Project report
MIDI MONSTER

EDA385 – Design of Emedded Systems, Advanced Course
Advisor: Flavius Gruian

Johan Wennersten (dt08jw2@student.lth.se)
Arvid Lindell (dt08al6@student.lth.se)

Abstract

This report describes relevant technical details in a project for the course EDA385 at Lunds tekniska högskola. The primary purpose of the project was to construct a music video game. The system is built on a Nexys-3 board with a Spartan-6 FPGA. We have written game software and created two custom hardware devices to handle video and sound output.

Contents

1 Introduction

Good introductory picture :) Gives an idea of how close you were to the initial proposal.

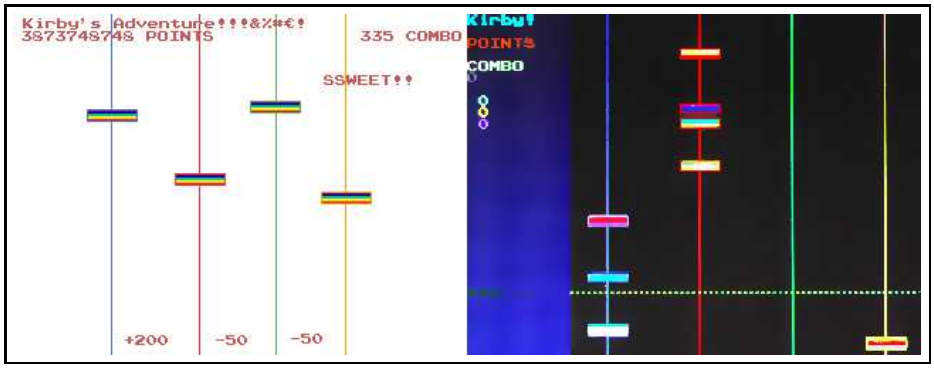

Figure 1: Initial sketch (left) and finished result (right).

We have built a music player with a game similar to Dance Dance Revolution attached, on a Digilent Nexys 3 board with a Spartan-6 FPGA. The initial idea was to combine the timing of the music with a game. Blocks slide from the top of the screen down towards the bottom. The blocks arrive synchronized with the music, a block arrives at the finish line when a note is played. The goal of the game is to press one of four buttons on a custom built input device when a block arrives at the finish line in the corresponding lane.

When you correctly hit a block you get points, and if you hit several in a row the combo increases. A bigger combo amounts to more points per correctly hit block. When you miss a block, you lose the combo.

In figure 1 both the initial design sketch and the end result can be seen. Two video demonstrations can be found at `http://tinyurl.com/midim1` and `http://tinyurl.com/midim5`[1].

Very good idea to put up videos of the demos!

[1] Or `http://www.youtube.com/watch?v=0B3MQla0iic`, `http://www.youtube.com/watch?v=31o48_03lN0`.

This figure should be a bit more explicit, showing the actual interfaces (e.g buses and connections).

1.1 Components

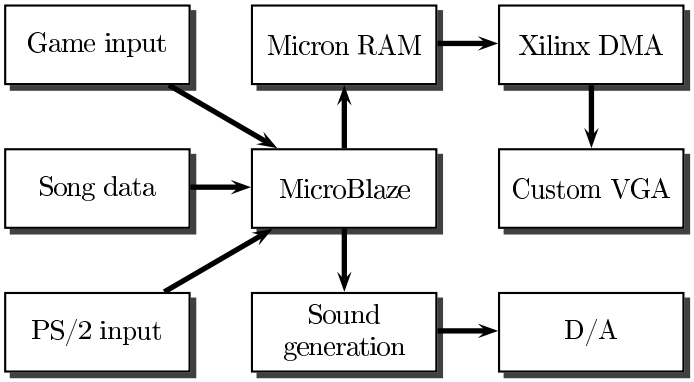

Figure 2: Hardware architecture.

A custom input device (see section 5.1) and two custom IP blocks have been constructed, a VGA controller (using an XPS-generated FIFO) and a sound generator used to generate music from preprocessed MIDI songs. Both controllers are connected to the CPU via the PLB bus. Figure 2 shows the most relevant data transfers in the system. A more complete system overview can be found in appendix B.

Hardware component	IP block
MicroBlaze CPU	microblaze
PCM controller	xps_mch_emc
Micron RAM controller	xps_mch_emc
Sound generation	soundklump*
VGA controller	vga_2*
DMA controller	xps_central_dma
PS/2 controller	xps_ps2
Timer	xps_timer
RS232 UART	xps_uartlite

Table 1: Instantiated components.

Instantiated components are shown in table 1. The BRAM block and controllers, clock generator, PLB etc are excluded from the table. IP blocks tagged with * are built from scratch. The MicroBlaze and PLB are clocked at 83.3 MHz. XPS 13.2 was used.

2 Protocols and methods

2.1 DDS

Avoid empty sections between chapter and section. You can always introduce the next sections in a few words here.

DDS (Direct Digital Synthesis) can in its most basic form be seen as a counter (phase accumulator) and a lookup table, with a clock input and an output. In the following text, the lookup table consists of a single period of a sinusoidal waveform. For simplicity's sake, let's assume that the counter and the LUT are scaled 1:1 (16 bit counter, 65536 values in the LUT).

When the counter is incremented (at the rising edge of the clock), the corresponding value in the lookup table is sent to the output. By incrementing the counter with different values, the output signal can change more (or less) often, and thus generate different frequencies. Of course, Nyquist still applies, so the maximum frequency that can be generated is $DDS_{clk}/2$.

When the increment is set to one, the corresponding generated frequency (base) becomes the DDS clock frequency divided by the number of samples ($S, \geq 2$) in the LUT.

This most basic approach will only allow us to play a limited set of frequencies, namely those that are integer multiples of our base frequency. To improve this we can make the counter use more bits and only use the top bits for LUT lookup. This essentially turns the counting operation into fixed point representation, and it gives us a more fine-grained control over the output frequencies. Instead of only being able to increment by integers we can now increment by multiples of $1/2^n$ (I_{min}) where n is the number of least significant bits ignored in the counter.

The accuracy (smallest frequency step, f_{step}) is given by the following formula.

$$f_{step} = \frac{DDS_{clk}}{S} \cdot I_{min}$$

The increment (I) for a given frequency (f) is given by the following formula.

$$I = \frac{S \cdot f}{DDS_{clk}}$$

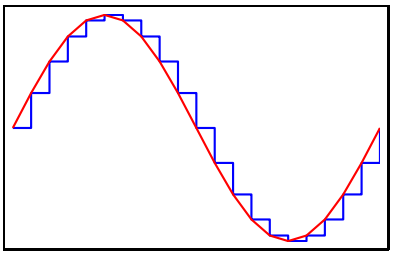

Figure 3: Aliased signal (blue) and interpolated (red).

A problem with DDS is aliasing (see figure 3). This is especially prevalent when the same sample is continuously sent to the output (which is what happens when low frequencies are generated). As seen in figure 3, a staircase pattern is generated in the output signal, and unwanted frequency components will appear. This can be mitigated to some extent by interpolating the missing samples.

2.2 VGA

VGA (Video Graphics Array) is a display standard introduced with the IBM Personal System/2 in 1987. [1]

The basic timing of VGA is provided by a pixel clock, at each rising edge a new pixel is sent to the display, in row order. When the end of a row is reached a horizontal sync signal is sent to the display for a specific duration to indicate that a new row is to be drawn. The original display type used with VGA, CRT displays, needed this time to move their electron beam back to its starting position at the next row. In a similar manner a vertical sync signal is sent once the bottom row is done to allow the beam to move back to the top.

In addition to this, there is an empty area around the entire image which only contains black pixels. This is usually divided into sections called the horizontal and vertical back and front porch. This can be used to center the image on the screen. [2]

2.2.1 Doublescan

The IBM VGA Reference Manual dictates that a compatible monitor should support a horizontal refresh of about 31.5 kHz, a vertical refresh of 50–70 Hz and a pixel clock of 25.175 or 28.332 MHz (25.175 MHz is used for 640x480 at 60Hz). The documentation also specifies a number of standardized resolutions, ranging from 320x200 to 640x480 pixels. [3]

The problem with having somewhat fixed sync signals is that widely varying resolutions (such as 320x200 and 640x480) cannot be realized 'cleanly'. A clean realization of 320x200 would amongst other things require a horizontal sync rate of about 15.75 kHz (at 70 Hz vertical refresh), a rate which according to the IBM Reference Manual need not be supported. The solution is simple, set the actual monitor resolution to 640x400 (which fits cleanly within the monitor's sync range), and draw each pixel and row twice to achieve the desired result. The technique of drawing each row twice is called doublescan. [3]

2.3 PS/2

There are many resources detailing the operation of the IBM PS/2-keyboard protocol, but the gist of it is that each transfer (scancode) consists of one start, stop and parity bit along with 8 data bits. If a scancode is preceded by the value `0xF0` (break code), the key has been released. Otherwise, it has been pressed.

The protocol is bi-directional (host-to-device is amongst other things used to set hardware repeat rate and status LEDs) and synchronous (~16KHz). [4]

2.4 MIDI

MIDI is based around commands. An example command might be "start playing channel 1 with note `0x03` with a velocity (how hard the key is pressed) of `0x40`". The corresponding data sequence for this particular command would be `0x91`, `0x03` and `0x40`. The note parameter corresponds to an offset piano note[2].

[2]See `http://en.wikipedia.org/wiki/Piano_key_frequencies` and `http://www.tonalsoft.com/pub/news/pitch-bend.aspx`. Since a piano only has 88 keys, and the MIDI notes range from 0 to 127, MIDI note 0 actually corresponds to piano key -20, and note 127 corresponds to piano key 107.

A MIDI file consists of a header (containing amongst other things the number of tracks and a base delay unit), and a number of tracks. Each track consists of a number of MIDI commands. The commands embedded in the tracks are separated by a delay parameter (7 to 28 bits). The final delay is determined by the delay parameter in conjunction with the base delay unit specified in the header.

The MIDI file format supports a wide variety of commands and other eccentricities, which makes writing a feature complete parser very time-consuming. [5]

3 Music

The music is composed of square, triangle and sawtooth waves. These are generated using DDS (Direct Digital Synthesis), which is a technique to playback a single sample (of for instance a single period of a waveform) at various speeds to create notes of different frequencies. See section 2.1 for a more detailed explanation.

3.1 Hardware

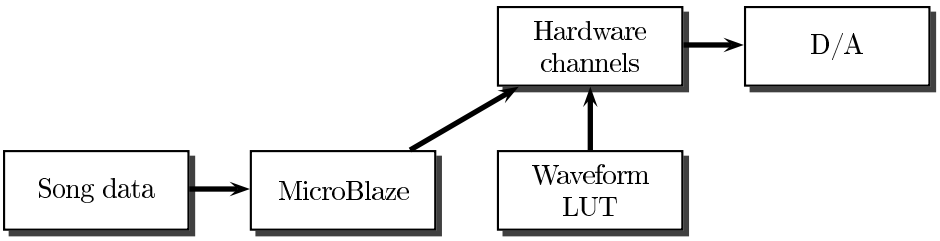

Figure 4: Audio data flow block diagram.

The hardware playback channels (a single IP block) are connected to the CPU using the PLB, and are controlled using software-accessible status registers. The resulting mix (of the 16 channels) is output using the data pins of PMOD A and is converted to an analog signal by a simple R2R-ladder. See figure 4 for the most relevant data flow.

Slices	Slice regs	Slice LUTs	BRAM
890	1497	2039	0

Which is the hardware in Fig 4 you give the utilization for here? It is not clear from this picture (Microblaze is not included, right?)

Table 2: Sound generator device utilization.

Device utilization can be seen in table 2.

3.1.1 Software registers

We use one 32-bit register for each channel, and they are accessed from the CPU using the format specified in figure 5.

```
                                                    lsb
[enable]  [waveform select]  [dds increment]
<------>  <--------------->  <------------->
1 bit     2 bits             16 bits
```

Figure 5: Register write description.

Waveform select specifies the waveform to use (00 = square, 01 = triangle, 10 & 11 = sawtooth), enable is set to 1 if the channel is to be active, and DDS increment sets the active increment for the channel.

3.1.2 DDS specifics

We use a DDS frequency of about 44092 Hz, with 256 samples per waveform, and ignore 8 bits of the counter. This gives us a frequency step of roughly 0.672 Hz. The following formula gives the frequency for the corresponding MIDI note (n). [6]

$$f(n) = (\sqrt[12]{2})^{n-69} \times 440\text{Hz}$$

According to section 2.1, the DDS increment for the specific MIDI note n is thus given by the following formula.

$$I = \frac{S \cdot f(n)}{DDS_{clk}}$$

We precalculate all the necessary increment values (corresponding to MIDI note 0–127) and store them in RAM (data section of the executable).

3.2 Software

See comment above about space between...

3.2.1 Song data

As mentioned in section 2.4 writing a feature-complete MIDI parser would be very time-consuming. In this project we decided to simplify matters by not trying to implement a complete MIDI-parser. What we care about are the events note on, note off, and delays. Using an already existing tool called MIDITONES[3] by Len Shustek we preprocess the MIDI files to keep only those events.

The tool also rearranges the notes to better suit the available hardware, which amongst other things removes the need for logic that determines if a specific tone generator is busy. In short, the MIDITONES data will never try to enable two notes on the same tone generator. The generated data is parsed easily, and an example structure can be seen in figure 6.

The note parameter is identical to the original MIDI specification (see section 2.4) and the delay parameter is specified in milliseconds. See the MIDITONES homepage for a complete specification of the format.

[3] Available at `http://code.google.com/p/miditones`.

```
[note [on | off]] [tone generator] [note] [delay]
<---------------> <--------------> <----> <----->
4 bits            4 bits           8 bits 16 bits
```

Figure 6: MIDITONES data.

MIDITONES will also try to keep notes from a specific MIDI channel to a single tone generator. This is used when extracting note data for the game logic (the lead, which is usually contained in a separate MIDI track, is used to generate note blocks, since it's most easily distinguished). See section 7.1 for information pertaining to note extraction.

3.2.2 Playback routine

The sound processing on the CPU is done in an interrupt routine connected to the built in hardware timer. Whenever a delay event is found a new timer interrupt is set at the corresponding delay and the function returns, and is thus run again when the delay has expired. Each time the function is executed it starts or stops the tone generators dictated by the sound data.

The interrupt routine also synchronizes the music and game logic, which is described in section 7.1.

4 Video

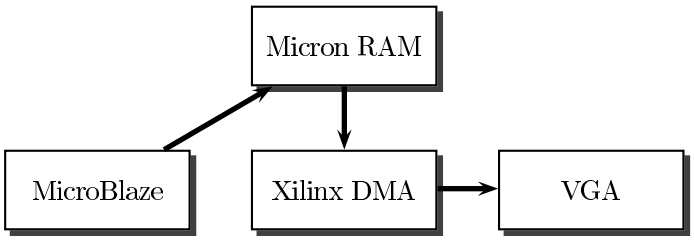

Figure 7: Video data flow block diagram.

The game is visualized using a monitor connected to the onboard VGA connector. We are using a resolution of 320x240 pixels (QVGA) with 8bpp. A custom VGA controller with a 'chaotic' background generator has been constructed. See figure 7 for the most relevant data flow.

Slices	Slice regs	Slice LUTs	BRAM
410	557	630	4

Table 3: VGA controller device utilization.

Device utilization can be seen in table 3.

4.1 Hardware

The entire frame is kept in Micron RAM. Video data is sent from Micron RAM to the VGA controller via the PLB (utilizing a FIFO), in turn using a Xilinx DMA controller.

The basic clock for the VGA controller is the pixel clock at 25 MHz, which gives the rate at which new pixels must be sent to the monitor. Due to limited data throughput from the Micron RAM to the VGA controller we chose to limit the resolution to QVGA. This is half the vertical and half the horizontal resolution of standard VGA. We draw each pixel twice next to each other and we draw each entire row twice aswell. See section 2.2.1 for clarification as to why the quadrupling of the output data needs to be done.

As mentioned before, video data is buffered in a FIFO connected to the VGA controller. Every even row the controller extracts new data from the queue, sends it to the screen, but also saves it in a local row buffer. On odd rows the FIFO is unused and data from the local buffer is used instead.

The video hardware also keeps track of the current horizontal and vertical pixel position, which is used to generate horizontal and vertical synchronization signals, and for algorithm le blar (see section 4.1.2).

4.1.1 Software accessible registers

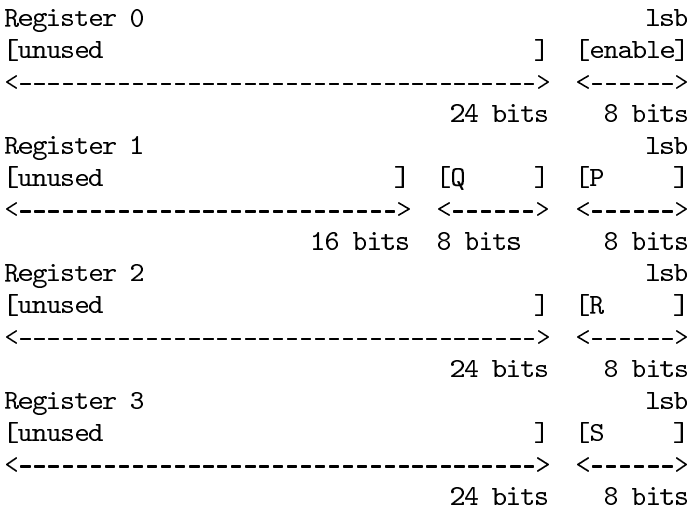

Figure 8: Video hardware register descriptions.

P, Q, R and S operation is described in section 4.1.2, enable is activated when all 8 bits are set (`0xFF`).

4.1.2 Background generator, algorithm le blar

Algorithm le blar is given by the following formula. P, Q, R and S are 8 bit unsigned integers specified by software. It is run at each rising edge of the pixel clock.

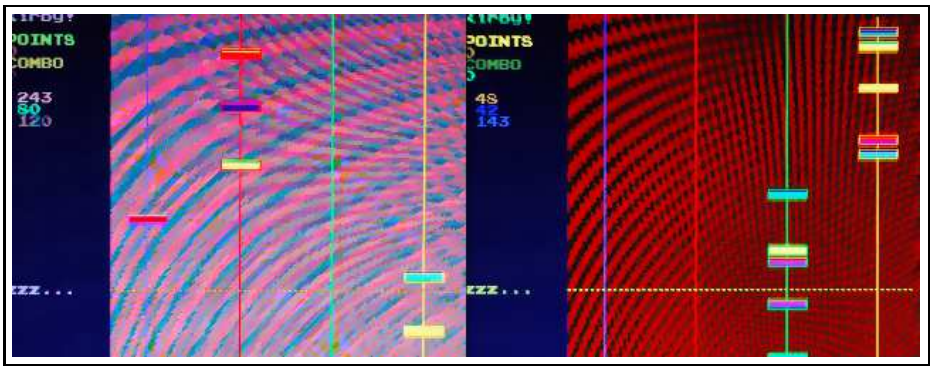

Figure 9: Two different backgrounds generated with le blar.

$$A = (A + R) \mod 256$$
$$B = (B + S) \mod 256$$
$$Pixel = \left(\frac{\left(count_h + \frac{A}{64}\right) \cdot \left(count_v + \frac{B}{16}\right)}{128} \wedge P \right) \oplus Q$$

Multiplying the horizontal and vertical count creates an arc-shaped pattern in the upper bits which translates into the red color. Incrementing A and B at each rising edge of the pixel clock makes the value of $Pixel$ for any given position different from one screen update to the next. Some parameter settings can lead to percieved continuous movement.

Setting P to zero gives the background the value specified in Q, creating an easy way to set single color backgrounds. Whenever the VGA controller finds a black pixel (value `0x00`) it substitutes it with the current background value.

4.2 Software

The `update_vga` function is called each iteration of the main loop in our software application. It is responsible for the dataflow to the VGA controller via the DMA-unit. It keeps track of how much of the video frame that has been sent so far, and if there is sufficient avaliable space in the FIFO connected to the VGA controller, it signals the DMA-unit to send another chunk of data (1280 bytes). Once the first chunk of data has been sent to the FIFO, the VGA device is enabled by sending the value `0xFF` to the first hardware register of the device (as described in section 4.1.1).

When an entire frame is sent and the last chunk is buffered in the VGA FIFO, the video memory is updated with new data as dictated by the game logic. Video data is thus never changed during a screen update, making sure that we get a tear-free, vertically synchronized image.

A very smart solution for skipping the picture stable. Reminds of the old ZX Spectrum and Commodore solutions :)

5 Input

5.1 Custom input device

Figure 10: Custom input device.

The custom input device we constructed is simply a plastic box with four arcade buttons in it (see figure 10). The buttons themselves use microswitches (with pullups, active low) and are connected to four pins of PMOD D.

Since our device utilization was very high (97%, see section 6.2), we did not have space for a separate PLB device to simply poll and report the state of the buttons. To still retain the functionality of the input device, we integrated the polling logic into the VGA controller, and simply read the state from one of its software-accessible registers.

5.2 PS/2-keyboard

A PS/2 keyboard is used to control various settings in the game such as song selection, background alteration and waveform setting. We use the built in XPS PS/2 controller which is polled in the main loop.

Originally, we wrote our own PS/2 controller which was able to keep track of the state of eight buttons. Due to an increased number of user-specifiable parameters (which require external inputs), we wanted to use something more easily modifiable and exchanged it for the standard Xilinx controller, and simply handle the different actions in software.

6 Device specifics

6.1 Memory

- The CPU uses BRAM blocks for instructions, heap, static data and stack, the CPU BRAM is a total of 32KB. The VGA FIFO (2048 bytes) also keeps its data in BRAM.
- The video frame data is placed in Micron RAM and is accessed by the VGA controller using DMA. The video frame data uses $320 \times 240 = 76800$ bytes of RAM.
- The DDS samples are kept in distributed RAM for ease of access. These have a total size of 1536 bytes.

- The parsed MIDI files (see 3.2.1) are kept non-volatile PCM memory. All eleven songs that we currently have take 67366 bytes.

6.2 Utilization

Resource	Used	Out of	%
Number of occupied Slices:	2,212	2,278	97%
Number of Slice LUTs:	6,440	9,112	70%
Number used as logic:	6,046	9,112	66%
Number used as Memory:	271	2,176	12%
Number used exclusively as route-thrus:	123		
Number of Slice Registers:	5,588	18,224	30%
Number used as Flip Flops:	5,583		
Number used as AND/OR logics:	5		
Number of bonded IOBs:	79	232	34%
Number of RAMB16BWERs:	20	32	62%

Table 4: Various utilization statistics.

7 Main application

-O	Text	Data	BSS	Total
3	26282	2800	2062	31144
2	18446	2800	2058	23304
1	18474	2800	2062	23336
0	22910	2812	2058	27780
s	18274	2800	2062	23136

Table 5: Binary sizes (bytes).

The main application consists of the sound-generating timer interrupt routine, and the `update_vga` function specified in section 3.2.1 and 4.2 respectively. In addition to this there is some logic to keep the game and music synchronized, described in section 7.1 below, input handling, as well as some GUI-code to update score and songnames etc.

The resulting sizes of the binary utilizing various degrees of compiler optimization are shown in table 5. Only the binaries compiled with `-O3` and `-Os` have been thoroughly tested. The heap size is 256 bytes, and stack size is 512 bytes.

7.1 Music and game logic synchronization

We want to make sure that the note blocks arrive at the finish line when the corresponding note is played. In addition we want the blocks to slide in from the top of the screen instead of suddenly appearing when they are to be played.

To achieve this we scan the song data a few seconds ahead of where the sound is playing and take note of incoming notes. From doing this we can find out how long it is until a given note is played, and given the vertical speed of blocks calculate a vertical offset, ie "where should this block start to arrive at the finish line in X ms". This offset will place the initial position of a block outside (above) the screen to create the smooth effect of blocks sliding in.

7.2 Main loop pseudocode

```
While forever {
    Send data to the VGA FIFO.
        If a complete frame has been sent {
            Send updated parameters to the background generator.

            Move blocks.

            Draw finish line.

            Draw gui texts and scores.

            Handle input.
        }

    Poll PS/2 input.
    Poll custom input device.
}
```

Instructive with pseudocode. (a flow chart would have worked also)

8 Conclusions and discussion

We are pleased with the end result. It coincided well with the expectations we had in the start of the project. If we were given more time to work on it, we would have written our own MIDI parser (a rudimentary version exists, which unfortunately only plays one song), and constructed more sophisticated sound hardware (channel volume control/drums etc). We would also fine-tune the game logics, and perhaps improve the graphics/background generator a bit.

8.1 Problems and solutions

It was tedious to test and debug interactions between the MB and custom IPs, we partly solved this by not integrating with the MB until we had thoroughly tested the component in a standalone manner. This approach also reduced the penalty of resynthesizing the entire hardware architecture.

We had some problems understanding error messages related to timing problems. Once we realized we were simply trying to do too many things sequentially in one clock cycle it was not very hard to split the actions into multiple cycles.

8.2 Learnings

Our Xilinx tools proficiency has greatly improved. We now better understand the practical applications of the various memory areas on the Nexys 3 (BRAM/PCM/Micron RAM). The Spartan-6 has also been partially demystified, with internal structurings and whatnot. We have also gotten better understanding pertaining to the MIDI and VGA standards.

8.3 Contributions

During most of the project (over 90% of the spent time), we have been working together on the same display, with two mice and two keyboards. It is therefore very difficult to say exactly who contributed to what.

9 References

[1] "Video Graphics Array." http://en.wikipedia.org/wiki/Video_Graphics_Array.

[2] "Nexys 3 Board Reference Manual." http://www.digilentinc.com/Data/Products/NEXYS3/Nexys3_rm.pdf.

[3] "IBM VGA Technical Reference Manual." http://www.mcamafia.de/pdf/ibm_vgaxga_trm2.pdf.

[4] "PS/2 Keyboard Interface." http://www.computer-engineering.org/ps2keyboard.

[5] "MIDI File Format." http://www.sonicspot.com/guide/midifiles.html.

[6] "Piano key frequencies." http://en.wikipedia.org/wiki/Piano_key_frequencies.

A good report, full with details, even if a bit brief in parts. The project in general is very good, with an impressive final result. Well done!

A Installation instructions

These instructions are written for Adept 2.3. Switch to the *Memory* tab and select the file `songs.bin` in the *Write File to Memory* section, leave *BPI Flash* selected and press *Write*. When finished, switch to the *Config* section and program the Nexys 3 with the `midimonster.bit` file.

The 8-bit soundstream is assigned to the data pins of PMOD A. Use a R2R-ladder or similar to convert to an analog signal. The main game input is delivered from four pins of PMOD D. You probably want to use a couple of microswitches to play, the signals are active low. The pins are assigned according to table 6.

Pin	Bit
T12	7
V12	6
N10	5
P11	4
M10	3
N9	2
U11	1
V11	0

Pin	Bit
G11	3
F10	2
F11	1
E11	0

Table 6: Audio out (left) and input (right) pin assignments.

Connect a standard USB-keyboard and use `F1-F11` to switch between the available tracks, `F12` to change sound waveform and `Space` to generate a new random background. `Backspace` generates a new backdrop base-color, `', *` generates a new mask (filters out color components from the final byte representation of the pixel), `Return` and `Right shift` generate new modifiers (which 'animate' the background).

B System overview