2 16 - - - -	SET 5,A 2 8 - - - -
Fx	SET 6,B 2 8 - - - -	SET 6,C 2 8 - - - -	SET 6,D 2 8 - - - -	SET 6,E 2 8 - - - -	SET 6,H 2 8 - - - -	SET 6,L 2 8 - - - -	SET 6,(HL) 2 16 - - - -	SET 6,A 2 8 - - - -	SET 7,B 2 8 - - - -	SET 7,C 2 8 - - - -	SET 7,D 2 8 - - - -	SET 7,E 2 8 - - - -	SET 7,H 2 8 - - - -	SET 7,L 2 8 - - - -	SET 7,(HL) 2 16 - - - -	SET 7,A 2 8 - - - -