МАТЕМАТИЧКА ГИМНАЗИЈА

МАТУРСКИ РАД

из предмета
Програмирање и програмски језици

на тему

Емулација

уз додатни пројекат

израда емулатора за GameBoy конзолу

Ученик
Петар Величковић, IV$_c$

Ментор
проф. Мијодраг Ђуришић

Београд, јун 2012.

С А Д Р Ж А Ј

Прва глава

УВОД

Сваки рачунарски систем се састоји од хардвера и софтвера. Када ова два дела раде у спрези, омогућавају нам велики спектар могућности, као што је израда докумената, претраживање Интернета, или играње игрица. Али, ова спрега је толико изражена да изазива значајну зависност ових компоненти, што значи да, уколико се једна компонента значајно промени, то може смањити функционалност друге (или је потпуно онемогућити). Како знамо да је хардвер у сталној еволуцији, постоје многобројни системи који су запостављени у корист новијих, док се на новим не може извршавати софтвер са старих (тај софтвер је *некомпатибилан* са новим системом). За ово постоје бројни примери: стари оперативни системи, игрице за конзоле које се више не производе, итд. Такође се сусрећемо и са обратним проблемом: како правимо све захтевнији софтвер, он ће захтевати више ресурса, ефикасније и брже извршавање операција... ово као последицу има брже застаревање компоненти хардвера, као што су процесор, меморије, и сл.

Емулација је један од главних метода којим спречавамо да старији софтвер, односно функционалности старијег хардвера не нестану у потпуности, тј. омогућавамо да се могу користити и на модернијим системима. У складу са тим ћемо најпре дефинисати овај појам.

1.1. Појам емулације. Поређење са симулацијом

У рачунарству, **емулацијом** (енгл. *emulation* – изворно значење: имитација) се назива поступак којим *имитирамо* функције једног рачунарског система на другом, за то користећи хардвер или софтвер (или обоје).

Хардверски емулатори имају облик картица, чипова и сл., и њиховим уграђивањем у неки систем им се додају неке могућности другог система. На пример, у ову категорију би спадао уређај који би некој играчкој конзоли дао могућност да чита своје игрице са другачијег медија од оног за који је дизајнирана; рецимо, USB читање за конзолу која иначе чита искључиво са дискова.

Софтверски емулатори су испрограмирани на машини на којој се врши емулација, користећи неки програмски језик као што је Assembler, C, C++, и сл. Као пример софтверски направљеног емулатора може послужити емулатор конзоле за видео игре, емулатор неког периферног уређаја (нпр. штампача) којим се осигурава да ће га већи распон рачунарских система препознати, или емулатор оперативног система којим можемо виртуелно покренути нпр. Windows 95 унутар Windows 7. Наравно, могуће је користити емулатор унутар емулатора. На пример, на Mac OS X платформи смо емулирали Windows, а унутар емулираног Windows-а смо емулирали конзолу PlayStation. Овим постижемо исти резултат као да смо на Windows платформи емулирали PlayStation (иако ће брзина емулације бити знатно већа у другом случају).

Изузетно је битно разликовати појам емулације од појма *симулације*. Ова два појма имају додирних тачки али се суштински разликују. Док емулација подразумева да дизајнер/програмер *имитира тражени уређај* и све његове особине, симулација подразумева само *опонашање функционалности*. Ради илустрације, емулатор аркадне игрице Pac-man је програм који имитира хардвер тог аркадног уређаја и чита оригинални ROM ове игрице, док би симулатор била игрица који само графички личи на аркадни уређај, али је програмирана конкретно за платформу на којој се налази.

У овом раду ћемо се претежно бавити софтверским емулаторима, зато што израда хардверских емулатора захтева знања која знатно превазилазе оквире предмета.

1.2. Примене емулације. Предности и мане

Поставља се питање: *које све уређаје можемо емулирати*? Теоријски одговор (који следи из Черч-Тјурингове тезе, која ће бити обрађена у следећој глави) је да можемо емулирати сваки рачунарски систем, дакле било који уређај који има неку врсту микропроцесора у себи. Ово подразумева рачунаре, калкулаторе, играчке конзоле, аркадне машине, мобилне телефоне, и тако даље.

Ово нам наравно даје најочигледнију теоријску примену емулације: ако бисмо ефикасно емулирали мноштво уређаја на једном, тиме бисмо повећавали број намена тог уређаја, и самим тим смањили или елиминисали потребу за уређајима које смо емулирали.

Емулација, заједно са миграцијом, чини једну од две могуће стратегије којима се спречава потпуно ишчезавање старијег хардвера или софтвера и података које су они направили, тзв. *дигитална презервација*. Миграција подразумева да се подаци направљени у старијој апликацији прилагоде новој апликацији, док се код емулације окружење старије апликације потпуно преноси на новију платформу. Миграција, иако је много лакша и захтева мање времена, резултује могућим губитком функционалности над оригиналним подацима, и мора се радити за сваку нову промену система; када корисници желе да се задржи изглед и све могућности старог окружења, емулатор је права опција.

Међутим, не можемо тако једноставно искористити пун теоријски потенцијал ове методе. Емулатори су захтевни за прављење, траже велику прецизност и опрез на ситне детаље. Чест проблем је и чињеница да оригинални произвођачи не документују све функционалности њиховог система, па се оне морају наћи поступком обрнутог инжењеринга. Такође, у зависности од сложености система која се емулира, емулација може бити спора. Већина модернијих система за који постоји емулатор углавном не може да се ефикасно емулира на данашњим рачунарима. На пример, емулација конзоле PlayStation 2 (изашла 2000. године) је омогућена појавом емулатора PCSX2, али, 12 година касније, чак и на јачим рачунарима се значајни удео игрица успорено извршава, а емулатор за PlayStation 3 (изашла 2006. године) је тек у планирању – тим који стоји иза PCSX2 предвиђа да ће требати најмање још 10 година да би се направио успешан емулатор за ову конзолу.

Још један аспект емулације који није претерано разјашњен јесте да ли је то легална активност. Емулатори се могу бесплатно скидати са Интернета, чиме се производ који има цену (чак иако се више не производи) практично нуди бесплатно. Иако је ефикасност емулатора често далеко од ефикасности самог уређаја, постоје основани разлози да се ово сматра неком врстом крађе интелектуалне својине, односно кршења ауторских права.

Ипак, емулација има доста предности. Мада је израда емулатора захтеван посао, на дужем временском периоду може да се покаже као исплативија, нарочито у већим организацијама – зато што ће један емулатор радити по истом принципу на свим модерним платформама, док би се миграција морала вршити на свакој платформи одвојено. Емулација такође има и велики потенцијал за напредак, зато што се највећи број емулатора ставља на Интернет, са доступном свом документацијом (open source). Ово отвара врата већим тимским пројектима на овом пољу, а уређаји данашњице су толико сложени да захтевају тимски рад да би се успешно емулирали.

Друга глава

ЕМУЛАЦИЈА

У првој глави смо се претежно концентрисали на обраду појма емулације користећи свакодневне примере и примене ове методе. Сада ћемо прићи овом појму са чисто теоријске стране, најпре се осврћући на тезу на којој се овај метод и заснива.

2.1. Черч-Тјурингова теза и њене последице

У теорији рачунарства, **Черч-Тјурингова теза** је комбинована хипотеза о природи функција чије вредности су ефективно израчунљиве (или у модернијем изразу, алгоритамски израчунљиве).

Задржимо се најпре на појму ефективно израчунљиве функције. Функцију зовемо **ефективно израчунљивом** уколико за њено израчунавање постоји ефективна метода (алгоритам). Да би метода била ефективна, мора да поштује следеће услове:

- Мора за сваки улаз дати неки излаз;
- Мора увек дати тачан излаз;
- Мора увек бити израчунљива у коначном броју корака;
- Мора радити за сваку инстанцу проблема.

Теза тврди да се било које ефективно израчунавање на неком уређају може превести у еквивалентно на уређају који се зове **Тјурингова машина**. Такође је било коју ефективно израчунљиву функцију могуће изразити преко рекурзивно задате функције. До ових закључака су различитим приступима дошли Алонзо Черч и Алан Тјуринг. Иако су подељени ставови о томе да ли се ова хипотеза може доказати, практично се универзално сматра тачном.

Од каквог је значаја ова теза за емулацију? Осврнимо се прво на једну од њених последица: Сваки програмски језик опште намене се може искористити за имплементацију било којег алгоритма. Уопштено, све што је ефективно израчунљиво се може ефективно израчунати сваким програмским језиком опште намене (у случају емулације неког хардвера, посматрамо израчунавања која обавља хардвер, и слично).

Разлог због којег можемо емулирати велики број рачунарских система је то што су они већински засновани на фон Најмановој архитектури. Ова архитектура, постављена 1945. године, подразумева рачунарски систем који има меморију насумичног приступа (RAM), на којој може било где писати било који процес; централну процесорску јединицу (CPU) са сопственим регистрима и имплементираним скупом инструкција, као и контролну јединицу. Све додатне компоненте као што су графика, звук и периферни уређаји су само проширење ове архитектуре. Она дефинише основну "узми па изврши" петљу неког хардвера помоћу које можемо у псеудокоду написати главну петљу у већини емулатора:

```
while <извршавамо инструкције>
{
  узми следећу инструкцију из меморије
  изврши узету инструкцију
  провери да ли има прекида
}
```

Почевши од ове основне петље, детаљном имплементацијом сваке компоненте траженог система стижемо до коначних емулатора.

2.2. Основна структура емулатора. Методи имплементације

Емулатори су углавном подељени у **модуле** (мање или више независне целине) који одговарају подсистемима рачунарског система који емулирамо. Модули са којима се најчешће можемо сусрести су:

- процесорски емулатор;
- емулатор за меморијски подсистем;
- емулатори улазних и излазних уређаја (I/O емулатори).

Магистрале се ретко емулирају, ради веће једноставности или перформанси, самим тим емулирани периферни уређаји готово увек директно комуницирају са емулираним процесором и меморијом.

Емулација процесора често представља најкомпликованији део једног емулатора. Због тога, многи програмери пишу емулаторе уз помоћ "препакованих" емулатора процесора, да би се могли више концентрисати на друге функционалности. Најлакши начин емулације процесора је **интерпретерски**. Интерпретер прати емулирани програмски код и, када наиђе на инструкцију, преко локалног процесора извршава инструкцију која је еквивалентна задатој. Ово је омогућено тиме што се регистрима и флеговима процесора који се емулира додељују променљиве. Ради илустрације интерпретера, можемо проширити петљу коју смо навели у претходном делу:

```
while <извршавамо инструкције>
{
  switch (Citaj(PC++)) //PC – program counter
  {
    case 0x00: //kod instrukcije 00
    case 0x01: //kod instrukcije 01
    …
    case 0xFE: //kod instrukcije FE
    case 0xFF: //kod instrukcije FF
    default: interrupt=true //instrukcija nije definisana skupom instrukcija
  }
  if (interrupt!=false)
  {
    //sta raditi ukoliko dodje do prekida
  }
}
```

Интерпретери су изузетно популарни зато што се много једноставније имплементирају од бржих алтернатива, и раде довољно брзо за емулацију већине старијих система. Међутим, овај метод је непрактичан за емулацију процесора који имају брзину истог реда величине као платформа на којој се емулира. Ова баријера је пробијена развојем техника **рекомпилације**.

Рекомпилација подразумева директно бинарно превођење података из емулираног програма у код који се може извршити на датој платформи. Ова метода изазива велики скок у перформансама, због чега се и користи за емулацију модернијих система. Постоје два начина којима се може извршити рекомпилација, *статички* и *динамички*. **Статичка рекомпилација** подразумева превођење програма пре извршавања, без даљих измена. Ово је брже од динамичке рекомпилације али неизводљиво у великом броју случаја: главни пример су самомодификујући кодови, који могу мењати своју структуру током извршавања; самим тим је немогуће предвидети како ће код изгледати без његовог извршавања. **Динамичка рекомпилација** подразумева превођење блокова инструкција по потреби, за време извршавања програма. Зато што је прилагодљива практично свим врстама кода, ова врста рекомпилације се и користи у највећем броју модерних емулатора.

За **емулацију меморије** такође постоји више приступа. Најочигледнији приступ би третирао меморију као обични низ објеката величине једне емулиране "речи". Овај модел губи ефективност када се деси да локације у рачунарској логичкој (релативној апликацији која је користи) меморији нису изједначене са физичком меморијом. Пример овога су сви системи који захтевају неку врсту напредног управљања меморијом тј. системи који у себи садрже меморијску управљачку јединицу (MMU). Као последицу овога, већина емулираних меморија садрже бар две операције за прецизно читање и писање у логичкој меморији. Ради илустрације, приказаћемо пример са системом који користи првих ROM_SIZE адреса да би сместио ROM, а остатак за смештање RAM-а:

```
void Pisi(word Adresa, word Vrednost)
{
  word AdresaL = Adresa + BazniRegistar; //logicka adresa
  if (AdresaL > ROM_SIZE && AdresaL < Limit) //ne mozemo pisati u ROM
  {
    Memorija[AdresaL] = Vrednost;
  }
  else
  {
    interrupt=true;
  }
}
word Citaj(word Adresa) //odnosi se na isti metod kod primera za procesor
{
  word AdresaL = Adresa + BazniRegistar;
  if (AdresaL < Limit)
  {
    return Memorija[AdresaL];
  }
  else
  {
    interrupt=true;
    return null;
  }
}
```

Што се тиче емулације **периферних уређаја**, постоје два основна приступа. Пошто се магистрале најчешће не емулирају као што смо навели, један од приступа подразумева израду модула за сваки уређај одвојено. Овај метод је кориснији у смислу да се сваки модул може засебно прилагодити карактеристикама уређаја и тако омогућити већу ефикасност. Други метод подразумева израду унификаторске апликације, која би радила са свим улазним и излазним уређајима, и слала њихове информације остатку система (односно, примала информације од остатка система за уређаје). Предност добро испрограмиране овакве апликације је што може пружити подршку за већи број уређаја.

Емулатори периферних уређаја у себи требају садржати основну структуру за контролу прекида, тј. процедуре које ће обавестити процесор да је дошло до прекида, као и процедуре за читање и писање из меморије, користећи сличне методе као и у случају емулације меморије.

Постоји велики број додатних компоненти (односно модула) који се могу емулирати, али смо овде навели основне приступе за ове три најчешће (и најбитније) компоненте. Са добрим познавањем ових основа, као и нивоом вештина у изабраном програмском језику, програмер може кренути у истраживање компоненти конкретног система који жели да емулира, и на крају може започети и саму израду.

У следећој глави ћемо испратити један конкретан такав рад, који укључује проучавање и емулирање једне од најиконичнијих handheld конзола у историји, GameBoy-а.

Трећа глава

АНАЛИЗА И ЕМУЛАЦИЈА GAMEBOY КОНЗОЛЕ

Након савладавања горенаведених метода за имплементацију различитих модула једног емулатора, поседујемо теоријску основу која нам је потребна. Међутим, за програмера рад тек почиње: сваки рачунарски систем се, ако игноришемо основну архитектуру, суштински разликује од другог; и зато, да бисмо успешно направили емулатор било којег уређаја, самим тим и GameBoy-a, неопходно је погледати и тај уређај детаљније.

3.1. Историјат конзоле

GameBoy (на изворном јапанском ゲームボーイ (Gēmu Bōi)) је handheld конзола коју је развила јапанска компанија Нинтендо. Први пут се појавила на тржишту 21. априла 1989. у Јапану, а недуго затим и на територији САД. Излазак ове конзоле се сматра револуцијом на handheld тржишту, пошто је ово прва конзола такве врсте која је остала шире запамћена, упркос многим технолошки супериорнијим приручним конзолама које су излазиле током времена. Успешност ове конзоле је довела до великог броја нових верзија (Game Boy Color, Game Boy Advance, Game Boy Micro) које су кумулативно продале више од 150 милиона примерака широм света. Конзола је додатно добила на популарности од изузетно популарне игре "Тетрис", која је у почетку долазила заједно са конзолом; сматра се да је само кертриџ од ове игре продат у 35 милиона примерака. Наравно, ова конзола располаже и другим чувеним насловима, од којих су многи везани за тада већ установљене серије видео игрица: "Покемон", "Супер Марио", "Зелда"...

Конзолу је осмислио Гунпеи Јокои (横井 軍平), који је поред ове конзоле дизајнирао и систем Game & Watch, и био је једна од главних фигура у развоју неколицине популарних и дуготрајних франшиза видео-игрица (Donkey Kong, Mario Bros...). Многи га сматрају за Томаса Едисона играчког света, и процењује се да би данас индустрија видео игрица била на много вишем нивоу да није трагично погинуо 4. октобра 1997. у саобраћајној несрећи.

3.2. Изглед и техничке спецификације

GameBoy располаже са четири операциона тастера означена са "A", "B", "SELECT" и "START", као и са подлогом са стрелицама. Са десне стране конзоле се налази подешавач јачине звука, а са леве стране сличан прекидач за мењање контраста. На врху се налази прекидач за паљење конзоле као и слот за кертрице. Конзолу обично покрећу четири AA батерије; захтева јачину струје од 150 mA и напон од 6 V. На доњој страни конзоле налази се прикључак од 3.5mm за слушалице. Са десне стране се такође налази конектор за "линк кабл", којим се могу повезати две конзоле, под условом да оба играча играју исту игрицу.

Техничке спецификације овог уређаја су:

- **Процесор**: Модификовани 8-битни Sharp LR35902 фреквенције 4.19 MHz, по функционалностима јако сличан Zilog-овом Z80 процесору;
- **RAM**: 8 kB статичког RAM-а (може се проширити до 32 kB);
- **VRAM:** 8 kB;
- **ROM:** 256-бајтни BIOS. Кертриџи од 256 kb до 8 Mb;
- **Звук:** Два таласна генератора, један PCM 4-битни wave канал за узорке, генератор шумова и један аудио улаз са кертриџа;

- **Дисплеј:** ЛЦД екран резолуције 160x144 пиксела;
- **Кадрова у секунди:** Просечно 59.7 FPS;
- **VBlank интервал:** 1.1ms;
- **Дијагонала:** 2.6 инча;
- **Палета**: Монохроматска са 4 нијансе сиве;
- **Димензије:** 90mm (дужина) x 148mm (висина) x 32mm (дубина).

У ова два поглавља смо направили кратак, али неопходан, увод у историјат и основне могућности ове конзоле. Сада напокон можемо започети са главним делом пројекта, то јест израдом емулатора. Наравно, израду започињемо од најважнијег дела сваког рачунарског система, на који ће после да се надограђује свака следећа компонента – од процесора.

3.3. Процесор

У претходној глави смо већ навели да се за емулацију процесора користе интерпретерске методе и методе рекомпилације. С обзиром да је GameBoy систем који има процесор перформанси неколико редова величине испод данашњих, потпуно је оправдано емулирати га интерпретерски. Дакле, емулацијом процесора GameBoy-а ћемо покушати да имитирамо оно што тај процесор ради у правој конзоли.

Процесор унутар GameBoy-а функционише под истим принципима као и већина других процесора: извршава неки програм тако што покрене неку серију инструкција коју му тај програм зада. Те инструкције се учитавају из меморије, и извршавају једна по једна. Да би се могло пратити где се процесор налази унутар програма, чува се број назван Program Counter (PC). Након што се инструкција учита из меморије, PC се увећа за број бајтова колико сачињава та инструкција.

У претходном делу смо навели да је процесор ове конзоле јако сличан Zilog-овом Z80 процесору. За њега важе следеће ствари:

- Процесор је 8-битни, дакле сва унутрашња израчунавања се обављају бајт по бајт;
- Меморијски интерфејс може адресирати до 65536 бајтова;
- Програмима се приступа кроз исту адресну магистралу као и нормалној меморији;
- Инструкција може бити величине од једног до три бајта.

Осим горенаведеног PC-ја, процесор садржи још седам регистара који могу да се користе за израчунавања: они се означавају са A, B, C, D, E, H и L. Сваки од њих има величину од једног бајта. Процесор такође садржи и показивач на стек (Stack Pointer – SP) који се користи у PUSH и POP инструкцијама (подршка основном LIFO (Last In, First Out) руковању подацима), и регистар за флегове F.

Пошто постоји 256 могућих вредности за први бајт инструкције, то нам даје 256 могућих инструкција у основној табели (тзв. опкодови). Процесор подржава и инструкције које су величине два бајта, а њима приступамо тако што прво задамо опкод који сигнализира процесору да ће у наредна два бајта бити унета инструкција. Ове инструкције се називају и CB-инструкцијама зато што је тај сигнализирајући опкод једнак 0xCB. Свака од ових инструкција се може симулирати у већини програмских језика, тако што манипулишемо интерним моделом процесора са горенаведеним регистрима. Непрактично је пренети целу табелу опкодова у овај рад, зато читаоцима у секцији са литературом остављамо линк на којем се може наћи комплетна табела основних и CB опкодова: [8].

Можемо закључити да основни модел процесорског интерпретера мора да садржи:

- Структуру која задржава стање свих регистара и симулира процесорски clock;
- Функције које симулирају сваку од инструкција;
- Интерфејс који ће да комуницира са меморијом.

У програмском језику C# овај модел можемо представити као класу, и регистре можемо представити на следећи начин:

```
public class X80
{
  private int A, B, C, D, E, H, L, PC, SP; //registri
  private bool FZ, FC, FH, FN; //flagovi
  public int ticks; //procesorsko vreme
}
```

Регистар са флеговима (F) је битан за функционисање процесора: аутоматски поставља одређене битове (флегове) у зависности од резултата претходно извршене операције. Постоје четири флега у овом процесору:

- ZF (Zero Flag): поставља се уколико је претходна операција дала нулу као резултат;
- OF (Operation Flag): поставља се уколико је претходна операција била одузимање;
- HCF (Half-carry Flag): поставља се уколико је током претходне операције доња половина бајта имала overflow преко 15;
- CF (Carry Flag): поставља се уколико је током претходне операције добијен резултат преко 255 (при сабирању) или испод 0 (при одузимању).

Сада можемо дати примере функција које симулирају неке од инструкција унутар овог процесора:

```
private void Add(int b) //dodavanje broja b registru A
{
  FH = (A & 0x0F) + (b & 0x0F) > 0x0F; //setovanje halfcarry flag-a
  A += b;
  FC = A > 255; //setovanje carry flag-a
  A &= 0xFF;
  FN = false; //operation flag ne treba biti setovan
  FZ = A == 0; //setovanje zero flag-a
  ticks += 4; //operacija zahteva 4 procesorska ticka
}

private void Compare(int b) //poredjenje broja b sa registrom A
{
  //Compare se unutar procesora vrši oduzimanjem broja b od registra A,
  //tako da moramo pravilno setovati flagove
  FH = (A & 0x0F) < (b & 0x0F); //setovanje halfcarry flag-a
  FC = b > A; //ako je b veći od A, dobijamo overflow
  FN = true; //izvršili smo oduzimanje
  FZ = A == b; //ako je A=b, dobili smo nulu kao rezultat
  ticks += 4; //operacija zahteva 4 procesorska ticka
}

private void NoOperation() //nije zadata nikakva instrukcija
{
  ticks += 4;
}
```

Имплементацијом ових и осталих операција добијамо модел процесора који лако може да манипулише сопственим регистрима. Тиме смо обавили већи део посла, међутим, процесор мора бити способан да запамти своје резултате у меморију да би био користан. На сличан начин, при емулацији процесора неопходно је имати интерфејс који ће комуницирати са емулираном меморијом. GameBoy нема компликован комуникациони интерфејс, тако да и емулирани интерфејс може бити једноставан.

У овом тренутку је сасвим довољно да процесор зна да интерфејс постоји; детаљи о томе како се манипулише меморијом нису последица процесорских функција. Процесор ће захтевати четири операције:

```
public int ReadByte(int address)
{
  //čita bajt sa date adrese
}

public int ReadWord(int address)
{
  //čita dva bajta sa date adrese
}

public void WriteByte(int address, int value)
{
  //piše bajt na datu adresu
}

public void WriteWord(int address, int value)
{
  //piše dva bajta na datu adresu
}
```

Након дефинисања ових операција, можемо симулирати и операције процесора које захтевају приступ меморији. Наведимо неколико примера:

```
private void Push(int rh, int rl) //push-ovati brojeve rh i rl na stek
{
  WriteByte(--SP, rh); //umanjivanje stack pointera i pushovanje rh
  WriteByte(--SP, rl); //analogno sa rl
  ticks += 11; //operacija zahteva 11 procesorskih tickova
}

private void Pop(ref int rh, ref int rl) //pop-ovati dva broja sa steka
{
  rl = ReadByte(SP++); //pop-ujemo rl i uvećavamo stack pointer
  rh = ReadByte(SP++); //analogno sa rh
  ticks += 10; //operacija zahteva 10 procesorskih tickova
}

private void LoadFromImmediateAddress(ref int r)
{
  //učitaj vrednost na trenutnoj lokaciji PC-a u registar r
  r = ReadByte(ReadWord(PC));
  PC += 2; //uvećavamo PC
  ticks += 13; //operacija zahteva 13 procesorskih tickova
}
```

Сада је за основни модел процесора потребно још само да дефинишемо основну "узми па изврши" петљу која је у срцу сваког модерног процесора. У оквиру ове петље узимамо текућу инструкцију, одређујемо коју функцију треба позвати, и позивамо ту функцију.

```
public void Step()
{
  PC &= 0xFFFF; //sprečavamo da PC prekorači vrednost 0xFFFF
  int opCode = ReadByte(PC++); //čitamo tekuću instrukciju
  switch (opCode)
  {
    case 0x00: //NOP
      NoOperation();
      break;
    case 0x01: //LD BC, NN
      LoadImmediate(ref B, ref C);
      break;
    case 0x02: //LD (BC), A
      WriteByte(B, C, A);
      break;
    . . .
    case 0xFE: //CP N
      CompareImmediate();
      break;
    case 0xFF: //RST 38H
      Restart(0x0038);
      break;
    default:
      throw new Exception("Unknown instruction");
  }
}
```

Овим смо завршили емулацију процесора. Наравно, иако је ово најважнији део система, процесор је сам по себи бескористан без осталих делова; од којих је дефинитивно најбитнија меморија. У следећем поглављу ћемо се бавити емулацијом меморије GameBoy-а.

3.4. Меморија

У готово сваком рачунарском систему, меморија са којом тај систем манипулише није једноставан и континуалан низ података, и GameBoy није никакав изузетак по овом питању. Како смо навели у претходном поглављу, процесор GameBoy-а има приступ над 65536 различитих локација на адресној магистрали. Сходно томе, можемо дефинисати "меморијску мапу" која дефинише све регионе којима процесор може приступити:

- **[0x0000 – 0x3FFF] ROM кертриџа, нулта банка:** Првих 16384 бајтова програма садржаног у кертриџу су увек доступни у овом региону. Садржи два битна подрегиона:
  - **[0x0000 – 0x00FF] BIOS:** Када се процесор покрене, PC започиње са адресе 0x0000, која означава почетак 256-бајтног GameBoy BIOS-а. Након што се BIOS покрене, уклања се из меморијске мапе, и овај простор постаје доступан за адресирање.
  - **[0x0100 – 0x014F] Заглавље кертриџа:** Овај део кертриџа садржи податке о називу и произвођачу, и мора бити записан у специфичном формату.
- **[0x4000 – 0x7FFF] ROM кертриџа, остале банке:** Свака од наредних "банака" величине 16000 бајтова од програма садржаног у кертриџу може да се омогући доступном процесору преко овог региона.
- **[0x8000 – 0x9FFF] Графички RAM (VRAM):** Овде се чувају подаци везани за позадине и спрајтове које користи графички подсистем.

- **[0xA000 – 0xBFFF] (Екстерни) RAM кертрица:** Уколико игрица захтева више RAM-а него што јој то може пружити хардвер, додатних 8000 бајтова је доступно за адресирање у овом региону.
- **[0xC000 – 0xDFFF] Радни RAM (WRAM):** GameBoy-евих интерних 8000 бајтова RAM-а, над којим процесор може слободно манипулисати.
- **[0xE000 – 0xFDFF] Радни RAM (Shadow):** Као последица склопа хардвера унутар GameBoy-а, идентична копија радног RAM-а је доступна 8000 бајтова више у меморијској мапи. Ова копија је доступна све до последњих 512 бајтова мапе, где се појављују други региони.
- **[0xFE00 – 0xFEFF] Object Attribute Memory (OAM):** Подаци о спрајтовима које је приказао графички подсистем се чувају овде, укључујући њихове позиције и својства.
- **[0xFF00 – 0xFF7F] Улазно-излазни (I/O) регион:** Сваки од подсистема у GameBoy-у садржи контролне вредности које дозвољавају програмима да праве ефекте и користе хардвер. Ове вредности су директно доступне процесору преко овог региона.
- **[0xFF80 – 0xFFFE] Нулти RAM (ZRAM):** Регион RAM-а велике брзине и величине 128 бајтова је доступан на крају меморије. Назива се нултим RAM-ом зато што се већина интеракције између програма и хардвера одвија преко овог региона.

Да би емулирани процесор могао да приступи сваком од ових региона одвојено, сваки се мора посматрати као специјални случај унутар функција за читање и писање које смо декларисали у претходном поглављу. Довољно је користити *if-else* ланац или *switch* наредбу. Наводимо једну од могућих имплементација:

```
public interface ICartridge //interfejs za rad sa kertridžom
{
  int ReadByte(int address);
  void WriteByte(int address, int value);
}

public ICartridge cartridge; //pristup memoriji kertridža
private byte[] highRam = new byte[256]; //ZRAM
private byte[] videoRam = new byte[8 * 1024]; //VRAM
private byte[] workRam = new byte[8 * 1024]; //WRAM
public byte[] oam = new byte[256]; //informacije o sprajtovima

public int ReadByte(int address)
{
  if (address <= 0x7FFF || address >= 0xA000 && address <= 0xBFFF))
  {
    //memorija kertridža
    return cartridge.ReadByte(address);
  }
  else if (address >= 0x8000 && address <= 0x9FFF)
  {
    //VRAM
    return videoRam[address - 0x8000];
  }
  else if (address >= 0xC000 && address <= 0xDFFF)
  {
    //WRAM
    return workRam[address - 0xC000];
  }
```

```
  else if (address >= 0xE000 && address <= 0xFDFF)
  {
    //WRAM (Shadow)
    return workRam[address - 0xE000];
  }
  else if (address >= 0xFE00 && address <= 0xFEFF)
  {
    //informacije o sprajtovima
    return oam[address - 0xFE00];
  }
  else if (address >= 0xFF80 && address <= 0xFFFE)
  {
    //ZRAM
    return highRam[0xFF & address];
  }
  else
  {
    //I/O region, trenutno nepokriven
  }
}

public int ReadWord(int address)
{
  int low = ReadByte(address); //donji bajt
  int high = ReadByte(address + 1); //gornji bajt
  return (high << 8) | low;
}
```

У горњој имплементацији смо за сада изоставили I/O регион, зато што манипулација тим регионом захтева мало ширу причу и зато ће му бити посвећено више пажње у даљем делу рада. Тада ћемо проширити овај део ReadByte функције.

Функције WriteByte и WriteWord се декларишу аналогно као и функције читања, осим што се нека вредност записује на тражену локацију уместо да се чита са ње (с тим да, наравно, морамо пазити да не омогућимо записивање у ROM делове).

Овим смо завршили обраду основних концепата везаних за манипулацију меморијом. Заједно са емулираним процесором, већ можемо постићи неку врсту емулације: програм можемо учитати у меморију, изрвршити га, и коректно ажурирати регистре у складу са инструкцијама које тај програм задаје. Оно што недостаје је како повезати све то са графичким излазом. У наредном поглављу ћемо се бавити овим проблемом: како изгледа структура графичког излаза код GameBoy-а, и како се тај излаз приказује на екрану.

3.5. Графичка процесорска јединица (GPU)

Са комплетираним структурама процесора и меморије, може се приступити емулацији периферних уређаја. Један од главних периферних уређаја GameBoy-а (као и већине других играчких система) је управо графички процесор: то је примарни излаз за ову конзолу, и већина рада централног процесора "отпада" на генерисање графике за њега. Због тога ћемо овом систему посветити посебну пажњу.

Нинтендов интерни назив за GameBoy је "Dot Matrix Game" ("игрица матрице тачкица"), зато што је његов екран пикселизовани ЛЦД димензија 160x144 пиксела. Ако сваки пиксел ЛЦД-а третирамо као пиксел унутар објекта класе Bitmap, можемо директно мапирати изглед екрана у сваком фрејму. За то можемо користити низ у коме ћемо чувати боју сваког пиксела засебно. Наводимо један од начина иницијализације:

```
private Bitmap bitmap; // bitmap
public Graphics graphics; // grafika za bitmap
private uint[] pixels = new uint[160 * 144]; // niz piksela

private void InitGraphics() //inicijalizacija grafike
{
  if (graphics != null) graphics.Dispose();
  graphics = CreateGraphics();
}

private void InitImage() //inicijalizacija slike
{
  InitGraphics();
  for (int i = 0; i < pixels.Length; i++)
  {
    pixels[i] = 0xFF000000; //inicijalizacija na crnu boju
  }
  // uzimanje pokazivaca na niz piksela
  GCHandle handle = GCHandle.Alloc(pixels, GCHandleType.Pinned);
  IntPtr pointer = Marshal.UnsafeAddrOfPinnedArrayElement(pixels, 0);
  // konstruktor za bitmap preko pokazivaca na niz
  bitmap = new Bitmap(160,144,160*4,PixelFormat.Format32bppPargb,pointer);
}

private void RenderFrame() //iscrtavanje frejma
{
  graphics.DrawImage(bitmap, 0, 0, Width, Height);
}
```

Након што смо алоцирали низ пиксела и иницијализовали Bitmap преко показивача на тај низ, за промену боје на позицији (x, y) довољно је изменити вредност pixels[y * 160 + x].

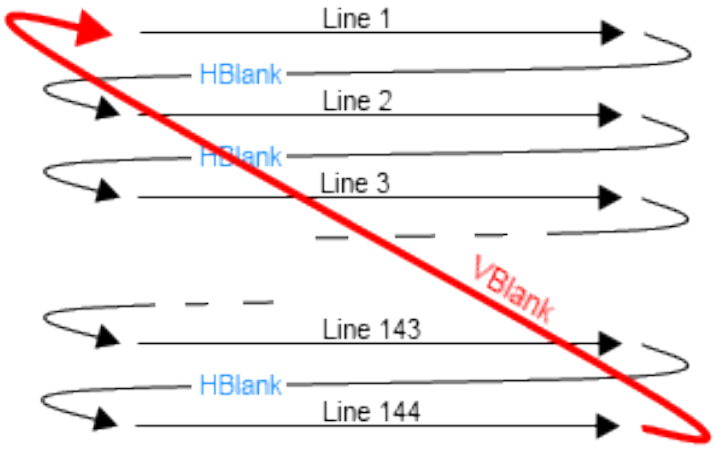
Пре него што пређемо на саму манипулацију низом пиксела, било би корисно да утврдимо када цртати нови фрејм; да успоставимо неку врсту глобалног мерења времена. Свако подешавање низа пиксела је временски скупа операција која захтева приступ већем броју различитих меморијских локација. Због тога, да бисмо избегли проблеме са перформансама на слабијим рачунарима, нећемо мењати фрејм сваки пут, већ ћемо мењати фрејм тек након што прође неки константни интервал фрејмова, уколико је потребно. Овиме смо добили значајно увећање перформанси. За прецизно мерење времена можемо користити објекат класе Stopwatch. Наводимо могућу имплементацију овог циклуса:

```
const int FRAMES_PER_SECOND = 60; // FPS
const int FRAMES_SKIPPED = 10; // preskačemo do 10 frejmova

public Stopwatch stopwatch = new Stopwatch(); // štoperica

public long FREQUENCY = Stopwatch.Frequency; //broj tickova / s
public long TICKS_PER_FRAME = FREQUENCY / FRAMES_PER_SECOND;
public long nextFrameStart; // vreme sledećeg frejma

public X80 x80; // procesor
```

```
private void IdleLoop(object sender, EventArgs e)
{
  if (x80 == null) return; //ako procesor nije inicijalizovan
  if (!Focused) return; //ako korisnik nije fokusiran na formu
  while (true)
  {
    int updates = 0; //koliko je frejmova prošlo u ovom ciklusu
    bool toUpdate = true; //da li treba ažurirati frejm

    do
    {
      UpdateFrame(toUpdate); //ažuriranje
      toUpdate = false; //ne treba dalje ažurirati u ovom ciklusu
      nextFrameStart += TICKS_PER_FRAME; //početak sledećeg frejma
    } while (nextFrameStart < stopwatch.ElapsedTicks && ++updates <
    FRAMES_SKIPPED);

    RenderFrame();
    //da li je preostalo tickova do sledećeg frejma
    long remainingTicks = nextFrameStart - stopwatch.ElapsedTicks;

    if (remainingTicks > 0) //ako jeste, privremeno zaustaviti program
    {
      Thread.Sleep((int)(1000 * remainingTicks / FREQUENCY));
    }

    else if (updates == FRAMES_SKIPPED) //ako smo preskočili max frejmova
    {
      nextFrameStart = stopwatch.ElapsedTicks;
    }
  }
}
```

Овим смо завршили имплементацију глобалног “часовника” који одређује тачно када треба ажурирати фрејм. Наравно, најважнији део и даље недостаје, а то је само ажурирање (у горњој имплементацији садржано у функцији UpdateFrame()). Када успешно поставимо и ову методу, завршили смо са основном имплементацијом графичког процесора.

Графички излаз GameBoy-a симулира катодне цеви при генерисању тренутног фрејма; код катодних цеви, екраном се прелази ред по ред снопом електрона, и када се дође до краја последњег реда, прелаз се враћа на врх. Приказ тога се може видети на слици десно. Као што се може видети, GameBoy-у треба више времена да генерише слику него само да пређе преко свих пиксела: неопходан је период хоризонталног прелаза (HBlank) после сваке линије, и период вертикалног прелаза (VBlank) при враћању на врх. Такође, графички процесор ове конзоле ће при исцртавању сваке линије да наизменично приступа VRAM-у и информацијама о спрајтовима. Ради потребе емулације, сматраћемо да су ова приступања дисјунктна и наизменична за сваку линију. Следећа табела приказује колико дуго процесору треба за сваку од горенаведених инструкција:



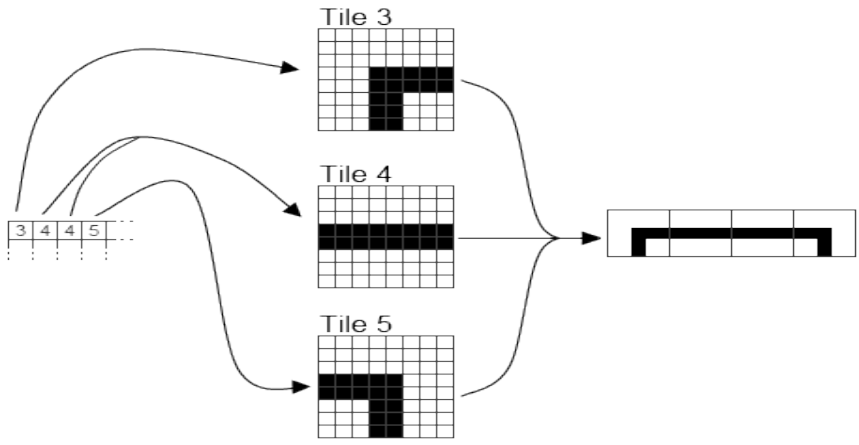
Период	Број мода	Потребно време (у tick-овима)
Приступање меморији о спрајтовима	2	80
Приступање VRAM-у	3	172
Хоризонтални прелаз	0	204
Један ред (прелаз + HBlank)		456
Вертикални прелаз	1	4560 (10 редова)
Цео фрејм (144 реда + VBlank)		70224

Сада имамо основне информације о томе како изгледа процес прављења фрејма, али нам и даље недостају информације о томе како се подаци о бојама пиксела у том фрејму чувају. Као и већина конзола тог доба, GameBoy није располагао са довољно меморије да би се низ пиксела чувао у меморији. Уместо тога је ангажован систем "плочица" (tile system). Плочице представљају скуп мањих битмапова који се чувају у меморији. Позадина се прави користећи референце на те битмапе. Главна предност овог система је што се једна плочица лако може користити више пута на екрану, једноставно користећи њену референцу.

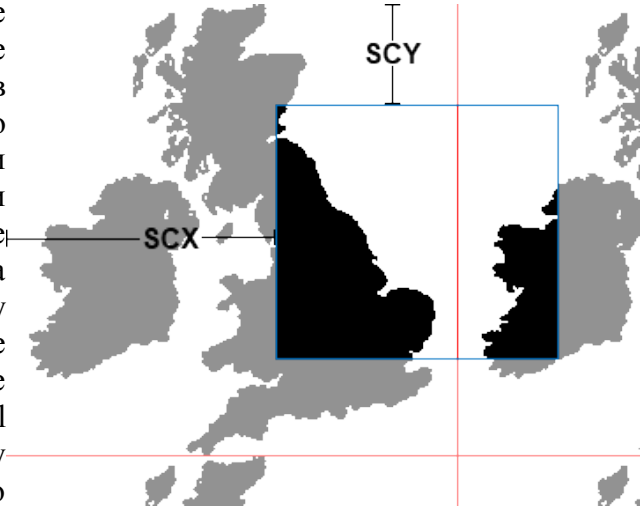
GameBoy-ев систем плочица користи плочице величине 8x8 пиксела, и 256 дистинктних плочица се могу користити при мапирању екрана. У било ком моменту могуће је држати два сета од 32x32 матрице плочица, и један од њих се у једном моменту користи за приказ. Меморија GameBoy-а даје довољно простора за 384 плочице, тако да је половина чуваних плочица дељена: први сет користи плочице са индексима 0 – 255, а други плочице са индексима -128 – 127. У наредној табели дајемо поделу VRAM-а на регионе у зависности од ове поделе:

Регион	Подаци
0x8000 – 0x87FF	Сет плочица #1 – плочице 0 – 127
0x8800 – 0x8FFF	Сет плочица #1 – плочице 128 – 255 Сет плочица #0 – плочице -128 – -1
0x9000 – 0x97FF	Сет плочица #0 – плочице 0 – 127
0x9800 – 0x9BFF	Мапа плочица #0
0x9C00 – 0x9FFF	Мапа плочица #1

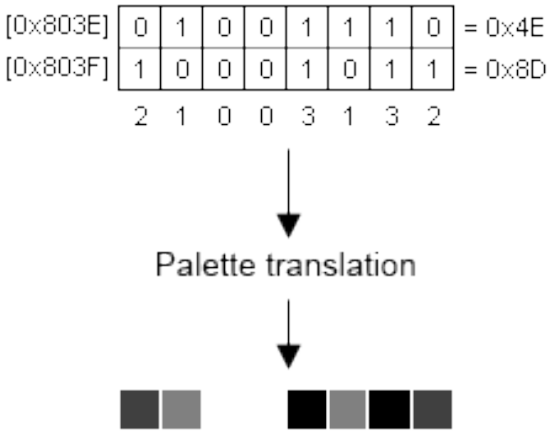
Када се дефинише позадина, њена мапа и сет плочица интерагују и производе текући фрејм, као што се може видети на доњој слици:



Као што смо навели, мапа позадине се састоји од 32x32 матрице плочица; ово даје величину од 256x256 пиксела. GameBoy-ев екран је резолуције 160x144 пиксела, што значи да се позадина може померати релативно у односу на екран. Графички процесор ово постиже тако што дефинише тачку на позадинској мапи која одговара горњој левој тачки екрана; померајући ову тачку између фрејмова, омогућено је померање позадине преко екрана. Ова тачка је дефинисана са два регистра: Scroll X и Scroll Y. Као илустрацију можемо употребити слику десно, где уоквирени део представља онај део позадине који ће бити приказан у фрејму.



GameBoy се често описује као монохроматска конзола, способна да приказује само црну и белу боју. Ово није сасвим тачно: GameBoy има способност приказивања четири боје; осим црне и беле, подржане су и две нијансе сиве (светла и тамна). Свака од ових боја се може представити користећи два бита. Међутим, код GameBoy-а постоји компликација: пар идентификационе вредности и боје није константан, тј. свака од четири могуће вредности ова два бита могу да одговарају било којој од ових боја. Зато је неопходно чувати низ од четири члана у коме ћемо памтити која боја одговара којој идентификацији у било ком тренутку. Додатна компликација је у начину на који се подаци читају из VRAM-а: при једном читању нам је доступан један ред једне плочице, а пошто се боја чува у два бита, неопходно је прочитати два реда и извршити померање битова, тако да, за неку колону, бит из првог прочитаног реда одговара мање значајном идентификационом биту боје (ради илустрације погледајте слику горе).



Сада најпре можемо искористити све ове информације да проширимо нашу класу процесора са новим променљивим и методом за ажурирање позадине:

```
public enum LcdcModeType { HBlank = 0, VBlank = 1, SearchingOamRam = 2,
   TransferingData = 3 } //imenovanje svih faza pri crtanju frejma

public LcdcModeType lcdcMode;
//boje
public const uint WHITE = 0xFFFFFFFF;
public const uint LIGHT_GRAY = 0xFFAAAAAA;
public const uint DARK_GRAY = 0xFF555555;
public const uint BLACK = 0xFF000000;

public bool TileMapSelect; //da li koristimo mapu pločica #0 ili #1

public int scrollX, scrollY;
//početne vrednosti niza koji mapira id u boju
public uint[] backgroundPalette = { WHITE, LIGHT_GRAY, DARK_GRAY, BLACK };
public uint[,] backgroundBuffer = new uint[256,256]; //pozadina
```

```
public void UpdateBackground()
{
  //pozicija početka mape pločica u VRAM-u
  int tileMapAddress = TileMapSelect ? 0x1C00 : 0x1800;

  for (int i = 0; i < 32; i++)
  {
    for (int j = 0; j < 32; j++, tileMapAddress++)
    {
      //pozicija početka informacija o tekućoj pločici
      int tileDataAddress = videoRam[tileMapAddress];

      //normalizacija pozicije u odnosu na VRAM
      if (tileDataAddress > 127) tileDataAddress -= 256;
      tileDataAddress = 0x1000 + (tileDataAddress << 4);

      int y = i << 3; //stvarna pozicija pločice na pozadini
      int x = j << 3; //stvarna pozicija pločice na pozadini

      for (int k = 0; k < 8; k++)
      {
        //uzimanje prvog i drugog reda koji odredjuju boju tekuceg reda
        int low = videoRam[tileDataAddress++];
        int high = videoRam[tileDataAddress++];
        for (int b = 7; b >= 0; b--)
        {
          backgroundBuffer[y+k,x+b] = backgroundPalette[(0x02 & high) |
          (0x01 & low)]; //podesavanje boje
          //shiftovanje do sledeće kolone
          low >>= 1;
          high >>= 1;
        }
      }
    }
  }
}
```

Konačno možemo, uz pomoć ovih promenljivih i ove metode da definišemo metodu UpdateFrame(). Da bismo simulirali procesorsko vreme koje će protećhi tokom svake od faza ažuriranja pozadine, definisaćemo i metodu ExecuteCPU() koja će određeni broj puta da izvrši procesorski ciklus.

```
private void ExecuteCPU(int ticks)
{
  do
  {
    x80.Step();
  } while (x80.ticks < ticks);
  x80.ticks -= ticks;
}
```

```
private void UpdateFrame(bool toUpdate)
{
  if (toUpdate)
  {
    uint[] backgroundPalette = x80.backgroundPalette;
    uint[,] backgroundBuffer = x80.backgroundBuffer;
    for (int y = 0, pixelIndex = 0; y < 144; y++)
    {
      //pristup podacima o sprajtovima
      x80.lcdcMode = LcdcModeType.SearchingOamRam;
      ExecuteCPU(80);
      //pristup VRAM-u
      x80.lcdcMode = LcdcModeType.TransferingData;
      ExecuteCPU(172);
      //HBlank
      x80.lcdcMode = LcdcModeType.HBlank;
      ExecuteCPU(204);
      //ažuriranje pozadine
      x80.UpdateBackground();

      int scrollX = x80.scrollX;
      int scrollY = x80.scrollY;

      for (int x = 0; x < 160; x++, pixelIndex++)
      {
        uint currColor = 0;
        currColor = backgroundBuffer[0xFF&(scrollY+y),0xFF&(scrollX+x)];
        pixels[pixelIndex] = currColor;
      }
    }
  }
  else //samo izvrši procesorske radnje ali ne ažuriraj
  {
    for (int y = 0; y < 144; y++)
    {
      x80.lcdcMode = LcdcModeType.SearchingOamRam;
      ExecuteCPU(80);
      x80.lcdcMode = LcdcModeType.TransferingData;
      ExecuteCPU(172);
      x80.lcdcMode = LcdcModeType.HBlank;
      ExecuteCPU(204);
    }
  }
  //VBlank
  x80.lcdcMode = LcdcModeType.VBlank;
  ExecuteCPU(4560);
}
```

Овим смо комплетирали основну емулацију графичког процесора. Приметимо да још увек постоје елементи који недостају, од којих су најочигледнији спрајтови; њима ћемо посветити више пажње у једном од наредних поглавља.

Сада имамо три најважнија модула у нашем емулатору, самим тим све што нам је потребно да заправо видимо неке резултате нашег рада. Међутим, пре него што можемо успешно покренути неки ROM, морамо да одрадимо још неке ствари којима ћемо створити "спрегу" ова три модула, и управо о томе ће бити речи у следећем поглављу.

3.6. Интеграција

Да бисмо успешно свезали све компоненте које смо до сада направили, неопходно је обезбедити процесору и меморији приступ одређеним регистрима који припадају графичком процесору. Након што то успешно урадимо, емулатор је већ припремљен за основну употребу.

Графички процесор GameBoy-а располаже са низом регистара који се мапирају у меморију у I/O регион. За успешну емулацију са позадинском сликом, следећи регистри се морају омогућити (постоје и други регистри које нуди графички процесор, и на неке од њих ћемо се осврнути касније у раду):

Адреса	**Регистар**
0xFF40	ЛЦД контрола
0xFF42	Scroll Y
0xFF43	Scroll X
0xFF44	Индекс тренутне линије
0xFF47	Позадинска палета

О позадинској палети је већ било речи у претходном поглављу (у питању је низ од четири елемента у коме се памти која се идентификација мапира у коју боју). Такође је већ било речи о Scroll вредностима, а индекс линије не треба даље разјашњавати. То нам оставља део са ЛЦД контролним регистром. Он се састоји од 8 одвојених флегова који контролишу различите делове графичког процесора:

Бит	**Функција**	**Ако је 0**	**Ако је 1**
0	Позадина	Искључена	Укључена
1	Спрајтови	Искључени	Укључени
2	Величина спрајтова (у пискелима)	8x8	8x16
3	Мапа плочица за позадину	#0	#1
4	Сет плочица за позадину	#0	#1
5	“Прозор”	Искључен	Укључен
6	Мапа плочица за “прозор”	#0	#1
7	Дисплеј	Искључен	Укључен

Можемо приметити да имамо три регистра везана за функционалности које до сада нисмо прекрили: спрајтови, који представљају додатне објекте које можемо померати по позадини, и “прозор” који је додатни слој који се може приказати изнад позадине. Спрајтове ћемо детаљније прекрити у једном од наредних поглавља, док су принципи рада “прозора” потпуно исти као и код позадине.

Сваки од ових регистара можемо представити као једну *bool*, односно *int* вредност унутар класе процесора. Испод следи могућа имплементација у функцији ReadByte:

```
public bool backgroundDisplayed;
public bool spritesDisplayed;
public bool largeSprites;
public bool backgroundTileMapSelect;
public bool TileSetSelect;
public bool windowDisplayed;
public bool windowTileMapSelect;
public bool lcdControlOperationEnabled;
public int scrollX, scrollY;
public int ly;
public uint[] backgroundPalette = { WHITE, LIGHT_GRAY, DARK_GRAY, BLACK };

public int ReadByte(int address)
{
  if (address <= 0x7FFF || address >= 0xA000 && address <= 0xBFFF))
  {
    . . .

  else
  {
    //samo prikazan I/O region
    switch (address)
    {
      case 0xFF40: //LCD kontrola
      {
        int ret = 0;
        if (lcdControlOperationEnabled)
        {
          ret |= 0x80;
        }
        if (windowTileMapSelect)
        {
          ret |= 0x40;
        }
        if (windowDisplayed)
        {
          ret |= 0x20;
        }
        if (TileSetSelect)
        {
          ret |= 0x10;
        }
        if (backgroundTileMapSelect)
        {
          ret |= 0x08
        }
        if (largeSprites)
        {
          ret |= 0x04;
        }
        if (spritesDisplayed)
        {
          ret |= 0x02;
        }
        if (backgroundDisplayed)
        {
          ret |= 0x01;
        }
        return ret;
      }
```

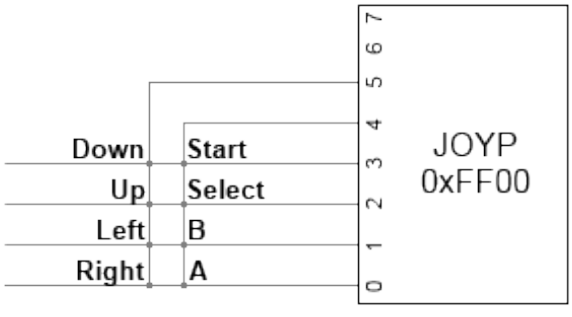
```
        case 0xFF42: //Scroll Y
          return scrollY;
        case 0xFF43: //Scroll X
          return scrollX;
        case 0xFF44: //indeks trenutne linije
          return ly;
        case 0xFF47: //pozadinska paleta
        {
          int ret = 0;
          for (int i = 3; i >= 0; i--)
          {
            ret <<= 2;
            switch (backgroundPalette[i])
            {
              case BLACK: ret |= 3; break;
              case DARK_GRAY: ret |= 2; break;
              case LIGHT_GRAY: ret |= 1; break;
              case WHITE: break;
            }
          }
          return ret;
        }
      }
    }
  }
```

Аналогно овоме треба изменити и WriteByte методу. Такође треба током извршавања функције за ажурирање фрејма ажурирати индекс тренутне линије. Уз све ове промене, коначно имамо емулатор који је способан да прикаже некакав графички излаз.

Међутим, овај емулатор није ни близу жељеном резултату: само је способан за приказивање позадине, и не даје било какве могућности за контролу самом играчу. Тим проблемом ћемо се бавити у следећем поглављу.

3.7. Улаз

Као што смо навели у анализи техничких спецификација GameBoy-а, ова конзола располаже са осам различитих тастера којима играч може да контролише неке акције у игри: четири стрелице, “A”, “B”, “SELECT” и “START”. Могуће је да у исто време буде притиснуто више тастера. Код већине тастатура, тастери су постављени у мрежу колона и редова које можемо третирати као жице, прко којих неки тастер може да направи везу. Када је нека колона активирана, било који редови повезани са њом ће се такође активирати, и хардвер тада може да препозна активне редове да би одредио тренутно притиснуте тастере.



Код GameBoy-а, ова мрежа има две колоне и четири реда, што је добро, у смислу да се све неопходне конекције могу чувати у једном 8-битном регистру. (Види слику)

Са слике се одмах може закључити начин на који начин GameBoy одређује који су тастер притиснути:

- Уписати 0x10 или 0x20 у регистар: ово ће активирати четврти, односно пети бит;
- Сачекати неко време да повезани редови промене вредност регистра;
- Проверити доња 4 бита регистра да би се сазнало који су редови активни за ту колону.

Код који ће процесирати притиске тастера и одредити који је тастер притиснут је релативно једноставан за куцање у C# језику. Оно што је мало теже је искуцати начин на који ће се примљене информације интерпретирати у процесору. Зато најпре приказујемо имплементацију унутар класе процесора:

```
//boolovi za svaki taster
public bool leftKeyPressed;
public bool rightKeyPressed;
public bool aButtonPressed;
public bool bButtonPressed;
public bool startButtonPressed;
public bool selectButtonPressed;
public bool keyP14, keyP15; //bool za kolone

public void KeyChanged(Keys keyCode, bool pressed)
{
  switch (keyCode)
  {
    case Keys.D:
      bButtonPressed = pressed;
      break;
    case Keys.F:
      aButtonPressed = pressed;
      break;
    case Keys.Enter:
      startButtonPressed = pressed;
      break;
    case Keys.Shift:
      selectButtonPressed = pressed;
      break;
    case Keys.Up:
      upKeyPressed = pressed;
      break;
    case Keys.Down:
      downKeyPressed = pressed;
      break;
    case Keys.Left:
      leftKeyPressed = pressed;
      break;
    case Keys.Right:
      rightKeyPressed = pressed;
      break;
  }
}

public void WriteByte(int address, int value)
{
  . . .
  else
  {
    //samo prikazan Keypad deo
    switch (address)
    {
      case 0xFF00: //key pad
        keyP14 = (value & 0x10) != 0x10;
        keyP15 = (value & 0x20) != 0x20;
        break;
    }
  }
}
```

```
public int ReadByte(int address)
{
  . . .
  else
  {
    //samo prikazan Keypad deo
    switch (address)
    {
      case 0xFF00: //key pad
        if (keyP14)
        {
          int ret = 0;
          if (!downKeyPressed) ret |= 0x08;
          if (!upKeyPressed) ret |= 0x04;
          if (!leftKeyPressed) ret |= 0x02;
          if (!rightKeyPressed) ret |= 0x01;
          return ret;
        }
        else if (keyP15)
        {
          int ret = 0;
          if (!startButtonPressed) ret |= 0x08;
          if (!selectButtonPressed) ret |= 0x04;
          if (!bButtonPressed) ret |= 0x02;
          if (!aButtonPressed) ret |= 0x01;
          return ret;
        }
        break;
    }
  }
}
```

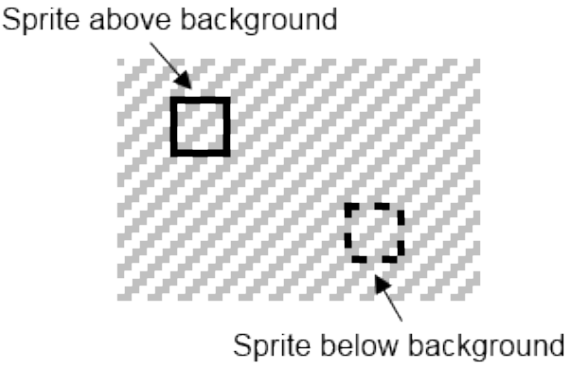
Сада нам је једино преостало да у самој апликацији дефинишемо шта радити када дође до притиска (односно отпуштања) тастера:

```
private void Form1_KeyDown(object sender, KeyEventArgs e)
{
  x80.KeyChanged(e.KeyCode, true);
}
private void Form1_KeyUp(object sender, KeyEventArgs e)
{
  x80.KeyChanged(e.KeyCode, false);
}
```

Овим смо успешно решили проблем улаза, и сада се на емулатору могу играти једноставније игре као што је икс-окс. Међутим, већина игара се мора играти наслепо, зато што се може приметити да у већини игара нигде неће бити показивача који ће указати играчу где се тренутно налази на пољу. Игрице производе ове индикаторе тако што користе спрајтове. Спрајтови се могу дефинисати као “плочице” које се могу помоћу графичког процесора сместити изнад позадине и независно померати. Велики број игара за GameBoy изузетно примењује спрајтове у својој имплементацији, и зато је следећи и врло важан корак правилно их емулирати.

3.8. Спрајтови

Као што смо већ навели, GameBoy игрице екстензивно користе спрајтове: могуће је поставити их изнад и испод позадине, као и да их буде више истовремено. Спрајтови су у суштини плочице, исте врсте какве користи и позадина; то значи да је сваки спрајт димензије 8x8 пиксела (с тим да је могуће приказати их и у димензијама од 8x16 пиксела, сетовањем одговарајућег графичког регистра који смо навели у поглављу о интеграцији). Спрајт се такође може слободно кретати по позадини, и може бити делимично или потпуно изван екрана. Спрајт изнад позадине је потпуно приказан, а спрајт испод позадине само у пикселима где позадина има вредност боје 0. Наравно, на сличан начин ће и спрајт изнад позадине да пропусти позадину на пикселима где има вредност боје 0. (Ради илустрације погледати слику десно)



Да би се овакав изглед симулирао унутар емулатора, најједноставнији начин би био најпре нацртати спрајтове испод позадине, затим позадину, па онда спрајтове изнад позадине. Међутим, овај алгоритам је непрактичан у смислу да ће поновити поступак прављења спрајтова два пута. Боље је прво исцртати позадину, па затим на основу приоритета спрајта и боје позадине на некој позицији одредити да ли цртати тај пиксел спрајта или не. GameBoy задаје једну додатну компликацију код његовог система за спрајтове: спрајт може бити "обрнут" хоризонтално или вертикално од стране хардвера током исцртавања. Ова могућност је направљена у оригиналној конзоли да би се чувала меморија, зато што се, на пример, кретање уназад објеката може симулирати као кретање унапред уз примењено коректно обртање.

GameBoy има довољно меморије да чува информације за око 40 спрајтова, у меморијском региону за који смо током анализирања меморије само навели да се зове OAM (Object Attribute Memory). Сваки од 40 спрајтова поседује 4 бајта у OAM-у. Следи детаљан опис информација које су садржане у ова 4 бајта:

<table>
<tr><th>Бајт</th><th>Опис</th></tr>
<tr><td>0</td><td>Y координата горњег левог темена
(вредност која се памти је заправо Y – 16)</td></tr>
<tr><td>1</td><td>X координата горњег левог темена
(вредност која се памти је заправо X – 8)</td></tr>
<tr><td>2</td><td>Индекс плочице коју користи спрајт</td></tr>
<tr><td>3</td><td>Модификације спрајта:
<table>
<tr><th>Бит</th><th>Опис</th><th>Ако је 0</th><th>Ако је 1</th></tr>
<tr><td>7</td><td>Приоритет спрајта</td><td>Изнад позадине</td><td>Испод позадине</td></tr>
<tr><td>6</td><td>Вертикално обртање</td><td>Нормалан</td><td>Обрнут</td></tr>
<tr><td>5</td><td>Хоризонтално обртање</td><td>Нормалан</td><td>Обрнут</td></tr>
<tr><td>4</td><td>Индекс палете</td><td>#0</td><td>#1</td></tr>
</table></td></tr>
</table>

Као што се може приметити, графички процесор има приступ двема палетама за спрајтове; сваки од 40 спрајтова може да користи неку од те две палете, у зависности од вредности која стоји у његовом делу OAM-а. Ове палете се чувају у I/O региону меморије одмах после позадинске палете (на адресама 0xFF48 и 0xFF49) и њима се манипулише на исти начин као и са позадинском палетом. Због тога нећемо наводити код којим приступамо палетама за спрајтове.

Податке о плочицама које користе спрајтови можемо унутар класе процесора чувати у четвородимензионалном низу величине 256 * 8 * 8 * 2. Димензије ће представљати, редом: индекс плочице, Y координату, X координату, индекс палете. Приликом ажурирања ових плочица, што морамо обавити пре него што можемо исцртати спрајтове, неопходно је ажурирати овај низ тамо где је потребно. Следи могућа имплементација тог ажурирања:

```
//palete za sprajtove
public uint[] objectPalette0 = { WHITE, LIGHT_GRAY, DARK_GRAY, BLACK };
public uint[] objectPalette1 = { WHITE, LIGHT_GRAY, DARK_GRAY, BLACK };
//četvorodimenzionalni niz
public uint[,,,] spriteTile = new uint[256, 8, 8, 2];

public void UpdateSpriteTiles()
{
  for (int i = 0; i < 256; i++)
  {
    int address = i << 4; //pozicija i-te pločice u VRAM-u
    for (int y = 0; y < 8; y++)
    {
      int low = videoRam[address++]; //donji bitovi boje
      int high = videoRam[address++]; //gornji bitovi boje
      for (int x = 7; x >= 0; x--)
      {
        //indeks unutar palete
        int paletteIndex = (0x02 & high) | (0x01 & low);
        //shiftovanje na sledeću kolonu
        low >>= 1;
        high >>= 1;
        if (paletteIndex > 0)
        {
          spriteTile[i, y, x, 0] = objectPalette0[paletteIndex];
          spriteTile[i, y, x, 1] = objectPalette1[paletteIndex];
        }
        else //indeks je 0, sprajt će biti transparentan u ovom pikselu
        {
          spriteTile[i, y, x, 0] = 0;
          spriteTile[i, y, x, 1] = 0;
        }
      }
    }
  }
}
```

Сада можемо интегрисати овај низ и методу унутар методе за ажурирање фрејма коју смо написали у поглављу о графичком процесору. Као што је већ напоменуто, цртање спрајтова ћемо вршити након исцртавања позадине. GameBoy врши исцртавање спрајтова линију по линију, исто као што и црта позадину; зато ћемо и у нашем емулатору спрајтове цртати на овај начин. При пролазу кроз сваку линију ћемо испитати да ли се сваки од спрајтова налази у тој линији; уколико јесте процесираћемо текући пиксел тог спрајта, и у зависности од његовог приоритета и транспарентности позадине нацртати. Следи нови код за функцију UpdateFrame(); изостављени су претходно додати делови за цртање позадине.

```
private void UpdateFrame(bool toUpdate)
{
  if (toUpdate)
  {
    byte[] oam = x80.oam;
    for (int y = 0, pixelIndex = 0; y < 144; y++)
    {
      //izostavljeni delovi sa crtanjem pozadine
      x80.UpdateSpriteTiles();

      if (x80.spritesDisplayed) //ako se sprajtovi prikazuju
      {
        uint[,,,] spriteTile = x80.spriteTile;
        if (!x80.largeSprites) //ako su velicine 8x8
        {
          //prolazimo kroz svih 40 sprajtova u OAM-u
          for (int address = 0; address < 160; address += 4)
          {
            int spriteY = oam[address]; //Y - 16
            int spriteX = oam[address + 1]; //X - 8
            if (spriteY==0||spriteX==0||spriteY>=160||spriteX>=168)
            {
              continue; //ako je sprajt van ekrana
            }
            spriteY -= 16; //normalizacija Y-a
            if (spriteY > y || spriteY + 7 < y)
            {
              continue; //ako je sprajt van tekuće linije
            }
            spriteX -= 8; //normalizacija X-a

            int spriteTileIndex = oam[address + 2]; //indeks pločice
            int spriteFlags = oam[address + 3]; //modifikacije sprajta
            //odredjivanje modifikacija
            bool spritePriority = (0x80 & spriteFlags) == 0x80;
            bool spriteYFlipped = (0x40 & spriteFlags) == 0x40;
            bool spriteXFlipped = (0x20 & spriteFlags) == 0x20;
            int spritePalette = (0x10 & spriteFlags) == 0x10 ? 1 : 0;

            int spriteRow = y - spriteY; //red u odnosu na sprajt
            //pozicija u nizu piksela
            int screenAddress = y * 160 + spriteX;
            for (int x = 0; x < 8; x++, screenAddress++)
            {
              int screenX = spriteX + x; //tekuća X koordinata ekrana
              if (screenX >= 0 && screenX < 160) //ako je na ekranu
              {
                //boja tekućeg piksela sprajta, uz eventualna obrtanja
                uint color = spriteTile[spriteTileIndex, spriteYFlipped ?
                7 - spriteRow : spriteRow, spriteXFlipped ? 7 - x : x,
                spritePalette];
                if (color > 0) //ako sprajt nije transparentan ovde
                {
                  if (spritePriority) //ako sprajt nema prioritet
                  {
                    if (pixels[screenAddress] == 0xFFFFFFFF)
                    {
                      pixels[screenAddress] = color;
                    }
                  }
```

```
                    else
                    {
                      pixels[screenAddress] = color;
                    }
                  }
                }
              }
            }
          }
          else
          {
            //analogni kod za sprajtove velicine 8x16
          }
        }
      }
    }
    else
    {
      //ukoliko ne ažuriramo frejm...
    }
}
```

Успешно смо поставили систем за спрајтове, и сада ће неки број једноставних GameBoy игрица радити у потпуности. Међутим, велики број игрица неће радити без додатне компоненте: начина на који се одређује када се фрејм може поново исцртати. Скоро свака игра врши "освежавање" екрана док је екран у фази VBlank-а, пошто се ове промене неће приказати све док графички процесор поново не почне ажурирати фрејм.

Неке једноставне игре ће проверити када је екран у VBlank-u тако што ће гледати у којој се линији тренутно налази графички процесор при ажурирању; кад дође до линије 144, тада је VBlank започео. Ово захтева значајну количину процесорске снаге током понављања циклуса. Зато је чешћа метода информисање игре о неком догађају када се деси: ова порука се назива прекидом. У наредном поглављу ће бити нарочито речи о VBlank прекиду, и како се исти може симулирати да би се ова порука допремила до емулиране игрице.

3.9. Прекиди. VBlank прекид

Из претходног пасуса се може лако закључити да је концепт прекида јако користан: најважније информације се преко прекида могу директно "сервирати" процесору уместо да процесор самостално проверава за њих. Овим се значајно штеди време и процесорска снага.

Са становишта хардвера, процесор мора да привремено прекине извршавање онога што тренутно ради када му се достави неки прекид, и уместо тога почне да процесира тај прекид на адекватан начин. Процесор треба да провери да ли има прекида током сваке нове итерације процесорског циклуса. Уколико прекид постоји, процесор треба да запамти позицију где се до тада налазио и изврши "скок" до адресе која се бави разрешењем тог конкретног прекида.

Код GameBoy-а постоји пет различитих жица које транспортују прекида; оне потичу из разноврсних периферија ове конзоле. Свака од њих има сопствену адресу за разрешење:

Прекид	Адреса разрешивача
VBlank	0x0040
ЛЏД статус	0x0048
Прекорачење тајмера	0x0050
Серијски линк	0x0058
Притисак тастера	0x0060

У конкретном случају VBlank-а, жица се убацује на дно екрана: чим графички процесор заврши са ажурирањем свих линија екрана и стигне до дна, прекид се окида и процесор извршава скок до адресе 0x0040, извршавајући разрешивач за VBlank прекид.

Постоје и ситуације у којима прекиде не треба процесирати (можда је нпр. операција која се тренутно извршава у процесору високог приоритета и не треба је прекидати). Да би овакви случајеви били правилно имплементирани, већина процесора (па ни GameBoy-ев није изузетак) садржи неку врсту "главног флега" за прекиде. Ако и само ако је овај флег активан, процесор ће процесирати прекиде. GameBoy садржи два регистра унутар меморије са којима контролише прекиде; спецификације су дате у табели:

<table>
<tr><th>Регистар</th><th>Адреса</th><th>Опис</th><th>Детаљи</th></tr>
<tr><td>Омогућавање прекида (главни флег)</td><td>0xFFFF</td><td>Када су битови подешени, респективни прекиди се могу процесирати</td><td>
<table>
<tr><th>Бит</th><th>Ако је 0</th><th>Ако је 1</th></tr>
<tr><td>0</td><td>VBlank искљ.</td><td>VBlank укљ.</td></tr>
<tr><td>1</td><td>ЛЏД искљ.</td><td>ЛЏД укљ.</td></tr>
<tr><td>2</td><td>Тајмер искљ.</td><td>Тајмер укљ.</td></tr>
<tr><td>3</td><td>Серијски искљ.</td><td>Серијски укљ.</td></tr>
<tr><td>4</td><td>Тастер искљ.</td><td>Тастер укљ.</td></tr>
</table>
</td></tr>
<tr><td>Флегови прекида</td><td>0xFF0F</td><td>Када су битови подешени, респективни прекид се догодио</td><td>Исти распоред као и за 0xFFFF</td></tr>
</table>

Пошто се ови регистри налазе у меморији, њихову имплементацију обављамо преко класе процесора: додавањем *bool*-ова и изменом функција ReadByte и WriteByte:

```
public bool interruptsEnabled = true; //da li je omogućeno procesiranje

public bool keyPressedInterruptRequested; //da li je došlo do prekida?
public bool keyPressedInterruptEnabled; //da li je omogućeno procesiranje
//analogno za ostale prekide
public bool serialIOInterruptRequested;
public bool serialIOInterruptEnabled;
public bool timerOverflowInterruptRequested;
public bool timerOverflowInterruptEnabled;
public bool lcdcInterruptRequested;
public bool lcdcInterruptEnabled;
public bool vBlankInterruptRequested;
public bool vBlankInterruptEnabled;
```

```
public int ReadByte(int address)
{
  //prikazan samo I/O region
  else
  {
    switch (address)
    {
      case 0xFF0F: //Interrupt flag
      {
        int ret = 0;
        if (keyPressedInterruptRequested)
        {
          ret |= 0x10;
        }
        if (serialIOInterruptRequested)
        {
          ret |= 0x08;
        }
        if (timerOverflowInterruptRequested)
        {
          ret |= 0x04;
        }
        if (lcdcInterruptRequested)
        {
          ret |= 0x02;
        }
        if (vBlankInterruptRequested)
        {
          ret |= 0x01;
        }
        return ret;
      }
      case 0xFFFF: //Interrupt enable
      {
        int ret = 0;
        if (keyPressedInterruptEnabled)
        {
          ret |= 0x10;
        }
        if (serialIOInterruptEnabled)
        {
          ret |= 0x08;
        }
        if (timerOverflowInterruptEnabled)
        {
          ret |= 0x04;
        }
        if (lcdcInterruptEnabled)
        {
          ret |= 0x02;
        }
        if (vBlankInterruptEnabled)
        {
          ret |= 0x01;
        }
        return ret;
      }
    }
  }
}
```

Аналогно треба ажурирати и WriteByte методу. Преостало је још само да симулирамо одговор процесора на прекиде; пре тога треба имплементирати процесорске функције које се баве прекидима (скок до разрешивача, итд). Следи могућа имплементација најважније три:

```
private void Return() //vraćanje na prvu adresu sa stacka
{
  Pop(ref PC);
}

private void Interrupt(int address) //razrešivač na adresi address
{
  interruptsEnabled = false; //dok procesiramo prekid ne možemo ostale
  Push(PC); //pamćenje trenutne pozicije na stack
  PC = address; //skok do adrese
}

private void ReturnFromInterrupt() //povratak iz prekida
{
  interruptsEnabled = true;
  Return();
  ticks += 4; //operacija zahteva 4 procesorska ticka
}
```

Сада можемо имплементирати и одговор процесора:

```
public void Step()
{
  if (interruptsEnabled)
  {
    if (vBlankInterruptEnabled && vBlankInterruptRequested)
    {
      vBlankInterruptRequested = false;
      Interrupt(0x0040);
    }
    else if (lcdcInterruptEnabled && lcdcInterruptRequested)
    {
      lcdcInterruptRequested = false;
      Interrupt(0x0048);
    }
    else if (timerOverflowInterruptEnabled&&timerOverflowInterruptRequested)
    {
      timerOverflowInterruptRequested = false;
      Interrupt(0x0050);
    }
    else if (serialIOInterruptEnabled && serialIOInterruptRequested)
    {
      serialIOInterruptRequested = false;
      Interrupt(0x0058);
    }
    else if (keyPressedInterruptEnabled && keyPressedInterruptRequested)
    {
      keyPressedInterruptRequested = false;
      Interrupt(0x0060);
    }
  }

  //ostatak iteracije procesorskog ciklusa koji smo ranije naveli
}
```

Успешно смо имплементирали процесорско руковање прекидима. Сада је још само остало одредити моменте у којима се одређени прекиди шаљу. Овде ћемо се задржати само на VBlank прекиду, док се остали или прослеђују на сличан начин као VBlank прекид, или покривају компоненте GameBoy-а које нисмо обрадили овим радом.

Када би био прави моменат да се процесору проследи VBlank прекид? Логичан одговор је, наравно, када се активира VBlank фаза у функцији која ажурира фрејм. Додавањем овог слања смо комплетирали имплементацију VBlank прекида. Следи имплементација у методи UpdateFrame():

```
private void UpdateFrame(bool toUpdate)
{
  if (toUpdate)
  {
    . . .
  }
  else
  {
    . . .
  }
  //prikazujemo samo deo sa ulazom u VBlank
  x80.lcdcMode = LcdcModeType.VBlank;
  //ovo je pravi momenat za slanje prekida
  if (x80.VBlankInterruptEnabled)
  {
    x80.VBlankInterruptRequested = true;
  }
  ExecuteCPU(4560);
}
```

Са додатком само VBlank прекида већ смо знатно проширили број игрица које се могу у потпуности емулирати на нашем емулатору. Једна од њих је и "Тетрис", игра која је прославила ову конзолу, са продатих 35 милиона кертрица.

Овим смо завршили поглавље о прекидима, које је уједно и последње поглавље ове главе.

ЗАКЉУЧАК

Главни задатак овог матурског рада и пројекта је био јасан: емулирати GameBoy конзолу и што концизније документовати тај поступак. Као неопходан увод је најпре нешто речено о емулацији уопштено, а затим се емулацији уређаја приступило корак по корак, пратећи некакав логичан редослед. По мом мишљењу, пројекат је успешно изведен, и као производ је добијен емулатор који може ефикасно да емулира добар део оригиналних ROM-ова за GameBoy.

Потребно је напоменути, наравно, да постоји већи број компоненти GameBoy-а које нису покривене овим радом. Неке од њих, као што је читање података из ROM-а кертриџа, додатни прекиди, тајмери... су имплементирани накнадно у овај пројекат, док су неки други, као што је емулација звука, нажалост изостављени. Иако сам доста научио о једној од омиљених конзола из детињства, за одређене додате делове емулатора сам ипак морао користити готов код: читање из ROM-а је један од њих. Међутим, готово све компоненте које су наведене у склопу овог рада су испрограмиране са моје стране, уз екстензивну употребу разноврсне литературе (као што се и може видети из списка литературе).

Надам се да ће мој приступ овој теми бити окарактерисан као добар и практичан. Један од већих циљева ми је био да ову, углавном непознату тему ученицима Математичке гимназије, изложим на такав начин да би, евентуално, будуће заинтересоване генерације ове школе могле да лако овладају основним вештинама неопходним за емулацију користећи овај рад. Сматрам тему емулације изузетно занимљивом и корисном, али и темом која захтева велику посвећеност и спремност на самостално истраживање о уређају који се емулира: често се каже да особа која мора да пита “Како да направим емулатор?” није ни приближно спремна да га направи.

На самом крају овог рада, желео бих да се захвалим:

Ментору овог рада и професору програмирања и програмских језика у четвртом разреду **Мијодрагу Ђуришићу**, који ми је током израде рада неизмерно помогао са саветима и програмерским “триковима”;

Професорки рачунарства и информатике у трећем разреду **Невенки Спалевић**, без чијег неопходног увода у свет хардвера вероватно не би било ни концепта за овај рад;

Мојим друговима из Математичке гимназије **Ђорђу Николићу** и **Вањи Шарковић**, који су ми често давали мотивацију током израде рада;

Свим ауторима документација хардвера за GameBoy, без чијег би мукотрпног рада у документовању сваке функционалности ове мале, али изузетне конзоле, мој посао био много комплекснији;

Било коме ко није наведен при горњим захвалностима, а допринео је стварању овог рада на било који начин.

У Београду, јун 2012.

Петар Величковић

ЛИТЕРАТУРА

[1] Wikipedia: “Emulator”
http://en.wikipedia.org/wiki/Emulator

[2] Koninklijke Bibliotheek: “Projecten Emulatiewatis”:
http://www.kb.nl/hrd/dd/dd_projecten/projecten_emulatiewatis-en.html

[3] Stanford University Encyclopedia of Philosophy: “The Church-Turing Thesis”
http://plato.stanford.edu/entries/church-turing/

[4] Wikipedia: “GameBoy”
http://en.wikipedia.org/wiki/GameBoy

[5] Nintendo Wikia: “Game Boy”
http://nintendo.wikia.com/wiki/Game_Boy

[6] Marat Fayzullin: “How To Write a Computer Emulator”
http://fms.komkon.org/EMUL8/HOWTO.html

[7] Ramesh M. Gaonkar: “Z-80 Microprocessor: Architecture, Interfacing, Programming, and Design (3rd Edition)

[8] Imran Nazar: “GameBoy Z80 Opcode Map”
http://imrannazar.com/Gameboy-Z80-Opcode-Map

[9] Jan Verhoeven: “Overview of the GameBoy Color Hardware”
http://verhoeven272.nl/cgi-bin/FS?fruttenboel%2FGameboy&Gameboy+section&GBtop&GBsummary&GBcontent

[10] Jan Verhoeven: “GameBoy Pan Docs”
http://verhoeven272.nl/cgi-bin/FSgz?fruttenboel%2FGameboy&Fruttenboel+GameBoy&GBtop&pandocs&GBcontent