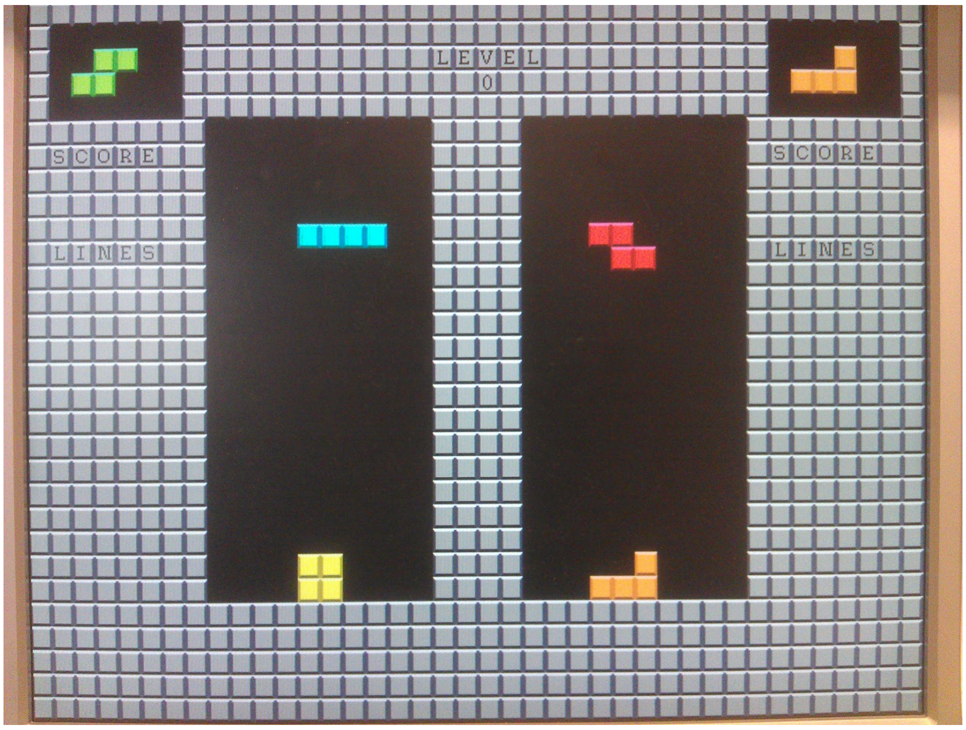
Computer Science
Faculty of Engineering
Lund University

Project Report
Two Player Tetris

Alexander Aulin, E07 (et07aa0)
Niklas Claesson, E07 (et07nc7)



Design of Embedded Systems
Advanced Course (EDA385)
October 26, 2012, Lund

Contents

usually a system (embedded) = hw + sw

Abstract

A game which is similar to tetris was implemented on an embedded system. The system contained a graphic accelerator which was written in VHDL and a software part written in C. Using the keyboard as input, two players can play head-to-head until one player looses.

The hardware block was intentionally written very generic which made it larger than necessary. This makes it possible to use the same accelerator in other game implementations. The conclusion was drawn that the hardware could be a lot smaller.

Using the software to load the memories at startup also doubled the memory requirements.

All in all, everything worked as expected but a lot of design improvements is mentioned in the discussion.

improvements are
(improvement is)

1 Introduction

The purpose of this project was to combine the knowledge from several courses, for instance "Design of Embedded Systems", and create a complete system using part hardware accelerated and part software. Two player tetris was chosen because it required inputs and outputs, could use a graphic accelerator and required some game logic.

The final architecture can be seen in figure 1. The blocks that are highlighted with a wider border was implemented, the other were IP cores provided from Xilinx or Digilent. The keyboard block is an PS2 interface which was implemented as an AXI peripheral. Since it was difficult to write testbenches for the AXI bus it was decided that the graphic accelerator shouldn't be an AXI bus peripheral. Instead the accelerators memories were accessed using an *AXI BRAM Controller* IP core.

blocks … were
(block … was)

All the game logic was written in C and executed on the MicroBlaze.

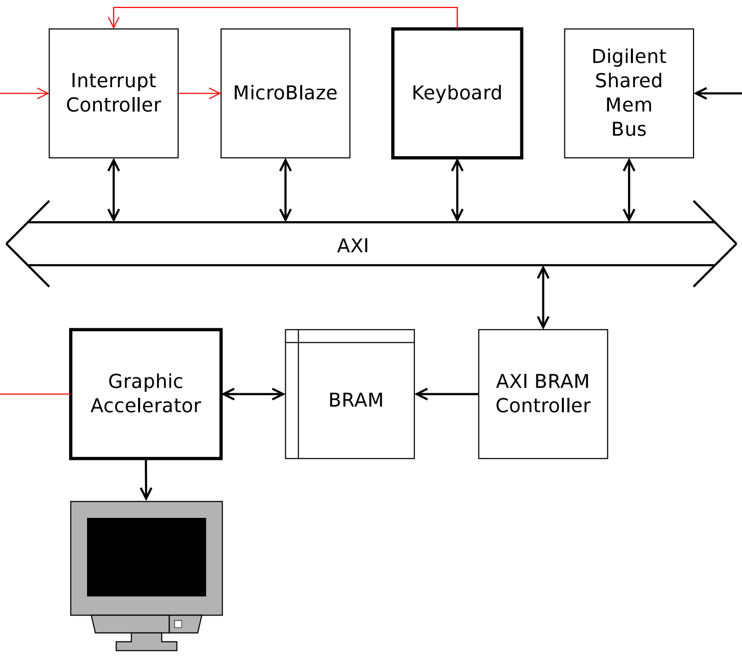


Figure 1: *Final architecture*

When the project result is compared with the propasal there were of course a lot of differences. Most of them because the proposal was incomplete. But one difference were the interrupt controller that was added after the project proposal. It was first thought that only the interrupt from the graphic accelerator was needed. Polling was initally used to interface the PS2 keyboard but the result was not satisfying. Polling is not software independent since you have to do it in the main loop this gives higher response times which is bad in gameplay point of view. Therefore an IP core was used to handle two interrupts, one for the keyboard and one for the graphic accelerator.

Ok, nice explanation.

2 Hardware

The graphical accelerator, see figure 2, on its highest level, is divided in to two blocks: *Renderer top* and *VGA top*. Renderer top consists of a background renderer, referred to as *BG calc*, and object renderer, referred to as *object calc*. Along with this modules comes different memories as well. In VGA top the most important modules are the *RGB MUX* and *VGA Controller*.

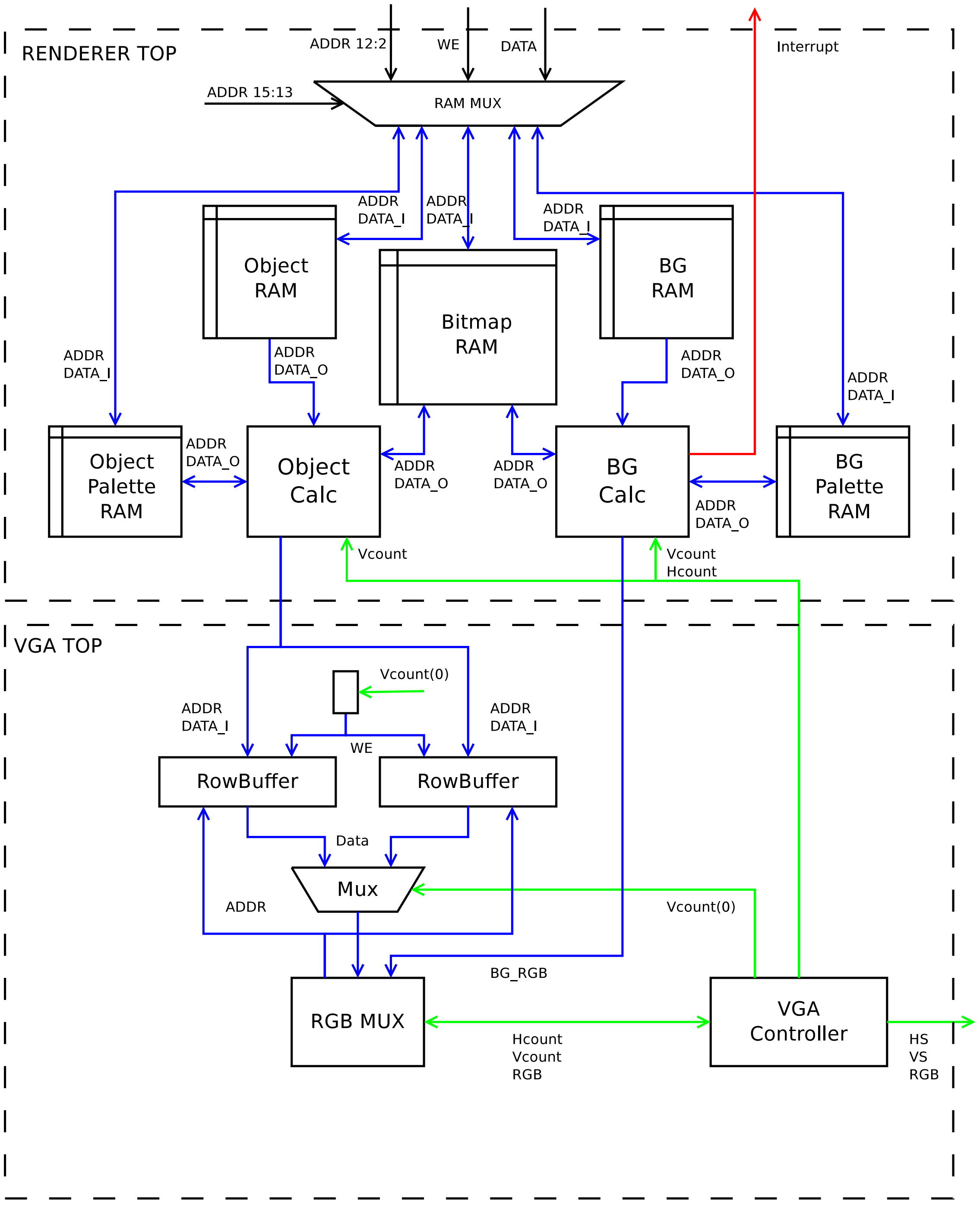


Figure 2: *Hardware Architecture*

Good schematic description of the accelerator!

2.1 Renderer Top

BG calc has read ports to *BG RAM*, *BG palette RAM* and the *bitmap RAM* in order to generate the background tiles. *Object calc* in turn reads from *object RAM*, *object palette RAM* and the *bitmap RAM* to generate foreground sprites. The microblaze is able to write to all the memories through the *RAM MUX* which uses the the highest bits on the address as chip select.

2.1.1 Bitmap RAM

This RAM stores information about every bitmap i.e. every pattern that can be used for tiles and sprites. These pattern are 16x16 pixels. Where every pixel is represented with 2 bits which gives 4 color variations when paired with a palette. The RAM has 32 bit word length thus can one bitmap line be stored in one word. Bitmap is made 1024 addresses deep. Which almost allocates two whole 18k RAM block, except from the shorter word length. This makes 64 different bitmaps possible. In table 3 it can be seen how the main tetris block is represented in the bitmap RAM.

```
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
10 00 00 00 00 00 00 00 00 00 00 00 00 00 00 10
10 10 00 00 00 00 00 00 00 00 00 00 00 00 10 10
10 10 10 01 01 01 01 01 01 01 01 01 01 10 10 10
10 10 10 01 01 01 01 01 01 01 01 01 01 10 10 10
10 10 10 01 01 01 01 01 01 01 01 01 01 10 10 10
10 10 10 01 01 01 01 01 01 01 01 01 01 10 10 10
10 10 10 01 01 01 01 01 01 01 01 01 01 10 10 10
10 10 10 01 01 01 01 01 01 01 01 01 01 10 10 10
10 10 10 01 01 01 01 01 01 01 01 01 01 10 10 10
10 10 10 01 01 01 01 01 01 01 01 01 01 10 10 10
10 10 10 01 01 01 01 01 01 01 01 01 01 10 10 10
10 10 10 01 01 01 01 01 01 01 01 01 01 10 10 10
10 10 10 11 11 11 11 11 11 11 11 11 11 10 10 10
10 10 11 11 11 11 11 11 11 11 11 11 11 11 10 10
10 11 11 11 11 11 11 11 11 11 11 11 11 11 11 10
```

Figure 3: *Tetris block representation in bitmap RAM*

2.1.2 Background RAM

The background RAM stores values to represent the tiles in the background. This values are stored in the same order as they are displayed thus left to right and up to down. Each tile is represented in one 9 bit word, consisting of 6 bit representing the bitmap used in this tile and 3 bits to select palette. Since the screen consists of tileblocks of 16x16 pixels at a resolution of 640x480 there are 1200 tiles and thus 1200 words in the Background RAM.

2.1.3 Palette RAM

Both palette RAMs, *Object Palette RAM* and *BG Palette RAM*, are identical. They both store 8 palettes, each consisting of 4 different colors. Since the vga output on the board is 8 bits each color is represented 8 bits. These are small memories which synthesizes to distrubuted RAM.

2.1.4 Background renderer

This block takes `hcount` and `vcount` as inputs signals. These are used to get correct timing when addressing the RAMs. It has read ports to `BG RAM`, `bitmap RAM` and `BG palette RAM`. It outputs the `BG_RGB` signal to *VGA top* and generates the interrupt for the MicroBlaze when going in to `idle` state.

BG calc idles until last pixel count in blank. It then starts to address the *BG RAM* to able to output the first pixels on the frame correct. Most of the time is spent in state `WAIT FOR NEXT TILE`, see figure 6a. The state machine waits until the last pixel, the 15th pixel i.e. `hcount(3 downto 0)="1111"`, the output `BG_RGB` is done by a separate process which can be seen explained in figure 4.

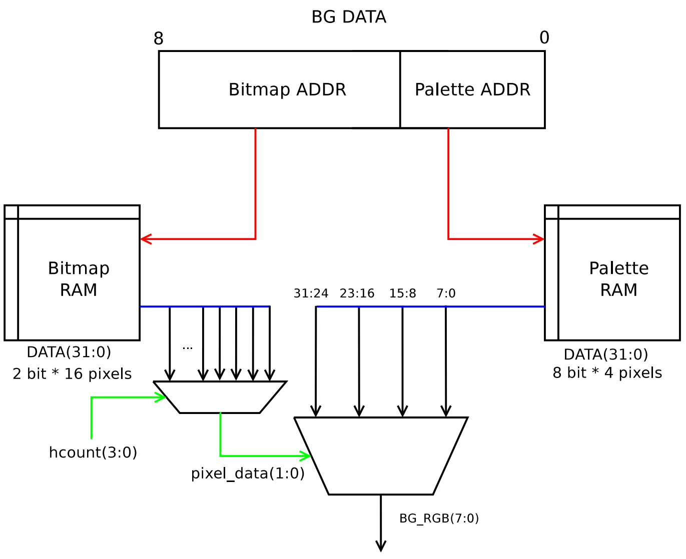


Figure 4: *The output process of the Background renderer*

2.1.5 Object RAM

Stores the X and Y coordinates followed by the bitmap and palette that is supposed to rendered. I.e. One objects in hardware is one bitmap big. X is represented by 10 bits, Y by 9 bits, palette by 3 bits and rest of the 32 bit word is left for bitmap address.

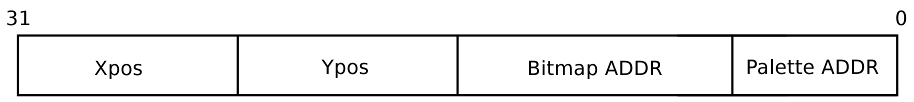


Figure 5: *Object word*

2.1.6 Object Renderer

The Object renderer works cyclic due to use of row buffers. I.e. `vcount` is taken as an input and the same actions are executed for every line. I has read ports for Object RAM, bitmap RAM, and object palette RAM. The calculated output data is written to the row buffer in VGA renderer.

Every cycle is started with clearing the current buffer of data from the before. Afterward it moves to the `Find Objects` state, see figure 6b. There it starts going through all the objects placed in object RAM. As soon as an object is found on the line it jumps to `write object` state. It writes the current row of the object and afterwards jump back to find objects to complete the iteration through *object RAM*.

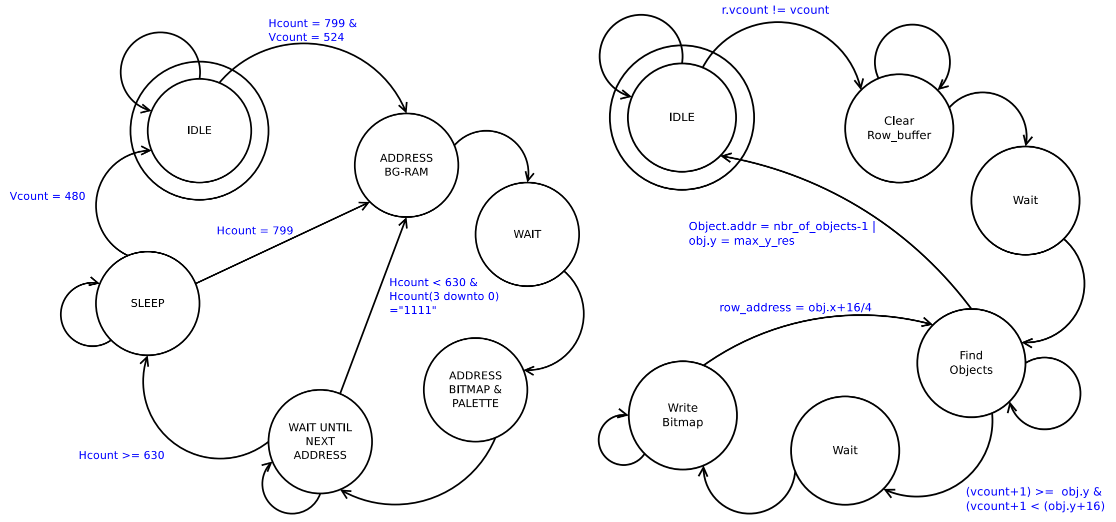
2.2 VGA Top

The most important functions in *VGA Top* is *Row buffers*, *RGB MUX* and *VGA controller*.

Row buffers Stores information about the objects on the present line. Every word is 36 bit and contain information about 4 pixels. Each represented by a 8 bit RGB code and one bit for if there is and object or not.

RGB MUX has a read port to the row buffers and takes `BG_RGB` as input. The main function is to check if there is an object and in that case output the `RGB` from the row buffer and in other case output `BG_rgb`.

VGA Controller Generating the `HS` and `VS` signals with a 25 MHz clock input. Outputs `hcount` and `vcount` signals for the rest of the hardware design.



(a) *Background renderer* **(b)** *Object renderer*

Figure 6: *Finite State Machines*

Very good description using state machines! Often a figure like this says much more than a few pages of text (good that you describe it in text still).

2.3 FPGA usage statistics

In table 1 the total FPGA usage, MicroBlaze system and graphic accelerator, can be seen. This can be compared with only grapical accelerators utilization in table 2. In table 3 BRAM usage information according to the reported BRAM allocation in end of ISEs Block Memory generator. Somehow these figures do not add up together. The synthesis tools states 22 RAMB16 (RAMB16 is actuallt 18kb) but the manual summation of the stated allocation from the Block Memory Generator ends up with 21 RAMB16. The RAMB8 blocks is not allocated in the graph acc when it is synthesized by itself. It was thought these ram blocks was allocated by the two row buffers.

The synthesis reported that the crical path in the total design gave a frequency of 99.5 MHz and 134.0 MHz just for the graphic accelerator.

Why two frequencies? What do these refer to?

Primitive	Used (Utilization)
Slice registers	2 740 (15%)
Slice LUTS	3 525 (38%)
Slices	1 367 (60%)
RAMB8	2 (3%)
RAMB16	22 (68%)
DSP48A1	4 (12%)

Table 1: *Total FPGA usage*

Primitive	Used (Utilization)
Slice registers	281 (1%)
Slice LUTS	581 (6%)
Slices	195 (8%)
RAMB8	0 (0%)
RAMB16	6 (18%)
DSP48A1	1 (3%)

Table 2: *Graphical Accelrator FPGA utilization*

Memory	Type needed	Size	RAM type synthesized
Bitmap	True Dual Port	32768b	2x True Dual port 18kb BRAM
Object	Simple Dual Port	16384b	1x True Dual port 18kb BRAM
Background	Simple Dual Port	10800b	1x True Dual Port 18kb BRAM
2x Rowbuffer	Simple Dual Port	2x5760b	2x Simple Dual Port 9kb BRAM
2x Palette	Simple Dual Port	2x512b	2x Simple Dual Port 512b Distributed RAM
Total		71984b	5x True Dual Port 18kb RAM
MicroBlaze		32kB	16x True Dual Port 18kb BRAM

Table 3: *Memory Statistics*

3 Software

The software was written to be run on a single processor using a single thread. That way the MicroBlaze Standalone kernel could be used and there were no requirements of any scheduling or context switching.

3.1 Event driven

The program is event driven since it only reacts upon external interrupts (figure 7). If it is the keyboard interrupt it modifies the game state or player state.

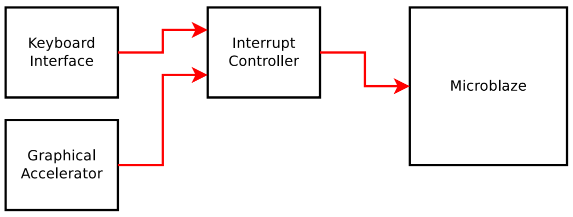


Figure 7: *Interrupts in the system*

Here is a list of all keyboard commands available:

- Game state keys

  Space Pauses / Unpauses game

  Backspace Resets game

- Player 1 state keys

  A Move tetramino left

  D Move tetramino right

  S Hold down to increase speed of tetramino

  G Rotate tetramino left

  H Rotate tetramino right

- Player 2 state keys

  ← Move tetramino left

  → Move tetramino right

  ↓ Hold down to increase speed of tetramino

  , Rotate tetramino left

  . Rotate tetramino right

The graphic accelerator interrupt will fire at about 60hz, at the end of each drawn frame. This gives the software a window of time to write to the graphic accelerators memories without any graphical glitches. Both the background and the objects can be overwritten with new data. Smart solution to avoid flicker.

We also use this interrupt as a form of clock which steps the game one "tick" forward. A "tick" is the games smalles timestep, which is about $\frac{1}{60}$ of a second. After every tick both players tetraminos are moved down according to the game speed or player speed.

3.2 Storage

There is a global struct called `game_state` which contains both the players structs (`player_state`), a global objects list and some additional information. The `player_state` contains player speed, score, current tetramino and next tetramino and some other parameters. The tetraminos are lists (figure 8) of the kind of objects that the graphical accelerator can output. These objects are 32 bit bit-fields that store x- and y-coordinates, which bitmap and what palette to use.

A well known implementation of lists from the Linux kernel header "list.h" was used. Making any struct a part of a list is as easy as adding a member of type `struct list_head`. This means that any struct can be part of as many lists as required, simply by adding more members of the list head type.

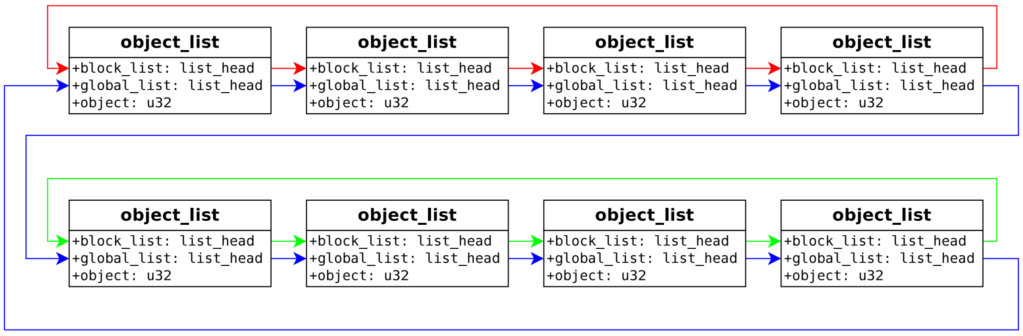


Figure 8: *Objects can be part of both block list, which builds up tetraminos, and global list, which is written to the graphic accelerator.*

To display an object, the main software needs to add the object to the global objects list. This list is traversed and written to the graphic accelerators object RAM after each frame.

3.3 Game logic

Before any move is made, either left, right, down or any rotation, the software checks if there will be a collision. Since a higher resolution than original tetris is used we have to check intervals of 16 pixels instead of only the objects neighbours. This requires some extra logic but makes the gameplay more smooth because the tetraminos can move 1 pixel in every frame.

3.4 Random numbers

A pseudorandom number generator is an algorithm (equation 1) for generating a sequence that seems random. The seed (x_0) was the amount of ticks accumulated when the game was started. The other parameters (a, c and m) were random fixed prime numbers.

$$x_{n+1} = (a \cdot x_n + c) \mod m \tag{1}$$

4 Installation

Follow these instructions to setup and compile the project:

- Install Xilinx ISE Design Suite: Embedded edition 14.2.

- Download and unpack Nexys 3 Board Support files from digilent.com[1]. Don't forget to add the custom IP cores according to the `readme.txt` in the `Digilent_AXI_IPCore_Support_v_1_33` directory.
- Run Xilinx Platform Studio (XPS).
- Create a new project using the "Base System Builder" wizard.
  - Enter a project file name.
  - Select the *AXI* bus.
  - Enter the path to the lib directory in your unpacked AXI Base Support Files as *Project Peripheral Repository Search Path*.
  - Use the defaults in the guide except for peripherals where you can remove all but *Digilent_Shared_Mem_Bus_Mux* and *RS232_Uart_1*.
- Add 25 Mhz clock.
  - Configure the IP Core *clock_generator_0*. By right clicking on it and selecting *Configure IP*.
  - Open the *CLKOUT1* box and enter 25 000 000.
- Make the following pins in *clock_generator_0* external by right clicking on them in *Ports* tab.
  - *CLKOUT0*.
  - *CLKOUT1*.
- Add an external pin for Graphic Accelerator Interrupt.
  - Open the *Ports* tab.
  - Find the button in the upper right area that has the description *Add External Port*.
  - Add a new external port with the name *graph_acc_INTERRUPT_pin*, direction *I* and class *INTERRUPT*.
- Instantiate an *AXI Interrupt Controller* which is located in *Clock, Reset and Interrupt* in the *IP Catalog* tab.
- Open the *Bus Interfaces* tab.
- Connect the MicroBlaze INTERRUPT bus to the *axi_intc_0_INTERRUPT* bus.
- Open the *Ports* tab.
- Connect the following pins on *axi_intc_0*: *Processor_clk* and *Processor_rst* to *clock_generator_0*:*CLKOUT0* and *proc_sys_reset_0*:*Peripheral_Reset*.
- Add interrupts to controller
  - Find the button in the upper right area that has the description *Open Interrupt Control Dialog*
  - Add *ps2_interface_0*:*dav* and *External Ports*:*graph_acc_INTERRUPT_pin*.
- Connect *Digilent_Shared_Mem_Bus_Mux*
  - Open the *System Assembly View*
  - Open the tab *Ports*
  - Connect *S_AXI_ACLK* to *clock_generator_0*:*CLKOUT0* which is located in *Digilent_Shared_Mem_Bus_Mux*, *(BUS_IF) S_AXI*.
  - Make *(IO_IF) mem_mux_bus_0* external.
  - Find *Digilent_Shared_Mem_Bus_Mux_Mem_DQ_pin* in *External Ports* and rename it to *Digilent_Shared_Mem_Bus_Mux_Mem_DQ*.
- Instantiate a PS2 peripheral.
  - Copy the directory `pcores/ps2_interface_v1_00_a` from the source code into the projects pcore directory.

[1] http://www.digilentinc.com/Data/Products/NEXYS3/Nexys3_BSB_Support_v_2_6.zip

  - Rescan user repositories from the menu *Project*
  - In the tab *IP Catalog* open *Project Local PCores* and *USER* to find the pcore.
  - double click on *PS2_INTERFACE* to instantiate the core in the project.
  - Select the tab *Ports*.
  - Make *kb_data* and *kb_clk* external.
- Instantiate an *AXI BRAM Controller* which is located in *Memory and memory controller*.
  - Select *User will make necessary connections and settings*. Otherwise XPS will instantiate a BRAM Core. This controller will be connected to the graphic accelerator via external ports instead.
  - In the tab *Bus interfaces* connect its *S_AXI* bus port to the *axi4lite_0* bus.
  - Right click on the IP core and select *Configure IP*.
  - Locate *Slave Single Port BRAM* and enable it.
  - Switch tab to *Ports* and select all checkboxes in *Port Filters*.
  - Make the *(BUS_IF) BRAM_PORTA* external.
  - Connect *S_AXI_ACLK* to *clock_generator_0*:*CLKOUT0*.
- Clear UCF file – another UCF file will be added to the ISE project later instead.
  - Select the *Project* tab.
  - Double click on *UCF File: data/system.ucf*
  - Remove all content
  - Save file
- Generate AXI addresses.
  - Enter tab *Addresses*.
  - Change *axi_intc_0* and *axi_bram_ctrl_0* to 64K.
  - Change *microblaze_0_d_bram_ctrl* to 32K.
  - Run *Generate Addresses*.
- Quit XPS.
- Run ISE Project Navigator (ISE).
- Create a new Project.
  - Select *HDL* as *Top-level source type*.
  - Select *Spartan6* as *Family*.
  - Select *XC6SLX16* as *Device*.
  - Select *CSG324* as *Package*.
  - Select *VHDL* as *Preferred language*.
- Add all source files from `vhdl/src` by right clicking in the box with *Empty view* and selecting *Add source...*
- Add the UCF file `system.ucf` from `vhdl/ucf`.
- Add the XPS project.
- Click on *System (system.xmp)* in hierarchy.
- Run *Generate Top HDL Source*.
- Right click *tetris_top* and choose *Set as Top Module*.
- Find *system_i – system (system.xmp)* and run *Export Hardware Design to SDK with Bitstream*. This will take about 10 minutes depending on your hardware.
- ISE will run Xilinx SDK (eclipse) for you.

- Select *File* > *New* > *Project...*.
- Select *Xilinx C Project*.
- Select *Empty Application*.
- Right click on *src* in *empty_application_0* and select import.
- Select *General* > *File system*.
- Import all files from `C/src`.
- To fit the program to the FPGA you have to make the complier optimize for size and put the heap and the stack on the PSRAM.
  - Right click on the software project and select *C/C++ build settings* to change compiler settings
  - Right click on the software project and select *Generate Linker Script* to move the heap and the stack.
- Program FPGA.
- Play Tetris!

Detailed description. However, I meant that starting from the FPGA configuration file, what does one need to get the game running. The steps for building the system from scratch were not required.

5 Problems & Conclusions

While trying to initalize the BRAMs in the hardware design it was discovered that MicroBlaze addressing is byte oriented while we wanted to write complete 32bit words. So when incrementing a pointer it adds 4 to the address. This means that the 2 least significant bits from the BRAM controller to the hardware memories should be thrown away.

The initialization data, bitmap and palette data, that is loaded on to hardware in the initialization phase of the software occupies twice the BRAM on the FPGA as needed. The RAM is allocated by the graphics accelerator and then it's programmed into the MicroBlaze BRAM on programming. This would of course have been better to store in non-volatile memory. Since it is only loaded once and has no speed requirements.

The main part of the MicroBlaze BRAM is occupied by .text i.e. the instructions, about 30kB. Since MicroBlaze is configured with 32 kB this means that almost all the BRAM is occupied with instructions. It would ofcourse be better to store in an larger external memory accessing using an instruction cache. This would have made it possible to place heap and stack in BRAM instead of placing it on the slow PSRAM. Which would have been a much neater design.

Not a sentence! Do not split phrases like this.

A lot of problems encountered were probably misjudged and it might not even been errors. This due to to poor testing. For exmaple ill thought testing data generating an output thought to be wrong. When it might have been right. The lesson we learned is to construct well defined tests, where the expected output is known. One of this incidents is described below:

The hardware was designed with AXI bus, and therefore a new IP core provided by digilent to interface the external memories was used, called Digilent mem bus mux. But this core has no documentation what so ever. For example it outputs a 17 bit databus. The bugs this introduced was probaby software related since we now using it with correct behavior. But we still miss some documentation describing this ip core.

Similar problem occured when first testing the hardware with some "random" input data to the background memory.

6 Contributions

The graphic accelerator was written mostly by Alexander, only the object calculation block was written by Niklas. The PS2 interface was written by Niklas and most of the software was also written by him, excluding everything related to the rendering of the background and som specific game features. Both have contributed equally to this report, the proposal and both presentations.

Overall a very successful project. Well done!

PS. some references would have been nice
here at the end.