
EDA385 - Embedded Systems Design - Advanced Course
2008-10-15

Henrik Kjellberg, et05hk6
Carl Hakenäs, e04ch
Daniel Fritze, d03df
Stefan Frid, d04sfr

# Abstract

This report is a part of the project course Design of Embedded Systems, Advanced Course. In the course a clone of the video game Guitar Hero was created. The target platform was the Spartan 3E based FPGA board Digilent Nexys2 running a Microblaze CPU. One objective with the design was to make a general game platform and not a design limited to Guitar Hero. Focus has been on video performance. This report discusses software and hardware in detail as well as performance results and possible improvements. Design methodology and project plan is also discussed.

# Contents

# Introduction

The general idea behind the project was to make a simplified version of the popular console/PC music game Guitar Hero.

In the original version of the game, basic gameplay revolves around playing guitar notes in sync with the music. On the screen, a scrolling guitar fretboard is shown, scrolling in 3D towards the player. Color coded notes are placed on the fretboard. When the notes reach a certain line, the player has to hold down the corresponding button(s) on the supplied guitar controller, and hit the strum bar. If the note is not played correctly, a short "missed note" sample is played, and the guitar channel is muted. Difficulty rises as note complexity and amount is increased.

Our game is vastly simplified compared to the original, but maintains the basic idea of hitting animated notes using a guitar controller. We omitted nearly everything except main gameplay (introduction, menus, options, career gameplay etc.). Some gameplay elements, such as whammy bar and "Star Power" will initially be left out. Our graphics will be 2D, and without most of visual flair of the original (including animated rock avatars, audience, fretboard etc.). The original version of the game has two stereo channels for the music, one with only guitar, and one with the rest of the instruments/vocals. Our version only features one channel of stereo audio, with audio separation done between the two speakers. This eliminates the need for sound mixing.

Thes unit is based upon a Digilent Nexys 2 system design platform, containing a Xilinx Spartan 3E FPGA, 16Mbytes of SDRAM and 16Mbytes of Flash ROM. The FPGA hosts a Microblaze™ microprocessor wich runs the game software. Besides the Microblaze™ the unit holds a custom made bursted VGA controller with hardware sprites and a framebuffer, a DMA controller for memory shoveling, a timer with a interrupt controller to generate periodical interrupts in the microprocessor and a custom made audio controller.
A custom made guitar control is connected using pullups and regular button inputs.

The software is mainly interrupt driven. On a frequency of  2kHz the game is stepped forward, with checking of inputs and updating of graphics and sound. The only thing done in the main loop is the generation of the flame effect.

# Hardware

## General

Device utilization summary:
---
Selected Device : 3s1200efg320-5

| | | | |
|---|---|---|---|
| Number of Slices: | 7177 | out of 8672 | 82% |
| Number of Slice Flip Flops: | 6474 | out of 17344 | 37% |
| Number of 4 input LUTs: | 9862 | out of 17344 | 56% |
| Number used as logic: | 9345 | | |
| Number used as Shift registers: | 261 | | |
| Number used as RAMs: | 256 | | |
| Number of IOs: | 73 | | |
| Number of bonded IOBs: | 73 | out of 250 | 29% |
| IOB Flip Flops: | 64 | | |
| Number of BRAMs: | 25 | out of 28 | 89% |
| Number of MULT18X18SIOs: | 4 | out of 28 | 14% |
| Number of GCLKs: | 1 | out of 24 | 4% |
| Number of DCMs: | 1 | out of 8 | 12% |

| Overview | |
|---|---|
| Generated on | Thu Oct 16 17:42:08 2008 |
| EDK Version | 10.1.02 |
| FPGA Family | spartan3e |
| Device | xc3s1200efg320-5 |
| # IP Instantiated | 21 |
| # Processors | 1 |
| # Busses | 3 |

## MicroBlaze™

The MicroBlaze™ embedded processor soft core is a reduced instruction set computer (RISC) optimized for implementation in Xilinx® Field Programmable Gate Arrays (FPGAs). Figure 1-1 shows a functional block diagram of the MicroBlaze core.

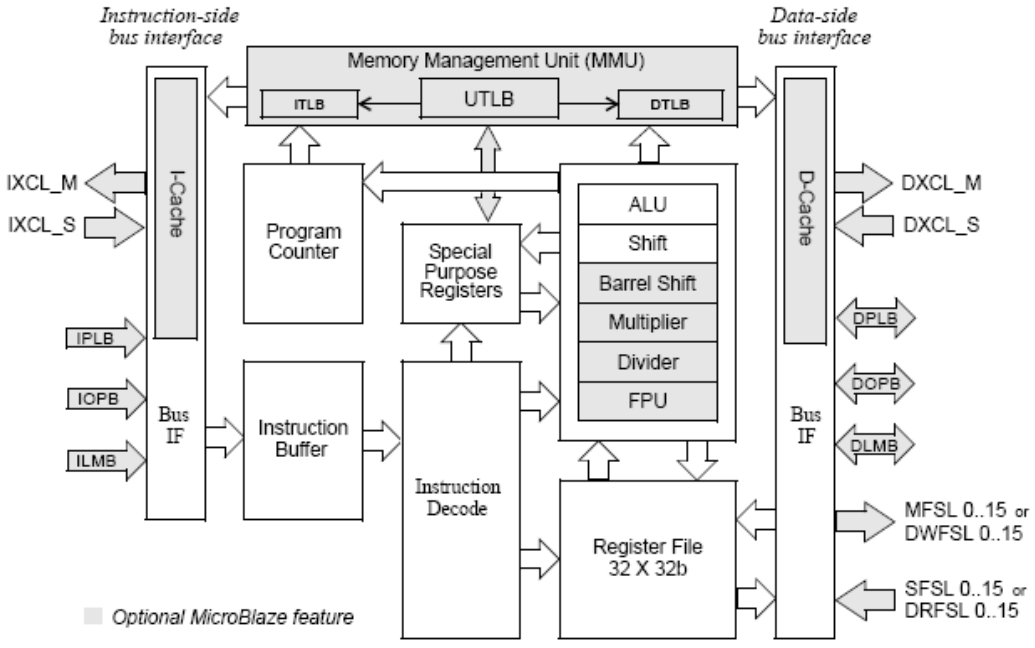


*Figure 1-1:* **MicroBlaze Core Block Diagram**

**Timer**

The timer used is a XPS Timer/Counter core for the Processor Local Bus (see figure 1). The timer is configured by the Microblaze™ through the PLB bus to send a interrupt signal at 2KHz. The Interrupt signal is then connected to a interrupt controller which arranges all available interrupts by priority order and forwards them to the Microblaze™.

| Post Synthesis Device Utilization | | | |
|---|---|---|---|
| **Resource Type** | **Used** | **Available** | **Percent** |
| Slices | 330 | 8672 | 3 |
| Slice Flip Flops | 368 | 17344 | 2 |
| 4 input LUTs | 387 | 17344 | 2 |
| IOs | 213 | NA | NA |
| bonded IOBs | 0 | 250 | 0 |

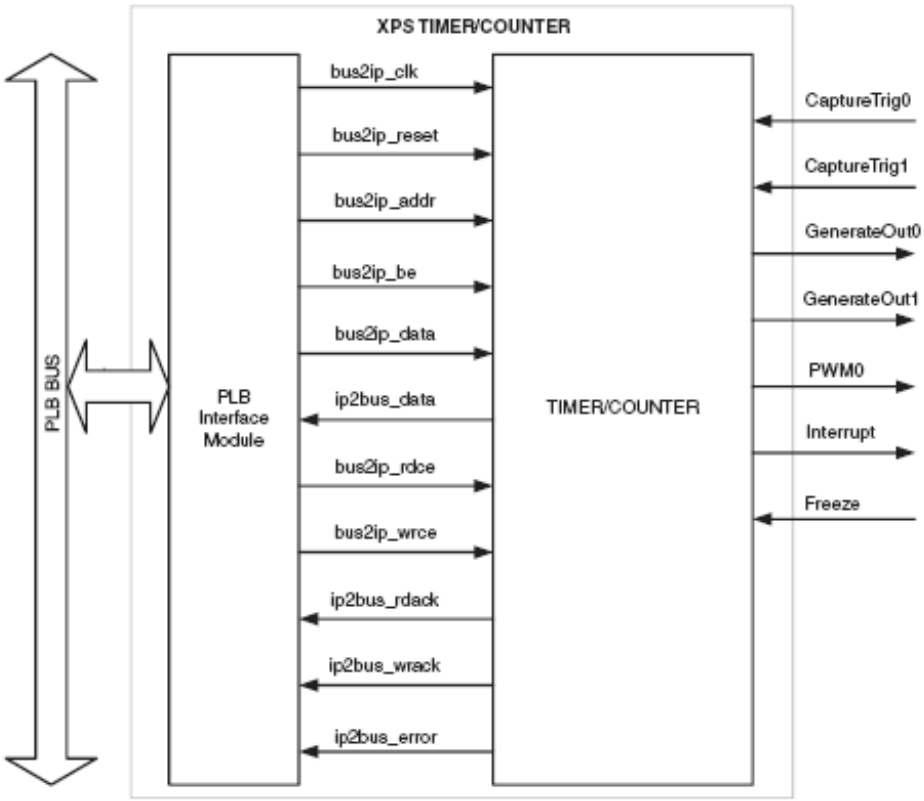


*Figure 1:* **XPS Timer/Counter Top-Level Block Diagram**

## Interrupt Controller

The XPS Interrupt Controller concentrates multiple interrupt inputs from peripheral devices to a single interrupt output to the system processor. When a interrupt occurs, the MicroBlaze™ processor asks the interrupt controller what peripheral that requested the interrupt and executes the corresponding interrupt handler.

| Post Synthesis Device Utilization | | | |
|---|---|---|---|
| Resource Type | Used | Available | Percent |
| Slices | 86 | 8672 | 0 |
| Slice Flip Flops | 123 | 17344 | 0 |
| 4 input LUTs | 87 | 17344 | 0 |
| IOs | 208 | NA | NA |
| bonded IOBs | 0 | 250 | 0 |

In current version the system only uses one interrupt, namely the timer interrupt, but the interrupt handler is implemented for future expansions.

## RS232 Controller

The XPS Universal Asynchronous Receiver/Transmitter (UART) Lite Interface is simply used for debug purposes.
By using the supplied drivers and Xilinx custom made print-function xil_print(), debug data were printed on the serial port and displayed with the Microsoft HyperTerminal.

| Post Synthesis Device Utilization | | | |
|---|---|---|---|
| Resource Type | Used | Available | Percent |
| Slices | 116 | 8672 | 1 |
| Slice Flip Flops | 151 | 17344 | 0 |
| 4 input LUTs | 141 | 17344 | 0 |
| IOs | 209 | NA | NA |
| bonded IOBs | 0 | 250 | 0 |

## SPI interfacing

The SPI protocol is a very simple way of connecting peripherals to a micro controller. Two devices are connected together with one being the master and one being slave. The data is sent in full duplex, ie. one line for master output and one line for slave output, the so called MOSI and MISO (Master Out, Slave In and Master In, Slave Out).

Amongst many other, Xilinx supply a core IP for a SPI-module. This IP has a wide support for various applications, such as FIFO-transfer queue, polled and interrupt based operation and multiple masters and slaves on the same SPI-bus.

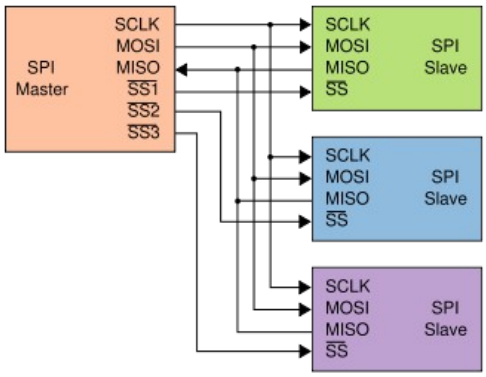


*Figure 2: Standard SPI Bus*

The idea was to use SPI for communication between the guitar control and the FPGA. In the FPGA-Hero there is 5 buttons on the guitar control, so the eight bits in the SPI shift register should be used as on/off-flags where each bit represented one button. The three remaining bits could then be used for expansion in form of a whammy-bar in case of extra time.
The FPGA was supposed to be the master and the guitar control was supposed to be the slave, based on a AVR microprocessor.

## The guitar control

The guitar ended up with four push buttons on the guitar neck and a toggle switch as the strum. All switches were pulling to Vcc through a resistor and shorting to ground when pressed. The switches were connected through flat wire to a small proto-board with the pull-up resistors and a connector to the Pmod connector.

## General Purpose Input/Output

The General Purpose Input/Output (GPIO) core is used to connect input buttons to the MicroBlaze™ through the PLB bus.
The GPIO is used in polled mode and checked from the 2KHz interrupt to check if the player presses any buttons on the guitar control.

| Post Synthesis Device Utilization | | | |
|---|---|---|---|
| **Resource Type** | **Used** | **Available** | **Percent** |
| Slices | 60 | 8672 | 0 |
| Slice Flip Flops | 92 | 17344 | 0 |
| 4 input LUTs | 52 | 17344 | 0 |
| IOs | 303 | NA | NA |
| bonded IOBs | 0 | 250 | 0 |

## DMA Controller

The XPS Central DMA Controller provides simple Direct Memory Access (DMA) services to peripherals and memory devices on the PLB. The controller transfers a programmable quantity of data from a source address to a destination address without processor intervention.
It is used to transfer the framebuffer data from the SDRAM to the VGA controller FIFO when triggered from the MicroBlaze™

| Post Synthesis Device Utilization | | | |
|---|---|---|---|
| **Resource Type** | **Used** | **Available** | **Percent** |
| Slices | 585 | 8672 | 6 |
| Slice Flip Flops | 565 | 17344 | 3 |
| 4 input LUTs | 1025 | 17344 | 5 |
| IOs | 392 | NA | NA |
| bonded IOBs | 0 | 250 | 0 |

## VGA Controller

The VGA controller consists of three parts

- VGA signal generator, generates 640x480 60Hz signal
- Framebuffer, displays a portion of RAM memory on screen
- Sprite generator, draws sprites using hardware rendering

| Post Synthesis Device Utilization | | | |
|---|---|---|---|
| **Resource Type** | **Used** | **Available** | **Percent** |
| Slices | 3879 | 8672 | 44 |
| Slice Flip Flops | 2824 | 17344 | 16 |
| 4 input LUTs | 5063 | 17344 | 29 |
| IOs | 216 | NA | NA |
| bonded IOBs | 0 | 250 | 0 |
| BRAMs | 8 | 28 | 28 |
| MULT18X18SIOs | 1 | 28 | 3 |

The signal generator uses two counters to generate the sync pulses needed for the VGA pulse generation. The hcnt counter is the horizontal counter and vcnt is the vertical counter. When the horizontal counter reaches a certain value the vertical counter is increased. Both counters are reset when they reaches a certain value.
The horizontal counter is increased with the 50MHz system clock. Pulses and counter values are given by the following:

- Horizontal max value: 1599
- Vertical max value: 520
- Hsync is set to zero when hcnt is below 190
- Vsync is set to zero when vcnt is below 1

Software decides the active region for the VGA signal. This is used to control the size of the framebuffer and thereby also the amount of data needed to read from RAM. A register exists in the output to reduce screen artifacts.

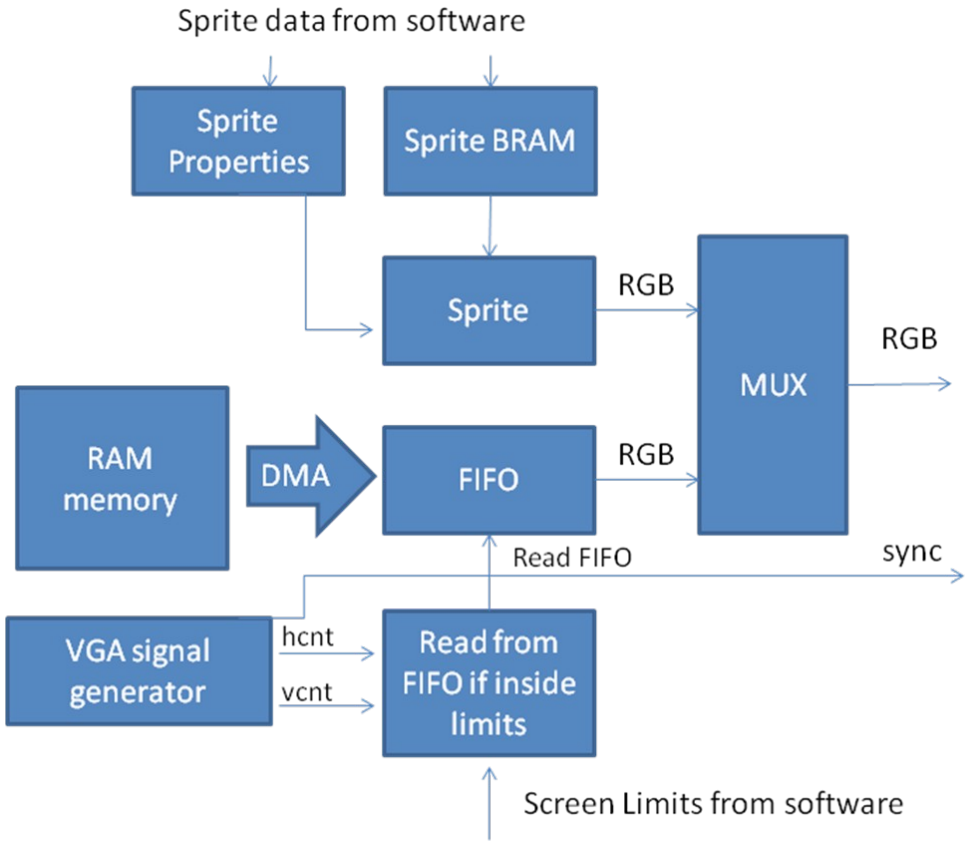


*Figure 3: VGA Controller overview*

**Framebuffer**

The vga controller reads data from the FIFO when the current pixel is inside the active region. It is not the VGA controls responsibility to assure that correct data exists in the FIFO. This is handled by a software trigged DMA controller. The data read from the FIFO is 32bits wide which corresponds to 4 pixels. To assure stable operation a register is located between the FIFO output and the actual drawing logic.

The bottleneck for the framebuffer is the RAM where the framebuffer resides. According to the Nexys2 reference manual the read cycle is 70ns and 16bits. The maximum bandwidth would then be:

```
1/(70*10^-9)  * 2bytes= 27.24 MB/s
```

With a FIFO as big as the framebuffer and infinite speed of the FIFO the maximum theoretical resolution can be obtained by:

27.24 MB/s / 60s = 237677 bytes = 991x480

But the maximum usable resolution obtained the frame buffer is 420x480 bytes. Without the FIFO the RAM would have had the same speed as the pixel clock, 25 Mb/s. Also no other operations would be allowed to access the RAM during this operation.

**Sprites**
The VGA controller enables hardware based rendering of sprites. Several sprites of any size can be rendered to the screen simultaneous. Sprites are drawn 'on-the-fly' and are never rendered to the framebuffer. Sprites are drawn above framebuffer pixels. The graphical sprite data is stored in a BRAM inside the VGA controller called pixel data. This is a 8bits wide memory representing the pixels in 8bit format, if all bits are zero the pixel is treated as transparent. All sprites have three properties: location, size and pixel data offset.

- Location is the x&y screen coordinates of the sprite, these coordinates are in 640x480 resolution including all non-visible screen areas.
- Size determines a sprites width and height. These can be any number ranging from 0-255. However behavior when a sprite is larger than the pixel data is undocumented.
- Pixeldata offset determines where in the pixel data buffer a sprite has its content. Pixeldata address for a pixel is calculated using:

```
address = offset + width * y + x;
```

Where x and y are coordinates within the sprite. These properties are stored in forms of arrays where each sprite has its own index. All properties can be changed during runtime via registers. A detailed explanation of these registers can be found below.

**Rendering**
A sprite's visibility is evaluated during creation of the VGA is signal. All sprites are evaluated simultaneously for all pixels. The location is compared with the current pixel drawing position. If the pixel is within the sprites boundaries the pixel data coordinates is calculated. Using these coordinates and the pixel data offset a pixel data address is calculated.
Only one sprite can be drawn on one pixel, if several sprites exists on the same pixel only the sprite with highest array index will be drawn. The sprites are evaluated on a 50MHz clock, to meet timing requirements and BRAM specifications the procedure is pipelined in four stages.
The stages are:

1. Determine if a sprite should be drawn on this pixel if so calculate sprite coordinates .
2. Calculate sprite address using pixel data offset and above coordinates.
3. Set address to BRAM
4. Read from BRAM, draw pixel if it's not transparent else draw framebuffer pixel.

The pipeline introduces a 4 pixel delay, this is compensated by having a counter that is 4 pixel ahead of the horizontal counter.
All pixel properties are stored in arrays where each sprite has a unique index. Since all sprites are evaluated simultaneous the number of if statement grows with the number of possible sprites. The maximum number of sprites is determined by a constant in VHDL and cannot be changed during runtime.
The current implementation uses 40 sprites and a sprite memory of 8Kb. With 40 sprites the systems critical path is in pipeline stage 1. Lesser sprites will create a shorter path in stage 1. Each sprite's properties occupy the following bits as registers:

- Location 11x2 bits
- Size 8x2 bits
- BRAM offset 16 bits
- Total 54 bits

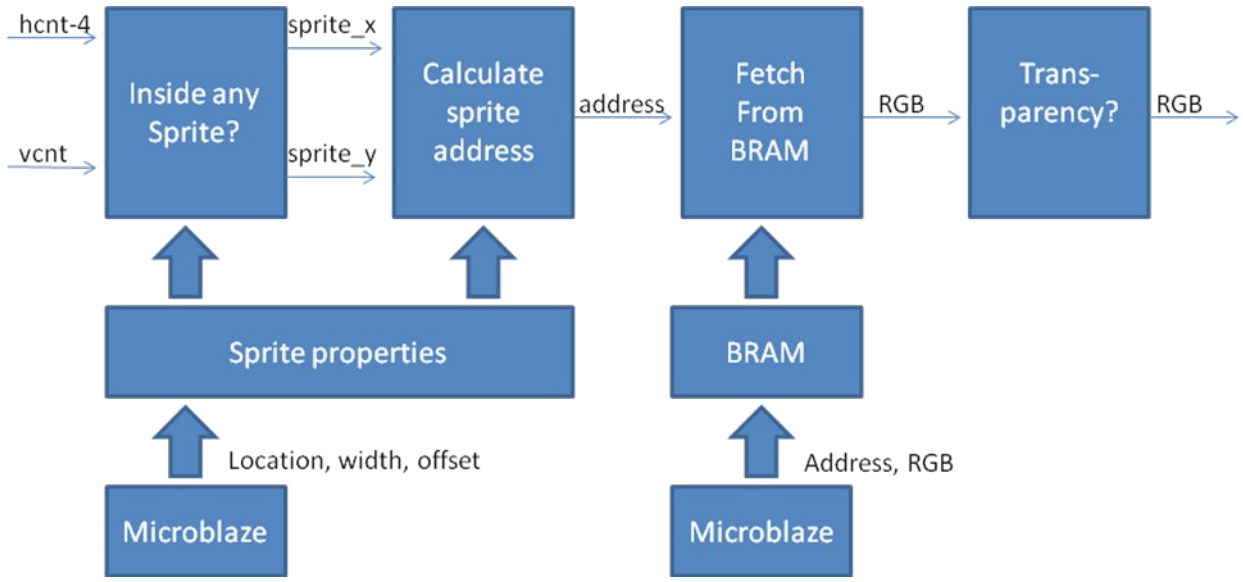


*Figure 4: Sprite pipeline*

**Sprite properties**

Sprite properties can be access using the following input signals:

```
valid_sprite std_logic;
set_sprite_size std_logic;
set_sprite_offset std_logic
fill_sprite_mem std_logic;
sprite_index std_logic_vector(7 downto 0);
sprite_x std_logic_vector(10 downto 0);
sprite_y std_logic_vector(10 downto 0);
sprite_address std_logic_vector(15 downto 0);

Location of a sprite
valid_sprite = '1', sprite_index = sprite_number, sprite_x, sprite_y =
location.
Sprite size
set_sprite_size = '1', sprite_index = sprite_number, sprite_x, sprite_y =
size.
Sprite pixel data offset
set_sprite_offset = '1', sprite_index = sprite_number, sprite_address =
offset
Set pixel data
fill_sprite_mem = '1', sprite_address = address, sprite_x(7 downto 0)=RGB
```

The sprite properties can be changed anytime. A sprites location can change when its being drawn. Doing this may result in artifacts.

Possible improvements

All elements of the location and size array needs to be accessed simultaneous, therefore they must be implemented as registers. Since only one sprite can be visible at once only one pixel data offset needs to be accessed. To save resources in terms of LUTs this property could be implemented as distrusted or block RAM. This would save about 30% of RAM.

Both pipeline stage 1 and stage 2 requires a lot of logic to be evaluated during one clockcycle. Additional pipeline stages would remove this limitation, for instance: the additions and multiplication in stage two could be split into two stages.

## Audio Controller

Audio is stored in flash memory, PCM coded with 22.05kHz 8bits stereo. The audio controller reads from a PLB connected FIFO and produces a PWM signal that corresponds to the PCM sample. Data is being read out from the FIFO with the same speed as the sample frequency.

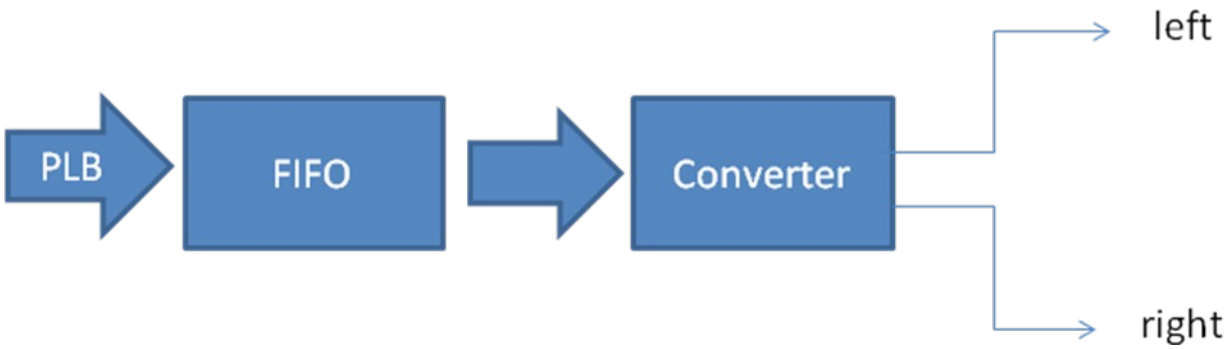


*Figure 5: Audio Controller overview*

The converter seen in fig 5 is built up by two counters. One counter that scales the clock frequency and the other counter that produces the actual PWM signal.
The PWM period is the same as the sampling frequency. The PWM counter is ranging from 0 to 255 which are the same as the PCM resolution. If the PWM counter is below the current sample value a '1' is presented on the output else '0'. The PWM counter is updated when the scaling counter reaches a certain value given by:

$$50000000 \text{ Hz} / 22050 \text{ Hz} / 256 = 8{,}86$$

When rounded to 9 the actual samplings frequency is 21.701kHz.

# Software

## Initialization

In the beginning of our program we set VGA controller parameters telling what resolution we are using. After that the background picture is read from the flash memory and written to the frame buffer. The sprites are then read from the flash and written to the sprite memory. Last of all the intro is started.

## Game logic

The purpose of the game logic is to determine if the player manages to hit buttons at the correct time, shown on the monitor, and then adjust the sound accordingly. To do this we need to have music, and data that tells when to hit the different notes. As mentioned before we have borrowed this from the open source project Frets on fire. The timings of the notes were stored in midi files so we had to write a converter in java. The converter takes a midi file as input and creates four c arrays as output. Each of the arrays contains data telling at which time in ms that particular note should be played. In our game we have four structs that contains: time arrays, number of notes the array contains, the note that should be played next, the lowest note to be shown on screen.

Our program contains two parts, a infinite main loop and a interrupt that occurs with 2 kHz frequency. All time critical parts of our program are handled by the interrupt. It contains of two software parts that are executed in this order: handle input, draw column. The main loop is only used for a flame effect that is not important to make the game work.

The handle_input method checks which buttons are pressed and compares them to the ones supposed to be pressed. It also checks if a note has passed the last position it can be hit at. If the player managed to hit a note the points increases. If he hits a note thats not supposed to be hit or if a note falls to far down, the guitar sound stops and instead a fail sound is played.

The draw_column method is called four times, once for each note. This method calculates which notes in the array that should currently be visible on screen and at what coordinates they should appear. Then the notes are drawn on screen using the method draw_hw_sprite.

## Hardware management

The 2 kHz interrupt also contains two hardware management parts. The first one is calling the update_sound method, which makes sure that the sound buffer is filled at all time, by copying data from the flash memory to the audio FIFO. The second one makes sure the VGA FIFO is filled all time. This is done by initiating the DMA to copy a block data from the RAM memory and placing them in the queue.

## Graphics

We have two ways of displaying graphics on the screen. We can either do it with a frame buffer or with sprites drawn in hardware. We use the frame buffer to draw background pictures and our flame animation. The hardware sprites are used to draw the falling notes, the letters and numbers. The reason that we don't do this in the frame buffer is that we only can update about one fifth of the screen before the next frame is drawn. We would also have to use double buffering to prevent flickering.

## Tools

**MIDI note reader**
The game software uses a number of C arrays with time stamps at which notes should be played. The amount of work needed to generate good note arrays for a song would be very large. Because of this, we use note data from a open source Guitar Hero clone, Frets On Fire. In this game, note data is stored in a MIDI file.

Much of the information contained in these MIDI files is not needed by us. Note sets for different difficulty levels are included, as well as bass guitar information. Some other data, such as “Star Power” markers, are also included.

We built a simple Java program that uses java.sound.midi classes to read and parse a MIDI file. These classes provide us with many tools for filtering the data, such as only working with the guitar track, and difficulty selection by choosing only notes from a certain octave.

When invoked, the program reads “notes.mid”, reads through the MIDI event information, filters out unwanted events, converts the timing information of the events to real time, and writes the result in a C struct format to the console. We do not read the BPM of the song from the MIDI file, this has to be hard-coded in the Java source file.

**Image converter**
Our VGA controller uses 8 bits per pixel, 3 bits red/green and 2 bits blue. Since 3:3:2 RGB RAW is no standard format, we had to build a simple image converter.
We built this application in Java, using java.awt libraries to load and process the image.
When invoked, the program opens an image specified in the source code and outputs this file as a 3:3:2 RGB RAW, suited to be displayed with our VGA controller.

**Flash programmer**
The flashprogramming tool is used to program the flashmemory over USB with a file. It requires that ExPort is running in the background. The FPGA board must be running the default program or another software capable of programming the FLASH using CFI over EPP. For runtime specific operations run the program without parameters.

**Audio merger**
Combines two audio files to a single file with two channels put at alternating bytes.

## Development

Since we had quite short time to develop the whole project we decided to work simultaneously on both software and hardware. We started by working on game design, audio controller and inputs using SPI. The game design was done by working on a frame buffer in DOS simulated by using DOS-box. The input from the keyboard was also simulated in the same format as we would be getting it later. We decided not to try to simulate the sound as it would be far to much trouble. The sound controller was also close to completed by this time. After a few weeks we had our game partially working on DOS.

After that we started to work on the VGA controller using a frame buffer. We managed to get it working but after we tried our code on the frame buffer we realized that we wouldn't be able to write as much as needed without flickering. To solve this problem we decided to use the frame buffer for background pictures and effects, and used hardware sprites to handle the drawing of the notes, digits and numbers.

On the hardware side FLASH and audio was first prioritized and the biggest question marks. This because we realized that flash and audio was the least the documentation areas and also not many other groups worked on this. After audio and flash was implemented the VGA controller was developed The VGA controller got more and more advanced in order to meet our growing performance requirements.

The development of the SPI interface ran into several problems, mainly caused by lack of documentation. When time were running out we decided to skip the SPI interface and instead use the simpler XPS GPIO core for the guitar control.

# Problems

## SPI-problems

Before finding a large library of example-codes for all of Xilinx's core IP's we struggled to understand and decompose the supplied drivers. When we finally got the example codes we threw away all our old SPI-code and tried the examples. When trying the example of a polled implementation everything seamed to work fine, but when trying to connect the core to the outside world and the prototype of the guitar control, nothing happened. When we finally solved the problem, it turned out to be that in loop back mode (as used by the example code), the slave select-bit didn't have to be set in order to transmit data. But when not in loop back mode, the transmission is hold until the slave select-bit is set. When the SPI module in the FPGA finally was working, it turned out that the guitar control didn't get all bits right and the returned value was very random. Since time was running out, we decided to skip the SPI solution and instead use the simpler button-handler IP, seen in the board's default program. The guitar control was modified to shorten the pins in one of the Pmod expansion ports to ground when buttons were pressed and otherwise pulled up to Vcc. The drawback of using the built in button-handler instead of a external microprocessor with SPI is the limitations in expansions.

## FLASH-problems

At first we tried to program the 16 MB flash memory from our program running on the FPGA. We tried to program it using the built in XilFlash libraries but it failed. We also tried to program the flash memory using a JTAG cable but with no success.

The default program on the FPGA board contains a FLASH selftest and code for accessing the flash. By analyzing this code a tool was created using *Digilent* Port Communications API. With this the flash memory could successfully be programed, however only half of the words could be used the rest was corrupted. Later our supervisor discovered an error in the system generated by the wizard, this solved our problem and we got a working flash memory.

Since we had our own customized tool that was suited for our needs we didn't bother to try the XilFlash library or the JTAG.

## VGA-problems

We had problems with artifacts with data on certain byte alignments.  After inserting a register between the FIFO and drawing logic the artifact disappeared. However the extra delay the register caused introduced some bugs but these where taken care of.

# Contributions

**Hardware:**
Mainly Carl and Henrik

**Software:**
Mainly  Stefan and Daniel

**Testing:**
Everyone

**Report:**
Everyone

**System Architecture:**
Everyone

# Installing FPGA HERO on the Nexys2 board

Hardware Configuration
---
1. Connect the PModAMP1 module into the top row of Pmod JD
2. Connect headphones/speakers to the left audio jack
3. Connect the guitar Pmod JB, make sure that the connection is aligned to the left
   (VCC and GND should be connected)

Software Configuration
---
1. Unzip the fpgahero9.zip
2. Start ExPort, run initialize the chain and turn on the board (default config must be running)
3. Execute fpgahero9/media/prog_audio.bat
4. Execute fpgahero9/media/prog_images.bat
5. Start Xilinx Platform Studio
6. Open fpgahero9/system.xmp
7. Execute Device Configuration -> Update Bitstream
8. In Export choose fpgahero9/implementation/download.bit
9. Disable the ROM by clicking it
10. Click Program Chain