Assembly Chess

Xu Ji, Bora Mollamustafaoglu & Gun Pinyo

June 14, 2013

1 Introduction

The game of chess dates back to the 6th century in South-East Asia. Originally played with ceramic and resin, we thought it would be interesting to use modern tools to see if we could code it for the digital world using the Raspberry Pi.

2 Coordinating Our Work & Team Dynamics

We were aware from the inception of our idea that its success depended on having enough time, and consequently we decided to get the assembler and emulator finished as early as possible. Early on, we decided to use the games room inside our hall to work in, because it was usually empty, had a large HDMI screen to test on, and was geographically close to each of us. Our strategy for the compulsory part was simply working full time on the project, whilst thinking ahead about what equipment we needed for the extension. We tried to be resourceful in this respect. For example, we used the HDMI television screens inside our halls of residence to test our display, and we did not even need to purchase HDMI-HDMI cables since we found televisions that already had them attached. When we realised we needed extra parts (primarily buttons and extra wires and resistors) we did some research on prices and split our orders into the parts which could be found on RS and Maplin, and the parts that could only be found on Amazon (the buttons), before sending the former to Tristan and buying the latter ourselves. This meant we could minimize our spending on components.

Before we decided on our extension, we had been optimistic about our timing, and indulged in splitting to consider different ways of implementing emulate. From assemble onwards, we both gelled as a team and became more focused. We decided we needed to delegate our programming better, so our strategy shifted from allowing each person to consider different implementations of code, to deciding on the specification for the code beforehand, as a team, before delegating each other to write different sections of it. We found that merging the contributions from each team member into the final result, as opposed to choosing one person's code and trying to bring everyone else's code into it, resulted in completing the code in much less time.

We acknowledged our different backgrounds and tried to take advantage of our different skills - we were all proficient in programming C for the assembler and emulator, but parts of the assembly chess were more suited to some than others. Gun had developed games before and had the most experience in assembly, so it was decided that he should work on the game logic. Xu, who liked debugging and tackling new ground, was chosen to focus on the display. Bora, who came up with our image compression scheme, was chosen to write the C program dcd_gen and became responsible for graphics, including the storage of the piece templates using assembly directives, and writing draw_square(), which is used to generate square images on the screen.

Despite focusing on the areas which most appealed to each of us, we also made sure to go over each other's work and help with other sections of the code. Not only did this mean we would cross-check and pick up mistakes that had been missed, but also that we were each given the opportunity to learn about all aspects of game development in assembly. For example, we debugged Gun's game logic with our graphics, and Xu wrote the promote_pawn() method, which is called directly by the game logic.

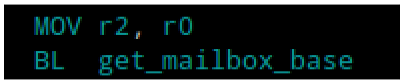
We noticed that our teamwork started off relatively fragmented and became more cohesive as the code grew. This was partly out of necessity - having completed our separate parts we needed to make sure they came together - but mostly because as our game became more complex we increasingly had to consult each other's knowledge on design details and bugs. The three of us started coding on different laptops, since we were at our most productive when all three of us were actually coding and early on in the project producing functioning code was our greatest priority. Later on when we were merging all of our code together, it became necessary to use only one laptop to do the merging. Xu and Bora also found they worked well together, and discovered that pair programming was highly effective whilst working on the display methods. Since we were often sitting at the same place for hours each day, pair programming meant that when one person's concentration drifted, the other could re-establish focus.

In a small team of three, we did not feel the need to have a set leader role, preferring to discuss everything and make decisions together. We feel that this has really worked well as each team member felt free to contribute and engaged equally with the project. It also meant the workload was fairly split between the three of us. As we bonded as a team our communication increased - we used all channels including phone, email and social media - and we found ourselves encouraging each other to focus and implement extensions. The greatest motivation for our progress was not moving chess pieces around in digital space - it was seeing our hard work actually come together and producing output on the screen.

3 Extensions to the Assembler & Coding Practice

To support a program of this scale and scope, we had to extend many aspects of our assembler:

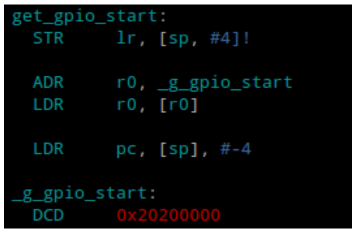
- Method calling was vital, so we decided to add support for the Branch with Link instruction.



```
MOV r2, r0
BL  get_mailbox_base
```

Figure 1: Calling a method

- In order to save registers when calling methods, and to take full advantage of what we had already implemented, we decided to use our single data transfer instruction to push and pop registers. As we had fully implemented single data transfer, enabling the writeback bit to be set, there was no need to spend time implementing block data transfer for this purpose.
- Fully implementing single data transfer meant we could easily use a stack. The stack pointer is initialised in init_pins() to the start of a block of 1KB. This block resides below the declarations of all the global arrays that correspond to our chess pieces.
- We thought for a long time about how to implement global variables, which we use heavily within our program for both game constants and graphics. We could not store them in arbitrary locations, since the values would have been interpreted as instructions and executed.
  - We decided to support the DCD and ADR directives, as this was closest to real ARM assembly. DCD allowed us to store words at specific locations in memory, and ADR allowed us to load the physical address of any label into a register.
  - However, after watching our program crash, we realised that there must be a limitation on the ADR instruction, as the offset from the program counter must fit inside the 8 bit second operand for data processing instructions. This means in general there could only be an offset of 255 bytes between the program counter at the time of the ADR instruction and the variable itself. Our solution to this was to have setters and getters for global variables located next to the variables themselves, since functions can be called without restrictions in distance.
  - Instead of dotting them around the program, we strategically located our global variables at the end of our code, with the exception of the GPIO buffer data, which has a special alignment requirement and hence is placed at instruction number 0x10.



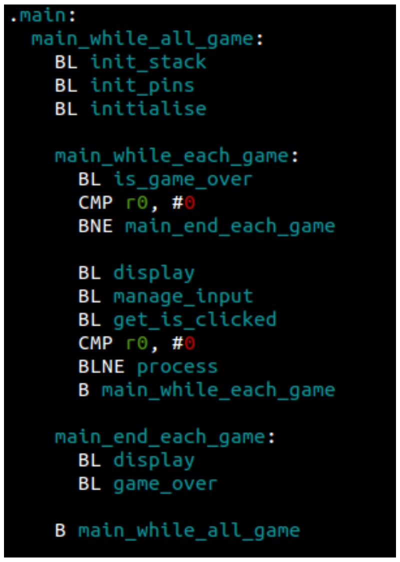
```
get_gpio_start:
  STR     lr, [sp, #4]!

  ADR     r0, _g_gpio_start
  LDR     r0, [r0]

  LDR     pc, [sp], #-4

_g_gpio_start:
  DCD     0x20200000
```

Figure 2: A "getter" function that demonstrates using STR, LDR to push and pop registers prior to fetching a global variable.

- The full range of conditions on all mnemonics is supported, and not just the conditions specified on the branch instruction. This would enable more concise code at little extra cost. We also wrote a full implementation of the shifter, allowing for register shifts, and added full support for comments and labels, which we implemented by parsing for semicolons and colons. Working in a language as low level as assembly meant comments were vital in annotating our code so we could understand it later.
- To improve the readability of our code when dealing with immediates, we decided to use characters (for example 'K' for King, 'Q' for Queen) to represent the piece types in our assembly. Hence we added support for ASCII character immediates. Each piece type intuitively corresponds to a different character, reducing the likelihood of programmer errors when dealing with pieces.
- Anticipating that the code was going to get messy very quickly, we established several protocols to standardise our program and make the code more elegant and well organised. We established protocols for pushing and popping registers, deciding to use r0 for passing arguments to functions, and protocols for calling methods. We also decided that all our global variables would start with _g_, and all subroutines within methods would begin with the method name, since we anticipated that this would avoid conflicts and confusion when calling methods.

4 Game loop

The skeleton of our game is the .main method, which is a standard game loop implemented in assembly. After we display our board, manage_input() loops until an input is detected from the player. The boolean _g_is_clicked reflects whether the player has pressed the select button and determines if the input needs to be processed by the game logic method process(). This continues until checkmate occurs, at which point a game over message is displayed, and the game resets by entering a new game.



```
.main:
  main_while_all_game:
    BL init_stack
    BL init_pins
    BL initialise

    main_while_each_game:
      BL is_game_over
      CMP r0, #0
      BNE main_end_each_game

      BL display
      BL manage_input
      BL get_is_clicked
      CMP r0, #0
      BLNE process
      B main_while_each_game

    main_end_each_game:
      BL display
      BL game_over

    B main_while_all_game
```

Figure 3: Our game loop

5 Input

We really wanted to avoid using third party code, hence we aimed from the start to write everything by ourselves. We have stuck true to our resolution. So instead of using USB drivers (or in fact any third party code at all) the input from the player is obtained by reading from GPIO pins which are wired up to momentary push buttons.

After much discussion, we decided the optimal input scheme would be to have six buttons. Four of these are directional (left, down, up, right) to allow cursor movement, the fifth is a select button for making moves, and the sixth is a reset button to restart the game. We use a 10k Ohm resistor in the circuit to avoid pulling too much current when the button is pressed. To ensure the safety of our Pi board, we also added a 1k Ohm resistor before the GPIO pin, to prevent the risk of short-circuiting the board if we accidentally set a pin to be an output pin.

During our testing and debugging of the game, we realised that the length of an average single click of a button set the GPIO pin three times, causing the current position on the board to move three squares rather than one. We solved this by writing a wait function that delays the detection of input from the player.

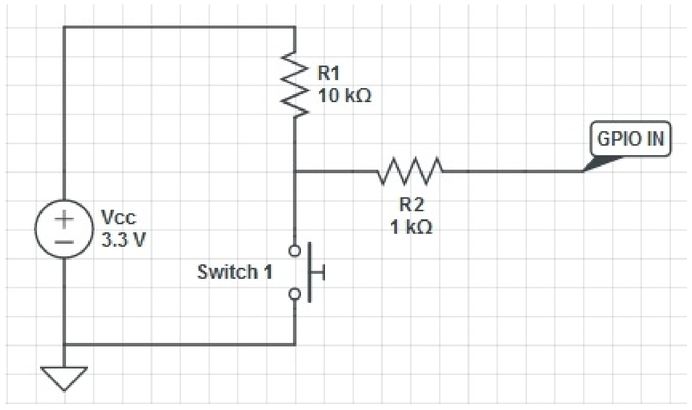


Figure 4: GPIO pin circuit (eLinux).

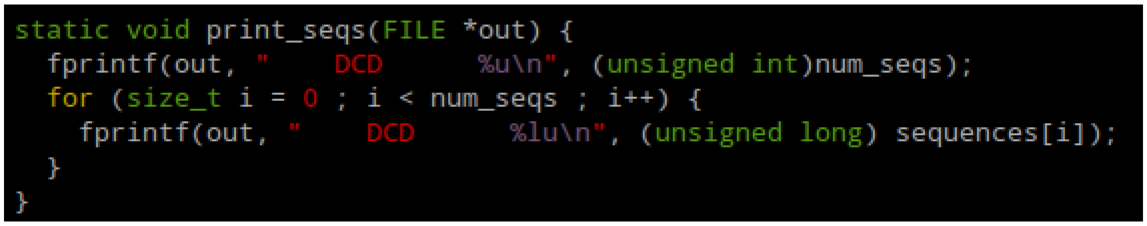
6 Graphics

A key decision we made was how to store the images we were going to print on the screen, since we had to draw Kings, Queens, Bishops, Knights, Rooks and Pawns. The most efficient implementation would have been to have a function which could draw each piece, but due to the non-linear shapes that make up the pieces, we quickly decided this was infeasible and we would need to store the images on some kind of pixel-by-pixel basis.

However, upon calculating how much memory we would need if we stored one pixel per memory word, we quickly realised that this was too inefficient. As we could not do halfword loads and stores from memory, we chose 32 bit depth for our colours for convenience. At 80 * 80 pixels per square our six different piece types would have resulted in 38,400 lines of memory allocation directives.

We thought that there must be a way to decrease this. Observing that each image was made up of long sequences of pixels with the same colour, we chose to apply a basic compression scheme: for a given sequence, the number of pixels is encoded along with the colour in a single word. The bottom 2 bits give the colour, which can be one of 'background', 'piece', or 'outline', and the remaining 30 bits represent the number of pixels. This would decrease dramatically the directives to just over 1,500.

As we did not really want to write 1,500 lines of just directives either, we decided to automate it. We wrote a small C program (dcd_gen) to apply this compression to the 'ASCII-art' output (from a photo editor) of our piece images, in order to reduce the complexity in both reading and manually editing the images. Although we had to spend some time writing this tool, it was worth it as we could now automatically produce compressed images that were ready to use in our program almost instantaneously. This meant that any future changes to the images of the pieces would be easy to incorporate.



```
static void print_seqs(FILE *out) {
  fprintf(out, "    DCD      %u\n", (unsigned int)num_seqs);
  for (size_t i = 0 ; i < num_seqs ; i++) {
    fprintf(out, "    DCD      %lu\n", (unsigned long) sequences[i]);
  }
}
```

Figure 5: A helper method within dcd_gen.c, a tool that automatically takes ASCII art and compresses it directly to assembly directives.

7 Output

The most difficult part of the I/O was displaying our chess board on the screen. The first issue we encountered was simply displaying anything on the screen. Alex Chadwick's Baking Pi tutorial had given us an insight into the overall process - writing a block of data including the frame buffer width, height, and colour pitch, into a specific mailbox address, receiving the pointer to the frame buffer returned by the GPU, and checking for errors and non-cooperation from the GPU at every stage.

However, it seemed that no matter how much we looked at our function set_frame_buffer(), we could not get it to retrieve the framebuffer. Our first breakthrough was realising the kernel was loaded at address 0x00008000, and not at zero like we had assumed. This was due to a revision by the manufacturers, and we discovered that could be reverted by including the flag kernel_old=1 inside the config.txt file.

This mattered because we knew the address of our message to the GPIO had to contain zeros in the bottom 4 bits, due to the fact that it would be bitwise or-ed with the mailbox number we were writing to before we wrote it to the mailbox. If the bottom four bits were not zero, we would not be able to tell the GPU the number of the mailbox we wanted to write to. In the end our solution was to make sure our block of data was allocated at the start of our kernel, and manually pad the lines before it with 15 instructions to ensure zeros in the last 4 bits.

Another breakthrough was discovering the importance of the config.txt file. We came across an article on eLinux that gave a complete list of the flags that could be set inside this file, which gave us the option to select resolutions, compatability with different display standards (CEA, DMT). Through experimentation with the HDMI display, we realised that vWidth and vHeight within our data block (the fields that Alex Chadwick "had no idea" about) corresponded to the virtual width and height of our display, as opposed to the Width and Height, which corresponded to the actual screen resolution.

We were very excited after we managed to draw plain colour and lines on the screen by writing directly to the GPU buffer. In case that did not work, we had several backup plans, including writing to our own pixel buffer before copying it serially to the framebuffer, and using LCDs instead of HDMI. Even though we eventually completed everything, this practice was useful as it meant we were less stressed about our bugs and ensured that we would have options if the time restraint meant we had to move on.

The solution to getting the display to work seemed to be just extensive research and pure persistence.

8 Gameplay, Testing & Example

Chess is a complicated game, and writing it in assembly proved to be a challenging task. In order to avoid the difficult prospect of debugging the chess game using just the LED (as the graphics were not finished), we decided to write a prototype of the game in C first to eradicate semantic errors. This was worth it as it saved time and effort debugging the code later. We then straightforwardly rewrote the code in assembly, focusing on making sure the syntax was correct as we did not have to think about the game at the lowest level. The translation process taught us how structures in C could be conveyed in assembly, including if-else, getters and setters, switch statements and arrays.

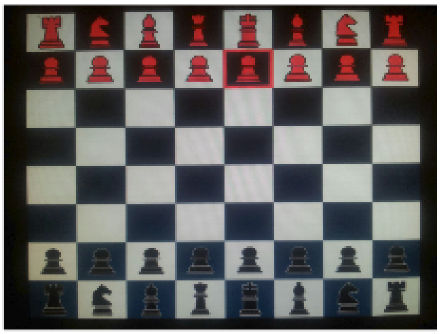
The easiest code to test was undoubtedly our compression program written in C, since we had the tools within our IDE Eclipse available to us. When debugging our assembly, it was useful to strategically trigger the LED light to determine where the flow of the code was going. After the game was properly displaying on the screen, we found that simply playing it many times unearthed most of the bugs.

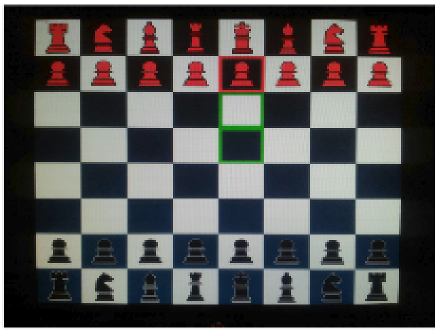
Example Chess Gameplay

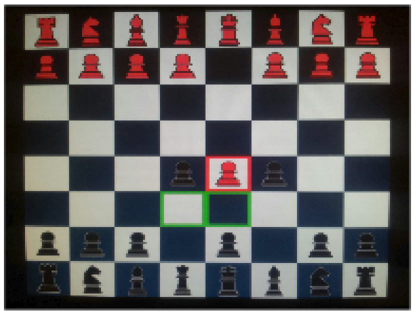
Possible moves are highlighted in green for every piece once it has been selected.

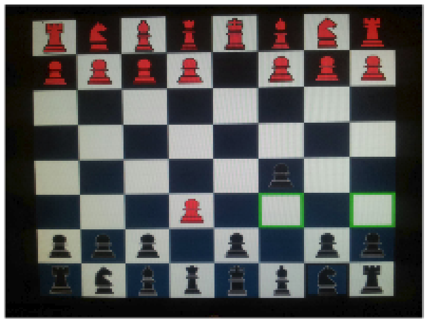
The border of the currently hovered-over piece is the colour of the player - either red or black.

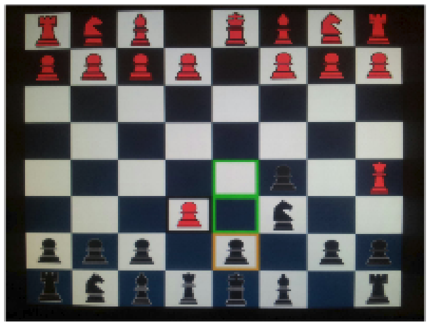
The last piece that has been selected is bordered by gold. (The hover colour takes precedence.)

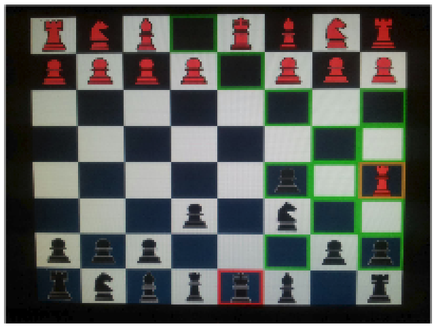

1. The starting screen. First to play is Red.


2. Red pawn is advancing.


3. En passant.


4. The black knight is about to advance.


5. The black pawn is about to take the red pawn. The border around the red pawn symbolises the cursor of the black player.


6. Possible moves for the red queen. We know she is about to take the black king, as red's cursor is hovering over him.

7. Game over screen. The background corresponds to the winner's colour.

9 Summary

Because our project required knowing the ARM architecture beyond what we had studied in class, we used online resources (such as the ARM documentation, BCM2835 ARM Peripherals manual, Alex Chadwick's Baking Pi, Raspberry Pi forums and Archlinux/ eLinux) extensively.

Predictably, we faced many problems of different kinds when implementing the game. The main problem with I/O was nobody in our knowledge had written a game in assembly without an operating system before, and we were often searching the internet for hours to work out, for example, why the GPU was not giving us a valid framebuffer pointer. It did not help that the Broadcom peripherals manual was filled with little errors, and Baking Pi was written for the previous generation Pi. The graphics required using various software simultaneously - GIMP, gedit, vim and our own dcd_gen - in addition to an artistic eye for detail. On the other hand, the chess logic itself was difficult to implement simply due to the complexity of the game, its many special moves and the fact that we were doing it all in low level assembly.

We had to think ahead at every stage in the development process, from specifying the code together beforehand to thinking about the electronic parts we had to order. We could have extended the language to be a perfect replica of the actual ARM assembler, but as this would give us no time to implement our actual chess game, we had to make trade offs and work out the functionality of the assembler that we actually needed. As we were aiming for an entertaining and enjoyable game, we had to think not only about the elegance of the code but also user experience. We enjoyed the unexpected fun of being creative and coming up with the colour schemes for pieces, players, background and selected squares. We have more ideas in mind, and we are currently considering further extensions including pawn promotion.

We started the project expecting to improve our understanding of the ARM7TDMI, Broadcom BCM283 SoC and the ARM assembler. We came out with much more than we had anticipated - despite venturing into a very low level realm we have managed to successfully implement a high level game. Working constantly with each other for many hours every day, we have bonded as a team and learnt the benefits of forward planning and good communication. We used the opportunity to build up these teamwork skills in the first half of the project whilst working on the emulator and the assembler, and found them to be instrumental whilst developing the extension. We are elated that we have actually coded up our idea and ended up with a working game of chess capable of running on a machine without any operating system.