BameGoy

Nintendo GameBoy Hardware Emulator

Donovan Sproule (das2313), Nicolas Alarcon (na2946), Claire Cizdziel (ctc2156)
GameBoy Specs

**SoC**: Nintendo DMG-CPU (Sharp LR35902)

**CPU**: 4.194304 MHz Sharp SM83

**Screen**: 45.5mm x 41.5mm

**Memory**: 64 KB
- On SoC: 256 B "bootstrap" ROM, 127 B High RAM
- Internal: 8 KB RAM, 8 KB Video RAM
- External: (in cartridge) up to 1 MB ROM, up to 128 KB RAM

**Resolution**: 160 (w) x 144 (h) pixels (10:9 aspect ratio)
- 4 possible colors of gray

**Power**: 70–80 mAh, 4 x AA batteries

**I/O**: joypad, audio, graphics, LCD

(https://en.wikipedia.org/wiki/Game_Boy)
GB-Z80: Using modified open-source Z80, also operating at 4.19 MHz

Timing: Instructions in multiples of 4 cycles

Interrupts: V-Blank, LCD controller, timer, serial, and joypad

I/O: performed through memory load/store instructions

Modifications: for this specific usage, there were modifications, shown in the table on the right.
Memory

64 KiB Memory allocated as follows:

GameBoy Memory Areas

<table>
<thead>
<tr>
<th>Memory Area</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SFFFF</td>
<td>Interrupt Enable Flag</td>
</tr>
<tr>
<td>SFF80-SFFFFE</td>
<td>Zero Page - 127 bytes</td>
</tr>
<tr>
<td>SFF00-SFF7F</td>
<td>Hardware I/O Registers</td>
</tr>
<tr>
<td>SFEA0-SFEFF</td>
<td>Unusable Memory</td>
</tr>
<tr>
<td>SFE00-SFE9F</td>
<td>OAM - Object Attribute Memory</td>
</tr>
<tr>
<td>SE000-SFDFE</td>
<td>Echo RAM - Reserved, Do Not Use</td>
</tr>
<tr>
<td>SD000-SDFFF</td>
<td>Internal RAM - Bank 1-7 (switchable - CGB only)</td>
</tr>
<tr>
<td>SC000-SCFFFF</td>
<td>Internal RAM - Bank 0 (fixed)</td>
</tr>
<tr>
<td>SA000-SBFFFF</td>
<td>Cartridge RAM (If Available)</td>
</tr>
<tr>
<td>S9C00-S9FFF</td>
<td>BG Map Data 2</td>
</tr>
<tr>
<td>S9800-S9BF</td>
<td>BG Map Data 1</td>
</tr>
<tr>
<td>S8000-S97FF</td>
<td>Character RAM</td>
</tr>
<tr>
<td>S4000-S7FFF</td>
<td>Cartridge ROM - Switchable Banks 1-xx</td>
</tr>
<tr>
<td>S0150-S3FFF</td>
<td>Cartridge ROM - Bank 0 (fixed)</td>
</tr>
<tr>
<td>S0100-S014F</td>
<td>Cartridge Header Area</td>
</tr>
<tr>
<td>S0000-S00FF</td>
<td>Restart and Interrupt Vectors</td>
</tr>
</tbody>
</table>

(https://crocidb.com/post/gameboy-emulator1/)
During a scanline, PPU enters specific modes with distinct functions:

Mode 2: OAM Scan
- PPU searches OAM memory for sprites and stores in buffer
- 80 T-cycles (PPU checks new OAM every 2 T-cycles)

Mode 3: Drawing
- PPU transfers pixels to LCD

Mode 0: H-Blank
- “Padding” for remainder of scanline - til reaches total of 456 T-cycles
- PPU “paused”

Mode 1: V-Blank
- “Padding“ similar to H-Blank
- Takes place at the end of every frame for longer duration
**PPU Modes & Timing**

- **mode 2**: searching objects
  - 80 T-cycles

- **mode 3**: drawing
  - 172 – 289 T-cycles

- **mode 0**: hblank
  - 87 – 204 T-cycles

- **LY = 0**
  - shortest
  - longest

- **LY = 1, 2, 3, 4**
  - longest
  - shortest

- **LY = 5**
  - shortest
  - longest

- **LY = 143, 144, 145**
  - vblank
  - 10 lines (6.5% of frame)
  - 4560 T-cycles

- **total**: 70224 T-cycles
- **59.7 fps**
Object Attribute Memory (OAM)

GameBoy can display up to **40** moveable objects/sprites
- Maximum of **10** objects per scanline
- Each object consist of **4** bytes
- PPU compares (LCDC bit 2) to determine sprite height

Object attributes in memory at **$FE00-$FE9F**

Writing to OAM:
- Data is written to a buffer in WRAM
- WRAM copied to OAM
- Direct OAM writing only works during HBlank or VBlank
Object Attribute Memory (OAM)

OAM DMA Transfer:
- Writing to $FF46 – DMA initiates DMA transfer from WRAM to OAM

Selection priority:
- Only selects first (up-to) 10 objects to be drawn; can apply to off-screen objects
- Setting Y=0 would therefore “hide” sprite

Drawing priority:
- Priority given to smaller X coordinate
- If X identical, sprite located first in OAM has higher priority

(https://www.copetti.org/writings/consoles/game-boy/)
PPU Block Diagram

- Memory
- Background Fetcher
- Background FIFO
- Sprite Fetcher
- Sprite FIFO
- Mixer
- Pixel_out
Background Fetching

Fetching pixels takes 2 T-Cycles to complete, with the process:

1. Fetch Tile No.
   a. For background pixel - offsets tile data by \(32 \times \frac{((LY + SCY) \& 0xFF) \div 8}{100}\)

2. Fetch Tile Data (Low)
   a. Fetches first byte of tile data - with offset of \(2 \times ((LY + SCY) \mod 8)\)

3. Fetch Tile Data (High)
   a. Same as Low, except next byte is read and stored

4. Push to FIFO
   a. Only executes if FIFO is fully empty
   b. Step 4 usually restarts twice before pushing - Steps 1-3 take 6 T-cycles, PPU takes 8 T-cycles to shift out all 8 pixels
Pixel FIFO

Individual pixels are pushed to the LCD one by one
- Each pixel holds the information of color, palette, sprite priority, and background priority
- The **pixel fetcher** is responsible for loading the FIFO registers

Background and Pixel FIFO are merged when pushed to LCD

(https://hacktix.github.io/GBEDG/ppu/#the-concept-of-ppu-modes)
SCX register allows for scrolling background on a per-pixel basis

- While shifting pixels out of background FIFO, start of scanline $\text{SCX mod 8}$ pixels are discarded
- Simultaneously, per-tile horizontal scrolling is handled with fetching process
- Results in PPU Mode 3 extending by SCX mod 8 cycles
Sprite Fetching

The Sprite Fetcher works very similarly to the background fetcher:

1. Fetch Tile No.
2. Fetch Tile Data (Low)
3. Fetch Tile Data (High)
4. Push to FIFO

- Tiles taken from $8000$-$87FF$ and unsigned numbering

If (X-Position of any sprite in buffer) ≤ (current Pixel-X-Position + 8), sprite fetch is initiated:
- resets the Background Fetcher to step 1 and temporarily pauses it
- pixel shifter to the LCD is also suspended

(https://gbdev.gg8.se/wiki/articles/OAM_DMA_tutorial)
Sprite Timing

Once sprite fetch is completed
- PPU starts pushing pixels to LCD
- Background fetcher is restarted
- Delay occurs if <6 pixels remaining in Background FIFO (delay = 6 - \(\text{REMAINING\_PIXEL\_COUNT}\))

Sample timing diagram:

[Diagram showing sprite timing with X=68]

(BG FIFO Reloaded)

(BG Fetcher Pixel Shifter)

(Sprite with X=68)

...Fetch BG
...Shift BG

Fetcher Paused & Reset
Sprite Fetch 6 Cycles
Shift Remaining 4 Cycles
Wait 2 Cycles
Shift BG...

4 Pixels in BG FIFO
Shifted Out

0 Pixels in BG FIFO

(https://hacktix.github.io/GBEDG/ppu/#the-concept-of-ppu-modes)
Example of tiles creating background map:
Tile data stored in **VRAM** in **$8000-97FF**

Tiles can be displayed as part of **Background/Window** and/or moveable **Sprites**

**$8000 Method:**
- $8000: base pointer
- TILE_NUMBER: unsigned 8-bit integer
- Add $8000 to (TILE_NUMBER * 16)

**$8800 Method:**
- $9000: base pointer
- SIGNED TILE_NUMBER: signed 8-bit integer
- Add $9000 to (SIGNED TILE_NUMBER * 16)
Tile Map

Game Boy contains two 32×32 tile maps in VRAM at $9800$-$9BFF$ and $9C00$-$9FFF$

Each tile map has 1 byte indexes of tiles to be displayed
- Tiles obtained through address from VRAM selected via LCDC register

Background-to-Object Priority
- BG/OBJ priority declared in 3 places: BG Map bit 7, LCDC bit 0, OAM bit 7
- Sample priority table shown below:

<table>
<thead>
<tr>
<th>LCDC bit 0</th>
<th>OAM attr bit 7</th>
<th>BG attr bit 7</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>OBJ</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>OBJ</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>OBJ</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>OBJ</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>OBJ</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>BG color 1–3, otherwise OBJ</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>BG color 1–3, otherwise OBJ</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>BG color 1–3, otherwise OBJ</td>
</tr>
</tbody>
</table>
# Interrupt Handling

<table>
<thead>
<tr>
<th>Interrupt</th>
<th>INT</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VBlank</td>
<td>$40</td>
<td>Requested every time Game Boy enters VBlank</td>
</tr>
<tr>
<td>STAT</td>
<td>$48</td>
<td>Can be triggered by various sources</td>
</tr>
<tr>
<td>Timer</td>
<td>$50</td>
<td>Requested every time the timer overflows</td>
</tr>
<tr>
<td>Serial</td>
<td>$58</td>
<td>Requested upon completion of a serial data transfer (8 serial clock cycles after starting transfer)</td>
</tr>
<tr>
<td>Joypad</td>
<td>$60</td>
<td>Requested when a button is pressed</td>
</tr>
</tbody>
</table>
The GameBoy is capable of 4 colors
- Each color takes 2 bits
- Each tile in tile data set is held in 16 bytes

Each color is achieved with the equation:
- \((\text{Each bit of first byte}) + (\text{bit in same position on second byte})\) => calculates color number
Scrolling

GameBoy scrolling is 160x144 pixels, but background map is 256x256 pixels → scrolling

- Background is defined at the top-left of screen
- By moving this point between frames, the background can scroll
- Top-left is defined by registers Scroll X, Scroll Y
Proper timing ensure proper synchronization with the CPU, frame rate and refresh rate, memory access, and rendering pipelines (For testing)

- Calibrated clock cycles using gtkwave
- Example final timing diagram of PPU
Timing

Example timing diagram of memory mechanism

- "dots [8:0]" represents instructions processed by the PPU, for the sake of testing
- System stalls during testing but not deployment
- "dot" increments, then moves onto the next instruction.
PPU Testing

Using interactive test benches, gradually built up capabilities of PPU

- Example images of initial PPU testing
- Found issues with timing and synchronization
PPU Testing

Tile map drawn from one tile

Tile map drawn from multiple tiles
**PPU Testing**

Sprites implemented

Sprites implemented and handles LCDC flags, window, multiple BG maps, and scrolling

DONE!
Video

GameBoy Video: 160x144 pixels
VGA: 1280 x 1024 pixels @ 60Hz

VGA Game Window: **960 x 864 pixels** (centered on screen)
- GameBoy pixel data written into framebuffer @ 4MHz
- VGA reads framebuffer with pixel frequency @ 108 MHz
  - One pixel every cycle
  - Referenced Lab 3 as a VGA scaling resource
- Background set to Pantone 292!

<table>
<thead>
<tr>
<th>General timing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Screen refresh rate</td>
</tr>
<tr>
<td>Vertical refresh</td>
</tr>
<tr>
<td>Pixel freq.</td>
</tr>
</tbody>
</table>

(http://www.tinyvga.com/vga-timing/1280x1024@60Hz)
NES Controller
- Based on states of output pins (P14-15) and input pins (P10-13), CPU can identify a button press
- Configured to our CPU using libusb (referenced lab2)
Obtained Tetris cartridge from online open source

Cartridge header ($0100-$014F) provides the information for running a game, namely:

- 0147 - Cartridge type
- 0148 - ROM Size
- 0149 - RAM Size

Cartridge ROM loaded to On-Chip RAM on DE1-SoC

PPU Compilation

Analysis and Synthesis Resource Utilization of PPU

---

### Analysis & Synthesis Resource Utilization by Entity

<table>
<thead>
<tr>
<th>Compilation Hierarchy Node</th>
<th>Combinational ALUTs</th>
<th>Dedicated Logic Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>734 (729)</td>
<td>488 (456)</td>
</tr>
<tr>
<td>1</td>
<td>3 (3)</td>
<td>16 (16)</td>
</tr>
<tr>
<td>2</td>
<td>2 (2)</td>
<td>16 (16)</td>
</tr>
</tbody>
</table>

---

### Flow Summary

<table>
<thead>
<tr>
<th>Flow Status</th>
<th>Successful - Sat Aug 3 15:18:00 2024</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quartus Prime Version</td>
<td>21.0 Build 842 10/21/2021 SJ Lite Edition</td>
</tr>
<tr>
<td>Revision Name</td>
<td>soc_system</td>
</tr>
<tr>
<td>Top-level Entity Name</td>
<td>soc_system_top</td>
</tr>
<tr>
<td>Family</td>
<td>Cyclone V</td>
</tr>
<tr>
<td>Device</td>
<td>5CSEMA5F31C6</td>
</tr>
<tr>
<td>Timing Models</td>
<td>Final</td>
</tr>
<tr>
<td>Logic utilization (in ALMs)</td>
<td>3,576 / 32,070 (11%)</td>
</tr>
<tr>
<td>Total registers</td>
<td>2493</td>
</tr>
<tr>
<td>Total pins</td>
<td>356 / 457 (78%)</td>
</tr>
<tr>
<td>Total virtual pins</td>
<td>0</td>
</tr>
<tr>
<td>Total block memory bits</td>
<td>705,792 / 4,065,280 (17%)</td>
</tr>
<tr>
<td>Total DSP Blocks</td>
<td>0 / 87 (0%)</td>
</tr>
<tr>
<td>Total HSSI RX PCSs</td>
<td>0</td>
</tr>
<tr>
<td>Total HSSI PMA RX Deserializers</td>
<td>0</td>
</tr>
<tr>
<td>Total HSSI TX PCSs</td>
<td>0</td>
</tr>
<tr>
<td>Total HSSI PMA TX Serializers</td>
<td>0</td>
</tr>
<tr>
<td>Total PLLs</td>
<td>1 / 6 (17%)</td>
</tr>
<tr>
<td>Total DLLs</td>
<td>1 / 4 (25%)</td>
</tr>
</tbody>
</table>

---

### Analysis & Synthesis Resource Usage Summary

<table>
<thead>
<tr>
<th>Resource</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Estimate of Logic utilization (ALMs needed)</td>
<td>3694</td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3 CombinationALUT usage for logic</td>
<td>5620</td>
</tr>
<tr>
<td>1 -- 7 input functions</td>
<td>175</td>
</tr>
<tr>
<td>2 -- 6 input functions</td>
<td>1518</td>
</tr>
<tr>
<td>3 -- 5 input functions</td>
<td>1256</td>
</tr>
<tr>
<td>4 -- 4 input functions</td>
<td>1102</td>
</tr>
<tr>
<td>5 -- &lt;=3 input functions</td>
<td>1629</td>
</tr>
<tr>
<td>5 Dedicated logic registers</td>
<td>2211</td>
</tr>
<tr>
<td>6 I/O pins</td>
<td>356</td>
</tr>
<tr>
<td>7 I/O registers</td>
<td>186</td>
</tr>
<tr>
<td>9 Total MLAB memory bits</td>
<td>0</td>
</tr>
<tr>
<td>10 Total block memory bits</td>
<td>705792</td>
</tr>
<tr>
<td>11</td>
<td></td>
</tr>
<tr>
<td>12 Total DSP Blocks</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td></td>
</tr>
<tr>
<td>14 Total PLLs</td>
<td>4</td>
</tr>
<tr>
<td>15</td>
<td></td>
</tr>
<tr>
<td>16 Total DLLs</td>
<td>1</td>
</tr>
<tr>
<td>17 Maximum fan-out node</td>
<td>soc_system_soc_system</td>
</tr>
<tr>
<td>18 Maximum fan-out</td>
<td>903</td>
</tr>
<tr>
<td>19 Total fan-out</td>
<td>37130</td>
</tr>
</tbody>
</table>
Game Play Testing with PPU

Game Boy boot screen

Dr. Mario start screen
We utilized an open source emulator to base our systems on:

- Used functional CPU in conjunction with our **PPU, hardware/software interface**, and **peripherals (joypad)**
- Independently tested our PPU with testbenches, performing successfully in simulation
  - Ran into issues with integration into larger emulator
  - Confirmed PPU issues by successfully running open source game play
Tested open source CPU to ensure functionality

- Generated Qsys file shown below
- Managed interconnects and removed SDRAM (used utilized on-chip RAM instead)
Open Source Game Play

Confirmed the function of our peripherals and hardware/software interface through integration with open source emulator, shown below.

Tetris game play

Dr. Mario game play
Demo!