Факултет за електротехника и информациски технологии

ПРОЕКТНА ЗАДАЧА ПО МИКРОПРОЦЕСОРСКА ЕЛЕКТРОНИКА

ТЕМА:

CHIP-8 Интерпретер

Изработил:

Бојан Софрониевски, 1/2018

Ментор:

проф. д-р Зоран Ивановски

Јуни, 2021 година

Содржина

Вовед

CHIP-8 е едноставен интерпретиран програмски јазик кој за прв пат се користел за компјутерите кои се продавале како комплет за да ги состави самиот корисник, кон крајот на 1970-тите години и почетокот на 1980-тите години, со главна цел да го олесни и забрза пишувањето на игри. Овие едноставни компјутери, како што биле COSMAC VIP, DREAM-6800 и Telmac 1800, биле направени за да се поврзат на телевизор кој служел како дисплеј, имале 1 kB до 4 kB RAM и хексадецимална тастатура како влез, за интеракција со корисникот. CHIP-8 бил креаиран од страна на Joseph Weisbecker, кој според [1] го опишува јазикот како многу покомпактен во споредба со BASIC, односно програмите, пред се игрите, зафаќале неколку пати помалку меморија во однос на тие напишани во BASIC, а поради недостапноста на секундарната меморија во тоа време, било неопходно корисниците да ги внесуваат програмите секој пат кога би сакале да ги извршат, па CHIP-8 овозможувал брзо препишување на програмите, со помалку линии код. Друга предност била тоа што CHIP-8 интерпретерот имал помали барања на хардверски ресурси, во однос на BASIC. Во почетокот на 1990-тите години, CHIP-8 повторно почнал да заживува, кога бил направен интерпретер за графичкиот калкулатор HP48, со што многу се олеснило пишувањето игри за овој калкулатор. Денес, голем број на луѓе кои сакаат да се запознаат со емулација на компјутерските системи, почнуваат со CHIP-8. Важно е да се забележи дека програмите што извршуваат CHIP-8 псевдомашински инструкции не се нарекуваат емулатори, туку интерпретери, бидејќи зборот емулатор се користи кога се емулира вистински хардвер.

Цел на оваа проектна задача е да се напише CHIP-8 интерпретер во x86 асемблерски јазик. Работата на интерпретерот е тестирана во DOSBox емулаторот, MS-DOS 6.22 виртуелна машина и FreeDOS, а како ROM-ови се користат неколку игри напишани во CHIP-8, како што се клонови на Space Invaders, Bomber, Pong, Tetris и др.

Краток опис на CHIP-8

1. Меморија

CHIP-8 јазикот може да адресира максимум 4 kB (4096 B) RAM, што одговара на мемориски адреси 000h до FFFh. Првите 512 B од меморијата, на мемориски адреси 000h до 1FFh, оригинално биле наменети за интерпретерот и не смееле да се користат од страна на програмите. Во модерните имплементации, во овој мемориски простор се сместени графиките(анг. sprites) за хексадецималните цифри, од 0 до F, по 5 бајти за секоја (димензиите на графиката за цифрите се 8x5 пиксели). Програмите започнуваат на адреса 200h.

2. Регистри

CHIP-8 има 16 8-битни регистри за општа намена, означени со Vx, каде x е хексадецимална цифра. Регистарот VF не треба да се користи во програмите, бидејќи се користи како знаменце по извршувањето на определени инструкции.

Има еден 16-битен регистар наречен I, кој служи за адресирање на меморијата, па се користат само 12-те бита со помала тежина.

Постојат и два 8-битни регистри наменети за тајмерот за доцнење и тајмерот за звук. Кога овие регистри имаат ненулта вредност, автоматски се декрементираат со фреквенција од 60 Hz.

Постојат и така наречени „псевдо-регистри“, кој не можат да се пристапат од програмите. Таков регистар е 16-битниот програмски бројач (PC), кој ја содржи адресата на инструкцијата што тековно се извршува. Исто така има и 8-битен регистар, покажувач на стекот, кој покажува на врвот од стекот. Стекот претставува LIFO структура од 16 16-битни вредности, во кои се зачувува адресата на која интерпретерот треба да се врати по извршувањето на некоја субрутина.

3. Дисплеј

Дисплејот што се користи во CHIP-8 јазикот е монохроматски дисплеј, со големина 64x32 пиксели. Исцртувањето на екранот се врши со графики со големина која може да биде максимум 15 бајти, чија бинарна репрезентација соодвествува на еден пиксел од дисплејот. Според ова, максималната големина на графиката што ќе се исцрта може да

биде 8x15 пиксели. Координатниот почеток е сместен во горниот лев ќош, x-оската е насочена надесно, а y-оската е насочена надолу.

4. Тастатура

Компјутерите на кои оригинално се користел CHIP-8 јазикот имале хексадецимална тастатура, со распоред на копчињата зададен подолу. Во нашиот интерпретер ќе направиме мапирање на копчињата, исто прикажано подолу.

1	2	3	C
4	5	6	D
7	8	9	E
A	0	B	F

↔

1	2	3	4
Q	W	E	R
A	S	D	F
Z	X	C	V

5. Тајмери и звук

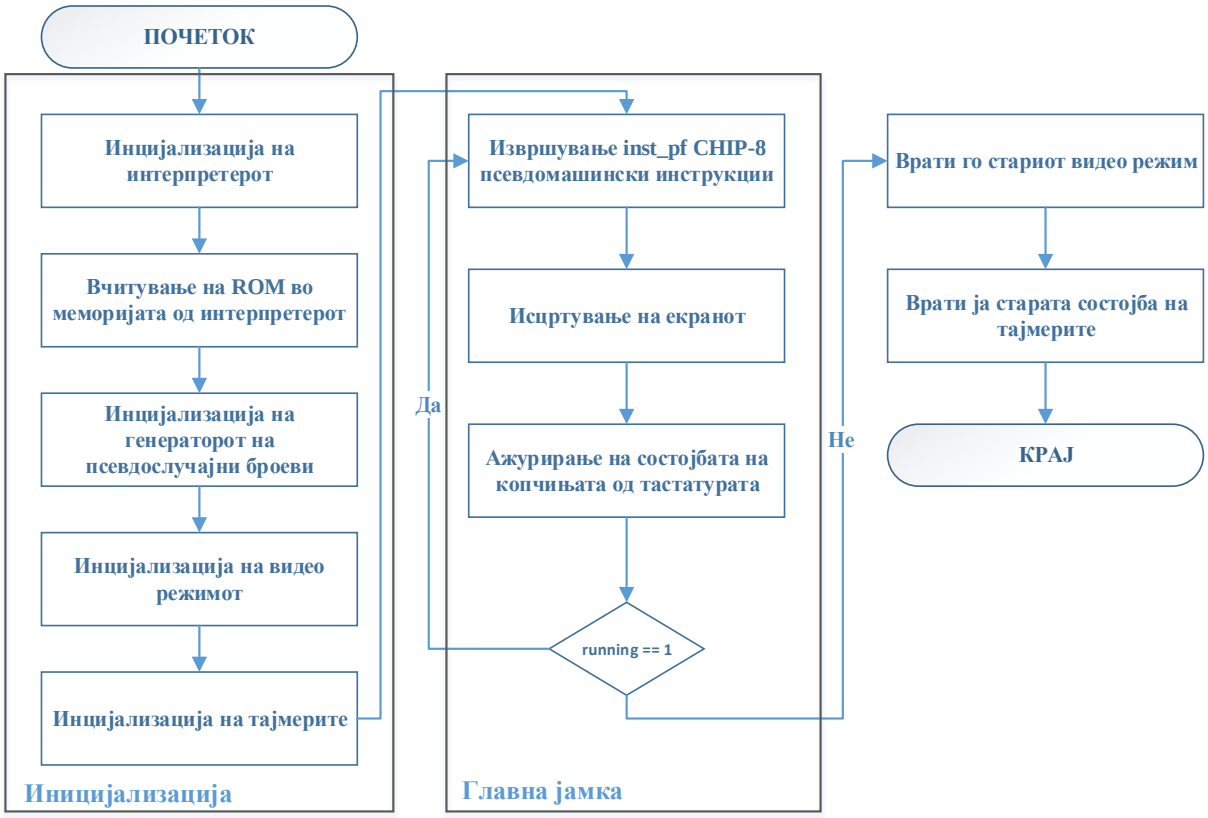
CHIP-8 има два тајмери. Првиот тајмер, наречен тајмер за доцнење (delay timer – DT), е активен кога регистарот поврзан со него има ненулта вредност. Се додека овој регистар има ненулта вредност, се одзема 1 од регистарот со фреквенција 60 Hz, а кога вредноста ќе стане 0, тајмерот се деактивира. Вториот тајмер е наменет за генерирање на звук, кој на ист начин го декрементира регистарот поврзан со него како и тајмерот за доцнење, со иста фреквенција од 60 Hz, но се додека регистарот има вредност поголема од нула, се генерира тон со однапред определена фреквенција, која ја дефинира креаторот на интерпретерот.

CHIP-8 псевдомашински инструкции

CHIP-8 има вкупно 36 различни инструкции, за извршување математички операции, за графика и за контрола на текот на програмата. Инструкциите се со должина од 2 бајти, зачувани во big-endian формат во меморијата (позначајниот бајт е зачуван на пониска адреса). Листата на инструкции, заедно со нивниот операциски код, може да се видат од [2].

Опис на алгоритмот

Блок дијаграм на алгоритмот



Во продлжение ќе дадеме подетален опис на чекорите од алгоритмот.

1. Иницијализација

1.1. Иницијализација на интерпретерот

Во овој чекор правиме инцијализација на сите променливи поврзани со интерпретерот. За таа цел имплементирани се две тесно поврзани процедури: `Init` и `Reset`. `Init` процедурата прави чистење на меморијата наменета за интерпретерот, која ја дефинираме како низа `mem` во податочниот сегмент, со големина 4096 бајти. Чистењето го правиме така што сите елементи од низата ги поставуваме на нула. Бидејќи треба да поставиме иста вредност 0 на голем број локации, може да ја искористиме `STOSB` инструкцијата, како што е наведено во следниот сегмент:

```
mov     ax, ds                  ; 1. Cistenje na memorijata
mov     es, ax
lea     di, mem
```

```
mov     cx, mem_size
xor     al, al
rep     stosb
```

По чистењето на меморијата, на почеток од меморијата ги копираме графиките за хексадецималните цифри, кои ги дефинираме во податочниот сегмент со низата `fontset`. Бидејќи треба да копираме поголем број бајти од една во друга низа, може да ја користиме `MOVSB` инструкцијата.

`Reset` процедурата е задолжена за иницијалзиација на сите регистри поврзани со интерпретерот, бришење на низата која ќе ја содржи состојбата на пикселите за дисплејот и ресетирање на низата која ги чува состојбите на копчињата (дали се притиснати). Програмскиот бројач, дефиниран како променлива `PC` во податочниот сегмент, се поставува на вредност 200h (адресата каде што започнува програмскиот код за интерпретерот). Vx регистрите, дефинирани како низа со големина 16 бајти `V` и регистарот `I` се инцијализираат на вредност 0. Истото се прави и со покажувачот на врвот од стекот, `stk_p`, како и самиот стек, `stk` се иницијализира со нули. Регистарот за тајмерот за доцнење, `delay_t` и регистарот за тајмерот за звук `sound_t` исто така се поставуваат на вредност 0. За ресетирање на состојбата на копчињата се користи процедурата `ResetKeyState`, која поставува нули во низата `keys`, која има големина 16 бајти и секој елемент ја претставува состојбата на соодветното хексадецимално копче. За бришење на низата за состојбата на пикселите од дисплејот, `display` се користи процедурата `ClearDisplay`, која покрај бришењето на низата, го поставува и знаменцето `disp_flag` кое ќе го објасниме во делот за исцртување на екранот од компјутерот.

1.2. Вчитување на ROM во меморијата на интерпретерот

Следен чекор е да ја вчитаме CHIP-8 програмата во меморијата на интерпретерот. За таа цел, името на ROM фајлот во кој се содржи програмата го земаме од DOS конзолата, имплементирано во процедурата `GetFileName`, со користење на прекин 21h со `(AH)=0Ah`, кој соодвествува на бафериран влез. Форматот на баферот е однапред зададен во описот на самиот прекин и се состои од:

- 1 бајт за максималниот број на карактери што може да се зачуваат во баферот
- 1 бајт за бројот на карактери што биле внесени од страна на корисникот, без притиснатиот Enter (знак Carriage Return - CR)

- n бајти, кој ги содржат внесените знаци, вклучувајќи го и CR

Истиот е дефиниран во податочниот сегмент со низата `in_buffer`. Следниот сегмент од процедурата `GetFileName` е задолжена за вчитување на името на фајлот и негово складирање во низата `fname`:

```
    mov     ah, 0Ah
    lea     dx, in_buffer
    int     21h
    lea     si, in_buffer + 1
    movzx   cx, byte ptr [si] ; zemi go brojot na vneseni znaci
                              ;od strana na korisnikot, bez CR
    mov     ax, ds
    mov     es, ax
    lea     si, in_buffer + 2
    lea     di, fname
    rep     movsb         ; kopiraj go vneseniot string vo fname
```

Потоа правиме отворање на фајлот со име `fname`, за читање, имплементирано во процедурата `OpenFile`, со користење на прекин 21h, со `(AH)=3Dh` и `(AL)=01h`. Доколку фајлот со зададеното име не може да се отвори за читање, на пример доколку истиот не постои во работниот директориум, прекинот го поставува знаменцето за пренос, па доколку е поставено го известуваме корисникот дека соодветниот фајл не може да се отвори и програмата завршува. Во спротивно, справувачот за фајлот го зачувуваме во `fhandle`.

Вчитувањето на фајлот во меморијата на интерпретерот се прави со процедурата `LoadMemory`, со следниот сегмент:

```
     call    FileSize     ; zemi ja goleminata na fajlot
     mov     cx, ax
     cmp     cx, mem_prog_data
     jg      file_size_error
     lea     dx, mem
     add     dx, 200h
     mov     ah, 3Fh
     int     21h ; zapisi ja sodrzinata od fajlot vo mem, na 200h
     jmp     lm_end
file_size_error:
     stc
```

Прво, со повикување на процедурата `FileSize` ја одредуваме големината на програмата што треба да ја вчитаме во меморијата на интерпретерот. Големината на датотеката ја одредуваме така што покажувачот за позиција во фајлот, со помош на прекинот 21h со `(AH)=42h`, `(AL)=02h`, `(CX)=(DX)=00h`, го поставуваме на крајот од

датотеката, а по извршувањето на прекинот, во AX ќе го имаме бројот на бајти од почетокот на фајлот, што всушност претставува големината на фајлот. Пред да се вратиме од процедурата FileSize, покажувачот за позиција во фајлот го поставуваме на почеток од фајлот. Споредуваме дали големината на ROM фајлот е соодветна. Доколку ROM фајлот е поголем од максимално дозволената големина за програма во интерпретерот, го поставуваме знаменцето за пренос, кое може да го провериме и да испишеме соодветна порака доколку големината на ROM фајлот не е соодветна. Во спротивно, ја запишуваме содржината на фајлот во меморијата на интерпретерот, на почетна адреса 200h, со помош на прекинот 21h со (AH)=3Fh, а во CX е големината на фајлот (бројот на бајти што треба да се исчитаат од фајлот). По вчитувањето на фајлот, истиот може да го затвориме со користење на процедурата CloseFile.

1.3. Иницијализација на генераторот на псевдослучајни броеви

За генерирање на псевдослучајни броеви, ќе користиме имплементација на поместувачки регистар со линеарна повратна врска (анг. Linear-Feedback Shift Register – LFSR). 32-битниот поместувачки регистар, дефиниран како низа од 4 бајти shift_reg го иницијализираме со почетна состојба на битовите со помош на процедурата InitRandom, со бројот на одбројувања на системскиот тајмер од полноќ, кој може да го добиеме во регистрите CX:DX со прекинот 1Ah со (AH)=00h. Алгоритмот подетално ќе го опишеме при описот на псевдомашинската инструкција од CHIP-8 која користи случаен број.

1.4. Иницијализација на видео режимот

Видео режимот што ќе го користиме за исцртување на екранот го поставуваме со процедурата InitVideo, која со користење на прекинот 10h со (AH)=00h и (AL)=13h, поставува графички видео режим, со резолуција 320x200 пиксели и 256 бои. Пред промена на видео режимот, процедурата го зачувува стариот видео режим во video_mode.

1.5. Иницијализација на тајмерите

Со процедурата TimersInit правиме иницијализација на системскиот тајмер и тајмерот 2, како и поставување на нов вектор на прекин за прекин 1Ch. Системскиот тајмер ќе го поставиме да одборојува со фреквенција од 60 Hz, а истиот ќе го користиме за ажурирање на двата регистри за тајмерите од CHIP-8 интерпретерот, како и за обезбедување

на фиксен број на 60 рамки што ќе се процесираат и исцртуваат во една секнуда. За таа цел, системскиот тајмер ќе го поставиме да работи во режим 2, со бинарно броење и во него ќе запишеме 1193182 / 60 = 19 886. Тајмерот 2 пак ќе ни служи за генерирање на поворка од квадратни импулси со фреквенција зададена во `beep_f` кој ќе ги користиме за генерирање на тон на звучникот. Во продолжение е даден сегментот од `TimersInit` со кој ги правиме подесувањата на тајмерите:

```
    ; za brojacot 1
    mov     dx, 43h
    mov     al, 34h
    out     dx, al
    mov     dx, 40h
    mov     ax, 19886 ; frekvencija na brojacot 0 = 60 Hz
    out     dx, al
    mov     al, ah
    out     dx, al
    ; za brojacot 2
    mov     dx, 43h
    mov     al, 0B6h
    out     dx, al
    mov     dx, 42h
    mov     ax, [beep_f] ; frekvencija na brojacot 2 = beep_f
    out     dx, al
    mov     al, ah
    out     dx, al
```

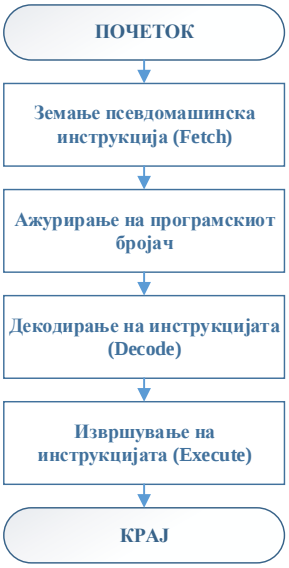
2. Главна јамка на интерпретерот

Главната јамка на интерпретерот е зададена со следниот код:

```
interpreter:                ; glavna jamka na interpreterot
    mov     ah, 00h
    int     1Ah
    mov     ax, dx
    sub     ax, [frame_last]
    cmp     ax, 1
    jb      interpreter
    movzx   cx, byte ptr [inst_pf]
instruction:
    call    ExecuteCycle    ; izvrsi edna instrukcija
    loop    instruction
    call    DrawScreen      ; iscrtaj go ekranot
    call    KeyState        ; zemi ja sostojbata na kopcinjata
    mov     word ptr [frame_last], dx
    ; vo slucaj da se pritisne ESC, running kje bide 0
    cmp     byte ptr [running], 1
    je      interpreter
```

Со едно извршување на оваа јамка всушност е испроцесирана една рамка. Во една секунда ќе се извршат приближно 60 рамки во секунда. За да обезбедиме 60 рамки во секунда, на почеток од јамката го земаме бројот на одбројувања на системскиот тајмер од полноќ. Понезначајните 16 бита ги одземаме со бројот на одборјувања на што сме го зачувале во претходната рамка, во `frame_last`, и се додека оваа разлика не стане 1 (системскиот тајмер одбројува на секои 1/60 s), ја повторуваме оваа процедура. Со ова воведуваме донцење, доколку претходната рамка се процесирала побрзо од 1/60 s. Потоа извршуваме неколку CHIP-8 псевдомашински инструкции, чиј број е зададен во `inst_pf`. Со пробување, за најголем дел од игрите вредност од 10 во `inst_pf` дава добри резултати. Вредноста на `inst_pf` може да се менува со копчињата **[** за намалување на `inst_pf` и со **]** за зголемување на `inst_pf`. Извршувањето на една инструкција се прави со повикување на процедурата `ExecuteCycle`.

2.1. Процедура `ExecuteCycle`



Извршувањето на псевдомашинска инструкција е претставено на блок дијаграмот погоре. Првиот чекор е земање на инструкција од меморијата на интерпретерот, а потоа ажурирање на програмскиот бројач. Тоа се прави со следниот код:

```
 lea     si, mem
add     si, [PC]
mov     ax, [si] ; Zemanje na instrukcija od memorijata
xchg    al, ah
mov     [opcode], ax
add     word ptr [PC], 2 ; Azuriranje na programskiot brojac
```

Бидејќи сите инструкции се со големина од 2 бајти, исчитуваме два бајти од мемориска адреса `mem + PC`. Бидејќи CHIP-8 користи big-endian формат, а x86 користи little-endian, со инструкцијата `XCHG` правиме замена на позначајниот и понезначајниот бајт. Инструкцијата ја зачувуваме во променливата `opcode`. На програмскиот бројач му додаваме 2, за да го содржи офсетот за следната инструкција што ќе треба да се изврши.

Потоа треба да се направи декодирање на инструкцијата. Најголем дел од CHIP-8 инструкциите може целосно да се декодираат само со користење на позначајни 4 бита. Бидејќи постојат вкупно 16 можности за првите 4 бита, поефикасно ќе биде наместо да низа од IF-ELSE искази, да користиме табела за скокови (анг. jump table). Табелата за скокови ја дефинираме во податочниот сегмент:

```
opcode_jmptable  dw      opcode_0uuu
                 dw      opcode_1nnn
                 dw      opcode_2nnn
                 dw      opcode_3xkk
                 dw      opcode_4xkk
                 dw      opcode_5xy0
                 dw      opcode_6xkk
                 dw      opcode_7xkk
                 dw      opcode_8xyu
                 dw      opcode_9xy0
                 dw      opcode_Annn
                 dw      opcode_Bnnn
                 dw      opcode_Cxkk
                 dw      opcode_Dxyn
                 dw      opcode_Exuu
                 dw      opcode_Fxuu
```

каде секој елемент од оваа табела претставува лабела од процедурата `ExecuteCycle`. За да лабелите бидат достапни и надвор од процедурата, MASM налага лабелите да се означат со две две точки, на пример `opcode_Bnnn::`. Табела на скокови користиме и за инструкциите каде првите 4 бита соодветсвуваат на 8h (`opcode_8xyu_jmptable`) и на Fh (`opcode_Fxuu_jmptable`), бидејќи истите се декодираат и со последните 4 бита и ги има повеќе на број. Во продолжение е даден кодот кој ја искористува табелата `opcode_jmptable`, за декодирање на првите 4 бита:

```
    and     ax, 0F000h
    shr     ax, 11
    lea     si, opcode_jmptable
    add     si, ax ; vo si e adresata od soodvetnata labela
    jmp     word ptr [si]
```

Следно ќе дадеме опис на некој од псевдомашинските инструкции (зададени со операцискиот код) и како истите ги извршуваме.

- **Операциски код 00EEh: враќање од субрутина**

Оваа инструкција го намалува покажувачот на врвот на стекот од интерпретерот, ја зема адресата што е зачувана на соодветната позиција во стекот и ја поставува таа адреса во програмскиот бројач. Во продолжение е даден кодот што ја извршува оваа инструкција:

```
opcode_00EE:      ; opcode 00EEh: Vrakjanje od subrutina
    sub     word ptr [stk_p], 2 ; namali go stack pokazuvacot
    lea     si, stk
    add     si, [stk_p]
    mov     ax, [si]  ; zemi ja adresata od vrvot na stekot
    mov     [PC], ax  ; i postavi ja vo programskiot brojac
    jmp     execute_cycle_end
```

`stk_p` всушност го содржи офсетот од `stk`, каде што може да се постави нова адреса во стекот. Поради тоа, го `stk_p` го намалуваме за 2 (`stk` е низа од зборови).

- **Операциски код 2NNNh: повикување на субрутина на адреса NNNh**

При извршување на оваа инструкција, прво ја зачувуваме адресата на следната инструкција што треба да се изврши на врвот на стекот од интерпретерот, го ажурираме покажувачот на врвот на стекот и во програмскиот бројач ја запишуваме адресата NNNh:

```
opcode_2nnn:: ; opcode 2NNNh:povikuvanje na subrutina na adresa NNNh
    lea     si, stk
    add     si, [stk_p]
    mov     ax, [PC]
    mov     [si], ax     ; zacuvaj go PC na stekot
    add     word ptr [stk_p], 2 ; azuriraj go pokazuvacot na stekot
    mov     ax, [opcode]
    and     ax, 0FFFh
    mov     [PC], ax  ; vcitaj go PC so adresata na subrutinata
    jmp     execute_cycle_end
```

- **Операциски код 5XY0h: скокни ја следната инструкција ако Vx == Vy**

Сите инструкции кај кои има скокање на следна инструкција ако е задоволен или не е задоволен некој услов, скокањето го правиме така што на програмскиот бројач додаваме уште 2. Во продолжение е даден кодот што ја извршува оваа инструкција:

```
opcode_5xy0::
    mov     ax, [opcode]
    mov     si, 00F0h
    and     si, ax
    shr     si, 4      ; vo si e y
    mov     di, 0F00h
```

```
and     di, ax
shr     di, 8       ; vo di e x
lea     bx, V
mov     dl, [bx][si]  ; vo dl e Vy
cmp     [bx][di], dl
jne     execute_cycle_end ; ako Vx!=Vy, dodaj na PC samo 2
add     word ptr [PC], 2  ; ako Vx==Vy, dodaj na PC uste 2
jmp     execute_cycle_end
```

За сите инструкции каде што има потреба до пристапување на регистар за општа намена, Vx и/или Vy од интерпретерот, користиме базно-индексирано мемориско адресирање: во `BX` ја поставуваме почетната адреса од низата `V`, додека во индексните регистри (`SI` и/или `DI`) го изолираме редниот број на регистарот, зададен во инструкцијата.

- **Операциски код 8XY4h: на Vx додади ја вредноста од Vy (Vx = Vx + Vy) ; VF=1, доколку имало пренос, инаку VF=0**

Во оваа инструкција интересно е да се посочи дека знаменцето за пренос што го користи x86 при операцијата собирање доволно е да се зачува во регистарот VF. Тоа го правиме со инструкцијата `setc` (во `BX` е почетната адреса на низата `V`):

```
setc byte ptr [bx + 0Fh]  ; VF = 1 ako ima prenos, inaku 0
```

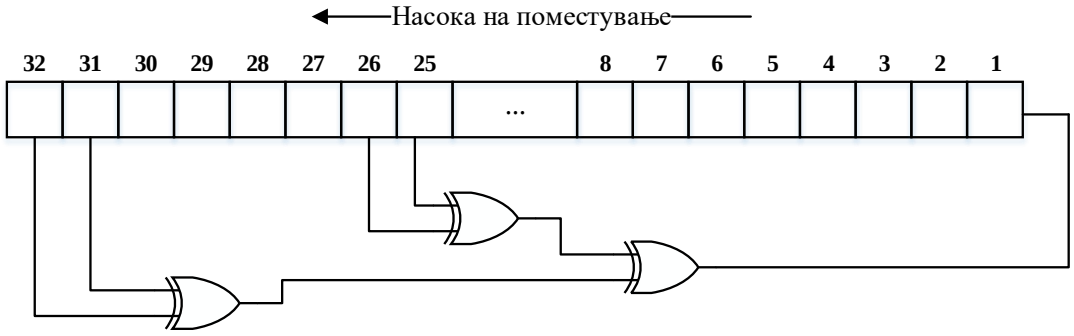
- **Операциски код 8XY5h: од Vx одземи ја вредноста на Vy (Vx = Vx - Vy) ; VF=0, доколку имало позајмување, инаку VF=1**

Во оваа инструкција интересно е да се посочи дека знаменцето за позајмување што го користи x86 при операцијата за одземање работи обрантно од начинот на позајмувањето што го користи инструкцијата 8XY5h. Поради тоа, VF го поставуваме со инструкцијата `setnc` (во `BX` е почетната адреса на низата `V`):

```
setnc byte ptr [bx + 0Fh]  ; VF = 0 ako imalo pozajmuvanje, inaku 0
```

- **Операциски код CXKKh: во Vx зачувај го резулатот од И операција на KKh со случаен број**

Во оваа иструкција ќе го опишеме алгоритмот за генерирање на псевдослучаен број користејќи поместувачки регистар со линеарна повратна врска. Овој алгоритам е имплементиран во процедурата `GetRandom`, која генерира псевдослучаен број со големина 1 бајт и го сместува во променливата `rand`. Шемата според која се имплементира алгоритмот е дадена во продолжение.



Изолирањето на 32-от, 31-от, 26-от и 25-от бит и примена на исклучиво ИЛИ операција, за да го добиеме новиот најнезначаен бит во `AL`, го правиме со следниот код:

```
    mov     al, [shift_reg + 3]
    mov     bl, al
    and     bl, 20h
    shr     bl, 5    ; vo bl e bit 30
    mov     bh, al
    and     bh, 80h
    shr     bh, 7    ; vo bh e bit 32
    mov     ah, al
    and     ah, 2h
    shr     ah, 1    ; vo ah e bit 26
    and     al, 1h   ; vo al e bit 25
    xor     al, ah
    xor     al, bl
    xor     al, bh  ; vo al e noviot bit
```

Потоа, знаменцето за пренос го поставуваме според вредноста во `AL`. Ова го правиме бидејќи ќе ја користиме `RCL` инструкцијата за поместување на битовите во `shift_reg`. Поместувањето го правиме со следниот код:

```
    cmp     al, 0 ; CF kje ima vrednost 0, nezavisno od (al)
    jz      shift
    stc           ; ako (al)=1, postavi go CF
shift: ; pomesti go pomestuvackiot registar za 1 vo levo, preku CF
    rcl     byte ptr [shift_reg], 1
    rcl     byte ptr [shift_reg + 1], 1
    rcl     byte ptr [shift_reg + 2], 1
    rcl     byte ptr [shift_reg + 3], 1
```

Псевдослучајниот број го генерираме така што правиме исклучиво ИЛИ на вториот и третиот бајт од `shift_reg`.

- **Операциски код DXYNh: исцртај графика на координати (Vx, Vy), со ширина 8 пиксели (фиксна) и висина Nh пиксели. Бајтите за графиката се исчитуваат од меморијата, со почетна адреса зачувана во I регистарот. Ако настане промена на некој пиксел од 1 на 0, VF=1, инаку VF=0**

Карактеристично за оваа инструкција е да опишеме ги ажурираме пикселите во низата за дисплејот, `display`. За пристап на пикселите во `display` ќе користиме базно-индексирано мемориско адресирање. Бидејќи низата `display` треба да ја пристапуваме како дводимензионална матрица, офсетот за пикселот на координати (x, y) ќе го пресметаме како x + 64y. Со следниот код, во `DI` го поставуваме офсетот што ќе одговара на почетните координати за исцртување на графиката, кои се исчитани од Vx и Vy и се зачувани во `DL` и `DH` соодветно:

```
 mov     al, dh
mov     bl, display_width
mul     bl
mov     di, ax
movzx   ax, dl
add     di, ax
```

Потоа, во јамка, исчитуваме бајт од меморијата на интерпретерот, од адреса зачувана во регистарот I, со што ги добиваме 8-те пиксели што ќе треба да ги исцртаме. Да нагласиме дека елементите во низата `display` одговараат на еден пиксел од екранот, за разлика од 8-те пикселите од редот на графиката, кои во меморијата се зачувани во 1 бајт. Потоа изолираме бит по бит (што одговара на пиксел) од исчитаниот бајт, и доколку вредноста на битот е нула, продолжуваме со следниот бит(пиксел). Причина за ова е тоа што исцртувањето на дисплејот се прави со исклучиво ИЛИ на претходната состојба на пикселот и новата состојба на пикселот. Доколку новиот пиксел е исклучен, исклучиво ИЛИ операција со нула не прави промена на другиот бит (пиксел). Доколку новата состојба на пикселот е вклучена, ја проверуваме старата состојба на соодвентиот пиксел од `display`. Доколку и старата состојба била 1, со исклучиво ИЛИ операцијата ќе се направи исклучување на пикселот, што всушност CHIP-8 го користи за детекција на судар. Во тој случај, VF се поставува на вредност 1. Да напомене дека доколку графиката треба да се исцрта на координати што се надвор од дисплејот (што е случај со повеќе игри), тогаш истите делови од графиката се исфрлаат (се отсекуваат). На крајот од инструкцијата се поставува `disp_flag` на 1, со што се означува дека е потребно исцртување на екранот од

компјутерот. Во продолжение е даден кодот што ги реализира наведените чекори. Во `SI` е сместена адресата од меморијата на интерпретерот, од каде треба да се исчитуваат бајтите за графиката, `CL` е бројач за тековниот пиксел од редот од графиката што тековно се процесира, во `CH` е бројот на редови(бајти) од кои се состои графиката, а во `BX` е почетната адреса на `display`.

```
opcode_Dxyn_row:
    mov     al, [si] ; 8-te pikseli od tekovniot red od sprite-ot
opcode_Dxyn_pixel:
    mov     dl, 80h
    shr     dl, cl
    and     dl, al  ; sporedi dali noviot piksel e aktiven
    jz      opcode_Dxyn_next_pixel
    cmp     byte ptr [bx][di], 1 ;dali stariot piksel bil aktiven
    jne     opcode_Dxyn_no_collision
    lea     bx, V    ; postavi go VF na 1, imalo kolizija
    mov     byte ptr [bx + 0Fh], 1
    lea     bx, display
opcode_Dxyn_no_collision:
    xor     byte ptr [bx][di], 1 ; invertiraj go pikselot

opcode_Dxyn_next_pixel:
    inc     cl
    cmp     cl, 8 ; dali sme gi pominale site pikseli od redot
    je      opcode_Dxyn_next_row ; ako da, odi na sleden red
    inc     di
    jmp     opcode_Dxyn_pixel

opcode_Dxyn_next_row:
    dec     ch
    cmp     ch, 0
    jz      opcode_Dxyn_end
    add     di, display_width - 7
    cmp     di, display_size ; ako di izlegol od granicite na
                             ; display, sprite-ot e otsecen
    jge     opcode_Dxyn_end
    inc     si
    xor     cl, cl
    jmp     opcode_Dxyn_row
```

- **Инструкции кои извршуваат некое дејство, во зависност од состојбата на копчињата (Операциски кодови EX9Eh, EXA1h, FX0Ah)**

Важно да се нагласи за инструкциите кои работат со состојбата на копчињата е доколку соодветниот услов е исполнет, да се направи ресетирање на состојбата на копчињата. Ако не се направи ресетирање на состојбата на копчињата, доколку повторно дојде на ред да се изврши иста таква инструкција, условот повторно може да биде исполнет,

иако корисникот не го притиснал копчето повторно. Ресетирањето на копчињата се прави со повикување на процедурата `ResetKeyState`.

2.2. Исцртување на екранот

Исцртувањето на екранот од компјутерот се прави со помош на процедурата `DrawScreen`, само доколку `disp_flag` е поставен на вредност 1. Бидејќи дисплејот на CHIP-8 има димензии 64x32 пиксели, а видео режимот на компјутерот го поставивме на 320x200 пиксели резолуција, ќе направиме скалирање на пикселите, така што еден пиксел од CHIP-8 ќе одговара на 5x5 пиксели од компјутерскиот екран. Бидејќи исцртувањето на пиксел по пиксел со прекин е прилично неефикасно, за исцртување на екранот од компјутерот ќе правиме директно запишување во меморијата на графичката картичка, во VGA режим, која е мапирана во меморијата на компјутерот почнувајќи на адреса A000h:0000h. Бидејќи и екранот, како и `display`, е организиран како матрица, пристап до пикселот со координати (x, y) од екранот го правиме со офсет x + y * 320 од A000h:0000h. Во продолжение е даден делот процедурата, што го прави исцртувањето на екранот, со забелешка дека во `ES` е зачувана вредност A000h:

```
    xor     dx, dx
    ; vo dl e tekovnata kolona, vo dh e tekovniot red od display
    lea     bx, display
    mov     si, dx              ; za zemanje piksel od display
    mov     di, 20
    mov     al, pixel_color
    mov     cx, screen_width    ; ramka okolu igrata
    rep     stosb
draw_column:
    mov     ah, pixel_size
    push    di ; zacuvaj do kade na ekranot sme stignale so iscrtuvanje
    xor     al, al
    cmp     byte ptr [bx][si], 0
    jz      draw_pixel
    mov     al, pixel_color
draw_pixel:                     ; iscrtuvanje na kvadratot
    mov     cx, pixel_size
    rep     stosb
    dec     ah
    cmp     ah, 0
    jz      draw_next_column
    add     di, screen_width - pixel_size  ; sleden red za VGA
    mov     cx, pixel_size
    jmp     draw_pixel
draw_next_column:
    inc     si
    inc     dl                  ; zgolemi ja kolonata
    cmp     dl, display_width ; dali sme pominale red od display
    je      draw_next_row
```

```
    pop     di
    add     di, pixel_size
    jmp     draw_column
draw_next_row:
    pop     cx          ; za da go otstranime posledno zacuvaniot di
                        ; (vekje di e postaven na tocnata lokacija)
    inc      dh     ; zgolemi go brojacot za redici vo display
    cmp     dh, display_height  ; dali sme pominale red od display
    je      draw_screen_reset_flag
    xor     dl, dl
    jmp     draw_column
draw_screen_reset_flag:
    mov     al, pixel_color
    mov     cx, screen_width       ; ramka okolu igrata
    rep     stosb
    mov     byte ptr [disp_flag], 0
draw_screen_end:
```

2.3. Ажурирање на состојбата на копчињата

Ажурирањето на состојбата на копчињата се прави со помош на процедурата `KeyState`. На почеток од оваа процедура, со прекинот 16h со `(AH)=1h`, проверуваме дали има притиснато копче. Доколку нема притиснато копче, знаменцето за нула ќе биде поставено и можеме веднаш да излеземе од процедурата. Доколку има притиснато копче од тастатурата, со прекинот 16h со `(AH)=0h` ќе го земеме ASCII кодот на притиснатото копче (прекинот ќе го врати ASCII кодот во `AL`). Потоа проверуваме со повеќе IF-ELSE услови, кое е притиснатото копче. Мапирањето на копчињата од хексадецималната тастатура на тастатурата од компјутерот веќе го прикажавме, и доколку некое копче е притиснато, во низата `keys`, со соодветен офсет за притиснатото копче, поставуваме 1. Покрај копчињата што се користат за маприање на хексадецималната тастатура, се проверуваат и следниве копчиња, зададени со нивното дејство:

ASCII код на копчето	Дејство
ESC (1Bh)	крај на интерпретерот
`	ресетирање на состојбата на интерпретерот
[	намалување на бројот на инструкции што се извршуваат во една рамка
]	зголемување на бројот на инструкции што се извршуваат во една рамка

Главната јамка се извршува сè додека `running` = 1 (односно се додека не се притисне копчето Escape на тастатурата).

3. Враќање на стариот видео режим и старата состојба на тајмерите

По притискање на копчето Escape, со што престанува извршувањето на главната јамка, пред да излеземе од програмата ги враќаме стариот видео режим, кој го зачувавме во `video_mode`, со повикување на процедурата `RestoreVideo`, и старата состојба на броење на тајмерите и стариот вектор за прекин 1Ch, со повикување на процедурата `TimersRestore`.

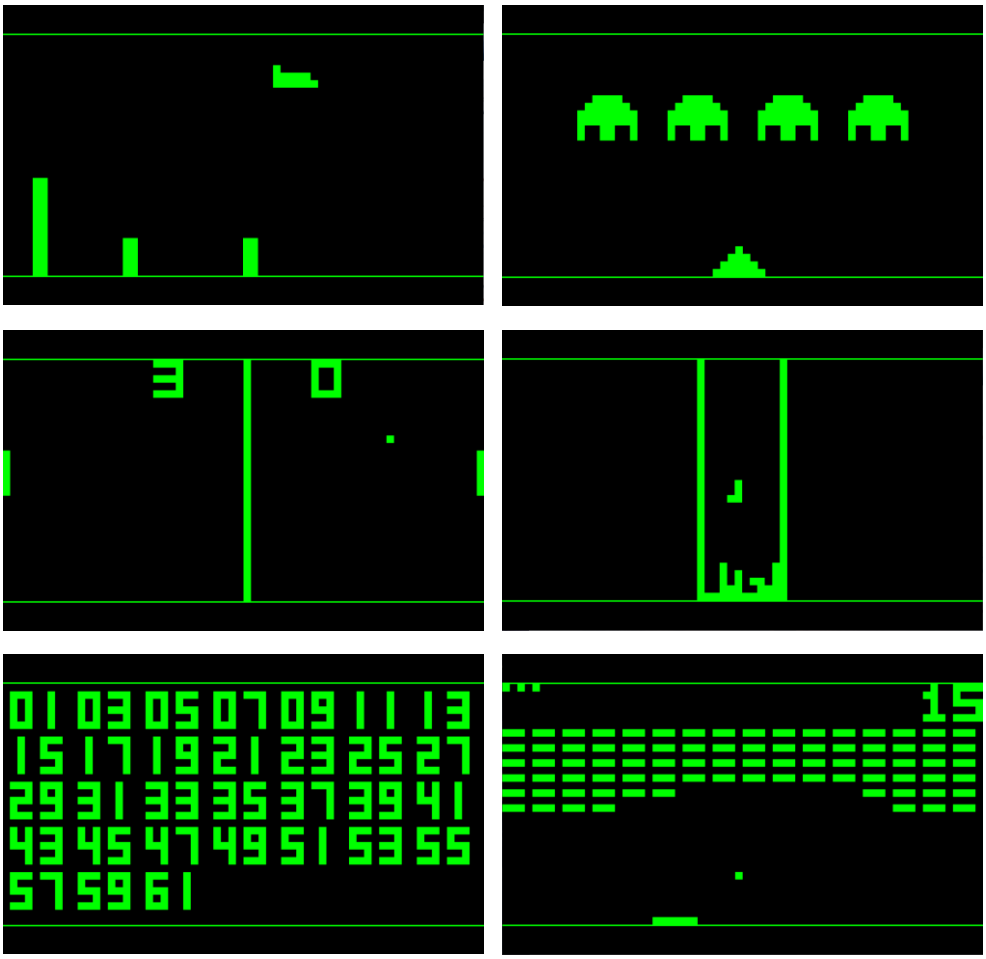
4. Процедура за опслужување на прекинот 1Ch

Процедурата за опслужување на прекинот 1Ch е `TimersInt`, која доколку регистрите поврзани со тајмерите `delay_t` и `sound_t` имаат ненулта вредност, ги намалува за еден. Покрај тоа, инструкцијата што го поставува `sound_t` (FX18h), поставува и знаменце `beep_flag`. Ова знаменце ќе го користиме како информација во `TimersInt` дали треба квадратните импулси што ги генерира тајмерот 2 да се пропуштат на звучникот од компјутерот. Пропуштањето на квадратните импулси и активирањето на звучникот се прави со запишување 1 на битовите 0 и 1 на портата 61h. Важно е на почеток од процедурата да се вчита `DS` со адресата од податочниот сегмент, бидејќи при тестирање во MS-DOS и FreeDOS, `DS` имаше променета вредност и не можеше да се пристапи до податочниот сегмент, најверојатно поради прекинот 8h. Во продолжение е дадено ажурирањето на `sound_t`:

```
sound_timer_check:
    cmp     byte ptr [beep_flag], 1
    jne     update_sound_timer
    mov     byte ptr [beep_flag], 0
beep_on:            ; zapocni go beep zvukot
    mov     dx, 61h
    in      al, dx   ; iscitaj ja portata 61h
    or      al, 03h  ; postavi gi bitovite 0 i 1 na vrednost 1
    out     dx, al   ; pocetok na beep zvukot
update_sound_timer:
    cmp     byte ptr [sound_t], 0
    je      timers_int_end
    dec     byte ptr [sound_t]
    cmp     byte ptr [sound_t], 0
    jne     timers_int_end
beep_off:
    mov     dx, 61h
    in      al, dx
    and     al, 0FCh
    out     dx, al
```

Заклучок

Во продолжение се дадени неколку слики од работата на интерпретерот, при испробување некој класични игри напишани во CHIP-8. Почнувајќи од горе лево, ROM-овите за тестирање се: BLITZ, INVADERS, PONG2, TETRIS, GUESS и BRIX.



Пишувањето на интерпретер или емулатор е интересен предизвик, од кој може да се научи многу за работата на компјутерските системи, но исто така и да се размислува за оптимизации на кодот, со цел платформата на која го емулираме системот да не внесува преголема латентност при извршувањето на инструкциите. Користењето на асемблерски јазик за пишување на интерпретер/емулатор може до некој степен да обезбеди поголема ефикасност на кодот за сметка на поголемото вложено време за развој на истиот. Краен заклучок е дека асемблерски код треба да се користи само за критични делови од интерпретерот/емулаторот, за кои би ни била потребна најголема оптималност, а компајлерот не може истата да ја постигне.

Користена литература

[1] J. Weisbecker, "An Easy Programming System," *Byte Magazine,* том 03, бр. 12, pp. 108-122, 12 1978.

[2] "Cowgod's Chip-8 Technical Reference v1.0," 30 Август 1997: http://devernay.free.fr/hacks/chip8/C8TECH10.HTM. [Пристапено на 8 Јуни 2021].

[3] З. Ивановски, *Предавања и аудиториски вежби по Микропроцесорска електроника,* 2021.