Deck Shuffler

Ashnah Khalid Khan 22889
Syed, Danial Haseeb 12429

This document contains a step-by-step walkthrough of our deck shuffling algorithm that was written in the primitive instructions of the MIPS assembly language.

We have implemented the widely-used Knuth shuffle, which is both intuitive to programme and efficient to run (requiring only $O(n)$ time). The primary data structures we have used to represent the deck of cards are *arrays* and the main programming construct used to fill them out were *nested loops.*

The Deck

We have used the traditional 52-card deck. It is divided into 4 *suits* (♥ = `H`, ♦ = `D`, ♣ = `C`, ♠ = `S`), each of which contains 13 *ranks.* The ranks are further split into 8 *number-cards* (`2`, `3`, `4`, `5`, `6`, `7`, `8`, `9`,), 4 *picture-cards* (`T`, `J`, `Q`, `K`) and the special *ace* (`A`).

Each *card* is a combination of a rank and a suit. For example, `2H` represents the 2 of hearts. Finally, we represent the deck as a sequence of cards, separated by spaces.

The Algorithm

We will attempt to explain our algorithm in decreasing layers of abstraction. Thus, we present a high-level overview first.

Our algorithm starts off by initialising two empty arrays of size 52 each; one for the `rank` and one for the `suit` of each card in the deck. It also initialises an array that has each `SUIT` as an element.

Next, it fills out both the empty arrays using nested loops and generates a full deck of cards. For example, the i^{th} card in this deck can be accessed by printing `rank`[i] and `suit`[i] consecutively. For comparison purposes later, our algorithm prints this (unshuffled) deck.

Then, the Knuth shuffle takes place by sequentially swapping each card by another random card in the deck. The resulting deck is then printed as output.

The Details

As seen in the previous section, our algorithm is divided into two major parts; the first is the generation of the deck, and the second part shuffles that deck. Each of these parts have some *code blocks* associated with them. For example, deck generation includes:

1. `FillRankArray` looping 4 times,
    2. `AddAce`,
    3. `AddNumberCards` looping 8 times,
    4. `AddPictureCards`,
5. `FillSuitArray` looping 4 times,
    6. `AddSuits` looping 13 times, and
7. `PrintDeck` looping 52 times.

The shuffling part just includes `Shuffle` and `PrintShuffledDeck`, both looping 52 times. There are also a few helper code blocks used for tasks such as formatting the output (`PrintNewLine`), managing the temporary registers (`ResetRegisters`), and exiting the programme (`Halt`).

Now, we will look at each of these code blocks in all their gory detail.

`FillRankArray`

In our deck of 52 cards, the set of 13 ranks repeats exactly 4 times. Taking advantage of this inherent symmetry, we have made this outer loop to run 4 times (the loop counter is stored in the temporary register `$t0`). On each run, it adds 13 elements to the `rank` array (which is indexed using the temporary register `$t1` that is intialised to 0).

This code block is also tasked with initialising (and resetting) the temporary register `$t2` to 2, which is used by `AddNumberCards`.

`AddAce`

Here, we simply add the first rank to the deck. The character `A` is stored in register `$s0`. Then, the contents of `$s0` are placed in `rank[$t1]`. Finally, we increment the index (stored in `$t1`).

`AddNumberCards`

Using a special loop counter stored in the temporary register `$t2`, this code block sequentially adds all the number cards to the deck. Since the loop counter is a character itself, we store its contents in the register `$s0`. Then, the contents of `$s0` are placed in `rank[$t1]`. Finally, we increment the index (stored in `$t1`) and the loop counter by 1.

`AddPictureCards`

Since there is no meaningful loop counter for this code block, it simply adds the picture cards one by one, incrementing the index by 1 each time. Once it has added `K`, the last picture card, it increments `$t2` by 1 and loops back to `FillRankArray`.

`FillSuitArray`

This code block works similarly to `FillRankArray`, except that it adds the elements to the `suit` array instead (which is indexed using the temporary register `$t5` that is intialised to 0).

It also ends up being much simpler because suits are not divided into multiple categories, like ranks are. This outer loop runs 4 times, with its counter stored in `$t3` (initialised to -1 because the first thing it does is increment this index by 1). This also represents the index of the static `SUIT` array, as it is those elements that we are actually looping through.

As done previously, this block also initialises the temporary register `$t4` to 0, for use by its inner loop.

`AddSuits`

The logic of adding each element largely remains the same. This time, however, we must first retrieve the correct suit symbol by storing `SUIT[$t3]` in `$s0`. Then, we place the contents of `$s0` in `suit[$t5]`. Finally, we increment the index (stored in `$t5`) and the loop counter by 1.

The Helper Blocks

The inner workings of the printing blocks and helper blocks like `ResetRegisters` are fairly straightforward. Therefore, we will not go into detail here and instead move on to the much more interesting shuffling part of the algorithm.

`Shuffle`

This code block is the crown jewel of the programme. It is the MIPS implementation of the Knuth shuffle that runs by looping 52 times on the deck that has already been generated. The loop counter is stored in the temporary register `$t0` that is initialised to 0.

It starts by generating a random number between 0 and 51, and storing it in `$a0`. This random number is used to index the `rank` array; thus `rank[$a0]` is placed into `$t2`. Then, `rank[$t0]` is placed into `$t3`, allowing the two to be swapped. This is achieved by storing the contents of `$t2` in `rank[$t0]` and the contents of `$t3` in `rank[$a0]`.

A similar process is then applied to the `suit` array, except the same random number as before is used. Then the loop counter is incremented by 1 and the loop is repeated.

Q.E.D.

If all goes well, the programme should output something like the following:

```
AH 2H 3H 4H 5H 6H 7H 8H 9H TH JH QH KH AD 2D 3D 4D 5D 6D 7D 8D 9D
TD JD QD KD AC 2C 3C 4C 5C 6C 7C 8C 9C TC JC QC KC AS 2S 3S 4S 5S
6S 7S 8S 9S TS JS QS KS

TD AC 3S 2C 8C 9H 7S 3C 3H 7C 9C JC QC 4D 8D TH 5D 4C AS 7H 4S 5H
7D 8H KS QH 4H AH 2S TC 6D 5C 9D 2D AD 6H QS 6C 6S 9S 5S JH QD 8S
3D KC KD 2H TS JS JD KH
```