EDA385 Bomberman

Fredrik Ahlberg
ael09fah@student.lu.se

Adam Johansson
rys08ajo@student.lu.se

Magnus Hultin
ael08mhu@student.lu.se

2013-09-23

Abstract

This report describes how a *Super Nintendo Entertainment System*-like system running the classic game Bomberman was developed and implemented on a Nexys 3 development board. Custom IPs were written in VHDL and the software, running on a Xilinx MicroBlaze soft processor, was written in C. Four original Nintendo gamepads was used for player input, and a speaker was added for playback of music and sound effects. The project turned out great, instructions on how to connect gamepads and a speaker is included in this report along with timing diagrams. The project took seven weeks and was done as a part of the course "Design of Embedded Systems - Advanced Course" taken at LTH in Lund, Sweden.

1 Introduction

In this chapter the concept behind and goals of the project is explained as well as a few design ideas. An overview of the system architecture is also presented. In the following chapters detailed descriptions of the custom IPs and program code will be given along with installation instructions for anyone wanting to test the game. Lastly there will be a discussion about problems that occurred, possible improvements to the design and thoughts about how the project turned out.

2 Concept

In the year 1995 the game Super Bomberman 2 was released in Europe by Hudson Soft for the Super Nintendo Entertainment System (SNES). In it, the player controlled a small "bomberman" that navigated a maze and could place bombs that blasted both obstructions and enemies out of the way. It contained both a single player storyline and a multiplayer component. The multiplayer component was the only focus of this project. In it up to four players battled it out in a maze-like arena collecting powerups and bombs until only one player remained. Figure 1 and 2 show a screendump of the original game alongside with a picture of the FPGA running the finished project.

Figure 1: Screenshot from *Super Bomberman 2*

3 Architecture

The original goal for the project was to implement a working Bomberman multiplayer game for up to four players without powerups or any sound. Original Nintendo gamepads were to be used, as well as original graphic elements and the game was to be displayed in 640x480 resolution @ 60 Hz using a VGA interface. The system was to be designed as software independent as possible, meaning that in theory any game could be played by running a different program.

The final system achieved all this, as well as support for music and sound effects, and the software was extended to include three different power ups improving the overall feel of the game. The graphics components, music and

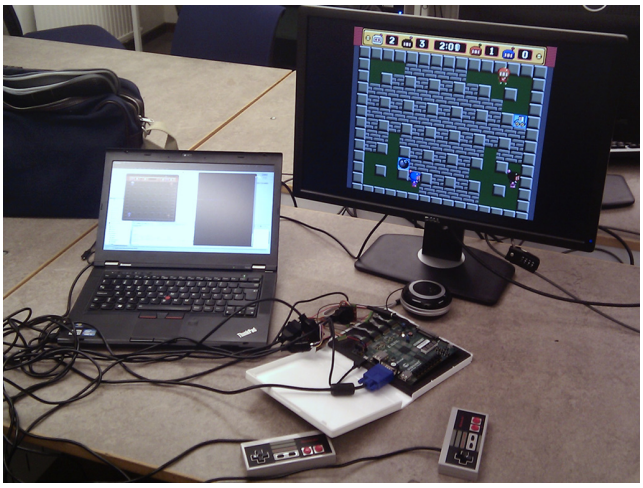

Figure 2: The final system up and running.

sound effects ended up being stored on the onboard flash memory since it would be too big to store in program memory.

In figure 3 a block schematic of the finished hardware is shown. All communication with the peripherals is done over the AXI bus. The custom IPs are implemented with the AXI Memory bus interface, so that they can easily be connected to any system using the AXI BRAM Controller IP, and memory mapped by the MicroBlaze. The GPU outputs the VGA RGB signals as well as an interrupt signal (IRQ). The IRQ is connected to the MicroBlaze using an AXI Interrupt Controller, meaning that more interrupts could easily be added and handled by the system if wanted (a keyboard interrupt for example). The serial flash memory is accessed using the AXI Quad SPI Interface core. This core is configured to run in a read-only mode, reducing the complexity of the data transfer and eliminating the need to read or write from any registers to set up a transfer. The flash memory is pre-loaded with the game components using Digilent Adept.The VGA timing generator was made available in a previous course and simply generates a horizontal and vertical counter used by the GPU as well as VGA synchronisation signals for the monitor.

4 Hardware

4.1 GPU

The GPUs task was to render the graphical elements in the game, meaning it had to decide the color of each pixel at the exact time it was needed by the screen. It was designed to be simple to program, requiring as little information from the user as possible, but still versatile and capable of displaying a lot of graphics elements, keeping in mind that it should be able to render any SNES-like game, not only Bomberman. It also had to meet the timing constraints of the system. Figure 5 shows a slightly simplified version of the entire GPU.

There are two components that make up the graphics. Tiles that are used

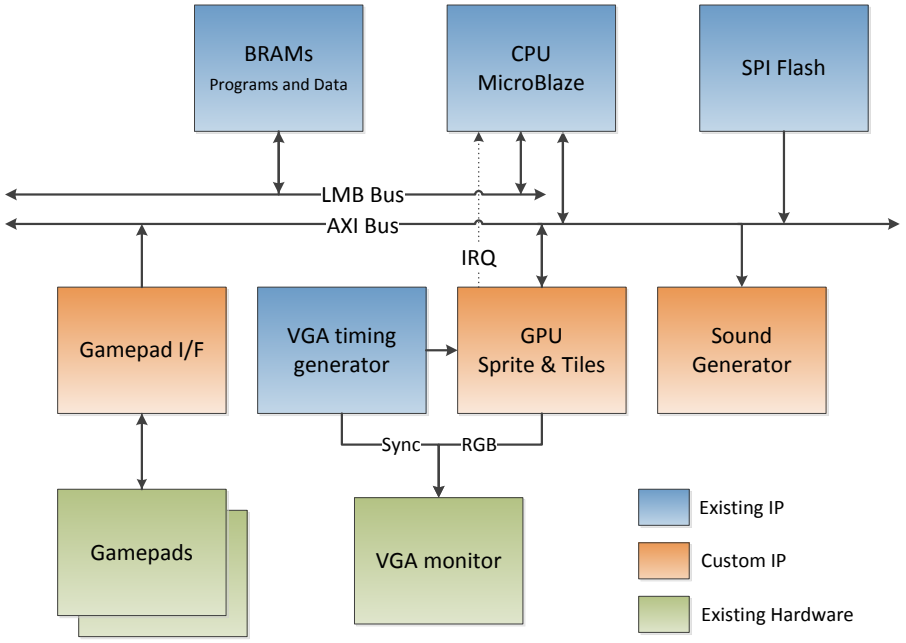

Figure 3: Overview of the hardware architecture

for background, and sprites that are used for foreground objects. Both tiles and sprites are 16x16 pixels large, however, tiles are aligned to a grid and cannot be moved independently. Sprites on the other hand can be offset by any value from this grid. Sprites and tiles are made up by bitmaps, and a palette index. The bitmaps are made up by 4-bit color indexes, meaning that each separate sprite or tile can contain up to 16 unique colors. To color the sprite or tile the color index needs to be looked up in a palette. Each palette contains up to 16 colors, and by changing the palette index, the same sprite or tile can be rendered in different colors, and different sprites or tiles can use the same palette. This saves a lot of memory. Figure 4 shows an example of how the bitmap - palette combination works.

Rendering the background is simple and deterministic and thus performed on-the-fly by cascading the tilemap and bitmap memories, addressing the tilemap with `hcount` and `vcount` from the timing generator.

The sprite rendering is, to the contrary, much more complex as the work needed to render a single pixel depends on the number of active sprites and their positions on the screen. This problem is solved by rendering the sprite contents for each scan line "in advance" and storing it in a line buffer. The line buffer consists of a BRAM which is divided into two halves (lines) of 512 pixels each. One line is read and output to the display while the next line is rendered into the other half of the memory.

The rendering is controlled by an FSM, which is clocked at 100 MHz, but synchronized with the VGA timing. When a line in the buffer have been displayed and thus its memory become outdated, the FSM starts rendering the next line:

- Clear the line in the line buffer by writing "transparent" pixels at every

position.

- Iterate through the 512 sprite slots in the *Sprite Attribute Memory* (SAM). Each sprite occupies two 18-bit words in the SAM. The first word contains the `X` and `Y` coordinates and an `enable` flag.

  If the current sprite is not enabled or if the currently rendered scan line does not intersect the sprite, judging from its `Y` coordinate, then the FSM will naturally continue with the next sprite slot. Otherwise:

- Read the next sprite attribute word, containing `bitmap_idx`, `palette_idx`, `invx` and `invy`, and store it in the FSM state. `invx` and `invy` are flags which mirrors the sprite in X resp. Y direction.

- Read the 16 pixels of the intersection of the sprite bitmap, and write them into the linebuffer together with the palette index, and with the transparency bit flipped. The pixel write is only committed if the color index value is not equal to the index of the "transparent color"; thus sprites can partially overlap eachother.

The color and palette data from the tilemap core, as well as the color, palette and transparency data from the sprite core is then fed to the palette block. The color and palette indices are selected from either tilemap or sprites using muxes, controlled by the transparency bit from the sprite core, and are then used to address the palette RAM. The resulting 18-bit word contains the red, green and blue color values as 6-bit words. Those are then used to control the video DAC on the development board.

The GPU is connected to the AXI bus using a single port AXI BRAM controller, which allows direct read and write operations to any of the memories contained in the GPU. Read support was originally not considered necessary, but was added late in the development to allow sprite sorting.

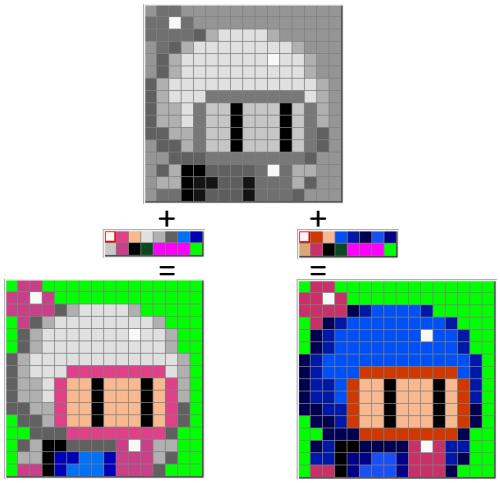

Figure 4: Illustration of the usage of palette and bitmap

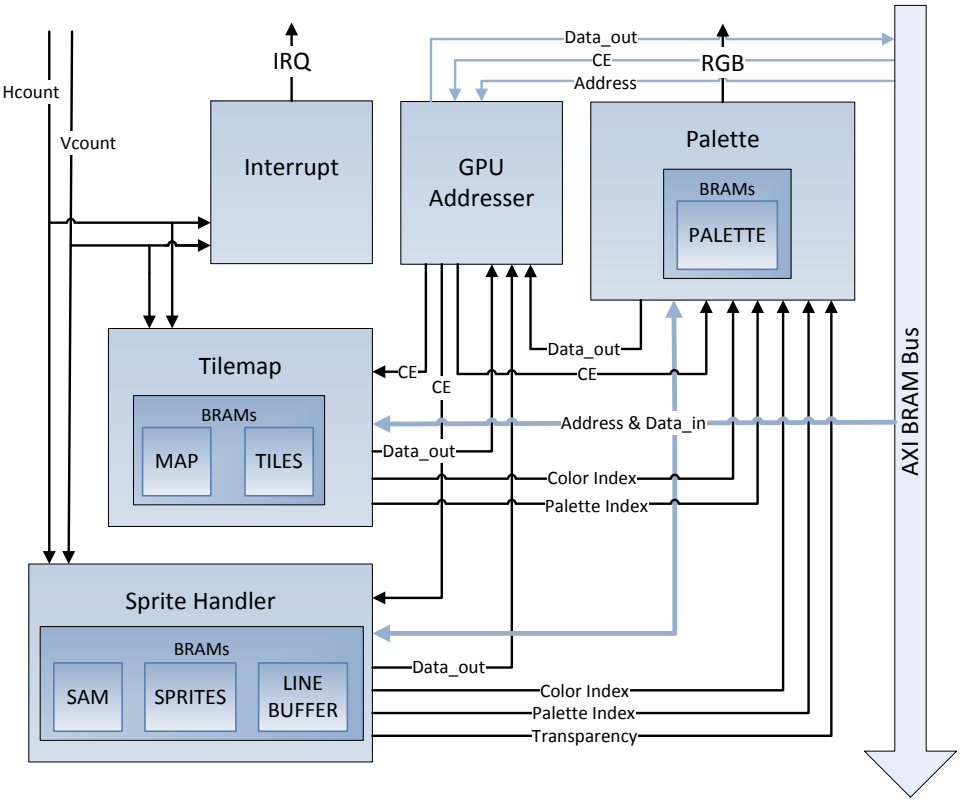

Figure 5: Overview of the GPU

4.2 Sound generator

A simple way to generate audio signals is to use pulse width modulation (PWM). It works by changing the duty-cycle of a square wave. When passed through a low-pass filter (or a speaker), the square wave will be integrated into an analog signal, if the duty-cycle is increased the analog signal will rise and if decreased the analog signal will fall. Audio encoded into raw (pulse code modulated) data only contains different duty-cycle values, that when modulated correctly will generate a good sounding audio wave. Figure 6 shows a simplified schematic of the sound generator. Internally, an address is being incremented at the sample rate, reading new sample data from the dual-port BRAM to the PWM component. This address will correctly overflow when reaching the last address of the BRAM so that it will never run out of data. Data samples are continuously being fed into the sound buffer by the MicroBlaze. When one half of the buffer is being sampled by the PWM component, the other is being updated by the CPU. To know which half to update, the msb of the cyclic sample address is wired to the data_out signal, available for polling at any time by the CPU.

On the PWM side of the generator, the data sample is stored in a register and updated at the sample frequency. To generate the square wave an 8-bit counter is incremented at the system clock frequency. At the start of the counter the output signal is set to high. The counter value is then compared to the sample

data and when the values are equal the output signal is set to low. When the counter reaches 256 it overflows and the output signal is once again set to high.

The sample frequency was chosen by dividing the system clock frequency with 10 times the resolution of the duty-cycle. This means that the `PWM_out` signal will switch exactly 10 times for every audio sample, this being thought to give a clearer more even sound. 8-bit data samples were used resulting in a resolution of 256 different duty-cycle widths and a sample frequency of about 39.1 KHz. This also means that two samples can be stored at each memory location, since the BRAMs are 18 bits wide.

The sound generator is connected to the host using an AXI BRAM controller. Reading from the sound controller returns a flag that indicates which buffer half is currently read by the core, to allow synchronization with software.

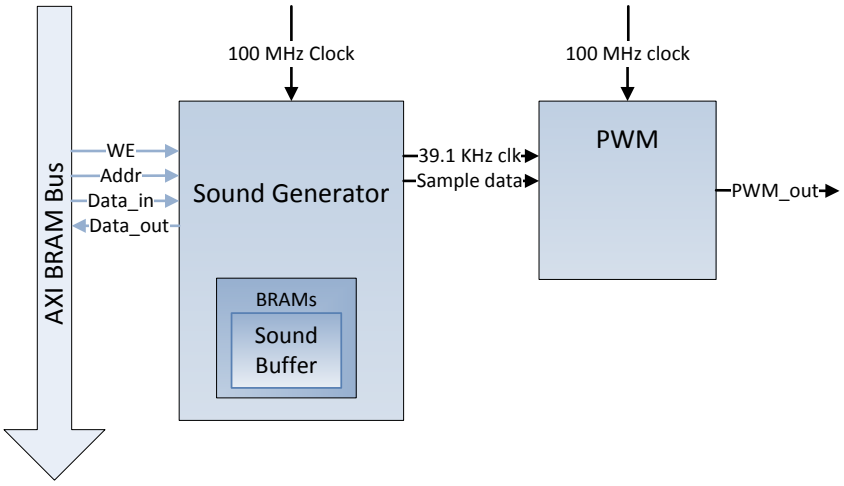

Figure 6: The sound generator

4.3 Gamepad interface

A NES gamepad was used for the input interface of the game. The NES gamepad was used back in 1985 for Nintendo's first game console. It has a four direction cross and 4 buttons (A, B, start, select as shown in Figure 7).

Figure 8 shows the pin-out of the game-pad. In this application, 5 out of 7 pins are used. `GND` and `VCC` provides power to the gamepad, whereas `PULSE` (serial clock), `LATCH` (asynchronous parallel load) and `DATA` (shift register output) are used for communication. The gamepad is designed to be run at 5V (being the native core voltage of the NES), but when tested with an oscilloscope and a pair of signal generators it works as well at 3.3V, which is the voltage supplied by the development board. This simplified the development of the interface by avoiding both level conversion of the IO and sourcing a higher supply voltage. For I/O on the development board four Pmod connectors where used.

Figure 7: Button layout of a NES gamepad

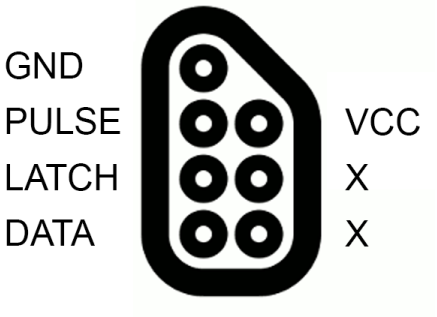

Figure 8: Pin-out of a NES gamepad cord

The gamepad has a parallel load shift register that loads the state of the buttons when `LATCH` is asserted. The 8-bit data word can then be clocked out using `PULSE`. The data signal from input comes in the order as shown in Figure 9. The latch and pulse signal is generated from the interface I/O core and the data is then stored in registers. The current state of the gamepad is always available as a memory mapped register on the AXI bus.

Four identical gamepad controller cores are instanciated in the system; one for each gamepad.

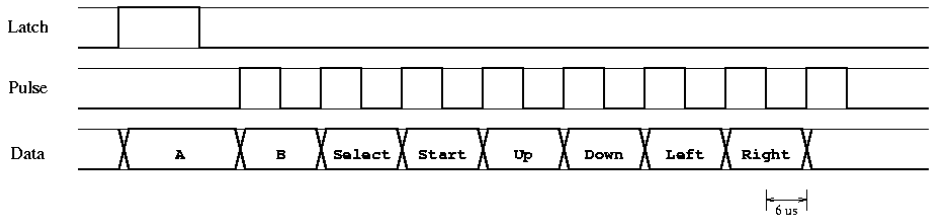

Figure 9: Serialized data output, latch and pulse signal in to gamepad

5 Software

The game logic was written in C, to be compiled using Xilinx SDK and run on a MicroBlaze soft core. The game is interfaced by two functions, `init_game`, called at startup to initialize the game state and download the graphics to the GPU, and `run_game`, invoked through the `vblank` interrupt at 60 Hz to update the game state.

The `run_game` function performs the following steps:

- read the gamepad inputs and store as a bitfield in memory,
- updates all bombs, including animation, calculating blast and detecting hit players,
- moving players, detecting collision agains walls and powerups,
- updating the scores and the time at the top of the screen,
- sorting all active sprites according to their Y coordinates, in order to produce an illusion of depth when players walks through bombs or other players,
- disabling unused sprite slots

All low-level routines was implemented in a separate module, called Hardware Abstraction Layer. The HAL is responsible for configuring the interrupt controller, reading gamepads, generating pseudo-random numbers and rendering sound.

The sound engine is run "in the background" when the vblank interrupt routine is not executing, thus implicitly prioritizing the game logic over filling the sound buffer. This is because the game logic update has to complete before the start of the active area of the next frame, to avoid visible glitches or tearing, whereas the sound buffer may be filled whenever during the video frame. Therefore the sound engine busy-waits when synchronizing to the sound hardware. The same behaviour could be implemented using a buffer-empty interrupt from the sound generator hardware and prioritized, nested interrupts in the MicroBlaze, but this was considered an overly complex solution for this application. The primary gain using such a solution would be lower power consumption, as the MicroBlaze could be halted while waiting for an interrupt.

The sound is rendered by simply mixing the PCM streams of the music and any active effects and writing it to the ring buffer in the sound generator. The streams are read directly from the SPI Flash, which is conveniently memory mapped using the execute-in-place (XIP) feature in the memory controller.

A large amount of non-volatile memory could have been saved if the music was instead generated using a sequencer and/or synth in software on the MicroBlaze, but this was deemed out of scope for this project.

To parallelize the workload during development, an emulator framework was written in C using `libSDL` which allowed the game logic to be compiled and tested on a PC. The emulator code replaces the HAL code used when targetting the real hardware. Even though this kind of emulation in general only applies at source code level, this emulator was written to make use of the same data structures as in the real sprite handler and tilemap memories.

Graphics data was generated using a purpose-build python application called `pixl8`, shown in figure 10. This allowed interactive design of bitmaps and palettes and generated C header files to be used directly in the application code.

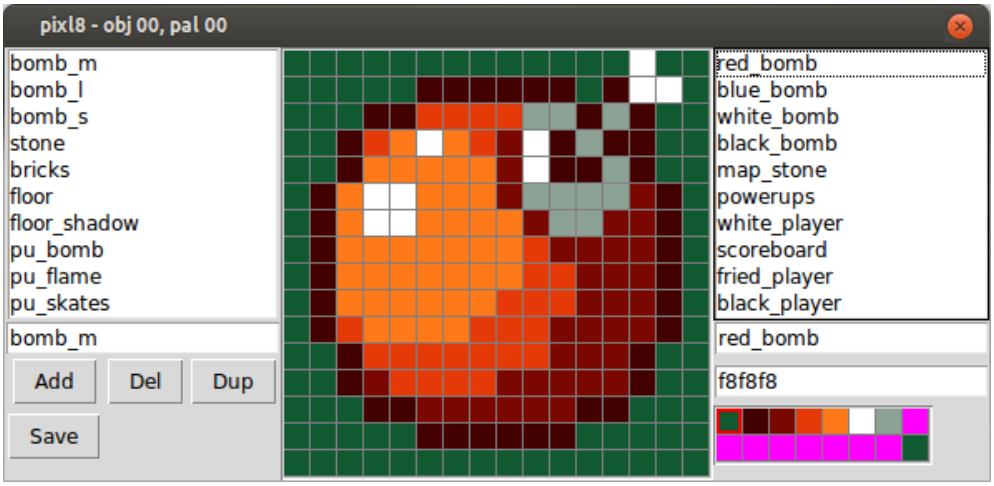

Figure 10: pixl8, a bitmap and palette editor

6 Installation

The VGA cable is connected to a monitor and the game-pad is connected to four Pmod connectors on the board. Where pin 12(`VCC`), pin 11(`GND`), pin 10(`CUP`), pin 4(`OUT0`) and pin 3(`D1`). Download the download.bit file using Digilent Adept tool. The controls of the game are as follow. "A" is used to place bombs and the direction cross to move the bomberman characters.

7 Problems and conclusions

The project was largely successful with the GPU and gamepad interface implemented as planned, as well as the sound generator, enabling music and sound effect, even though this was considered an optional bonus. The software was developed in time and performed good enough.

There were some problems with implementing and debugging the AXI bus interface of the custom IP cores, but those were resolved and did not affect the project. Some software debugging was performed on the MicroBlaze using XMD.

The overall development was straight-forward and successful, as a result of a well-designed architecture.

8 Contributions

- Xilinx software: Adam, Fredrik

- GPU: Adam, Fredrik, Magnus
- Gamepad I/F: Magnus
- Sound Generator: Adam, Magnus
- Game logic: Mostly Fredrik, Magnus did the powerups part
- Tools: Fredrik

9 References

1. NES Controller `http://www.mit.edu/~ tarvizo/nes-controller.html`
2. Digilent Adept `http://www.digilentinc.com/Products/Detail.cfm?Prod=ADEPT2`