The Nintendo GAME BOY®

A Case Study in Retro Emulation

Frédéric Hamel
Programming Technical Director at Behaviour Interactive

```
ld   hl,values
values_loop:
push bc
push de
push hl

push bc
pop  af

; Switch stack
ld   (temp),sp
ld   a,(hl+)
ld   h,(hl)
ld   l,a
;    call print_regs
ld   sp,hl

; Set registers
ld   h,d
ld   l,e
ld   a,$12
ld   bc,$5691
ld   de,$9ABC

jp   instr
instr_done:
; Save new SP and switch to yet another stack
ld   (temp+2),sp
ld   sp,$DF70

call checksum_af_bc_de_hl

; Checksum SP
ld   a,(temp+2)
call update_crc_fast
ld   a,(temp+3)
call update_crc_fast

ldsp temp

pop  hl
pop  de
pop  bc
inc  hl
inc  hl
ld   a,l
cp   <values_end
jr   nz,values_loop

pop  hl
ld   a,l
cp   <values_end
jr   nz,hl_loop
```

The Plan

- Introduction
- General emulation principles
- Game Boy hardware overview
- Overview of an emulator I wrote

This talk alternates between general Game Boy emulation and a more specific focus on C++.

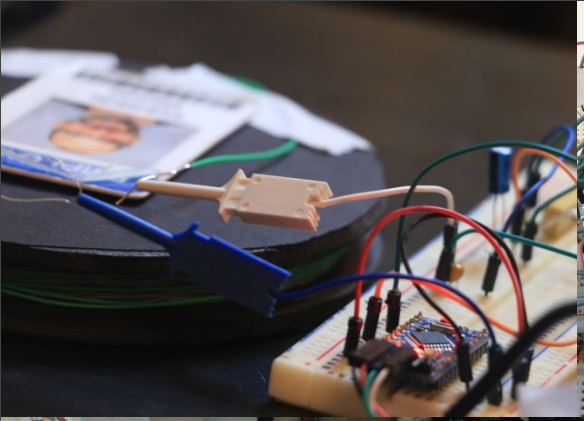
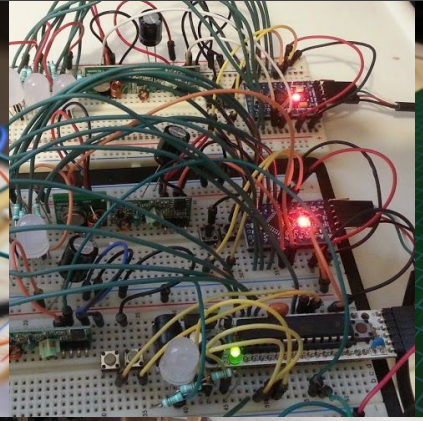
If you have questions, great! Please note the slide number and we'll circle back at the end!

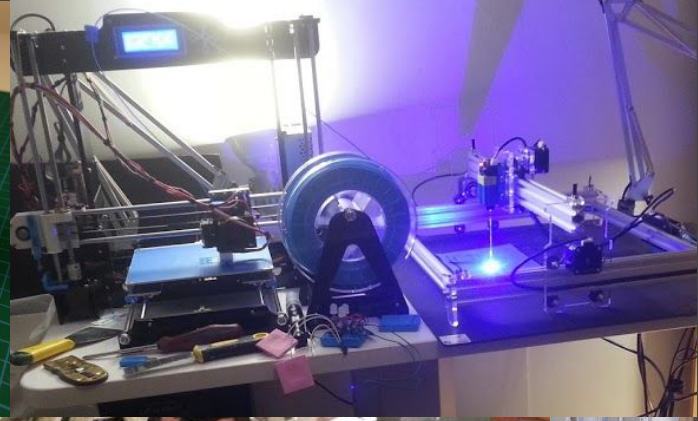
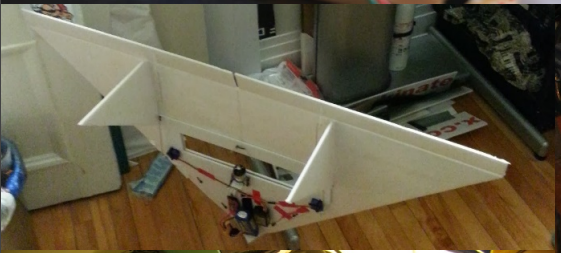
These slides are published here: https://goo.gl/jQQY3c

Who I Am

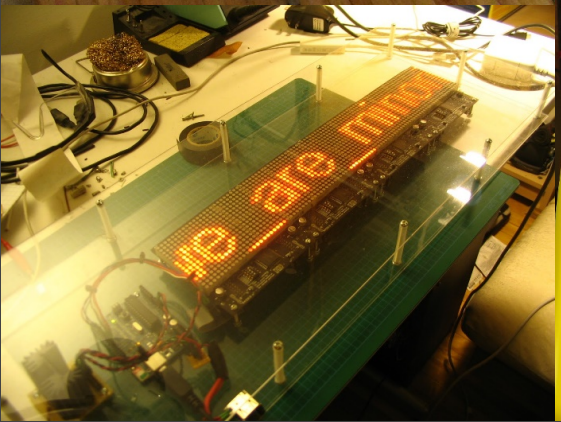
- Graduated in 2004 in computer engineering at the University of Waterloo
- As a programmer, I've worked for Atari/Infogrames, Electronic Arts, Eidos Montréal, Minority, and Behaviour
- Since 2014, I've been a technical director for Behaviour

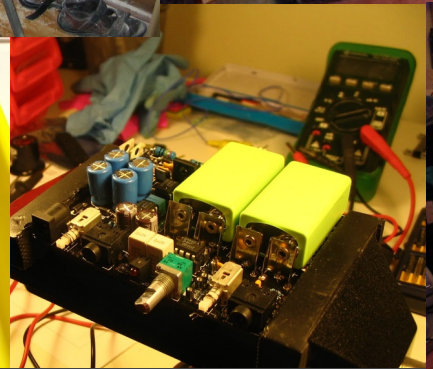
MEDAL OF HONOR EUROPEAN ASSAULT
SPORE HERO
RANGO
PAPO & YO
MIDDLE-EARTH SHADOW OF MORDOR
DIRT RALLY
EA SPORTS FIFA
ARMY OF TWO
MySims SKYHEROES
DISNEY·PIXAR BRAVE
THIEF
HALO WARS
DISHONORED
GEMINOSE




we are




What is an Emulator?

- According to Wikipedia:

  > In computing, an emulator is hardware or software that enables one computer system (called the **host**) to behave like another computer system (called the **guest**).

- Our guest is of course the Game Boy, and our main host of concern is the Windows PC
  - The emulator is fairly portable; it uses Simple DirectMedia Layer (SDL)
  - Note that I was concerned with the original DMG-01 only; I did not attempt to emulate the Game Boy Color or the Super Game Boy
  - Unless otherwise specified, this talk concerns software emulators

How to Emulate a System

- Step 1: understand how the system works
  - Reading publicly-available documentation...
  - …or usually, lots of reverse engineering
- Step 2: develop an emulator based on the rules of the system
  - Typically software
  - Occasionally firmware-based, e.g. firmware on an FPGA
  - Rarely hardware, but it happens

Why a Game Boy Emulator?

- Mainly, it's relatively well-documented
- Quite simple with only a few quirks, so it's a reasonably well self-contained project
  - Compared to e.g. the NES's PPU, the Game Boy LCD controller is much simpler
- I owned one along with many games

Credit Where Credit is Due

- I started from widely-available specification docs
  - "Pan docs" are a great start and the most frequently-quoted reference
  - DP's Game Boy CPU Manual for more detail about the CPU opcodes
  - Very recently, AntonioND's Cycle-Accurate Game Boy Docs (2015+) and GameBoy-Online to find my last known execution bugs
- I did very little if any hardware reverse-engineering
  - But I did discover (apparently novel) evidence of an unsold rotary controller peripheral that worked with the Game Link port!

More Credit

- In order to run an emulator, you need at least one ROM image
  - I spent lots of time with blargg's excellent suite of test ROMs
  - ...and moved on

The Game Boy


```
ld   hl,values
values_loop:
push bc
push de
push hl

push bc
pop  af

; Switch stack
ld   (temp),sp
ld   a,(hl+)
ld   h,(hl)
ld   l,a
;    call print_regs
ld   sp,hl

; Set registers
ld   h,d
ld   l,e
ld   a,$12
ld   bc,$5691
ld   de,$9ABC

jp   instr
instr_done:
; Save new SP and switch to yet another stack
ld   (temp+2),sp
ld   sp,$DF70

call checksum_af_bc_de_hl

; Checksum SP
ld   a,(temp+2)
call update_crc_fast
ld   a,(temp+3)
call update_crc_fast

ldsp temp

pop  hl
pop  de
pop  bc
inc  hl
inc  hl
ld   a,l
cp   <values_end
jr   nz,values_loop

pop  hl
ld   a,l
cp   <values_end
jr   nz,hl_loop
```

Game Boy Hardware Overview

Battery indicator

LCD screen

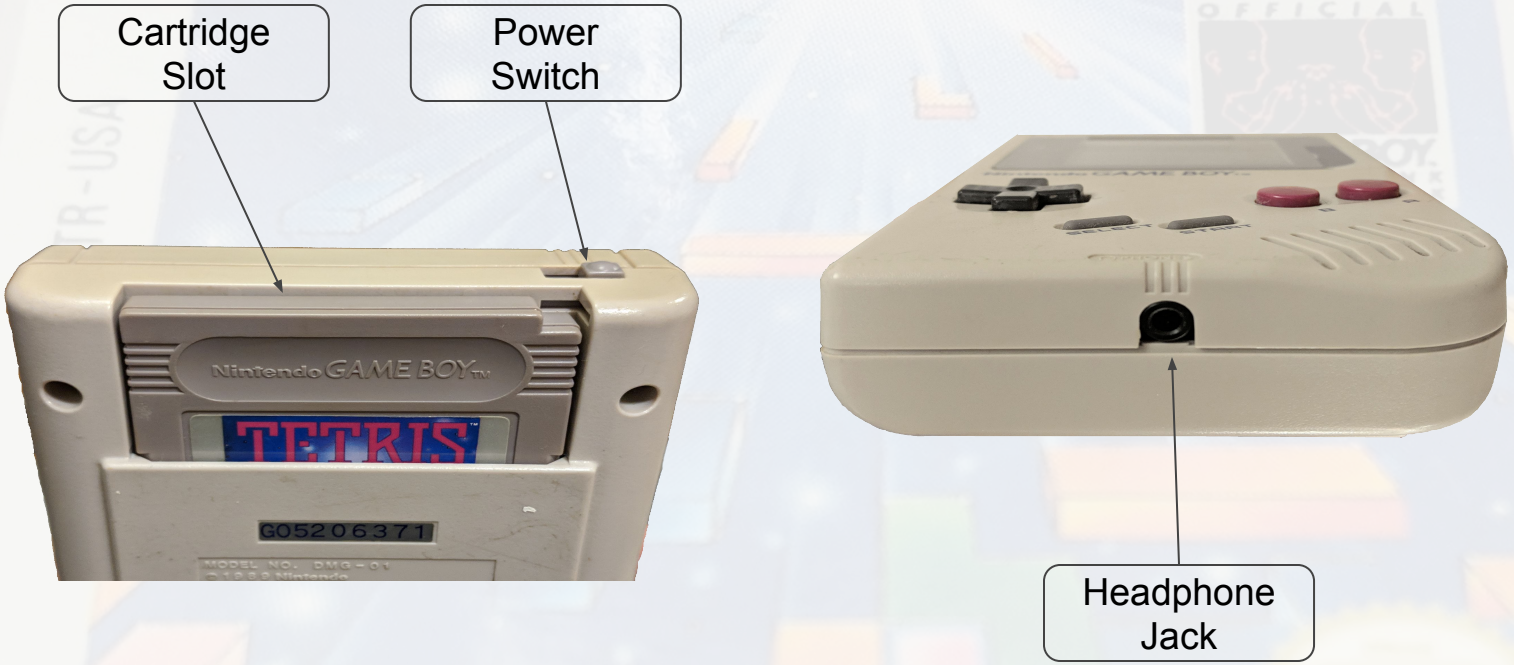
Directional Pad

Select / Start Buttons

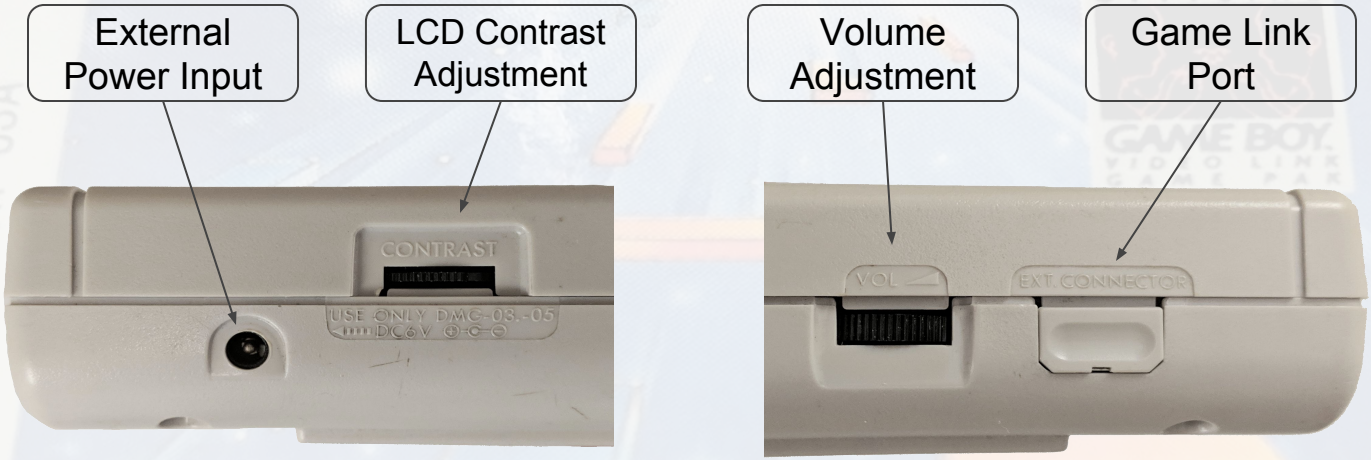
A / B Buttons

Speaker

Game Boy Hardware Overview (cont'd)



Game Boy Hardware Overview (cont'd)

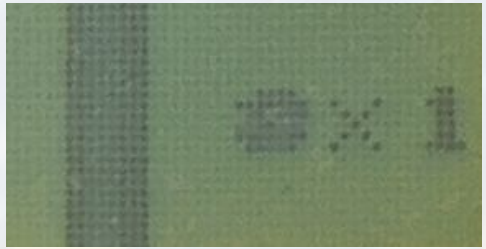


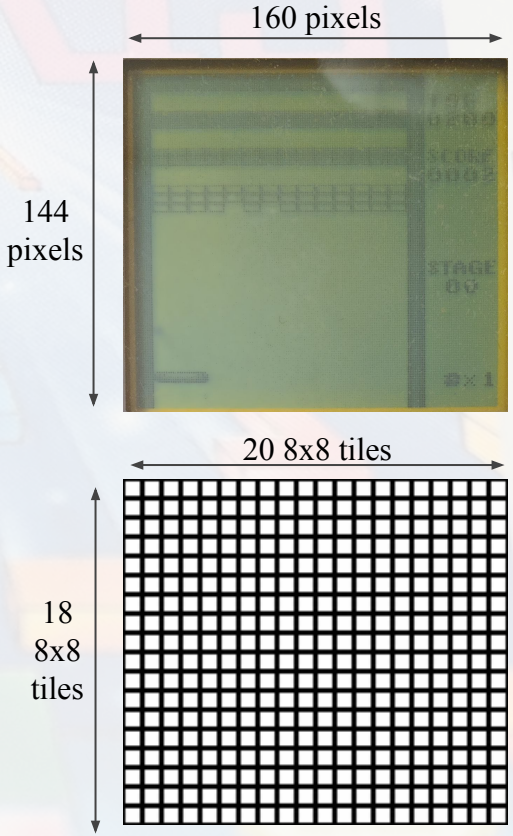
Hardware Specs

- CPU: 8-bit Sharp LR35902, similar to Intel 8080 with some Z80 goodies
- Master clock speed: ~4 MHz
  - CPU: ~1 MIPS (~4 clocks per instruction)
- 8KiB of RAM
- 8KiB of video RAM (VRAM)
- Cartridge size: 32KiB ROM, up to 2MiB ROM with an extra 32KiB of RAM

Hardware Specs (cont'd)

- LCD: 160x144, or 20x18 tiles of 8x8 pixels; 59.7 FPS
- Sprites: max 40 per screen, 10 per horizontal line of pixels
- 4 grayscale colors
- Sound: 4-channel stereo
- Power consumption: 0.7W





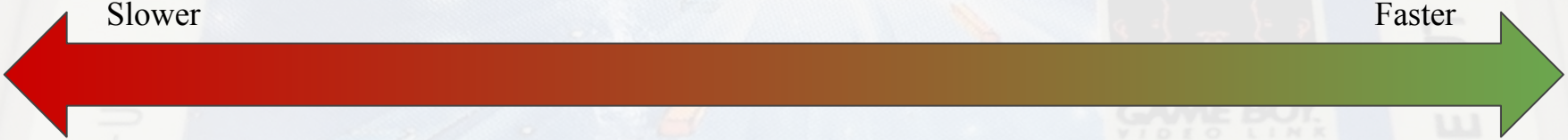
My Goals for this Project

- Clarity and correctness first, speed second
  - Except for CPU - I wanted to try some compile-time techniques to reduce code duplication; this turned out elegant but somewhat opaque
- Have fun
  - Learn about the hardware
  - Try out some C++ ideas
  - Most of the emulator was written in C++11 around the end of 2014
- Accuracy is important but there are diminishing returns
  - Not looking to replace great emulators like Gambatte or bgb

Emulation Strategies

Lower-level emulation
More accuracy
Slower

Higher-level emulation
Less accuracy
Faster



Physical-level emulation: emulating signals, circuit internals, exact sub-instruction clock counts

- Example: Higan

Low-level emulation: emulating CPU instructions atomically, respecting the documented hardware limits but not emulating the inside of the circuitry perfectly

- Most emulators, including this one

High-level emulation: for modern systems that only provide an API with scant hardware access, emulation of API endpoints can be sufficient (fifth/sixth generation of consoles and on)

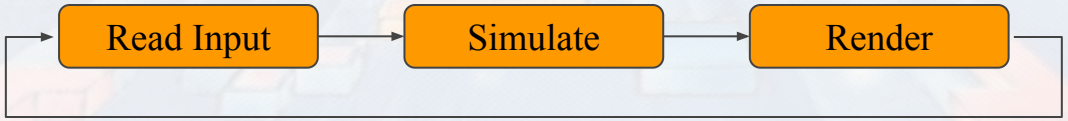
- Example: UltraHLE, WINE

Real-Time Interactive Software

- Unlike much event-driven or utility software, interactive software tends to contain an “update loop”
- Generally, instead of reacting to outside events (program start, button clicks, etc.) the update loop executes repeatedly, as fast as possible

A Typical Update Loop

```
while (!done)
{
    read input
    t = time elapsed during last iteration of this loop
    simulate(t) // reacts to input, advances time of simulation
    render() // produces picture and sound using current simulation state
}
```



This produces a maximal number of images per second, with minimal time spent between user input and a visible or audible result.

The Emulator's Main Loop

```
while (!done) {
    SDL_Event event;
    while (SDL_PollEvent(&event)) {
        switch (event.type) {
        ...
        case SDL_QUIT: done = true; break;
        }
    }
    ...
    auto seconds = elapsedMicroseconds / 1000000.0f;
    if (!paused) gb.Update(seconds);

    SDL_RenderClear(pRenderer.get());
    SDL_RenderCopy(pRenderer.get(), gb.GetFrontFrameBufferTexture(), NULL, NULL);
    SDL_RenderPresent(pRenderer.get());
}
```

Read input

Simulate (update)

This is where the magic is.

Render

The Game Boy Update

```
void Update(float seconds) {
    m_cyclesRemaining += seconds * MemoryBus::kCyclesPerSecond;
    while (m_cyclesRemaining > 0) {
        auto cyclesUsed = m_pCpu->ExecuteSingleInstruction();
        m_cyclesRemaining -= cyclesUsed;

        const float secondsPerClockCycle = 1.0f / MemoryBus::kCyclesPerSecond;
        auto secondsSpentOnInstruction = secondsPerClockCycle * cyclesUsed;

        m_pTimer->Update(secondsSpentOnInstruction);
        m_pJoypad->Update(secondsSpentOnInstruction);
        m_pLcd->Update(secondsSpentOnInstruction);
        m_pSound->Update(secondsSpentOnInstruction);
        m_pGameLinkPort->Update(secondsSpentOnInstruction);
    }
}
```

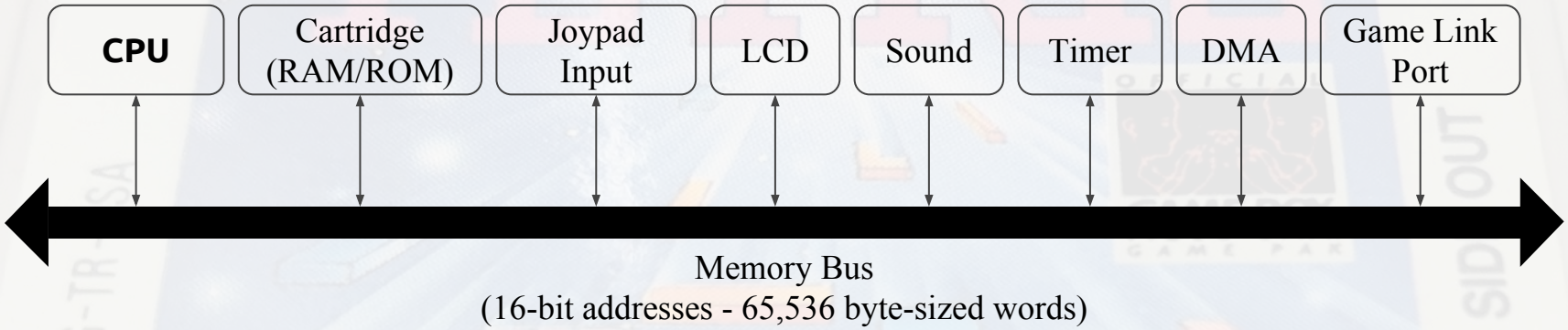
Repeat until all update time exhausted

Emulate one instruction

Convert the number of cycles used by the instruction into seconds

Update all devices with the time spent on the CPU instruction

Game Boy System Diagram



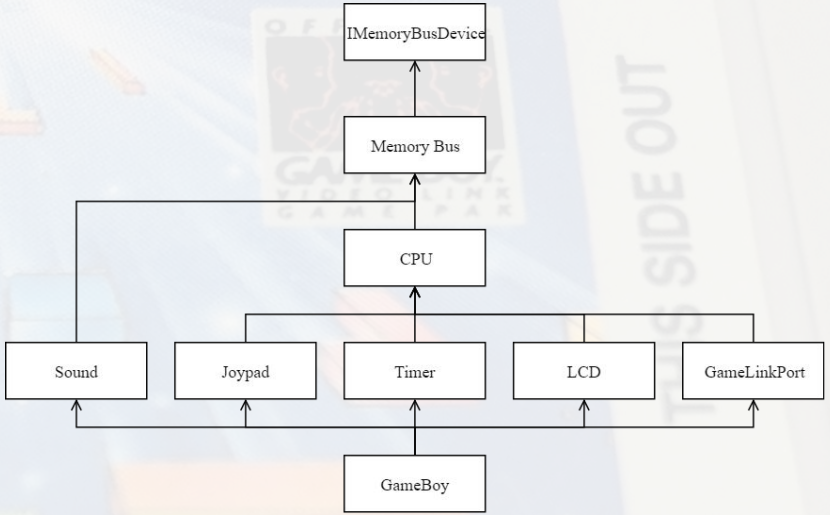
- On the Game Boy, everything is memory-mapped
  - Von Neumann architecture, like x86 and ARM
    - No distinction between code and data; they are in the same memory
  - To talk to devices, the CPU reads and writes from memory addresses
  - Every device has a set of dedicated addresses
    - For example, to read the state of the buttons, the CPU reads address 0xFF00

The Emulator

- I chose to organize my emulator by device; there is roughly one class per device
- Let’s go through them one by one

Quick Note - Code Organization

- Most emulator code is in headers with `#pragma once` guards
- The vast majority of functions are defined directly inline, in the class declaration
  - Because of the header guards, there is a strict ordering of dependencies; the dependency graph cannot contain cycles, which reduces coupling
  - No need to forward-declare any functions or members



Quick Note - Code Organization

- Only one .cpp file is marked for compilation; it includes all the header code along with a few other things
  - As is typical of such unity/glob builds, all code is seen exactly once and only once by the compiler via the main translation unit
- Reasonable iteration time for a C++ program; a full release rebuild takes ~5 seconds on an 8+ year-old computer
  - So do incrementals, however :-)

The CPU


```
                ld   hl,values
values_loop:
                push bc
                push de
                push hl

                push bc
                pop  af

                ; Switch stack
                ld   (temp),sp
                ld   a,(hl+)
                ld   h,(hl)
                ld   l,a
;               call print_regs
                ld   sp,hl

                ; Set registers
                ld   h,d
                ld   l,e
                ld   a,$12
                ld   bc,$5691
                ld   de,$9ABC

                jp   instr
instr_done:
                ; Save new SP and switch to yet another stack
                ld   (temp+2),sp
                ld   sp,$DF70

                call checksum_af_bc_de_hl

                ; Checksum SP
                ld   a,(temp+2)
                call update_crc_fast
                ld   a,(temp+3)
                call update_crc_fast

                ldsp temp

                pop  hl
                pop  de
                pop  bc
                inc  hl
                inc  hl
                ld   a,l
                cp   <values_end
                jr   nz,values_loop

                pop  hl
                ld   a,l
                cp   <values_end
                jr   nz,hl_loop
```

The CPU - Opcode Families

- Relatively simple instruction set
  - Transfers: LD, LDI, PUSH, POP
  - Arithmetic: ADD/ADC, SUB/SBC, CP, INC, DEC, DAA
  - Bitwise operations: AND, XOR, OR, CPL, RLCA/RLA, RRCA/RRA, RLC/RL, RRC/RR, SLA, SRA, SWAP, SRL, BIT, SET, RES
  - Flow control: JP, JR, CALL, RET, RETI, RST
  - Misc: CCF, SCF, NOP, HALT, STOP, DI, EI
- Vast majority of instructions work with 8-bit operands, but a select few work with 16 bits
  - All 16-bit instructions are longer to execute; they are internally decomposed into 8-bit operations

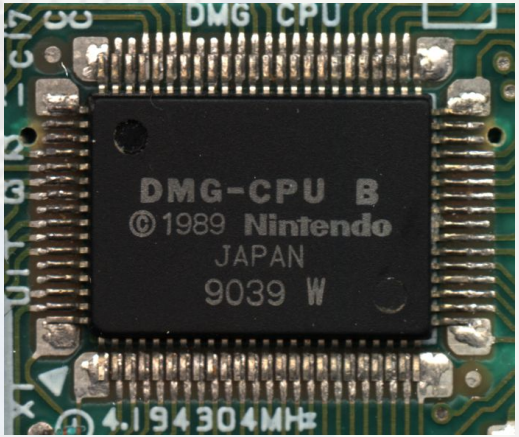


Image source: https://wornwinter.wordpress.com/2015/02/14/adventures-in-gameboy-emulation-part-2-the-cpu/

The CPU - Register Layout

- Few registers
  - 8-bit: A (accumulator), B, C, D, E, H, L, F (flags)
  - 16-bit: PC (program counter), SP (stack pointer)
- 16-bit memory interface means 65,536 addressable 8-bit words
- When the Game Boy starts, the CPU “effectively” starts reading and executing opcodes at address 256 (0x0100) in memory
  - The first 256 bytes are occupied by a boot ROM that takes care of copy protection

Opcodes

- The CPU does the following:
  - Fetch the opcode at the address pointed to by PC
  - Fetch any required operands for that opcode and advance PC accordingly
  - Execute the opcode
  - Repeat
- The first byte in each opcode tells the CPU what to do
- The bytes that follow contain any required operands, such as an address or an immediate value (little endian)

Address	0x0100	0x0101	0x0102	0x0103	0x0104	0x0105	0x0106
Value	0x23	0x80	0xC6	0xFF	0xEA	0x04	0xC0

Opcodes

- The CPU does the following:
  - Fetch the opcode at the address pointed to by PC
  - Fetch any required operands for that opcode and advance PC accordingly
  - Execute the opcode
  - Repeat
- The first byte in each opcode tells the CPU what to do
- The bytes that follow contain any required operands, such as an address or an immediate value (little endian)

Address	0x0100	0x0101	0x0102	0x0103	0x0104	0x0105	0x0106
Value	0x23	0x80	0xC6	0xFF	0xEA	0x04	0xC0
Meaning	INC HL	ADD A,B	ADD A,n	n	LD (nn),A	nn low byte	nn high byte

Opcode Table

	x0	x1	x2	x3
0x	NOP	LD BC,nn	LD (BC),A	INC BC
1x	STOP	LD DE,nn	LD (DE),A	INC DE
2x	JR NZ,n	LD HL,nn	LDI (HL),A	INC HL
3x	JR NC,n	LD SP,nn	LDD (HL),A	INC SP
4x	LD B,B	LD B,C	LD B,D	LD B,E
5x	LD D,B	LD D,C	LD D,D	LD D,E
6x	LD H,B	LD H,C	LD H,D	LD H,E
7x	LD (HL),B	LD (HL),C	LD (HL),D	LD (HL),E
8x	ADD A,B	ADD A,C	ADD A,D	ADD A,E
9x	SUB A,B	SUB A,C	SUB A,D	SUB A,E
Ax	AND B	AND C	AND D	AND E
Bx	OR B	OR C	OR D	OR E
Cx	RET NZ	POP BC	JP NZ,nn	JP nn
Dx	RET NC	POP DE	JP NC,nn	XX
Ex	LDH (n),A	POP HL	LDH (C),A	XX
Fx	LDH A,(n)	POP AF	LDH A,(C)	DI

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

CPU C++ Implementation

- Many emulators implement each opcode individually, resulting in a large amount of similar but not-quite-copy-pasted code
- By using templates, I was able to dramatically cut down on the amount of code duplication

Opcode Table

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	NOP	LD BC,nn	LD (BC),A	INC BC	INC B	DEC B	LD B,n	RLC A	LD (nn),SP	ADD HL,BC	LD A,(BC)	DEC BC	INC C	DEC C	LD C,n	RRC A
1x	STOP	LD DE,nn	LD (DE),A	INC DE	INC D	DEC D	LD D,n	RL A	JR n	ADD HL,DE	LD A,(DE)	DEC DE	INC E	DEC E	LD E,n	RR A
2x	JR NZ,n	LD HL,nn	LDI (HL),A	INC HL	INC H	DEC H	LD H,n	DAA	JR Z,n	ADD HL,HL	LDI A,(HL)	DEC HL	INC L	DEC L	LD L,n	CPL
3x	JR NC,n	LD SP,nn	LDD (HL),A	INC SP	INC (HL)	DEC (HL)	LD (HL),n	SCF	JR C,n	ADD HL,SP	LDD A,(HL)	DEC SP	INC A	DEC A	LD A,n	CCF
4x	LD B,B	LD B,C	LD B,D	LD B,E	LD B,H	LD B,L	LD B,(HL)	LD B,A	LD C,B	LD C,C	LD C,D	LD C,E	LD C,H	LD C,L	LD C,(HL)	LD C,A
5x	LD D,B	LD D,C	LD D,D	LD D,E	LD D,H	LD D,L	LD D,(HL)	LD D,A	LD E,B	LD E,C	LD E,D	LD E,E	LD E,H	LD E,L	LD E,(HL)	LD E,A
6x	LD H,B	LD H,C	LD H,D	LD H,E	LD H,H	LD H,L	LD H,(HL)	LD H,A	LD L,B	LD L,C	LD L,D	LD L,E	LD L,H	LD L,L	LD L,(HL)	LD L,A
7x	LD (HL),B	LD (HL),C	LD (HL),D	LD (HL),E	LD (HL),H	LD (HL),L	HALT	LD (HL),A	LD A,B	LD A,C	LD A,D	LD A,E	LD A,H	LD A,L	LD A,(HL)	LD A,A
8x	ADD A,B	ADD A,C	ADD A,D	ADD A,E	ADD A,H	ADD A,L	ADD A,(HL)	ADD A,A	ADC A,B	ADC A,C	ADC A,D	ADC A,E	ADC A,H	ADC A,L	ADC A,(HL)	ADC A,A
9x	SUB A,B	SUB A,C	SUB A,D	SUB A,E	SUB A,H	SUB A,L	SUB A,(HL)	SUB A,A	SBC A,B	SBC A,C	SBC A,D	SBC A,E	SBC A,H	SBC A,L	SBC A,(HL)	SBC A,A
Ax	AND B	AND C	AND D	AND E	AND H	AND L	AND (HL)	AND A	XOR B	XOR C	XOR D	XOR E	XOR H	XOR L	XOR (HL)	XOR A
Bx	OR B	OR C	OR D	OR E	OR H	OR L	OR (HL)	OR A	CP B	CP C	CP D	CP E	CP H	CP L	CP (HL)	CP A
Cx	RET NZ	POP BC	JP NZ,nn	JP nn	CALL NZ,nn	PUSH BC	ADD A,n	RST 0	RET Z	RET	JP Z,nn	Ext ops	CALL Z,nn	CALL nn	ADC A,n	RST 8
Dx	RET NC	POP DE	JP NC,nn	XX	CALL NC,nn	PUSH DE	SUB A,n	RST 10	RET C	RETI	JP C,nn	XX	CALL C,nn	XX	SBC A,n	RST 18
Ex	LDH (n),A	POP HL	LDH (C),A	XX	XX	PUSH HL	AND n	RST 20	ADD SP,d	JP (HL)	LD (nn),A	XX	XX	XX	XOR n	RST 28
Fx	LDH A,(n)	POP AF	LDH A,(C)	DI	XX	PUSH AF	OR n	RST 30	LDHL SP,d	LD SP,HL	LD A,(nn)	EI	XX	XX	CP n	RST 38

Let's focus on the big central block of load instructions (LD)

Opcode Table

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
4x	LD B,B	LD B,C	LD B,D	LD B,E	LD B,H	LD B,L	LD B,(HL)	LD B,A	LD C,B	LD C,C	LD C,D	LD C,E	LD C,H	LD C,L	LD C,(HL)	LD C,A
5x	LD D,B	LD D,C	LD D,D	LD D,E	LD D,H	LD D,L	LD D,(HL)	LD D,A	LD E,B	LD E,C	LD E,D	LD E,E	LD E,H	LD E,L	LD E,(HL)	LD E,A
6x	LD H,B	LD H,C	LD H,D	LD H,E	LD H,H	LD H,L	LD H,(HL)	LD H,A	LD L,B	LD L,C	LD L,D	LD L,E	LD L,H	LD L,L	LD L,(HL)	LD L,A
7x	LD (HL),B	LD (HL),C	LD (HL),D	LD (HL),E	LD (HL),H	LD (HL),L	HALT	LD (HL),A	LD A,B	LD A,C	LD A,D	LD A,E	LD A,H	LD A,L	LD A,(HL)	LD A,A

Opcode Table

	xxxx0000	xxxx0001	xxxx0010	xxxx0011	xxxx0100	xxxx0101	xxxx0110	xxxx0111	xxxx1000	xxxx1001	xxxx1010	xxxx1011	xxxx1100	xxxx1101	xxxx1110	xxxx1111
0100xxxx	LD B,B	LD B,C	LD B,D	LD B,E	LD B,H	LD B,L	LD B,(HL)	LD B,A	LD C,B	LD C,C	LD C,D	LD C,E	LD C,H	LD C,L	LD C,(HL)	LD C,A
0101xxxx	LD D,B	LD D,C	LD D,D	LD D,E	LD D,H	LD D,L	LD D,(HL)	LD D,A	LD E,B	LD E,C	LD E,D	LD E,E	LD E,H	LD E,L	LD E,(HL)	LD E,A
0110xxxx	LD H,B	LD H,C	LD H,D	LD H,E	LD H,H	LD H,L	LD H,(HL)	LD H,A	LD L,B	LD L,C	LD L,D	LD L,E	LD L,H	LD L,L	LD L,(HL)	LD L,A
0111xxxx	LD (HL),B	LD (HL),C	LD (HL),D	LD (HL),E	LD (HL),H	LD (HL),L	HALT	LD (HL),A	LD A,B	LD A,C	LD A,D	LD A,E	LD A,H	LD A,L	LD A,(HL)	LD A,A

Opcode Table

	xxxx0000	xxxx0001	xxxx0010	xxxx0011	xxxx0100	xxxx0101	xxxx0110	xxxx0111	xxxx1000	xxxx1001	xxxx1010	xxxx1011	xxxx1100	xxxx1101	xxxx1110	xxxx1111
0100xxxx	LD B,B	LD B,C	LD B,D	LD B,E	LD B,H	LD B,L	LD B,(HL)	LD B,A	LD C,B	LD C,C	LD C,D	LD C,E	LD C,H	LD C,L	LD C,(HL)	LD C,A
0101xxxx	LD D,B	LD D,C	LD D,D	LD D,E	LD D,H	LD D,L	LD D,(HL)	LD D,A	LD E,B	LD E,C	LD E,D	LD E,E	LD E,H	LD E,L	LD E,(HL)	LD E,A
0110xxxx	LD H,B	LD H,C	LD H,D	LD H,E	LD H,H	LD H,L	LD H,(HL)	LD H,A	LD L,B	LD L,C	LD L,D	LD L,E	LD L,H	LD L,L	LD L,(HL)	LD L,A
0111xxxx	LD (HL),B	LD (HL),C	LD (HL),D	LD (HL),E	LD (HL),H	LD (HL),L	HALT	LD (HL),A	LD A,B	LD A,C	LD A,D	LD A,E	LD A,H	LD A,L	LD A,(HL)	LD A,A

4x16

Opcode Table

	xxxxx000	xxxxx001	xxxxx010	xxxxx011	xxxxx100	xxxxx101	xxxxx110	xxxxx111
01000xxx	LD B,B	LD B,C	LD B,D	LD B,E	LD B,H	LD B,L	LD B,(HL)	LD B,A
01001xxx	LD C,B	LD C,C	LD C,D	LD C,E	LD C,H	LD C,L	LD C,(HL)	LD C,A
01010xxx	LD D,B	LD D,C	LD D,D	LD D,E	LD D,H	LD D,L	LD D,(HL)	LD D,A
01011xxx	LD E,B	LD E,C	LD E,D	LD E,E	LD E,H	LD E,L	LD E,(HL)	LD E,A
01100xxx	LD H,B	LD H,C	LD H,D	LD H,E	LD H,H	LD H,L	LD H,(HL)	LD H,A
01101xxx	LD L,B	LD L,C	LD L,D	LD L,E	LD L,H	LD L,L	LD L,(HL)	LD L,A
01110xxx	LD (HL),B	LD (HL),C	LD (HL),D	LD (HL),E	LD (HL),H	LD (HL),L	HALT	LD (HL),A
01111xxx	LD A,B	LD A,C	LD A,D	LD A,E	LD A,H	LD A,L	LD A,(HL)	LD A,A

8x8 - same information, different presentation

Opcode Table

	xxxxx000	xxxxx001	xxxxx010	xxxxx011	xxxxx100	xxxxx101	xxxxx110	xxxxx111
01000xxx	LD B , B	LD B , C	LD B , D	LD B , E	LD B , H	LD B , L	LD B ,(HL)	LD B , A
01001xxx	LD C , B	LD C , C	LD C , D	LD C , E	LD C , H	LD C , L	LD C ,(HL)	LD C , A
01010xxx	LD D , B	LD D , C	LD D , D	LD D , E	LD D , H	LD D , L	LD D ,(HL)	LD D , A
01011xxx	LD E , B	LD E , C	LD E , D	LD E , E	LD E , H	LD E , L	LD E ,(HL)	LD E , A
01100xxx	LD H , B	LD H , C	LD H , D	LD H , E	LD H , H	LD H , L	LD H ,(HL)	LD H , A
01101xxx	LD L , B	LD L , C	LD L , D	LD L , E	LD L , H	LD L , L	LD L ,(HL)	LD L , A
01110xxx	LD (HL), B	LD (HL), C	LD (HL), D	LD (HL), E	LD (HL), H	LD (HL), L	HALT	LD (HL), A
01111xxx	LD A , B	LD A , C	LD A , D	LD A , E	LD A , H	LD A , L	LD A ,(HL)	LD A , A

(reformat a little for clarity)

Opcode Table

	xxxxx000	xxxxx001	xxxxx010	xxxxx011	xxxxx100	xxxxx101	xxxxx110	xxxxx111
01000xxx	LD B, B	LD B, C	LD B, D	LD B, E	LD B, H	LD B, L	LD B, (HL)	LD B, A
01001xxx	LD C, B	LD C, C	LD C, D	LD C, E	LD C, H	LD C, L	LD C, (HL)	LD C, A
01010xxx	LD D, B	LD D, C	LD D, D	LD D, E	LD D, H	LD D, L	LD D, (HL)	LD D, A
01011xxx	LD E, B	LD E, C	LD E, D	LD E, E	LD E, H	LD E, L	LD E, (HL)	LD E, A
01100xxx	LD H, B	LD H, C	LD H, D	LD H, E	LD H, H	LD H, L	LD H, (HL)	LD H, A
01101xxx	LD L, B	LD L, C	LD L, D	LD L, E	LD L, H	LD L, L	LD L, (HL)	LD L, A
01110xxx	LD (HL), B	LD (HL), C	LD (HL), D	LD (HL), E	LD (HL), H	LD (HL), L	HALT	LD (HL), A
01111xxx	LD A, B	LD A, C	LD A, D	LD A, E	LD A, H	LD A, L	LD A, (HL)	LD A, A

Opcode Table

	xxxxx000	xxxxx001	xxxxx010	xxxxx011	xxxxx100	xxxxx101	xxxxx110	xxxxx111
01000xxx	LD B , B	LD B , C	LD B , D	LD B , E	LD B , H	LD B , L	LD B ,(HL)	LD B , A
01001xxx	LD C , B	LD C , C	LD C , D	LD C , E	LD C , H	LD C , L	LD C ,(HL)	LD C , A
01010xxx	LD D , B	LD D , C	LD D , D	LD D , E	LD D , H	LD D , L	LD D ,(HL)	LD D , A
01011xxx	LD E , B	LD E , C	LD E , D	LD E , E	LD E , H	LD E , L	LD E ,(HL)	LD E , A
01100xxx	LD H , B	LD H , C	LD H , D	LD H , E	LD H , H	LD H , L	LD H ,(HL)	LD H , A
01101xxx	LD L , B	LD L , C	LD L , D	LD L , E	LD L , H	LD L , L	LD L ,(HL)	LD L , A
01110xxx	LD (HL), B	LD (HL), C	LD (HL), D	LD (HL), E	LD (HL), H	LD (HL), L	HALT	LD (HL), A
01111xxx	LD A , B	LD A , C	LD A , D	LD A , E	LD A , H	LD A , L	LD A ,(HL)	LD A , A

Metafunction declaration

Metafunction specializations

```
template <int Opcode> struct b0_2 { enum { Value = Opcode & 0x7 }; };
template <int N> Uint8& B_C_D_E_H_L_iHL_A_GetReg8();
template <> Uint8& B_C_D_E_H_L_iHL_A_GetReg8<0b000>() { return B; }
template <> Uint8& B_C_D_E_H_L_iHL_A_GetReg8<0b001>() { return C; }
template <> Uint8& B_C_D_E_H_L_iHL_A_GetReg8<0b010>() { return D; }
template <> Uint8& B_C_D_E_H_L_iHL_A_GetReg8<0b011>() { return E; }
template <> Uint8& B_C_D_E_H_L_iHL_A_GetReg8<0b100>() { return H; }
template <> Uint8& B_C_D_E_H_L_iHL_A_GetReg8<0b101>() { return L; }
template <> Uint8& B_C_D_E_H_L_iHL_A_GetReg8<0b111>() { return A; }
```

Missing case <0b110>, which maps to (HL)... We will come back to this!

Opcode Table

	xxxxx000	xxxxx001	xxxxx010	xxxxx011	xxxxx100	xxxxx101	xxxxx110	xxxxx111
01000xxx	LD B, B	LD B, C	LD B, D	LD B, E	LD B, H	LD B, L	LD B,(HL)	LD B, A
01001xxx	LD C, B	LD C, C	LD C, D	LD C, E	LD C, H	LD C, L	LD C,(HL)	LD C, A
01010xxx	LD D, B	LD D, C	LD D, D	LD D, E	LD D, H	LD D, L	LD D,(HL)	LD D, A
01011xxx	LD E, B	LD E, C	LD E, D	LD E, E	LD E, H	LD E, L	LD E,(HL)	LD E, A
01100xxx	LD H, B	LD H, C	LD H, D	LD H, E	LD H, H	LD H, L	LD H,(HL)	LD H, A
01101xxx	LD L, B	LD L, C	LD L, D	LD L, E	LD L, H	LD L, L	LD L,(HL)	LD L, A
01110xxx	LD (HL), B	LD (HL), C	LD (HL), D	LD (HL), E	LD (HL), H	LD (HL), L	HALT	LD (HL), A
01111xxx	LD A, B	LD A, C	LD A, D	LD A, E	LD A, H	LD A, L	LD A,(HL)	LD A, A

Opcode Table

	xxxxx000	xxxxx001	xxxxx010	xxxxx011	xxxxx100	xxxxx101	xxxxx110	xxxxx111
01000xxx	LD B, B	LD B, C	LD B, D	LD B, E	LD B, H	LD B, L	LD B,(HL)	LD B, A
01001xxx	LD C, B	LD C, C	LD C, D	LD C, E	LD C, H	LD C, L	LD C,(HL)	LD C, A
01010xxx	LD D, B	LD D, C	LD D, D	LD D, E	LD D, H	LD D, L	LD D,(HL)	LD D, A
01011xxx	LD E, B	LD E, C	LD E, D	LD E, E	LD E, H	LD E, L	LD E,(HL)	LD E, A
01100xxx	LD H, B	LD H, C	LD H, D	LD H, E	LD H, H	LD H, L	LD H,(HL)	LD H, A
01101xxx	LD L, B	LD L, C	LD L, D	LD L, E	LD L, H	LD L, L	LD L,(HL)	LD L, A
01110xxx	LD (HL), B	LD (HL), C	LD (HL), D	LD (HL), E	LD (HL), H	LD (HL), L	HALT	LD (HL), A
01111xxx	LD A, B	LD A, C	LD A, D	LD A, E	LD A, H	LD A, L	LD A,(HL)	LD A, A

```
template <int Opcode> struct b3_5 { enum { Value = (Opcode >> 3) & 0x7 }; };
template <int N> Uint8& B_C_D_E_H_L_iHL_A_GetReg8();
template <> Uint8& B_C_D_E_H_L_iHL_A_GetReg8<0b000>() { return B; }
template <> Uint8& B_C_D_E_H_L_iHL_A_GetReg8<0b001>() { return C; }
template <> Uint8& B_C_D_E_H_L_iHL_A_GetReg8<0b010>() { return D; }
template <> Uint8& B_C_D_E_H_L_iHL_A_GetReg8<0b011>() { return E; }
template <> Uint8& B_C_D_E_H_L_iHL_A_GetReg8<0b100>() { return H; }
template <> Uint8& B_C_D_E_H_L_iHL_A_GetReg8<0b101>() { return L; }
template <> Uint8& B_C_D_E_H_L_iHL_A_GetReg8<0b111>() { return A; }
```

Same code as previous slide

Dealing with Memory

```
template <int N> Uint8& B_C_D_E_H_L_iHL_A_GetReg8();
template <> Uint8& B_C_D_E_H_L_iHL_A_GetReg8<0b000>() { return B; }
template <> Uint8& B_C_D_E_H_L_iHL_A_GetReg8<0b001>() { return C; }
template <> Uint8& B_C_D_E_H_L_iHL_A_GetReg8<0b010>() { return D; }
template <> Uint8& B_C_D_E_H_L_iHL_A_GetReg8<0b011>() { return E; }
template <> Uint8& B_C_D_E_H_L_iHL_A_GetReg8<0b100>() { return H; }
template <> Uint8& B_C_D_E_H_L_iHL_A_GetReg8<0b101>() { return L; }
(no case 0b110)
template <> Uint8& B_C_D_E_H_L_iHL_A_GetReg8<0b111>() { return A; }
```

(6)	x110
LD	B ,(HL)
LD	C ,(HL)
LD	D ,(HL)
LD	E ,(HL)
LD	H ,(HL)
LD	L ,(HL)
	HALT
LD	A ,(HL)

- All cases above return a **`Uint8&`**, which means we can simply read and write to the reference as needed to implement opcodes
- However, case 6 maps to (HL), which means accessing memory at an address given by the 16-bit pointer in the HL register pair
  - This is a bit more complex

Dealing with Memory (cont'd)

- In the emulator, memory accesses map to a function call because code must run
  - There are many memory-mapped devices that must perform computation when they receive or give data
- How to get around this problem of wanting straight variable access when you can have it, and function calls when you can't?

Dealing with Memory (cont'd)

- Solution: apply the fundamental theorem of software engineering

> "We can solve any problem by introducing an extra level of indirection."
>
> -Butler Lampson

- We will create a layer of code to branch between direct variable access and function calls

Dealing with Memory (cont'd)

- We could have created a "fancy" data type that overrides operator=() and operator Uint8() to implement read and write, but I wanted to keep things as simple as possible for the optimizer...
- ...so I went for another simple compile-time selection:

```
template <int N> Uint8 B_C_D_E_H_L_iHL_A_Read8() {
   return B_C_D_E_H_L_iHL_A_GetReg8<N>(); }
template <int N> void B_C_D_E_H_L_iHL_A_Write8(Uint8 value) {
   B_C_D_E_H_L_iHL_A_GetReg8<N>() = value; }

template <> Uint8 B_C_D_E_H_L_iHL_A_Read8<0b110>() {
   return Read8(HL); }
template <> void B_C_D_E_H_L_iHL_A_Write8<0b110>(Uint8 value) {
   Write8(HL, value); }
```

Direct member variable access whenever possible; simple assignments and variable reads

Functions called to read and write from/to the address pointed to by HL when required

Example Code Generation

```
Uint16 CPU::DoExecuteSingleInstruction() {
    Uint8 opcode = Fetch8();       // reads in the next byte and advances PC
    Sint32 instructionCycles = -1; // number of clock cycles used by the opcode; will be returned from this function

    #define OPCODE(code, cycles, name) case code: instructionCycles = (cycles); name<code>(); break;
    switch (opcode)     {
    ...
    OPCODE(0x03, 8, INC_0_3__3)
    OPCODE(0x13, 8, INC_0_3__3)
    OPCODE(0x23, 8, INC_0_3__3)
    OPCODE(0x33, 8, INC_0_3__3)
    ...
    OPCODE(0x40, 4, LD_4_7__0_F__NO_7__6)
    OPCODE(0x41, 4, LD_4_7__0_F__NO_7__6)
    OPCODE(0x42, 4, LD_4_7__0_F__NO_7__6)
    ...
```

Example Code Generation

```
Uint16 CPU::DoExecuteSingleInstruction() {
    Uint8 opcode = Fetch8();       // reads in the next byte and advances PC
    Sint32 instructionCycles = -1; // number of clock cycles used by the opcode; will be returned from this function

    #define OPCODE(code, cycles, name) case code: instructionCycles = (cycles); name<code>(); break;
    switch (opcode)     {
    ...
    case 0x03: instructionCycles = 8; INC_0_3__3<0x03>(); break;
    case 0x13: instructionCycles = 8; INC_0_3__3<0x13>(); break;
    case 0x23: instructionCycles = 8; INC_0_3__3<0x23>(); break;
    case 0x33: instructionCycles = 8; INC_0_3__3<0x33>(); break;
    ...
    case 0x40: instructionCycles = 4; LD_4_7__0_F__NO_7__6<0x40>(); break;
    case 0x41: instructionCycles = 4; LD_4_7__0_F__NO_7__6<0x41>(); break;
    case 0x32: instructionCycles = 4; LD_4_7__0_F__NO_7__6<0x32>(); break;
    ...
```

Example Code Generation

```
Uint16 CPU::DoExecuteSingleInstruction() {
    Uint8 opcode = Fetch8();       // reads in the next byte and advances PC
    Sint32 instructionCycles = -1; // number of clock cycles used by the opcode; will be returned from this function

    #define OPCODE(code, cycles, name) case code: instructionCycles = (cycles); name<code>(); break;
    switch (opcode)     {
    ...
    OPCODE(0x03, 8, INC_0_3__3)
    OPCODE(0x13, 8, INC_0_3__3)
    OPCODE(0x23, 8, INC_0_3__3)
    OPCODE(0x33, 8, INC_0_3__3)
    ...
    OPCODE(0x40, 4, LD_4_7__0_F__NO_7__6)
    OPCODE(0x41, 4, LD_4_7__0_F__NO_7__6)
    OPCODE(0x42, 4, LD_4_7__0_F__NO_7__6)
    ...
```

Example Code Generation

```
template <int N> void LD_4_7__0_F__NO_7__6() {
    // This line of code implements 63 opcodes (8 inputs by 8 outputs, minus HALT)
    b3_5_B_C_D_E_H_L_iHL_A_Write8<N>(b0_2_B_C_D_E_H_L_iHL_A_Read8<N>());
}

#define OPCODE(code, cycles, name) case code: instructionCycles = (cycles); name<code>(); break;
```

```
        ; x86 assembler
        OPCODE(0x6F, 4, LD_4_7__0_F__NO_7__6) ; LD L,A
00EE38F9  mov         al,byte ptr [esi+9]       ; read the correct register (in this case, A)
00EE38FC  mov         edi,4                     ; instructionCycles = (cycles);
00EE3901  mov         byte ptr [esi+0Eh],al     ; write to the correct register (in this case, L)
00EE3904  jmp         $LN521+0Ch (0EE5655h)     ; jump to end of switch/case
```

Factoring Efficiency

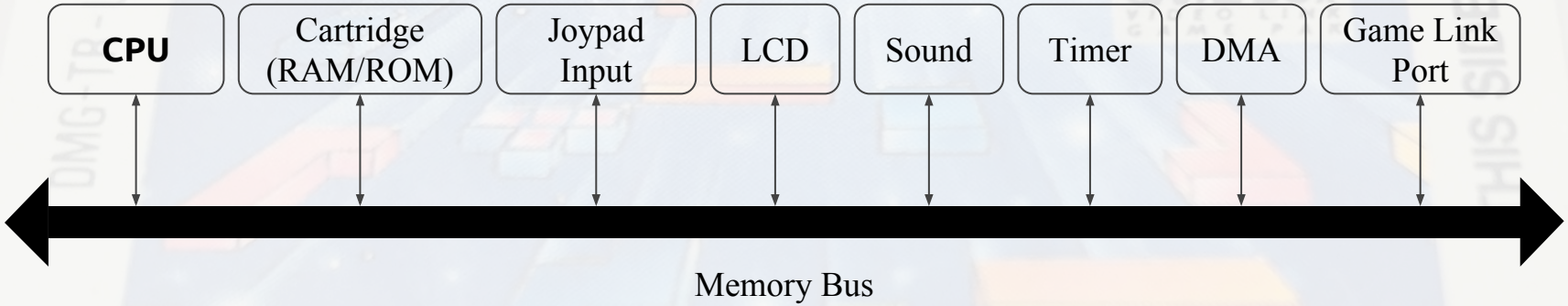
- Using this method to implement blocks of instructions that vary only in their operands, 77 C++ functions are used to implement 500 opcodes
  - If we set aside the “unique” instructions, 37 C++ functions implement 463 opcodes with no code duplication
  - This is an average of 12.5 opcodes implemented per C++ function
  - As we saw, the resulting code is very efficient, and it is also easily debugged in non-optimized configurations

The Memory Bus


```
ld   hl,values
values_loop:
push bc
push de
push hl

push bc
pop  af

; Switch stack
ld   (temp),sp
ld   a,(hl+)
ld   h,(hl)
ld   l,a
;    call print_regs
ld   sp,hl

; Set registers
ld   h,d
ld   l,e
ld   a,$12
ld   bc,$5691
ld   de,$9ABC

jp   instr
instr_done:
; Save new SP and switch to yet another stack
ld   (temp+2),sp
ld   sp,$DF70

call checksum_af_bc_de_hl

; Checksum SP
ld   a,(temp+2)
call update_crc_fast
ld   a,(temp+3)
call update_crc_fast

ldsp temp

pop  hl
pop  de
pop  bc
inc  hl
inc  hl
ld   a,l
cp   <values_end
jr   nz,values_loop

pop  hl
ld   a,l
cp   <values_end
jr   nz,hl_loop
```

The Memory Bus

- The CPU can’t do much by itself; it needs to communicate with other devices in the system to produce interesting results



Memory Bus Devices

- All devices in the system implement this interface

```
class IMemoryBusDevice {
public:
   virtual bool HandleRequest(MemoryRequestType requestType, Uint16 address, Uint8& value) = 0;
};
```

- The memory bus holds a list of memory bus devices
- When it receives a read/write request for a given address, it asks each device in turn if it can handle it
  - If the device handles it, the request is complete
  - If the device cannot handle the request, the memory bus moves down to the next device in the list

Handling a Memory Access

```
Uint8 Read8(Uint16 address) {
   Uint8 result;
   auto numDevices = static_cast<Sint8>(m_devices.size());
   for (Sint8 deviceIndex = 0; deviceIndex < numDevices; ++deviceIndex) {
       const auto& pDevice = m_devices[deviceIndex];
       if (pDevice->HandleRequest(MemoryRequestType::Read, address, result)) {
           return result;
       }
   }
   // (should never reach this line - error handling and such)
}
```

- However, as you might expect, if this is done for every memory access, it is quite slow
  - The solution is to cache which device handles each memory address, since this information is constant

Cartridges and ROM


```
                ld   hl,values
values_loop:
                push bc
                push de
                push hl

                push bc
                pop  af

                ; Switch stack
                ld   (temp),sp
                ld   a,(hl+)
                ld   h,(hl)
                ld   l,a
;               call print_regs
                ld   sp,hl

                ; Set registers
                ld   h,d
                ld   l,e
                ld   a,$12
                ld   bc,$5691
                ld   de,$9ABC

                jp   instr
instr_done:
                ; Save new SP and switch to yet another stack
                ld   (temp+2),sp
                ld   sp,$DF70

                call checksum_af_bc_de_hl

                ; Checksum SP
                ld   a,(temp+2)
                call update_crc_fast
                ld   a,(temp+3)
                call update_crc_fast

                ldsp temp

                pop  hl
                pop  de
                pop  bc
                inc  hl
                inc  hl
                ld   a,l
                cp   <values_end
                jr   nz,values_loop

                pop  hl
                ld   a,l
                cp   <values_end
                jr   nz,hl_loop
```

Cartridges

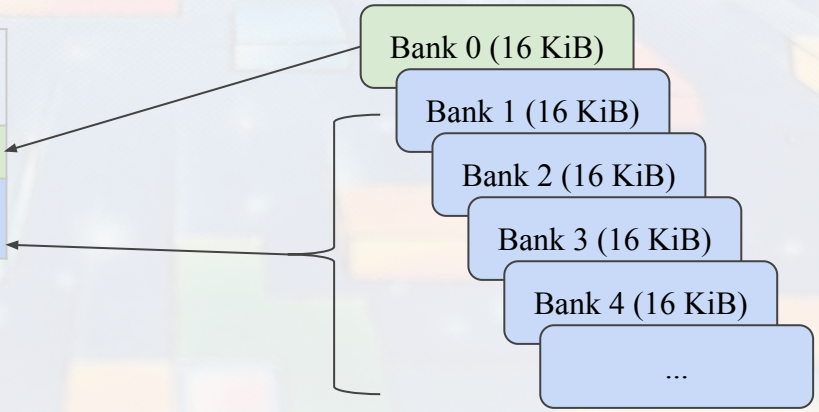
- 32k of the addressable memory space is dedicated to the cartridge
  - 32k is not a lot to build games, though some do
- To get around this, the vast majority of cartridges use **Memory Bank Controllers** (MBCs)
- MBCs are extra chips inside the cartridge itself

Cartridge ROM Banking

- MBCs allow the upper 16k of ROM to be *banked*
  - Cartridges can hold up to 2 MiB of ROM
  - By writing to a specific address, the CPU can decide which area - which **bank** - of this ROM is exposed

CPU Address Space

Starting Address	End Address	
0000	3FFF	First 16 KiB of ROM - Always Bank 0
4000	7FFF	Bank-switchable 16 KiB of ROM - only one bank is visible to the CPU at a time



The Joypad


```
                ld   hl,values
values_loop:
                push bc
                push de
                push hl

                push bc
                pop  af

                ; Switch stack
                ld   (temp),sp
                ld   a,(hl+)
                ld   h,(hl)
                ld   l,a
;               call print_regs
                ld   sp,hl

                ; Set registers
                ld   h,d
                ld   l,e
                ld   a,$12
                ld   bc,$5691
                ld   de,$9ABC

                jp   instr
instr_done:
                ; Save new SP and switch to yet another stack
                ld   (temp+2),sp
                ld   sp,$DF70

                call checksum_af_bc_de_hl

                ; Checksum SP
                ld   a,(temp+2)
                call update_crc_fast
                ld   a,(temp+3)
                call update_crc_fast

                ldsp temp

                pop  hl
                pop  de
                pop  bc
                inc  hl
                inc  hl
                ld   a,l
                cp   <values_end
                jr   nz,values_loop

                pop  hl
                ld   a,l
                cp   <values_end
                jr   nz,hl_loop
```

The Joypad

- The joypad is very simple at the hardware level: it is composed of one single register, called P1_JOYP, that lives at 0xFF00

Bit	7	6	5	4	3	2	1	0
Read/Write	N/A	N/A	Write	Write	Read	Read	Read	Read
Purpose	Unused	Unused	Select button keys	Select direction pad	Start or Down	Select or Up	B or Left	A or Right

- A game can read the full joypad by doing the following:

```
write 0 to bit 5 and 1 to bit 4 // select buttons (0 is active)
read bits 0-3                   // read buttons
write 0 to bit 4 and 1 to bit 5 // select d-pad
read bits 0-3                   // read d-pad
```

Joypad Implementation

```
class Joypad : public IMemoryBusDevice {
   ...
   void Update(float seconds) {
       ...
       const auto pKeyState = SDL_GetKeyboardState(nullptr);

       if ((P1_JOYP & Bit5) == 0) { // Buttons
           SetBitValue(P1_JOYP, 0, pKeyState[SDL_SCANCODE_E] == 0); // A: E
           SetBitValue(P1_JOYP, 1, pKeyState[SDL_SCANCODE_R] == 0); // B: R
           SetBitValue(P1_JOYP, 2, pKeyState[SDL_SCANCODE_Q] == 0); // Select: Q
           SetBitValue(P1_JOYP, 3, pKeyState[SDL_SCANCODE_W] == 0); // Start: W
       }

       if ((P1_JOYP & Bit4) == 0) { // D-pad
           SetBitValue(P1_JOYP, 0, pKeyState[SDL_SCANCODE_RIGHT] == 0);
           SetBitValue(P1_JOYP, 1, pKeyState[SDL_SCANCODE_LEFT] == 0);
           SetBitValue(P1_JOYP, 2, pKeyState[SDL_SCANCODE_UP] == 0);
           SetBitValue(P1_JOYP, 3, pKeyState[SDL_SCANCODE_DOWN] == 0);
       }
   }
```

Joypad Implementation

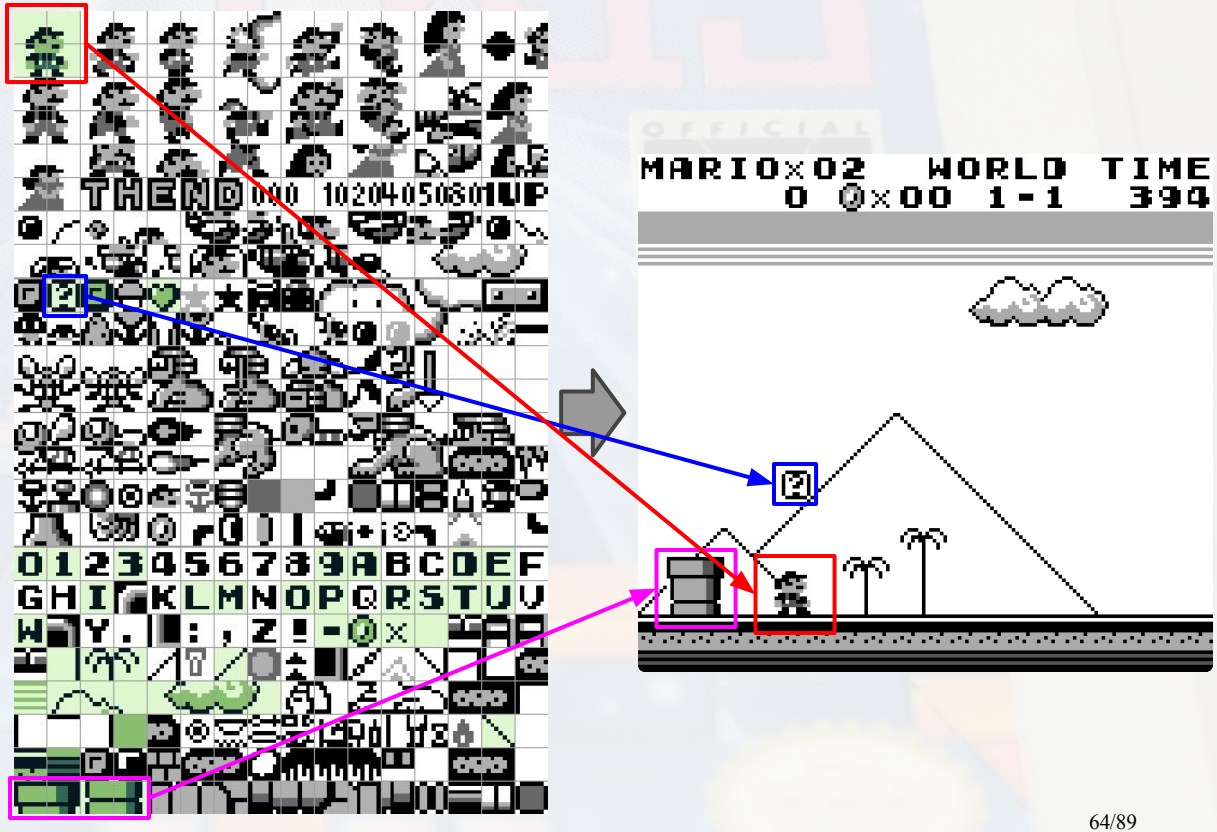
```
class Joypad : public IMemoryBusDevice {
   ...
   virtual bool HandleRequest(MemoryRequestType requestType, Uint16 address, Uint8& value) {
       switch (address) {
       case Registers::P1_JOYP: {
               if (requestType == MemoryRequestType::Write) {
                   //  The incoming written value can only affect the top four bits of the P1_JOYP register
                   P1_JOYP = (P1_JOYP & 0x0F) | (value & 0xF0);
               } else {
                   // return the current P1_JOYP register value
                   value = P1_JOYP;
               }
               return true;
           }
           break;
       }
       return false;
   }
```

The LCD Controller


```
ld   hl,values
values_loop:
push bc
push de
push hl

push bc
pop  af

; Switch stack
ld   (temp),sp
ld   a,(hl+)
ld   h,(hl)
ld   l,a
;    call print_regs
ld   sp,hl

; Set registers
ld   h,d
ld   l,e
ld   a,$12
ld   bc,$5691
ld   de,$9ABC

jp   instr
instr_done:
; Save new SP and switch to yet another stack
ld   (temp+2),sp
ld   sp,$DF70

call checksum_af_bc_de_hl

; Checksum SP
ld   a,(temp+2)
call update_crc_fast
ld   a,(temp+3)
call update_crc_fast

ldsp temp

pop  hl
pop  de
pop  bc
inc  hl
inc  hl
ld   a,l
cp   <values_end
jr   nz,values_loop

pop  hl
ld   a,l
cp   <values_end
jr   nz,hl_loop
```

LCD Tiles

- Like many consoles from that era, the Game Boy does not have a frame buffer
- Instead, images are composed from 8x8 tiles which are combined into background and sprite layers

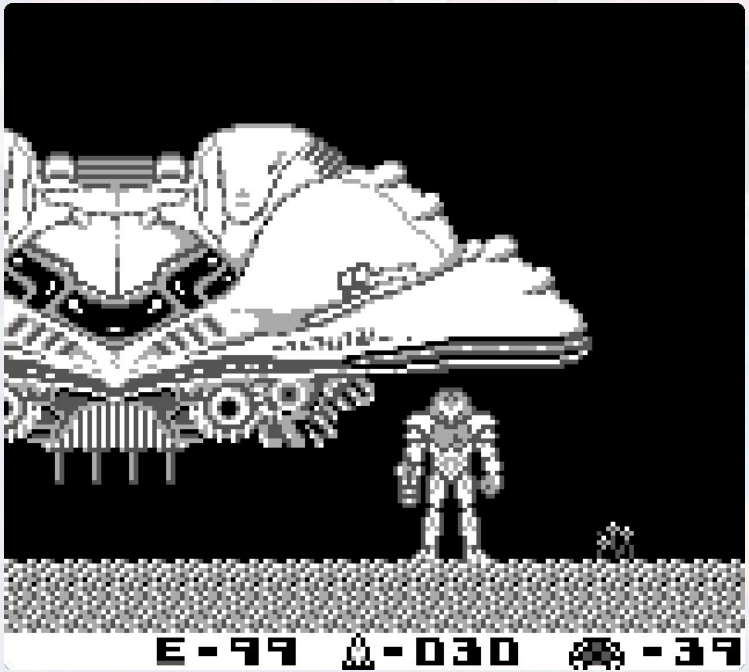


Tile map screenshot from BGB

LCD Layers

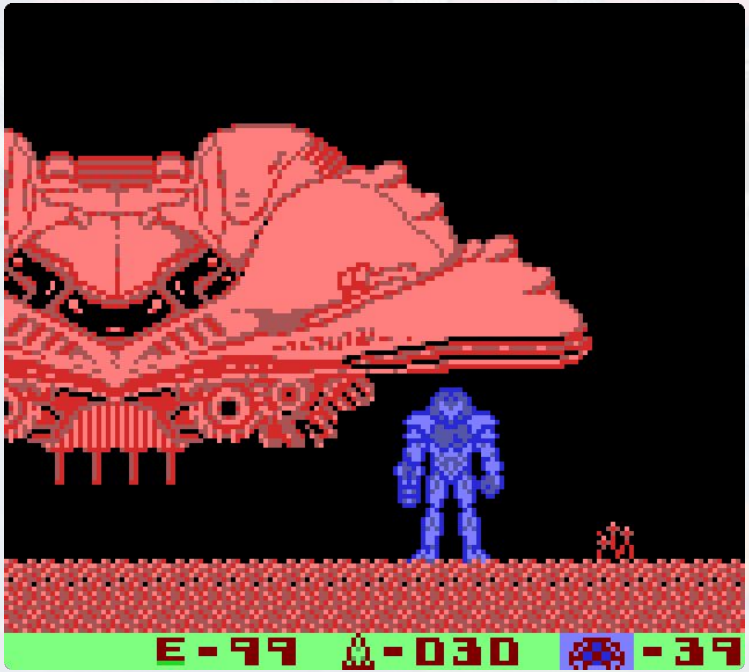
- Three layers make up the final image
  - The **background**
    - Made up of tiles, which fill up the screen
    - Can be scrolled pixel-by-pixel
  - The **window**
    - Made up of tiles, which are displayed above the background
    - Can be positioned pixel-by-pixel
  - The **sprites**
    - Made up of 1x1 or 2x1 tiles
    - Can be displayed above or below the background and window
    - Up to a maximum of 40 sprites total, but a maximum of 10 per line

LCD Layer Example



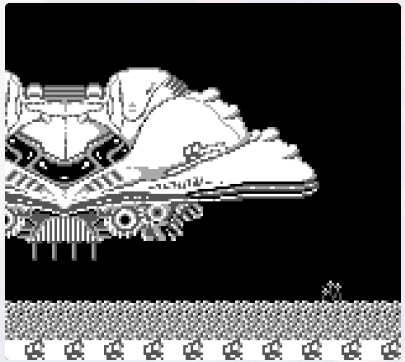
LCD Layer Example

Background
Window
Sprites



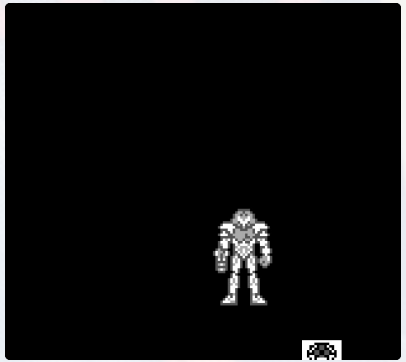
LCD Layer Example

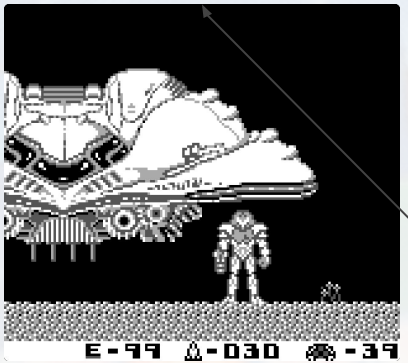
Background



Window

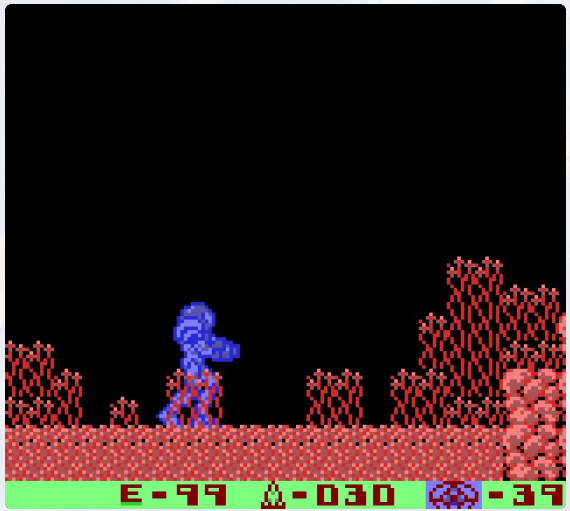
Sprites



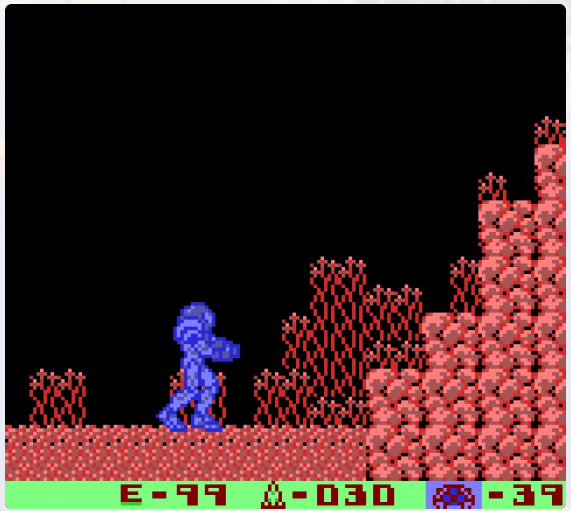


LCD Layer Tricks

- Games can use layer sorting to give intricate illusions
  - E.g. Metroid 2 frequently toggles layer ordering to create an illusion of depth



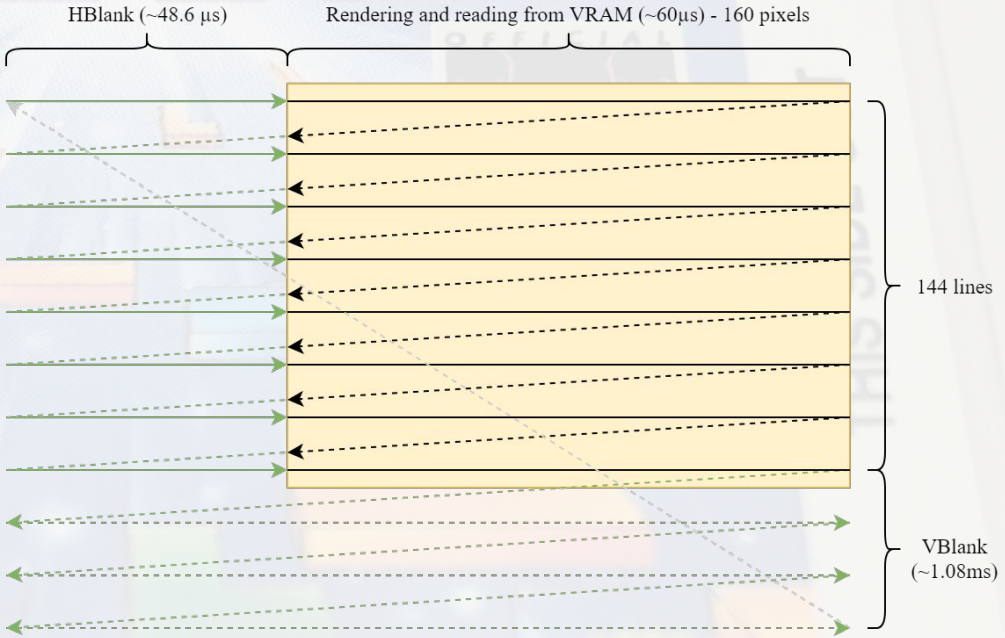
Samus is behind the plants



A few steps over, Samus is in front of the plants

LCD Controller Timing

- Though there is no physical need for it, the LCD controller implements a CRT-like line-by-line tracing pattern
- This allows software to change the contents of VRAM on every line during HBlank, and on every frame during VBlank



LCD Controller Timing

- This allows for many popular scanline tricks which involve leveraging the HBlank interval to change the layout of visible tiles between every drawn line

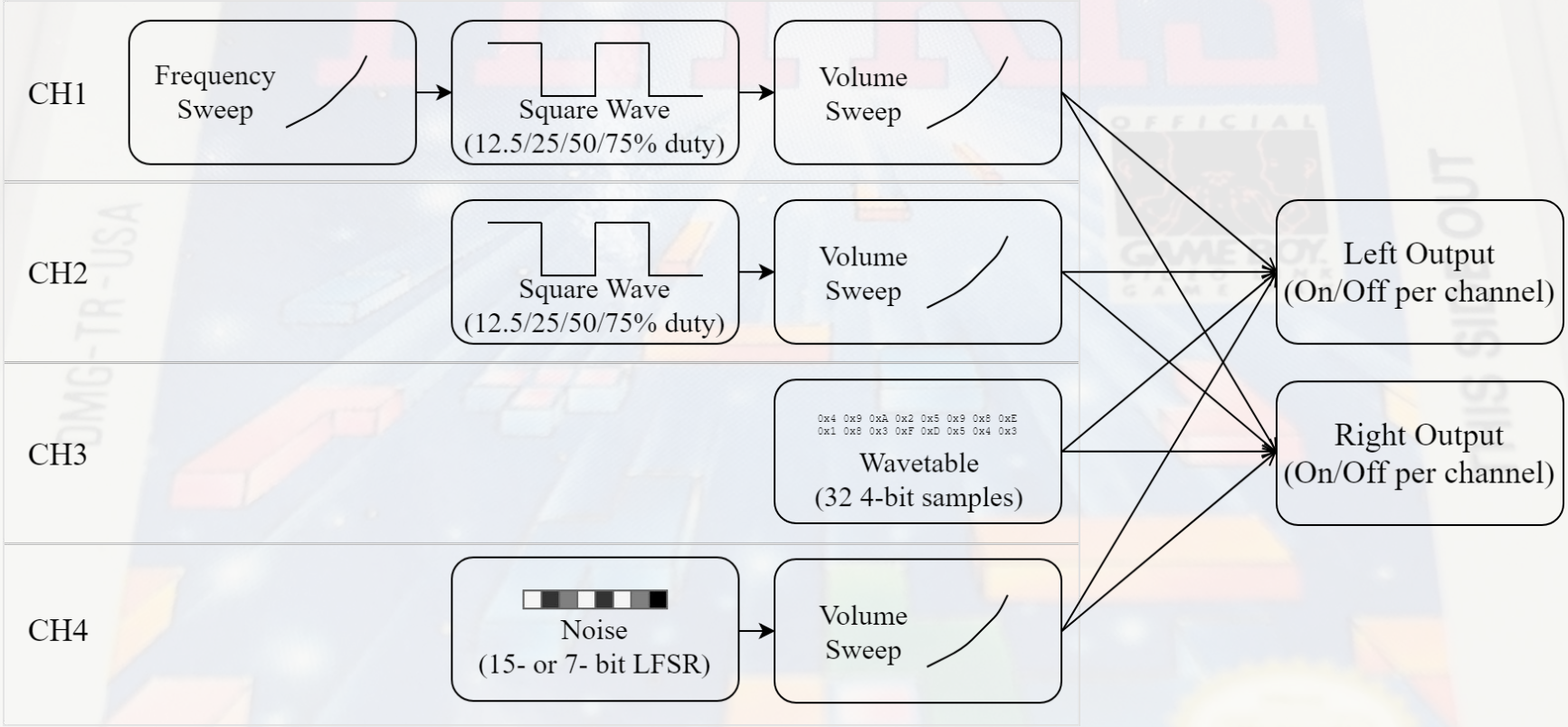
LCD C++ Implementation

- The LCD controller implementation is straightforward, and there is no real magic
  - All of it fits in <700 lines of Allman/BSD-style C++
- HBlank/VBlank timing is implemented using a simple explicit state machine
- When VBlank is hit, the finished frame is presented to the user
- Rendering is handled one full scanline at a time, since the CPU can't alter any memory during that time
  - Mainly two operations
    - Array indexing computations to figure out what pixels go where
    - Lots of bit/flag testing to figure out the relative priorities, horizontal/vertical flips, etc.

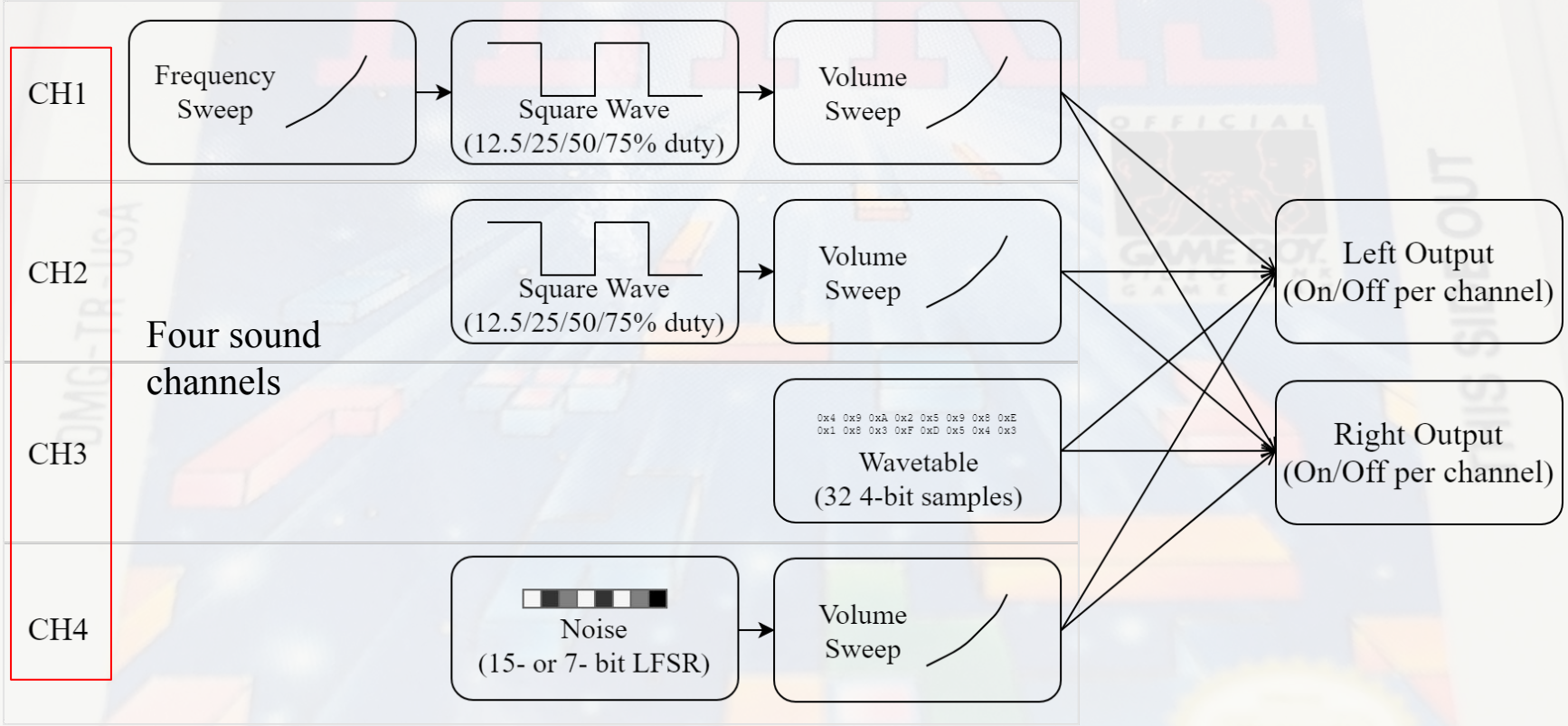
The Sound Controller


```
ld   hl,values
values_loop:
push bc
push de
push hl

push bc
pop  af

; Switch stack
ld   (temp),sp
ld   a,(hl+)
ld   h,(hl)
ld   l,a
;    call print_regs
ld   sp,hl

; Set registers
ld   h,d
ld   l,e
ld   a,$12
ld   bc,$5691
ld   de,$9ABC

jp   instr
instr_done:
; Save new SP and switch to yet another stack
ld   (temp+2),sp
ld   sp,$DF70

call checksum_af_bc_de_hl

; Checksum SP
ld   a,(temp+2)
call update_crc_fast
ld   a,(temp+3)
call update_crc_fast

ldsp temp

pop  hl
pop  de
pop  bc
inc  hl
inc  hl
ld   a,l
cp   <values_end
jr   nz,values_loop

pop  hl
ld   a,l
cp   <values_end
jr   nz,hl_loop
```

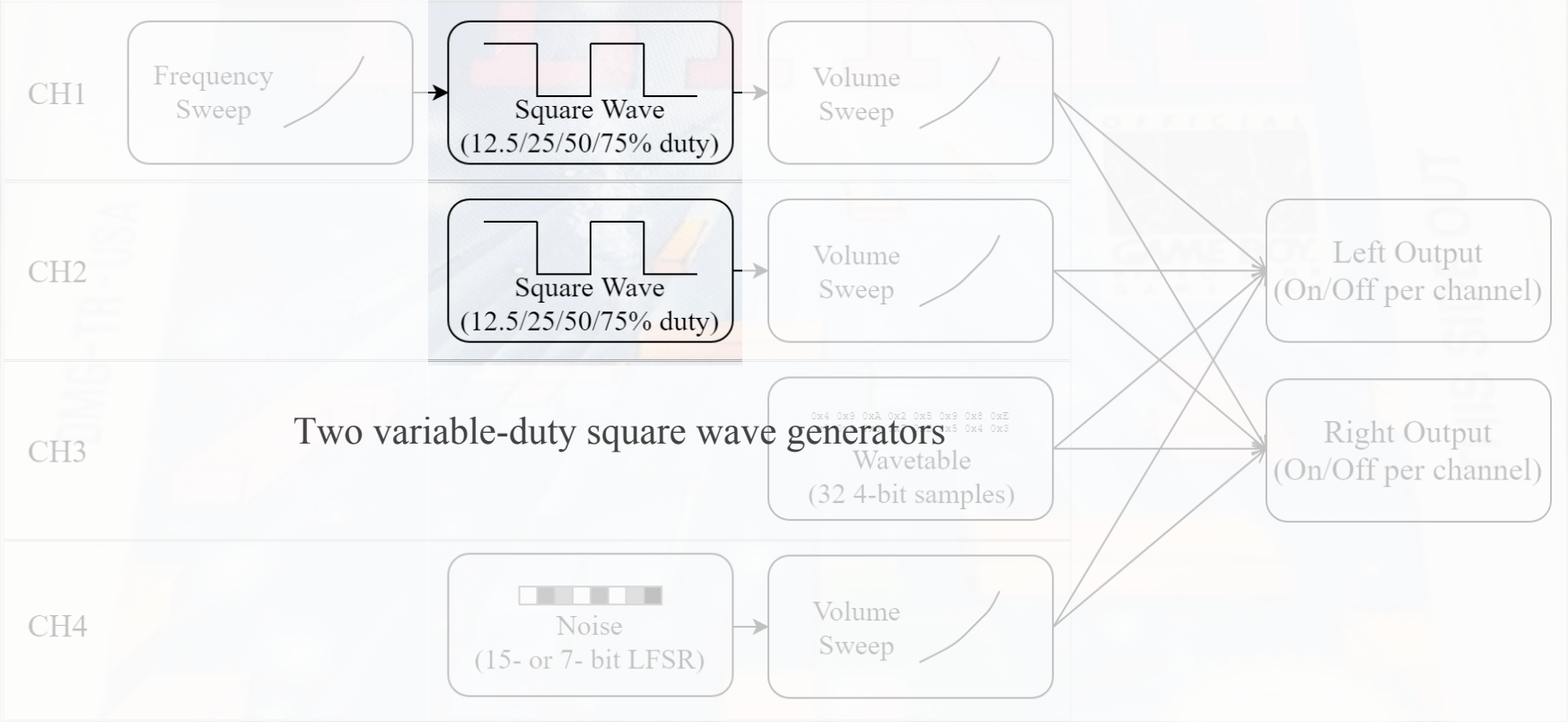
Sound Controller



Sound Controller

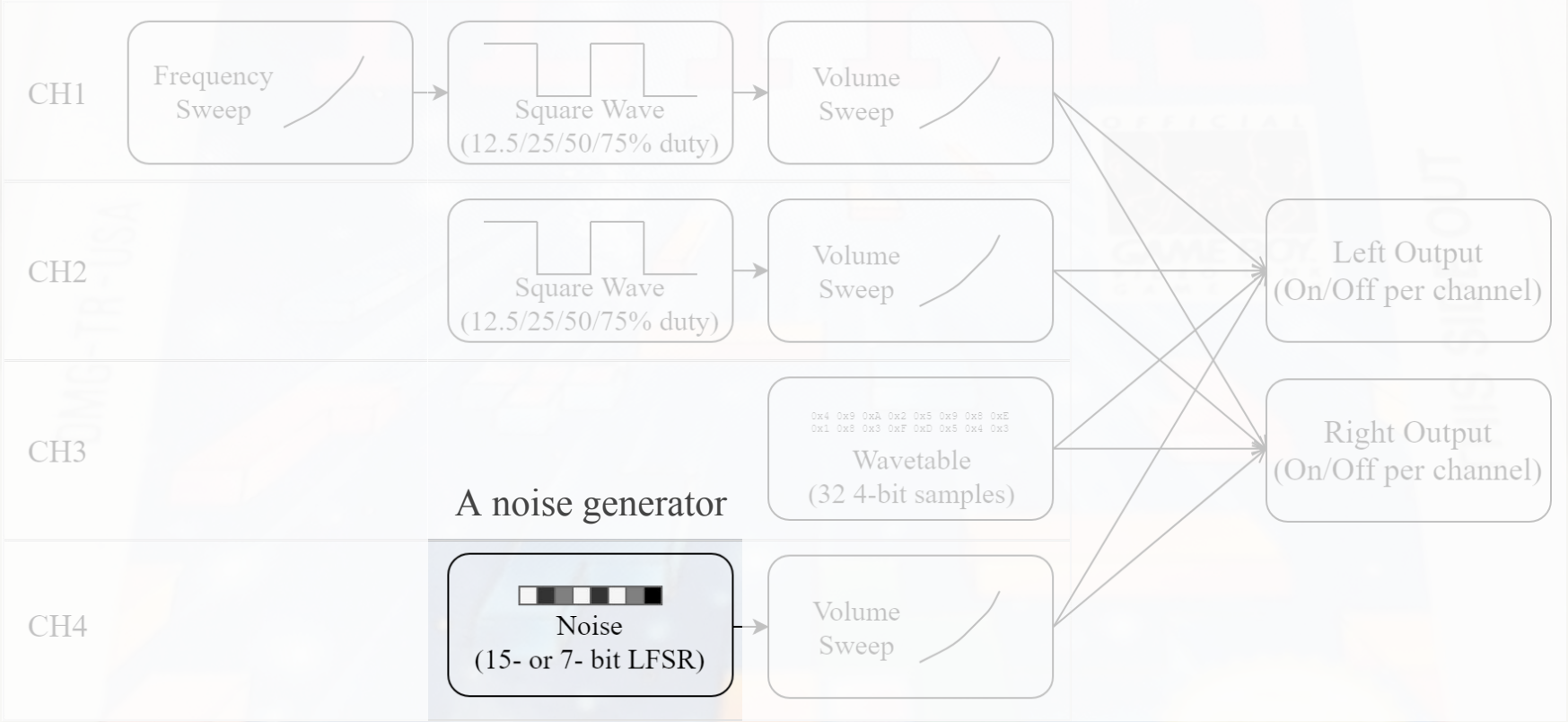


Sound Controller

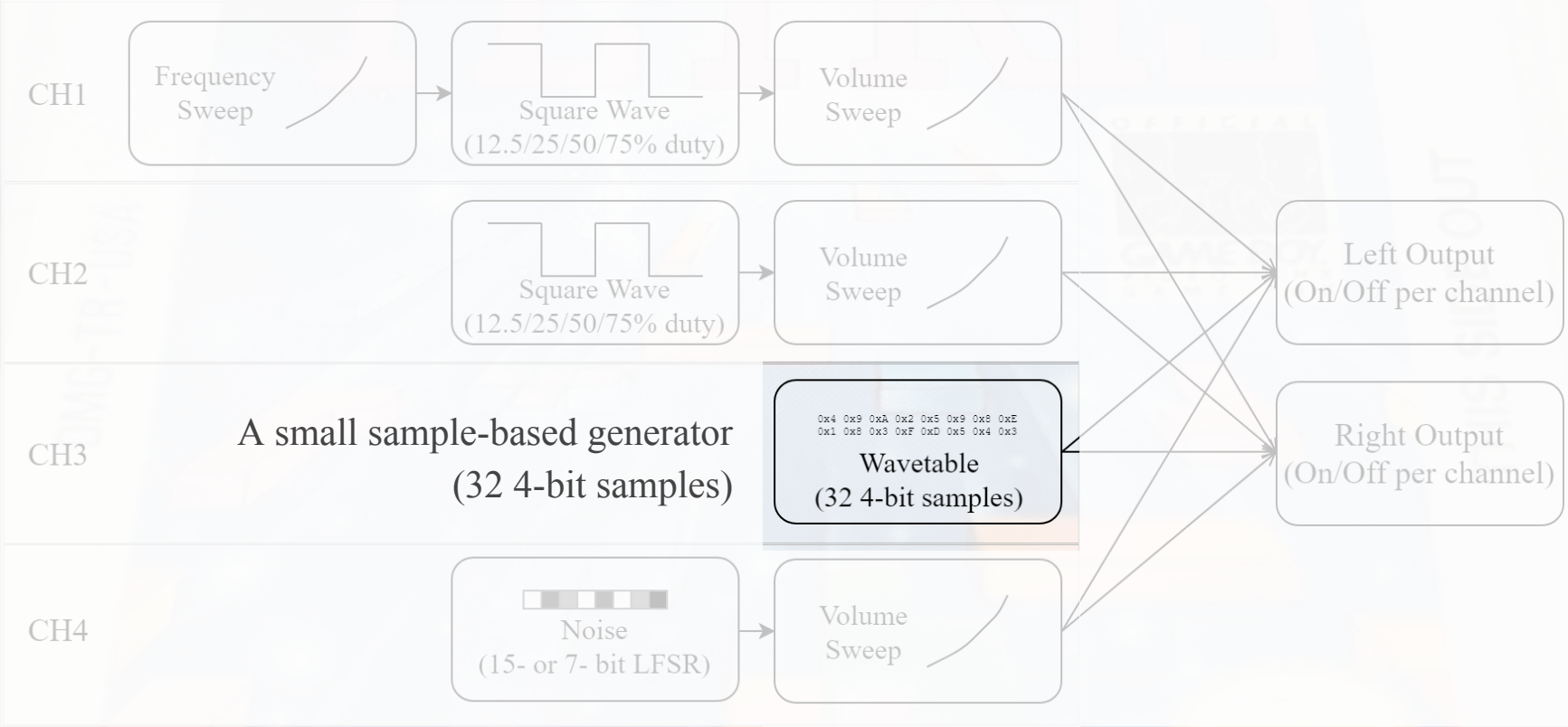


Two variable-duty square wave generators

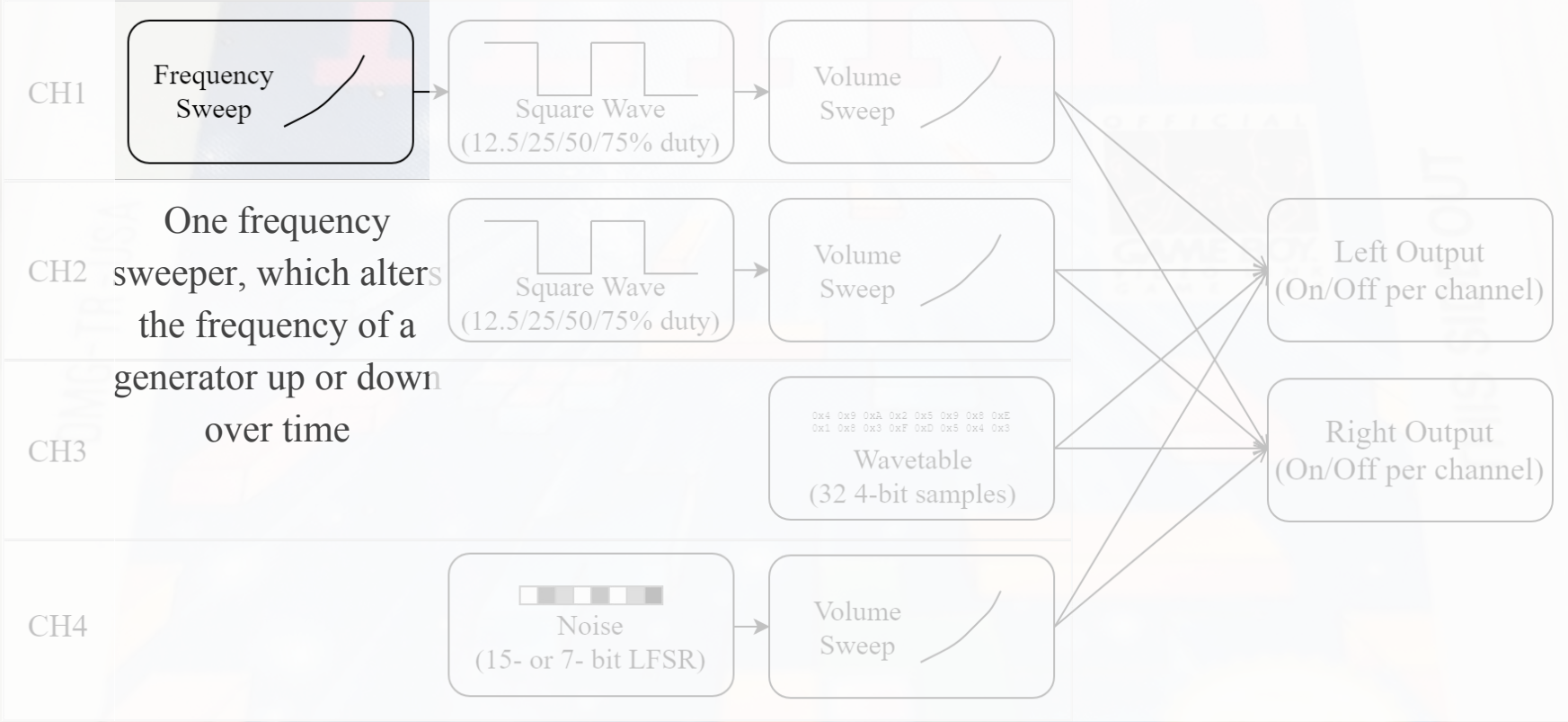
Sound Controller



Sound Controller

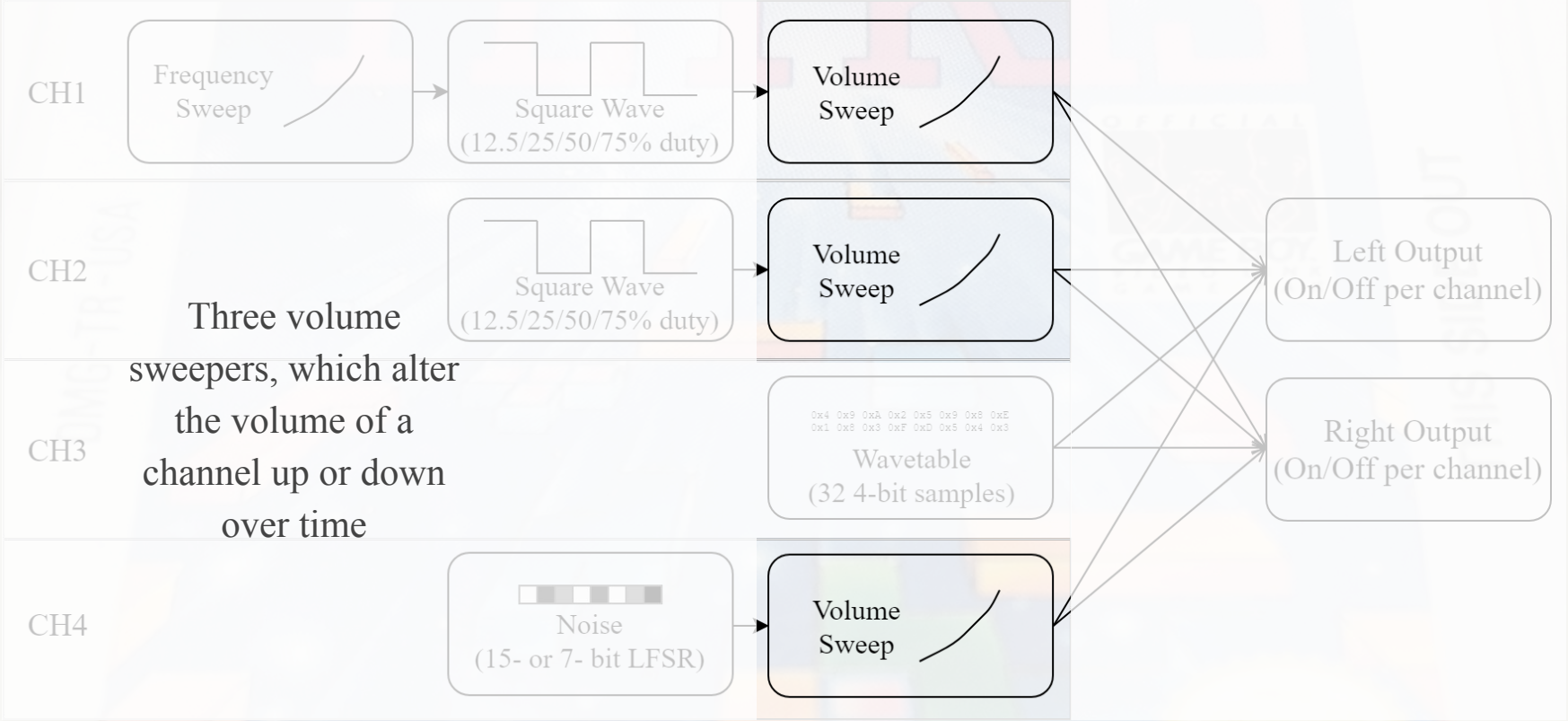


Sound Controller



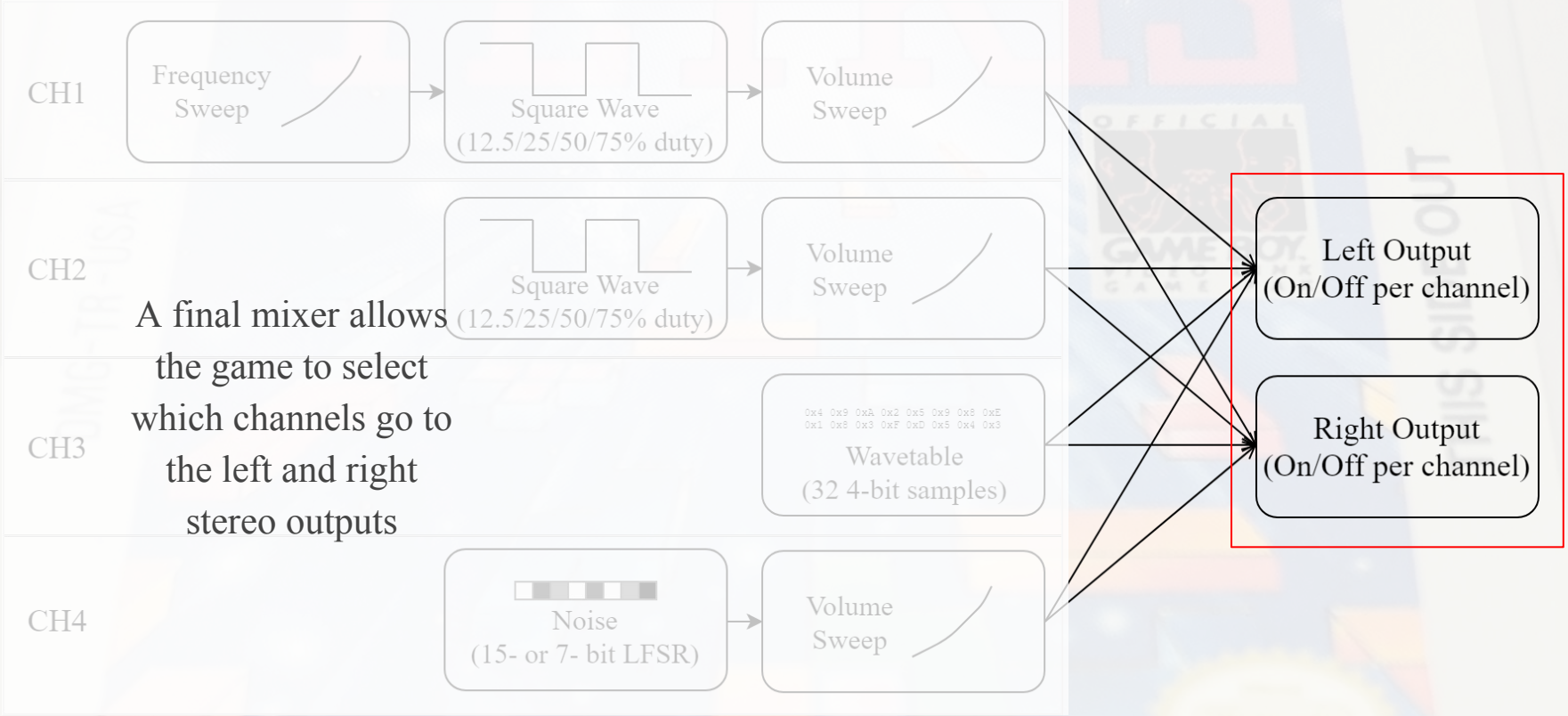
One frequency sweeper, which alters the frequency of a generator up or down over time

Sound Controller



Three volume sweepers, which alter the volume of a channel up or down over time

Sound Controller



A final mixer allows the game to select which channels go to the left and right stereo outputs

Sound Controller C++ Impl.

- Ironically, the sound controller code is quite a bit longer and more complex than the LCD code, because there are a lot of little details to get right
- There are small classes for each of the components in the sound controller - `FrequencySweep`, `VolumeEnvelope`, `SquareWaveGenerator`, `NoiseGenerator`, etc.
  - Each does a little bit of work, and their outputs are all appropriately combined together
- SDL is used to output audio
  - When it is low on samples, it asks the emulator for a new batch of 1024 samples

Other Small but Important Bits


```
ld   hl,values
values_loop:
push bc
push de
push hl

push bc
pop  af

; Switch stack
ld   (temp),sp
ld   a,(hl+)
ld   h,(hl)
ld   l,a
;    call print_regs
ld   sp,hl

; Set registers
ld   h,d
ld   l,e
ld   a,$12
ld   bc,$5691
ld   de,$9ABC

jp   instr
instr_done:
; Save new SP and switch to yet another stack
ld   (temp+2),sp
ld   sp,$DF70

call checksum_af_bc_de_hl

; Checksum SP
ld   a,(temp+2)
call update_crc_fast
ld   a,(temp+3)
call update_crc_fast

ldsp temp

pop  hl
pop  de
pop  bc
inc  hl
inc  hl
ld   a,l
cp   <values_end
jr   nz,values_loop

pop  hl
ld   a,l
cp   <values_end
jr   nz,hl_loop
```

The Timer

- The timer is a very simple piece of circuitry built around a counter
- The counter continuously increments at a chosen frequency
- When the timer counter overflows, it fires a CPU interrupt
- The exact time period between overflows/interrupts is configurable

DMA Transfer

- Inside the LCD controller, there is a small Direct Memory Access (DMA) controller that can perform block memory transfers to VRAM
- Given a source address, the DMA controller will copy 160 bytes to the area of VRAM that controls the sprite positions and layouts
  - With careful programming, this can be used to alter the sprite layout mid-frame and display more than 40 sprites on the whole screen
    - E.g. 40 sprites in the top half of the screen, and another 40 in the bottom half

Game Link Port

- The Game Link port is a simple serial port, mainly used for multiplayer
- Transfers are byte-sized with no start, stop, or parity bits
- Transfers occur in full duplex - that is, a byte is shifted out while another is shifted in from the outside world
- The bit clock can be internal or external (e.g. another Game Boy)
- For the DMG-01, transfer rate is about 1 KiB/s

Conclusion

- Emulator source is here: https://github.com/raptorofaxys/GBEmu
- Slides: https://goo.gl/jQQY3c
- If you want to write your own emulator, look online for documentation concerning your system of choice - chances are somebody's already done a lot of reverse-engineering
- If you are looking for a simple starting point, look into CHIP-8
- Hack away!

Quick Shout-Outs

- Huge thanks to Antonio Maiorano and Dominic Hudon for help reviewing and correcting the talk material!
- Thanks to Antonio Maiorano, Dominic Hudon, Jean-François Pérusse and Antoine Bouchard for many interesting emulation-related discussions!
- As usual, a huge thanks to my spouse Laurence Sylvestre-Tardif for putting my with my antics :-)

Thank You!

Questions?