Design of Embedded Systems Advanced Course
Mario Breakout

Adam Dalentoft, `dat11ada@student.lu.se`
Simon Wallström, `dat11swa@student.lu.se`
Viktor Sannum, `dat11vsa@student.lu.se`

November 4, 2015

Contents

Abstract

When developing an embedded system there are a lot of important aspects to consider. Not only are there numerous technical details which have to be considered in order to end up with an as efficient and cost effective system as possible. Knowledge in many sub fields in system development is required as well as a well working development process. In our project we encountered numerous obstacles related to many of these areas, from having to learn a new programming language and style (VHDL) in a fluent fashion, to gain understanding of how many components have to interact over a system bus and implementation specifics in the VGA standard.

However, in the end, the biggest issues were not technical but rather methodical, we switched from a static waterfall-like method to a more dynamic one. For future project more focus should be on how to divide work between developers more effectively as well as what should be done when road block issues are encountered.

In the project, we chose to develop a classic arcade game and while the final product was kind of what was described in the initial proposal, a lot of the other implementation specific details have been changed in the end.

1 Introduction

This is the final report for the project in the Design of Embedded Systems Advanced Course (EDA385). The goals of this report is to discuss and conclude the work done during the project. This includes different hardware and software solutions and the interface between them when implementing a Breakout style game. The project, as mentioned, is a Breakout style video game where the player controls a paddle along the bottom of the screen. The paddle is used to bounce a ball against a wall of bricks, with the goal to clear the play field from bricks.

The sections Hardware and Software describe the implementations and design decisions made in the project. Following, is a brief user manual that covers installation of the software and controlling the game. Finally, the section Conclusions will conclude the report, cover problems encountered and lessons learned.

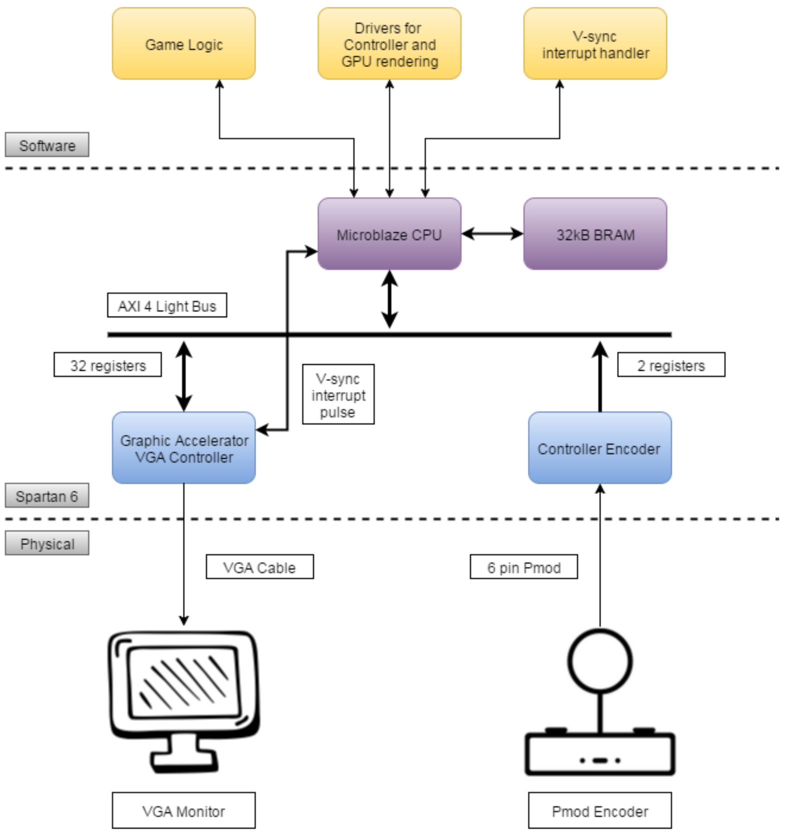

Figure 1: Architecture overview; blue components represents custom IPs, purple already existing and yellow is custom software.

As can be seen in Figure 1, the architecture consists of two custom made *VHDL* modules: one for the input controller and one for the VGA graphics output controller. The hardware modules are communicating with a Microblaze processor, and is thus accessible by the game logic, via an *Axi4Lite* bus. Which is one of the major deviations in the final project from the project proposal. In the proposal *FSL* buses where used. Another difference is that the controller is based on a rotary encoder instead of the on board buttons, as mentioned as a possible improvement.

2 Hardware

The physical hardware base is a Nexys 3 Spartan 6 FPGA board[1]. This is configured with a MicroBlaze processor[3]. The processor has access to 32kB of BRAM memory and runs at a clock speed of 100MHz. The FPGA is also configured with a custom made graphics controller and an user input controller. The custom controllers are communicating with the software on the Microblaze over an *Axi4Lite* bus as well as interrupts. The final system is estimated to use 0.173W which is almost the same as the dual core setup used during the labs of the introduction course. The comparison with a dual core setup continues to hold when looking at the utilization of the board with 15% of Slice Registers used, 40% Slice LUTs and 66% Occupied Slices. It is interesting that the custom made graphics core uses almost the same amount of hardware as an extra Microblaze core would take.

The hardware architecture is created using Xilinx Platform Studio (XPS)[2]. With this tool different hardware parts are connected to a MicroBlaze processor and synthesized. XPS was also used to get information about the hardware utilization. To build, simulate and test the VHDL code for custom made hardware, ISE Project Navigator was used[6].

2.1 Graphics

The graphics controller is responsible for generating the VGA signal which is displayed on the screen. It consists of several separate VHDL components, as follows:

The generated VGA signal is a standard 640x480 pixels 60 Hz signal which requires a pixel clock at 25MHz. The component *vga_clk_25* is responsible for taking a 100MHz clock signal and generate a 25MHz signal out of that.

vga_controller grabs the 25MHz signal and generates the correct timings, synchronization signals and blanking signals which are required for VGA output.

vga_src receives the two signals vCount and hCount from *vga_controller* which indicates which pixel currently is to be rendered. *vga_src.vhd* contains a number of hard coded sprites and uses 32 memory mapped registers (currently only 23 are actually used) to determine the color output for a given pixel. All graphics are logically draw in layers,

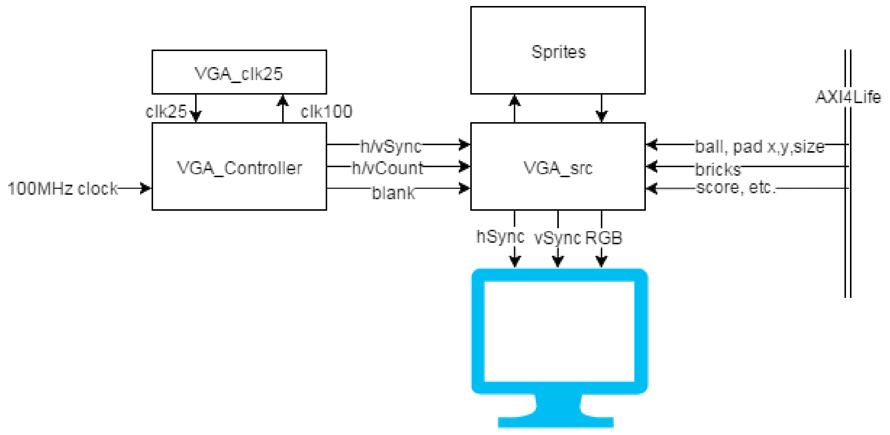

Figure 2: VGA controller overview

if one layer determines a pixel to be transparent, the next layer is drawn etc. If all layers are transparent, the background is drawn.

The layout of the components and signals in the graphics controller is illustrated in Figure 2.

The memory mapped registers have different responsibilities: The first 24 are representing the various blocks which make up the wall in the game. Each block is represented by 4 bits (which would allow for up to 16 different types of block sprites), which means that one register holds 8 blocks. The position of each block on the screen is implicitly given by its position in the registers and the blocks are laid out in a 16*10 grid (which means that only 20 registers are actually currently used). One register keeps the ball x and y position (10 bits each), its size and x direction. Another register keeps the paddle x position (10 bits) and size. A third register holds the score (4 bits per digit) and amount of lives (not used). Refer to Figure 3 for a simplified overview of the memory mapped register layout.

The memory mapped registers are auto generated using Xilinx EDK. This makes them simple to integrate.

One noticeable detail with this implementation is that much of the functionality is hard coded making the *vga_src* component quite inflexible. An earlier implementation used a frame buffer (on external cellular memory), however this turned out to require too much development time (more than a week and a half was spent developing this while not generating good enough results) while also being deemed to be a too much overkill solution. Yet still a lot of improvements, such as: cleaner code, removing unused registers and code or implement the features that would use them could be made on the current implementation.

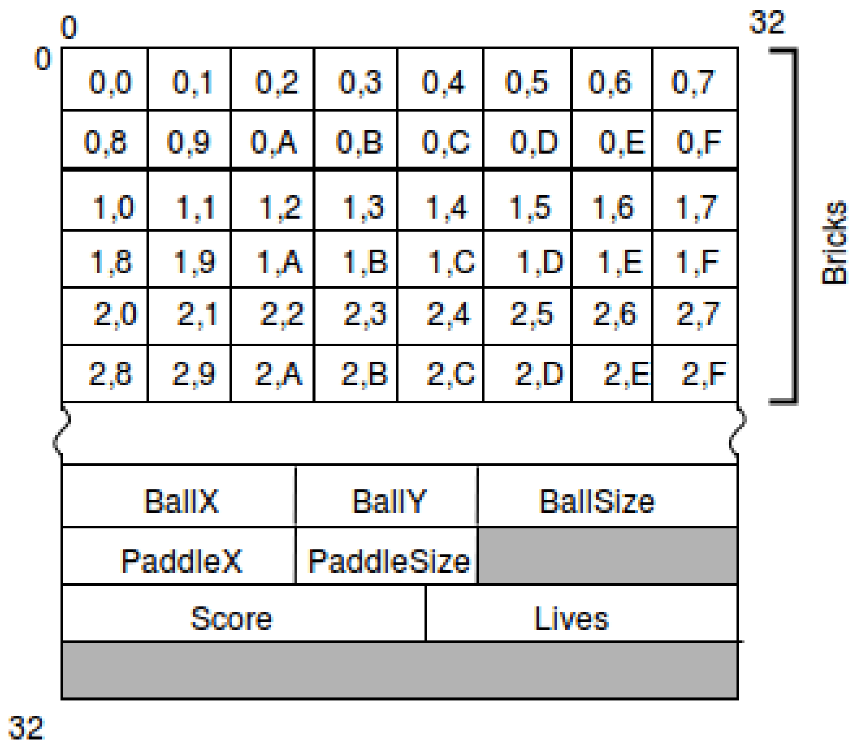

Figure 3: Memory mapped registers overview

2.2 Controller

In the beginning of the project the controller was implemented with the push buttons on the Nexys 3 board. This implementation was very simple and and built on the *xgpio* api available with the Microblaze core. The controller worked but the time plan allowed a custom made one instead. In the 'possible improvements' section in the project proposal both PS2 controller and analog steering wheel was mentioned. The analog steering wheel was chosen for the retro feeling.

The final controller is based on a Pmod Rotary Encoder from Diligent [7], described in Figure 4. The encoder is connected to the first row (6 pins) of the first Pmod connection on the Nexys board. Four of these pins are used in the WHDL module that handles the controller and the other two are for power. Two of the four outputs, A and B, are used to decode if the rotary shaft is rotated to the left or to the right. The other two are BTN, a push button, and SWT, a switch. In Figure 5 a timing diagram of a right rotation of the shaft is shown.

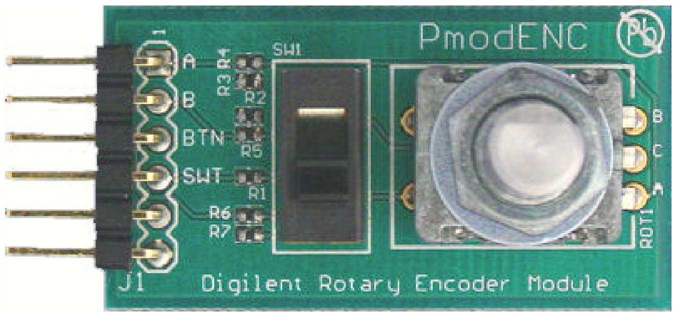

Figure 4: Diligent Pmod Rotary Encoder, used as controller for the paddle [7].

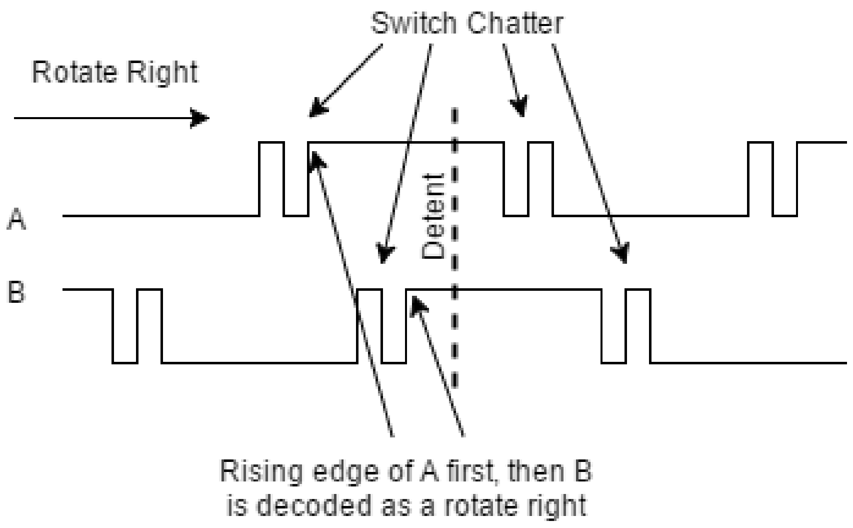

Figure 5: Timing diagram for the Rotary Encoder, this diagram shows the output to A and B when the rotary shaft is rotated to the right [8].

The *VHDL* module that handles the input from the rotary encoder is based on an example implementation provided by Diligent [7]. The provided *Debouncer* module is used to get rid of noise on the output ports. The module writes a position variable between 0 and 31 to a memory mapped *Axi4Lite* register as well as the button and switch values.

This pcore was integrated in the XPS project and connected to the Axi bus and the external ports previously mentioned. The auto-generated *.h*-file for Axi communication is slightly modified to provide a *get_position()* function. This is the way the rotary encoder is integrated in the software.

3 Software

The Eclipse based Xilinx Software Development Kit 14.2 (XSDK) is used for for software development and compilation[4]. Adept 2.4 by Digilent is used to load the compiled software to the Nexys 3 board[5].

The software used in the project are mainly game logic and communication with the hardware. All the software is written in C, both the finished version and the simulations and tests. To avoid problem with memory allocation and memory leakage, dynamic memory allocation is completely avoided. Hence no libraries are needed to be included which is relevant as the system got limited memory. Dynamic memory is also a problem in an embedded system, especially this one, as it got no operating system or garbage collection to control the memory.

The code was developed in several iterations and got simulations to check functionality before the final hardware is done. To do these simulations, the C and C++ compatible graphics library SDL 2.0 is used [9]. This library is rather advanced in the sense that whole games can be made by it. However, it also got the functions to make rectangles and basic keyboard recognition. Thereby the simulation is just the most basic version of the program, just to see that the functionality of the game logic is correct.

The complete code which runs on the board is separated in several files in order to achieve some sort of structure. The main game logic is located in the file *breakout.c*. This file contains initiation of variables, render function, update game and collision detection. The collision detection works from a ball point of view. That is, the ball is constantly checked if it is about to hit another object (brick, wall or paddle). This solution deals with the problem of the ball going through the bricks and thereby making strange bounces of the brick. Each time the display is updated the collision detection is called to determine the preferable action for the ball.

The file *vga_renderer.c* contains all the communication with the vga-controller. That is, draw ball, bricks, pad and score board to the screen. This is done by directly setting coordinates from the software to the registers in the hardware. However, the software values do not correspond directly to the register value describing the correct position on the screen. Therefore the values need to be shifted to fit the register vectors. There is support for several sprites both in software and in hardware. From the write functions it is possible to specify which sprite is to be written to a specific position on the screen. Each sprite got a corresponding value in hardware which make the sprite handling almost entirely hardware based.

The file *main.c* deals with the main loop and the interrupt handling of the program. The main function register an interrupt handler to react on signals from the hardware. When the processor receive an interrupt on the *(*vSync) signal this handler is triggered.

This in turn calls the game update function which updates the internal state model and renders any changes on screen. Any calls to hardware is contained in the driver file *vga_renderer.c*.

The memory requirement for the software is low, at around 6 kB. The system has 32 kB of memory at its disposal and there is plenty of memory left for variables which are not allocated until runtime. All graphics are drawn directly by the custom made vga-controller, which makes the memory clear of large sized graphics.

To make the mapping between software and hardware possible several auto generated files are used. These files mainly contain constants that map to registers in hardware to make it easier to communicate. There are also files with initiation functions for the platform, namely *knob.h* and *platform.h*.

4 User Manual

This section cover the installation and instructions on playing the game.

4.1 Installation

Shipped with this report is both a *.bit* file ready to be loaded on an Nexys 3 board and a package with XPS and XSDK projects with the implementation of the hard- and software. The XPS project holds a folder named *pcores*, where the VHDL implementation of the GPU and controller is saved. The XSDK workspace is named *ws* and holds the C code for the game and interfaces. To prepare the board for the game, a Pmod Rotary Encoder needs to be connected to the top row of the first Pmod connection (JA1) and the VGA port needs to be connected to a compatible monitor.

4.2 Playing the game

The controls are simple, to steer the paddle left and right rotate the rotary encoder. Push down the rotary encoder to release the ball and start the game. The player can at any time pause the game with the switch on the rotary encoder.

The game starts with a demo level. The first game level loads when the rotary encoder is pushed down. On each level the player is able to place the paddle anywhere on the screen before starting. When all bricks on the screen are gone, the next level is loaded. The coins only give extra points and do not need to be cleared.

The player has five lives, thus the game ends and restarts when all lives are lost. There are four levels that are looped until the player dies. As a special feature, the player can activate a bot that plays automatically on switch eight on the board.

5 Conclusion

This section contains problems encountered and lessons learned working on the project. This also include solutions and other thoughts about the work during the process.

5.1 Problems and possible solutions

As the project is rather free in terms of design choices and ways to do things, the only limit is the hardware. It is easy to get stuck on a path if it seems to be the right one. In this case, the software was built differently from the beginning. It had a collision detection build on comparing the balls coordinates with the coordinates of all the blocks every rendering cycle. This solution was not only computation heavy but also had a rather difficult bug. When the ball came towards the bricks in a specific angle it sometimes went straight through the brick instead of bouncing off it. Many hours were put to just try to solve this problems as the rest of the collision detection was working fine. However, this bug was never solved and the collision detection was instead changed to the one described in the software section. Thereby the bug was avoided and the collision detection also got better in terms of computation heaviness. In this case much time was put on trying to solve the first bug and it was not solved until another member of the group (Viktor) read through the code and came up with the new idea. Therefore it is important to not get too stuck on one track but try to discuss the problem with someone else and find a new path to success.

Similarly, a lot of work was put into developing the first iteration of the graphics controller and working with the on board memory chip. In the end these ideas were scrapped when it was realized that there was a great risk that it would not be finished in time. Instead a rather 'hacky' approach was used and a working graphics controller was built in roughly one or two days, and refined as more time progressed. Not only was a lot of work wasted but it had also limited the work the other team members could do, since they were getting more and more dependent on working graphics. The most valuable lesson learned from this is not to have a grand ambitious plan from the start but rather start small with something which at least is working and work the way up from there. Ideally the 'hacky' approach should have been tried first, changing to the cleaner one if it was seen as a nice or necessary improvement and if time would have allowed for it. One can kind of see a similarity to the Waterfall vs. Agile development techniques in this.

Testing is very important when it comes to system development. In terms of the software, as previously mentioned, a simulation was made to make sure the software works before it is put together with the rest of the system. This made the development much easier, since one could completely exclude software bugs when putting the whole system together. The other parts of the system got their own tests as well. Hardware uses test benches in VHDL and the controller part uses basic software to check the output.

Because of the well-tested parts of the system, the completion of it and a running version is easy to put together. Therefore more time can be put into fixing and polishing the product than into finding and fixing bugs that could have been found in a much earlier stage of the project.

When first testing the rotary encoder as controller for the game, a big problem was noise and stability on the input *A* and *B*. This was quickly solved by modifying and using a part of Diligents example code, the *Debouncer*. This is acting as a low pass filter on the ports from the encoder.

5.2 Lessons Learned

As previously mentioned both the software (collision detection) and the hardware (GPU) changed during the project, but, before the new implementations, both parts were stuck for a while. The lesson learned from this is to not be afraid to throw away an implementation and start over with more knowledge from the failed attempt. The road to success usually goes through a couple of iterations of non-optimal ideas before the optimal solution is found.

Another lesson learned is that it takes time to compile the hardware, which can lead to a lot of wasted time. As the project went on the group members learned to use this time to do other useful things, such as work on the presentation, report or some other part of the system. Coffee breaks ware also scheduled around hardware compilation. The lesson also included planning of the implementation before compiling and not compile for a crash-and-burn test, as sometimes done in smaller software systems.

Something that would be done differently if the project started today is the initial planning and time estimation. As it turned out the GPU took a lot more time than expected. It would have been better to let two people work on it to get a working GPU in to the system earlier and then be able to work on improvements if needed.

5.3 Contribution Statement

Contribution by group member:

Simon implemented VHDL module for the controller, interfaced it in the software and tested AXI4Lite communication. Wrote the controller section in this report as well as the intro, the user manual and hopefully something more...!

Adam wrote game logic and simulation in C. Refactored the code and tested the full system together with Viktor. Wrote the sections in the report connected to software, that is the software section and parts of the conclusion.

Viktor was responsible for designing the graphics controller, writing it in VHDL, Simon helped with the integration and AXI parts. He also wrote the section concerning graphics.

References

[1] Diligent store site for 'Nexys 3 Spartan 6' FPGA board.
http://www.digilentinc.com/Products/Detail.cfm?NavPath=2,400,897&Prod=NEXYS3&CFID=18681702&CFTOKEN=933d14222352826d-2F6836B0-5056-0201-02E1D9AB2D87993A
(15/10-15)

[2] Xilinx, Xilinx Platform Studio
http://www.xilinx.com/tools/xps.htm
(19/10-15)

[3] Xilinx support documentation, MicroBlaze
http://www.xilinx.com/support/documentation/sw_manuals/mb_ref_guide.pdf
(19/10-15)

[4] Xilinx, Xilinx Software Development Kit
http://www.xilinx.com/tools/sdk.htm
(24/5-15) Adept by

[5] Digilent inc.
http://www.digilentinc.com/Products/Detail.cfm?NavPath=2,66,828&Prod=ADEPT2
(19/10-15)

[6] Xilinx, ISE Project Navigator
http://www.xilinx.com/tools/projnav.htm
(19/10-15)

[7] Diligent store site for 'PmodENC - Rotary encoder', this page contains schematics, reference manual and examples available for download. http://digilentinc.com/Products/Detail.cfm?NavPath=2,401,479&Prod=PMOD-ENC (28/9-15)

[8] Diligents reference manual for 'PmodENC - Rotary encoder'. http://digilentinc.com/Data/Products/PMOD-ENC/PmodENC_rm.pdf (28/9-15)

[9] Graphics library for the software simulation
https://www.libsdl.org/
(13/10-15)

A Eye Candy

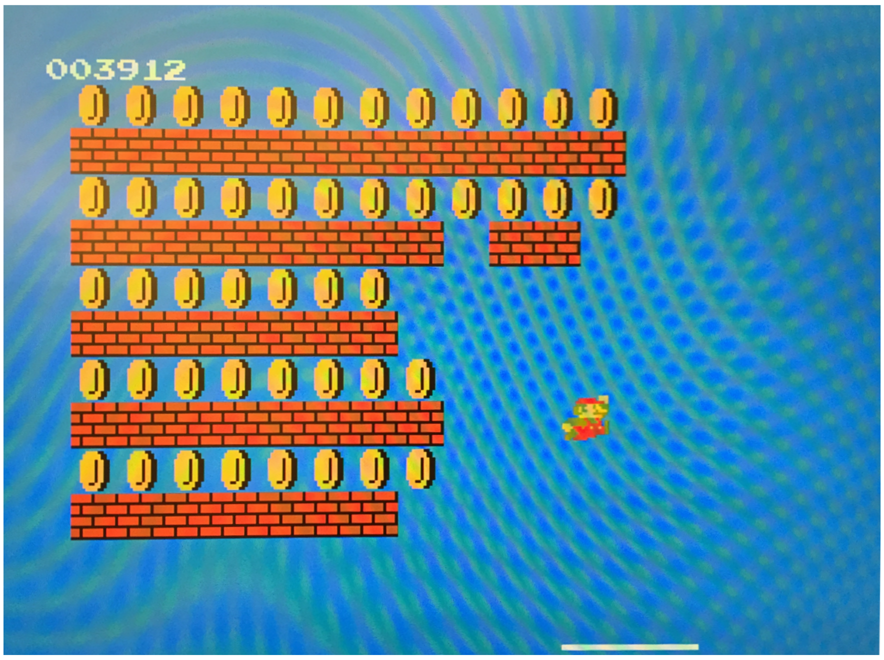

Figure 6: First level.

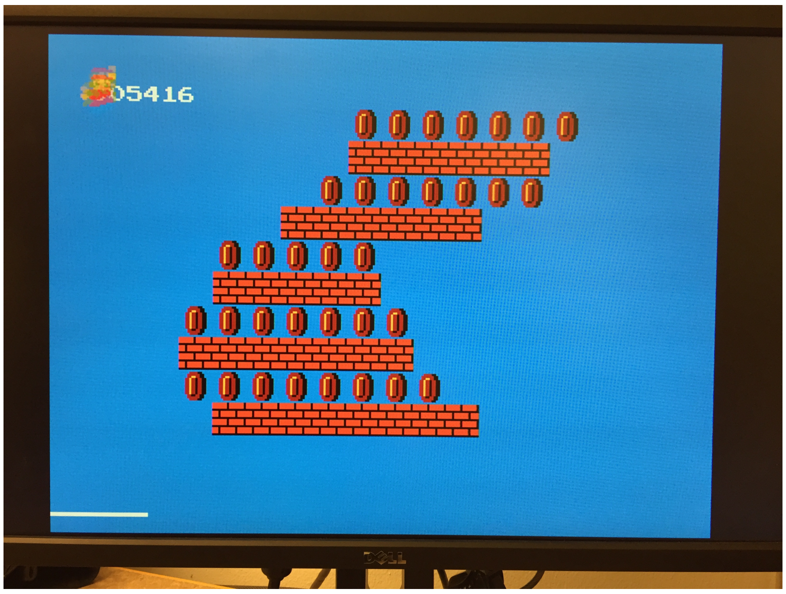

Figure 7: First level.

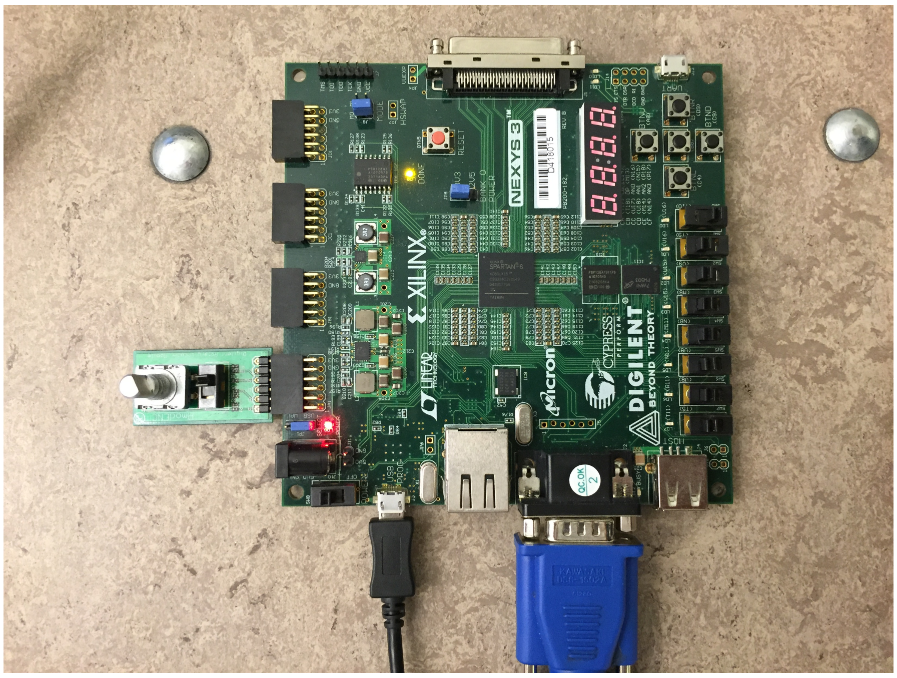

Figure 8: Board with controller.