

Tetris game with NES gamepad

EDA385 Final Report

Jimmy Assarsson, `dt08ja2@student.lth.se`
Linus Karlsson, `dt08lk9@student.lth.se`
Antoine Morineau, `morineau.antoine@gmail.com`

October 26, 2012

Abstract

This report describes the design of a Tetris-like video game, constructed in both hardware and software on an FPGA development board. Both the hardware and software are described, together with how the different parts communicate with each other. The end result of the project is a game that can be played with a gamepad, on an external VGA-compatible monitor and with background music.

Contents

| | | |
|----------|--|-----------|
| 1 | Introduction | 2 |
| 1.1 | Changes from initial proposal | 3 |
| 2 | Hardware | 3 |
| 2.1 | VGA Controller | 3 |
| 2.2 | FSL Handler | 4 |
| 2.3 | The interface | 4 |
| 2.3.1 | Dynamic interface | 4 |
| 2.3.2 | Decoding the vectors | 4 |
| 2.3.3 | Static interface | 6 |
| 2.3.4 | Multiplexer | 6 |
| 2.4 | Music | 6 |
| 2.5 | Processor, memory and other components | 7 |
| 2.6 | Device utilisation | 7 |
| 3 | Software | 8 |
| 3.1 | Timers | 8 |
| 3.2 | Game logic | 11 |
| 3.2.1 | Moving the block | 11 |
| 3.2.2 | Collision detection | 12 |
| 3.2.3 | Clear lines | 12 |
| 3.2.4 | Communication with the hardware | 12 |
| 3.2.5 | Interrupt service routine | 13 |
| 3.3 | NES gamepad | 13 |
| 3.3.1 | The gamepad | 13 |
| 3.3.2 | Software | 15 |
| 3.4 | Memory usage | 15 |
| 4 | Documentation | 16 |
| 5 | Problems and conclusions | 17 |
| 6 | Lessons learned | 17 |
| 7 | Contributions | 18 |
| A | Evolution of the project | 19 |
| B | FSL protocol | 19 |

1 Introduction

This is the final report of the project in the course *EDA385: Design of Embedded Systems, Advanced Course* at Lund University, Faculty of Engineering. The project has been done in a three person group, over the scope of seven weeks. The goal of this project was to implement a Tetris video game on an FPGA development board. The development board used in this course is *Digilent Nexys 3 Spartan-6 FPGA Board* [1].

Tetris is a puzzle game where the player should rotate and move tiles so that they form complete lines. The player gets points for every line cleared, and the goal is to get as many points possible. As the player clears more lines, the level of the game increases. This increases the speed and makes the game more difficult. The game ends when the player fills the entire height of the game area.

The video game is displayed on a monitor connected to the board with a VGA cable. The game is controlled by a NES gamepad. Score and other statistics are shown on the monitor next to the game area. Furthermore, music is heard as the user plays. The tempo of the music increases as the user reaches higher game levels.

An overview of our hardware architecture is shown in figure 1. We have written two custom hardware components: VGA Controller and Concert. The software is written in C and is running on the MicroBlaze processor. Communication between software and hardware is done using the FSL (Fast Simplex Link). Furthermore, we have a connection between the VGA Controller and the Concert component. This connection makes the music tempo depend on the current level of the game.

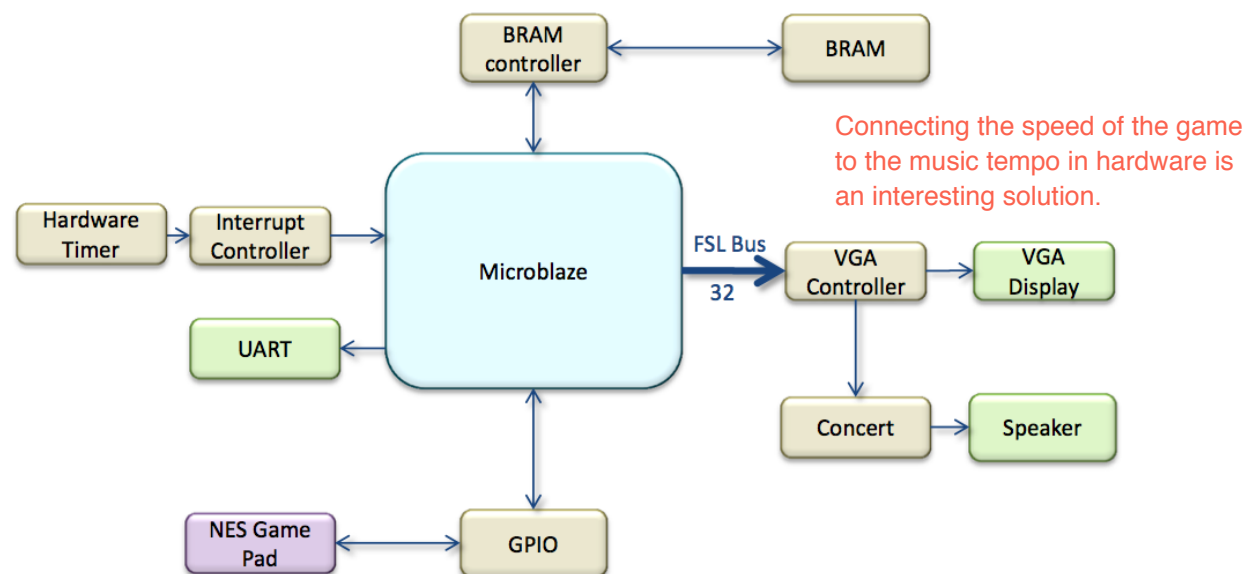


Figure 1: Overview of hardware architecture

This diagram is not very accurate, since most of these peripherals are connected via a PLB bus. This is not obvious here.

1.1 Changes from initial proposal

We have followed our initial project proposal well. The differences between the final project and our proposal is listed below:

- The score is shown on the monitor instead of the 7-segment display.
- Sound is implemented.
- There are different levels implemented in the game.

Very nice that you implemented more than you set out to do.

All of these changes were listed as possible improvements in our project proposal.

2 Hardware

2.1 VGA Controller

In order to display a picture on the screen, different signals must be sent. Monitoring the screen is done with the signals HS and VS. As we want a 60 Hz refresh rate, the timing of both those signals is shown in figure 2.

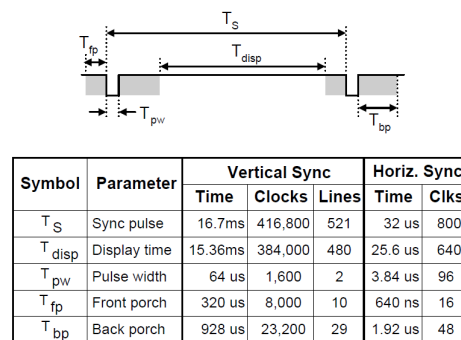


Figure 2: Timing to monitor the screen [5, p. 17] Good you give a reference!

The architecture implemented to create those signals is showed in figure 3 .

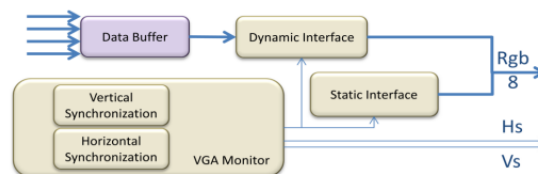


Figure 3: VGA controller architecture

The components: *Horizontal synchronization* and *Vertical synchronization* are two state machines. Each state displays one time, according to the table in figure 2. Thanks

to two counters, we know the current position of the pixel on the screen. Finally, as the pixels must be switched at the frequency of 25 MHz, we have implemented a new clock which gives the correct frequency. The occupancy of this part is given in table 1.

2.2 FSL Handler

The FSL handler is the link between the software and the hardware parts. This component reads and stores the words in registers. The words are updated each time the game state changes. We store 23 words. The 20 first words are using to display the game area and the three others to give other information. Indeed, there are only 20 lines in the game area. So we have one word for each line of squares.

Moreover, the software part and the hardware part do not use the same clock; they are asynchronous. That is the reason why we just read the vectors which contain useful information for the line displayed. The others are just stored and can be updated by the software whenever it wants. We use two components, `fsl_handler`, which reads and stores the vectors and `data_buffer`, which is a state machine. At each state, we read the current vectors and send it to the dynamic interface. Then, they are decoded and displayed on the screen, as it is explained in the section below. The size of this part is given in the table 1. We can see that lots of registers are used to store our vectors. Indeed, $23 \text{ words} * 32 \text{ bits} = 736$, which explains the number obtained.

More information regarding the FSL protocol can be found in appendix B.

2.3 The interface

The Tetris interface is created with blocks. For the whole interface, we use blocks of 9 different colours. Still, for the game, only 7 colours are required. In order to stay in the mood of Tetris, everything is written with blocks. We use two different sizes in order to be able to display everything. The top line of the title does not separate the letters. It is a wink of the goal of the Tetris game, making lines. The flags are here to show to the player that this project was realized by both Swedish and French students. At the bottom right hand corner, the three letters are J L A for Jimmy, Linus and Antoine. The information is displayed on the left of the screen in light colours to be seen in a flash.

2.3.1 Dynamic interface

The dynamic interface is designed to display all of the information given by the software, such as the scores, the number of lines, the current level, the pause sign and, last but not least, the game. In order to achieve this goal, we use seven different components. Indeed, this part must decode the vectors and display the squares at the right position.

2.3.2 Decoding the vectors

The FSL bus sends 23 words of 32 bits each. The twenty first words are used to display the game. They are subdivided into ten vectors of three bits each. This way, we know

the position and the colour of the squares. An example is showed in figure 4.

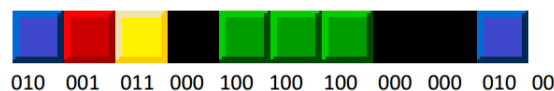


Figure 4: Example of a line in the game

Once the vector is decoded, we still have to create the square. Indeed, in our interface, all of the blocks are not made by creating a simple square. We made it with a 3D perspective. In order to do that, we use three different colours for each blocks, as it shows in figure 5.



Figure 5: Creation of a block

The component *Choice colour* is the keystone of this realisation. Indeed, it returns the three useful colours when we ask for one block. Then we display the three colours correctly using combinational logic. We choose the colours using one of our file, `test_colour`. This file display all of the colours available from the FPGA on the screen, as it is shown in figure 6.



Figure 6: Colours available with the FPGA

Displaying the game area is quite easy because one vector represents one line of blocks. Things become trickier when the vectors do not give the position. It is the case for all of the other information. For instance, the score is coded in BCD (Binary Coded Decimal). So we easily know the figure we need display but not the position. As the position is fixed, we use another component, *score controller* which gives the y position of the top of the square. This value is fixed during all of the time of one line of squares is displayed and changes at the beginning of the second line. Thus, we have got a mark

and using this reference to display the three colours of the block, we do not have to write the conditions for all of the lines, but only for one. Then, we create the blocks. Finally, we choose which colour we want to display. Or the current colour, or black if there is no block. An example is done in figure 7(a) and 7(b).

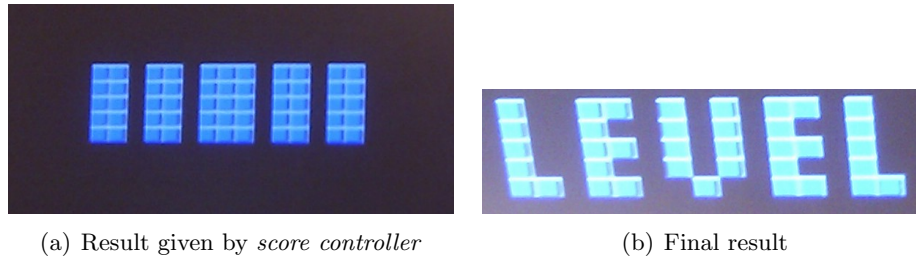


Figure 7: Building the level text

The size of the dynamic interface is given in the table 1.

2.3.3 Static interface

The static interface uses the same principle as the dynamic interface to create and to display the blocks. The difference is that we know the position, the colour and what we want to display. This interface displays the title, the flags, the signature and the game edge. The size of the static interface is given in the table 1. If we compare it with the size of the dynamic interface, we see that we use less slices and less registers. Static interface deals only with three components against seven for the dynamic one. Moreover, we do not need to store any vectors, we just use constants to display what we want.

It is not very clear whether you use a memory to store the current game status, or you need to send this continuously, for every VGA frame... please explain.

2.3.4 Multiplexer

Static and dynamic interface are linked using an OR gate. Indeed, we know that the positions for the static and dynamic blocks are never the same. So we do not have any conflict. Before sending the information to the screen, we have put a multiplexer. It is a security, if something unexpected happens. This multiplexer allows to display data only when the screen is ready for. During timings to come back at the beginning of a row or at the beginning of the screen, the multiplexer output is forced at '0'.

2.4 Music

In order to be as close as possible of the real Tetris game, we wanted to display the music. This music is also implemented in hardware. The sound is created modulating the frequency of the output signal. As we send digital signal or square signals, the sound is near to the 8-bit music. We have subdivided the partition in three different parts which are displayed in loop. Each part is a state machine and each state gives a new number. This number is using in a counter, to be able to change the frequency of

| Component | Slices | | Slice registers | |
|-------------------------|-------------|--------------|-----------------|--------------|
| | Count | Percentage | Count | Percentage |
| VGA controller | 1086 | 43.3 % | 979 | 34.9 % |
| Counters | 111 | 4.4 % | 100 | 3.6 % |
| FSL handler | 330 | 13.1 % | 738 | 26.3 % |
| Dynamic interface | 524 | 20.9 % | 126 | 4.5 % |
| Static interface | 121 | 4.8 % | 15 | 0.5 % |
| Concert | 100 | 4.0 % | 98 | 3.5 % |
| MicroBlaze | 732 | 29.3 % | 812 | 29.0 % |
| Timer | 146 | 5.8 % | 301 | 10.7 % |
| <i>Other components</i> | 437 | 17.5 % | 614 | 21.9 % |
| Total | 2501 | 100 % | 2804 | 100 % |

Table 1: Device utilisation

the beginning we thought about another way. We wanted to use specialized but reusable component. For example, one component to create a square, one component to create a zero, which would use the component square, one component to display all the figures which would use the component zero, one, which use themselves the component square. But, using this way, we saw that we used 600 % of the size. So we changed our mind. With more time, we would use memories.

These numbers are collected from the *Module Level Utilization* report in XPS. However, when we instead look at the *Design summary* we see that the total utilisation is 1684 slices (73 %). This is significantly lower than the total of 2501 seen in table 1. We believe this may because of optimizations done between different components, perhaps there are slices that can be merged or shared. In that way the total number of slices actually used may be lower than the sum of all required slices for each component.

3 Software

All software was written in C. A simplified flowchart describing the game logic can be found in figure 10.

3.1 Timers

We have used two timers in our software. We have used the `xps_timer` IP from Xilinx [2], which gives us two timers in a single component. One timer polls the NES controller, and the other moves the current block down. When a timer expires, it causes an interrupt in the MicroBlaze. The interrupt handling is then done in two steps. First an interrupt handler is called which clears and acknowledges the interrupt. This is a slightly adapted version of the one that is included with `xps_timer` IP component. This interrupt handler

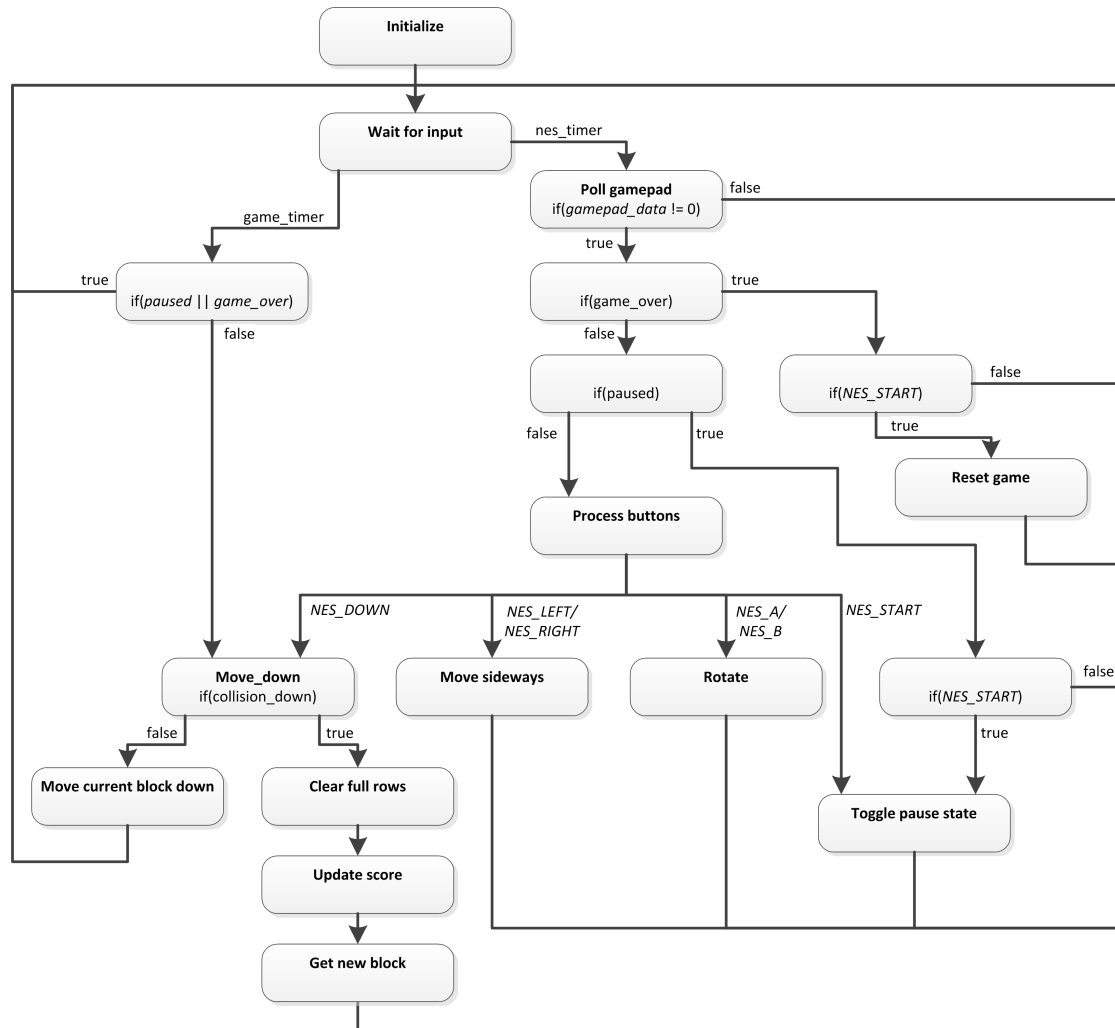


Figure 10: Simplified flowchart of the software

then calls the interrupt handler which actually contains the tasks that should be done in the interrupt.

The interrupt handler then executes the correct code. This makes the game completely interrupt driven, since the `main()` function is used only for initialization. The code used for initialization of the interrupt controller (the first step above) can be seen in the code listing below. We initialize the interrupt controller with a function (`customXTmrCtr_InterruptHandler`) that checks which of the timers that expired, and clears the interrupt bit etc.

```

/* Register the interrupt handler in the vector table */
// Initialize interrupt controller
XIntc_Initialize(&InterruptController, INTC_DEVICE_ID);

// Connect interrupt source to interrupt handler
// function and start controller
XIntc_Connect(&InterruptController, TMRCTR_INTERRUPT_ID,
              (XInterruptHandler) custom_XTmrCtr_InterruptHandler,
              (void*) &TimerCounterInst);
XIntc_Start(&InterruptController, XIN_REAL_MODE);

// Enable interrupt XPAR_XPS_INTC_0_DEVICE_ID
XIntc_Enable(&InterruptController, TMRCTR_INTERRUPT_ID);

// Register the interrupt controller handler with the exception table.
Xil_ExceptionInit();

Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,
                             (Xil_ExceptionHandler)
                             INTC_HANDLER,
                             &InterruptController);
// Enable non-critical exceptions.
Xil_ExceptionEnable();

```

The second step of the interrupt is the initialization of the timers. At first we set the "real" interrupt handler that will actually execute our code for the gamepad polling and game logic.

```
XTmrCtr_SetHandler(&TimerCounterInst, timer_isr, &TimerCounterInst);
```

We then set options on both timers such that they cause interrupts and so that they repeat once they expire:

```

XTmrCtr_SetOptions(&TimerCounterInst, TIMER_CNTR_NES,
                   XTC_INT_MODE_OPTION | XTC_AUTO_RELOAD_OPTION);
XTmrCtr_SetOptions(&TimerCounterInst, TIMER_CNTR_GAME,
                   XTC_INT_MODE_OPTION | XTC_AUTO_RELOAD_OPTION);

```

We then continue by setting the reset value of both timers, determining the period time of the interrupts.

```

XTmrCtr_SetResetValue(&TimerCounterInst, TIMER_CNTR_GAME,
                      RESET_VALUE_GAME);
XTmrCtr_SetResetValue(&TimerCounterInst, TIMER_CNTR_NES,
                      RESET_VALUE_NES);

```

`TIMER_CNTR_GAME` and `TIMER_CNTR_NES` are constants set to `0xFFECCCCC` and `0xFD000000`. These were calculated by first measuring how many cycles we got each second, it turned out to be $T = 75497472$. Since we wanted a frequency of 60 Hz for the polling of the gamepad, we divided this number by 60 to get the period time $T_{NES} = 1258291$. The timers cause an interrupt when they wrap around, i.e. when they reach the value zero. Thus, we must set the timer's reset value to $FFFFFFFF_{16} - 1258291_{10} = FFECCCCC_{16}$.

Finally both timers are started with:

```
XTmrCtr_Start(&TimerCounterInst, TIMER_CNTR_GAME);
XTmrCtr_Start(&TimerCounterInst, TIMER_CNTR_NES);
```

3.2 Game logic

The game area is 20 rows times 10 columns. In the implementation we use a 22×12 `unsigned char` array for representing the game area, where the outermost elements represents the border. The array is used for keeping track of which positions that are occupied by blocks. The value 0 is used for representing an empty position and 1 – 8 are used for representing the colour of the square that occupies the position. We refer to this array as the `fixed_game_area`, since the player can not move these blocks. We use the same approach for keeping track of the `next_block`, with a 2×4 `unsigned char` array instead.

A vector of four `point_t` elements is used for representing the movable block, or as we call it the `current_block`. `point_t` is a struct which contains a x and y position and the colour of the block.

We use another 22×12 `unsigned char` array to make it easier to write the data on the FSL bus. This array is a copy of the `fixed_game_area` merged with the `current_block` and we refer to it as the `combined_game_area`.

We use some other variables for representing the game state; `score`, `lines`, `game_over` and `paused`. The levels are in the range 0 – 10, you reach a new level for every tenth of cleared line.

3.2.1 Moving the block

There are three different main options for moving a block:

- Move down
- Move sideways
- Rotate

When the block is moved sideways we only have to make sure that no collision will occur, if it does not the current blocks x position is updated with the new position.

A block can be rotated in a clockwise or counterclockwise direction. Since some of the shapes does not have a strict (an integer value) center of rotation, we have to approximate a center of rotation. The center of rotation is rounded towards the down

right when the block is rotated in a clockwise direction, and towards down left for counterclockwise rotations.

If a collision will occur when a block is moved down, the `current_block` is merged with the `fixed_game_area`, all full lines are cleared (see section 3.2.3 for more details), the `next_block` becomes the `current_block` and a new `next_block` is generated. Otherwise the y position of the block is updated.

3.2.2 Collision detection

An element with a nonzero value represents an occupied position in the `fixed_game_area`. When collision detection is performed the new desired position of `current_block` is compared with the `fixed_game_area`. A collision is detected if there is any occupied position in `fixed_game_area` that overlaps with a desired position of `current_block`.

When the game is initialized the outermost elements of the `fixed_game_area` are set to the colour of the border and the rest of the elements are set to zero. This makes collision detection much easier, since the border will be treated like any other block and no custom collision detection is required.

3.2.3 Clear lines

When clearing lines, each line in `fixed_game_area` is checked from the bottom to top. If a complete line is found, everything above that line is moved down one step and thereby removes the line. The player get different scores, based on how many lines that were cleared. The scoring is according to:

- 1 line will add 1 to score.
- 2 lines will add 2 to score.
- 3 lines will add 4 to score.
- 4 lines will add 8 to score.

Clear lines is always performed whenever a collision downwards occurs.

3.2.4 Communication with the hardware OK!

All game data is sent on the FSL bus whenever the game changes its state. The data is sent in words of 32-bits. For the complete game we use 23 words.

The first 20 words are the lines of the `combined_game_area`, where the most significant bit (MSB), corresponds to the colour of the leftmost element. The next word contains the `score` (in 4-bit BCD format), `level` and the current state of `paused`. The succeeding word contains the colours of `next_block`. The final word contains the number of cleared `lines`. More details regarding the FSL data is shown in appendix B.

3.2.5 Interrupt service routine

The interrupt service routine (ISR) is executed whenever one of the hardware timers overflows (see section 3.1 for more details).

NES timer

Could you have your own NES hw interface generate interrupts whenever a button is pushed instead of polling it on timer?

When the NES timer interrupt is raised, the gamepad is polled. If no button is pushed nothing happens, otherwise each button is handled according to figure 10.

If a button is held down, the same function will be executed again. Since the gamepad is polled very frequently, a "single push" were most times treated as a hold. To avoid this problem we added a counter which counts for how many interrupts the same button combination has been held. If a combination is held for a certain number of interrupts, the same function will be executed once again.

Game timer

The `current_block` is moved down (see section 3.2.1 for more details) whenever the game timer interrupt is raised. The period time of the game timer is decreased when a new level is reached, by increasing its reset value.

3.3 NES gamepad

As input device for the game, we have used a NES (Nintendo Entertainment System) gamepad. A picture of the gamepad can be seen in figure 11. As can be seen on the picture, the NES gamepad has eight buttons: up, down, left, right, start, select, A and B.

3.3.1 The gamepad

This gamepad is a serial device, and needs to be polled for data. Reading the description of the protocol from [3], we can make a timing diagram as in figure 12. The protocol can be described like this:

1. Send a pulse on the **Latch** pin. This makes the gamepad store all buttons currently pressed.
2. Read the state of the A button by reading the **Data** pin.
3. Send a pulse on the **Pulse** pin to make the next button available on **Data**.
4. Repeat until all buttons have been read.



Figure 11: NES gamepad

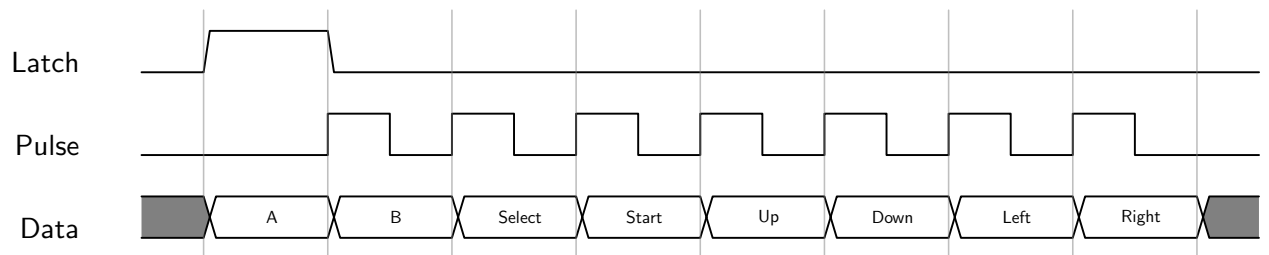


Figure 12: Reading NES gamepad

3.3.2 Software

Since there is no need for a specific duration on either the latch or the pulse signal, it is easy to implement the polling in software. We have used two `xps_gpio` IPs from Xilinx [4] to interface our gamepad. One component is used for input and the other for output. The components are connected to the PLB bus, and can be accessed from C code by using pre-written functions. The NES gamepad itself has been connected to one of the boards PMod ports, PModA. By adding the following lines to our `.ucf` file, we could connect the external pins to our GPIO components:

```
Net fpga_0_pmod_ja_1_o_pin<0> LOC = T12 | IOSTANDARD=LVC MOS33; # JA1
Net fpga_0_pmod_ja_1_o_pin<1> LOC = V12 | IOSTANDARD=LVC MOS33; # JA2
Net fpga_0_pmod_ja_3_i_pin LOC = N10 | IOSTANDARD=LVC MOS33; # JA3
```

The location of the pins were located in the Nexys 3 reference manual [5, p. 21].

To use the GPIO components from the code, we need to initialize them with the following code:

```
XGpio_Initialize(&nes_output, XPAR_PMOD_JA_OUTPUT_DEVICE_ID);
XGpio_Initialize(&nes_input, XPAR_PMOD_JA_INPUT_DEVICE_ID);

/* Set data direction. */
XGpio_SetDataDirection(&nes_input, 1, 0xffffffff); /* All inputs. */
XGpio_SetDataDirection(&nes_output, 1, 0x0); /* All outputs. */
```

After this we can use the GPIO components simply by setting a value to 1, wait for some time, and then setting the value back to zero again. For example, to write to the **Latch** pin, we use the following code:

```
XGpio_DiscreteWrite(&nes_output, 1, 0x1); /* Write to latch */
for (delay = 0; delay < 30; delay++); /* Wait. */
XGpio_DiscreteClear(&nes_output, 1, 0x1); /* Restore to zero. */
```

The empty for loop uses a variable declared as `volatile int delay`; to avoid compiler optimizations and create a delay. Since the gamepad does not require the signal to be high for a specific time, we have chosen 30 somewhat arbitrarily. However, we want this to be a small number since this interrupt occurs 60 times per second. To read a bit from the **Data** pin, we use the following line:

```
u8 temp = XGpio_DiscreteRead(&nes_input, 1) ^ 0x1;
```

This reads from the input GPIO component. The final XOR is because the **Data** is active-low.

3.4 Memory usage

When creating our project in Xilinx Platform Studio, we chose to use 32 kB of BRAM. In the Xilinx SDK we used the default values for stack and heap size, which is 1 kB for each. Our final memory usage is printed in table 2. As can be seen from the table, we use quite a lot of our available memory, 27.4 kB which is 86 % of the available memory.

| text | data | bss | stack | heap | total |
|-------|------|------|-------|------|-------|
| 21778 | 1428 | 2812 | 1024 | 1024 | 28066 |

Table 2: Memory usage for the software

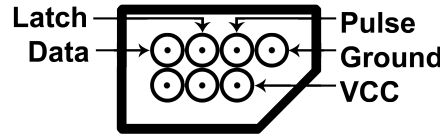


Figure 13: Pinout of the NES gamepad

At the end of the project, as we added more and more code to the software, we got issues with the memory usage. The compiler gave errors which said that it could not fit text, data, bss, stack and heap in the memory at the same time.

Instead of increasing the available memory by using more BRAM, we changed the compiler settings. By using the *Optimize for size (-Os)* flag, we could reduce the memory required for the instructions such that everything fitted inside our memory.

4 Documentation

To test the project you need access to an archived file of our project, available on the course web page (a link here would be appropriate). Furthermore, you need access to a Digilent Nexys 3 board, a VGA compatible monitor, a NES gamepad and finally a PModAMP1 if you want to hear the music.

First we need to connect all peripherals to the Nexys 3 board:

1. Connect the VGA monitor to the Nexys board with a VGA cable.
2. Connect the Nexys 3 board to the computer with a USB-cable for programming the board.
3. Connect the NES gamepad to the PModA port as follows: **Clock** → JA1, **Latch** → JA2 and **Data** → JA3. Also connect the **Ground** and **VCC** pins to appropriate outputs on the board. See the pinout of the gamepad in figure 13.
4. Connect the PModAMP1 to the upper part of the PModB port.

When everything is connected the board must be programmed by opening the project files in the `tetris/` directory in Xilinx Platform Studio. The project is created with version 14.2 of the software. Synthesize the project and start Xilinx SDK by pressing the Export to SDK button. When Xilinx SDK starts, choose `tetris/workspace/` as the workspace folder. The final step is to press the Program FPGA button in to program the board.

5 Problems and conclusions

As with all projects we had our fair share of problems. A constant source of annoyance was the Xilinx tools, especially the time required for a full synthesis. Much of the time spent on the project has been time waiting for XPS to synthesize the project. The workflow was usually to make some minor change in the hardware, re-import and reconnect the component in XPS and then make a synthesis of roughly ten minutes.

After some time we realized that we could get around this problem by using the ISE Design tools instead, it works well when debugging for example graphical issues that is static and does not require the software part. By using ISE we could reduce the time of synthesis to a more reasonable time of around two minutes. However, when developing the dynamic parts of the game, and testing the communication between hardware and software, we found no other option than to use XPS.

As we described earlier in section 2.6 we had to redesign the way we drew blocks on the screen. With the initial design we had a utilisation of over 600 % for the VGA controller alone.

The timers caused a great deal of trouble before we got them to work well. It was very hard to find good documentation on how to use the timers with interrupts. After a lot of hours with testing we finally managed to get the interrupts to fire and to run some code in the interrupt routine. When we got this working solution we avoided touching the interrupt handling code as much as possible.

After getting the timers to work we noticed some bugs that only occurred after the program had been running for a while, and mostly when buttons were pressed on the gamepad. It turned out to be a problem when the interrupt handler was running while another interrupt occurred. When this happened, the first interrupt was never acknowledged, and when the first interrupt routine finished, no more interrupts could be handled. This problem was fixed by always clearing the interrupt bit for the timer with the highest frequency, i.e. the gamepad polling timer.

If we would have made the project again, we would have changed the interrupt routine such that it only sets flags. These flags would then be checked by an infinite loop in the `main()` function. That way we do not get the problem with an interrupt routine that takes a long time to execute.

yes, good conclusion

6 Lessons learned

We have learnt a lot during this project. We have gained more experience with the different Xilinx tools. One example is that ISE Design tools can be used for only synthesizing our custom hardware parts, which saved us a lot of time when debugging them. We have also learnt how to implement and import custom IPs.

A lot of problems can be avoided by discussing problems and solutions, both within the team and outside it. Development becomes easier with proper documentation. It is both easier and faster to develop and debug software than hardware. Debugging a system of both hardware and software is very hard. We have experienced the importance

of dividing problems into smaller parts when debugging them.

7 Contributions

For this project, the work has been subdivided into three parts since the beginning. This way, we worked in parallel and we gained in efficiency. Antoine worked on the hardware part. He designed the interface and implemented it in hardware. The software has been coded by both Linus and Jimmy. Jimmy worked on the game logic. Linus worked on the NES Gamepad controller. They implemented together the FSL bus protocol. Linus created the components in the platform and linked them on the board in order to make tests. The music and the FSL hardware part were designed by the three of us.

All of the reports and the presentation were made, or prepared, by all of the group. Everyone were specialized on the part he worked on.

- Antoine has written sections: 2, 7, A.
- Jimmy has written sections: 3.2, 6, B.
- Linus has written sections: 1, 3.1, 3.3-3.4, 4, 5.

References

- [1] Digilent, *NexysTM3 Spartan-6 FPGA Board* <http://www.digilentinc.com/Products/Detail.cfm?NavPath=2,400,897&Prod=NEXYS3>
- [2] Xilinx, *LogiCORE IP XPS Timer/Counter (v1.02a)*, http://www.xilinx.com/support/documentation/ip_documentation/xps_timer.pdf
- [3] *The NES Controller Handler*, <http://www.mit.edu/~tarvizo/nex-controller.html>
- [4] Xilinx, *XPS General Purpose Input/Output (GPIO) (v2.00a)* http://www.xilinx.com/support/documentation/ip_documentation/xps_gpio.pdf
- [5] Digilent, *Nexys3TM Board Reference Manual*, revision: December 28, 2011 http://www.digilentinc.com/Data/Products/NEXYS3/Nexys3_rm.pdf
- [6] Xilinx, *MicroBlaze Soft Processor*, <http://www.xilinx.com/tools/microblaze.htm>
- [7] Flavius Gruian. GCD: Hw - A Hardware Solution (laboratory session 4, eda380). <http://fileadmin.cs.lth.se/cs/Education/EDAN15/labs/lab4/EDA380Lab4.pdf>, March 4 2009. Accessed October 25, 2012.

A Evolution of the project

During the project, we made some photos to see the evolution of the project. The first results was the VGA controller showing a static image, and the software printing to the UART, as can be seen in figure 14(a) and 14(b).

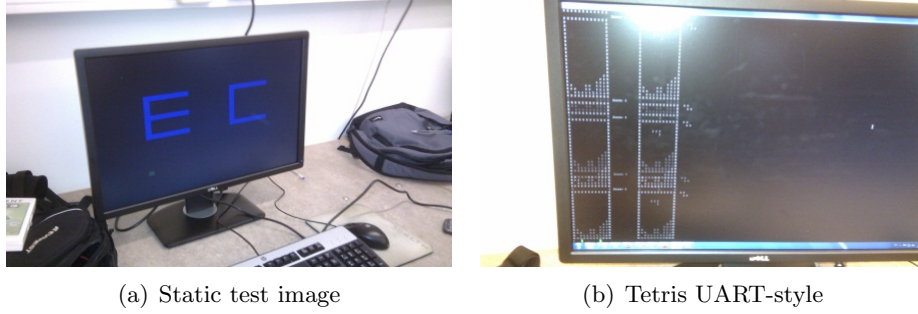


Figure 14: First stage of the project

As we can see in the game area, we use different symbols to differentiate the block types. Later, each symbol will be switched into a color. After those first results, we chose the three different colors for each block and linked the hardware with the software. We also improved the logic of the game and implemented the gamepad handler. This gave us some new results as can be seen in figure 15(a) and 15(b).

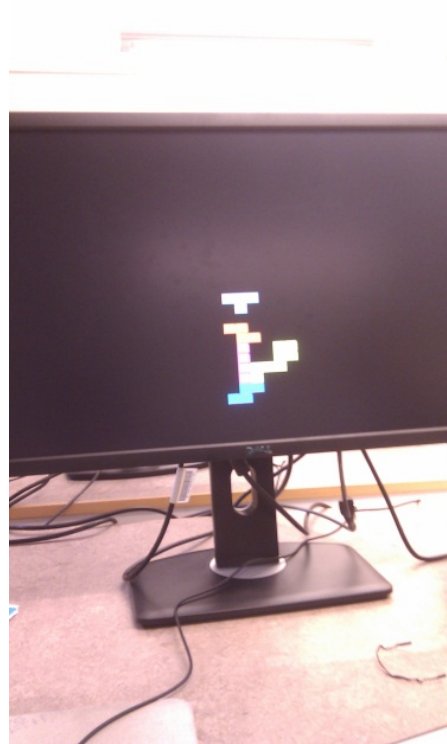
At the end, we added the interface, finished the game logic, implemented the music, added levels and a pause sign. The final result can be seen in figure 16.

B FSL protocol

The arrangement of the data sent on the FSL bus is shown in figure 17. The FSL handler reads the data according to the timing diagram in figure 18.



(a) Colours available on the FPGA



(b) Dynamic blocks

Figure 15: Second stage of the project



Figure 16: Final version of the game

| 32-bit FSL bus | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|----------------|--------------------|----|----|----|----------------|----|----|----|---------------|----|----|----|-------------|----|----|----|-------------|----|----|----|-------------|----|---|---|-------------|---|---|---|-------------|---|---|---|-------------|--|--|--|-------------|--|--|--|
| | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | | | | | | | | |
| D0 | Colour 0.0 | | | | Colour 0.1 | | | | Colour 0.2 | | | | Colour 0.3 | | | | Colour 0.4 | | | | Colour 0.5 | | | | Colour 0.6 | | | | Colour 0.7 | | | | Colour 0.8 | | | | Colour 0.9 | | | |
| D1 | Colour 1.0 | | | | Colour 1.1 | | | | Colour 1.2 | | | | Colour 1.3 | | | | Colour 1.4 | | | | Colour 1.5 | | | | Colour 1.6 | | | | Colour 1.7 | | | | Colour 1.8 | | | | Colour 1.9 | | | |
| D2-D17 | ⋮ | | | | ⋮ | | | | ⋮ | | | | ⋮ | | | | ⋮ | | | | ⋮ | | | | ⋮ | | | | ⋮ | | | | ⋮ | | | | | | | |
| D18 | Colour 18.0 | | | | Colour 18.1 | | | | Colour 18.2 | | | | Colour 18.3 | | | | Colour 18.4 | | | | Colour 18.5 | | | | Colour 18.6 | | | | Colour 18.7 | | | | Colour 18.8 | | | | Colour 18.9 | | | |
| D19 | Colour 19.0 | | | | Colour 19.1 | | | | Colour 19.2 | | | | Colour 19.3 | | | | Colour 19.4 | | | | Colour 19.5 | | | | Colour 19.6 | | | | Colour 19.7 | | | | Colour 19.8 | | | | Colour 19.9 | | | |
| D20 | Score ten thousand | | | | Score thousand | | | | Score hundred | | | | Score ten | | | | Score unit | | | | Level | | | | Pause | | | | | | | | | | | | | | | |
| D21 | Colour 0.0 | | | | Colour 0.1 | | | | Colour 0.2 | | | | Colour 0.3 | | | | Colour 0.4 | | | | Colour 0.5 | | | | Colour 0.6 | | | | Colour 0.7 | | | | Colour 0.8 | | | | Colour 0.9 | | | |
| D22 | Lines ten thousand | | | | Lines thousand | | | | Lines hundred | | | | Lines ten | | | | Lines unit | | | | | | | | | | | | | | | | | | | | | | | |

Figure 17: Data sent on the FSL bus

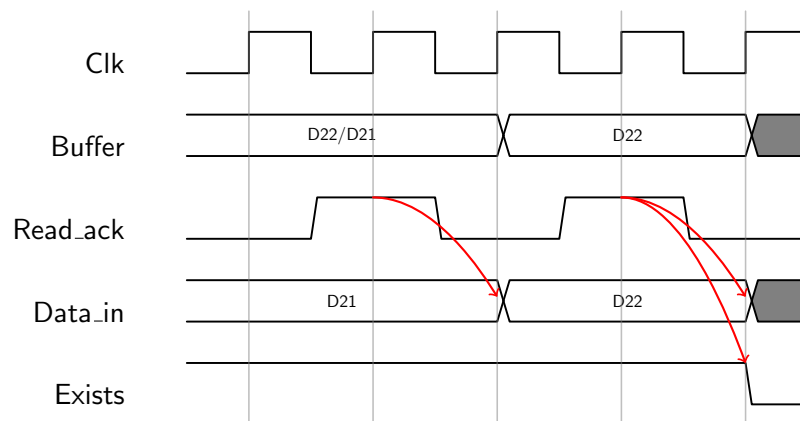


Figure 18: Reading FSL data. Once **Data.in** is read, **Read_ack** must be driven high for one clock cycle. This informs the buffer that the data has been read, causing new data to be shift in on the next rising edge. This may occur only if **Exists** is high. Note that reading data requires at least two clock cycles - one to store and acknowledge the read and one to allow the buffer to fetch new data.[7]

Overall very good report, with enough level of detail and well written!