CS 21 Machine Problem: Pokémon Red Editor

Department of Computer Science
College of Engineering
University of the Philippines - Diliman

1 Introduction

For this project, you are to create a MIPS-based program that allows persistent editing of certain game values of the *Pokémon Red* game.

1.1 Pokémon Red

Pokémon Red is a role-playing game released more than two decades ago for the Game Boy console. Due to the franchise maintaining its popularity, attempts at reverse engineering the game have been numerous with a few yielding notable success—the information obtained from these form the foundation of the project detailed below.

1.2 ROM

Read-only memory (ROM) is a type of nonvolatile memory that is not readily modifiable and is used mostly to store programs. As implied by the name, end users are not expected to make modifications to programs stored in ROM.

In the context of the Game Boy console, each game is distributed in an external ROM unit (usually 1 MiB) called a *cartridge*. A specialized setup is needed to *dump* the stored program in cartridges into hard disks to be further processed via personal computers. This raw cartridge data dump is commonly referred to as a *game ROM*.

1.3 RAM

As discussed during lecture class, RAM chips are primarily used as the hardware representing main memory. Transient data generated during program execution is normally stored in RAM.

Game Boy save data is usually stored in a separate *byte-addressable* cartridge RAM chip (usually 32 KiB) that is powered by an internal battery to avoid data loss (due to the use of volatile storage). Loading save game data usually involves reading from save data RAM and populating main memory RAM based on the read values. Knowing the data mapping of the game and being able to modify stored bytes on ROM or RAM allows players to effectively *cheat* the game.

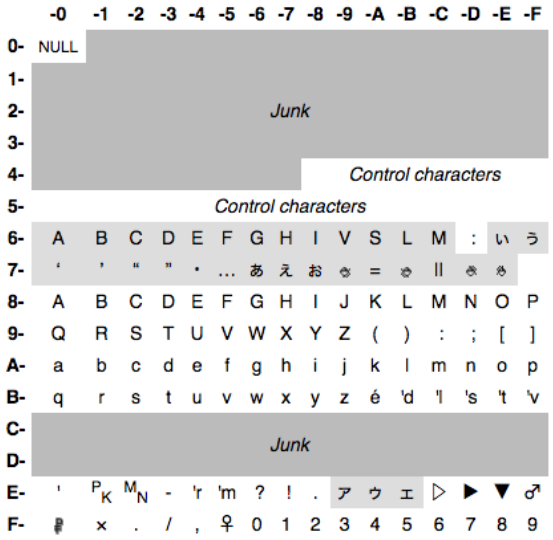

Figure 1: Proprietary English character encoding map (bytes are in hex)

2 Project Specifications

Create a MIPS-based program that allows persistent editing of Pokémon Red **save data** (`.sav`) and the actual **game ROM** (`.gb`).

2.1 Character encoding

Pokémon Red does **not** use ASCII to display any form of text. Instead, it uses a proprietary character encoding format[1] described in Figure 1.

For this project, you are only required to support the following byte ranges:

- `0x7f` (single space)
- `0x80-0xb9` (`A-z`)
- `0xe0` (`'`)
- `0xe3` (`-`)
- `0xe6-0xe8` (`?`, `!`, and `.`)
- `0xf3-0xf4` (`/` and `,`)
- `0xf6-0xff` (`0-9`)

In addition, certain bytes are to be rendered as follows:

- `0x00` as `^` (caret)
- `0x4f` as \
- `0x50` as `&` (not displayed; mapping is used for input)

[1] `https://bulbapedia.bulbagarden.net/wiki/Character_encoding_in_Generation_I`

- `0x51` as `|`
- `0x55` as `_` (single underscore)
- `0x57` as `#`
- `0x58` as `$`
- `0xba` as `@`

All other (unsupported) bytes must be rendered as `*` (single asterisk) whenever present. The `0x50` byte indicates the end of a string—this must **not** be displayed if a string is to be printed out.

2.2 Pokémon indices

Each kind of Pokémon has an *index number* (one-indexed) used mostly to determine which element in an array of values corresponds to which kind of Pokémon. The list[2] dictates that the first index (`0x01`) corresponds to *Rhydon* while the last index (`0xbe`) corresponds to *Victreebel*.

In case you are unfamiliar with the kinds of Pokémon and how they look like, please refer to the following references (note that they are sorted by *Pokémon number*):

- `https://pokemondb.net/pokedex/national#gen-1` (only *Generation 1* or the `#001` to `#151` are relevant)
- `https://www.pyimagesearch.com/wp-content/uploads/2014/03/pokemonsprites.png` (first row is `#001` to `#012`)

Note that the *Pokémon number* and *index number* are distinct values—*Nidorino* is Pokémon number `#033` but has an index number of 167 (`0xa7`).

2.3 Address mapping

A large chunk of the Pokémon Red game data has been mapped out through copious amounts of reverse engineering done meticulously enough that an equivalent disassembly[3] has been made possible.

The functions of the data across particular address ranges have been decently documented for both the ROM[4] and the save data[5]. As an example, the byte of data in ROM address `0x616d` has been identified to determine the kind of Pokémon featured during the introductory text of the game (the default value `0xa7` corresponds to the Pokémon index of *Nidorino*). Modifying the byte found here will change the Pokémon displayed if the modified ROM is played.

2.4 Save data

2.4.1 Checksum

To verify the integrity of save data, an 8-bit *checksum*[6] is employed. The checksum is computed as follows:

[2] `https://github.com/pret/pokered/blob/master/constants/pokemon_constants.asm`
[3] `https://github.com/pret/pokered/`
[4] `https://datacrystal.romhacking.net/wiki/Pok%C3%A9mon_Red_and_Blue:ROM_map`
[5] `https://bulbapedia.bulbagarden.net/wiki/Save_data_structure_in_Generation_I`
[6] `https://bulbapedia.bulbagarden.net/wiki/Save_data_structure_in_Generation_I#Checksum`

- Initialize the unsigned integer checksum to `0`
- For every byte inclusively between `0x2598` and `0x3522`, add its value (as an unsigned integer) to the checksum (truncating significant overflow bits ala *modulo 256*)
- Invert all the bits of the resulting checksum

This single-byte value is stored in offset `0x3523` of the save data. Integrity of the current save data is verified by computing the checksum using the current values and ensuring that the same checksum stored in `0x3523` is obtained. If a checksum mismatch occurs, the save data is discarded and results in a `The save data is destroyed!` prompt to the player.

Any modification made without recomputing and storing the proper checksum will most likely render the game unplayable. **Ensure that this is always properly recomputed and written before your program exits.**

2.4.2 Player name edit

The 11-byte player name is stored starting at address `0x2598` using the character encoding referred to in Figure 1. The order of characters in memory is as is; the name `RED` will result to the bytes `0x91`, `0x84`, and `0x83` starting at address `0x2598` and terminated with a `0x50` byte. Note that the game allows at most seven (7) characters despite 11 bytes being used.

When this function is invoked, display (in a separate line) the current player name stored in the save data using the scheme found in Section 2.1. After which, expect string input with at least one and at most seven valid characters. This string input should be written to the save data encoded such that it appears as the intended characters in-game (i.e., store this using the proprietary encoding).

2.4.3 Money edit

Player money is stored using three bytes starting at address `0x25f3` as a six-digit *packed binary-coded decimal* (packed BCD) value ranging from `0` to `999999`. Each digit is represented using four bits; as such, the two nibbles of each byte in memory represent two adjacent digits. This value is stored in **big endian**. As an example, the value `93406` will be stored in memory (in order) as `0x09`, `0x34`, `0x06`.

When this function is invoked, display (in a separate line) the current amount of money stored in the save data (no leading zeros for nonzero amounts) with the format `P <AMOUNT>` (e.g., `P 93406`). After which, expect an input integer ranging from `0` to `999999`—this should be the new value of current money stored (in the proper format). The in-game amount of money should reflect this.

2.4.4 Badge edit

In Pokémon Red, the primary objective of the player is to obtain eight *gym badges*. The badges are named as follows:

- Boulder Badge (`BOULDERBADGE`)
- Cascade Badge (`CASCADEBADGE`)

- Thunder Badge (`THUNDERBADGE`)
- Rainbow Badge (`RAINBOWBADGE`)
- Soul Badge (`SOULBADGE`)
- Marsh Badge (`MARSHBADGE`)
- Volcano Badge (`VOLCANOBADGE`)
- Earth Badge (`EARTHBADGE`)

A single byte in address `0x2602` stores whether or not the player has obtained a particular badge. Each of the eight bits corresponds to a badge starting with the *most significant bit* representing the Boulder Badge and following in order until the *least significant bit* for the Earth Badge. A set bit (`1`) indicates that the particular badge has been obtained with an unset bit (`0`) indicating otherwise.

When this function is invoked, output text must be displayed (see below) and input (with format below) should be expected.

The format of the input which should result in the new byte to be stored is an eight-character string containing only `0` (ASCII `0x48`) and/or `1` (ASCII `0x49`) with each character corresponding to the badge in the same scheme described above (i.e., the first character is for the Boulder Badge).

The format of the output corresponding to obtained gym badges is the badges obtained (in order; use capitalized version) separated by a single space (no trailing spaces). If there are no badges obtained, output `NONE` in a separate line. To constrast with a nonempty example, a `01010010` byte read should display the following in a separate line: `CASCADEBADGE RAINBOWBADGE VOLCANOBADGE`

2.4.5 Item bag editing

The player can store at most 20 items in the bag. As there are 256 items, exactly one byte is needed to index[7] each item uniquely. The quantity of each item in the bag is represented using one byte, allows a maximum theoretical amount of 255 (game logic normally prevents more than 99).

As described in this[8] reference, the bag is stored starting at address `0x25c9`. The following offsets from `0x25c9` are defined as follows:

- 0: number of distinct items in bag (T; one byte)
- $2k - 1$: entry of k^{th} distinct item in the bag (two bytes; spans until offset $2k$)
- $2T + 1$: terminating byte (`0xff`)

The entry of each distinct item is represented using two bytes. The first byte is the *item index* (determines the kind of item) while the second byte is the *quantity* (`0-255`).

When this function is invoked, display each item in the bag (in order starting from the first entry). Each entry must be displayed with format `<#> <NAME> x<QUANTITY>`

[7] `https://bulbapedia.bulbagarden.net/wiki/List_of_items_by_index_number_(Generation_I)`

[8] `https://bulbapedia.bulbagarden.net/wiki/Save_data_structure_in_Generation_I#Item_lists`

(name format below; separate line each) with `<QUANTITY>` as a regular integer and `<#>` starting from `1` and incremented by one for each succeeding line. If there are no items in the bag, display the string `EMPTY` in a separate line. As an example, a single Potion in the bag should yield an output line containing `1 POTION x1`.

Apart from showing the current items in the bag, an input string in the format of `<#> 0x<ID> <QUANTITY>` wherein `<#>` corresponds to the same `<#>` described above, `<ID>` is a lowercase hexadecimal value **always containing two symbols** (`0` must be `00`), and `<QUANTITY>` is an integer with range `1-255`. The input should change whatever is the `<#>`th item in the bag into the item with index `<ID>` with an amount equal to `<QUANTITY>`.

When showing item names, refer to the data starting at ROM address `0x472b`. Each item name is terminated by a `0x50` character and should be decoded using the scheme in Section 2.1. The said item name data is stored as a *jagged array*. Note that the `0x50` byte of the last item is found at address `0x4a91`.

2.5 Game ROM

2.5.1 Checksum

As with the save data, the ROM also has its own checksum to verify its own integrity. A two-byte *checksum*[9] stored starting at address `0x14e` (big endian) is employed. The checksum is computed simply by summing up the bytes of the entire ROM (with the exception of the checksum bytes themselves).

For some strange reason, all iterations of the Game Boy console do not verify checksum integrity. Despite this, your program must be able to install a valid checksum after all changes have been made. **Ensure that this is always properly recomputed and written before your program exits.**

2.5.2 Title screen Pokémon display

The title screen of Pokémon Red showcases various Pokémon rotated one at a time in a *random* fashion. The list of Pokémon that appear are stored in addresses `0x4399` and `0x4588-0x4597` of the ROM with each byte corresponding to a Pokémon index. `0x4399` dictates the first Pokémon to (always) be displayed.

When this function is invoked, the names of the said Pokémon must be displayed in separate lines and in increasing address order as defined in the ROM—use the Pokémon names in the array starting at address `0x1c21e` of the ROM. Text should be decoded as per the scheme presented in Section 2.1.

Note each element of the said array is exactly 10 characters with `0x50` bytes padding names less than ten characters. Note that since the characters must be terminated with at least one `0x50`, a maximum of nine characters is allowed per Pokémon.

2.5.3 Dialogue search

The ROM has dialogue scripts scattered throughout its address space. Unfortunately, the dialogue mapping is not as well-documented as the other parts of the ROM. To help locate the addresses of particular dialogue texts, you are to provide a way to locate the

[9] `http://gbdev.gg8.se/wiki/articles/The_Cartridge_Header`

address of an input phrase.

When this function is invoked, expect a search string (limited to 30 characters) to be given. After which, the address containing the *first instance* of the search string (i.e., smallest memory address) should be shown in lowercase hexadecimal preceded by `0x` without leading zeros (e.g., `0x8a426` should be printed for the input `Hello there`) or `NOT FOUND` if the search string was not found in the ROM.

Control characters may be entered as input as defined by the scheme in Section 2.1. Also, it is expected that this substring search is to be performed from the first address of the ROM up to the end (treat the entire memory as if it contains purely valid dialogue text).

2.5.4 Mom's dialogue edit

In conjunction with the dialogue search, you are also to provide a way to edit the first dialogue of the protagonist's mother (found in-game at the first floor of the house you start in). The starting address of the said dialogue will not be readily given—you are to search for the said address yourself (ideally using the required dialogue search code).

When this function is invoked, the currently-saved dialogue at this address has to be displayed—you will have to take account of the control characters `0x57` and `0x58` that mark the end of a dialogue string. Control characters may be both entered as input and displayed on screen as defined by the scheme in Section 2.1. **Assume no limit to how large the dialogue text to be printed will be** (*Hint: do* ***not*** *use the print string syscall*). Printing continues until displaying the first `0x57` or `0x58` read in sequence.

With regards to editing, a maximum of 150 characters (not including the terminating `0x00`) should be expected. **Do not insert the terminating** `0x00`. There is no need for the program to add in any extra control characters; assume the input will supply all control characters needed). The input may surpass the length of the original dialogue text; in this case, simply overwrite characters beyond the original length until the input length.

For reference, the expected output line for the default dialogue value in the ROM is
`MOM: Right.\All boys leave_home some day._It said so on TV.|PROF.OAK, next\door, is looking_for you.#`

2.6 User input format

The input format described below is for *standard input* (`STDIN`). Note that `STDIN` input during automated checking will be done via *input redirection*.

The first line of input is the filename of the ROM to be edited while the second line of input is the filename of the save file to be edited. Note that the number of bytes to be read from each file is defined in the following section.

The next line of input corresponds to a function with mapping defined in Table 1. The expected behavior of each function as detailed above is summarized below (after executing the chosen function, the program must write proper ROM and save data checksums and terminate):

Input	Function
1	Player name edit
2	Money edit
3	Badge edit
4	Item bag edit
5	Title screen Pokémon display
6	Dialogue search
7	Mom's dialogue edit

Table 1: Function mapping

1. Player name edit
    - Display the stored player name
    - Expect a new player name as input to be stored (1-7 characters)
2. Money edit
    - Display the stored money `P <AMOUNT>`
    - Expect a new money amount as input to be stored (`0-999999`)
3. Badge edit
    - Display each badge currently obtained (space-separated, use capitalized names), or `NONE` if none
    - Expect a string with 8 characters (0s and 1s) as input to be stored (see Section 2.4.4 for the format)
4. Item bag edit
    - Display each item currently in the bag (one entry per line; see Section 2.4.5 for the format) or `EMPTY` if none
    - Expect an input string `<#> 0x<ID> <QUANTITY>` that changes a particular bag entry (see Section 2.4.5 for details)
5. Title screen Pokémon display
    - Display each Pokémon displayed in order of appearance in the ROM (one entry per line; see Section 2.5.2 for the format)
6. Dialogue search
    - Expect an input string to search (30 characters at most)
    - Display the address in ROM memory containing the *first instance* of the search string (see Section 2.5.3 for the format) or `NOT FOUND` if the search string does not exist in the ROM
7. Mom's dialogue edit
    - Display the current text of the first dialogue of the protagonist's mother as defined in the ROM
    - Expect an input string as a replacement (150 characters at most; more details found in Section 2.5.4)

2.7 File input format

Assume that the file format for both the ROM and save data files are valid and consistent with respect to how they are originally laid out. Both files will always have a fixed number of bytes (1048576 bytes for the ROM and 32768 bytes for the save data).

2.8 Output format

Refer to the output formats defined above for each function. The output must be generated on `STDOUT`. During checking, `STDOUT` will be redirected into a file representing program output—this means that **no line of input will be included in the output**.

For ease of manual testing, you may include extra output lines that **must** start with a `#` character for debugging—these lines will be ignored (i.e., deleted or treated as though the line and its newline character were not there) during automated checking. Ensure that no expected output is in the same line starting with `#` as it will be ignored.

2.9 Scoring rubric

Criterion	Points
Save data checksum is properly computed	5
Player name is properly displayed and modified	10
Money is properly displayed and modified	10
Badge data is properly displayed and modified	10
Item bag is properly displayed and modified	10
Item names are properly displayed from the ROM (i.e., not hardcoded)	10
ROM checksum is properly computed	5
Title screen Pokémon is properly displayed	10
Pokémon names are properly displayed from the ROM (title screen editing is required)	10
Dialogue is properly searched and displayed	10
Mom's dialogue is properly displayed and modified (dialogue search is required)	10
	100/100

2.10 Testing

- You may want to test your modifications on both `.gb` and `.sav` files by actually *playing* the game and checking the values in-game if they have been successfully modified
- Files that may help with testing for *academic purposes* may be found in the UVLê course site
- Valid save files may be generated simply by saving in-game—a `32 KiB .sav` file should be generated (normally in the same folder as your `.gb` file; this is emulator-specific)
- The active save file is generally the `.sav` with the same base filename as the `.gb` file—use this information to easily maintain an organized set of save files for testing (you may ask for help from your instructor regarding this)
- It might be helpful to create a separate program that allows you to change the byte under an input address—this allows you to manually test if the addresses and/or values you are entering as input are right (test this via running the game)

- Grading will be purely automated—ensure that your output conforms exactly to the specifications
- Expect all test input to be valid—no malformed input test cases will be given (i.e., no need for input validation)

3 Submission Details

Submit your MIPS assembly source code with the filename `<studentnumber>.asm` (no angle brackets) through the UVLé submission module.

Deadline: 2018 May 25, 11:59pm (before Saturday)

Note: non-working or late submissions will NOT be credited.

4 Disclaimer