RadIceCream: A RISC-V Assembly Game

Nirva Neves de Macedo Rodrigo Rafik Menezes de Moraes Mariana Simion dos Santos

University of Brasilia

0 Abstract

This scientific paper describes the game RadIceCream, created by Computer Science undergraduates Nirva Neves de Macedo, Rodrigo Rafik Menezes de Moraes and Mariana Simion dos Santos, inspired by the previously Flash and nowadays HTML5 game BadIceCream by publisher and developer Nitrome. The game was developed in Assembly Language for RISC-V processors. More specifically, it was developed utilising the RISC-V Assembly Runtime Simulator (From this point on, referred to simply as "RARS") as an Integrated Development Environment, in conjunction with the tool FPGRARS – a far less glitchy version of RARS itself that can run assembly code much more quickly. Together, these utilities made the project possible.

1 Introduction

BadIceCream is a browser game released in 2010 by the publisher Nitrome. While it is a very simple game, recreating this game in assembly language proved itself a quite extreme challenge. Much of the game had to be simplified to fit the schedule, however this does not mean there is no polish in the game's assembly implementation. In fact, despite the enormous codebase, when well analysed most of the assembly "functions" in RadIceCream are indeed simple and are made to appear, as best as it could be done, organised.

RadIceCream is thus not an exact recreation – one could say it only recreates the original game's mechanics, but in no way its art and style. In some areas, simpler art and style proved itself easier to implement. As such, this game attempts to subvert the original's concept in a certain way, stylized much more ironically and presenting subtle satire, with the goal of simply adding to the fun factor. Moreover, the more complex enemies had to be disregarded in favour of possibly entirely new enemies (albeit not *that* exciting).

Beyond just the aforementioned tools, it should be mentioned that a couple of other key tools – which could have been alternatives – were vital for a much quicker development process. Mainly, paint.net, which is an image manipulation program that was widely used for the creation of sprites, backgrounds and other graphical elements; bmp2oac3.exe, which is a tool for conversion of bitmap image files into data files that can be accessed by the two runtime simulators used through a rendering function; Notepad++, which was used extensively as a text editor for data matrices and other data files with structures that were purpose-built for the game, given that it is a far more sophisticated editor than the simplistic one that comes built into RARS. Finally, GitHub and Git were always used after the first few functions were written and settled to keep track of changes and allow easier contribution by the project's authors.

The file structure of the project is rather simple, being mainly broken down into three parts, merely for organisation. The main code is contained within the main folder, as well as the changelog and some of the tools used for development (not to mention this document itself also shares that space). Within the main folder are also subfolders, which are all accessed by the program during runtime. They are two: The sprites folder, which is self-explanatory, and the maps folder, which contains the matrices for each level, their backgrounds and other backgrounds used throughout the game, as well as other level information files. All of these will be explained in detail throughout the following section.

2 Methodology

This section will describe most of the development process. While a chronological description of that process would be most interesting, the authors have elected that, ultimately, the section should be organised according to the relevance of the problems that were faced during development and their subsequent solutions, as well as the order in which they appear in the codebase, as that provides much deeper insight as to how the game performs certain things such as loading a level or rendering something on the screen. Simpler issues aren't explained as thoroughly.

And, to briefly explain the choice of FPGRARS over RARS as our runtime: Some functions simply didn't seem to work properly under RARS. Specifically, the background render function for some entirely unknown reason was never being run - or rather, it was being run but its effects weren't shown, which was probably a glitch in the simulator's display itself. FPGRARS ran the function normally. Moreover, FPGRARS is *much* faster than RARS. Ideally, instructions should be run instantly, as that gives us more control over the time the program takes to run (Through the "sleep" System Call). Instant isn't realistic, but near-instant is provided by FPGRARS. However, since all things in life aren't perfect, FPGRARS' thread eventually crashes after too many system calls. In practice, the game crashes after about six minutes of runtime - this is not at all the game's fault but the simulator's. Were it not for this glitch specifically, the game would run flawlessly.

2.1 Modularity

The first, and perhaps most important aspect of the codebase is its modularity. Modular here defined as something that is easily modified and incremented. The code is written in such a way that adding a new function does not break others (or, if it breaks them, it's still a very easily fixed problem). To achieve this, the order in which functions appear, the way loops are built and the way the data section is organised were thoroughly studied. Modularity was specifically important for the continued addition of features up to completion, as well as the subsequent changing of features that required changing for other features to work. In summary, it is an attempt to make the program as linear as possible – algorithm followed by algorithm such that one causes as little interference as possible in the other. This was made particularly difficult by actions and functions involving tickrates, as all functions must act in synchronicity with the tickrate. However, tickrate changes are mostly only applied to the in-game enemies and the visible clock. Moreover, modularity implies the easy implementation of "higher level" game features like levels. Adding a new level is as easy as creating the relevant level files and including the files in the code, as well as using the levelLoader to load it (which are about 7 lines of code added to main.s that don't affect anything else in the slightest.). So, with modularity, comes the infrastructure to build a much more complete game.

As such, the infrastructure of the game was designed first, with the planned additions in mind. A coherent and consistent infrastructure made the implementation of features towards the end of the project far easier. This trade-off is worth it: while we initially moved a lot slower, working on things that the player does not really interact with like rendering functions and matrix manipulation algorithms, or even loading systems, this in turn made the addition of features down the line speed up exponentially. Imagine if we had taken a different approach, focusing solely on adding feature after feature without a real plan. The result would be an unfinished game, as instead of making implementation exponentially easier, it would have made it exponentially harder, as every added function would require debugging the entire code.

2.2 Memory Usage

It is *impossible* (never say never, I suppose) to design such a game without using memory extensively. There is an immense amount of information that needs to be taken into account by the game's code to get it going. If we only ever use registers, we effectively only have 120 bytes of memory available (this is without even considering that x0 is hardwired to 0). Just a single level already occupies more than twice as much as that. Not only that, but the exclusive use of registers brings forth issues with code readability. So, the necessity of memory usage is obvious. While we still have to comment what kind of value a register is assuming when it's not loading something from memory, the usual case is a register assuming a memory value, modifying it or using it accordingly, and storing it back in memory at the same location (or at another relevant location). That comes with the advantage of making it clear, within the line of code itself, what kind of value that register is getting (e.g. "lw t0, playerEnergy" - it's clear what value it's assuming, so it doesn't have to be commented).

How exactly memory is used in this game can be divided into two main categories. One is in regard to memory values that are used and modified in the game loop or during runtime in general. These can be further divided into simple variables that occupy one word or one byte and have a mnemonic label attached to them, arrays and lists which usually contain groups of values, each group of values representing an identifiable object (The most prominent example would be values pertaining to enemies - enemies have an identifying number and, for each enemy, there is a position. There is, thus, an array of X, Y coordinates, each coordinate represented by one byte. If, in total, there are 8 enemy IDs, the array storing enemy positions is 16 bytes long), and finally big, reserved spaces that are used to create copies of files that should not be modified while the game runs. For example, the level matrices. When you load a level and start playing it, the level matrix you are modifying is actually a temporary copy of the original level matrix created by the levelLoader function and saved in the currentLevel memory address label. This is necessary because, say, if we modified the original matrix instead, as long as we don't close the game, that level's state will never be reset. That means if one dies while playing the level, he may not load it again as it was when the level started. In simpler terms, you wouldn't be able to restart levels without closing and reopening the game. The currentLevel label reserves a 308-byte space, which is the same exact size of any original level matrix. The other category is simply composed of the game files that are included in the code with the .include directive. These files should (and can only be) be edited and accessed externally to the main code.

While level backgrounds are never modified, they too have a copy that is created every time you load a level. This makes the code easier to read, and more elegant in general, since any time the background has to be rendered again, whatever function is doing it simply has to refer to levelBackground instead of a specific level background label that is dependent on the level you are currently playing. This costs a lot of memory, however: the reserved space for a level background alone is approximately 77 kilobytes of memory.

2.3 Rendering

We may now talk about the actual functions written in the main code. Possibly the second most important of those functions is the displayPrint function, present at the very end of the code. It is one of the only functions that get actually "called" and returned from. This function is the basis of any visuals in the game - anything and everything that is rendered is rendered through it. The way it works is rather simple, at least in comparison to other algorithms that will soon be presented. It takes four arguments: the x and y positions within the display where you want something to start rendering, the pointer to the address of the .data image you want to print, and an offset that is only used for spritesheets. *De facto*, there are two additional arguments, which are the width and the height of the image to be printed. However, that information is contained within the .data files themselves - they are the first two words in those files. All that information is put into registers.

Just before proper rendering begins, a precheck is made to prevent illegal prints. An illegal print is any attempt to print anything outside the boundaries of the Bitmap Display. Printing beyond or before Y limits would cause memory errors, and printing beyond or before X limits would cause some pretty bad looping visual glitches. This is done by using the width and height of the image we want to print, and the position where we start printing it. If part of the image "fell out" of these boundaries, the image simply would not start rendering, and the function would return. From there, the process is indeed very simple. A pointer is created such that it points to the location in the display where printing will begin by using the first two arguments of the function. From there, there is a nested loop - one loop for printing along the columns inside a loop for printing along the lines. As such, it prints every pixel (as many pixels as the width of the image) in one line then goes to do the same in the next line, until the same number of lines as the image height have been printed, then it returns. The process of actually copying a pixel from the image and pasting it on the display is done by simply copying the byte in the image pointer and saving it to the bitmap pointer. For performance, since it is known that every image, we render has a width that is a multiple of four, this is not actually done by saving byte by byte, but rather, word by word. This speeds up rendering by a factor of approximately 4.

Sometimes, we want to render an image from a spritesheet. That is very simple, but we need an indexer and a multiplier. The multiplier is equal to the number of pixels in each image on the spritesheet, and the index refers to the image from the sheet we want to render. It's 0 for the first image, 1 for the second and so on. We then multiply the indexer by the multiplier and save that in the offset register (a3 in the displayPrint function). The rendering will start based on that offset, and thus the desired image will render. Some images work with more intricate indexing, such as the player character, which has two independent states: direction and whether they're using the special move. So, there are two indexing steps such that we end up on the desired image for the desired combination of states. This is done by getting two indexes, one for each state, and multiplying one of them by the number of possible states that the other index may assume, then adding them, multiplying by the multiplier, and saving in the offset argument.

There is an additional function related to rendering, called the frameswitch. FPGRARS provides two "frames" we can work with and switch between. When the instructions for rendering are being run, if things are rendered in only one frame every time, there are visible artefacts - one can essentially visualise the rendering process. To prevent this ugliness from reaching the player's aesthetically sensitive eyes, everything is rendered in the opposite frame from the one that is being shown. While we are looking at one frame, the next is being rendered. When that is done, the shown frame is switched, and the previous frame is used for rendering. The function itself is rather small and is also one of the few other functions that can be returned from. Reading through it is enough to understand how it works.

2.4 The Use of Matrices

Now that the second most important function has been presented, here is *the most important* function in the codebase. The matrix itself is not a function (obviously), but the level's matrix or map is used by a function called mapRender but is also accessed by many other procedures that manipulate it. In summary, the mapRender function calls the displayPrint function multiple times to render different objects according to all positions of the level's matrix, and the state of those objects. It is essentially the interface between the display and the game's code, but it also does some slight manipulation to some matrix objects itself. The matrix itself is a collection of numbers that each represent a game object in each level. Each position in the matrix corresponds to a 16x16 pixel cell on the display. A 0, for example, is an empty cell. A 9 is the player, a 1 is an impassable unmodifiable cell, a 2 is a breakable block, a 3 is a collectible, a 4 is a breakable block with a collectible in it and a 5 is an enemy.

On every tick, the mapRender function "walks" with a pointer through the matrix by incrementing on every loop, and renders each cell in the appropriate position, according to the number on that matrix position. Essentially, it saves the number on the matrix pointer to a register and compares it against all known identifier numbers. It then loads the appropriate image according to that number. If the number is not identifiable. It renders an empty cell. If a cell needs to be updated (Mostly just for breaking and building animations), it updates it and then renders (by loading the proper arguments and calling displayPrint). Once it finishes, the code continues normally. It should also be mentioned that the way the mapRender function provides the appropriate X and Y arguments for the displayPrint function is, intuitively, by multiplying the currently checked matrix X and Y positions by 16.

The importance of using a matrix to display the game's objects resides in the fact that it makes it far simpler, in all other parts of the code, to modify the game's state. If moving a character is as simple as erasing a 9 from one position and placing it in another, and the mapRender function will simply represent that action by removing the player from one position and placing it in another, it's shown that the code for all sorts of events in the game become easier to implement. The matrix is thus, in some ways, beyond just a concrete feature of the code, a principle. Most things in the game are done by simply altering, comparing and verifying matrix values. Other matrices are used for various purposes, but they all come back or are related to the main level matrix.

2.5 Level Loading

The way a level is loaded relies on a function that is run just before the game loop. It resets some values to their defaults and copies some data from memory into temporary spaces for manipulation. While on the surface this sounds very simple, it is in fact a rather lengthy algorithm. Although it's not a function that returns anywhere, it is indeed treated as a function rather than a simple procedure. Were it placed somewhere else in the code; it would have a return at the end. It takes four arguments, all pointers: One pointer to the level matrix, one pointer to the level background image, one pointer to the level information file and one to the level collectible updates file. This allows for level selection - one can provide pointers to information pertaining to multiple levels. It starts by resetting values in memory that are always the same at the start of any level: Points, player energy, player state, the ticker register and a few others. After that, it copies all information from the level matrix file into the currentLevel memory space as was described in the **2.2 - Memory Usage** subsection. This is done with a nested loop not too dissimilar from the one used in the displayPrint function - taking information from one place to another, word by word (since the matrix width is 20 and thus divisible by 4). The same is done for the level background in quite literally the same manner, both procedures using the respective pointers that were provided earlier.

2.6 The Game Loop

The start of the loop is described by an intuitively named label called gameLoop, and it is jumped to after everything that had to happen on the previous tick has happened. "Ticks" are what we refer to as one iteration of the gameLoop, and the tickrate is thus how many times a tick runs every given amount of time. The tickrate is approximately once every 50 milliseconds, or 20 times per second. Since the game is rendered on every tick, the framerate is also approximately 20fps. The tickrate can be defined and modified by changing the value of a0 for a sleep system call at the very end of the gameLoop (just before a "j gameLoop" instruction). Timing isn't exact using this method but it's also irrelevant. Whatever the game calls a second is just a measure of time used to perform certain time-reliant actions. The register s11 is used as a global tick counter and gets incremented by one on every tick. The game loop contains all relevant functions that run while the player is on a match and is also responsible for updating variables such as the timer as the game progresses.
To give an example of a smaller function that might run within the game loop, the Collectible Updates (from here on, referred to as colupdates) procedure is another matrix manipulator that either adds new collectibles to the level after the player is done collecting everything or flags a victory if there are no more updates to be made under the same condition. It is separate from the mapRender function to increase code readability, and of course that makes it far easier to implement.

2.7 The Player

The player is identified by the number 9 in the matrix. It can interact with the game via user input and is effectively the character the user controls. The player can perform two actions: move or do its special. Movement is achieved by pressing directional keys W, A, S or D and the result is that the player's position may be updated accordingly depending on the target cell. The other action is the special - it can create or destroy contiguous lines of blocks that will engulf and "freeze" collectibles in its path, or unfreeze them, depending on the type of cell the player is facing and the player's direction.

2.8 The Enemies

Enemy implementation involved two main problems: their behaviour and their rendering. While rendering may appear to be simple on first thought, it will be explained later why enemies are probably the hardest object to render in this game. When it comes to behaviour, enemies were programmed such that their movement is not deterministic but can be somewhat predicted. This adds some neat variety to the gameplay and forces the player to use adaptive strategy instead of attempting to figure out an optimal path. Enemy movement is possibly the most complex algorithm in the game after the mapRender algorithm.

While the player's movement and behaviour is determined by input from the user, enemies in the game don't have the luxury of a brain. So, movement and behaviour are *performed* similarly but *decided* very differently. There are two enemy types: A simple moving enemy that causes a game-over on touch and a slightly more intricate enemy that moves the same way as the previous one except this one blows up lines and columns on occasion. The simple movement algorithm therefore applies to both enemies, and the second enemy has a second algorithm to run that is somewhat like the player's break algorithm, except this breaking goes in all directions, breaks the entire line or column, and passes through impassable cells, and kills the player if they are caught in it. Unfortunately, in the case of the enemy explosion algorithm, it is not contained entirely within the Enemy AI procedure. The enemy AI updates an explosion matrix which is then rendered and utilised by a secondary renderer function that deals only with the enemy explosions. Once the explosion animation is done, everything that's related to the explosion is reset.

2.9 Music and Sound Effects

There is a significant contrast between the ease of implementation of music, and the ease of implementation of sound effects. One-note sound effects were possibly the single easiest thing to implement in the game - all one has to do is to make the appropriate sound effect call whenever a certain event occurs. It's too simple to describe further - that really is the entire explanation. Music, however, has a tempo that must be tied to the game's tickrate in order to be properly played. Not only that, but it is also extremely difficult to translate notes and instruments into numbers by hand.

3 Obtained Results

The final product is a fully playable game that can be run in the game's own directory through a terminal command (described in README.txt). The gameplay is very similar in pace and objectives to the original, despite having only two enemy types. The art style is vibrant while still consistent with the chosen aesthetic. While the game pales in comparison to the original BadIceCream, one must remind themselves that RadIceCream was essentially built from scratch. More than anything, it's evidence of the emergent properties of computer code - while individually, the lines of code are just taking numbers and changing and rearranging them, when this is done in an orderly fashion, we are presented with a finished product that performs a given task.

4 Conclusion

First and foremost, it should be noted that designing a game within the severe limitations of assembly language, as well as the limitations posed by the runtime simulator itself among other things, is perhaps not the best of ideas unless for scientific or entertainment purposes. Full games written in assembly nowadays are usually projects for older hardware – mostly consoles. Games that *had* been written in assembly before likely all fall into one of those situations. Artistic expression is, perhaps, another applicability of creating a game using these low-level mechanisms.

With that in mind, for educational purposes as was the objective of this project, the game proved itself a mind-tangling challenge, perhaps impossible to truly finish for all those who attempted it, including the authors themselves (Perhaps nothing is ever *truly* finished anyways). It can easily be assumed that all participants and contributors had a learning experience that certainly builds upon the concepts learned in the Introduction to Computational Systems class and, moreover, refines that knowledge to whole new levels of understanding of the intricacies concerning the functionality of a computer system.

However, this was not the only area that was built upon throughout the project. All around, it involved concepts of game design and forced the authors to learn their way around concepts that, to a naive viewer who simply played the game, seem very simple, but are indeed quite complex. Manipulating matrices, navigating around data structures that the authors themselves designed shows that this was not only an exercise in assembly language itself, but an exercise in mathematics, logic, game design, arts and the real-world application of these subjects.

Finally, remember that it is impossible to truly capture the essence and all aspects of the game's code in just this paper. You are encouraged to go and read through the project's files yourself. Lots of intricacies were left out in favour of a much broader explanation. You may also feel free to contact the authors with any questions through the project's GitHub page, which will be linked in the **5 - References** section. We sincerely hope that this project's files may help future Introduction to Computer Systems students to find their way around their own projects, as the ones that came before us served as great references, especially in terms of music implementation.

5 References

1. The RISC-V Instruction Set Manual Vol. 1
https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf

2. RARS' GitHub Repository
https://github.com/TheThirdOne/rars

3. Introduction to Input and Output in RARS by Victor Lisboa - https://youtu.be/YZTJX0P9wN8

4. Professor Marcus Vinicius Lamar's materials for Assembly Language and I/O (No link)

5. RadIceCream's GitHub Repository
https://github.com/NirvaCx/RadIceCream

6. All MIDI Instruments' Sounds, Numbered
https://youtu.be/IYdq06l8qXI?si=tSwBfETysr4kSUOW

7. Leonardo Riether's FPGRARS GitHub page
https://github.com/LeoRiether/FPGRARS

8. BadIceCream by Nitrome
https://www.nitrome.com/html5-games/badicecream/

9. Introduction to RARS and FPGRARS by Victor Lisboa
https://youtu.be/kEZ37dw_Pp4?feature=shared