Implementing the Nintendo Entertainment System on a FPGA

Jonathan Sieber

June 25, 2013

Contents

Chapter 1

Introduction

In this work I try to implement the Nintendo Entertainment System (NES) on a FPGA platform.

The NES is one of the most famous video game consoles of the 8-bit era. Using custom designed hardware that was primarily optimized for low cost, and was not very powerful at that time, it still was the basis for a big library of high quality games, that are still fun to play today.

Besides being a practical exercise in hardware design, this project aims to be a continuation of the efforts of the emulator scene, to conserve video game history by bringing it to new hardware platforms.

1.1 NES Technical Data

- 8-Bit 6502-like CPU clocked at 1.6 MHz
- 32k of Program ROM space (extensible through bank switching)
- 2 KiB Program SRAM
- Dedicated Picture Processing Unit with NTSC/PAL Output
- Support for scrollable background and up to 64 Sprites
- 4 waveform audio channels and sample playback integrated on the CPU

Nintendo used the MOS 6502 processor, that itself was once an innovative design that was used in home computers like the Commodore 64 or Apple][, or the previous generation console Atari 2600.

In contrast, the Picture Processing Unit (PPU) was considered rather advanced. It had access to an own memory bus and could draw sprites and a scrolling background independently from the CPU.

Nintendo decided to keep the hardware as cheap as possible, which was part of the marketing strategy. Games shipped as ROM in a cartridge, which allowed to include more memory when manifacturing cost dropped over the years. This practice extended the lifetime of the console, and was also used in later consoles by Nintendo and competitors.

1.2 FPGA Reimplementation

The NES emulator developer scene has already thoroughly reverse engineered the hardware and provides lots of documentation. These are mostly behavorial descriptions, either written from the perspective of developing new software (Homebrew), or creating a PC based emulator. The challenge is to put all these little pieces of information together, infer what the hardware must have looked like internally, and creating a description of it in a Hardware Design Language.

As the digital logic of the NES fits easily into the area and timing constraints of a modern FPGA, high-accuracy emulation would be possible without performance problems. A low-cost FPGA board could even support several video game architectures, like intended by the FPGAArcade[1] project.

1.3 Goal

The Digilent Genesys[2] was selected as development platform for this project. It contains a Xilinx Virtex 5 LX50T FPGA, which is an order of magnitude bigger than what would be necessary for the NES.

[1] http://www.fpgaarcade.com/
[2] http://www.digilentinc.com/Products/Detail.cfm?Prod=GENESYS

Since writing an emulator that is compatible with 100% of all games is out of the scope of this project, it is limited to getting a single game to run. Super Mario Brothers was chosen because of its prominance that is above any other NES game. It is an interesting choice from a technical perspective, because it relies on several NES Quirks that are tricky to implement in a PC based emulator.

This project is also thought as a basis for teaching hardware design in a practical course. Extending the project to support more games, or adding image enhancement functionality may provide an interesting future work.

Chapter 2

Nintendo Entertainment System

Figure 2.1:

2.1 History

The NES was first released in Japan in 1983 under the brand name “Famicom”. In the same year, the US-american video game market was undergoing a massive recession. Competition from home computers, and a flood of video game

platforms with a vast library of inferior quality games led to an oversaturation of the market, and in consequence, the collapse of an entire industry branch.

Despite of the success in Japan, Nintendo was met with skepticism by american distributors, so they decided to release and market the console there themselves and released the "Nintendo Entertainment System" with a redesigned casing in 1985.

To keep control over game releases, and inhibit a flood of low-quality third party games that killed the Atari 2600, Nintendo included a lockout-system in the NES. Third-party game developers had to go through an approval process by Nintendo to get this license chip.

Of course the NES can easily be modified to disable the lockout system, and later it was circumvented without console modification by different companies[1], but worked overall, and helped establish a business model that is still common in the video game console industry today.

For a more detailed history see:

http://web.archive.org/web/20100101161115/http://nintendoland.com/history/hist3.htm

http://nesdev.com/NESDoc.pdf

2.2 Hardware Overview

The heart of the NES is the 6502 based CPU. It is clocked at 1.79 Mhz (NTSC version), supported by the Picture Processing Unit (PPU), that can generate a video signal independently from the CPU.

The games are shipped in a cartridge, or sometimes 'Game Pak'. These contain a PCB with ROM ICs that are connected directly to the system buses. This eliminates the need for loading times.

The CPU masters a 16 bit address bus with an 8 bit data bus, that connects the internal 2KiB SRAM, and memory mapped registers for interacting with the internal Audio Processing Unit (APU), the controller pad, and the PPU.

The upper 32KiB of the addressing range are reserved for the program ROM (PRGROM) on the cartridge. The cartridge can ship additional hardware like a battery-backed RAM for savegames that maps to a free lower address range.

The PPU has its own memory bus that it uses for drawing. The cartridge contains another ROM chip that holds graphic patterns of the background and sprites (movable objects). The NES includes another 2 KiB of SRAM for the PPU that contains information which of these patterns should be displayed

[1] http://en.wikipedia.org/wiki/10NES#Circumvention

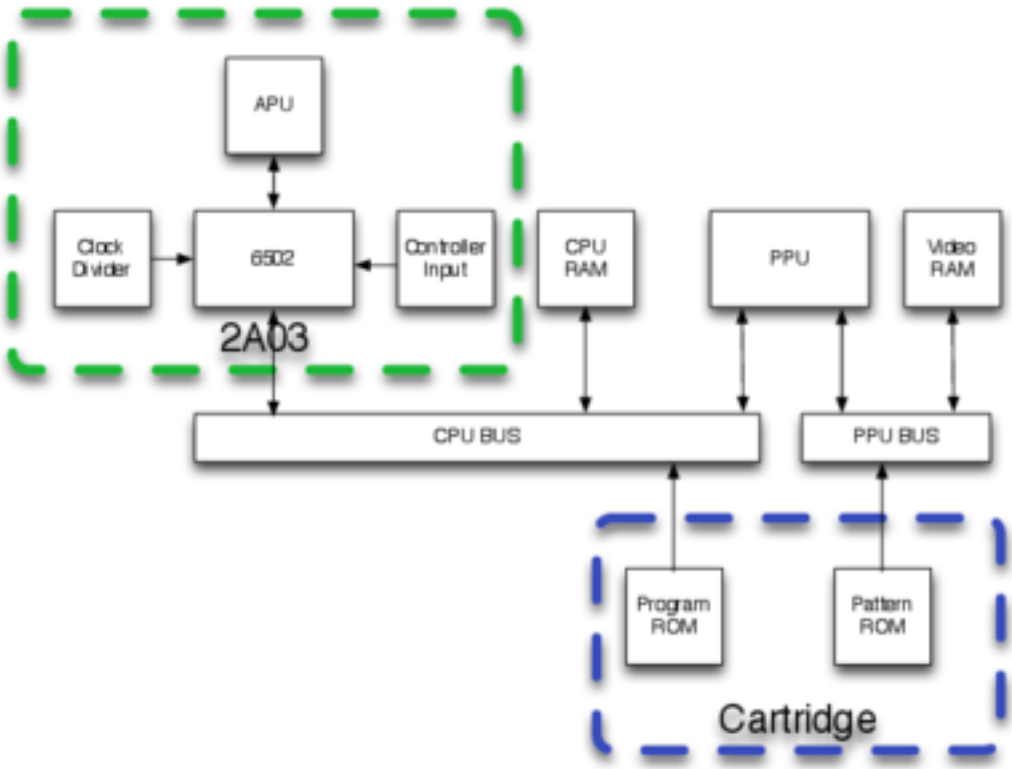

APU
Clock Divider
6502
Controller Input
2A03
CPU RAM
PPU
Video RAM
CPU BUS
PPU BUS
Program ROM
Pattern ROM
Cartridge

Figure 2.2:

for the background. The sprite positioning information is stored in a 256-byte DRAM integrated into the PPU IC.

The PPU is also connected to the CPUs Non-Maskable Interrupt (NMI), through which it signals that it finished drawing the screen. At this point, the cathode ray in the monitor/TV is positioned at the bottom right of the visible image, and needs some time to go back to the top left to draw the next screen. This time window is called Vertical Blank period (VBlank). During VBlank, the PPU stops accessing memory and the CPU can update the VRAM data.

2.3 Central Processing Unit

The NES CPU is based off the MOS Technology 6502. The 6502 was already a bit outdated when the NES was released, but once was considered a revolutionary architecture. There is plenty of information available documenting the 6502 history and technical details. For a quick introduction, I can recommend these slides: http://www.gernoth.net/rdf/ref2011/6502fabian_wenzel.pdf

The CPU is integrated on an ASIC (labelled RP2A03), that also integrates the Audio Processor, a DMA Unit, a clock divider and a few additional pins related to controller input.

Note that the hexadecimal notation in this documentation follows the convention for 6502 assembly, which looks like $1337 (usual is 0x1337).

2 Phase Clock The 6502 internally divides the clock in 2 phases, which allows it to access memory in a single clock cycle and introduce some simple form of instruction pipelining. These phases are represented by the signals PHI1 and PHI2 (actually ϕ, but I can not use this in source code).

PHI2 is essentially an inverted PHI1. Strictly speaking they both have a slightly extended low period, which ensures that they are never high at the same time.

At the rising edge of PHI1, the CPU begins to calculate the address for the next memory access and asserts the address bus before the falling edge.

During PHI2, the semantics depend on whether the CPU signals a read or write through the RW pin.

In write mode, the CPU drives the data bus from some time during PHI2 to the falling edge.

In read mode, external devices may drive the data bus. This is why PHI2 is the external clock on the NES and used for chip select circuitry in combination with an address decoder. When writing, the data is guaranteed to be available at the end of PHI2.

Sprite DMA Unit For updating to the internal PPU memory, there are memory mapped registers at $2003 and $2004. $2003 is used to set the internal address, $2004 writes a value with auto-increment.

Since most games would want to update all 256 bytes of sprite data in each frame, the 2A03 integrates a unit that can halt the CPU and copy one memory page to $2004 directly. This saves significant overhead compared to copying everything in code on the CPU. The DMA Unit needs 2 cycles per byte, while doing this in 6502 assembly would require additional memory accesses for fetching the opcodes of the load, store and loop instructions.

2.4 Picture Processing Unit

The Picture Processing Unit (PPU) is a dedicated video generation chip that directly generates a TV Signal (PAL or NTSC, depending on regional hardware variant). It is sometimes referred to by the ICs package name, (RP)2C02 (or 2C07 in case of PAL).

It features one background layer with support for scrolling, and up to 64 sprites , of which 8 can be displayed in a single scanline. The background is stored in 2 KiB of internal SRAM, the sprite data is in an PPU internal 256 byte DRAM.

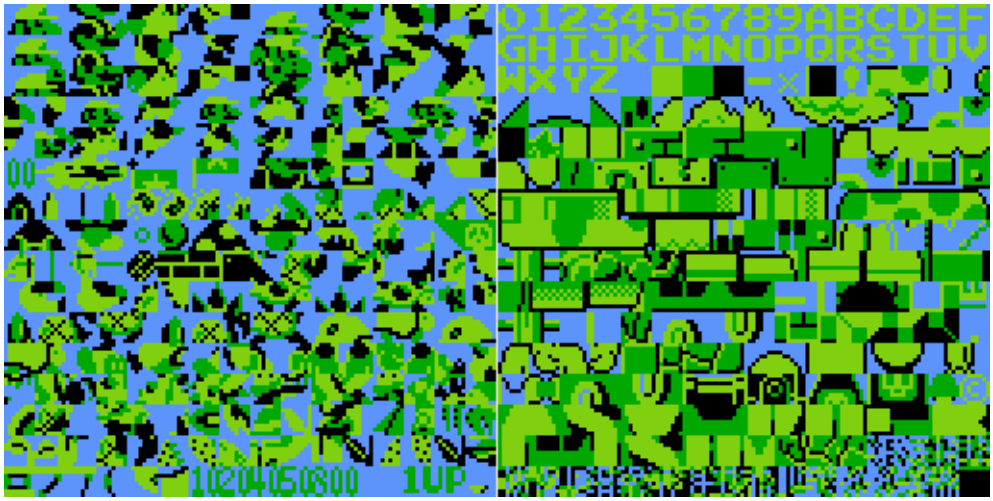


Figure 2.3: The entire graphics data of Super Mario Bros.

Pattern Table Graphics data for background and sprites, usually called patterns, is stored in a dedicated ROM on the cartridge. These are divided into two regions, from $0000 to $0fff and $1000 to $1fff. The PPU can be configured

which one is used for background tiles and for sprites. Figure 2.3 shows the two pattern tables of Super Mario Brothers.

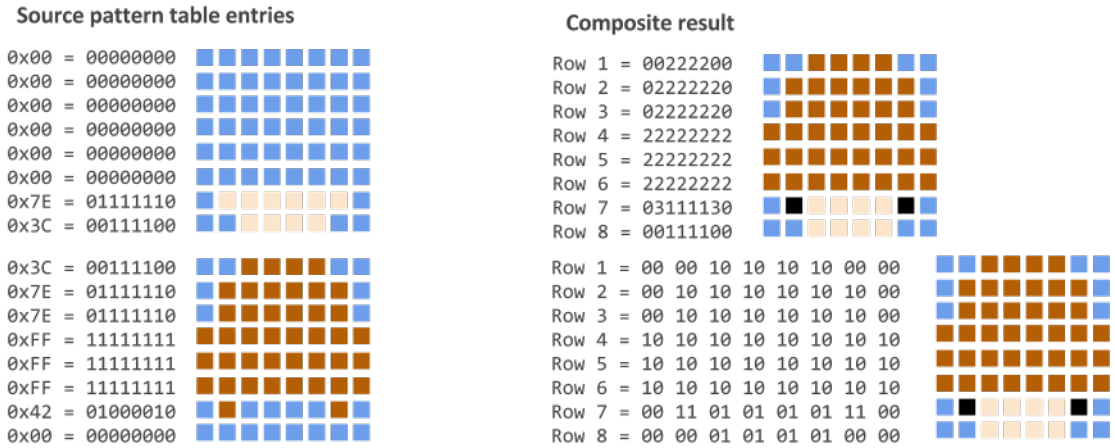


Figure 2.4: Taken from http://opcode-defined.quora.com/How-NES-Graphics-Work-Pattern-tables

The graphics data is arranged in tiles of 8x8 pixels. Those patterns store 2 bits of color information and are arranged as so-called bitplanes. This means that 1 byte directly represents one color bit of an 8 pixel line. So 8 bytes represent Bit A of 8x8 pixel tile, followed by 8 bytes with bit channel B of the same image. Figure 2.4 illustrates this. This may seem a bit inconvenient for generating and manipulating those images, but has advantages for efficient hardware design.

Background Layer The background layer consists of 32x30 tiles, which results in the screen resolution of 256x240 pixels. Which patterns make up the background is stored in an array of 8-bit indices, which is called the name table.

Since the pattern table only stores 2 bits of color (of which one value is reserved for transparency), there is a need for additional color information. The 32x30 tile index information fills only 960 byte of a kilobyte, so the remaining 64 bytes are used to encode additional 2 color bits per 16x16 tile. This region is called the Attribute Table.

There is addressing room for 4 name tables that are arranged in a rectangular fashion. This can be thought of as one big 64x60 tile name table. The PPU scroll registers allow to select a window of 32x30 tiles.

The NES contains only 2KiB of RAM. The 2 addressing lines A10 and A11 that contain the nametable index are routed through the cartridge, so games can decide on either vertical or horizontal mirroring and a matching scrolling mode,

or ship additional RAM on the cartridge to allow diagonal scrolling through all 4 screens.

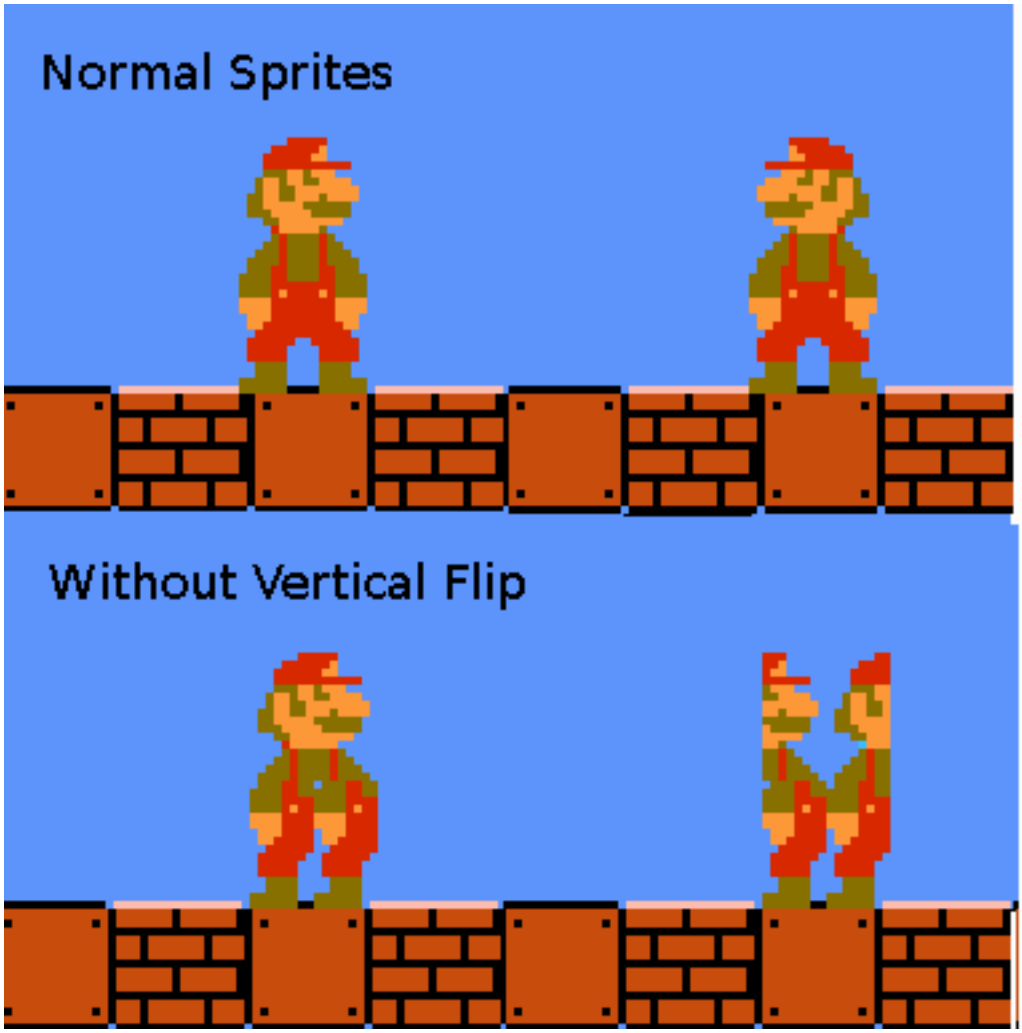


Figure 2.5: Poor Mario with and without the sprite X flip feature

Sprite Layer The PPU also support up to 64 Sprites, which are used for objects that move independently from each other and the background, like the player figure, enemies, projectiles and such. The sprites are 8x8 pixels in size, just like the background tiles.

This sprite data is stored in a 256 byte RAM inside the PPU, that is often referred to as Object Attribute Memory (OAM). Each sprite uses 4 bytes: Horizontal

position, vertical position, pattern index and control flags.

The control flags allow to flip the pattern in horizontal and vertical direction. This is used for example to allow objects to go left and right, or simply save pattern ROM space for symmetric objects.

Another flag allows to control whether sprites appear in front of the background, or are only visible on transparent background pixels.

Figure 2.6: The NES Color Palette (NTSC Version)

Color Palette The PPU supports up to 56 colors (In theory, 64 are addressable, but 8 of them are different shades of blacker-than-black). The color and attribute bits of the tiles and sprites are translated to the actual output colors by a lookup in the palette RAM.

This palette RAM is a memory that holds four 3-color palettes for the background, and another four for the sprites likewise. The attributes selects which palettes will be used, while the color bits coming from the pattern ROM select one of the 3 colors, or zero for transparency.

Sprite 0 Collision Detection The sprite at the first memory location has an implicit special property. If a non-transparent pixel of both sprite #0 and a background tile appears at the same place, a flag that can be read by the CPU is set.

This is not very useful as collision detection per se, since the CPU has no way to find out were exactly the collision happened, or detect more than one collision per frame. It can be used for timing purposes, to detect whether the PPU rendering has already reached a certain position.

Video Timing Unlike modern computer graphics hardware, and like most 2D video game consoles, the NES does not use a framebuffer. Instead the PPU runs synchronous to the video signal and generates each pixel on-the-fly.

This means that games have to run at a fixed framerate, and the video chip capabilities dictates how many objects can be drawn to the screen. For example, only 8 sprites can be drawn on a single line, because the PPU has only 8 registers for selecting sprites, that are filled in the horizontal blank period.

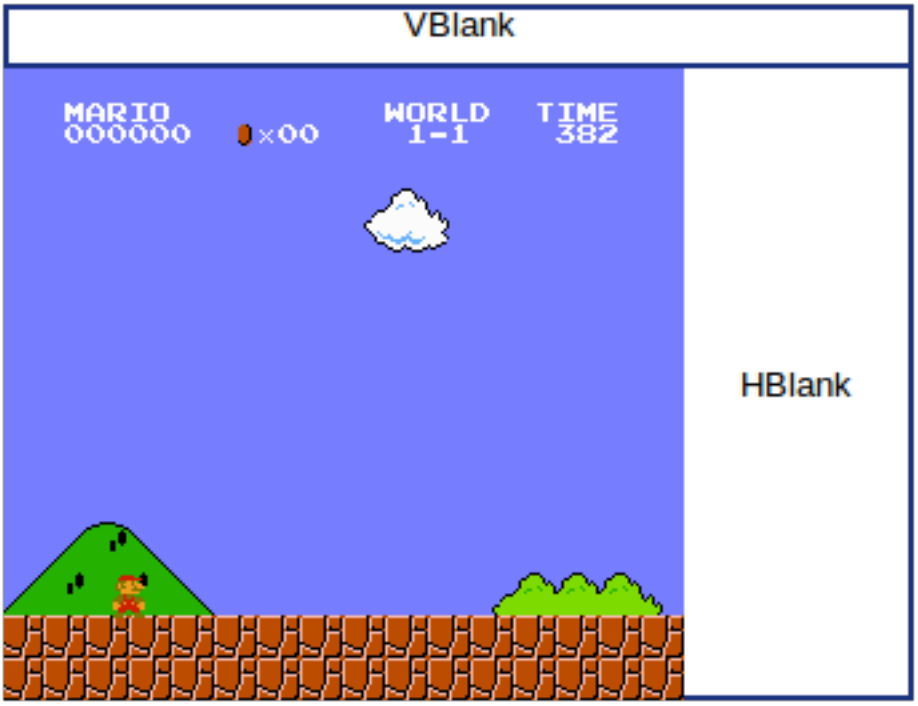


Figure 2.7: Per frame, there is one large VBlank period, and a shorter HBlank period after each scanline

During rendering, the PPU continously accesses the PPU memory, which blocks it for CPU access. At the beginning of the VBlank period, the PPU generates an interrupt. The interrupt handler in the CPU then usually calls the graphics update routine to write to the nametables and copy the sprite data via the DMA unit. The programmer can temporarily disable the PPU to gain more time for memory accesses. This is usually done in the initialization phase, as the VBlank time is not sufficient to completely fill the nametables.

Memory Bus The PPU reuses the data pins for the lower 8 bits of the address. These are buffered by a latch on the PCB. This probably allowed Nintendo/Ricoh to choose a smaller (and cheaper) IC package, allthough it effectively halves the memory bandwith by requiring two PPU clocks instead of one for each memory access.

2.5 Audio Processing Unit

The 2A03 CPU chip of the NES integrates a unit for audio playback. It is referred to as Audio Processing Unit (APU), or sometimes pseudo-APU (pAPU).

It features 2 square wave channels, one triangle, a noise generator based on a LFSR (linear feedback shift register) and a delta modulation channel for sample playback.

These waveforms can be controlled in frequency, volume and duration, which is sufficient for generating music.

The rectangle waves are usually used for melody and sound effects. They are programmable to 4 different pulse widths, which result in a slightly different timbre. The rectangle channels also integrate an effect unit that allow to program a frequency sweep. This is used for sound effects, like Marios jumps (think “booiiiiing”).

The triangle channel is usually used for bass accompaniment, as the waveform waveform contains less harmonics than a square. It does not feature a volume control register like the other channels.

Super Mario Brothers, and many other games refrained from using the DMC Unit, as audio samples have a relatively high memory demand, leading to higher cartridge manifacturing cost if long samples are used.

The 2A03 groups the audio output into 2 pins, One contains the rectangle channels, the other the noise, triangle and DMC Channel. Those are filtered and mixed in a non-linear way by the analog circuitry on the mainboard.

Like the PPU, the APU offers interrupts. One is fired periodically at 60 Hz (But still potentially asynchronous to the VBlank Interrupt), and another fires when the DMC unit is finished with a sample.

2.6 Cartridge

Figure 2.8: The bare PCB of the SMB Cartridge

NES Games are delievered as a cartridge, containing a PCB (Printed Circuit Board) with ROM chips that contain program code and graphic data. Those are connected to the system with a 60-pin-connector, which allows to directly access this memory as part of an address bus and elemininating the need for data copying and loading times.

With linear addressing on the 16 bit wide bus, the Program ROM size is limited to 32 KiB. Later games demanded more, so Nintendo introduced a Memory Mapper Chip (MMC), that allowed to access larger ROMs through bank switching.

This technology also enabled game developers to ship additional hardware features on the cartridge, like more CPU RAM, as 2 KiB was simply not enough for some games.

Memory Mapper At release time, a NES Cartridge offered 32 KiB of visible program ROM in the CPU Address space, and another 8k of Graphical ROM for the PPU Bus. This was sufficient for simpler games, but 2 years later the games scraped the limit. Super Mario Bros. was already a really tight fit, as it uses all of the 32768 bytes of program ROM, and used a few unused sprite pattern slots in PPU ROM for storing the startup title screen. This data can be seen in the lower right of figure 2.3.

Later when ROM space became cheaper, bigger games were built, with a total ROM capacity reaching into the megabyte range. This was realized through a technique that is similar to what was known as “Bank Switching” in the PC world at that time. For this, later game cartridges shipped with a circuit called “Memory Mapper”, which is responsible for making the whole ROM available

from the limited address space window. Other vendors manifactured their own mappers, either as an ASIC, or built from discrete logic. These mappers differ in terms of bank size, additionally supported RAM, the PPU Memory model and the configuration register semantics. Since this project's goal is to run games from a ROM dump without the original cartridge, all mapper chip varieties eventually need to be implemented to gain full compatibility with all NES games.

Lockout System The third chip at the right in figure 2.8 is the lockout chip (sometimes called 10NES) that was used in the non-japanese NES versions to implement regional lockout and force game developers to go through the Nintendo approval process.

The idea is that the same chip is built into the console and checks the presence of the cartridge chip via pseudo-cryptographic authentication codes. If the the cartridge chip was not present, the console chip would hold down reset so the console would not start up without hardware modification.

Of course third party vendors found ways to circumvent this system and Nintendo skipped it in later NES revisions. This project follows this decision.

2.7 Controller Input

The controller electronic is extremly simple, as it contains nothing more than a "4021" 8-bit shift register with parallel load functionality, where each bit corresponds to a button on the controller.

The serial output is connected to the memory bus. The CPU integrates an address decoder for $4016 and $4017, and a 3 bit write only register at $4016 for controlling the shift register load enable.

There are various edge cases involving various alternative NES input devices, or the expansion port, that use more than one serial bit and differ between regional variants, these are ignored here for simplicity.

Chapter 3

Emulation

A video game console emulator is a program that allows to execute a video game on a entirely different system.

This is achieved by building an interpreter (or JIT-compiler in some cases) for the CPU of the emulated system, and providing a model of the peripheral hardware. The challenge here is that console games are written for one specific system with a fixed configuration, and are not robust towards slight inaccuracies in the emulated environment.

Since Nintendo never released any technical documentation, the NES hackers had to figure out what this environment looked like by reverse engineering.

The game cartridge is represented by a "ROM File", that contains memory dumps of the game. Since additional hardware may be shipped on the cartridge, the emulator needs to implement different cartridge configurations, which is specified in the ROM file header.

Replacing the physical Cartridge with a file that can be easily copied offers revolutionary distribution options, of which most do not involve giving any money to Nintendo, which is why they like to call it "copyright violation".

Emulators are tremendously useful for creating new software or modifying existing games. Fans translated games that were never released outside of Japan, fixed bugs and hacked new levels and game mechanics into old games.

Many emulators also improve the gameplay experience of the original system, with specialized image scaling algorithms, save states lessening the insane difficulty level of many games, or multiplayer options over network.

3.1 Early Emulation approaches

The first NES emulator were developed without a complete understanding of the NES internals, and some games did just not run correctly. The developers took the easy route and modified the game code itself to somehow run in the changed emulator environment.

These workarounds were specific to the emulator, and even may have made the game incompatible with the original hardware, or other more accurate emulators. This led to an undesirable lock-in effect, until the next generation of emulators achieved compatibility with the unmodified ROMs

3.2 Cycle-accurate Emulation

> Back in the late ́90s, Nesticle was easily the NES emulator of choice, with system requirements of roughly 25MHz. This performance came at a significant cost: game images were hacked to run on this emulator specifically. Fan-made translations and hacks relied on emulation quirks that rendered games unplayable on both real hardware and on other emulators, creating a sort of lock-in effect that took a long while to break. At the time, people didn ́t care about how the games originally looked and played in general, they just cared about how they looked and played in this arbitrary and artificial environment.
>
> These days, the most dominant emulators are Nestopia and Nintendulator, requiring 800MHz and 1.6GHz, respectively, to attain full speed. The need for speed isn ́t because the emulators aren ́t well optimized: it ́s because they are a far more faithful recreation of the original NES hardware in software.

[1]

Fully accurate emulation requires that all timing-critical events happen in the same way as in the original hardware. For a conservatively written emulator, this means that only one CPU clock can be simulated before updating the other components. Good software design is necessary to achieve acceptable performance with this approach.

The drawback of this approach is bad performance, as continously switching between different code sections contradicts caching strategies of modern CPUs.

(Kelley) attempts to solve this problem in a novel way, by recompiling NES code to native instructions with LLVM, but concludes it is not practically useful for emulators without dynamic JIT-compilation that can not differentiate between code and code treated as data.

[1] http://arstechnica.com/gaming/2011/08/accuracy-takes-power-one-mans-3ghz-quest-to-build-a-perfect-snes-emula
1/

3.3 Applications

Meanwhile, emulator technology is used by Nintendo in the "Virtual Console", a platform for re-releasing old games titles on their current consoles like Wii and 3DS. It is based on software-emulation of the original titles, and implements a range of consoles, from their own products NES and SNES, to those of former competitors, like Sega Genesis, SNK Neo Geo and others.

More in the scientific realm, VII (2013) uses a NES Emulator to build a machine learning gameplay algorithm that is based on maximizing the values in RAM that represent progress, like score, horizontal position etc.

I recommend watching his video:

http://www.youtube.com/watch?v=xOCurBYI_gY

3.4 Sources of information

An extensive list of the available documentation can be found at http://www.nesdev.com/

Another important source is the NESDev wiki, that basically contains the same information, but is still occasionally refined to clear out ambiguous issues.

Topic	Title	URL
CPU	2A03 Technical Reference	http://nesdev.com/2A03%20technical%20reference.txt
PPU	2C02 Technical Reference	http://nesdev.com/2C02%20technical%20reference.TXT

Overview NESDoc http://nesdev.com/NESDoc.pdf

For specific unresolved questions, there is the option to just ask in the IRC channel #nesdev@freenode. I perceived the community as friendly and helpful.

There is still reverse engineering going on in form of the Visual 2A03/2C02 project. This work is based on the Visual 6502 project,[2] that is dedicated to analyzing classic ICs by opening the chip package analyzing high resolution chip photographies.

The NESDev wiki provides an interesting tutorial on how to interpret these chip images yourself:

http://wiki.nesdev.com/w/index.php/Visual_circuit_tutorial

[2] Http://visual6502.org/ http://www.qmtpro.com/nes/chipimages/visual2a03/ http://www.qmtpro.com/nes/chipimages/visual2c02/.

Chapter 4

FPGA Implementation

This chapter shall give an overview of the current state of implementation, its flaws and incomplete parts, and describes the the difficulties that arose while developing it.

Highlighted Words refer to HDL modules that can be found in the source code. *This Highlight* refers to actual filenames.

4.1 Top Module

The module **NES_Mainboard** represents the NES Core, that is independent from the periphery of a specific FPGA board. It includes CPU (**NES_2A03**), PPU (**NES_2C02**) and the **CartridgeROM**, with a bit of address decoding logic.

It also contains the multiplexer for the data bus in read direction, replacing the original tristate logic.

Genesys_NES is the top module for synthesis and contains the HDMI and AC97 drivers for connecting **NES_Mainboard** to the Digilent Genesys board. It also routes through the pins for the controller pad.

It uses a Block RAM based framebuffer to make the NES clock speed independent from the video refresh rate, in contrast to the original NES that runs synchronous the NTSC pixel clock.

4.2 CPU

The starting point of this work was the NES On-A-Chip project by Leach. It reuses an open source 6502 core from opencores.org, and basically adds a clock

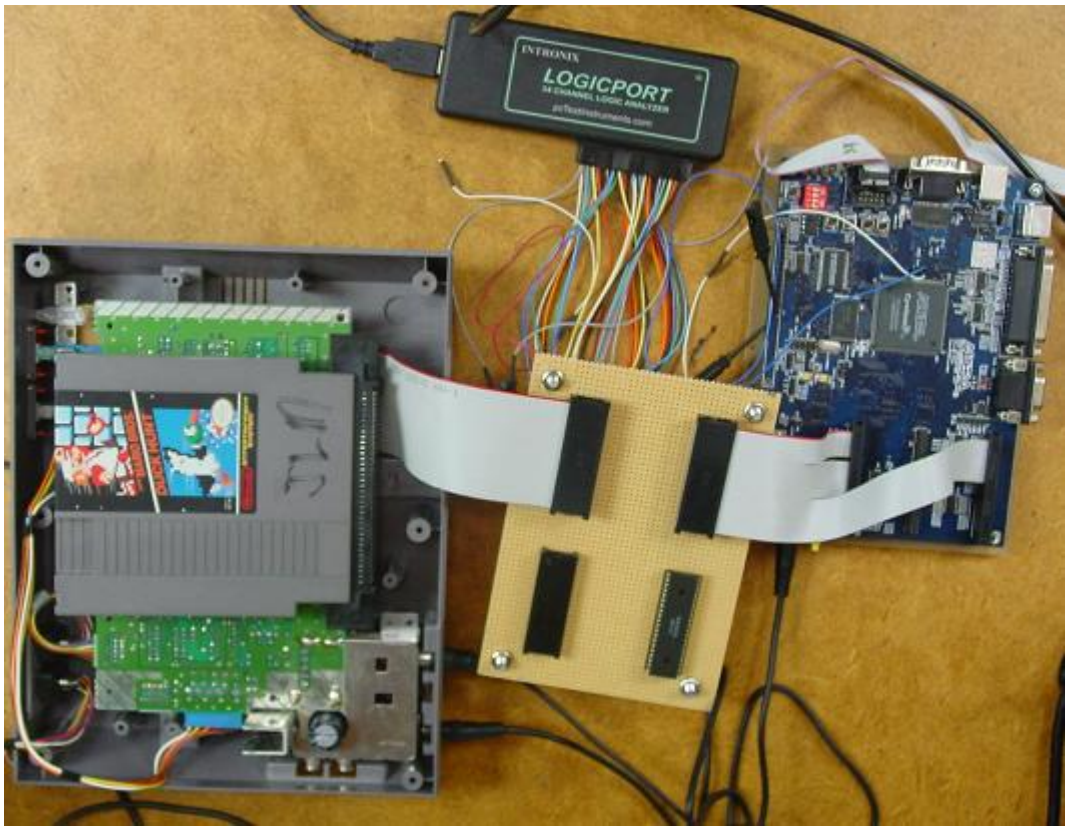


Figure 4.1: The setup by Dan Leach, replacing the original CPU with an FPGA

divider and the DMA unit. What makes it valuable as a starting point is that it was already successfully tested as replacement for the 2A03 in an actual NES. I adopted the original name of the module, **NES_2A03**, found in *Dan_2A03.vhd*.

Since the original code was developed for an Altera FPGA, some minor syntax issues appeared when synthesizing with the Xilinx design tools. In the behavorial simulation, the CPU worked, but not after synthesis. Post-synthesis simulation revealed that there was timing problem related to one of the divided clock signals.

Dan Leachs CPU uses a simple clock divider to generate a divide-by-12 signal with different phases. The 6502 uses 2 clock phases, and this implementation introduced more for signalling and debugging bus access timings. One of these signals was used as a clock, not a clock enable. After changing the affected code to a clock enable like behaviour, the design worked.

4.3 Cartridge

The module **CartridgeROM** implements the program and graphic pattern ROM. It is currently restricted to simple linear ROMs, and does not support any memory mappers or other possible extra cartridge features. Nametable mirroring is currently hardwired to vertical mirroring in the PPU.

The Digilent Genesys Board only provides DDR2 DRAM, which requires a memory controller that handles addressing and memory refreshing. Xilinx offers a memory controller generator as black box IP through CoreGen. I perceived the generator as a very unpleasant piece of software, since it requires to enter all DDR memory pins in a GUI dialog, which had a bug that prevented me from correctly entering one specific pin. I tried to workaround around this by editing this specific pin later in the generated files, but was unable to synthesize the generated example design. Since i did have the competence to debug this strange synthesizer error, i gave up and decided to use the internal Block RAM.

So i decided to use the RAM integrated on the FPGA chip, the Block RAM. The disadvantage of Block RAM is that it requires a rather big FPGA that is vastly oversized for the rest of the NES. The Virtex 5 LX50T offers 1.7 Mbit of BRAM (taken from the Genesys board manual), while the largest officially released NES games ship up to 6 Mbit of ROM.

A simpler solution would be to use an platform that offers SRAM, which could in return be based on a much smaller and cheaper FPGA.

Describing BlockRAM-based ROMs XST supports inferring BlockRAM from behavorial descriptions, so in theory it should be possible to describe it in an vendor-neutral way.

The simplest approach is to store all the data in the VHDL file by writing a giant switch-case statement that return the ROM value for every address. It is

a good idea to automatically generate this file with a script. One example is *cartridge.py*, a MyHDL script that also describes the **CartridgeROM** module, and reads the ROM content from a NES ROM file.

While XST has no problem with these large files, they slowed down the ModelSim simulation software at the design elaboration phase. For Modelsim i used *CartridgeROM.vhd*, which relies on VHDL File Input/Output operations to load the ROM data at the start of simulation. Since VHDL does not support generic binary IO, i wrote *romconv.py* to convert NES ROM files to an intermediate format that is compatible with ModelSims representation of the ROM array signals.

A third option i did not examine more closely is the Data2MEM tool by Xilinx, that allows to patch the existing Block RAMs in a bitstream file with new data. This is usually used for developing software for a softcore CPU.

Xilinx software limitation While it is possible to let XST infer initialized BlockRAM from the behavorial ROM description, it is very slow at this. I used the ROM description style recommended in the XST User Guide. Synthesizing something as simple as a 32+8 KiB ROM bumps up the synthesis time from 10 to 30 minutes on my system.

As a workaround, i used Coregen to pre-generate a netlist containing a BRAM. This is basically equivalent and just as slow as describing the ROM in HDL, but it only needs to be done once. The problem could also be solved by a build system that supports incremental compilation.

4.4 PPU

The PPU is usually considered the most complicated part of the NES by emulator developers, which i can confirm, as it took the longest time of all modules to implement and required several iterations of rewriting the code until i fully understood it.

The PPU is divided into the top module **NES_2C02**, and the submodules **TileFetcher**, **SpriteSelector** and **Loopy_Scrolling**

- H/V Pixel position counters
- VBlank interrupt generation
- The CPU port and status registers
- Color multiplexer combining sprite and background pixels
- Sprite 0 collision detection

- Nametable Video RAM
- RAM access multiplexer for tile and sprite modules

TileFetcher The module **TileFetcher** implements the background layer. It computes the addresses it wants to access based on the loopy_v register and outputs the current background pixel color.

The memory accesses are to the index, pattern and attribute bytes are interleaved, coordinated by the current horizontal pixel position. The fetched tile bits are put into a shift register for output.

Since the scrolling addressing register only allows to offset full tiles, there is a "Fine Scrolling X" to offset 0 to 7 pixels within the output shift register.

SpriteSelector The **SpriteSelector** completely handles the drawing of sprites, including the internal Sprite RAM (or Object Attribute Memory). The Sprite RAM is accessible by the CPU port through a set of Address/Data_in/Data_out/WriteEnable lines.

Since the TileFetcher pipeline needs to continously read memory while drawing the scanline, the sprite unit has to make all PPU memory accesses during the HBlank period. During the scanline, the internal Sprite RAM is accessed to select up to 8 sprites for the next scanline and store them in another buffer. During the HBlank period, **TileFetcher** does not need to do any memory accesses and **SpriteSelector** can fetch the pattern data for the pre-selected sprites.

These patterns are stored in a shift register, which is activated in the next scanline when the horizontal position of the sprite is reached. The outputs of these shift registers are routed through a multiplexer, that selects either the first non-transparent sprite, and forwards whether is sprite has priority over background, and whether it is the primary sprite.

Pixel Multiplexer This is implemented as a process in *PPU.vhd*, but in hindsight it would be better to place it into a separate module.

The pixel multiplexer takes the color outputs from **TileFetcher** and **SpriteSelector**, and decides which pixel color should be displayed. If both units display a color, the output depends on the sprite priority flag, otherwise it selects the non-transparent signal, or if both are transparent, the background color.

Sprite 0 collision should also be detected here, but is currently found in the CPU_Port process.

Loopy's Scrolling Registers To implement scrolling, the PPU used a set of intelligently chained counters, so that the addressing logic for tile fetches

could simply be implemented by incrementing these on every pixel. To save on registers, these counters were implemented with the same registers as the VRAM address port at $2006.

The idea behind this is that the CPU should do all VRAM accesses either in the VBlank time, or temporarily switch of PPU rendering. While rendering, the PPU continously access memory, and additional memory requests would either have to be ignored, or mess up the rendering.

Yet game developers found a way to make use of this dual use register, and change the scrolling values in the middle of rendering a frame. This technique requires careful timed code, but can be used to horizontally, or even vertically split the screen into sections with different scrolling values. For example, Super Mario Brothers relies on this to display the status bar on top as a part of the background layer, without moving like the rest of the layer beneath.

This phenomenon was first described in the reverse engineering community in 1994 by someone under the pseudonym “loopy”, which became a popular term to name the internal registers. This implementation follows this convention.

The PPU sub-module **Loopy_Scrolling** implements this dual-use register.

Known bugs Besides several things that could be written in a nicer way, there are several issues with the current implementation that i would like to address in a rewrite.

PaletteRAM has an address range of 32 bytes, but the first position of each 4 bytes refers to the same memory location, the background color. This mirroring is not implemented, instead we have a real 32 byte RAM. SMB writes the background color to offset 16, so this is currently hardcoded as background color.

The PPU can be configured to use 8x16 pixel sprites. They use a different addressing scheme that goes over the entire Pattern ROM space. There are some games that use this mode instead of 8x8 sprites, which would be unsupported at the moment.

The NTSC/PAL color generation circuit in the PPU can be influenced by 3 status bits, to emphasize either red, green, or blue and respectively darken the remaining colors. This could be implemented by extending **ColorPalette** to support additional palettes, but it is not relevant to most games. There is another bit for grayscale mode that is also unimplemented.

4.5 Controller Input

For simplicity reasons i decided to purchase an original NES controller and just connect it with the FPGA.

Since i did not have the heart to mutiliate a piece of working historic hardware, i decided to leave the original input connector and improvise a makeshift jack.

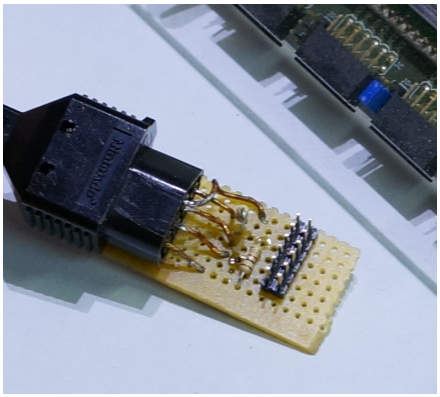


Figure 4.2: The improvised controller adapter

Because the chip in the NES Controller is not 3.3 Volt compatible, i needed to jumper the FPGA board to provide the board input voltage. A simple voltage divider made from 2 resistors is used to lower the voltage of the serial output to an FPGA-friendly level.

I did implement the address decoder in NES_Mainboard, since the one in **Dan_2A03** did not work for some reason. On this spot i also implement an interesting quirk:

Since the upper 5 bits of the data bus are not driven while reading $4016, the electric capacity on the bus stores the last value. In this case, because of the opcode layout of the 6502, it is always the higher address byte $40, resulting in reads of either $40 or $41, not $00 and $01 as one might expect.

4.6 HDMI Output

HDMI Output on the Digilent Genesys board is implemented with a Chrontel CH7301C DVI transmitter IC.

HDMI/DVI video uses TMDS (Transition-minimized digital signalling) as signalling standard, which basically encodes a digital representation of a VGA signal to a few high speed differential lines.

This means the FPGA has to provide unsigned RGB values and a horizontal and vertical synchronisation signal. The CH7301C needs to be configured via I2C to generate a matching internal clock and takes care of the rest.

For faster implementation, i reused the xps_tft EDK module. This implementation is fixed to a resolution of 640x480 pixels and contains an I2C state machine

for configuration and handles the instantiation of DDR registers necessary for communication with the Chrontel IC.

The original NES PPU ran synchronous to the NTSC clock to eliminate the need for expensive frame buffer memory. Since this project should later be run at different resolutions with an option to implement various scaling algorithms, this was not an option, so i decided implement a framebuffer with dual port RAM. The HDMI module uses one port of this RAM and scales the image to double resolution by simply doubling every pixel in horizontal direction, and every scanline in vertical direction.

Since the NES only knows 56 different colours, the framebuffer can store color indices with 6 Bit per Pixel. To save on BlockRAM, the internal framebuffer stores a 6 bit color index, that is defined as RGB in the **ColorPalette** module. This conversion is included in **HDMIOutput** for now, allthough it is probably not the conceptually right place.

4.7 APU

The APU was implemented last, and is the only major module that is written in the MyHDL language. This allows for a much more modular approach. *apu_convert.py* converts the module to Verilog code in *APU_Main.v*

The implementation can be found in apu.py. Each of the audio channels has an own module:

- **APU_Pulse**
- **APU_Triangle**
- **APU_Noise**

These are instantiated by the top module **APU_Main**, which also handles the register address decoding. **APU_FrameCounter** implements the internal APU timing counter, that is based on the 60 Hz frame interrupt, and is also used for **APU_Envelope** and **LengthCounter**, which are implement automatic volume decay respectively a note length limit.

The APU is instantiated in the CPU module **Dan_2A03**, like in the original design. While it could theoretically be moved to the top level **NES_Mainboard**, a later implementation that supports the DMC sample playback would require write access to the address bus and the possibility to halt CPU execution with the RDY flag.

Known Bugs The sweep unit of the pulse channels is still unimplemented. This results in wrong sound effects in SMB.

I also did not implemented the APU interrupts, the status read register, and the DMC unit because they are not used by SMB and therefore could not be tested.

4.8 AC97 Output

AC97 is a standardized interface for communicating with an audio DAC/ADC. The Digilent Genesys board features a National Semiconductor LM4550, which adheres to the AC97 standard.

I found an AC97 driver at opencores that worked right away in a test design with a sawtooth wave test output. I designed *ac97_top.vhd* as an interface for this module, offering only the ports necessary for my application.

Since the audio output is directly generated by fast clocked digital logic, and the AC97 output samples this at 48 kHz, the Nyquist-condition is violated, which means the output contains aliasing artefacts.

This results in a slight degradation of sound quality. The general solution to this problem is to process the signal with a low-pass filter before sampling it. A FIR filter could be implemented on the FPGA with DSP resources.

4.9 Utilization Report

Device Utilization Summary				[-]
Slice Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Registers	3,628	28,800	12%	
Number used as Flip Flops	3,627			
Number used as Latch-thrus	1			
Number of Slice LUTs	4,129	28,800	14%	
Number used as logic	4,103	28,800	14%	
Number using O6 output only	3,399			
Number using O5 output only	231			
Number using O5 and O6	473			
Number used as Memory	9	7,680	1%	
Number used as Shift Register	9			
Number using O6 output only	9			
Number used as exclusive route-thru	17			
Number of route-thrus	254			
Number using O6 output only	246			
Number using O5 output only	8			
Number of occupied Slices	1,410	7,200	19%	
Number of LUT Flip Flop pairs used	5,223			
Number with an unused Flip Flop	1,595	5,223	30%	
Number with an unused LUT	1,094	5,223	20%	
Number of fully used LUT-FF pairs	2,534	5,223	48%	
Number of unique control sets	446			
Number of slice register sites lost to control set restrictions	188	28,800	1%	
Number of bonded IOBs	66	480	13%	
Number of LOCed IOBs	66	66	100%	
IOB Flip Flops	14			
Number of BlockRAM/FIFO	24	60	40%	
Number using BlockRAM only	24			
Number of 36k BlockRAM used	22			
Number of 18k BlockRAM used	2			
Total Memory used (KB)	828	2,160	38%	
Number of BUFG/BUFGCTRLs	4	32	12%	
Number used as BUFGs	4			
Average Fanout of Non-Clock Nets	4.07			

Figure 4.3: Utilization Report

Chapter 5

Testing Tools

5.1 Testbenches

For developing audio and video components, in software as in hardware, it is crucial to be able to test the visible and audible results, instead of just being able to look at waveforms in a simulator.

Synthesis for verification The first naive notion of a testbench is to simply synthesize the whole project and visually check the results, maybe using a NES emulator for reference.

While this makes sense as both the first and final test for the project, it only provides a very bad workflow in between. Even with such a small project, Xilinx ISE still needs about 10 minutes from a change in the source code to a downloadable bitstream. This is detrimental to work motivation, and the debugging options on the result are very limited.

Full Game Emulation Writing a testbench for a legacy system is hard when the exact operation of the component is unknown, and is only specified be the requirement to be compliant with the original system. Since i started the project with an almost working CPU, it was easy to assemble a simulation model that could execute an existing ROM, and generate test stimuli for the PPU.

A simpler approach to test is to just assemble a simulation model that can execute an existing ROM file, and compare the results with an known-to-work emulator. The major drawback of this approach is simulation performance, as the whole system has to be emulated on the RTL level, instead of just the unit-under-Test, in this case the PPU.

SMB needs about 30 frames to boot, and about 100 more to start a game. This takes about 5 minutes on my Core2Duo testing machine, and requires a fast proprietary simulator like ModelSim. This is still not ideal for the programmers motivation, but much better than a 10 minutes of synthesis run that allows no introspection of the results.

In hindsight, it might have been better to start with an emulator that generates valid test signals for the developed component. Some problems regarding complex interactions between CPU and PPU can only be debugged with a model of the full system.

Audio Testbench Since i wrote the APU in MyHDL, i had the full power of Python at my hands to create a smaller and faster testbench than the full system emulation. This is found in *apu_tb.py*.

It is based on the NSF file format. Nintendo Sound Files are basically ROM files stripped from the graphics data, extended with a header that contains the adresses of the game music routines. The Python 6502 interpreter Py65[1] is used to execute the ROM and generate test stimuli for the APU.

cpu_bus.py implements the bridge between the CPU interpreter and the HDL description of the bus. It only supports writes, and dismisses reads and APU interrupts, but that seems to be enough for most NSFs.

ac97wav.py contains a module that samples the audio output at 48 kHz, like an AC97 codec, and writes to a .wav file, automatically named after the NSF filename and song number.

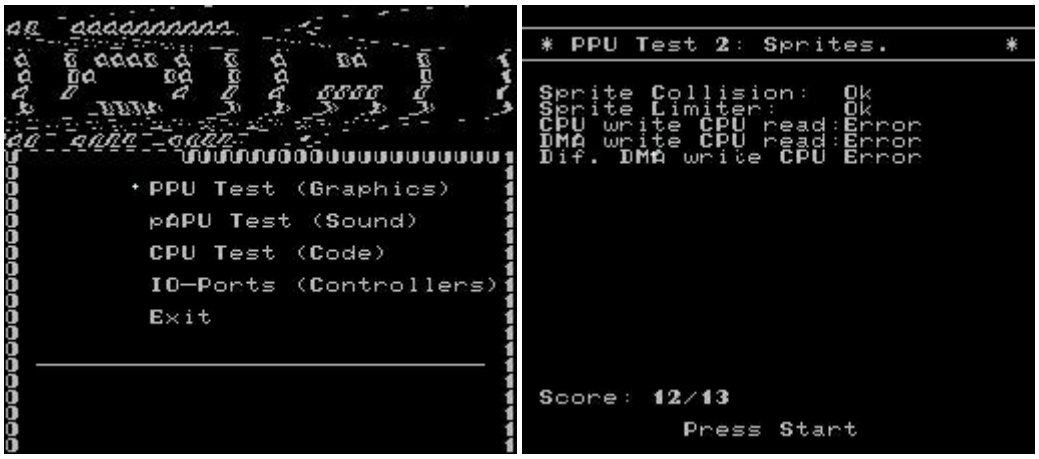
5.2 Testing ROMs

Super Mario Brothers This was of course the ROM i used the most for the debugging.

A most interesting resource is smbdis.asm, a completely commented disassembly of the SMB program code. This was very useful for debugging various hangups of the game code, since the current code position could be read from the address bus in the waveform.

https://gist.github.com/strfry/5788896

[1] https://github.com/mnaberez/py65



NEStress

NEStress (available on NESDev.com) is a self-testing ROM that is written for emulator developers.

It tries to test as much as possible from inside the NES. The CPU test tests various instruction, and also the specific operation of the undefined opcodes of the 6502. The latter test partly fails, but it does not seem to matter to SMB.

The PPU test covers various aspects of the CPU access port, the DMA Unit and the status flags. The main screen demonstrates a cool wave effect, that relies on a correct implementation of the scrolling registers.

The screenshots above show an emulator that seems to have not implemented the sprite DMA unit correctly. Note that not all emulators pass all the tests, even if they work fine for most games. But NEStress can still be a useful tool when one is at a dead-end while debugging, or to simply visualize progress on accuracy.

5.3 Self-written Tools

FBView Since Modelsim is unable to interpret a memory as bitmap for visually inspecting the framebuffer, i used the VHDL textio module to dump the framebuffer array to a file.

Technically the file format is unspecified. Luckily at least ModelSim and GHDL have a relatively simple though format. The bits are written out consecutevely as bytes, for representing the different values std_logic can take.

Relevant VHDL Code:

```
constant fb_size : integer :=   340 * 261 - 1;
type fb_ram_type is array(fb_size downto 0) of std_logic_vector(5 downto 0);
type FramebufferFileType is file of fb_ram_type;
```

```
[...]

FB_DUMP_proc : process (FB_Address, FB_DE)
  file my_output : FramebufferFileType open WRITE_MODE is "fbdump_top.out";
  variable my_line : LINE;
  variable my_output_line : LINE;
begin
  if rising_edge(FB_DE) and FB_Address = X"0000" then
    write(my_line, string'("writing frame"));
    writeline(output, my_line);
    write(my_output, fb_ram);
  end if;
end process;
```

fbview.c is a small C program i wrote to display the results of the full game testbench. It parses the binary dump formats, translates the NES color palette to RGB and displays the images using SDL (Simple DirectMedia Layer), and allows navigating through the generated video frames.

romconv.py This is a small python script that converts simple 32k NES ROM files, splits the program and pattern ROM part, and writes them either to a .coe file for the Xilinx CoreGen BRAM generator, or a special format that can be read by the VHDL code in *CartridgeROM.vhd*.

5.4 MyHDL

During the later stages of development, before writing the APU, i became tired of the tedious VHDL syntax and looked for higher level language that may act as a frontend for VHDL/Verilog.

I discovered MyHDL, an open source hardware description language based on the Python scripting language. It offers automatic conversion of a subset of the language to VHDL or Verilog, which allows it to be integrated in a proprietary toolchain.

MyHDL provides a framework to write a RTL model in Python code, that is simply executed for simulation. This means that the whole expressiveness of Python can be used for writing testbenches. This approach is similar, and offers similar advantages to other verification languages like SystemVerilog or SystemC.

For more detailed information, visit http://www.myhdl.org

MyHDL is still in an experimental status. Besides that, it has been successfully used to develop the digital portion of a mixed-signal ASIC. (Decaluwe 2010)

5.4.1 Code Example

This shall be a quick example of the MyHDL syntax:

```
from myhdl import *

def ShiftRegister(CLK, RST, Qin, Qout, Dout, Size=16):

  Q = intbv()[Size:0]

  @always(CLK.posedge)
  def my_sequential_process():
    if RST:
      Q.next = intbv(0)
    else:
      Q.next = concat(Qin, Q[Size - 1 : 0])

  @always_comb
  def my_combinatorial_process():
    Qout.next = Q[Size - 1]
    Dout.next = Q

  return instances() # Equivalent to return my_sequential_proces, my_combinatorial_process
```

What can be seen here is a function that represents the module. This function returns the processes that describe the behaviour, and possibly other module instances.

Processes are also functions, but are wrapped in a decorator like @always, that tells MyHDL when the function must be called, which makes it somewhat similar to a sensitivity list.

The .next syntax of signals explicitly model the delayed assignment of a signal, similar to the `<=` operator of VHDL.

Note that the type of a Signal is automatically determined by its initial value. Bit vector width is set with a slicing operation.

This way the type of a signal is only declared once where it is instantiated, and then passed as a function argument, instead of being re-declared multiple times for every appearance in another module.

5.4.2 Advantages

Compared with VHDL, MyHDL offers a radically simplified type system and a more concise syntax that simplifies refactoring code across modules.

Using Python greatly simplifies I/O code, without the need for external conversion tools. For example, the APU testbench directly reads and parses a NES ROM file for simulation, and writes .wav files for aural verification.

Using a widespread general purpose programming language also has the advantage of being able to reuse the debugging experience in that language. Starting a MyHDL program from the terminal has the lowest turnaround time of the simulators i have used so far. Python is often paired with a Test-Driven Development approach, a process that bears superficial resemblance with the common hardware development paradigm, where thorough testing is without any alternative. In TDD, tests are written before the actual code, and included in an automated testing system that allows to quickly discover regressions. This usually leads to more modular and smaller systems in general, and provides a basis for effective refactoring. It remains to be researched how this methodology can be successfully applied to hardware development.

Besides the current language level, several interesting ideas are currently drafted as MyHDL Extension Proposals (MEP). These include features allow for object oriented modeling (current MyHDL only supports modules as plain functions) or automatic inference of reset behaviour.

Being open source, MyHDL requires no license server that occasionally stops responding in the middle of work.

5.4.3 Shortcomings

MyHDL is an open source project still under development and may contain bugs that need to be worked around during development. These are mostly related to the conversion feature, since only a subset of Python can be automatically mapped to equivalent VHDL/Verilog code.

With Python being a interpreted language, simulation is several orders of magnitudes slower than commercial hardware simulators. It is possible to execute MyHDL code in PyPy, a python runtime written in python that implements JIT-compiling, performance can be increased by a factor of 10 to 20.

As the simulation performance effectively limits the design size, it is not possible to simulate the whole NES system without converting to VHDL/Verilog and using a fast external simulator.

This limitation can be circumvented by limiting RTL simulation to a single module under development, and replace the remaining modules with a software based behavorial model. Applying this approach to this project boils down to writing an NES emulator in python, and then iteratively converting the components to RTL code.

5.5 Other Useful Tools

FCEUX is a NES emulator that supports several interesting debugging features. This include a debugger with a integrated disassembler and viewers for visualizing PPU data, like the nametables at different mirroring modes and the pattern ROMs.

See http://www.fceux.com

Chapter 6

Future Work

The current state of the project is basically a console that runs a single game.

I am unsatisfied with the code quality in some parts, especially in the older modules were i lacked the fundamental understanding and HDL experience to do it better.

Stating from my current experience, i would like to continue exploring alternative languages like MyHDL, that allow to rewrite the code in a more concise way. Hofstra (2012) compares MyHDL with VHDL and two novel haskell-based hardware description languages, Kansas Lava and ClaSH, that may be interesting for evaluation.

Fortunately i am able to continue work on this project in the next semester to prepare some components to be used as exercises on HDL programming.

6.1 ROM Loading

Currently the project is able to load simple NES games.

Unfortunately, the game is hardcoded into the bitstream, and trying other games requires knowledge of the code, a working FPGA toolchain and a full synthesis run.

To make this work as a casual gaming console, there must be simpler ways to transfer a ROM file downloaded from the internet on the FPGA. For example, the FPGA might read them from a filesystem on a SD card or a USB stick.

The other requirement is a more flexible cartridge module that implements different mappers and reads the configuration and ROM data from a RAM.

Reading and Parsing the NES ROM File, and then writing it to this RAM is the kind of task that is best done by a microcontroller, or an embedded softcore.

As stated in the cartridge implementation details, a different development board with SRAM might facilitate this.

Another issue to consider are games that ship battery-backed RAM to store their savestate. Most modern non-volatile memories, like Flash or EEPROM allow only a limited number of write cycles. Games might also use their battery-backed RAM as work RAM, so it might be a bad idea to save every write that is made there. A simple solution might be to use an explicit save button on the hardware.

An entirely different approach would be to manifacture an adapter for original NES cartridges. The connector is still available as a replacement part for repair. Of course, since i am located in europe, it would be nice for the console to support PAL mode, allthough i am not sure if this is technically necessary for all PAL games.

6.2 Accuracy Features

Currently the game is only well tested with Super Mario Brothers. Besides the known omissions mentioned in the implementation chapter, there might be other bugs that prevent other games from working. This task goes hand in hand with the implementation of a cartridge that supports memory mappers.

It might be helpful to do first create a working NES emulator in Python, or reuse an existing one, which could be part for part be replaced by MyHDL code.

If the CPU causes any problems in the future (NEStress still complains about some undocumented opcodes), there are better alternatives that could be reused. The FPGA64 project for example offers a HDL model of the 6502 that includes correct timing and undocumented opcodes.

6.3 Video Scaling

The current HDMI output only supports a resolution 640x480 and doubles each pixel to scale up the NES resolution.

Emulators have already developed better algorithms

Kopf and Lischinski (2011) even generate a smooth vector representation, by exploiting the fact that graphic assets of this gaming era are usually hand-drawn pixel for pixel, which makes every pixel a feature.[1]

Their algorithm would need to be adapted to work on in an emulator, where the scaler has to run on the full screen, since the graphics data is stored and small tiles, and are assembled to game objects at run-time.

[1] http://research.microsoft.com/en-us/um/people/kopf/pixelart/

However, their comparison data suggest that simpler algorithms like the hq2x/hq4x family, or EPX also generate satisfactory results. These algorithms should match nicely to a hardware implementation. This might be candidate for a lab exercise.

Chapter 7

Conclusion

During this project, that took much longer than expected, i learned a lot about FPGA technology and development methodology.

In hindsight, i underestimated the time necessary for understanding the NES hardware, since its interactions are rather complex. It might have been easier to first write a purely software based emulator before trying to bring it to an FPGA.

Another lesson was that the FPGA vendor tools often introduce unnecessary complexity, and that many things about hardware design are better learned with a low level do-it-yourself approach. For example, Coregen tries to make things simpler by providing a shiny GUI that allows me to configure a FPGA primitive like Block RAM, but at the same time gets in the way of learning of to do this purely in HDL code. Another example is the synthesis build system, where Xilinx offers various ways (ISE, xflow, xtclsh) that only exist because the synthesis tools (xst, map, etc.) have an awful command line user interface, which makes it hard to write a makefile myself.

Coming from a software development background with an interest in open source software, I searched for free software solutions that simplify the development process. MyHDL was an interesting find, but it is limited by the fact that it is only a frontend to traditional toolchains, that rely on either VHDL or Verilog, that would be considered ancient languages in the software world.

OpenCores.org was also a valuable resource, as i could reuse an unmodified AC97 driver i got from there. Still, reusability of many hardware modules is limited, which could be a sign that todays HDLs are inadequate for cooperation. There are commercial products that try to provide better design tools and languages, but these are not publicly available, and in consequence, worthless for open source developers.

I think there is a lot more potential for open source development and free

software in the FPGA world. Drawing an analogy with the creation of open source operating systems like GNU/Linux, we seem to be at a very early point of this development.

In the 80ies, when Richard Stallman saw the need to create an alternative to the proprietary platforms and toolchains people were forced to use, he began with an editor and various other tools that were developed with a closed toolchain. The next step was the GCC (GNU Compiler Collection), the first free software C compiler at this time. This allowed to modify the compiler for new platforms and easily port the existing free software, which was the basis for the development of the Linux kernel. This in turn created an open source ecosystem that started producing new and innovative programming languages, and heavily influenced todays software development methods.

In the FPGA world, we seem to be stuck at the stage of having open source editors and various applications, that all still rely on the vendor software to produce a working bitstream. The vendors state on the one hand that building a new place&route software is infeasible, and on the other hand keep their bitstream format secret, forbidding any reverse engineering in their licensing terms, so that no one can even give it a try. Allthough human interaction with place & route is sometimes necessary for big and complex designs, it is only possible through specifying constraints in an UCF file, and doing everything else in the HDL.

@fpgatools tries falsify this common knowledge by reverse engineering the bitstream format of a small Xilinx Spartan 6 FPGA, and providing a C-based framework for doing place&route manually. Maybe there are design problems that could benefit from specialized approaches to the place & route problem.

Chapter 8

References

Decaluwe, Jan. 2010. *MyHDL-based design of a digital macro*. http://www.jandecaluwe.com/hdldesign/digmac.h

Hofstra, Matthijs. 2012. "Comparing Hardware Description Languages." http://referaat.cs.utwente.nl/conference/17/paper/7344/com.

Kelley, Andrew. . http://andrewkelley.me/post/jamulator.html.

Kopf, Johannes, and Dani Lischinski. 2011. "Depixelizing Pixel Art." *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2011)* 30 (4): 99:1–99:8.

Leach, Dan. "NES-On-A-FPGA." http://cegt201.bradley.edu/projgrad/proj2006/fpganes.

VII, Tom Murphy. 2013. *The First Level of Super Mario Bros. is Easy with Lexicographic Orderings and Time Travel . . . after that it gets a little tricky.* http://www.cs.cmu.edu/~tom7/mario/mario.pdf.

Chapter 9

Appendix

README.md