

# ARM11 Project Extension - PiPong

Group 12 - Giacomo Guerri, Dylan Gape, Frances Tibble, Oliver Brown

June 12, 2015

## 1 Introduction

Pong is one of the oldest arcade games and its widespread popularity contributed to the emergence of the early video game industry. As a team, we thought it would make an interesting extension to our first year project. We were determined to learn more about what is behind a simple computer game and the development process required to create one.

## 2 PiPong

Much like the original game, our version of Pong is played between two human opponents whose aim is to prevent the ball from touching their side of the screen. The game is won by the first player to score 11 points. Each opponent has a controller with two buttons allowing him or her to move their bat vertically and to deflect the ball's trajectory. To make our game more appealing to the users, we decided to add a start screen to welcome the players and a “game over” screen to congratulate the winner.



Figure 1: The game start screen.

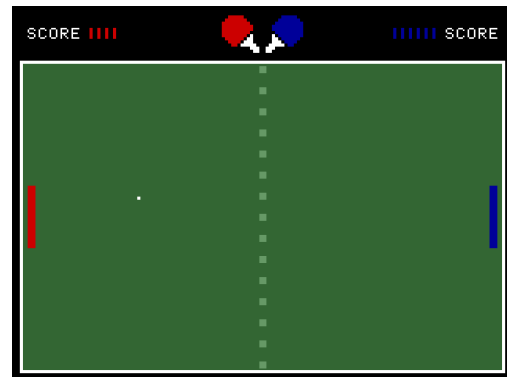


Figure 2: The game graphics.

PiPong can run on a Raspberry Pi without the aid of an existing operating system. The assembly instructions for the game are turned into binary code through our own ARM assembler and they replace the kernel of the Raspberry Pi communicating directly with the CPU and the GPU. In order to achieve this, we expanded our assembler with more instructions such as multiple transfer instructions and directives.

## 2.1 Expanding the Assembler

To facilitate our extension we had to make a large number of improvements to our assembler. Firstly we had to enable the calling of routines by adding the branch with link `bl` instruction. This sets the link register during a branch. This way, when we have finished executing a routine we can return to the same point by moving the link register back into the program counter register.

It became necessary to back up registers when many jumps to routines were nested. Initially we did it by backing up to higher registers, but this soon became unmanageable, requiring the implementation of a stack. We created a stack by reserving an area of memory using assembler directives, creating a label to the end of this area and then using load multiple `ldm` and store multiple `stm` instructions. This allowed us to store many registers to memory at once through the automatic incrementation `ldmia` or decrementation `stmdb` of the stack pointer `sp` register. Block transfer instructions were also necessary for copying the back buffer (described in section 2.2) to the GPU framebuffer.

To store data required before execution we used a `.4byte` and a `.space` directive. `4byte` simply inserts a word at the current position with a given value and `space` fills from the current position with a number of 0 words.

We had problems retrieving literal and global values during the extension. The large space occupied by the stack and framebuffer data structures meant that when literals were appended to the end of the file, their offset would be out of range. It was also tricky to determine the addresses of the symbols representing labels during the first pass, since assembler directives such as `.space` would have had to be parsed. For both of these reasons we thought it would simplify the task if we converted our assembler to take one pass over the data and to make use of the `.ltorg` directive. `ltorg` instructs the assembler to dump the current table of literals at the current position in the binary file. This meant we could write the literals in range and then fill space for our large buffers and the stack. To make a one pass assembler we adopted the concept of “fixups”. When the assembler encounters a forward reference to a label, or any literal, it adds a fixup record to either the forward reference table or the literal table with a function callback. Then, when the literal or label are defined, the callback is processed and the parts of the instruction that were impossible to assemble before are assembled.

We also added a number of extra data processing operations such as `mvn` and `bic`, although these could have been defined in terms of other instructions.

We adopted the following calling convention in our code to keep things standardised: any register greater than or equal to `r4` was required to be saved to the stack. Furthermore, all routines had to be well commented with an explanatory list of arguments and with a list of any registers between `r0` and `r4` overwritten by the routine.

## 2.2 Handling Output and Graphics

Getting the graphics for our game to work on the Pi was a tough challenge. There was little information about the BCM2835 frame buffer online. We found “Baking Pi” [1] and eLinux’s page on the Raspberry Pi framebuffer [2] to be very useful, however, the former was not updated and the eLinux article was a bit lacking in places, so a deal of trial and error was required.

The overall output process required writing to the “mailbox” and requesting the GPU to fill in a structure describing the frame buffer, including a field to hold a pointer to the array of pixels itself. We discovered that there were alignment constraints and that `config.txt` needed to be

carefully configured to make the GPU use the given resolution.

Once all this went to plan, we were be able to write directly into this array and have the GPU update the screen. This would work if our game rarely updated the screen, but due to the constant movement of the objects on the screen it was necessary to clear and redraw it with each iteration of the game loop. The result was a large amount of flickering and vertical tearing as the GPU was displaying the framebuffer between updates.

To solve this problem we had to add a “back” buffer and make use of double buffering. We reserve space for a display buffer in our binary file and clear and draw each frame in it. Then, only when all the drawing is finished, we copy the contents of this buffer to the GPU’s frame buffer. This eliminated all flickering and tearing. To speed up the process of copying and clearing buffers we used the block data transfer instructions we had already implemented. We adopted RGB565 16-bit colour for our project. Getting full 32 bit to work was temperamental at best and 24 bit was harder to optimise. Any images such as those on the start and game over screen, as well as the “Score” text were simply stored in the binary as a series of 4byte words and copied to the back buffer as required.

## 2.3 Handling User Input

GPIO, being considerably more well documented than graphics, was a lot easier to implement. The routine `init_pins` in `main.s` sets the pins connected to the buttons on the controllers as inputs. At this point, whenever we refresh the frame, we simply check for the state of each of the GPIO pins wired to the breadboard. The blue controller is plugged to GPIO pins 18 and 23 for up and down input respectively. The red controller is plugged to pins 24 and 25.

The wiring of the red PiPong controller is shown in Figure 3.

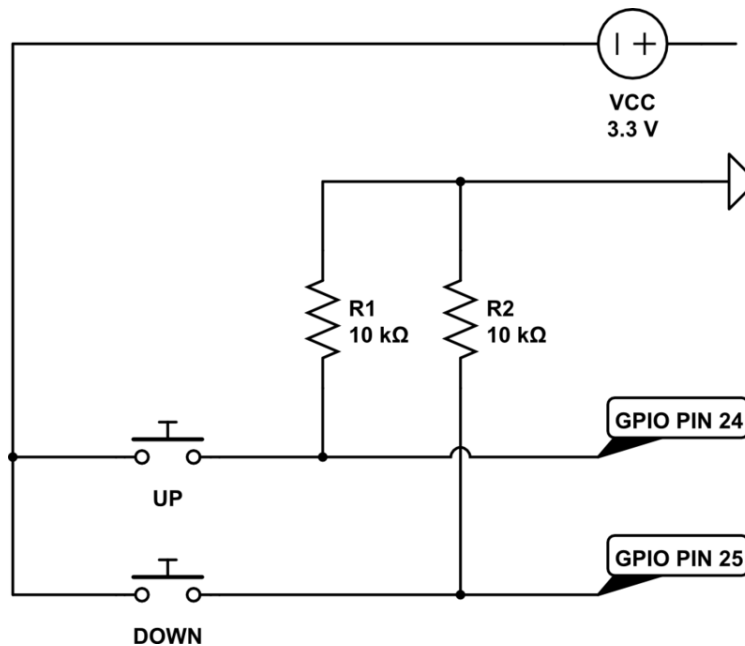


Figure 3: Circuit diagram of a PiPong controller.

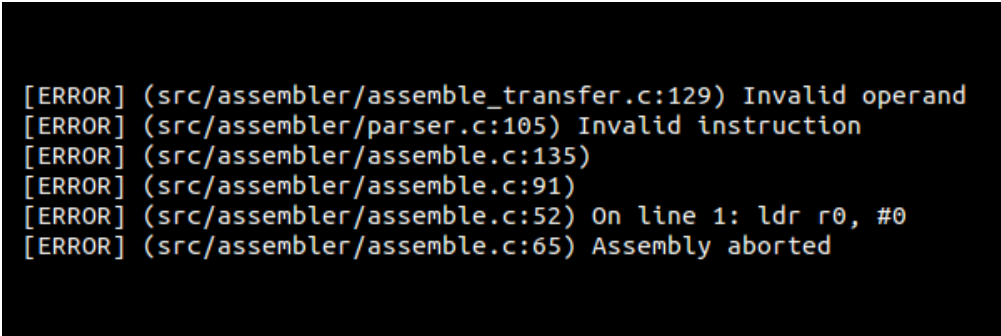
### 3 Testing and Debugging

An important part of monitoring the progress of our project was to make use of various types of automated testing. Initially, we set up a test server hosting the main test suite for this project and included a deploy utility to fetch and compile the latest version of our code base. This gave everyone easy access to the test suite and proved to be sufficient while building the emulator, as the tests output provided insight into all key areas for debugging.

When starting work on the assembler, we decided that we needed additional testing. As the assembler is built up of many modules, we needed to be able to test individual components, so that it would be easier to debug the assembler once everything had come together. We used a unit testing library known as Seatest which, due to its portability, allowed us to have consistent testing and easy access by including an option in `Makefile`.

Since we were lacking in an OS kernel and our emulator could not emulate the nuances of the Pi's GPU, debugging our extension was difficult. Trying to figure out what the GPU was actually giving back via the mailbox was a problem as we did not have tools like gdb or even printing values to a console. To get around this problem we developed a routine that could output a number from a register as a binary sequence of LED flashes. We had one LED representing 1 and another 0. Both LEDs turning on meant the end of the number sequence. We found this to be a rather offbeat yet surprisingly helpful way of debugging.

Another important issue we had to address as a team was error handling. We created a file called `error.h` which could be accessed anywhere in our code. The file contains useful functions to output error messages, warnings and simple informative messages. The function `check`, for example, accepts a condition and if false, prints an error message together with the name of the file which called it and the line number. This not only allowed us to handle errors nicely and discreetly in our code, avoiding duplication, but it also turned out to be an extremely useful debugging tool.



```
[ERROR] (src/assembler/assemble_transfer.c:129) Invalid operand
[ERROR] (src/assembler/parser.c:105) Invalid instruction
[ERROR] (src/assembler/assemble.c:135)
[ERROR] (src/assembler/assemble.c:91)
[ERROR] (src/assembler/assemble.c:52) On line 1: ldr r0, #0
[ERROR] (src/assembler/assemble.c:65) Assembly aborted
```

Figure 4: Example of error handling from the assembler.

## 4 Coordinating the Team Work

Working on a code base is difficult if it is not well organised. We established personal Git branches to easily review each other's files before approving merges to master. In order to make our code as clear as possible to everyone in the team, but also to everybody else, we added documentation using Doxygen, a helpful tool which works automatically from the source code. This improved the congruency of our code style and also our understanding of other team members contributions, since the purpose of the files and the functions within them were clearly commented.

Communication was frequent and effective, thanks to online messaging services to reach every team member on their computer or mobile. Working issues could be resolved quickly, while problems that were not easily addressable online were usually discussed during group meetings in the labs. These included experimenting with the Raspberry Pi hardware or with the only assembly file, where working individually was impractical. On reflection, our methods were suitable for the task we were working on.

We discussed as a group how to divide the work between us, so that each task was slightly challenging yet feasible for the team member it was assigned to. At any time the whole team was available to help clarify a question or to listen a suggestion. At the beginning we implemented only what the specification required. If we had had the foresight, we could have altered the structure of our initial implementation based on how we might have later extended the assembler to accommodate the game. Our main priority was to get everything working first and to an extent we did not know what the later requirements would have been.

In summary, for this small project, restructuring our work did not pose much of an issue. However, if we were to work on a larger project in the future, we would create a more considered project plan, as it would be far more practical to manage.

## 5 Reflections on the Project

### Giacomo Guerri

The project was not a big one, but being the team leader was tricky nonetheless. It was very important for me to know my team mates well, their working styles, their strengths and weaknesses, as well as my own. For example, I was aware not to be the strongest team member from the point of view of technical knowledge. I decided to let everyone, and Oliver in particular, help with making the right design decisions. It turned out to be a good strategy not only because it allowed me to learn a lot, but also because it created a productive environment, where nothing was in the way of good ideas. Receiving positive feedback, and seeing the great progress that we were making as a group, made me realise that the team was on the right path and that I just had to keep doing my best to collaborate and support them.

### Dylan Gape

As we had worked as a group before, I knew that we would be able to work well together. I was able to complete all the tasks given to me in a timely manner, and to a good standard, as said in my WebPA feedback. My experience with maintaining servers proved useful in setting up the test server and I was able to learn more about Git. Throughout the project, I developed a greater understanding of programming in C, and was able to learn about programming in a group. For

the next project, I would improve my sleeping pattern, as I was often starting to work later than other group members and working later into the night.

## Frances Tibble

I felt very comfortable in the group; like my team members, I am very hard working and dedicated to tasks I am set. This was recognised in my feedback, stating that I have made great contributions and been a “helpful group member”. Initially I was concerned that my programming skills, being weaker compared to my team mates’, would result in being unable to contribute as equally to the code base as I would like. On receiving feedback, I realised I was perhaps overly critical of myself, as my group were satisfied with how I was working and that my tasks had been completed “to a high standard”. In addition, I discovered new qualities that I will try to utilise in future projects, which include code documentation and idea generation.

## Oliver Brown

I was very lucky to have cooperated with Giacomo, Frances and Dylan during our group project. They were all hard working and came at the project with a great attitude. The group benefited most from my technical ability, Giacomo set overall tasks but I was responsible for dividing up tasks further in the development of the assembler and emulator.

This was a good strategy since it helps to have a leader who can step back, see the bigger picture and tell the group when they think something is wrong.

Even though the task was comparatively small to a large software project I feel I have learnt a lot about the software development lifecycle. We definitely followed an agile development method, keeping in good communication and quickly adapting to our own feedback, regularly refactoring code as well as frequently merging all our branches with the master branch on Git.

However, if I were to do the project again I feel I would spend more time gaining a deeper understanding of the problem, for example reading more about ARM assembly and how assemblers work before getting stuck in as there were a few occasions, particularly with regard to the extension, where we had to backtrack and make changes. Receiving feedback was gratifying but a little unnecessary as we were all in close communication at all times and welcomed constructive criticism.

## References

- [1] Alex Chadwick, *Baking Pi — Operating Systems Development*, University of Cambridge, Computer Laboratory
- [2] RPi Framebuffer, *RPi Framebuffer*, Online source: [http://elinux.org/RPi\\_Framebuffer](http://elinux.org/RPi_Framebuffer), accessed on 12th June 2015