FPGABoy

An Implementation of the Nintendo GameBoy on an FPGA

Trevor Rundell, Oleg Kozhushnyan

6.111 Final Project

May 14, 2009

Table of Contents

Table of Figures

Table of Tables

Abstract

FPGABoy is an re-implementation of the classic GameBoy system, built on an FPGA. The system was developed almost entirely from scratch using an open source implementation of a Z80 processor.

System Overview

At the highest level, a game system is just a box that takes some sort of user input, and produces video and audio as an output. We decided to implement such a system, but with a twist. Instead of creating our own, we decided to pay homage to one of the greatest hand held game consoles ever created, the Nintendo GameBoy.

The GameBoy was released in 1989 and since then has been very well documented by the hacker community. We used all the detailed documentation [1, 2] to our advantage when engineering our version of the legendary console. During our design stage we broke the system up into the components shown in Figure 1.

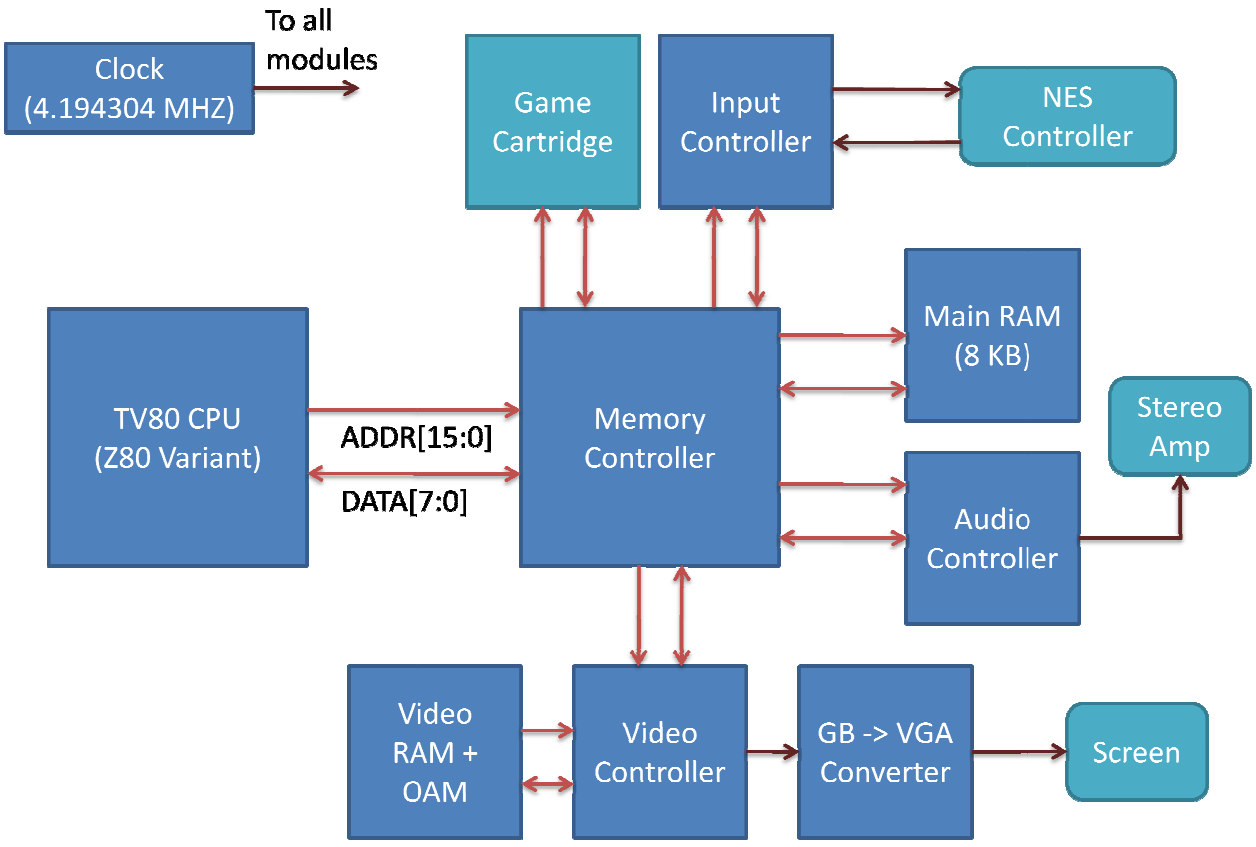


Figure 1. Global system diagram

At the core, we decided to use an open source implementation of the Z80 CPU known as TV80 in order to lessen our workload so we could concentrate on implementing the game system itself. The CPU communicates with the rest of the system through a module known as the Memory Controller. This module acts as a giant switch and handles the communication between the CPU and the rest of the sub modules.

The two primary sub modules that were at the top of our list for implementation were the Input Controller and the Video Controller. These two modules make up the meat of the project and are absolutely necessary for a functioning game system. Next was the Game Cartridge module and the Video Converter that handled interfacing between the FPGABoy and the cartridge and display peripherals. The third sub module known as the Audio Controller was placed low on our list and ultimately never implemented as it was not vital for the full system functionality.

We developed the design in Figure 1 very early on and stuck to it through the whole project. In the end, our final implementation looks nearly identical, at a high level, to our initial design.

Module Descriptions

Central Processing Unit (Trevor)

The original GameBoy uses a custom LR35902 processor, a variant of the Zilog Z80 8-bit processor. This processor has a well slightly different instruction set from the original Z80, but it is still very well documented. Rather than implement our own CPU, we decided to use the open source TV80 core [4], created by Guy Hutchison. The TV80 has built-in support (albeit experimental) for the GameBoy instruction set out of the block which greatly reduced our construction time for this module. However, the core is not without some extremely subtle bugs and a great deal of time was invested to track these down and fix them. As a result, FPGABoy contains a number of debugging features that expose various elements of the CPU's internal state. Additionally, breakpoints and instruction stepping allowed examination of the CPU state at very precise locations within the game's themselves.

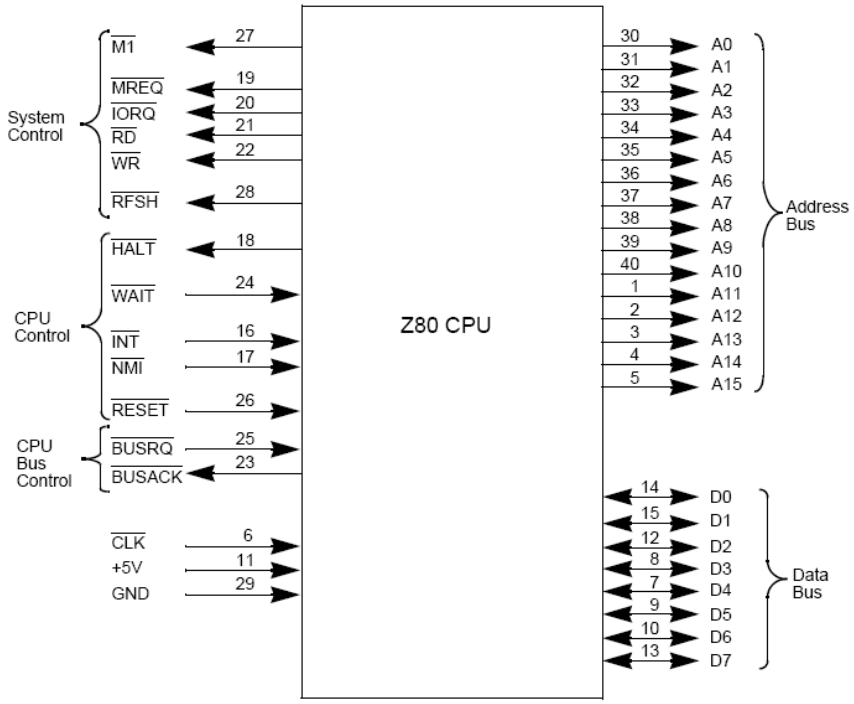


Figure 2: The Z80 pin configuration *(Source: www.zilog.com/docs/z80/um0080.pdf)*

CPU Debugging Features

Hex Display Debug Readout

We tried to make good use of the 64-bit hex display readout on the labkit to display as much information about the CPU's state as possible. To accomplish this the display was divided into five sections. The left most 8 digits are used to display the values on the CPU's address and data busses. Right right most 8 digits are used to display any one of five register combinations. By pressing buttons 0-3 and Enter on the labkit a user can switch between these register outputs to see every major aspect of the CPU state that can be controlled by a GameBoy game.

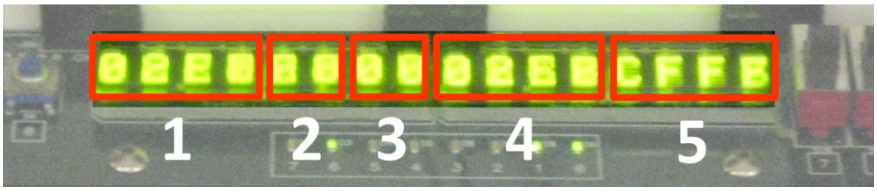


Figure 3: The hex display debug readout

Display	Readout
1	16 bit address bus
2	8 bit data in bus
3	8 bit data out bus

4	16 bit selectable register 0 -> AF 1 -> DE 2 -> PC 3 -> IME flags Enter -> N/A
5	16 bit selectable register 0 -> BC 1 -> HL 2 -> SP 3 -> Int Req / Ack Flags Enter -> Breakpoint Addr

LED Debug Readout

The LED's on the labkit are tied to various pins on the CPU. This allowed us to determine the I/O state of the CPU very quickly.

Table 1: LED debug readout

LED	Pin	LED	Pin
0	$\overline{\text{M1}}$	4	$\overline{\text{HALT}}$
1	$\overline{\text{MREQ}}$	5	$\overline{\text{RESET}}$
2	$\overline{\text{IORQ}}$	6	$\overline{\text{RD}}$
3	$\overline{\text{INT}}$	7	$\overline{\text{WR}}$

Debugging Switches

The labkit's switches can be used to facilitate fast debugging of the CPU.

Table 2: Debugging switches

Switch	Description
0	Power/Reset (switch off to reset)
1	Step enable
2	Breakpoint enable
3	Advance clock (when in step mode)
4-7	Set breakpoint digit

Breakpoint Module

A breakpoint module allows FPGABoy to halt execution when specific instructions are reached. A breakpoint can be set at any time using the hex readout and the labkit switches. The CPU address bus is continuously compared with this value and when the two match the CPU will

pause if breakpoints are enabled. A user can then step through instructions manually, resume normal execution or set a new breakpoint.

CPU Bugs

There were three bugs in the TV80 implementation that had to be fixed before FPGABoy was playable. These ranged from fairly straightforward to extremely subtle. We intend to submit our patches back to the TV80 core repository.

Carry Flag

We noticed that the 16-bit addition instruction LD HL,SP+DD was generating incorrect output. Upon further investigation it was revealed that the ALU's carry flag was not being properly cleared, effectively resulting in the second half of the add instruction always operating as if the carry flag were set. This caused results like the following: $8000h + 0020h = 8120h$.

We were able to fix this by rearranging the locations of the various ALU flags within the flags register itself. The problem ultimately stemmed from a slight difference in the positions of these flags between the GameBoy processor and an actual Z80.

I/O Port Loads

The instruction LD (FF00+C), A was executing as if it were LD (BC),A. This was causing routines that executed in the high ram area of the memory map to execute as NOPs instead. This was a fairly simple fix once we identified the bug.

RETI

The RETI instruction was failing to re-enable the global interrupt flag after returning from an interrupt. This caused many games to lock up or get stuck in menus while they waited for interrupts that never came.

Memory Controller (MMU) (Oleg)

The memory controller is one of the most important components of the FPGABoy system as it ties together all the other modules to allow them to communicate with the CPU. Essentially it acts as a large multiplexer that connects the address and data bus of the CPU to the other modules. In addition to handling the memory map, the module implements the ability for the video module to perform direct memory access(DMA). The DMA component is vital to the video module as it provides the fastest way to copy data into sprite memory. For more details on the video module refer to the corresponding section.

Memory Map

The memory map implemented by the memory controller takes the devices such as the input converter, video module, timer module and the interrupt module and assigns them to regions

in a 16-bit address space. Table 3 is an overview of the memory map as implemented by the FPGABoy.

Table 3. Memory map ***(Source: http://nocash.emubase.de/pandocs.htm#memorymap)***

Start Address	End Address	Mapping
0x0000	**0x3FFF**	16KB ROM Bank 0
0x4000	**0x7FFF**	16KB ROM Bank 1
0x8000	**0x9FFF**	8KB Video RAM
0xA000	**0xBFFF**	8KB External RAM
0xC000	**0xCFFF**	4KB Internal RAM Bank 0
0xD000	**0xDFFF**	4KB Internal RAM Bank 1
0xE000	**0xFDFF**	Echo of Internal RAM (0xC000 - 0xDDFF)
0xFE00	**0xFE9F**	160B Sprite RAM (OAM)
0xFEA0	**0xFEFF**	Reserved
0xFF00	**0xFF7F**	I/O Ports
0xFF80	**0xFFFE**	High RAM
0xFFFF		Interrupt Enable Register

The only time the memory map differs from Table 3 is during the start up sequence. When the FPGABoy first starts executing, the MMU maps the first 0x100 bytes onto a special boot ROM. For more information on the boot ROM, refer to the corresponding section. Afterwards, the first 32KB of the address space are always mapped to the read only block RAM that contains the copy of the GameBoy game. The next 32KB of the address space are taken up by the mappings to the various RAM's in the system. The 8KB external and 8KB internal RAM are implemented by block memories inside the MMU itself while the 8KB of video memory resides in the video module. Following that is 160B of sprite memory that also resides in the video module. The remainder of the mapping is either reserved or is associated with the registers to control various devices.

An important aspect of the memory map module is the ability to access memory banks. For GameBoy games larger than 32KB it is necessary to be able to modify the memory map to be able to access the remainder of the game data. This is done through the memory bank controller that is embedded in the game cartridge. The controller can remap the locations occupied by the 16KB ROM Bank 1 and 4KB RAM Bank 1 to be able to access an address space much larger than available. We do not implement the memory bank controller functionality but it is necessary to explain why the ROM and RAM are partitioned as they are and appear with different Bank numbers.

The most interesting area of the memory map is the region from 0xFF00 to 0xFF7F. That region is populated by the control registers for all the other modules in the system. Instead of implementing the registers in the memory module itself, they are implemented inside the

modules that they control. That is, the memory map module just redirects the address bus, data bus, and read/write flags to the respective modules for any accesses in the I/O region. It is up to the modules themselves to handle latching data and replying to the memory accesses.

Table 4. Memory module control registers

Address	Name	Function
0xFF50	ROM_DISABLE	Disable boot ROM mapping
0xFF46	DMA	Start DMA transfer

The memory map module is controlled through two registers. They are called the ROM_DISABLE and DMA registers located at 0xFF50 and 0xFF46 respectively. The ROM_DISABLE register handles the temporary boot ROM mapping during FPGABoy initialization. When the boot ROM completes, it writes the value 0x01 into the ROM_DISABLE register [2] which causes the memory module to remap the first 0x100 bytes back onto the 16KB ROM Bank 0.

Direct Memory Access (DMA)

The DMA register is used to activate the direct memory transfer between any address in the range 0x0000-0xF19F and the sprite memory(OAM) at 0xFE00-0xFE9F [1]. The DMA is activated when the GameBoy game writes the source address of the transfer into the DMA register. Because the register is only 8-bits wide, all transfers have to start on a 0x100 byte boundary as the lowest 8-bits of the address are ignored [1].

Once the DMA starts, the memory map module stores the starting address in an internal register and disables the ability of the CPU to access all addresses except for the high RAM. Thus it is vital for the DMA to only be initiated by code that resides in the high RAM, otherwise the behavior will be unspecified. While the CPU executes in high RAM, the memory module copies 0xA0 bytes from the start address and into the OAM memory. Once the transfer completes, the mappings are restored and the CPU is free to jump back into normal execution.

The benefit of DMA is that it allows for the whole sprite table to be copied in one memory operation instead of requiring 160 memory accesses and instructions which can be used to perform other game operations.

Essentially, the DMA process is implemented as a minor FSM within the memory map module. It copies one memory location every four cycles during which it loops through its states. In the first state it enables writes to video memory and then transitions to the second state. Then it waits for one clock cycle for the write to stabilize and then transitions into the third state which disables writes to the video module. In the fourth and final state it increments the memory location counter and upon completion disables the DMA operation.

Bootstrap Rom

One important feature of FPGA is the inclusion of the original GameBoy bootstrap ROM. This is a small, built-in, program that is executed when the system is first powered on. One function of this program is to initialize the memory mapped I/O registers to their proper values. The bootstrap ROM also acts as a primitive form of DRM to prevent unlicensed games from being played. It does this by performing a checksum on small range of bytes in the cartridge's memory. If the sum of these bytes matches the expected value, the system disables the bootstrap ROM and allows the lower 255 bytes of the cartridge memory to be accessed. If the checksum fails, the system will lock up and must be reset.

The most recognizable aspect of the bootstrap ROM is the Nintendo logo that appears at the top of the screen when the system is powered on. The logo is then scrolled down to the center of the screen and two tones are played, producing the familiar "da-ding" sound, after which normal execution of the cartridge program begins. Despite the importance of the bootstrap ROM to the GameBoy experience, most GameBoy emulators do not implement this functionality. We obtained a copy of the original GameBoy bootstrap ROM online [2].

Figure 4: The Nintendo logo as generated by the bootstrap ROM

Input Converter Module (Oleg)

The input converter module is responsible for interfacing the CPU with the Nintendo Entertainment System(NES) game pad controller. The primary function of the module is to de-serialize the signal generated by the NES game pad and to convert it into the format which is determined by the GameBoy specification.

NES Game Pad Controller

We selected the NES Game Pad as an input source for the FPGABoy system due to its superiority over the inputs provided on the lab kit and the simplicity of its data protocol. Furthermore it turned out that most of the inputs on the lab kit were used for debugging purposes which made the NES controller a perfect source for providing game input. The game pad has a total of eight buttons which are exactly the same as the buttons found on the

original GameBoy system. The buttons are the directional pad that provides the up, down, left and right inputs along with four others buttons labeled A, B, start and select.

In order to communicate with the game pad we had to understand the way it generates its output. The internal diagram of the NES game pad can be seen in Figure 5.

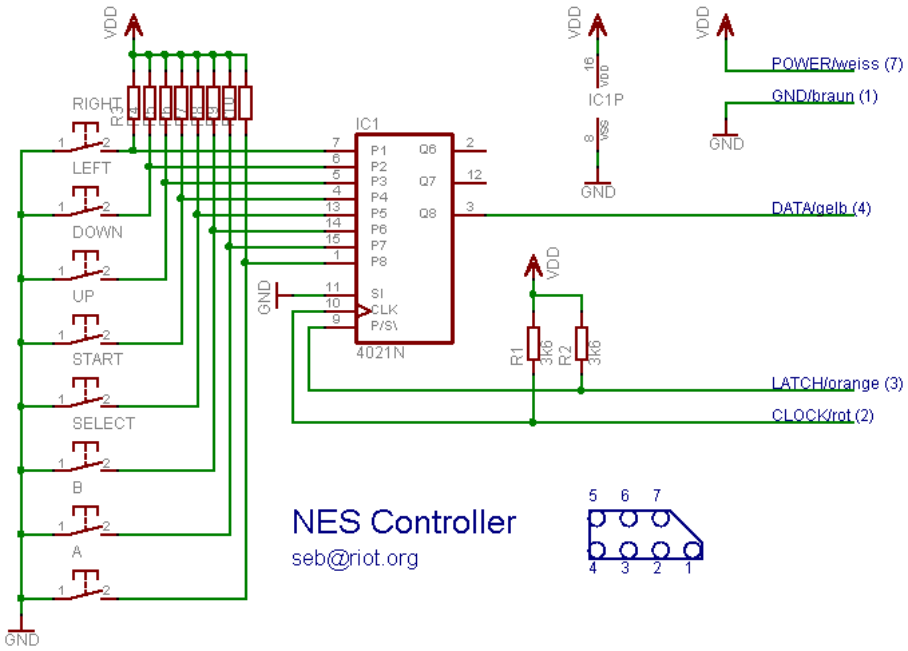


Figure 5. NES Controller schematic *(Source: http://seb.riot.org/nescontr/)*

The design of the game pad is very simple. There are eight button inputs that are pulled up high when they are not pressed. From this we can tell that we will read a value of one when a button is not pressed and a zero when it is held down as it will be shorted to ground. These inverted states are identical to the way the GameBoy expects to read the buttons as an input. The eight buttons are in turn connected to a 4021N chip. The 4021N is an eight bit shift register that converts its inputs into a serial stream.

In order to interact with the game pad, the state of the 4021N needs to be extracted. In order to do this three lines of communication are required. The lines three lines are the Data, Latch and Clock lines which are respectively the output and inputs of the 4021N chip. Whenever the Latch line goes high, the shift register starts to output its state. It outputs each one of its bits(which represent the button states) on the rising edge of the clock that is provided as an input on the Clock line. The following figure describes the communication protocol with the shift register.

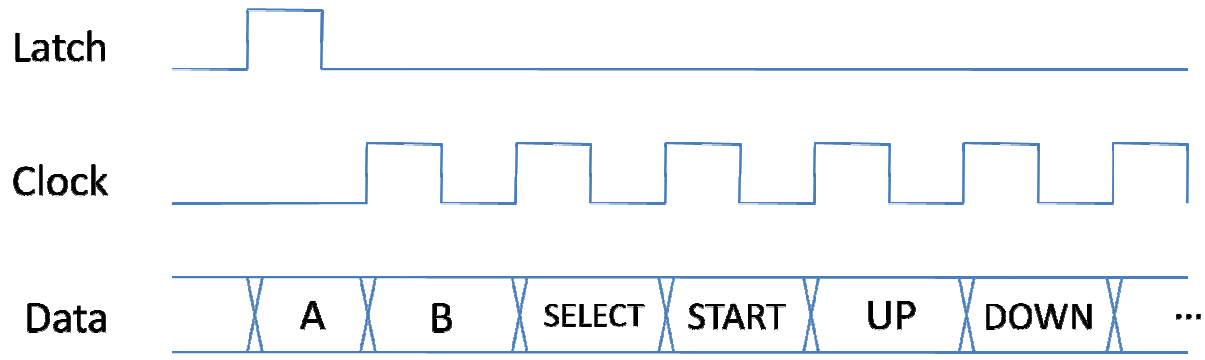


Figure 6. NES game pad communication protocol *(Source: http://seb.riot.org/nescontr/)*

Once all eight bits are read from the register, the latch can be pulled up again and the read sequence repeated. The input converter module repeats this sequence sixty times per second in order to guarantee that the user has the most responsive experience. Once the input converter has the state of all eight buttons in its internal register it is able to convert the input into the GameBoy format.

GameBoy Input Specification

The original GameBoy expects the input from the game pad in a distinct format. This format was the result of the limitation of the number of pins that were on the original CPU [1]. The limitation comes from that only six pins are used to read the state of all eight buttons. Two of the pins are used as selectors for either the four directional buttons or the A, B, select and start buttons. The other four pins read the state of the buttons selected by the previous four. The six pins are memory mapped directly into a register located at the 0xFF00 memory address [1] in the format shown in Figure 7.

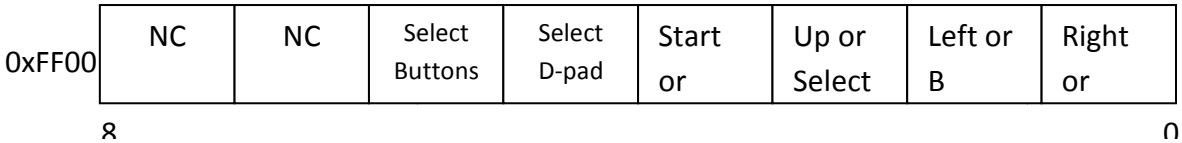


Figure 7. Input register format

When code that desires to read the input executes on the GameBoy, it writes a zero into either bit four or five of the input register to select which set of controls to read. Then it reads from the same register and extracts the state of the selected buttons from the lowest four bits. This is the functionality is supported by the converter.

Input Converter

The input converter is a combination of the previous two components and a conversion step. When the CPU asks to write into the register at 0xFF00, the input converter remembers the value that was written in order to determine the selector bits. Next, when the CPU reads from the register, the module looks at the remembered selector bits and writes the corresponding

button states onto the data bus. Because the button states are updated sixty times per second the FPGABoy is guaranteed to have the latest input at its disposal.

Interrupt Module (Trevor)

The original GameBoy is capable of generating five interrupts for various I/O devices within the system. FPGABoy supports all five interrupts, but the surrounding hardware is only capable of generating four of these. The exception is the serial transfer interrupt which is left unimplemented.

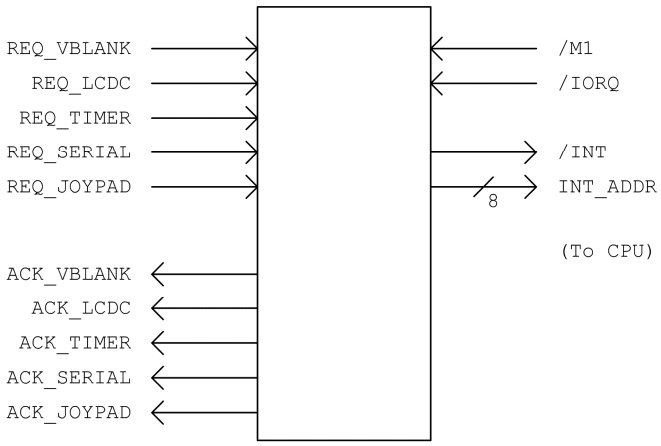


Figure 8: Interrupt module block diagram

For each interrupt there is one request pin (input) and one acknowledge (output) pin. These pins can be connected to whichever module is in charge of generating that particular interrupt.

The input module also maintains several direct connections to the CPU. The $\overline{\text{INT}}$ output can be asserted to request an interrupt to the CPU. The $\overline{\text{M1}}$ and $\overline{\text{IORQ}}$ inputs are used by the CPU to acknowledge that the interrupt request has been received. Lastly, the INT_ADDR output is the lower half of a 16-bit address that the CPU uses to determine which address to jump to when handling an interrupt.

I/O Registers

The interrupt module has two I/O registers which can be accessed through the MMU.

Interrupt Enable (IE) – FFFF (R/W)

```
Bit 0: V-Blank  Interrupt Enable  (INT 40h)  (1=Enable)
Bit 1: LCD STAT Interrupt Enable  (INT 48h)  (1=Enable)
Bit 2: Timer    Interrupt Enable  (INT 50h)  (1=Enable)
Bit 3: Serial   Interrupt Enable  (INT 58h)  (1=Enable)
Bit 4: Joypad   Interrupt Enable  (INT 60h)  (1=Enable)
```

Interrupt Flags (IF) – FF0F (Read Only)*

```
Bit 0: V-Blank  Interrupt Request (INT 40h)  (1=Request)
Bit 1: LCD STAT Interrupt Request (INT 48h)  (1=Request)
Bit 2: Timer    Interrupt Request (INT 50h)  (1=Request)
Bit 3: Serial   Interrupt Request (INT 58h)  (1=Request)
Bit 4: Joypad   Interrupt Request (INT 60h)  (1=Request)
```

The lower five bits of the IF register are tied directly to the corresponding REQ pin for each interrupt. The upper three bits are tied to ground.

* In the original GameBoy this register is R/W so games may force an interrupt request at any time. For simplicity, this register is implemented as a read only in FPGABoy. Any writes made to this location are ignored. This is consistent with other GameBoy emulator implementations.

Enabling Interrupts

On CPU reset, interrupts are initially disabled. They must be enabled in two places by the game before they can be used. The first is the global interrupt enable register which can be enabled and disabled using the EI, DI and RETI instructions. These are handled internally within the CPU. Games can also individually enable each of the five device interrupts by writing to the IE register at FFFFh.

Requesting Interrupts

Interrupts are generated by various modules within the system. For each interrupt, a device connects to the interrupt module's corresponding REQ and ACK pins. To request an interrupt the device asserts the REQ signal. This signal should remain asserted until the interrupt is acknowledged by the interrupt module.

When an interrupt is requested by a device, it is not necessarily passed through to the CPU, or immediately acknowledged. If the interrupt has not been enabled by the corresponding bit of the IE register, the request will be ignored until the bit becomes set at a later time. However, if the interrupt is enabled, the interrupt module asserts the $\overline{\text{INT}}$ signal and enters a wait state until the CPU acknowledges the request by the asserting both the $\overline{\text{M1}}$ and $\overline{\text{IORQ}}$ signals. If multiple requests are asserted on the same clock cycle, the interrupt with the lower priority is handled first. The order of precedence corresponds to the order of bits in the IE register, with lower bits having higher priority.

When the CPU interrupt acknowledge occurs, the module places the proper interrupt address on the INT_ADDR bus and the original request is acknowledged to the requesting device by asserting the corresponding ACK signal. This sequence can be represented by the following state transition diagram.

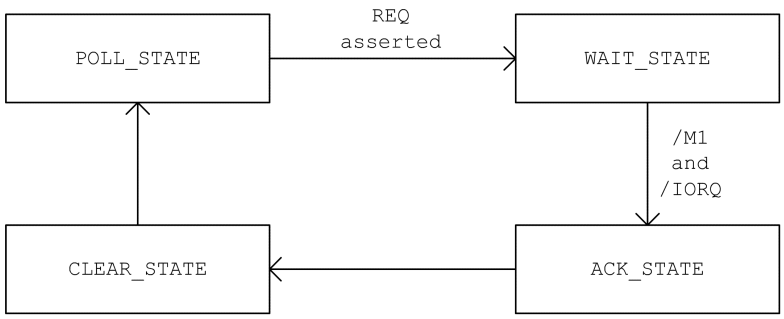


Figure 9: Interrupt module state transition diagram

Assuming the global interrupt enable flag is set, the timing of these states defined by the Z80 specification to be as follows.

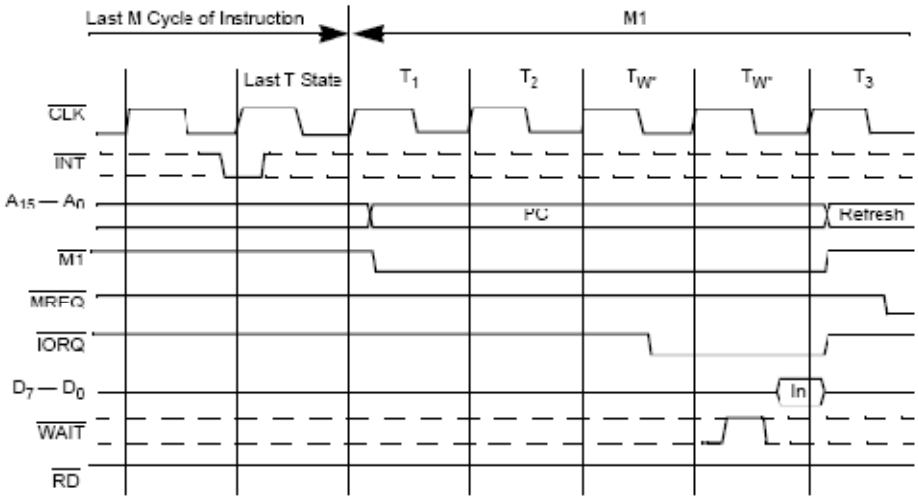


Figure 10: Z80 interrupt request / acknowledge cycle timing (www.zilog.com/docs/z80/um0080.pdf)

Handling Interrupts

Once the interrupt request has been acknowledged, the CPU must handle the interrupt. This is technically done within the CPU and without the help of the interrupt module, but it is included in this section for continuity.

There are three interrupt handling modes that the Z80 CPU can operate in, but the GameBoy does not have an instruction to set this mode programmatically. Instead, the interrupt mode is initialized to the proper value, 2, on a CPU reset. In interrupt mode 2, the CPU can jump to an arbitrary location in memory and begin executing from that address. However the address is not directly specified by the interrupt module. Instead, the 16-bit jump address must be obtained from a table memory. The interrupt module provides the lower 8 bits of the table entry's location and the upper 8 bits are provided by the CPU's 8-bit I register. Like the interrupt mode, the GameBoy's processor does not include instructions for modifying the value

of the I register programmatically so the value must be hard coded on CPU reset. In FPGABoy, we chose to assign this register the value FEh. This allowed us to place the interrupt address table in the only unused area of the GameBoy's memory map, the 96 bytes from 0xFEA0 - 0xFEFF. The table itself is implemented as a small ROM. Each entry corresponds to the 16-bit jump address for the corresponding interrupt. The bytes for each address are stored in little-endian order. The table entry and jump addresses for each of the GameBoy's interrupts are defined by the table below.

Table 5: Interrupt table entry and jump addresses

Interrupt	Table Entry Address	Jump Address
V-Blank	0xFEA0	0x0040
LCDC Stat	0xFEA8	0x0048
Timer Overflow	0xFEB0	0x0050
Serial Transfer	0xFEB8	0x0058
Joypad	0xFEC0	0x0060

Timer Module (Trevor)

FPGABoy implements a timer with four selectable frequencies ranging from 4096 Hz to 262144 Hz. The timer can be controller programmatically by a set of memory mapped registers contained within the module. Games can directly read the timer's counter register, or use the generated timer overflow interrupt to implement time based events. In order to maintain proper timing this module is clocked on the CPU's 4.125 MHz clock, rather than the 33 MHz that most other modules use.

I/O Registers

The timer module has four memory mapped registers which can be accessed through the MMU.

Divider Register (DIV) – FF04 (R/W)

This register is incremented once every 256 clock cycles. Writing any value to this register causes it to reset to zero. Overflow of this register has no effect.

Timer Counter (TIMA) – FF05 (R/W)

This counter is incremented using one of four selectable frequencies, controlled by the TAC register. The timer can also be stopped and started at any time, also via the TAC register. When the value in this counter overflows, a timer overflow interrupt is requested. When this occurs, the value in the TMA register is loaded into the counter, rather than simply resetting to zero.

Timer Modulo (TMA) – FF06 (R/W)

The value to be loaded into the TIMA register when an overflow occurs.

Timer Control (TAC) – FF07 (R/W)

```
Bit 2    - Timer Stop  (0=Stop, 1=Start)
Bits 1-0 - Input Clock Select
           00:   4096 Hz    (~4194 Hz SGB)
           01: 262144 Hz  (~268400 Hz SGB)
           10:  65536 Hz   (~67110 Hz SGB)
           11:  16384 Hz   (~16780 Hz SGB)
```

This register controls the frequency at which the TIMA register increments, as well as enabling and disabling the timer altogether. It has no effect on the frequency or state of the DIV register.

Implementation

The timer module contains four simple dividers, one for each selectable frequency. On each clock cycle, if the selected divider overflows (as indicated by an enable signal), the timer counter (TIMA) is incremented. The divider register (DIV) is always incremented on overflow of the fourth divider (16384 Hz) regardless of the selected frequency.

Video Module (Trevor)

The video module is one of the most complicated aspects of the FPGABoy system. It is primarily responsible for rendering the graphical output of the game. It also performs several other tasks, including the generation of two interrupts, control of 11 memory mapped I/O registers and brokering of access to two separate areas of the memory map, video memory (VRAM) and sprite object attribute memory (OAM). To perform all of these tasks, it uses a 29 state finite state machine, as well as several smaller state machines to control the rendering mode.

I/O Registers

The video module has 11 memory mapped I/O registers that provide information about and control over the rendering state.

LCD Control (LDCD) – FF40 (R/W)

```
Bit 7 - LCD Display Enable             (0=Off, 1=On)
Bit 6 - Window Tile Map Display Select (0=9800-9BFF, 1=9C00-9FFF)
Bit 5 - Window Display Enable          (0=Off, 1=On)
Bit 4 - BG & Window Tile Data Select   (0=8800-97FF, 1=8000-8FFF)
Bit 3 - BG Tile Map Display Select     (0=9800-9BFF, 1=9C00-9FFF)
Bit 2 - OBJ (Sprite) Size              (0=8x8, 1=8x16)
Bit 1 - OBJ (Sprite) Display Enable    (0=Off, 1=On)
Bit 0 - BG Display                     (0=Off, 1=On)
```

The LCDC register controls a number of high level rendering options, including enable/disable flags for each of the three renderable layers, the locations of tile and data maps in VRAM, and the ability to turn the display off altogether.

LCDC Status (STAT) – FF41 (R/W)

```
Bit 6 - LYC=LY Coincidence Interrupt (1=Enable) (Read/Write)
Bit 5 - Mode 2 OAM Interrupt         (1=Enable) (Read/Write)
Bit 4 - Mode 1 V-Blank Interrupt     (1=Enable) (Read/Write)
Bit 3 - Mode 0 H-Blank Interrupt     (1=Enable) (Read/Write)
Bit 2 - Coincidence Flag  (0:LYC<>LY, 1:LYC=LY) (Read Only)
Bit 1-0 - Mode Flag                             (Read Only)
          0: During H-Blank
          1: During V-Blank
          2: During Searching OAM-RAM
          3: During Transfering Data to LCD Driver
```

The STAT register can be used by games to determine the current rendering state, as well as to enable various interrupts. Writes to this register will only affect the upper five bits. Bits 2-0 are set by the video module to provide information about the current rendering mode. For more information about the various video modes see the modes and timing section below.

Background Scroll Y (SCY) – FF42 (R/W)
Background Scroll X (SCX) – FF43 (R/W)

The SCX and SCY registers determine the offset of the upper left corner of the background tile map. The background wraps, so even if these values are non zero the background will still be visible in all areas of the screen.

LCDC Y-Coordinate (LY) – FF44 (Read only)

The LY register contains the current line that is being rendered. If the line count is greater than 153, it resets to zero.

LY Compare (LYC) – FF45 (R/W)

A game may write a value to this register to utilize the LY coincidence interrupt. The video module continuously compares the value in this register to the current line count. If the values match and the LY coincidence interrupt is enabled, an interrupt will be generated.

Background Palette (BGP) – FF47 (R/W)
Object Palette 0 (OBP0) – FF48 (R/W)
Object Palette 1 (OBP1) – FF49 (R/W)

```
Bit 7-6 - Shade for Color Number 3
Bit 5-4 - Shade for Color Number 2
Bit 3-2 - Shade for Color Number 1
Bit 1-0 - Shade for Color Number 0
```

The BGP register defines the four pixel colors that are used for rendering background and window tiles. The OBP0 and OBP1 registers define the color palettes that can be used for

rendering sprites. Each color is a two bit value representing a different shade of grey, with 0 representing white and 3 representing black. To get an actual color from raw pixel data, the video module indexes into the appropriate color palette as if it were an array with four elements.

Window Position Y (WY) – FF4A (R/W)
Window Positon X (WX) – FF4B (R/W)*

The WY and WX registers control the upper left corner of the window tile map. Unlike the background tiles, window tiles do not wrap so this is truly an offset.

*The actual X offset of the window tile map is WX – 7. A value of 7 in this register means that the window tiles will be aligned to the left side of the screen.

Video Modes and Timing

When enabled, the video module continuously cycles through four different modes. Depending on which mode the module is in, some areas of memory may become inaccessible to the CPU. When the mode changes it may also cause several different interrupts to occur.

In order to meet the timing specifications required by the original GameBoy, FPGABoy's video module uses two different clocks. The first is the slower 4.125 MHz clock that runs the CPU. The second is a faster 33 MHz clock. While rendering and memory accesses are performed on the 33 MHz clock, all mode changes occur on the 4.125 MHz clock. The video module maintains two counters, a line count and a "pixel" count, which control the mode and are incremented on each tick of the 4.125 MHz clock. The pixel counter increments 456 times per line and the line count increments 154 times per frame. This results in an overall frame rate of approximately 60 frames per second.

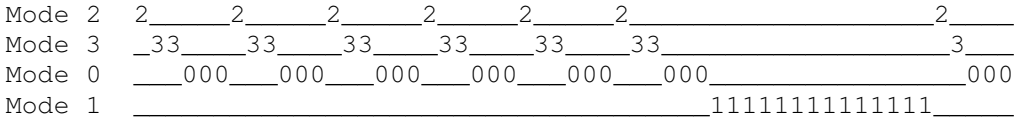


Figure 11: Video mode timing *(Source: http://nocash.emubase.de/pandocs.htm#lcdstatusregister)*

H-Blank – Mode 0

During the H-Blank period the CPU is free to access both the VRAM and OAM areas of memory. This occurs at the end of each rendered line and lasts for 204 cycles. When entering the H-Blank mode, the video module generates an LCDC status interrupt if the corresponding bit of the STAT register is set.

V-Blank – Mode 1

During the V-Blank mode the CPU is free to access both the VRAM and OAM areas of memory. This mode occurs when the line count is greater than 143 and lasts until line count returns to

zero, a total of 4560 cycles. When entering V-Blank mode, two interrupts may be generated. The first is the actual V-Blank interrupt which will always occur. The second is an option LCDC status interrupt which only occurs if the corresponding bit of the STAT register is set.

OAM Lock – Mode 2

During this period the CPU is unable to access the OAM area of memory. Any CPU reads and writes to this area will be ignored. The video module enters this mode for the first 80 cycles of each line. When entering this mode an LCDC status interrupt will be generated if the corresponding bit in the STAT register is set. In reality, FPGABoy does not make use of this mode for rendering so access to memory in the OAM region is only disabled for compatibility.

RAM Lock – Mode 3

During this period the CPU is unable to access both VRAM and OAM. Any CPU reads or writes to either one are ignored. The video module enters this mode immediately after the OAM Lock mode and lasts for 204 cycles. Entering into this mode does not generate an interrupt. All memory accesses required for rendering are performed during this time.

Video Rendering

Just like on the original GameBoy, FPGABoy renders each frame one line at a time. Each line contains 160 rendered pixels and each frame contains 144 rendered lines. During rendering, the current line is stored in a scanline buffer. This buffer allows multiple layers to be rendered separately without generating any actual display data. Once all layers have been rendered, the module reads the data out of the scanline and outputs it one pixel at a time.

While the video mode is controlled on the CPU's 4.125 MHz clock, the actual rendering process occurs on the 33 MHz clock. The principle reason for this was the extremely tight timing that would have been required for rendering with a 4.125 MHz clock. Since each tile and sprite requires been two and six memory access to retrieve the data required for rendering, it is much easier just to overclock the module rather than try to pipeline the process or reduce memory access time.

Video RAM Mappings

The video module uses the following mappings in Video RAM to locate tile data. Some of these regions overlap so special care must be taken to use the correct region as specified by the LCDC register.

Start Address	End Address	Mapping	Condition
0x8000	0x8FFF	BG and Window Tile ID Map	LDCD[4] -> 1
0x8800	0x97FF	BG and Window Tile ID Map*	LCDC[4] -> 0
0x9800	0x9BFF	BG Tile Data	LCDC[3] -> 0
0x9C00	0x9FFF	Background Tile Data	LCDC[3] -> 1

0x9800	0x9BFF	Window Tile Data	LCDC[6] -> 0
0x9C00	0x9FFF	Window Tile Data	LCDC[6] -> 1

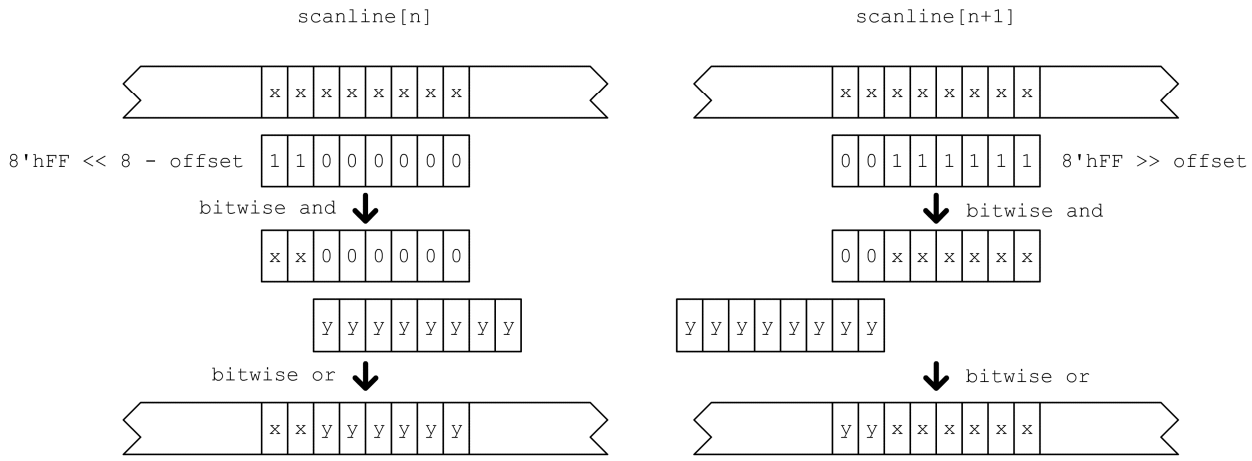
* This region uses signed offsets from -128 to 127 rather than unsigned offsets.

Scanline Buffer

The scanline buffer is implemented as a dual port block RAM, 8 bytes wide and with a depth of 20. Because each pixel is 2 bits, the video module actually has to maintain two scanline buffers, one for the higher order bits of each pixel and one for the lower order bits. These two buffers are combined after rendering is completed to produce the final output.

Using this format for the scanline has a number of important advantages. Each tile that is rendered, be it from a sprite, a window time or a background tile, is 8 pixels across. However, each of these layers can be offset from the left border by a different amount. This means that the video module must either be able to write individual pixels into the scanline, or it must be able to write partial bytes. Of these two options, individual pixels turned out to be extremely slow, both in terms of hardware synthesis and in required clock cycles. Thus, the scanline buffer was implemented as a byte array.

To solve the problem of writing partial bytes, the video module uses a simple masking and shifting technique to when writing non-aligned bytes into the scanline. By using this process we can effectively reduce the number of memory access required to write each byte by a factor of four. A dual port RAM is used so these two memory accesses can be run in parallel. This process is illustrated in the diagram below.



Background and Window Rendering

The background and window layers are both comprised of a grid of 8x8 tiles. The background can have up to 32 tiles in each row and these tiles wrap around if the SCX register is non-zero.

The window tiles do not wrap. Window tiles (if they exist) will always appear on top of the background.

Because the background and window layers have the same basic layout, they can be rendered at the same time. The video module will render 32 columns, one for each 8 pixel tile that can be displayed one each line. For each of these columns it is determined if the window is enabled and visible in that column. If so the window's tile id number (1 byte) is fetched from memory. If it is not and the background is, the background tile id number (1 byte) is fetched. In either case, the resulting tile id number is used to determine the memory address of the actual tile data (2 bytes). These bytes are then fetched and written into the high and low order scanline buffers using the masking / shifting technique. If neither the background nor the window is visible, null bytes are written into the buffer at the corresponding location. When rendering of all 32 background tile columns completed, the video module will begin to render the sprites. A simplified version of the state machine that controls this process is shown below.

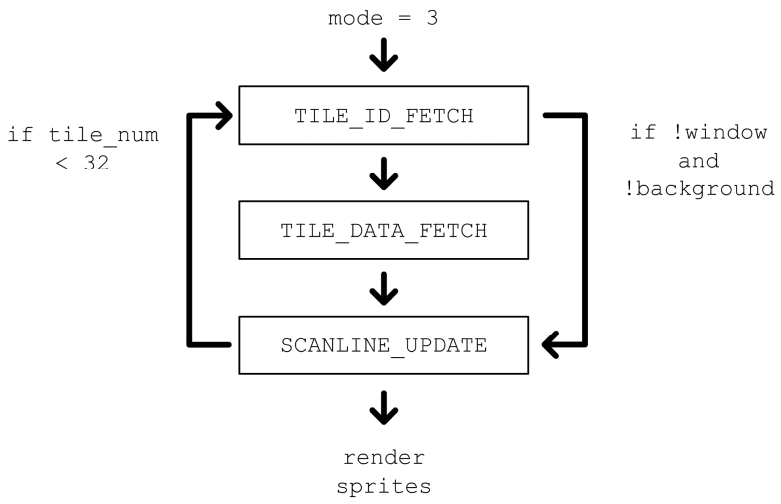


Figure 12: A simplified state diagram for rendering background tiles

Sprite Rendering

Sprites are rendered in a very similar manner to background tiles, but there are a few extra steps that must be taken. First, a lookup into OAM is performed for each sprite to determine its x and y positions (2 bytes). If those coordinates are determined to intersect with the line currently being rendered, two more attributes (2 bytes) are fetched from OAM which correspond to the sprite's location in the tile data map and a number of cosmetic attributes which determine which object palette should be used to color the sprite, whether that sprite appears above or behind the background and if the sprite should be flipped in either the x or y direction. Once these have been obtained, the actual sprite pixel data can be retrieved from VRAM (2 bytes). If the sprite does not intersect with the current line, the sprite is ignored.

Once all the sprite data has been read from memory, the sprite must be drawn. Unfortunately, the sprite cannot be drawn directly into the scanline. Instead, the video module must first determine the color of each pixel in the sprite by indexing into the appropriate object palette. The pixel value for the same location in the scanline must also be read and compared with that of the sprite. Depending on the attributes of the sprite and the color of the background pixel, the sprite's pixel may be masked in favor of the background pixel. This step is repeated for each pixel and the results are stored in a temporary 8-bit register until the high and low bits for all 8 pixels have been computed. At this point the data bytes are written into the scanlines, again using the shifting / masking technique. A simplified state diagram for this process is shown below.

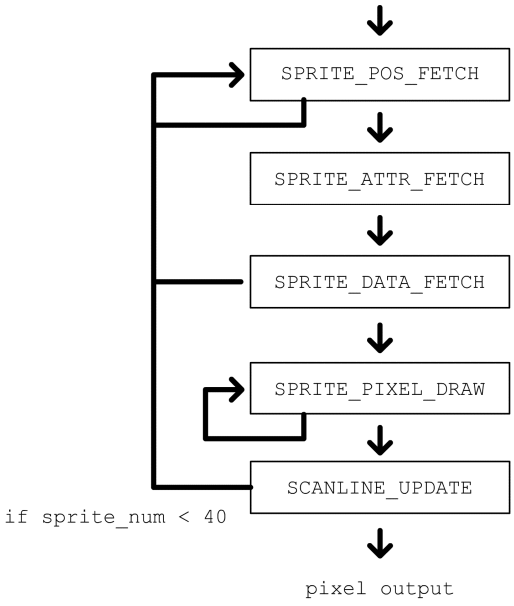


Figure 13: A simplified state diagram for rendering sprites

Pixel Output

Once the entire scanline has been computed, the video module outputs the results, one pixel at a time. This is done by reading out a single byte from the high and low order scanlines, and combining them. The only tricky part about this is that the bits have to be read backwards in order to achieve a correct image. This is because the furthest left pixel in each byte is actually the highest bit.

Video Converter (Oleg)

The video converter module interprets the output generated by the video module and converts it into a form such that it can be passed on to the VGA controller. Specifically, it converts from the FPGABoy's 160x144x2 resolution into VGA's 640x480x24 resolution. To perform the conversion smoothly, the module implements a double buffering system. That is, while the VGA controller is fetching color data from one buffer, the video module is writing into another.

This minimizes the amount of time during which there is no image on the screen and reduces flickering.

Double Buffering

The double buffering is implemented through the use of two different block RAMs, each of which is big enough to store 160x144x2 bits. An internal register keeps state of which of the two block RAMs acts as the front buffer and the inputs and outputs into the rams are multiplexed appropriately. Furthermore, the clocks that drive the block RAMs are also multiplexed as the pixel clock from the VGA controller that outputs to the screen and the clock from the video module are different. The difference in clocks poses no problem since only one type of operation happens for each clock. That is, the video module clock only times the writes while the pixel clock controls the timing of the reads. Since these operations happen on different memories no conflicts arise.

The RAM that is used as the back buffer takes as an input the address determined by the pixel count and line count that is output by the video module. The value that is written into that address also comes from the video module. When the video module completes a whole frame, it raises a vsync signal which signifies to the video converter that it is time to flip buffers. Thus the video converter flips the internal switch and swaps the block memories. Now the video module is able to restart writing into the new back buffer.

While the back buffer is being written by the vide module, the VGA controller reads from the front buffer. The address being read from is determined by the line and pixel count output by the VGA controller. In order to center the image and achieve scaling, the actual address is modified by dividing the pixel and line count by two and shifting it by an offset in the x and y directions. The shift amount is determined by two parameters in the module so the output can be placed anywhere on the screen. The VGA controller keeps reading from whichever front buffer is selected by the internal register to present a seamless image on the screen.

VGA Controller

The VGA controller used by the FPGABoy is identical to the one implemented in Lab 4. As an input it takes a clock and generates the proper output signals such as a pixel clock, delayed hsync and vsync, a blank signal and the line and pixel count for the screen. As stated above, the converter module uses the line and pixel counter to determine which address to read from the front buffer and to display on the screen. This component also facilitates the conversion of the 2-bit FPGABoy color to a 24-bit RGB color value. Since the game boy only supports four colors, the module remaps them onto different shades of gray. Table 6 illustrates the color mappings used between the FPGABoy output and the 24-bit VGA color.

Table 6. FPGABoy to RGB color mapping

FPGABoy Color	Color	RGB Color(24-Bit)
0x00		0xFFFFFF
0x01		0xAAAAAA
0x10		0x555555
0x11		0x000000

The disjoint nature of the two halves of the video converter made it essential in the debugging process. The main reason was that the VGA module could output the contents of the buffer even when the processor or the video module was halted, thus it made the video module much easier to test.

Cartridge Module (Oleg)

The cartridge module is intended to interface the FPGABoy with real game cartridges. The module has two components, one physical and one that was implemented solely in Verilog. The cartridge module is simply a bus that runs from the user I/O pins on the lab kit to the memory map module. Because of the block RAM abstraction used inside the memory module, it is trivial to use the cartridge module to remap the cartridge images stored internally on the block RAM. All that is necessary is to remap the first 32KB of address space along with read and write control signals onto the user I/O pins instead of the block RAM and the interfacing with the external cartridge is complete.

Cartridge Interface

In order to construct the cartridge interface we obtained an original GameBoy handheld device. After disassembling it the back plate with the cartridge slot was extracted and can be seen in Figure 14. Next, we soldered 32 wires to the pins that came out from the cartridge interface on the back of the GameBoy. Next these wires were connected to +5V, GND, and the user2[29:0] port.

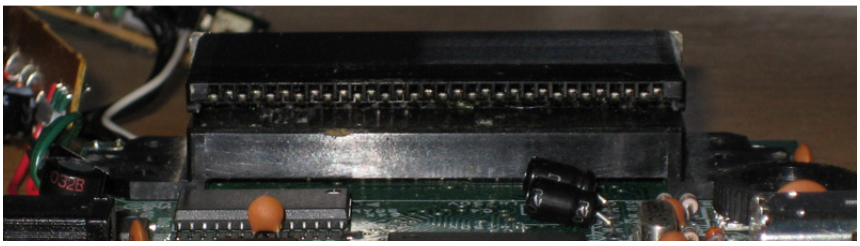


Figure 14. GameBoy back plate cartridge interface

This allowed us to attach real cartridges to the FPGABoy and insert them into our system. Unfortunately, due to lack of time we were unable to get this module to interface properly with the rest of the FPGABoy system. This is definitely something that can be worked upon in the future.

Testing and Debugging

One of the most difficult aspects of this project was to perform most debugging tasks. Because of the monolithic system structure, almost all the modules, except the video converter and input module had to be tested simultaneously. This made it very difficult to track down bugs as they could have originated from any component in the system. Furthermore, we had to aggregate our fixes and changes carefully in order to make sure we don't waste time re-synthesizing the project needlessly.

The most vital component of our testing and debugging system was our access to a high quality emulator [3]. It's ability to disassemble code, display contents of video memory, set breakpoints and view any of the GameBoy state imaginable provided an excellent reference point for finding any issues with our FPGABoy implementation. It is likely that without such a powerful tool this project would've resulted in failure.

Input Converter Module

The testing for the input converter module was done by first performing simulations in ModelSim. This was vital in order to verify that the output of the latch and clock signals were properly timed. Once that was tested, we implemented a simplified version of the lab kit test bench which mapped the eight button states to the LED's on the lab kit. After some toying with the timing of the data signal that was received from the game pad we were successful in making the LED lights light up when the respective button was pressed. This signified proper operation of the input converter module.

Memory Controller

The memory map module was tested over the course of the whole project. This was because the memory map was one of the first modules implemented as it was vital to the operation of the system. Because of that there were one or two incremental bug fixes, but overall it worked near perfectly from the start.

The second set of testing on the memory map module happened towards the end of the project when DMA was added to the system. This was tested by running a GameBoy game and placing a breakpoint when a routine in high RAM was entered. Execution in high RAM signifies the beginning of a DMA sequence. In fact, we determined the exact entry point of the procedure that performs the DMA initialization. Then, using the hex display we were able to output the data that was on the DMA bus and the address from which it was copying. When those values matched up with the reference as executed by a GameBoy emulator, we knew that the DMA operation was working properly. Thus the debugging of the memory controller was complete.

Video Converter Module

Much like the input converter module, the video converter was tested separately. Since our own VGA converter from the previous lab was utilized, there was some assurance that it worked well. Thus, all that was needed to be tested was the buffer flipping on vsync and the writing into the back buffer. A small test harness was established in its own lab kit test bench that would draw vertical stripes on the screen using different colors. Then, scan-line positioning was tested by displaying alternating color horizontal stripes. This type of testing revealed all the display artifacts. Using this information, most of the timing bugs that caused them were tracked down. The result was a very smooth converter module with hardly any flicker.

Video Module

The video module was initially tested using ModelSim. Contrived scenarios using preset register values and an emulator dump of video RAM were used to render a single line. The state and eventual pixel output was compared against a set of expect values to determine validity. Once the system had progressed to the point of mobile sprites and was actually playable, testing of the video module was don't mostly by play testing on the FPGA.

Game-play

One of the more fun aspects of testing was towards the end of the project development cycle. That testing involved playing the game and looking for various errors. Luckily, by this time our system was stable enough that there no visible errors that were caused by our modules. The other part of game play testing was to check our compatibility with other GameBoy games. This involved obtaining a variety of other images for testing and implementing a way to switch between games without having to re-synthesize.

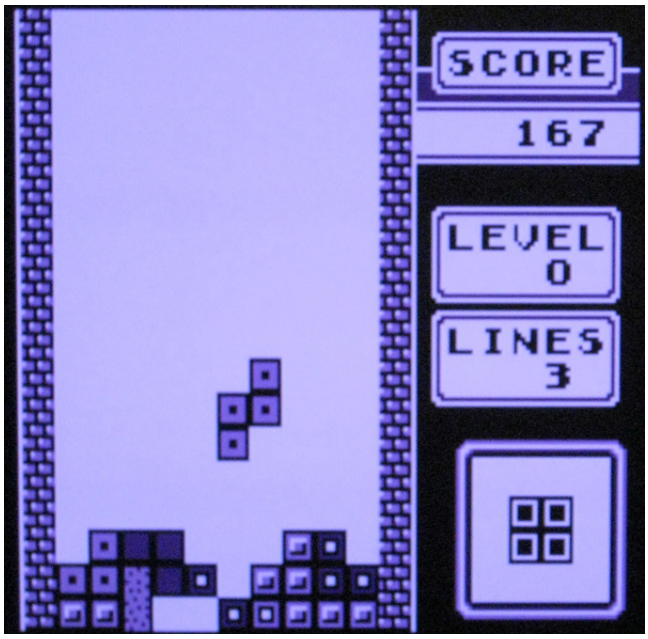


Figure 15. Game-play testing

As a result of game play testing we discovered that most of the other games had many compatibility issues. These issues were primarily caused by our lack of support for many of the advanced GameBoy features and thus were labeled outside of the scope of this project. An interesting point is that all of the games that were tested were from 1989 but only the first game, Tetris, was fully functional. This revealed the rate at which the advanced features of the GameBoy were exploited such a short time after its release. By implementing the rest of the GameBoy specification, the compatibility can definitely be improved in the future.

Conclusion and Future Work

Overall, the FPGABoy was a great success. We were able to implement a working system that is able to run and play GameBoy cartridges. There are significant compatibility issues with most games, but others such as Tetris appear to work near flawlessly. In the future there is a significant amount of work that could be done to improve the FPGABoy's compatibility, both in terms of rendering and fixing any remaining bugs in the TV80 core. Another step to facilitate better compatibility would adding support for external game cartridges. Our attempt to get this working proved it to be somewhat difficult. There are also two major GameBoy features that were not implemented in FPGABoy. Adding support for sound and serial connections would make this a truly complete system.

References

[1] kOOPa. nocash. "Everything You Always Wanted To Know About GAMEBOY," Pan Docs, 2001, <http://nocash.emubase.de/pandocs.htm#joypadinput>

[2] Neviksti. "Gameboy Bootstrap ROM," Gameboy Developement Wiki, 2008, <http://gameboygenius.8bitcollective.com/gbdev-wiki/articles/Gameboy_Bootstrap_ROM>

[3] "BGB Homepage," <http://bgb.bircd.org/>

[4] "TV80 Home Page." 15 May 2009 <http://ghutchis.googlepages.com/tv80homepage>.