Osaamista
ja oivallusta
tulevaisuuden
tekemiseen

Aleksanteri Hirvonen

Game Boy -emulaattori

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikka

Insinöörityö

25.4.2019

Tekijä Otsikko Sivumäärä Aika	Aleksanteri Hirvonen Game Boy -emulaattori 41 sivua 25.4.2019
Tutkinto	insinööri (AMK)
Tutkinto-ohjelma	tieto- ja viestintätekniikka
Ammatillinen pääaine	pelisovellukset
Ohjaaja	lehtori Miikka Mäki-Uuro

Insinöörityön tarkoituksena oli toteuttaa Game Boy -emulaattori ja tuoda esille toteutuksen aikana hyväksi havaittuja emulaattoriohjelmoinnin tekniikoita.

Työssä toteutettiin Game Boy -emulaattori C++-ohjelmointikielellä noudattaen suositeltuja ohjelmointitapoja, kuten abstrahointia, alustariippumattomuutta ja RAII-periaatetta. Toteutuksessa käytettiin apuna POSIX- ja Windows-rajapintaa tiedostonkäsittelyyn sekä SDL-kehityskirjastoa ikkunointiin, tapahtumienkäsittelyyn ja grafiikan esittämiseen. Lopuksi emulaattorista käännettiin ohjelmatiedoston lisäksi vielä selainpohjainen WebAssembly-versio Emscripten-työkalun avulla.

Game Boyn pelikasetin, muistin, suorittimen, keskeytysten ja pikselinkäsittely-yksikön emulointi dokumentoitiin. Emuloinnista esitettiin myös koodiesimerkkejä.

Työn tuloksena aikaansaatu Game Boy -emulaattori täyttää sille asetetut tavoitteet. Emulaattorilla voi pelata monia Game Boy -pelejä ongelmitta, ja sen toimivuutta testattiin testi-ROMeja käyttäen.

Työtä voidaan käyttää emulaattoriohjelmoinnin opiskeluun.

Avainsanat	emulaattori, tietokonearkkitehtuuri, Game Boy, WebAssembly

Author Title Number of Pages Date	Aleksanteri Hirvonen Game Boy Emulator 41 pages 25 April 2019
Degree	Bachelor of Engineering
Degree Programme	Information and Communications Technology
Professional Major	Game Applications
Instructor	Miikka Mäki-Uuro, Senior Lecturer

The goal of this Bachelor's thesis was to develop a Game Boy emulator and describe emulator programming techniques that proved out to be beneficial during the development process.

The Game Boy emulator was programmed using the C++ programming language, following recommended practices such as abstraction, platform-independence and RAII. To assist the development process, the POSIX and Windows API were used for file I/O, and SDL was used for windowing, event handling and graphics displaying. Ultimately, the Emscripten toolchain was used to compile a WebAssembly version of the emulator.

Emulation of the Game Boy cartridge, memory management unit, central processing unit, interrupts and pixel-processing unit was documented. The documentation was also backed up by code examples.

The developed Game Boy emulator fulfills the goals that were set. It runs most Game Boy games well, largely thanks to testing using test ROMs. However, there is still room for improvement, as neither audio nor the serial communication between two Game Boys were implemented.

Additionally, the thesis serves as a good base for learning emulator programming. After reading the thesis, the reader could develop a similar emulator themselves.

Keywords	emulator, computer architecture, Game Boy, WebAssembly

Sisällys

1 Johdanto

Insinöörityössä perehdytään emulaattoriohjelmointiin toteuttamalla Game Boy -emulaattori. Työn tavoitteena on saada aikaan emulaattori, jolla on mahdollista pelata Game Boy -pelejä tietokoneella. Tarkoituksena on, että insinöörityöraporttia voidaan käyttää myös pelikonsoliemulaattorin toteuttamisen opiskeluun.

Game Boy -emulaattori toteutetaan C++-ohjelmointikielellä käyttäen apuna SDL-kirjastoa ikkunointiin, tapahtumienkäsittelyyn ja grafiikan esittämiseen. Perinteisen ohjelmatiedoston lisäksi emulaattorista käännetään lopuksi selainpohjainen WebAssembly-versio Emscripten-työkalun avulla. Äänentoisto ja kahden Game Boyn välinen sarjaliikenne jätetään toteuttamatta aikarajoitteiden vuoksi.

Insinöörityöraportin luvuissa edetään emulaattorin osa-alueiden loogisessa toteutusjärjestyksessä, jotta lukija saa aidon kokonaiskuvan siitä, miten voisi itse toteuttaa vastaavan projektin. Merkittävät emulaattorin toteutukseen liittyvät valinnat perustellaan kussakin työvaiheessa tehtävien arvioiden pohjalta.

Työn teknisen luonteen vuoksi lukijalta oletetaan ohjelmointi- ja lukujärjestelmäosaamista. Hyvä tietokonearkkitehtuurin tuntemus helpottaa tekstin ymmärtämistä huomattavasti, mutta ei ole välttämätöntä.

2 Työn lähtötiedot

Game Boy -konsoli

Game Boy on Nintendon vuonna 1989 julkaisema 8-bittinen käsikonsoli, joka on samalla yksi maailman myydyimmistä pelikonsoleista. Kuten kuvasta 1 nähdään, Game Boy on hyvin yksinkertainen pelikonsoli: siinä on ristiohjaimen lisäksi vain neljä näppäintä (A, B, start ja select) ja sen piirtämä grafiikka sisältää enintään neljää eri harmaan sävyä. Tunnettuja Game Boy -pelejä ovat mm. Tetris sekä Pokemon Red ja Blue. (1.)

Kuva 1. Game Boy -konsoli (2).

Alkuperäisen Game Boyn lisäksi Game Boy -tuoteperhettä on jatkettu Game Boy Color- ja Game Boy Advance -konsoleilla, mutta tässä työssä keskitytään vain alkuperäiseen Game Boy -konsoliin (1).

Emulaattori

Tämän työn kontekstissa emulaattori on tietokoneohjelma, joka saa sitä suorittavan tietokoneen käyttäytymään kohdejärjestelmän tavoin eli emuloimaan kohdejärjestelmää (3). Käytännössä emulaattori siis toisintaa kohdejärjestelmän toiminnallisuuden ohjelmallisesti; esimerkiksi Game Boy -emulaattorilla voi pelata Game Boy -pelejä ilman alkuperäistä Game Boy -konsolia.

Laajemmin käsiteltynä emulaattorin määritelmään voitaisiin lukea mukaan myös laitteistopohjaiset emulaattorit (3), mutta niihin ei keskitytä tässä työssä.

Syklitarkkuus

Emulaattorin syklitarkkuus (cycle accuracy) mittaa sitä, kuinka tarkasti emuloitujen komponenttien ajoitukset vastaavat kohdejärjestelmän komponenttien ajoituksia (4).

Jos emulaattorin syklitarkkuus on kehno, asiat tapahtuvat emulaattorilla eri aikaan kuin alkuperäisellä laitteella, mikä saattaa pahimmassa tapauksessa johtaa kelvottomaan käyttökokemukseen. Hyvä syklitarkkuus taas saa emuloidun kokemuksen vastaamaan paremmin kokemusta alkuperäisellä laitteella. Täydellisen syklitarkkuuden saavuttaminen on kuitenkin haastavaa, koska useista laitteista, Game Boy mukaan lukien, opitaan edelleen uutta. Lisäksi täydellisen syklitarkkuuden emulointi on usein erittäin raskasta.

Tässä työssä toteutettavassa Game Boy -emulaattorissa pyritään kohtuulliseen syklitarkkuuteen. Tavoitteena on, että toteutettavalla emulaattorilla voi pelata useimpia Game Boy -pelejä ongelmitta. Harvinaisiin ja huonosti tunnettuihin emulointiongelmiin ei siis etsitä ratkaisuja.

Ohjelmointikielen valinta

Game Boy -emulaattorin tapauksessa käytettävällä ohjelmointikielellä ei ole kovin suurta merkitystä. Game Boy on alustana sen verran yksinkertainen, että emulaattorin mahdolliset suorituskykyongelmat ovat ennemminkin kiinni emulaattorin ohjelmoijan ohjemointitaidoista kuin käytettävän ohjelmointikielen rajoitteista. Edistyneempien alustojen emulointiin soveltuvien ohjelmointikielten määrä sen sijaan olisi huomattavasti pienempi etenkin hyvään syklitarkkuuteen pyrittäessä.

Yleisesti pelikonsoliemulaattorin ohjelmointiin on suositeltavaa valita ohjelmointikieli, jossa on vastineet emuloitavan pelikonsolin lukutyypeille. Esimerkiksi Game Boyn tapauksessa tarvittavia lukutyyppejä ovat 8- ja 16-bittiset kokonaisluvut. Mikäli valittu kieli ei tue emuloitavan alustan lukutyyppejä suoraan, emulaattorin ohjelmoijan täytyy itse kehittää työkalut niiden hallintaan.

Tässä työssä emulaattorin toteutuskieleksi valitaan C++, koska siinä on Game Boy -emulaattorin toteutusta ajatellen hyödyllisiä ominaisuuksia: 8- ja 16-bittiset kokonaislukutyypit ([u]int8_t ja [u]int16_t), luokat sekä kattava standardikirjasto. Lisäksi tekijän aiempi kokemus kielestä vaikuttaa valintaan.

Lukunotaatio

Koska suuri osa emulaattorien logiikasta perustuu bittipohjaiseen lukujenkäsittelyyn mutta binäärilukujen kirjoittaminen ja lukeminen on melko vaivalloista, tässä työssä käytetään paljon 16-kantaista heksadesimaalijärjestelmää, jonka yhdellä merkillä (0–9, A–F) voi ilmaista neljä bittiä kerrallaan. Heksadesimaalijärjestelmän luvut merkitään työssä 0x-etuliitteellä, kuten monissa ohjelmointikielissäkin.

3 Pelikasetin emulointi

Alkuperäisellä Game Boy -konsolilla pelattava peli ladataan laitteen muistiin kasettiliittimeen (cartridge port) liitetyltä pelikasetilta (game cartridge) (1). Koska tässä työssä toteutettava emulaattori on tietokoneohjelma, siinä ei ole konsolin fyysistä kasettiliitintä. Siksi peli täytyy ladata muistiin pelikasetin sijaan tietokoneella sijaitsevasta pelikasetin muistisisältöä vastaavasta tiedostosta. Näitä pelikasetin muistisisältöä vastaavia tiedostoja kutsutaan tyypillisesti ”ROMeiksi” (ROM, read-only memory). Tässä työssä ROMeja kutsutaan kuitenkin pääosin pelitiedostoiksi tarpeettoman sanantaivuttelun välttämiseksi.

3.1 Tiedoston avaaminen ja sulkeminen

Jotta emulaattori voisi lukea pelitiedostoa, täytyy tiedosto ensin avata. Tiedoston avaamiseen on saatavilla useita työkaluja: C++-standardikirjaston ifstream-luokat, yhteensopivan C-standardikirjaston fopen-funktio sekä ulkoisten ohjelmointirajapintojen funktiot ja luokat. Koska binääritiedoston koon lukeminen luotettavasti C++-standardikirjaston avulla onnistuu vasta uusimmassa C++17-standardissa (5) eikä C++-standardi sellaisenaan tue muistiinmappausta, käytetään tässä työssä pelitiedoston käsittelyyn ulkoisia ohjelmointirajapintoja.

Ulkoisista ohjelmointirajapinnoista yksinkertaisinta on käyttää saatavilla olevaa käyttöjärjestelmärajapintaa. Siksi tässä työssä käytetään GNU/Linuxille ja MacOS:lle

käännettäessä POSIX-rajapintaa (POSIX API) ja Windowsille käännettäessä Windows-rajapintaa (Windows API).

POSIX-rajapinnassa tiedosto avataan open-funktiolla ja suljetaan close-funktiolla (6; 7). Epäselvän tiedostonkäsittelykoodin ja turhien muistivuotojen välttämiseksi luodaan posix-nimiavaruuteen esimerkkikoodin 1 mukainen apuluokka File, joka automaattisesti konstruktorissaan avaa pyydetyn tiedoston ja destruktorissaan sulkee sen. Tällaista resurssienhallinnan automatisoivaa ohjelmointitapaa kutsutaan nimellä RAII (resource acquisition is initialization) (8).

```
namespace cherry_gb::file_io::posix {

class File final {
public:
      File(const char* const path, const int flags)
            : fd(open(path, flags))
      {
            if (fd == -1) {
                  std::ostringstream error_msg_stream;
                  error_msg_stream << "Opening file " << path << " failed";
                  throw errors::ErrnoError(error_msg_stream.str());
            }
      }

      ~File() { close(fd); }
      File(const File&) = delete;
      File& operator=(const File&) = delete;

      const int fd; // File descriptor
};

} // namespace cherry_gb::file_io::posix
```

Esimerkkikoodi 1. RAII-periaatteen mukainen luokka, joka automatisoi tiedoston avaamisen ja sulkemisen POSIX-rajapintaa käyttäen. ErrnoError on C++-standardikirjaston runtime_error-luokasta periytetty luokka, jonka virhekuvaukseen sisältyy errno-tilaa kuvaava viesti.

File-apuluokasta luodaan täsmälleen samalla periaatteella myös windows-nimiavaruuteen vastine, joka vain käyttää POSIX-rajapinnan sijaan Windows-rajapintaa. POSIX-rajapinnan open- ja close-funktioita riittävästi vastaavat funktiot Windows-rajapinnassa ovat CreateFileA ja CloseHandle (9).

3.2 Tiedostokoon lukeminen

Tiedostokoko saadaan POSIX-rajapinnassa luettua käyttämällä tiedoston tilan (file status) selvittämiseen tarkoitettua fstat-funktiota. Onnistuessaan fstat-funktio täyttää stat-struktuuriin tiedoston tilaan liittyviä tietoja, kuten tiedostokoon. Tiedostokoko tallentuu stat-struktuurin st_size-muuttujaan. Jotta tiedostokoon kysely olisi mahdollisimman yksinkertaista, määritellään posix-nimiavaruuden File-luokkaan esimerkkikoodin 2 mukainen get_size-funktio. (10.)

```
std::size_t File::get_size() const
{
      struct stat status;
      if (fstat(fd, &status) == -1) {
            throw errors::ErrnoError("Getting file status failed");
      }
      return static_cast<std::size_t>(status.st_size);
}
```

Esimerkkikoodi 2. Tiedostokoon lukeminen POSIX-rajapintaa käyttäen.

Windows-rajapintaa käytettäessä tiedostokoon lukemiseen soveltuu GetFileSizeEx-funktio. Onnistuessaan GetFileSizeEx-funktio tallentaa tiedostokoon parametrinä annetun osoittimen osoittamaan LARGE_INTEGER-muuttujaan. GetFileSizeEx-funktiota apuna käyttäen määritellään windows-nimiavaruuden File-luokkaan esimerkkikoodin 3 mukainen get_size-funktio. (11.)

```
std::size File::get_size() const
{
      LARGE_INTEGER size;
      if (GetFileSizeEx(handle, &size) == 0) {
            throw std::runtime_error("Getting file size failed.");
      }
      return static_cast<std::size_t>(size);
}
```

Esimerkkikoodi 3. Tiedostokoon lukeminen Windows-rajapintaa käyttäen.

3.3 Tiedoston mappaaminen muistiin

Sen sijaan, että avatun pelitiedoston sisältö erikseen kopioitaisiin sitä varten allokoituun muistisäiliöön, se voidaan yksinkertaisesti mapata muistiin, mikä mahdollistaa pelitiedoston käsittelyn osoittimen avulla taulukon tavoin. Etenkin suuria tiedostoja käsitellessä

muistiinmappauksella saavutetaan koodin yksinkertaisuuden lisäksi myös suorituskykyetuja: tiedostonkäsittely järjestelmäkutsuilla on hitaampaa kuin muistiinmappaamisen mahdollistama suora muistinkäsittely osoittimia käyttäen. (12.)

Muistiinmappausta varten luodaan posix- ja windows-nimiavaruuksiin RAII-periaatteen mukaiset FileMapping-apuluokat. POSIX-rajapinnassa muistiinmappaukseen käytetään mmap-funktiota ja mappauksen poistamiseen munmap-funktiota (13). FileMapping-apuluokan POSIX-toteutusta esitellään esimerkkikoodissa 4.

```
namespace cherry_gb::file_io::posix {

class FileMapping final {
public:
      FileMapping(
            void* const map_address, const std::size_t map_length,
            const int protection, const int flags, const int fd,
            const off_t offset
      )
            : length(map_length)
            , address(
                  mmap(map_address, map_length, protection, flags, fd, offset)
            )
      {
            if (address == MAP_FAILED) {
                  throw errors::ErrnoError("Memory-mapping failed");
            }
      }

      ~FileMapping() { munmap(address, length); }
      FileMapping(const FileMapping&) = delete;
      FileMapping& operator=(const FileMapping&) = delete;

      const std::size_t length;
      void* const address;
};

} // namespace cherry_gb::file_io::posix
```

Esimerkkikoodi 4. RAII-periaatteen mukainen luokka tiedoston muistiinmappaamisen ja mappauksen poistamisen automatisointiin POSIX-rajapintaa käyttäen.

Windows-rajapinnassa vastaavan toiminnallisuuden saavuttamiseksi tarvitaan useampia funktioita: muistiinmappaus tehdään CreateFileMapping- ja MapViewOfFile-funktioilla, ja mappaus poistetaan CloseHandle-kutsuilla. Huomioitavaa on, että CloseHandle-funktiota tulee kutsua sekä CreateFileMapping- että MapViewOfFile-funktion paluuarvolle. Muilta osin Windows-toteutus noudattaa täsmälleen samaa periaatetta kuin POSIX-toteutus. (12.)

3.4 Alustariippuvaisuuksien abstrahointi

Luvuissa 3.1 ja 3.3 esitellyt apuluokat ovat alustariippuvaisia (platform-dependent): windows-nimiavaruuden File- ja FileMapping-luokkia voi käyttää vain Windowsille käännettäessä ja posix-nimiavaruuden versioita vain POSIX-yhteensopiville alustoille käännettäessä. Jotta näiden apuluokkien käyttö muualla koodissa olisi mahdollisimman yksinkertaista, luodaan väliin vielä alustariippumattomat (platform-independent) File- ja FileMapping-rajapintaluokat, joiden sisälle alustariippuvaiset osat kätketään.

Alustariippumattoman File-rajapintaluokan toteutusta esitellään esimerkkikoodissa 5. Alustariippumaton FileMapping-rajapintaluokka toteutetaan täsmälleen vastaavalla periaatteella.

```
namespace cherry_gb::file_io {

class File final {
public:
      enum class OpenMode { READ_ONLY, ... };

      File(const char* const path, const OpenMode mode)
            : m_underlying_file(
                  path,
            #ifndef _WIN32
                  ... // POSIX API-specific arguments
            #else
                  ... // Windows API-specific arguments
            #endif
            )
      { }

      ~File() = default;
      File(const File&) = delete;
      File& operator=(const File&) = delete;

      std::size_t get_size() const { return m_underlying_file.get_size(); }

private:
      using UnderlyingFileType =
      #ifndef _WIN32
            posix::File;
      #else
            windows::File;
      #endif

      const UnderlyingFileType m_underlying_file;
};

} // namespace cherry_gb::file_io
```

Esimerkkikoodi 5. Alustariippuvaiset File-toteutukset kätkevän alustariippumattoman File-luokan toteutus.

3.5 Cartridge-luokka

Luvussa 3.4 esiteltyjä File- ja FileMapping-luokkia käyttäen toteutetaan vielä lopullinen rajapinta pelikasetin käsittelyyn: Cartridge-luokka. Sen tehtävänä on ensin varmentaa, että avattua pelitiedostoa voidaan emuloida Game Boy -pelinä ja sitten tarjota muulle emulaattorille pääsy pelitiedoston sisältöön.

Pelitiedoston emuloitavuuden varmentaminen on varsin yksinkertaista: tarkistetaan, että pelitiedoston koko on riittävän suuri Game Boyn muistikartan perusteella. Pääsy pelitiedoston sisältöön on myös helppo tarjota: määritellään funktio get_bytes, joka palauttaa osoittimen FileMapping-luokassa mapatun muistialueen alkuun.

Lisäksi Cartridge-luokkaan tallennetaan vielä pelitiedoston käyttämän muistipankkikontrollerin (MBC, memory bank controller) tyyppi, joka kuvailee pelitiedoston sisältöä ja vaikuttaa Game Boyn muistin toimintaan. Muistipankkikontrollerin tyypin määrittää pelitiedoston sisällä osoitteessa 0x147 sijaitseva 8-bittinen arvo (14, s. 53).

Cartridge-luokan toteutusta esitellään esimerkkikoodissa 6.

```
class Cartridge final {
public:
      constexpr static std::size_t MIN_SIZE = 0x8000; // 32 kB

      explicit Cartridge(const char* const path)
            // Client code can dismiss File after it has been mapped.
            // Hence, the temporary File.
            : Cartridge(
                  file_io::File(path, file_io::File::OpenMode::READ_ONLY)
            )
      { }

      ~Cartridge() = default;
      Cartridge(const Cartridge&) = delete;
      Cartridge& operator=(const Cartridge&) = delete;

      std::size_t get_size() const
      {
            return m_file_mapping.length;
      }

      const std::uint8_t* get_bytes() const
      {
            return static_cast<const std::uint8_t*>(m_file_mapping.address);
      }
```

```
private:
      explicit Cartridge(const file_io::File& file)
            : m_file_mapping(file, file_io::FileMapping::Protection::READ)
            , mapper_type((
                  [&]() {
                        if (get_size() < MIN_SIZE) {
                              throw std::runtime_error("File too small.");
                        }
                  }(),
                  parse_mapper_type(get_bytes()[0x147])
            ))
      { }

      const file_io::FileMapping m_file_mapping;

public:
      const mapper::Type mapper_type;
};
```

Esimerkkikoodi 6. Cartridge-luokan toteutus. Tiedostokoon tarkistus suoritetaan luokan yksityisessä konstruktorissa.

3.6 Pelitiedostopolun lukeminen

Määritellyn Cartridge-luokan julkinen konstruktori vaatii parametrikseen tiedostopolun, josta pelitiedosto ladataan. Edistyneissä emulaattoreissa käyttäjä yleensä syöttää tiedostopolun graafista käyttöliittymää käyttäen. Tässä työssä kuitenkin keskitytään pääasiallisesti emulointiin, joten graafista käyttöliittymää tiedostopolun syöttämiselle ei toteuteta. Sen sijaan tiedostopolku luetaan käyttäjän ohjelmalle syöttämistä komentoriviparametreistä esimerkkikoodin 7 mukaisesti.

```
int main(int argc, char** argv)
{
      if (argc <= CARTRIDGE_PATH_ARG_INDEX) {
            return 1;
      }
      try {
            const cherry_gb::game_boy::Cartridge cartridge(
                  argv[CARTRIDGE_PATH_ARG_INDEX]
            );
      } catch (const std::exception& error) {
            // ...
      }
}
```

Esimerkkikoodi 7. Pelitiedostopolun lukeminen komentoriviparametreistä.

4 Muistin emulointi

Muistin emulointi on lähtökohtaisesti hyvin yksinkertaista: muistia mallinnetaan säiliönä, jonka sisältöä muut emulaattorin komponentit voivat lukea ja muokata. Muistin emuloinnin haasteellisuus piilee kuitenkin siinä, miten emuloitavan kohdejärjestelmän muisti käyttäytyy. Esimerkiksi muistiin kirjoittamista ja muistista lukemista voi olla rajoitettu, tai tiettyjen muistiosoitteiden käyttö luku- tai kirjoitusoperaatioissa saattaa muuttaa ajettavan ohjelman suorituspolkua.

4.1 Muistinkäsittelyn rajapintaistaminen

Game Boyn osoiteväylä (address bus) on 16-bittinen, mikä tarkoittaa että Game Boylla on käytössä muistiosoitteet 0x0000–0xFFFF (yhteensä 65536 eli 2^{16} osoitetta). Tätä 16-bittistä osoiteavaruutta (address space) Game Boy käsittelee operaatiosta riippuen joko yksitavuisina (8-bittisinä) tai kaksitavuisina (16-bittisinä) yksikköinä. (14, s. 10.)

Koska muunnoksia 16-bittisten ja 8-bittisten kokonaislukujen välillä tullaan tarvitsemaan paljon, määritellään toisteisuuden välttämiseksi funktiot create_u16, get_msb (most significant byte, merkitsevin tavu) ja get_lsb (least significant byte, vähiten merkitsevä tavu). Merkitsevin tavu saadaan siirtämällä 16-bittistä kokonaislukua kahdeksan bittiä oikealle. Vähiten merkitsevä tavu taas saadaan valitsemalla 16-bittisen kokonaisluvun kahdeksan vähiten merkitsevää bittiä AND-bittioperaatiolla. Funktioita esitellään esimerkkikoodissa 8.

```
constexpr std::uint16_t create_u16(
     const std::uint8_t msb, const std::uint8_t lsb
)
{ return static_cast<std::uint16_t>((msb << 8) | lsb); }

constexpr std::uint8_t get_msb(const std::uint16_t value)
{ return static_cast<std::uint8_t>(value >> 8); }

constexpr std::uint8_t get_lsb(const std::uint16_t value)
{ return static_cast<std::uint8_t>(value & 0xFF); }
```

Esimerkkikoodi 8. Apufunktioita muunnoksiin 8-bittisten ja 16-bittisten kokonaislukujen välillä. Constexpr-avainsana mahdollistaa tietyissä tilanteissa muunnoksien laskemisen jo ohjelman käännösvaiheessa.

Käsiteltäessä muistia yksitavuisina yksikköinä tavujärjestyksellä (byte order) ei ole merkitystä, sillä käsiteltäviä tavuja on vain yksi kerrallaan. Kaksitavuisina yksikköinä käsiteltäessä tavujärjestyksellä taas on suuri merkitys: big endian -tavujärjestyksessä kaksitavuinen heksadesimaaliarvo 0xFF00 vastaa desimaaliarvoa 65280, kun taas little endian -tavujärjestyksessä se vastaa desimaaliarvoa 255.

Tavujärjestysvirheiden välttämiseksi luodaan esimerkkikoodin 9 mukainen Mmu-luokka (MMU, memory-management unit), joka tarjoaa tavujärjestysriippumattoman (endianness-agnostic) rajapinnan muistinkäsittelyyn. Luokkaan määritellään jäsenmuuttujaksi Game Boyn osoiteavaruutta vastaava tavutaulukko m_bytes. Tämän tavutaulukon käsittelyä varten määritellään funktiot read_u8 ja write_u8 (yksitavuiseen käsittelyyn) sekä read_u16 ja write_u16 (kaksitavuiseen käsittelyyn). Koska Game Boy käyttää little endian -tavujärjestystä, vähiten merkitsevä tavu tulee muistissa ensin.

```
class Mmu final {
public:
	constexpr static std::size_t SIZE = 0x10000; // 64 kB

	// ...

	std::uint8_t read_u8(const std::uint16_t address) const
	{
		return m_bytes.at(address);
	}

	std::uint16_t read_u16(const std::uint16_t address) const
	{
		return create_u16(
			read_u8(static_cast<std::uint16_t>(address + 1)),
			read_u8(address)
		);
	}

	void write_u8(const std::uint16_t address, const std::uint8_t value)
	{
		m_bytes.at(address) = value;
	}

	void write_u16(const std::uint16_t address, const std::uint16_t value)
	{
		write_u8(address, get_lsb(value));
		write_u8(static_cast<std::uint16_t>(address + 1), get_msb(value));
	}

private:
	std::array<std::uint8_t, SIZE> m_bytes;
};
```

Esimerkkikoodi 9. Mmu-luokan muistinkäsittelyrajapinnan toteutus.

4.2 Game Boyn muistikartta

Muistikartta (memory map) esittää, millaisiin osiin muisti on jaoteltu. Näitä osia voivat olla esimerkiksi lukumuisti (ROM, read-only memory), videomuisti (VRAM, video random access memory) tai muistiinmapattu siirräntä (MMIO, memory-mapped input/output). Game Boyn muistikarttaa esitellään taulukossa 1.

Taulukko 1. Game Boyn muistikartta (14, s. 6).

Muistialue	**Käyttötarkoitus**
0x0000–0x3FFF	Pelikasetin lukumuistipankki 0
0x4000–0x7FFF	Vaihdettava pelikasetin lukumuistipankki 1–n
0x8000–0x9FFF	Videomuisti
0xA000–0xBFFF	Pelikasetin staattinen hajasaantimuisti
0xC000–0xDFFF	Game Boyn sisäinen hajasaantimuisti
0xE000–0xFDFF	Kopio muistialueesta 0xC000–0xDDFF
0xFE00–0xFE9F	Objektiattribuuttikartta
0xFEA0–0xFEFF	Käyttämätön muistialue
0xFF00–0xFF7F	Siirräntärekisterit
0xFF80–0xFFFE	Zero page -hajasaantimuisti
0xFFFF	Sallitut keskeytykset

Game Boyn muistikartta alkaa lukumuistista, jota on 32 kilotavua. Ensimmäiset 16 kilotavua lukumuistista, osoitteet 0x0000–0x3FFF, sisältävät aina pelikasetin ensimmäisen lukumuistipankin (ROM bank). Loput 16 kilotavua lukumuistista, osoitteet 0x4000–0x7FFF, sisältävät vaihdettavan lukumuistipankin. Pelikasetti voi siis sisältää useita lukumuistipankkeja, joita vaihtelemalla saadaan tarjottua pelaajalle vaihtuvaa sisältöä. Esimerkiksi Super Mario Land -pelissä vaihdellaan lukumuistipankkeja pelin aikana, kun taas yksinkertainen Tetris-peli käyttää vain kahta lukumuistipankkia. (14.)

Jotta Mmu-luokka saadaan emuloimaan lukumuistia, määritellään sille konstruktori, joka kopioi Cartridge-oliolta saatavaa pelitiedostodataa lukumuistin alueelle m_bytes-

tavutaulukkoon. Kopiointiin soveltuu varsin hyvin standardikirjaston memcpy-funktio (15), joka kopioi halutun määrän tavuja annettuun kohteeseen määritellystä lähteestä. Lisäksi Mmu-luokkaan määritellään vielä funktio set_rom_bank, jonka avulla suoritetaan lukumuistipankin vaihto niin ikään memcpy-funktion avulla. Konstruktoria ja set_rom_bank-funktiota esitellään esimerkkikoodissa 10.

```
Mmu::Mmu(const Cartridge& cartridge)
      : m_cartridge(cartridge) // m_cartridge is of type const Cartridge&.
      , m_bytes()
{
      std::memcpy(m_bytes.data(), cartridge.get_bytes(), Cartridge::MIN_SIZE);
}

void Mmu::set_rom_bank(const std::uint8_t bank)
{
      constexpr auto bank_size = 0x4000; // 16 kB
      const auto start = bank * bank_size;
      if (m_cartridge.get_size() < (start + bank_size)) {
            throw std::runtime_error("Memory bank out of range.");
      }
      std::memcpy(
            &m_bytes.at(0x4000), m_cartridge.get_bytes() + start,
            bank_size
      );
}
```

Esimerkkikoodi 10. Laajennuksia Mmu-luokan toteutukseen lukumuistin emulointia varten.

Lukumuistin jälkeen Game Boyn muistikartan osoitteet 0x8000–0x9FFF on varattu videomuistille (14, s. 6), joka määrittelee näytölle piirrettävän grafiikan. Perinteisen ruutupuskurin (frame buffer) sijaan Game Boyn videomuisti käyttää tile map- ja tile set -rakenteita kuvaamaan näytölle piirrettävää grafiikkaa (16). Tähän perehdytään paremmin luvussa 8.

Videomuistin jälkeen muistikartassa on hajasaantimuistille (RAM, random access memory) varattu alue. Hajasaantimuistin ensimmäinen osio, osoitteet 0xA000–0xBFFF, on varattu pelikasetin ulkoiselle staattiselle hajasaantimuistille (SRAM, static RAM). Tämän staattisen hajasaantimuistin tilaa säilytetään yleensä pariston avulla, ja sitä käytetään esimerkiksi pelien tallennuksia varten. Loput hajasaantimuistista, osoitteet 0xC000–0xDFFF, on varattu Game Boyn sisäiselle keskusmuistille. Muistialue 0xE000–0xFDFF (echo RAM) on vain kopio muistialueesta 0xC000–0xDDFF, ja siksi sitä ei yleensä käytetä mihinkään. (14, s. 6.)

Osoitealuetta 0xFE00–0xFE9F kutsutaan objektiattribuuttikartaksi. Se on videomuistista erillinen alue, joka kuitenkin määrittää näytölle piirrettäviä objekteja. Objektiattribuuttikarttaan perehdytään paremmin luvussa 8.

Osoitteissa 0xFF00–0xFF7F on joukko muistiinmapattuja siirräntärekistereitä. Näistä rekistereistä voidaan lukea esimerkiksi keskeytysten, näppäinten, ajastimien ja grafiikan piirtämisen tilaa. Koska osaan siirräntärekisterien biteistä on myös mahdollista kirjoittaa, määritellään Mmu-luokkaan esimerkkikoodin 11 mukaiset funktiot write_u8_io ja write_u8_masked, joita käytetään siirräntärekistereihin kirjoittamiseen. (14.)

```
void Mmu::write_u8_io(const std::uint16_t address, const std::uint8_t value)
{
	switch (address) {
	case P1: // 0xFF00 (Joypad)
		write_u8_masked(P1, value, 0x30);
		break;
	// ...
	// Other masked I/O register writes
	// ...
	default:
		write_u8(address, value);
		break;
	}
}

void Mmu::write_u8_masked(
	const std::uint16_t address, const std::uint8_t value,
	const std::uint8_t write_mask
)
{
	write_u8(
		address,
		static_cast<std::uint8_t>(
			(read_u8(address) & ~write_mask) | (value & write_mask)
		)
	);
}
```

Esimerkkikoodi 11. Laajennuksia Mmu-luokan toteutukseen muistiinmapattujen siirräntärekisterien emulointia varten.

Muistiosoitteet 0xFF80–0xFFFE muodostavat Game Boyn zero page -hajasaantimuistin (zero page RAM). Sen käyttö on hyödyllistä, koska sen osoittaminen vie vain 8 bittiä 16 bitin sijaan. Esimerkiksi arvo osoitteessa 0xFF80 voidaan lukea zero page -muistia hyödyntämällä käyttämällä osoittimena vain yksitavuista arvoa 0x80. Myös Game Boyn pino (stack) alkaa oletusarvoisesti zero page -hajasaantimuistista.

Game Boyn osoiteavaruuden viimeinen osoitepaikka 0xFFFF säilyttää kullakin hetkellä sallittuja keskeytyksiä. Keskeytyksiin perehdytään luvussa 6.

5 Suorittimen emulointi

5.1 Suoritin

Suoritin (CPU, central processing unit) on komponentti, jonka tehtävä on suorittaa sille annettuja käskyjä (instructions). Nämä käskyt koostuvat yleensä kahdesta osasta: opcodesta (operation code), joka määrittää suoritettavan operaation, ja operandeista, jotka määrittävät suoritettavan operaation kohteet. Operandien määrä riippuu suoritin-arkkitehtuurista ja suoritettavasta käskystä. Käsky voi koostua myös pelkästä opcodesta, jolloin operaatio suoritetaan aina samoille operandeille.

5.2 Rekisterit

Käskyjä suorittaakseen suorittimen tarvitsee säilyttää sen nykyistä tilaa sekä käskyissä käsiteltävää dataa. Suoritin säilyttää näitä nopeasti saatavilla olevilla muistialueilla, joita kutsutaan rekistereiksi (processor register). Yleensä suorittimen käytössä on yleiskäyttöisiä rekistereitä (GPR, general-purpose register), erikoisrekistereitä (SPR, special-purpose register) ja tilarekisteri (flag register). (17.)

Yleiskäyttöinen rekisteri voi säilyttää sekä lukudataa että muistiosoitteita. Ohjelmoija voi siis käyttää yleiskäyttöisiä rekistereitä esimerkiksi säiliöinä matemaattisen laskutoimituksen termeille tai funktio-osoittimille (function pointer). (17.)

Erikoisrekisterit säilyttävät ohjelman suorituksen tilaa. Yleisimmät käyttötarkoitukset erikoisrekistereille ovat ohjelmalaskuri (PC, program counter) ja pino-osoitin (SP, stack pointer). Ohjelmalaskuri säilyttää ohjelman suorituksen nykyistä osoitetta, josta esimerkiksi seuraavaksi suoritettava käsky luetaan. Pino-osoitin taas säilyttää funktioiden osoitteiden ja parametrien tallettamiseen käytettävän pinon päällimmäisen alkion osoitetta. (17.)

Tilarekisteri säilyttää bittivektorina tietoja suorittimen tilasta. Tietty tilarekisterin bitti voi kertoa esimerkiksi oliko edellisen laskutoimituksen tulos nolla tai tapahtuiko laskutoimituksen aikana carry eli merkitsevimmän bitin ylivuoto. (17.)

5.3 Game Boyn suoritin

Game Boyn suorittimeksi mainitaan yleensä Sharp LR35902, jossa on paljon samoja ominaisuuksia kuin Intel 8080- ja Zilog Z80 -suorittimessa. Periaatteessa Sharp LR35902 on kuitenkin ennemminkin järjestelmäpiiri (SoC, system on chip) kuin pelkkä suoritin, koska se sisältää myös muistin, pikselinkäsittely-yksikön, äänikontrollerin ja muita komponentteja. Tarkkaa Game Boyn käyttämää ydinsuoritinta ei tämän työn kirjoittamishetkellä yleisesti tunneta, mutta todennäköisesti se on joko Sharp SM83 CPU core tai ainakin täysin SM83-yhteensopiva. (18; 19.)

Game Boyn suorittimessa on kahdeksan 8-bittistä rekisteriä: laskuri A (accumulator), tilarekisteri F (flags) sekä yleiskäyttöiset rekisterit B, C, D, E, H ja L. Nämä 8-bittiset rekisterit voidaan myös parittaa 16-bittisiksi rekistereiksi AF, BC, DE ja HL. Ohjelmalaskuri PC ja pino-osoitin SP ovat Game Boyn suorittimessa 16-bittisiä. (18.)

Game Boyn suorittimen käskykannassa (instruction set) on perustilassa 256 opcodea (0x00–0xFF). Näistä opcode 0xCB on varsin poikkeuksellinen: sitä käytetään etuliitteenä avaamaan toiset 256 opcodea, jotka keskittyvät bittimanipulaatioihin. Lisäksi huomionarvoista on, että perustilan 256 opcodesta 11 on käyttämättömiä ja niiden käyttö ohjelmassa saattaa aiheuttaa koko järjestelmän kaatumisen. (20.)

5.4 Cpu-luokka

Suorittimen rekisterit määrittävät suorittimen tilan, ja käskyjen suorittaminen taas on toiminnallisuutta. Siksi suorittimen ohjelmalliseen mallintamiseen sopii aiemmissa pelikasetti- ja muistitoteutuksissakin käytetty luokkarakenne. Luokan jäsenmuuttujiksi määritellään suorittimen rekisterit esimerkkikoodin 12 mukaisesti ja jäsenfunktioiksi suorittimen käskyt sekä apufunktiot.

```
class Cpu final {
public:
      explicit Cpu(Mmu& mmu)
            : m_mmu(mmu)
            , m_a(0x01), m_f(0xB0)
            , m_b(0x00), m_c(0x13)
            , m_d(0x00), m_e(0xD8)
            , m_h(0x01), m_l(0x4D)
            , m_sp(0xFFFE), m_pc(0x0100)
      { }

private:
      Mmu& m_mmu;
      std::uint8_t m_a, m_f, m_b, m_c, m_d, m_e, m_h, m_l;
      std::uint16_t m_sp, m_pc;
};
```

Esimerkkikoodi 12. Cpu-luokan alustava määrittely. Konstruktorissa asetetaan rekisterit alkuarvoihinsa. Rekisterien alkuarvot on poimittu lähteinä käytetyistä dokumenteista.

Käskyjen ja apufunktioiden toteutukseen perehdytään luvussa 5.7.

5.5 Tilarekisteri

Game Boyn tilarekisteri F sisältää neljä eri tilan ilmaisemiseen tarkoitettua lippua (flag): ZF (zero flag), NF (negative flag), HF (half-carry flag) ja CF (carry flag) (16). Koska monet Game Boyn suorittimen käskyistä käsittelevät näitä lippuja, määritellään lippujen käyttämille biteille nimetyt vakiot esimerkkikoodin 13 mukaisesti.

```
enum Flag : std::uint8_t {
      FLAG_Z = 0x80, // Operation result was zero.
      FLAG_N = 0x40, // Operation was a subtraction.
      FLAG_H = 0x20, // Half-carry
      FLAG_C = 0x10 // Carry
};
```

Esimerkkikoodi 13. Vakioiden määrittely tilarekisterin lipuille.

Lisäksi käsittelyä helpottamaan määritellään set_flags- ja reset_flags-funktiot, jotka vähentävät koodin toisteisuutta. Lippujen asettaminen set_flags-funktiossa toteutetaan OR-bittioperaatiolla. Lippujen nollaukseen taas käytetään AND-bittioperaatiota yhdessä NOT-bittioperaation kanssa. Set_flags-funktiosta toteutetaan myös versio, joka set-totuusarvoparametristä riippuen kutsuu joko lippuja asettavaa set_flags-funktiota tai lippuja nollaavaa reset_flags-funktiota. Funktioiden toteutusta esitellään esimerkkikoodissa 14.

```
void Cpu::set_flags(const std::uint8_t mask)
{
      m_f |= mask;
}

void Cpu::reset_flags(const std::uint8_t mask)
{
      m_f = static_cast<std::uint8_t>(m_f & ~mask);
}

void Cpu::set_flags(const std::uint8_t mask, const bool set)
{
      if (set) {
            set_flags(mask);
      } else {
            reset_flags(mask);
      }
}
```

Esimerkkikoodi 14. Apufunktioita tilarekisterin lippujen päivittämiseen. Parametrinä kaikkiin funktioihin annetaan lipuista koostuva bittimaski (bitmask).

5.6 8-bittisten rekisterien paritus

8-bittisten rekisterien A, F, B, C, D, E, H ja L paritusta 16-bittisiksi rekistereiksi AF, BC, DE ja HL tarvitaan todella monessa käskyssä. Paritusta varten tulee siis kehittää toteutusratkaisu.

Parituksen käsittely olisi mahdollista toteuttaa jokaisen käskyn kohdalla erikseen, mutta tämä johtaisi vakavaan koodin toisteisuuteen. Tällainen koodin toisteisuus loukkaa DRY-periaatetta (don’t repeat yourself) ja tekee ohjelmoijan työstä turhan vaivalloista. Jokin kestävämpi ratkaisu on siis tarpeen.

Moni olemassa oleva Game Boy -emulaattori käyttää parituksen toteutukseen union- ja struct-tyyppiä esimerkkikoodin 15 tavoin. Näin on periaatteessa mahdollista määritellä samalle muistialueelle 16-bittinen kokonaislukumuuttuja ja kaksi 8-bittistä kokonaislukumuuttujaa.

```
union {
      std::uint16_t af;
      struct { std::uint8_t a, f; };
};
```

Esimerkkikoodi 15. Rekisterien paritus union- ja struct-tyyppiä käyttäen.

Union- ja struct-tyyppien käytössä paritukseen piilee kuitenkin ongelmia:

- C++-standardi sallii vain unionin viimeiseksi asetetun jäsenmuuttujan lukemisen. Tämän kiertäminen vaatii kääntäjän epästandardeja laajennuksia (non-standard extensions).
- C++-standardi ei salli anonyymejä struct-määrittelyjä. Tämänkin kiertäminen vaatii epästandardeja laajennuksia.
- Esimerkki vaatisi big endian -tavujärjestyksen toimiakseen halutulla tavalla. Tavujärjestysriippumaton ratkaisu vaatisi esikääntäjän määritelmiä (preprocessor definitions) eri tavujärjestyksiä käyttäviä alustoja varten.

Vaikkakin union-struct-ratkaisu näyttää yksinkertaiselta ja olisi suorituskykymittapuulla tehokas, vastoin ohjelmointikielen standardia toimiminen ja ylimääräisten esikääntäjän määrittelyiden tarve tekevät ratkaisusta epästabiilin. Jokin vakaampi ratkaisu olisi mielekkäämpi.

Koska C++-kieli sallii operaattorien ylikirjoittamisen, on mahdollista luoda paritusta varten apuluokka, jota voi käyttää laskutoimitusoperaatioissa aivan kuin 16-bittistä kokonaislukua. Toteutetaan siis esimerkkikoodin 16 mukainen luokka Pair, jolle määritellään implisiittinen konversio uint16_t-tyyppiin sekä yhteen- ja vähennyslaskuoperaattorit uint16_t-tyypin muuttujien kanssa.

```
class Pair final {
public:
      Pair(std::uint8_t& msb, std::uint8_t& lsb)
            : m_msb(msb), m_lsb(lsb)
      { }

      ~Pair() = default;
      Pair(const Pair&) = delete;
      Pair& operator=(const Pair&) = delete;

      operator std::uint16_t() const // Implicit conversion
      {
            return create_u16(m_msb, m_lsb);
      }

      Pair& operator=(const std::uint16_t rhs)
      {
            m_msb = get_msb(rhs);
            m_lsb = get_lsb(rhs);
            return *this;
      }

      Pair& operator+=(const std::uint16_t rhs)
      {
            return *this = static_cast<std::uint16_t>(*this + rhs);
      }
```

```
        Pair& operator-=(const std::uint16_t rhs)
        {
              return *this = static_cast<std::uint16_t>(*this - rhs);
        }

        Pair& operator++() { return *this += 1; }
        Pair& operator--() { return *this -= 1; }

private:
        std::uint8_t& m_msb;
        std::uint8_t& m_lsb;
};
```

Esimerkkikoodi 16. Pair-luokan toteutus.

Toteutettu Pair-luokka noudattaa C++-standardia, ja sen toiminta on alustariippumatonta. Koska kääntäjät eivät aina kykene täydelliseen optimointiin, Pair-luokan suorituskyky ei välttämättä yllä täysin union-struct-toteutuksen tasolle. Saavutettu vakaus on kuitenkin arvokasta, ja suorituskykyhäviökin voidaan jättää huomiotta, koska Pair-luokan lukumuunnoksiin käytettävä aikaosuus on kokonaiskuvassa mitätön. Ratkaisuun voi siis olla tyytyväinen.

Cpu-luokkaan uusi Pair-luokka saadaan integroitua lisäämällä neljä Pair-tyyppistä jäsenmuuttujaa: m_af, m_bc, m_de ja m_hl. Tämän jälkeen niitä voidaan käyttää käskyjen toteutuksissa kuin 16-bittisiä kokonaislukuja.

5.7 Käskyjen toteutus

Suorittimen käskyjen toteuttamiseen on käytännössä kaksi eri vaihtoehtoa: jokaisen käskyn toteuttaminen erillisenä funktiona (tai switch-lauseen case-valinnan sisällä) tai parametrisoitujen yleispätevien funktioiden käyttö. Koska Game Boyn suorittimen käskykannassa todella monet käskyt suorittavat saman operaation eri operandeille, tässä työssä käytetään parametrisoituja yleispäteviä funktioita suorittimen käskyjen toteuttamiseen.

Yleispätevien funktioiden käyttöä selkeyttää kategorinen kahtiajako: määritellään funktiot sekä käskyille että käskyjen suorittamille operaatioille. Käskyfunktiot kutsuvat operaatiofunktiota ja palauttavat lisäksi käskyyn kuluvan ajan sykleinä (cycles). Käsky- ja operaatiofunktioiden toteutusta esitellään esimerkkikoodissa 17.

```
std::uint8_t Cpu::add_u8_u8_op(
      std::uint8_t target, const std::uint8_t source, const bool carry = false
)
{
      reset_flags(FLAG_N);
      const auto result_16 =
            static_cast<std::uint16_t>(target + source + carry);
      set_flags(FLAG_C, result_16 > 0xFF);
      set_flags(FLAG_H, ((target & 0xF) + (source & 0xF) + carry) > 0xF);
      target = static_cast<std::uint8_t>(result_16);
      set_flags(FLAG_Z, target == 0);
      return target;
}

std::uint8_t Cpu::adc_u8_u8_op(
      const std::uint8_t target, const std::uint8_t source
)
{
      return add_u8_u8_op(target, source, is_flag_set(FLAG_C));
}

Cycles Cpu::add_r8_r8_ins(
      std::uint8_t& target_register, const std::uint8_t source_register
)
{
      target_register = add_u8_u8_op(target_register, source_register);
      return 1;
}

Cycles Cpu::adc_r8_r8_ins(
      std::uint8_t& target_register, const std::uint8_t source_register
)
{
      target_register = adc_u8_u8_op(target_register, source_register);
      return 1;
}
```

Esimerkkikoodi 17. Esimerkkejä operaatio- ja käskyfunktioista. Op-pääte viittaa operaatiofunktioon ja ins-pääte käskyfunktioon. Cycles-tyyppi on alias kokonaislukutyypille unsigned long long int.

Itse käskyfunktioita täytyy kuitenkin vielä kutsua jostain. Tätä varten Cpu-luokkaan määritellään julkinen funktio execute_next_instruction, joka suorittaa aina seuraavan käskyn muistista. Seuraavan käskyn lukemiseen muistista määritellään vielä funktio read_u8, joka lukee muistista tavun ja kasvattaa ohjelmalaskuria. Funktioiden toteutusta esitellään esimerkkikoodissa 18.

```
Cycles Cpu::execute_next_instruction()
{
      switch (const auto opcode = read_u8(); opcode) {
      // ...
      case 0x80: return add_r8_r8_ins(m_a, m_b);
      // ...
      case 0x88: return adc_r8_r8_ins(m_a, m_b);
      // ...
      }
}
```

```
std::uint8_t Cpu::read_u8()
{
      return m_mmu.read_u8(m_pc++);
}
```

Esimerkkikoodi 18. Rajapintafunktio käskyjen suorittamiseen ja apufunktio käskyjen lukemiseen muistista.

Esimerkkikoodissa 18 esitellystä execute_next_instruction-rajapintafunktiosta muodostuu käytännössä koko emulaattorin ydin. Tässä vaiheessa aivan kaikkia käskyfunktioita ei kuitenkaan ole vielä mahdollista toteuttaa täysin, koska osa käskyistä vaatii keskeytyksiä.

5.8 GameBoy-luokka

Koska emuloitavien komponenttien määrä alkaa kasvaa, komponentit on syytä abstrahoida luokkaan sen sijaan, että niitä määriteltäisiin esimerkiksi main-funktion sisällä. Tätä varten toteutetaan esimerkkikoodin 19 mukainen GameBoy-luokka, jonka jäsenmuuttujia ovat Game Boyn komponentit eli tässä vaiheessa Cpu- ja Mmu-luokan oliot. Pelitiedoston pohjalta luotu Cartridge-olio annetaan GameBoy-luokalle konstruktorin parametrinä.

```
class GameBoy final {
public:
      explicit GameBoy(const Cartridge& cartridge)
            : m_cartridge(cartridge)
            , m_mmu(m_cartridge)
            , m_cpu(m_mmu)
      { }

      void run()
      {
            for (;;) {
                  m_cpu.execute_next_instruction();
            }
      }

private:
      const Cartridge& m_cartridge;
      Mmu m_mmu;
      cpu::Cpu m_cpu;
};
```

Esimerkkikoodi 19. GameBoy-luokan alustava toteutus. Luokkaa laajentavat myöhemmin työssä toteutettavat komponentit.

6 Keskeytysten emulointi

Keskeytys (interrupt) nimensä mukaisesti keskeyttää nykyisen suorituksen ja siirtyy suorittamaan keskeytyksenkäsittelijää (interrupt handler). Keskeytyksiä käytetään hälyttämään tapahtumista, jotka on syytä käsitellä heti. Game Boylla tällaisia tapahtumia ovat

- ruudun piirron loppuminen
- tietyn juovan piirtämisen aloitus
- ajastimen ylivuoto
- sarjaliikennesignaali
- näppäimen painallus.

Game Boylla keskeytyksiä hallitaan IME- (interrupt master enable), IF- (interrupt flags) ja IE-rekisterin (interrupt enable) avulla. (14.)

IME on täysin järjestelmän sisäinen totuusarvo, jota hallitaan EI- (enable interrupts), DI- (disable interrupts) ja RETI-käskyllä (return and enable interrupts). Jos IME:n arvo on false, mitään keskeytyksiä ei käsitellä. (14.)

IF ja IE sen sijaan ovat muistiinmapattuja siirräntärekistereitä. IF:n bitit vastaavat keskeytyspyyntöjä (IRQ, interrupt request), joita Game Boy lähettää aiemmin mainittujen tapahtumien yhteydessä. Lisäksi Game Boy -peli voi lähettää keskeytyspyynnön asettamalla haluttua keskeytystä vastaavan IF:n bitin arvoon 1. IE:n bitit taas määrittävät, mitä keskeytyksiä käsitellään, kun niistä saadaan pyyntö. Game Boy -peli voi esimerkiksi välillä jättää huomiotta kaikki tai tietyt keskeytykset.

Keskeytysten hallintaa varten luodaan IrqManager-luokka, joka vastaanottaa keskeytyspyyntöjä request_interrupt-funktion välityksellä ja vie ne suorittimen käsiteltäväksi handle_interrupts-funktiossa. Handle_interrupts-funktio noudattaa Game Boyn keskeytystenkäsittelyyn liittyvää priorisointia: IF- ja IE-rekisterin määrittämistä keskeytyksistä vain prioriteettijärjestyksessä ensimmäinen suoritetaan. Game Boy -pelin koodin täytyy keskeytyksenkäsittelyn jälkeen itse uudelleenasettaa IME arvoon true RETI- tai EI-käskyllä, jos keskeytyksiä on tarkoitus käsitellä lisää. Koska EI-käsky asettaa IME:n arvoon true vasta EI-käskyä seuraavan käskyn jälkeen, määritellään IrqManager-luokkaan

myös tämän viiveen hallitseva trigger_ei-funktio. IrqManager-luokan toteutusta esitellään esimerkkikoodissa 20.

```
class IrqManager final {
public:
	IrqManager(Mmu& mmu, cpu::Cpu& cpu)
		: ime(false)
		, m_cpu(cpu)
		, m_if(mmu.get_u8_ref(IF))
		, m_ie(mmu.get_u8_ref(IE))
		, m_ei(false)
	{ }

	~IrqManager() = default;
	IrqManager(const IrqManager&) = delete;
	IrqManager& operator=(const IrqManager&) = delete;

	void request_interrupt(const Interrupt interrupt)
	{
		m_if |= interrupt;
	}

	void handle_interrupts()
	{
		if (!ime) {
			// EI signal handling
			if (m_ei) {
				m_ei = false;
				ime = true;
			}
			return;
		}

		const auto interrupts = m_if & m_ie;
		for (const auto interrupt : { // Interrupts in priority order
			INTERRUPT_VBLANK, INTERRUPT_STAT, INTERRUPT_TIMER,
			INTERRUPT_SERIAL, INTERRUPT_JOYPAD
		}) {
			if ((interrupt & interrupts) != 0) {
				ime = false;
				m_if ^= interrupt;
				m_cpu.handle_interrupt(interrupt);
				return; // Only one interrupt is handled at once.
			}
		}
	}

	void trigger_ei() { m_ei = true; }

	bool ime; // Interrupt master enable
private:
	cpu::Cpu& m_cpu;
	std::uint8_t& m_if; // Interrupt flags
	std::uint8_t& m_ie; // Interrupt enable
	bool m_ei; // EI instruction signal
};
```

Esimerkkikoodi 20. IrqManager-luokan toteutus. For-silmukan sisällä kutsuttava Cpu-luokan handle_interrupt-funktio kutsuu call_u16_op-operaatiofunktiota käyttäen keskeytyksen määrittelemää osoitetta funktion tavoin.

Myös IrqManager-luokasta luodaan jäsenmuuttuja luvussa 5.8 esiteltyyn GameBoy-luokkaan.

7 SDL-kehityskirjaston käyttö

SDL (Simple DirectMedia Layer) on C-ohjelmointikielellä toteutettu alustariippumaton kehityskirjasto (cross-platform development library), joka tarjoaa matalan tason rajapinnan tietokoneen multimediakomponentteihin sekä työkaluja ikkunointiin ja tapahtumienkäsittelyyn (21). Koska Game Boy -emulaattorin tarvitsee esittää emuloitavan pelin grafiikkaa ikkunassa ja vastaanottaa näppäimenpainalluksia käyttäjältä, SDL on oiva kirjastovalinta; se mahdollistaa kaiken tarvittavan toiminnallisuuden. SDL:n sijaan voitaisiin käyttää myös esimerkiksi samankaltaista SFML-kirjastoa tai suoraan käyttöjärjestelmärajapintoja.

SDL on toteutettu C-kielellä, joka ei tue luokkia. C-kielessä yleinen tapa varmistaa varattujen resurssien vapauttaminen funktiosta poistuttaessa (mitä reittiä tahansa) on käyttää goto-lausetta ja hypätä funktion loppuun, jossa resurssit vapautetaan. Koska tässä työssä käytetään C++-kieltä, voidaan aiemmin esiteltyä yksinkertaista RAII-periaatetta käyttää SDL:n kanssa goto-lauseen sijaan.

Jotta SDL-rajapinnan funktioita voi käyttää onnistuneesti, täytyy SDL ensin initialisoida, mikä tehdään SDL_Init-funktiokutsulla. Vastaavasti ohjelman suljettaessa resurssien vapauttamiseksi kutsutaan SDL_Quit-funktiota. Näiden funktioiden käyttöä esitellään esimerkkikoodissa 21.

```
class Sdl final {
public:
      explicit Sdl(const Uint32 flags)
      {
            if (SDL_Init(flags) != 0) {
                  throw errors::SdlError("SDL initialization failed");
            }
      }

      ~Sdl()
      {
            SDL_Quit();
      }
```

```
      Sdl(const Sdl&) = delete;
      Sdl& operator=(const Sdl&) = delete;
};
```

Esimerkkikoodi 21. SDL_Init- ja SDL_Quit-kutsujen automatisointi RAII-periaatetta noudattaen. SdlError on C++-standardikirjaston runtime_error-luokasta periytetty luokka, jonka virhekuvaukseen sisältyy SDL_GetError-funktion palauttama merkkijono.

Game Boy -emulaattorin toteutusta varten tärkeimmät SDL-rakenteet ovat SDL_Texture, SDL_Renderer ja SDL_Window: SDL_Texture-rakenteessa säilytetään pelikuvaa tekstuurina, jonka SDL_Renderer-piirturi piirtää SDL_Window-ikkunaan. Lisäksi Game Boyn käyttämien värien generointiin SDL_PixelFormat on hyödyllinen. Sdl-luokan tavoin näille SDL-rakenteille luodaan RAII-periaatetta noudattavat resurssienhallinnan automatisoivat luokat esimerkkikoodin 22 mukaisesti.

```
class Window final {
public:
      Window(
            const char* const title,
            const int x, const int y, const int width, const int height,
            const Uint32 flags
      )
            : handle(SDL_CreateWindow(title, x, y, width, height, flags))
      {
            if (handle == nullptr) {
                  throw errors::SdlError("SDL window creation failed");
            }
      }

      ~Window()
      {
            SDL_DestroyWindow(handle);
      }

      Window(const Window&) = delete;
      Window& operator=(const Window&) = delete;

      SDL_Window* const handle;
};
```

Esimerkkikoodi 22. SDL_Window-rakenteen resurssienhallinnan automatisoiva luokka. Luokat muita SDL-rakenteita varten toteutetaan täsmälleen samalla periaatteella.

8 Pikselinkäsittely-yksikön emulointi

8.1 Pikselinkäsittely-yksikkö

Pikselinkäsittely-yksikkö (PPU, pixel-processing unit) on vastuussa Game Boyn piirtämän grafiikan käsittelystä ja piirtämisestä. Grafiikan piirto Game Boylla on juovapohjaista (scanline-based), eli näytölle piirretään grafiikkaa rivi kerrallaan ylhäältä alas. (18.)

8.2 Tilapohjaisuus

Pikselinkäsittely-yksikkö toimii Game Boylla tilapohjaisesti. Jokaisen piirrettävän juovan aikana käydään läpi kolme tilaa (mode):

- objektiattribuuttikartan tutkinta (OAM search)
- pikselien siirto (pixel transfer)
- h-blank (horizontal blank).

Lisäksi jokaisen piirrettävän ruudun välissä käydään läpi tila v-blank (vertical blank). (14.)

Pikselinkäsittely-yksikön nykyisen tilan määrittävät siirräntärekisterin STAT bitit 0–1 (14). Nykyistä tilaa hallitaan Ppu-rajapintaluokan funktioilla get_mode ja set_mode, jotka esitellään esimerkkikoodissa 23.

```
enum Mode { MODE_HBLANK, MODE_VBLANK, MODE_OAM_SEARCH, MODE_PIXEL_TRANSFER };

Mode Ppu::get_mode() const
{
	return static_cast<Mode>(m_stat & MODE_MASK);
}

void Ppu::set_mode(const Mode mode)
{
	m_stat = static_cast<std::uint8_t>((m_stat & ~MODE_MASK) | mode);
}
```

Esimerkkikoodi 23. Apufunktioita pikselinkäsittely-yksikön nykyisen tilan hallintaan.

Tilojen välisten siirtymien hallintaa varten toteutetaan Ppu-luokkaan funktiot update_mode_state ja handle_mode_change. Update_mode_state-funktio kasvattaa

nykyisessä tilassa kulunutta aikaa mittaavaa m_mode_cycles-jäsenmuuttujaa. Jos kulunut aika ylittää tilan sykliaikarajan, kutsutaan tilasiirtymän toteuttavaa handle_mode_change-funktiota. Funktioita esitellään esimerkkikoodissa 24.

```
void Ppu::update_mode_state(const cpu::Cycles cycles)
{
      m_mode_cycles += cycles;
      const auto mode = get_mode();
      const auto threshold = MODE_CYCLE_THRESHOLDS[mode];
      if (m_mode_cycles >= threshold) {
            // Carry spare cycles over to the next mode.
            m_mode_cycles -= threshold;
            handle_mode_change(mode);
      }
}

void Ppu::handle_mode_change(const Mode previous_mode)
{
      switch (previous_mode) {
      case MODE_HBLANK:
            handle_ly_lyc_comparison();
            render_scanline(m_ly);
            if (++m_ly == VBLANK_START_LINE) {
                  render_frame();
                  set_mode(MODE_VBLANK);
                  m_irq_manager.request_interrupt(INTERRUPT_VBLANK);
            } else {
                  set_mode(MODE_OAM_SEARCH);
            }
            break;
      case MODE_VBLANK:
            handle_ly_lyc_comparison();
            if (++m_ly == VBLANK_END_LINE) {
                  m_ly = 0;
                  set_mode(MODE_OAM_SEARCH);
            }
            break;
      case MODE_OAM_SEARCH:
            set_mode(MODE_PIXEL_TRANSFER);
            break;
      default: // MODE_PIXEL_TRANSFER
            set_mode(MODE_HBLANK);
            break;
      }
}
```

Esimerkkikoodi 24. Pikselinkäsittely-yksikön tilaa hallitsevien funktioiden toteutus.

8.3 Tausta

Game Boy -pelien tausta (BG, background) piirretään käyttäen tile map -tekniikkaa. Tile map -tekniikassa tile mapiksi kutsuttu muistialue sisältää numeroita, jotka kertovat tilen

piirtämiseen käytettävän grafiikan indeksin tile setissä. (16.) Taustan piirtämistä esitellään kuvassa 2.

Kuva 2. Esimerkki taustan piirtämisestä Super Mario Land -pelin alkuvalikossa. Näkymästä puuttuu vielä sienenmuotoinen valitsin start-valinnan vasemmalta puolelta, koska se ei ole osa taustaa vaan objekti.

Game Boyn tilet ovat 8x8-kuvia, joissa on mahdollista käyttää neljää eri harmaan sävyä (mustaa, tummaa harmaata, vaaleaa harmaata ja valkoista). Suorien harmaan sävyjen sijaan kuvien pikselien sävyt kuitenkin tallennetaan paletti-indekseinä. Paletti-indeksin perusteella haetaan sitä vastaava harmaan sävy senhetkisestä paletista. Game Boy -pelin on siis mahdollista muuttaa taustan piirtämiseen käytettävää palettia niin, että käytetyt harmaan sävyt vaihtavat kuvassa paikkaa, mutta kuvan rakenne pysyy samana. (16.)

Taustaa on myös mahdollista vierittää käyttämällä SCX- (scroll X) ja SCY-siirräntärekisteriä (scroll Y). Tätä ominaisuutta käytetään peleissä esimerkiksi pelikentän vierittämiseen pelaajan liikkuessa. (18.)

Taustajuovan piirtämisen toteutus on melko suoraviivainen prosessi: käydään läpi jokainen piste juovalla ja etsitään siinä käytetyn tilen grafiikasta pikselille värisävy. Pikselien värisävyt tallennetaan m_pixels-nimiseen Uint32-taulukkoon.

8.4 Ikkuna

Taustan lisäksi Game Boy -pelit voivat piirtää erillisen ikkunaksi kutsutun alueen. Ikkunan hallintaan käytetään samaa tile map -tekniikkaa kuin taustan hallintaan, mutta ikkunaa ei voi vierittää. Monissa peleissä ikkunaa käytetäänkin esimerkiksi pelaajan elämäpisteiden tai kerättyjen esineiden näyttämiseen, koska niiden ei haluta liikkuvan taustan mukana. Lisäksi ikkunaa voidaan käyttää esimerkiksi ”Pause”-tekstin näyttämiseen kuvan 3 mukaisesti, jotta teksti ei vaikuttaisi itse pelin tilaan. (18.)

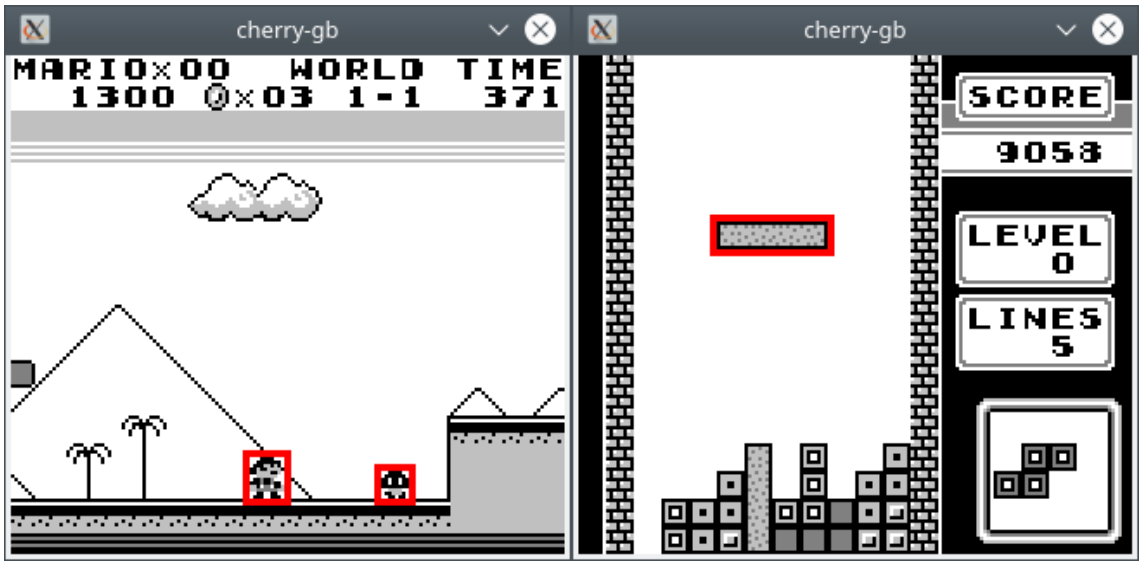
Kuva 3. Ikkunankäyttöesimerkki pelin ollessa pysäytettynä. Ikkuna on rajattu punaisella.

8.5 Objektit

Game Boyn terminologiassa objekti (OBJ, object) vastaa yleisesti käytettyä termiä sprite. Objekti on siis näytölle piirrettävä kaksiulotteinen kuva. Tarkalleen Game Boyn objekti on nelitavuinen pätkä muistia, joka kertoo

- käytettävän sprite-grafiikan indeksin
- piirrettävän kuvan y- ja x-sijainnin näytöllä
- piirtämiseen vaikuttavat attribuutit.

Attribuuteilla määritellään objektin paletti, piirtoprioriteetti suhteessa taustaan sekä X- ja Y-piirtosuunta. (18.) Objektien käyttöä peleissä havainnollistetaan kuvassa 4.



Kuva 4. Esimerkkejä objektien käytöstä peleissä Super Mario Land ja Tetris. Objektit ovat rajattu punaisella.

Objektit sijaitsevat Game Boyn objektiattribuuttikartassa (OAM, object attribute map) eli muistialueella 0xFE00–0xFE9F, jonne mahtuu yhteensä 40 objektia. Koska Game Boy kykenee piirtämään yhdelle juovalle kuitenkin korkeintaan kymmenen objektia, juovalle asettuvia objekteja saatetaan yksinkertaisesti jättää piirtämättä. Tämä näkyy peleissä tilanteina, joissa piirretyistä objekteista puuttuu rivejä.

Objektijuovan piirtäminen on hieman tausta- ja ikkunajuovan piirtämistä monimutkaisempi prosessi. Ensin juovalle asettuvat objektit täytyy lukea objektiattribuuttikartasta. Sen jälkeen objektit järjestetään siten, että X-sijainniltaan pienin tullaan piirtämään juovalle päällimmäisenä (viimeisenä). Lopuksi objektit piirretään juovalle lukemalla objektien pikselien värisävyjä käytettävien sprite-grafiikkaindeksien perusteella. (16.)

8.6 Lopullinen ruudun piirtäminen

Luvuissa 8.3, 8.4 ja 8.5 käsiteltiin juovien piirtämistä. Kun alin juova on piirretty, täytyy kaikkien juovien sisältö kuitenkin vielä piirtää ikkunaan. Tämä saadaan aikaan

SDL_UpdateTexture-, SDL_RenderCopy- ja SDL_RenderPresent-funktioilla. Ensin SDL_UpdateTexture-funktiolla päivitetään tekstuuri vastaamaan m_pixels-taulukon sisältöä. Sen jälkeen tekstuuri kopioidaan SDL_Renderer-piirturille. Lopuksi kuva vielä piirretään ikkunaan SDL_RenderPresent-funktiokutsulla. Prosessia havainnollistetaan esimerkkikoodissa 25.

```
void Ppu::render_frame()
{
      SDL_UpdateTexture(
            m_texture.handle, nullptr, m_pixels.data(),
            LCD_WIDTH * sizeof(Uint32)
      );
      SDL_RenderCopy(m_renderer.handle, m_texture.handle, nullptr, nullptr);
      SDL_RenderPresent(m_renderer.handle);
      frame_rendered = true;
}
```

Esimerkkikoodi 25. Lopullisesta ruudun piirtämisestä vastaavan funktion toteutus.

9 Emulaattorin testaus

Emulaattorin toiminnassa on helppo havaita suuret virheet ilman erityistä testausprosessia: väärin toimivan emulaattorin pelikuva näyttää usein rikkonaiselta. Kaikkia toimintavirheitä ei kuitenkaan ole helppo havaita paljain silmin, joten perusteellisempi testaus on tarpeellista.

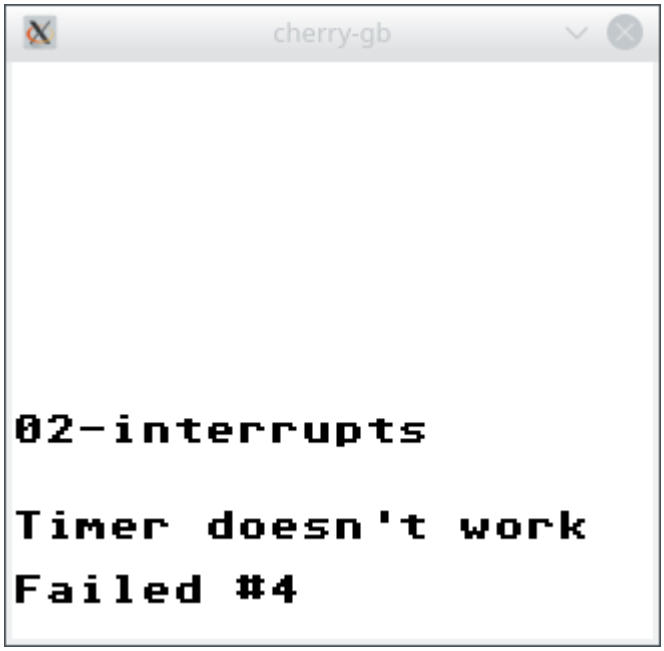
9.1 Testi-ROMit

Emulaattorin testaukseen helpointa on käyttää nk. testi-ROMeja. Testi-ROMit ovat peli-tiedostoja, jotka suorittavat joukon käskyjä ja tarkistavat vastaako järjestelmän tila käskyjen suorituksen jälkeen oletuksia. Jos testi-ROMia ajaneen emulaattorin tila poikkeaa testi-ROMin vahvistetuista oletuksista, emulaattorin voidaan todeta toimivan virheellisesti.

Game Boylle on internetissä jaossa lukuisia testi-ROMeja, joiden toiminta on varmennettu alkuperäisellä Game Boy -konsolilla. Näistä tunnetuimpia ovat Blargg-nimimerkkiä käyttävän henkilön luomat testi-ROMit, joilla voi testata mm. suorittimen käskyjen

toiminnallisuutta ja ajoituksia, muistin luku- ja kirjoitusajoituksia sekä erilaisia yleisesti tunnettuja virheitä.

Blarggin testi-ROMit ilmoittavat testien tulokset sekä piirtämällä ne näytölle että kirjoittamalla ne muistiin sarjaliikennerekisteriin SB. Epäonnistuneista testeistä ilmoitetaan myös testinumero, jotta virhettä emulaattorissa voi alkaa jäljittää tutkimalla testi-ROMin lähdekoodia testinumeron kohdalta. Epäonnistuneen testin aikaansaamaa näkymää esitellään kuvassa 5.



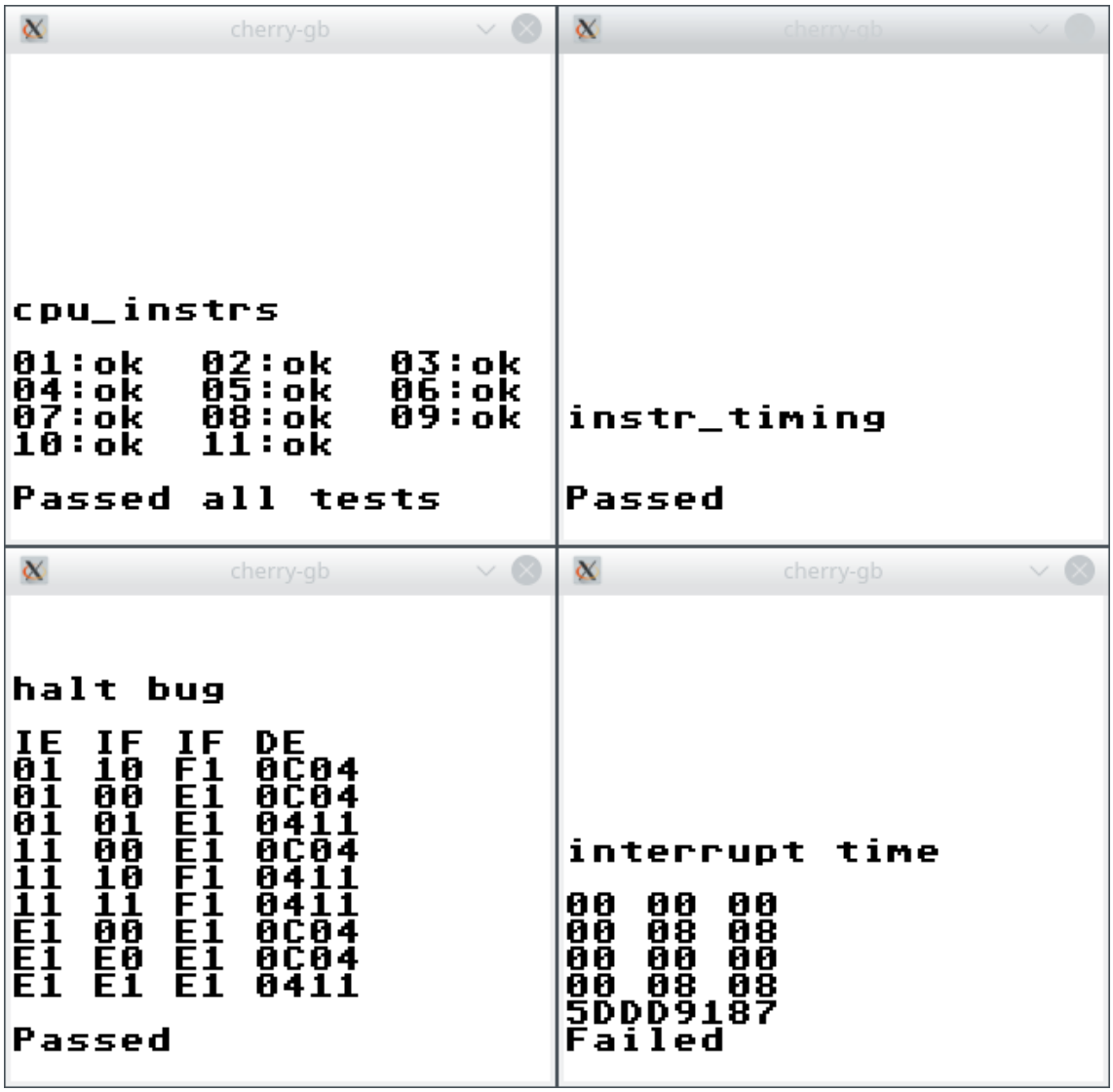
Kuva 5. Esimerkki epäonnistuneesta testistä Blarggin testi-ROMia suoritettaessa.

9.2 Testauksen tulokset

Tässä työssä toteutettu emulaattori läpäisee Blarggin testeistä kriittisimmät: käskyjen toiminnallisuutta testaavan cpu_instrs-testin ja käskyjen ajoituksia testaavan instr_timing-testin. Pienten lisäysten jälkeen myös HALT-käskyyn liittyvää virhettä testaava halt_bug-testi menee läpi onnistuneesti.

Täsmällisiä muistin luku- ja kirjoitusajoituksia testaava mem_timing-testi ja keskeytystenkäsittelyn ajoituksia testaava interrupt_time-testi sen sijaan epäonnistuvat.

Epäonnistumiset eivät kuitenkaan vaikuta tavalliseen pelikokemukseen merkittävästi; vain harvinaisissa rajatapauksissa peli saattaa emulaattorilla pelattuna käyttäytyä toisin kuin alkuperäisellä Game Boy -konsolilla. Testaustuloksiin voi siis olla tyytyväinen. Tuloksia esitellään kuvassa 6.



Kuva 6. Kuvankaappauksia testauksen tuloksista.

Testaustulosten lisäksi pelitestauksen perusteella ainakin Tetris, Super Mario Land ja Pokemon Red toimivat emulaattorilla moitteetta pitkällekin pelattuna. Kuva 7 osoittaa Pokemon Red -pelin toimivuutta.

Kuva 7. Kuvankaappauksia Pokemon Red -pelistä toteutetulla emulaattorilla pelattuna.

Muidenkaan samoja muistipankkikontrollerityyppejä käyttävien pelien kanssa ei ilmennyt ongelmia, mutta pelejä ei pelattu erityisen pitkälle.

10 WebAssembly-käännös

Emscripten on työkalu, jolla voi kääntää C- ja C++-ohjelmia selainpohjaisiksi asm.js- ja WebAssembly-versioiksi. Tämä mahdollistaa yleensä perinteistä JavaScriptiä paremman suorituskyvyn, koska sekä asm.js että WebAssembly pyrkivät minimoimaan JavaScriptin hitaiden ominaisuuksien käytön tarkoituksenaan saavuttaa lähes natiivi suoritusnopeus. Tässä työssä Emscriptenin käytöllä ei kuitenkaan pyritä tavoittelemaan suorituskykyetuja, vaan ainoastaan mahdollistamaan toteutetun emulaattorin kääntäminen selainpohjaiseksi. (22.)

Verkkosovellusten ja perinteisten ohjelmatiedostojen piirteiden eroavaisuuksien takia WebAssembly-käännös vaatii hieman muutoksia emulaattorin ydinsilmukkaan; perinteisen ohjelmatiedoston (lähes) ikuinen silmukka estäisi selaimen muun toiminnan. Ikuisen silmukan sijaan täytyy käyttää rakennetta, jossa silmukan sisältöä suoritetaan vain tiettyinä ajanhetkinä. Tähän soveltuu emscripten.h-tiedostossa esitelty funktio emscripten_set_main_loop, jolle annetaan parametrinä kutsuttava funktio sekä funktiokutsujen tiheyteen liittyviä kokonaislukuarvoja. Fps-parametriksi kannattaa antaa arvo

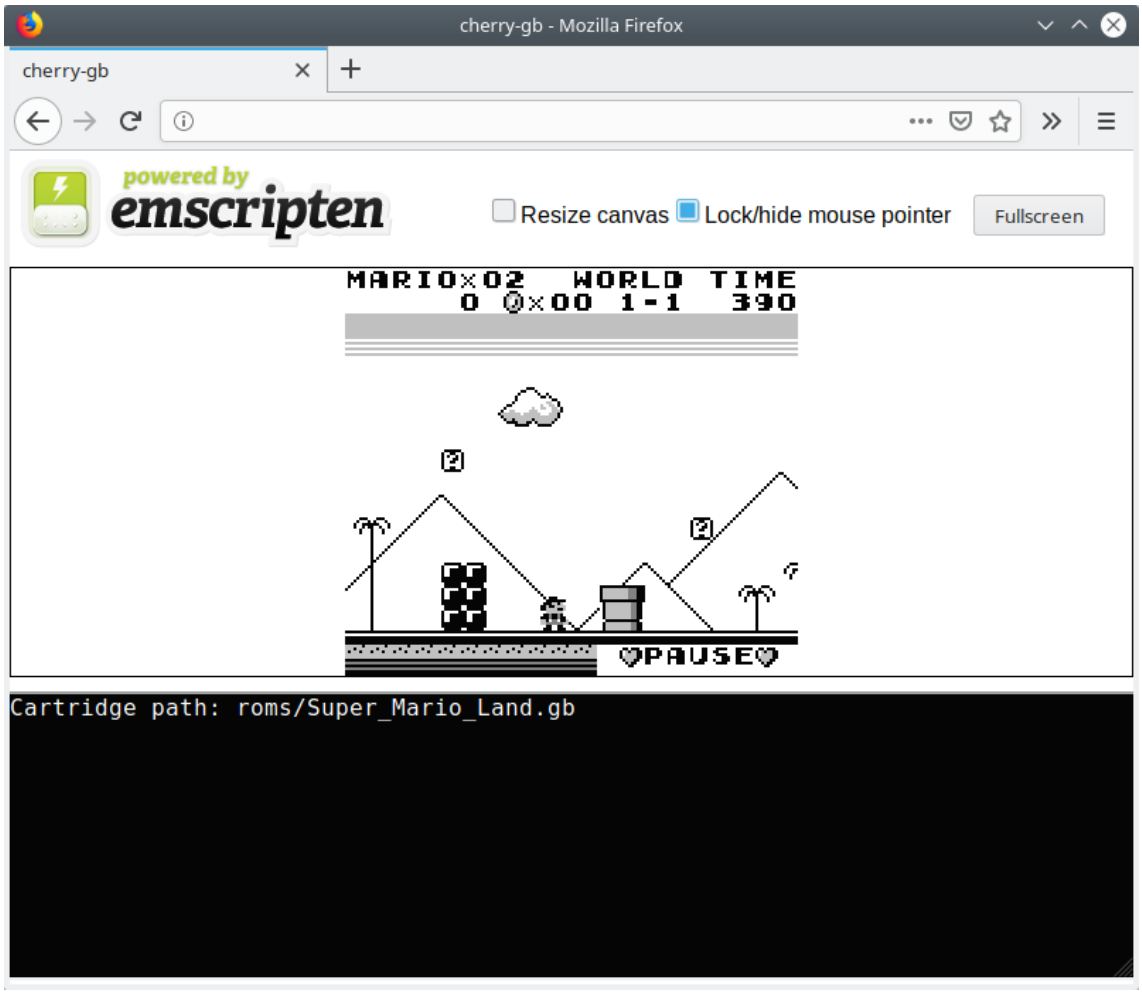
-1, jotta käännös käyttäisi JavaScriptin requestAnimationFrame-funktiota epätarkan setTimeout-funktion sijaan. (23.) Funktion emscripten_set_main_loop käyttöä esitellään esimerkkikoodissa 26.

```
#ifdef __EMSCRIPTEN__ // WebAssembly
	emscripten_set_main_loop(
		process_single_frame,
		-1, // Use requestAnimationFrame instead of setTimeout.
		1 // Call infinitely.
	);
#else // Executable
	for (;;) {
		process_single_frame();
	}
#endif
```

Esimerkkikoodi 26. Toiminnan mukauttaminen selainpohjaista WebAssembly-versiota varten. WebAssembly-käännös erotellaan ohjelmatiedostokäännöksestä käyttämällä apuna esikääntäjän määrittelyjä.

Esimerkkikoodissa 26 esitelty ratkaisu ei ole täydellinen, koska se ei huomioi aikaeroa requestAnimationFrame-tekniikan ja ohjelmatiedoston suorituksen välillä. Tämän työn tavoitteiden kannalta tarkkuus on kuitenkin riittävä, koska requestAnimationFrame-tekniikalla saavutetaan yleensä 60 hertsin kuvataajuus (frame rate) ja Game Boyn kuvataajuus on 59,7 hertsiä.

Lähdekoodimukautusten jälkeen emulaattori voidaan kääntää em++-kääntäjällä WebAssemblyksi. Kun output-tiedostoksi määritellään ”-o”-optiolla HTML-tiedosto, saadaan tulokseksi kuvan 8 mukainen selainpohjainen käännös emulaattorista.



Kuva 8. Emulaattorista käännetty WebAssembly-versio.

11 Yhteenveto

Työssä toteutettiin Game Boy -emulaattori, joka täyttää sille asetetut vaatimukset. Emulaattorilla on mahdollista pelata monia Game Boy -pelejä ongelmitta, ja siitä käännettiin perinteisen ohjelmatiedoston lisäksi myös selainpohjainen WebAssembly-versio. Emulaattorin toteutuksen aikana törmättiin moniin pulmiin, joihin kehitettiin selkeitä ratkaisuja: pelitiedoston käsittely käyttöjärjestelmärajapintoja käyttäen, komponenttien mallinnus luokkina sekä 8-bittisten rekisterien paritus Pair-luokkaa käyttäen. Emulaattorin toimintaa myös varmennettiin testaamalla.

Tulevaisuudessa emulaattoria on kuitenkin mahdollista parantaa. Emulaattoriin ei ole toteutettu äänentoistoa, kaikkia muistipankkikontrollerityyppejä tai kahden Game Boyn välistä sarjaliikennekommunikointia, ja saavutetuissa testaustuloksissakin on parantamisen varaa. Myös taaksepäin yhteensopivan Game Boy Color -konsolin väriominaisuuksien lisääminen on varteenotettava jatkokehitysvaihtoehto. Työssä kartutettua tietotaitoa käyttäen jonkin monimutkaisemmankin alustan emulaattorin toteutus on mahdollista.

Lähteet

1 Game Boy. 2019. Verkkoaineisto. Wikipedia. <https://en.wikipedia.org/wiki/Game_Boy>. Luettu 18.3.2019.

2 Game Boy. 2014. Verkkoaineisto. Wikipedia. <https://upload.wikimedia.org/wikipedia/commons/f/f4/Game-Boy-FL.jpg>. Ladattu 8.4.2019.

3 Emulator. 2019. Verkkoaineisto. Wikipedia. <https://en.wikipedia.org/wiki/Emulator>. Luettu 18.3.2019.

4 Emulation - What exactly is a cycle-accurate emulator? 2018. Verkkoaineisto. Retrocomputing Stack Exchange. <https://retrocomputing.stackexchange.com/questions/1191/what-exactly-is-a-cycle-accurate-emulator>. Luettu 18.3.2019.

5 Std::filesystem::file_size. 2018. Verkkoaineisto. Cppreference.com. <https://en.cppreference.com/w/cpp/filesystem/file_size>. Luettu 29.3.2019.

6 Open(3): open file. Verkkoaineisto. Linux man page. <https://linux.die.net/man/3/open>. Luettu 19.3.2019.

7 Close(2): close file. Verkkoaineisto. Linux man page. <https://linux.die.net/man/2/close>. Luettu 19.3.2019.

8 RAII. 2017. Verkkoaineisto. Cppreference.com. <https://en.cppreference.com/w/cpp/language/raii>. Luettu 19.3.2019.

9 CreateFileA function (fileapi.h). 2018. Verkkoaineisto. Microsoft. <https://docs.microsoft.com/en-us/windows/desktop/api/fileapi/nf-fileapi-createfilea>. Luettu 22.3.2019.

10 Fstat(2): file status. Verkkoaineisto. Linux man page. <https://linux.die.net/man/2/fstat>. Luettu 19.3.2019.

11 GetFileSizeEx function (fileapi.h). 2018. Verkkoaineisto. Microsoft. <https://docs.microsoft.com/en-us/windows/desktop/api/fileapi/nf-fileapi-getfilesizeex>. Luettu 22.3.2019.

12 Managing Memory-Mapped Files. 2010. Verkkoaineisto. Microsoft. <https://docs.microsoft.com/en-us/previous-versions/ms810613(v=msdn.10)>. Luettu 22.3.2019.

13 Mmap(3): map pages of memory. Verkkoaineisto. Linux man page. <https://linux.die.net/man/3/mmap>. Luettu 19.3.2019.

14 Díaz, Antonio Niño. 2015. The Cycle-Accurate Game Boy Docs. Verkkoaineisto. <https://github.com/AntonioND/giibiiadvance/blob/master/docs/TCAGBD.pdf>. Luettu 31.3.2019.

15 Std::memcpy. 2019. Verkkoaineisto. Cppreference.com. <https://en.cppreference.com/w/cpp/string/byte/memcpy>. Luettu 21.3.2019.

16 Korth, Martin. 2001. Pan Docs. Verkkoaineisto. <https://problemkaputt.de/pandocs.htm>. Luettu 1.4.2019.

17 Processor register. 2019. Verkkoaineisto. Wikipedia. <https://en.wikipedia.org/wiki/Processor_register>. Luettu 27.3.2019.

18 The Ultimate Game Boy Talk (33c3). 2016. Verkkoaineisto. YouTube. <https://youtube.com/watch?v=HyzD8pNlpwI>. Katsottu 4.4.2019.

19 Game Boy CPU isn't a Z80. What is it? 2019. Verkkoaineisto. Nesdev. <https://forums.nesdev.com/viewtopic.php?t=18335&p=232645>. Luettu 29.3.2019.

20 Gameboy (LR35902) OPCODES. Verkkoaineisto. Pastraiser. <http://www.pastraiser.com/cpu/gameboy/gameboy_opcodes.html>. Luettu 24.3.2019.

21 Homepage. Verkkoaineisto. Simple DirectMedia Layer. <https://www.libsdl.org>. Luettu 29.3.2019.

22 Main – Emscripten 1.38.27 documentation. Verkkoaineisto. Emscripten. <https://emscripten.org>. Luettu 2.4.2019.

23 Emscripten Runtime Environment – Emscripten 1.38.27 documentation. Verkkoaineisto. Emscripten. <https://emscripten.org/docs/porting/emscripten-runtime-environment.html>. Luettu 2.4.2019.