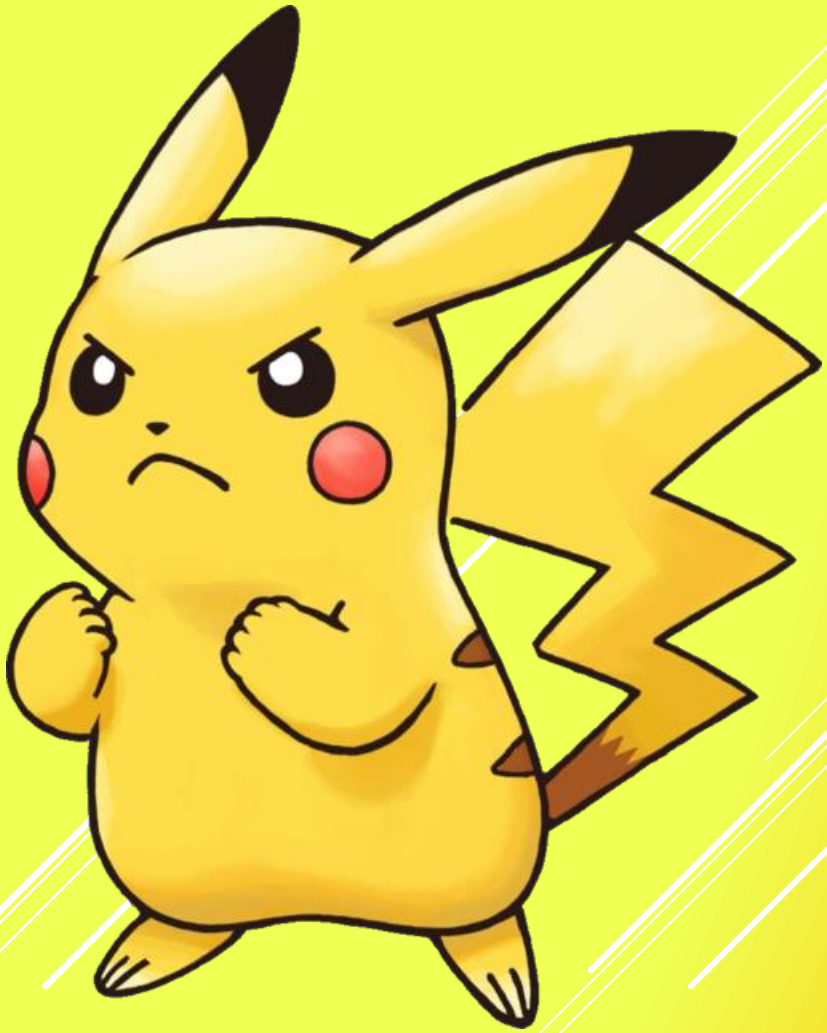


PIKACHU, SCELGO TE!

INTRODUZIONE

- In questa lezione avrei dovuto parlare interamente di questo progetto:

- Nato tra le pareti di questo dipartimento nel 1996 (circa)
- In principio: emulatore per alcuni sistemi arcade
- Ora: progetto mastodontico, emulatore per coin-op, console (MESS), microcomputer, calcolatrici (e, spulciando nel codice sorgente ho scoperto una primitiva emulazione di alcuni strumenti musicali: Alesis SQ-6, Korg M1, Roland TB-303 e TR-606...)

INTRODUZIONE

- In questa lezione avrei dovuto[...] questo progetto:

- Nato tra le pareti di questo dipa[...]rca)
- In principio: emulatore per alc[...]
- Ora: progetto mastodontico[...]onsole (MESS), microcomputer, calcolatrici (e, spulciando n[...]rto una primitiva emulazione di alcuni strumenti musical[...] Korg [...]B-303 e TR-606...)

INTRODUZIONE

- Perchè parlare di MAME in un corso di C++ per game design?

- Scritto originariamente in C, ma gradualmente vi è stato un passaggio C++
- Codice pulito, ben commentato, chiaro e sfrutta caratteristiche chiave di C++: emulazione di parti cardine di un sistema ben si adatta a paradigma programmazione ad oggetti!
- Esempio di progetto collaborativo, open source, coordinato e ben organizzato portato avanti da più di venti anni – e continuamente aggiornato...

- Documentazione è un po'... Meh...

INTRODUZIONE

- Progetto ha dimensioni mastodontiche!

```
C:\Users\User\Desktop\VP8 H.264>cloc-1.58.exe ..\mame-master\src
   10652 text files.
   10562 unique files.
    1437 files ignored.

http://cloc.sourceforge.net v 1.58  T=58.0 s (163.4 files/s, 72658.5 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
C++                           5735         633486         599896        2466065
C/C++ Header                  3570          87485          67111         338176
Lisp                           134           3016              0          11020
Objective C++                   19            860            259           3992
Python                           6            272            130           1546
C                                7              0             14            270
make                             4             53             43            184
Javascript                       1             23             49            141
Bourne Shell                     2             17             14             58
CSS                              1              0              0             14
-------------------------------------------------------------------------------
SUM:                          9479         725212         667516        2821466
-------------------------------------------------------------------------------
```

INTRODUZIONE

- Ma cos'è un emulatore?
- Quali sono le sue funzioni e quali le caratteristiche?
- In cosa differisce da un simulatore?

INTRODUZIONE

- Ma cos'è un emulatore?
- Quali sono le sue funzioni e quali le caratteristiche?
- In cosa differisce da un simulatore?

- Obiettivo: eseguire un programma su un'architettura diversa da quella per cui è stato originariamente scritto, senza modificarne il codice (o nel bytecode compilato) originale

- Per il MAME, sono le ROM di un determinato programma

EMULAZIONE DI UN SISTEMA

- Coin-op non è nient'altro che un computer come tanti...
- Generalmente composto dagli stessi componenti chiave con cui erano costruiti i computer coevi.
- Elenchiamoli!

EMULAZIONE DI UN SISTEMA

- Coin-op non è nient'altro che un computer come tanti...
- Generalmente composto dagli stessi componenti chiave con cui erano costruiti i computer coevi.
- Elenchiamoli!
- Processore (CPU)
- Memoria ROM
- Memoria RAM
- Chip sonoro
- (Chip video)
- Monitor
- Sistema di I/O
- (Chip custom per protezione, DSP, ecc...)

EMULAZIONE DI UN SISTEMA

- Come fare per ricostruire in sw una intera architettura così complessa?

- Coin-op: sistema eterogeneo in cui diverse periferiche (device) organizzate in maniera gerarchica parlano tra loro scambiandosi informazioni o comandi.

- OOP:
  - classi modellizzano un'idea / concetto / funzionalità ed espongono un'interfaccia verso altre classi per lo scambio di informazioni o comandi (incapsulazione)
  - OOP prevede organizzazione di gerarchia di classi (ereditarietà)

EMULAZIONE DI UN SISTEMA

- IDEA! (Ma molto ad alto livello...)
- Scrivo una classe per ogni periferca che compone il sistema che voglio emulare:
  - processore (CPU)
  - memoria ROM
  - memoria RAM
  - chip sonoro
  - (chip video)
  - monitor
  - sistema di I/O
  - (chip custom per protezione, DSP, ecc...)
- Faccio "dialogare le parti"
- Eseguo la ROM su questo sistema sw

EMULAZIONE DI UN SISTEMA

- IDEA! (Ma molto ad alto livello...)
- Scrivo una classe per ogni periferca che compone il sistema che voglio emulare:
  - processore (CPU)
  - memoria ROM
  - memoria RAM
  - chip sonoro
  - (chip video)
  - monitor
  - sistema di I/O
  - (chip custom per protezione, DSP, ecc...)
- Faccio "dialogare le parti"
- Eseguo la ROM su questo sistema sw

- MAME fa (più o meno) questo!

EMULAZIONE DI UN SISTEMA

- - processore (CPU)
  - **memoria ROM**
  - **memoria RAM**
  - chip sonoro
  - (chip video)
  - monitor
  - sistema di I/O
  - (chip custom per protezione, DSP, ecc...)

- Strutture più facili da emulare: posso modellizzarle come array monodimensionali e copiare ROM nelle locazioni adeguate

EMULAZIONE DI UN SISTEMA

- - processore (CPU)
  - memoria ROM
  - memoria RAM
  - chip sonoro
  - (**chip video**)
  - **monitor**
  - sistema di I/O
  - (chip custom per protezione, DSP, ecc...)

- Di solito non serve! Area di memori RAM era usata come framebuffer.
  In verità, vengono emulati anche quelli...

EMULAZIONE DI UN SISTEMA

- - processore (CPU)
  - memoria ROM
  - memoria RAM
  - chip sonoro
  - (chip video)
  - monitor
  - **sistema di I/O**
  - (chip custom per protezione, DSP, ecc...)

- Nei coin-op sono mappati in locazioni predefinite nello spazio di memoria RAM

EMULAZIONE DI UN SISTEMA

- - processore (CPU)
  - memoria ROM
  - memoria RAM
  - chip sonoro
  - (chip video)
  - monitor
  - sistema di I/O
  - (**chip custom per protezione**, **DSP**, **ecc...**)

- Eventuali classi ad-hoc

EMULAZIONE DI UN SISTEMA

- - **processore (CPU)**
  - memoria ROM
  - memoria RAM
  - **chip sonoro**
  - (chip video)
  - monitor
  - sistema di I/O
  - (chip custom per protezione, DSP, ecc...)

- Super! E queste?!?!?!?

EMULAZIONE DI UN SISTEMA

- - processore (CPU)
  - memoria ROM
  - memoria RAM
  - chip sonoro
  - (chip video)
  - monitor
  - sistema di I/O
  - (chip custom per protezione, DSP, ecc...)

- Digressione, prima di proseguire...
- Amici del PONG, perché Pong è stato inserito solo di recente nella lista dei giochi emulati?

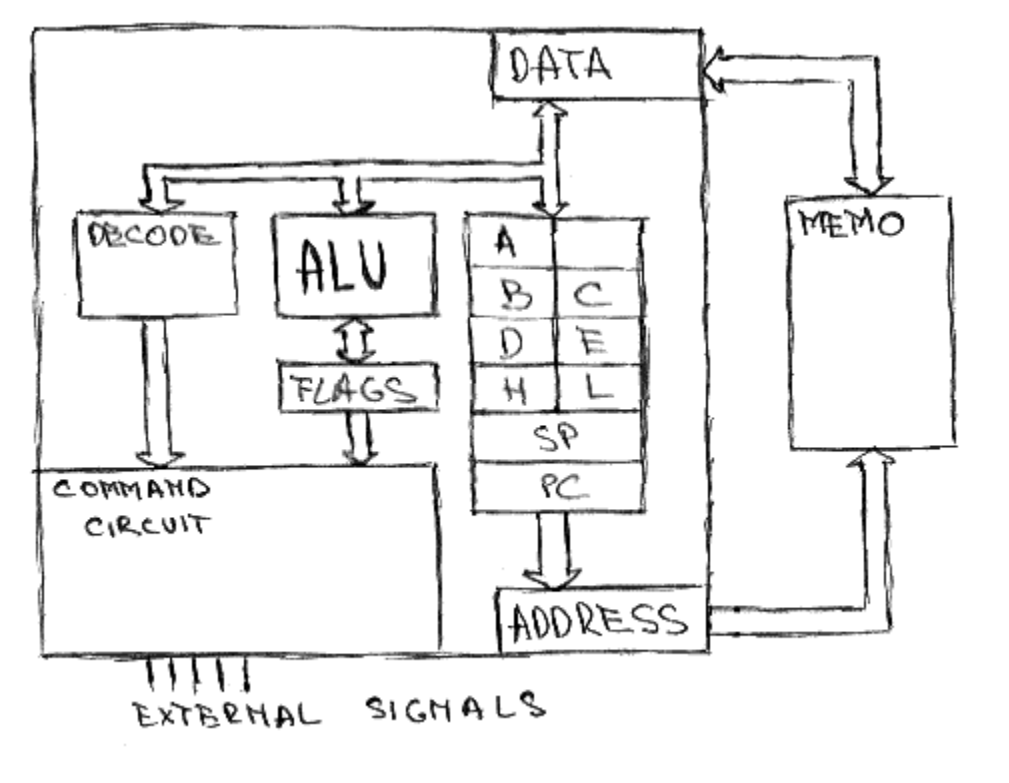
EMULAZIONE DI UN SISTEMA

- ~~- processore (CPU)~~
  ~~- memoria ROM~~
  ~~- memoria RAM~~
  ~~- chip sonoro~~
  ~~- (chip video)~~
  - monitor
  - sistema di I/O
  ~~- (chip custom per protezione, DSP, ecc...)~~

- Digressione, prima di proseguire...
- Amici del PONG, perché Pong è stato inserito solo di recente nella lista dei giochi emulati?
- Pong originale costruito con logiche discrete (no CPU, ROM, e altre diavolerie)
  In ultime release di MAME: "emulazione di circuiti"

EMULARE UNA CPU

- Emulare una CPU non è poi così difficile!
- Pensiamo a come è fatta e come lavora...

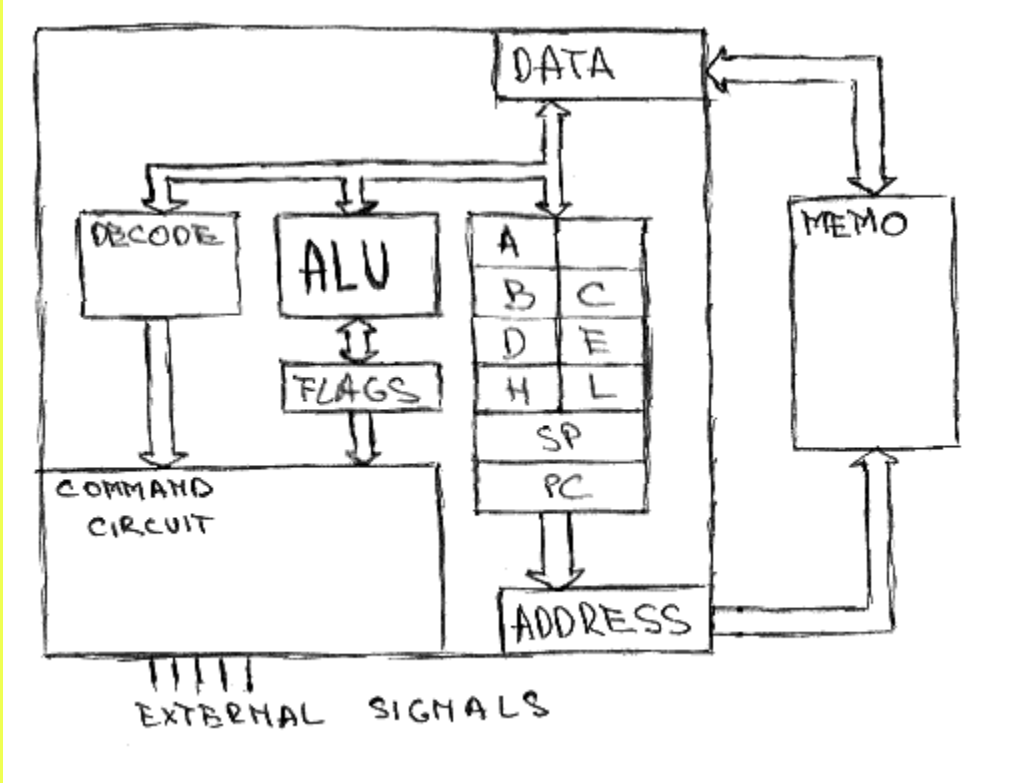


EMULARE UNA CPU

- Emulare una CPU non è poi così difficile!
- Pensiamo a come è fatta e come lavora...

- Input: assembly bytecode

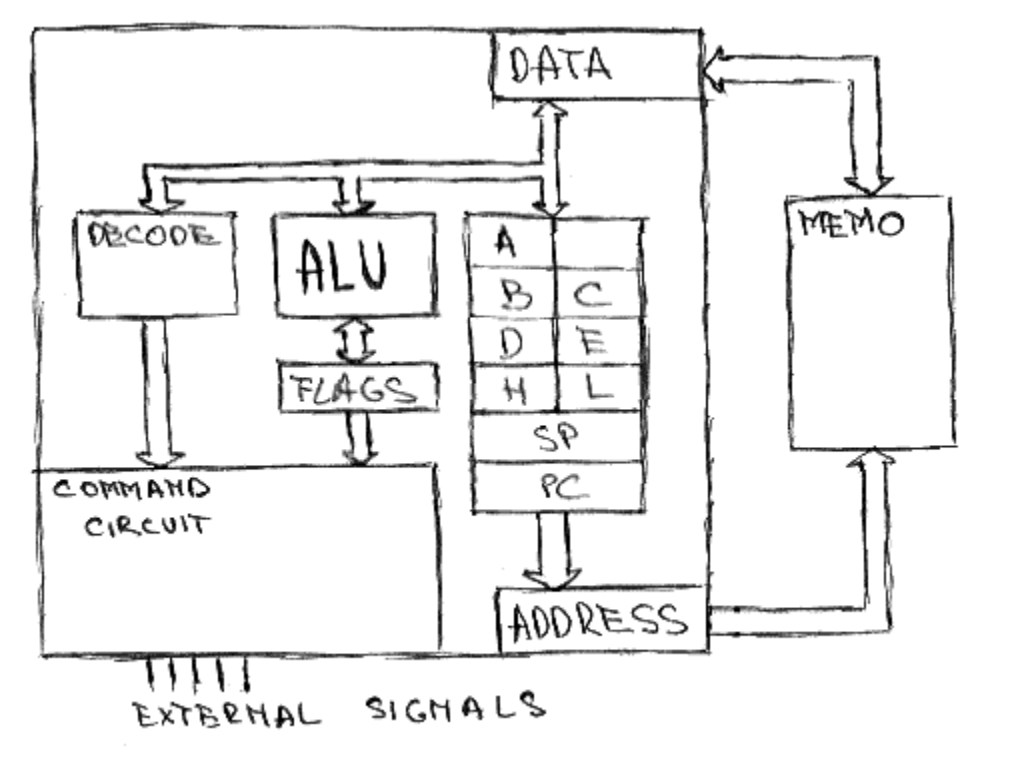
```
 1   1 0000                        ORG    $868
 2   2 0868 0000 0003 N:           EQU    3
 3   3 0868           RESULT:      DS.B 2
 4   4 086A           Z:           DS.B 50
 5   5 089C CE086A                 LDX    #Z          ; Point X to Z
 6   6 089F CD0003                 LDY    #N          ; get count
 7   7 08A2 EC31                   LDD    2,X+        ; Z(0) into D
 8   8 08A4 181A31    LOOP:        EMAXD  2,X+        ; D- Z(i)
 9   9 08A7 0436FA                 DBNE   Y,LOOP      ; Another number?
10  10 08AA 7C0868                 STD    RESULT      ; Store result
11  11 08AD 00                     BGND               ; Halt
```



EMULARE UNA CPU

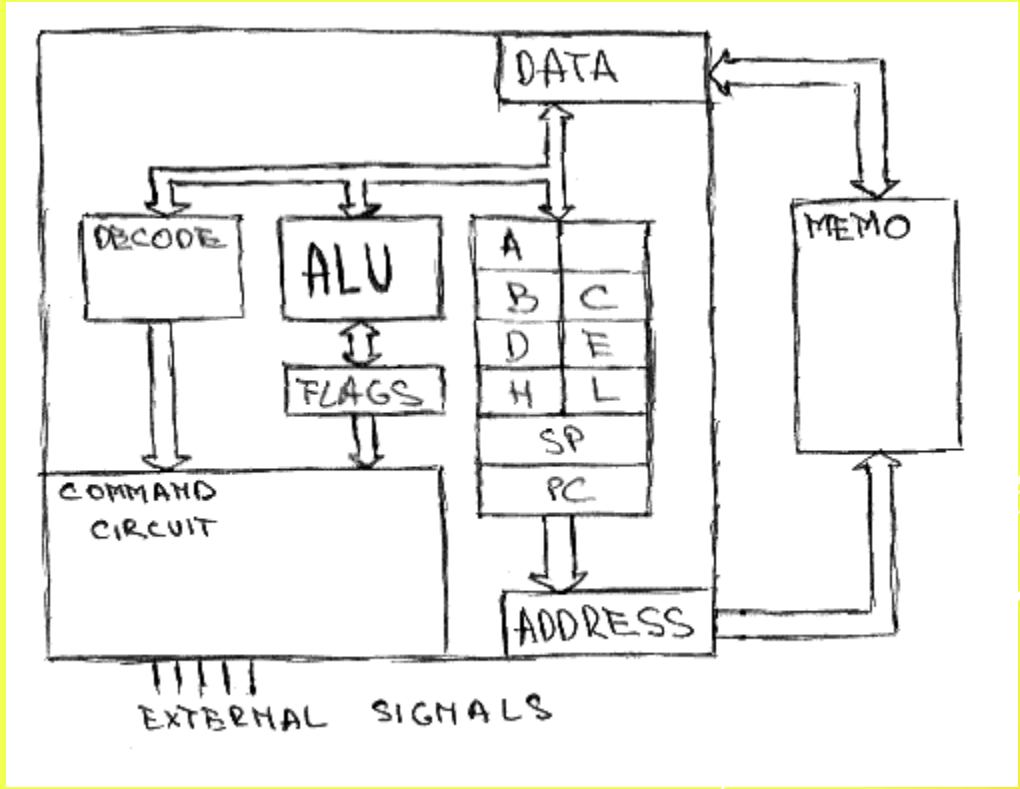
- Emulare una CPU non è poi così difficile!
- Pensiamo a come è fatta e come lavora...

- Esecuzione delle istruzioni in 4 step:
  - fetch: recupera l'istruzione da eseguire (e gli eventuali operandi) dalla ROM
  - decode: identifica l'istruzione da eseguire
  - execute: esegue l'istruzione
  - store: salva i risultati in un registo o in RAM



EMULARE UNA CPU

- Emulare una CPU non è poi così difficile!
- Pensiamo a come è fatta e come lavora...



- Esempi di opcode:

```
00h LD destination,source = load source with destination
40h ADD A,source          = add value in source to register A (A=A+source)
48h ADC A,source          = add with carry (A=A+source+cy)
50h SUB A,source          = substract source from A (A=A-source)
58h SBC A,source          = substract with carry (A=A-source-cy)
60h CMP A,source          = sets flags as SUB but leaves register A intact
68h AND A,source          = bitwise AND (A=A AND source)
70h OR A,source           = bitwise OR (A=A OR source)
78h XOR A,source          = bitwise XOR (A=A XOR source)
80h TEST A,source         = sets flags as AND but leaves register A intact
88h LD dest,nn            = load destination with an immediate value
90h INC destination       = increment destination (dest=dest+1)
98h DEC destination       = decrement destination (dest=dest-1)
a0h SHL destination       = shift left bits of destination (dest=dest*2)
a8h SHR destination       = shift right bits of destination (dest=dest/2)
b0h ROL destination       = rotate left bits of destination (wrap around)
b8h ROR destination       = rotate right bits of destination (-~-)
c0h RLC destination       = rotate left through carry (Cy<-dest<-Cy)
c8h RRC destination       = rotate right through carry (Cy->dest->Cy)
d0h PUSH reg16            = save reg16 on the stack (SP=SP-2;[SP]=reg16)
d4h POP reg16             = load reg16 from the stack (reg16=[SP];SP=SP+2)
d8h <16 bit operations>   = few operations on 16 bit registers
e0h <16 bit operations>   = few operations on 16 bit registers
e8h <special>             = instructions to help emulation
f0h CALL condition,adr    = value of PC is saved on the stack, JMP adr
f8h JMP condition,adr     = if condition is true, PC takes the value of ADR
```

EMULARE UNA CPU

- Proviamo a scrivere una classe che modellizza una CPU e il suo funzionamento!
- CHIP-8 cpu:
  - 35 opcode tutti lunghi 2 byte
  - 4k memoria in totale
  - 16 registri a 8 bit (15 general purpose + carry flag)
  - index register (I): registro 16 bit
  - program counter (PC): registro 16 bit
  - stack 16 livelli; stack pointer (SP): registro 16 bit
  - "schermo": 64 x 32 pixel a due colori (bianco o nero)

EMULARE UNA CPU

- Proviamo a scrivere una classe che modellizza una CPU e il suo funzionamento!
- CHIP-8 cpu:
  - 35 opcode tutti lunghi 2 byte

```
unsigned short opcode;
```

  - 4k memoria in totale

```
unsigned char memory[4096];
```

  - 16 registri a 8 bit (15 general purpose + carry flag)

```
unsigned char V[16];
```

  - index register (I): registro 16 bit

```
unsigned short I;
```

  - program counter (PC): registro 16 bit

```
unsigned short pc;
```

  - stack 16 livelli; stack pointer (SP): registro 16 bit

```
unsigned short stack[16], sp;
```

  - "schermo": 64 x 32 pixel a due colori (bianco o nero)

```
unsigned char gfx[64 * 32];
```

EMULARE UNA CPU

- Proviamo a sc onamento!

```
class chip8 {
    public:
        chip8();
        ~chip8();

        bool drawFlag;

        void emulateCycle();
        void debugRender();
        bool loadApplication(const char * filename);

// Chip8
        unsigned char  gfx[64 * 32];    // Total amount of pixels: 2048
        unsigned char  key[16];

    private:
        unsigned short pc;              // Program counter
        unsigned short opcode;          // Current opcode
        unsigned short I;               // Index register
        unsigned short sp;              // Stack pointer

        unsigned char  V[16];           // V-regs (V0-VF)
        unsigned short stack[16];       // Stack (16 levels)
        unsigned char  memory[4096];    // Memory (size = 4k)

        unsigned char  delay_timer;     // Delay timer
        unsigned char  sound_timer;     // Sound timer

        void init();
};
```

EMULARE UNA CPU

- Proviamo a scrivere una classe che modellizza una CPU e il suo funzionamento!
- Dimenticavo...

- Memory map:
- `0x000-0x1FF - Chip 8 interpreter (contains font set in emu)`
- `0x050-0x0A0 - Used for the built in 4x5 pixel font set (0-F)`
- `0x200-0xFFF - Program ROM and work RAM`

EMULARE UNA CPU

- Proviamo a scrivere una classe che modellizza una CPU e il suo funzionamento!
- Ora modelliziamo il ciclo di esecuzione: fetch, decode, execute, store
- ```
  void chip8::emulateCycle(){
    // Fetch Opcode
    // Decode Opcode
    // Execute Opcode

    // Update timers
  }
  ```
  ```

EMULARE UNA CPU

- Proviamo a scrivere una classe che modellizza una CPU e il suo funzionamento!
- Ora modelliziamo il ciclo di esecuzione: fetch, decode, execute, store
- ```
  void chip8::emulateCycle(){
    // Fetch Opcode
          opcode = memory[pc] << 8 | memory[pc + 1];
    // Decode Opcode
          switch( opcode ) case: [...]
    // Execute Opcode
          [...]
    // Update timers
  }
  ```
  ```

EMULARE UNA CPU

- Proviamo a scrivere una classe che modellizza una CPU e il suo funzionamento!
- Decode e execute:

```
switch(opcode & 0xF000)
    {
        case 0x1000: // 0x1NNN: Jumps to address NNN
            pc = opcode & 0x0FFF;
        break;

        [...]

        case 0x7000: // 0x7XNN: Adds NN to VX.
            V[(opcode & 0x0F00) >> 8] += opcode & 0x00FF;
            pc += 2;
        break;

        [...]

        default:
            printf ("Unknown opcode: 0x%X\n", opcode);
    }
```

EMULARE UN

- Proviamo a scrivere
- Manca solo init():

```
void chip8::init()
{
    pc      = 0x200;        // Program counter starts at 0x200 (Start adress program)
    opcode  = 0;            // Reset current opcode
    I       = 0;            // Reset index register
    sp      = 0;            // Reset stack pointer

    // Clear display
    for(int i = 0; i < 2048; ++i)
        gfx[i] = 0;

    // Clear stack
    for(int i = 0; i < 16; ++i)
        stack[i] = 0;

    for(int i = 0; i < 16; ++i)
        key[i] = V[i] = 0;

    // Clear memory
    for(int i = 0; i < 4096; ++i)
        memory[i] = 0;

    // Load fontset
    for(int i = 0; i < 80; ++i)
        memory[i] = chip8_fontset[i];

    // Reset timers
    delay_timer = 0;
    sound_timer = 0;

    // Clear screen once
    drawFlag = true;

    srand (time(NULL));
}
```

EMULARE UNA CPU

- Proviamo a scrivere una classe che modellizza una CPU e il suo funzionamento!
- E, ovviamente, il main:

```
int main(int argc, char **argv)
{
  // Set up render system and register input callbacks
  setupGraphics();
  setupInput();

  // Initialize the Chip8 system and load the game into the memory
  myChip8.initialize();
  myChip8.loadGame("pong");

  // Emulation loop
  for(;;)
  {
    // Emulate one cycle
    myChip8.emulateCycle();

    // If the draw flag is set, update the screen
    if(myChip8.drawFlag)
      drawGraphics();

    // Store key press state (Press and Release)
    myChip8.setKeys();
  }

  return 0;
}
```

TORNIAMO A MAME

- Torniamo al MAME!

~~TORNIAMO A MAME~~

- ~~Torniamo al MAME!~~
- Meglio di no.
  Concentriamoci su un solo sistema più semplice

~~TORNIAMO A MAME~~

- ~~Torniamo al MAME!~~
- Meglio di no.
  Concentriamoci su un solo sistema più semplice

EMULARE UN SISTEMA COMPLETO

- ~~Torniamo al MAME!~~
- Meglio di no.
  Concentriamoci su un solo sistema più semplice

EMULARE UN SISTEMA COMPLETO

- Nintendo Gameboy:
- CPU: Sharp LR35902 (core Zilog Z80 / Intel 8080)
- RAM: 8 kB (max 32 kB)
- Video RAM: 8 kB
- ROM: 256 b ("kernel") + cartucce
- Suono: generato internamente dalla CPU
- Display: 160 x 144 pixel; colori: 4 toni di grigio, 2-bit

EMULARE UN SISTEMA COMPLETO

- Nintendo Gameboy:
- CPU: Sharp LR35902 (core Zilog Z80 / Intel 8080)
- RAM: 8 kB (max 32 kB)
- Video RAM: 8 kB
- ROM: 256 b ("kernel") + cartucce
- Suono: generato internamente dalla CPU
- Display: 160 x 144 pixel; colori: 4 toni di grigio, 2-bit

- Possiamo usa quanto visto per emulare questo sistema?
  C++ può esserci utile?
- https://github.com/drhelius/Gearboy

EMULARE UN SISTEMA COMPLETO

- Main loop:
  progetto usa librerie QT per interfaccia e gestione thread
- rendering affidato a OpenGL
- m_pFrameBuffer è array di quad di unsigned char [160 x 144]

```
void RenderThread::run()
{
    m_pGLFrame->makeCurrent();
    Init();

    while (m_bDoRendering) {
        m_pGLFrame->makeCurrent();

        if (!m_bPaused) {
            m_pEmulator->RunToVBlank(m_pFrameBuffer);

            if (m_bResizeEvent) {
                m_bResizeEvent = false;
                m_bFirstFrame = true;
            }

            if (m_bMixFrames && !m_pEmulator->IsCGBRom())
                RenderMixFrames();
            else
                RenderFrame();
        }
        m_pGLFrame->swapBuffers();
    }

    SafeDeleteArray(m_pFrameBuffer);
    glDeleteTextures(1, &m_AccumulationTexture);
    glDeleteTextures(1, &m_GBTexture);
    glDeleteFramebuffers(1, &m_AccumulationFramebuffer);
}
```

EMULARE UN SISTEMA COMPLETO

- Classe principale:

```
class GearboyCore
{
public:
    GearboyCore();
    ~GearboyCore();
    void Init();
    void RunToVBlank(GB_Color* pFrameBuffer);
    bool LoadROM(const char* szFilePath, bool forceDMG);
    Memory* GetMemory();
    Cartridge* GetCartridge();
    void KeyPressed(Gameboy_Keys key);
    void KeyReleased(Gameboy_Keys key);
    void Pause(bool paused);
    bool IsPaused();
    void ResetROM(bool forceDMG);
    void EnableSound(bool enabled);
    void ResetSound(bool soft = false);
    void SetSoundSampleRate(int rate);
    void SetDMGPalette(GB_Color& color1, GB_Color& color2, GB_Color& color
    void SaveRam();
    void SaveRam(const char* szPath);
    void LoadRam();
    void LoadRam(const char* szPath);
    void SetRamModificationCallback(RamChangedCallback callback);
```

```
private:
    void InitDMGPalette();
    void InitMemoryRules();
    bool AddMemoryRules();
    void Reset(bool bCGB);
    void RenderDMGFrame(GB_Color* pFrameBuffer) const;

private:
    Memory* m_pMemory;
    Processor* m_pProcessor;
    Video* m_pVideo;
    Audio* m_pAudio;
    Input* m_pInput;
    Cartridge* m_pCartridge;
    CommonMemoryRule* m_pCommonMemoryRule;
    IORegistersMemoryRule* m_pIORegistersMemoryRule;
    RomOnlyMemoryRule* m_pRomOnlyMemoryRule;
    MBC1MemoryRule* m_pMBC1MemoryRule;
    MBC2MemoryRule* m_pMBC2MemoryRule;
    MBC3MemoryRule* m_pMBC3MemoryRule;
    MBC5MemoryRule* m_pMBC5MemoryRule;
    MultiMBC1MemoryRule* m_pMultiMBC1MemoryRule;
    bool m_bCGB;
    bool m_bPaused;
    GB_Color m_DMGPalette[4];
    bool m_bForceDMG;
    int m_bRTCUpdateCount;
    bool m_bDuringBootROM;
    bool m_bLoadRamPending;
    char m_szLoadRamPendingPath[512];
    RamChangedCallback m_pRamChangedCallback;
};
```

EMULARE UN SISTEMA COMPLETO

- Vediamo velocemente le componenti del sistema: memoria

```
void Memory::Init()
{
    m_pMap = new u8[65536];
    m_pWRAMBanks = new u8[0x8000];
    m_pLCDRAMBank1 = new u8[0x2000];
    m_pDisassembledMap = new stDisassemble[65536];
    Reset(false, false);
}
```

- Memoria di sistema in memory.m_pMap
- Funzioni per accesso a memoria (m_pMap è varibile privata)

- Gestione dei chip per bankswitch tramite classi dedicate MBCxMemoryRule.h e .cpp

EMULARE UN SISTEMA COMPLETO

- Vediamo velocemente le componenti del sistema: memoria
- Memory.h contiene anche questo!

```
const u8 kInitialValuesForFFXX[256] = {
    0xCF, 0x00, 0x7E, 0xFF, 0xD3, 0x00, 0x00, 0xF8, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xE1,
    0x80, 0xBF, 0xF3, 0xFF, 0xBF, 0xFF, 0x3F, 0x00, 0xFF, 0xBF, 0x7F, 0xFF, 0x9F, 0xFF, 0xBF, 0xFF,
    0xFF, 0x00, 0x00, 0xBF, 0x77, 0xF3, 0xF1, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
    0x71, 0x72, 0xD5, 0x91, 0x58, 0xBB, 0x2A, 0xFA, 0xCF, 0x3C, 0x54, 0x75, 0x48, 0xCF, 0x8F, 0xD9,
    0x91, 0x80, 0x00, 0x00, 0x00, 0x00, 0xFF, 0xFC, 0xFF, 0xFF, 0x00, 0x00, 0xFF, 0xFF, 0xFF, 0xFF,
    0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
    0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
    0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
    0x2B, 0x0B, 0x64, 0x2F, 0xAF, 0x15, 0x60, 0x6D, 0x61, 0x4E, 0xAC, 0x45, 0x0F, 0xDA, 0x92, 0xF3,
    0x83, 0x38, 0xE4, 0x4E, 0xA7, 0x6C, 0x38, 0x58, 0xBE, 0xEA, 0xE5, 0x81, 0xB4, 0xCB, 0xBF, 0x7B,
    0x59, 0xAD, 0x50, 0x13, 0x5E, 0xF6, 0xB3, 0xC1, 0xDC, 0xDF, 0x9E, 0x68, 0xD7, 0x59, 0x26, 0xF3,
    0x62, 0x54, 0xF8, 0x36, 0xB7, 0x78, 0x6A, 0x22, 0xA7, 0xDD, 0x88, 0x15, 0xCA, 0x96, 0x39, 0xD3,
    0xE6, 0x55, 0x6E, 0xEA, 0x90, 0x76, 0xB8, 0xFF, 0x50, 0xCD, 0xB5, 0x1B, 0x1F, 0xA5, 0x4D, 0x2E,
    0xB4, 0x09, 0x47, 0x8A, 0xC4, 0x5A, 0x8C, 0x4E, 0xE7, 0x29, 0x50, 0x88, 0xA8, 0x66, 0x85, 0x4B,
    0xAA, 0x38, 0xE7, 0x6B, 0x45, 0x3E, 0x30, 0x37, 0xBA, 0xC5, 0x31, 0xF2, 0x71, 0xB4, 0xCF, 0x29,
    0xBC, 0x7F, 0x7E, 0xD0, 0xC7, 0xC3, 0xBD, 0xCF, 0x59, 0xEA, 0x39, 0x01, 0x2E, 0x00, 0x69, 0x00
};
```

- Memory.reset() li copia negli ultimi 256 byte di m_pRam

EMULARE UN SISTEMA COMPLETO

- Vediamo velocemente le componenti del sistema: memoria
- Bootstrap rom, invece, è in boot_rom.h

```
// From GameBoy-Online emulator
const u8 kBootRomDMG[256] = {
    0x31, 0xFE, 0xFF, 0xAF, 0x21, 0xFF, 0x9F, 0x32, 0xCB, 0x7C, 0x20, 0xFB, 0x21, 0x26, 0xFF, 0x0E,
    0x11, 0x3E, 0x80, 0x32, 0xE2, 0x0C, 0x3E, 0xF3, 0xE2, 0x32, 0x3E, 0x77, 0x77, 0x3E, 0xFC, 0xE0,
    0x47, 0x11, 0x04, 0x01, 0x21, 0x10, 0x80, 0x1A, 0xCD, 0x95, 0x00, 0xCD, 0x96, 0x00, 0x13, 0x7B,
    0xFE, 0x34, 0x20, 0xF3, 0x11, 0xD8, 0x00, 0x06, 0x08, 0x1A, 0x13, 0x22, 0x23, 0x05, 0x20, 0xF9,
    0x3E, 0x19, 0xEA, 0x10, 0x99, 0x21, 0x2F, 0x99, 0x0E, 0x0C, 0x3D, 0x28, 0x08, 0x32, 0x0D, 0x20,
    0xF9, 0x2E, 0x0F, 0x18, 0xF3, 0x67, 0x3E, 0x64, 0x57, 0xE0, 0x42, 0x3E, 0x91, 0xE0, 0x40, 0x04,
    0x1E, 0x02, 0x0E, 0x0C, 0xF0, 0x44, 0xFE, 0x90, 0x20, 0xFA, 0x0D, 0x20, 0xF7, 0x1D, 0x20, 0xF2,
    0x0E, 0x13, 0x24, 0x7C, 0x1E, 0x83, 0xFE, 0x62, 0x28, 0x06, 0x1E, 0xC1, 0xFE, 0x64, 0x20, 0x06,
    0x7B, 0xE2, 0x0C, 0x3E, 0x87, 0xE2, 0xF0, 0x42, 0x90, 0xE0, 0x42, 0x15, 0x20, 0xD2, 0x05, 0x20,
    0x4F, 0x16, 0x20, 0x18, 0xCB, 0x4F, 0x06, 0x04, 0xC5, 0xCB, 0x11, 0x17, 0xC1, 0xCB, 0x11, 0x17,
    0x05, 0x20, 0xF5, 0x22, 0x23, 0x22, 0x23, 0xC9, 0xCE, 0xED, 0x66, 0x66, 0xCC, 0x0D, 0x00, 0x0B,
    0x03, 0x73, 0x00, 0x83, 0x00, 0x0C, 0x00, 0x0D, 0x00, 0x08, 0x11, 0x1F, 0x88, 0x89, 0x00, 0x0E,
    0xDC, 0xCC, 0x6E, 0xE6, 0xDD, 0xDD, 0xD9, 0x99, 0xBB, 0xBB, 0x67, 0x63, 0x6E, 0x0E, 0xEC, 0xCC,
    0xDD, 0xDC, 0x99, 0x9F, 0xBB, 0xB9, 0x33, 0x3E, 0x3C, 0x42, 0xB9, 0xA5, 0xB9, 0xA5, 0x42, 0x3C,
    0x21, 0x04, 0x01, 0x11, 0xA8, 0x00, 0x1A, 0x13, 0xBE, 0x00, 0x00, 0x23, 0x7D, 0xFE, 0x34, 0x20,
    0xF5, 0x06, 0x19, 0x78, 0x86, 0x23, 0x05, 0x20, 0xFB, 0x86, 0x00, 0x00, 0x3E, 0x01, 0xE0, 0x50
};
```

EMULARE UN SISTEMA COMPLETO

- Vediamo velocemente le componenti del sistema: CPU
- Esattamente quello che ci aspettiamo, con qualche variazione

```
public:
    Processor(Memory* pMemory);
    ~Processor();
    void Init();
    void Reset(bool bCGB, bool bootROM);
    u8 Tick();
    void RequestInterrupt(Interrupts interrupt);
    void ResetTIMACycles();
    void ResetDIVCycles();
    bool Halted() const;
    bool CGBSpeed() const;
    void AddCycles(unsigned int cycles);
    bool InterruptIsAboutToRaise();
    bool BootROMfinished() const;

private:
    typedef void (Processor::*OPCptr) (void);
    OPCptr m_OPCodes[256];
    OPCptr m_OPCodesCB[256];
    Memory* m_pMemory;
    SixteenBitRegister AF;
    SixteenBitRegister BC;
    SixteenBitRegister DE;
    SixteenBitRegister HL;
    SixteenBitRegister SP;
    SixteenBitRegister PC;
    bool m_bIME;
    bool m_bHalt;
```

```
private:
    u8 FetchOPCode();
    void ExecuteOPCode(u8 opcode);
    Processor::Interrupts InterruptPending();
    void ServeInterrupt(Interrupts interrupt);
    void UpdateTimers();
    void UpdateSerial();
    void UpdateDelayedInterrupts();
    void ClearAllFlags();
    void ToggleZeroFlagFromResult(u8 result);
    void SetFlag(u8 flag);
    void FlipFlag(u8 flag);
    void ToggleFlag(u8 flag);
    void UntoggleFlag(u8 flag);
    bool IsSetFlag(u8 flag);
    void StackPush(SixteenBitRegister* reg);
    void StackPop(SixteenBitRegister* reg);
    int AdjustedCycles(int cycles);
    void InvalidOPCode();
    void OPCodes_LD(EightBitRegister* reg1, u8 reg2);
    void OPCodes_LD(EightBitRegister* reg, u16 address);
    void OPCodes_LD(u16 address, u8 reg);
    void OPCodes_OR(u8 number);
```

...segue lista opcode functions…

EMULARE UN SISTEMA COMPLETO

- Vediamo velocemente le componenti del sistema: CPU
- main chiama Processor::tick()
  esegue una istruzione e aggiorna timer

```
if (!m_bHalt)
{
    if (m_iAccurateOPCodeState == 0)
        ServeInterrupt(InterruptPending());

    if (m_bDuringBootROM)
    {
        u16 pc_before = PC.GetValue();
        ExecuteOPCode(FetchOPCode());
        u16 pc_after = PC.GetValue();
        if ((pc_before == 0xFE) && (pc_after == 0x100))
        {
            m_bEndOfBootROM = true;
        }
    }
    else
        ExecuteOPCode(FetchOPCode());
}

UpdateDelayedInterrupts();
UpdateTimers();
UpdateSerial();
```

EMULARE UN SISTEMA COMPLETO

- Vediamo velocemente le componenti del sistema: CPU

```
u8 Processor::FetchOPCode()
{
    u8 opcode = m_pMemory->Read(PC.GetValue());
    PC.Increment();

    if (m_bSkipPCBug)
    {
        m_bSkipPCBug = false;
        PC.Decrement();
    }
    return opcode;
}
```

```
void Processor::ExecuteOPCode(u8 opcode)
{
    const u8* accurateOPcodes;
    const u8* machineCycles;
    OPCptr* opcodeTable;
    bool isCB = (opcode == 0xCB);
```

```
#endif

    (this->*opcodeTable[opcode])();

    if (m_bBranchTaken)
    {
        m_bBranchTaken = false;
```

EMULARE UN SISTEMA COMPLETO

- Vediamo velocemente le componenti del sistema: audio
- È classe wrapper per libreria esterna gb_sound_emu v0.2.0

EMULARE UN SISTEMA COMPLETO

- Vediamo velocemente le componenti del sistema: video
- Classe complessa, gestisce rendering su framebuffer esterno (m_pFrameBuffer) e procedure di disegno delle tile e sprite nella video memory interna (mappata in Memory.m_pMap)
- Per incrementare accuratezza di emulazione, timing del video è sincronizzato con l'interrupt vBlank.
  Quando vBlank è attivo, il driver LCD non sta accedendo alla VRAM per copia di framebuffer su schermo.
  vBlank avviene con frequenza di 59,73 Hz

EMULARE UN SISTEMA COMPLETO

- Vi invito a cercare di capire quale sia l'ordine delle chiamate per l'avvio dell'emulatore!
- https://github.com/drhelius/Gearboy

- Suggerimento:
  - platforms/qt-shared/main.cpp: viene creata la MainWindow di QT

  - platforms/qt-shared/MainWindow.cpp: costruttore di MainWindow crea oggetto Emulator che è classe wrapper di GearBoyCore; poi chiama init() di emulator; infine chiama init di renderThread

  - platforms/qt-shared/RenderThread.cpp: contiene membro funzione run() che abbiamo visto insieme; è il main loop!

MAME

- Bibliografia:

- http://www.multigesture.net/articles/how-to-write-an-emulator-chip-8-interpreter/
  http://emulator101.com/
  https://en.wikibooks.org/wiki/Emulation/How_does_it_work%3F
  http://fms.komkon.org/EMUL8/HOWTO.html
  https://github.com/drhelius/Gearboy
- https://realboyemulator.wordpress.com/2013/01/08/emulating-the-core-1/