# The Game of Yatzy

## Vybortseva Anastasia, Mashkin Nikolay, 2025

## Summary

As a result of our work, we have implemented “The game of Yatzy”. The project consists of two main parts: hardware (Logisim) and software (Cdm-16 assembly). In the hardware we have implemented all the game rules and main logic of players moves and the software takes care of the bot logic, so that it can successfully make moves and beat all the inexperienced players.



# Logisim

## Main schematic

The main playing area looks like this. Left side is dedicated for a human player and the right side is controlled by a bot.

Buttons on the red box control, which cubes are locked right now. Cubes are represented with numbers from 0 to 6 (0 means that it’s inactive) and indicators that show if the cube is locked or not as can be seen in the blue box.

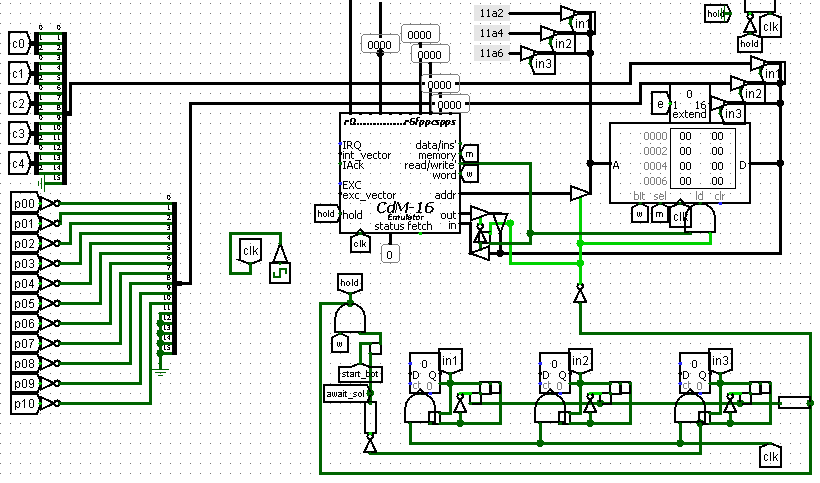
The bottom button from purple box controls the rolling of cubes which is available only if the diode on the bottom is red. Eleven buttons on the top are responsible for locking patterns.

Patterns are represented with the numbers, showing their value and indicators, showing which of them are yet to be locked as can be seen in the yellow box.

The bot side of the field is practically the same, but with few adjustments, so that it can be controlled automatically.

## 

There is also this part with the start button. As soon as button is pressed, the game begins and the starting player is chosen at random.



This part is dedicated to the bot logic and all the magic happens in here. Also, it is worth mentioning that we are using the Voun Neumann architecture for memory management.

In the red box you can see the “input data packagers”, essentially, they are just packing the game state into 3 16-bit numbers, so that the data can be comfortably stored into RAM.

The blue box part is responsible for correct data inputting into the memory. There are three timers, each with different lifetimes, which trigger one the corresponding data bits to be stored. While it is all happening, the processor is being kept on hold so that it doesn’t mess up anything.

## 

And also, there is this part, that is doing all of the “output parsing”, which is essentially just some button clicks being imitated depending on certain bits being set.

That’s it for the main schematic, so now let’s dive into more detail on the hardware.

## Utility stuff

It is important to mention all the utility schematics: tor, tor\_l, tand, tand\_l, tt, tt\_fancy, niceSR and tick.

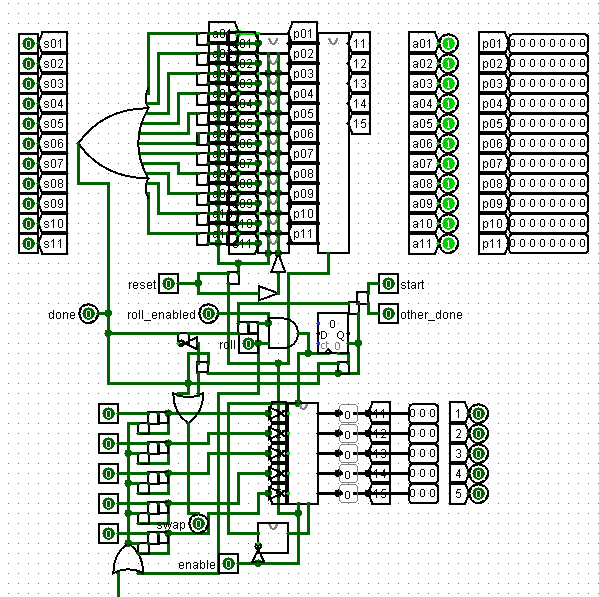
First four of them are just the tiny versions of logic gates, which are very convenient for small and elegant schematics.

Other two are T-triggers with simplified i/o: tt just has one input and one output, and tt-fancy is the same thing but also with enable input added to it.

niceSR is a regular SR with just two inputs and one output.

And tick is a micro-tick generator.

## Game logic



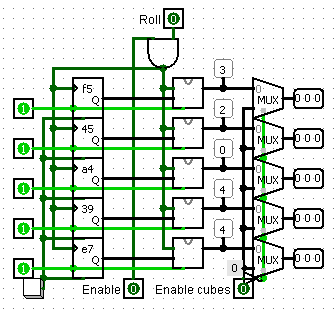
All of the game logic is basically implemented here.

Red box part is doing all the pattern calculations and locking logic.

In the blue box the roll count logic and roll enabling is implemented, as well as compiling the swap signal, which indicates that players turn has ended.

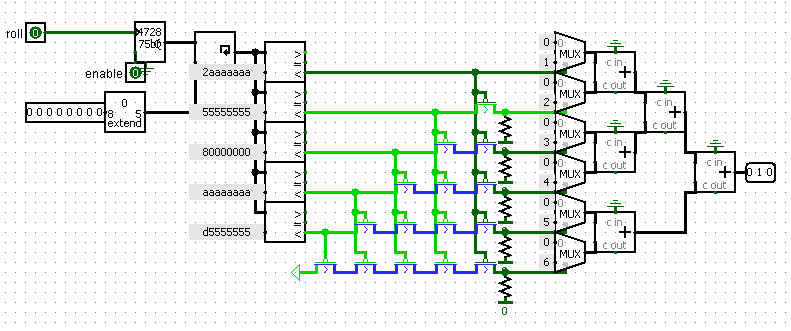
And the yellow box is responsible for rolling of the dice and resetting dice locks with each roll.

## Dice roller



Let’s take a closer look at the rolling of the dice.

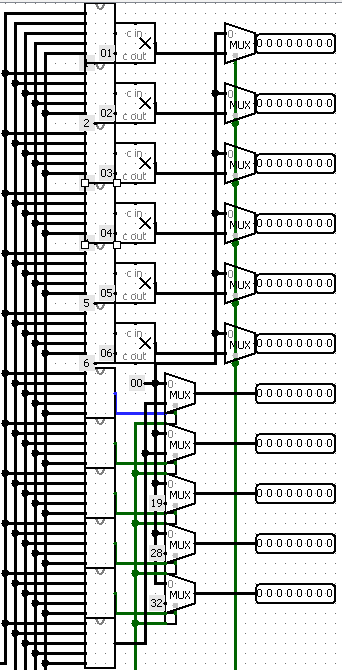
Here we can see 5 inputs for enabling each separate dice, as well as roll input ang roll enabler input. There are 5 random numbers being generated with each roll, which function as “seeds” for each of the dice.



Here is the D6 schematic. It generates a really big number, then cyclic shifts it based on the seed, and then, depending on what range does it fall into, the value of a dice is being chosen.

This is the best approach we could think of and although it might not be super precisely 1/6 chance for each value, it is close enough

## Pattern management

All of the pattern management is done here. This schematic takes 5 dice as an input and calculates, how many points can a player get, based on these values.

It is basically a distributor that relies on 11 other schematics, that each are responsible for one of the patterns and output, whether the pattern requirements are satisfied or not.

Six of those are responsible for detecting, how many there are cubes of a certain value. And the rest are slightly more complicated.

We can check for 3 or more cubes of the same value by just using first 6 detectors. The same goes for 4 or more cubes and exactly 5 cubes.

Then there is full house, which is a combination of exactly 2 same, exactly 3 same and not 5 same.

And finally, 5 in a row is checked via not 2 or more same and not 1 and 6 are present.

The last thing to cover is a point summator, but it is just a bunch of summators stacked together, so there is nothing interesting to show.

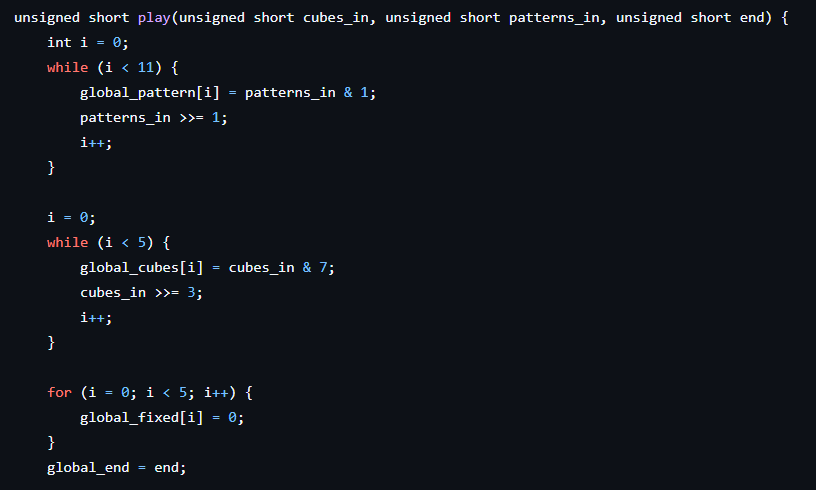
# Bot logic

Initially, for ease of understanding and convenience, the bot implementation was written in the C programming language and later recompiled into Assembler. Therefore, examples of C implementation will also be given as examples.

But first let's see what combinations we can make of our five dice and how many points they will bring:

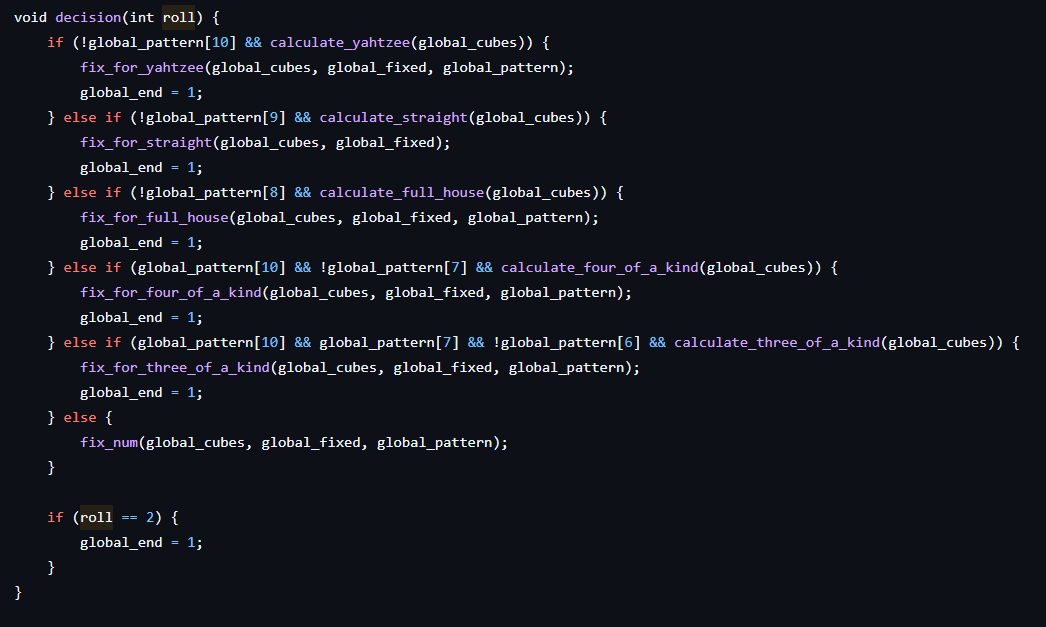
|  |  |
| --- | --- |
| NAME OF THE COMBINATION | HOW TO SCORE |
| Aces | Count and Add  Only Aces |
| Twos | Count and Add  Only Twos |
| Threes | Count and Add  Only Thress |
| Fours | Count and Add  Only Fours |
| Fives | Count and Add  Only Fives |
| Sixes | Count and Add  Only Sixes |
| 3 of a kind | Add Total  Of All Dice |
| 4 of a kind | Add Total  Of All Dice |
| Full House  (3 of one kind and 2 of the other) | SCORE 25 |
| Straight (Sequence of 5) | SCORE 40 |
| Yahtzee (5 of a kind) | SCORE 50 |

One of the most important features we can see is the basic implementation of a single dice roll for the bot. Let's consider it in more detail:

 As we can see, this function takes the values of the dice rolled, the statuses of the combination patterns, and a flag that indicates whether the current roll should end the turn.

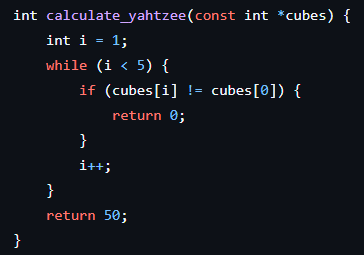
We then convert all these values into a form that we are comfortable with:

The template values, written to a single variable of unsigned short type, are placed in a list where the zero element corresponds to the status of the first template, the first element corresponds to the status of the second template, and so on. Similarly, we overwrite the template values in the list, only now we take into account that each cube occupies a value of 3 bits, not 1 bit, as it was with the templates.

 Next, we call a function that immediately determines if one of the complex combinations has fallen to us, and if it is still free. Then we change the flag so that the move is completed.

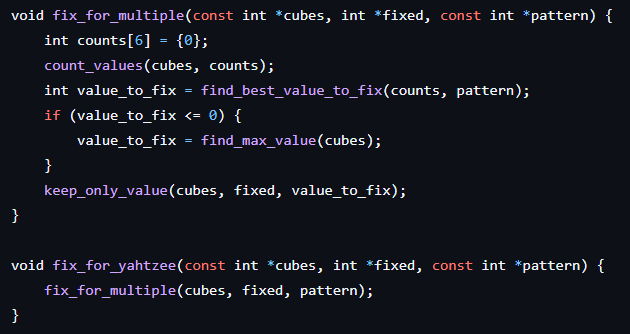
Let's take a closer look at what the decision function above does.

This function has a fairly simple idea. As we have already mentioned, it simply goes through all the complex combinations and checks if the combination matches and if it is still free. To do this, there is a function for each complex combination. Here is an example for the very first combination, which is also called Yahtzee:



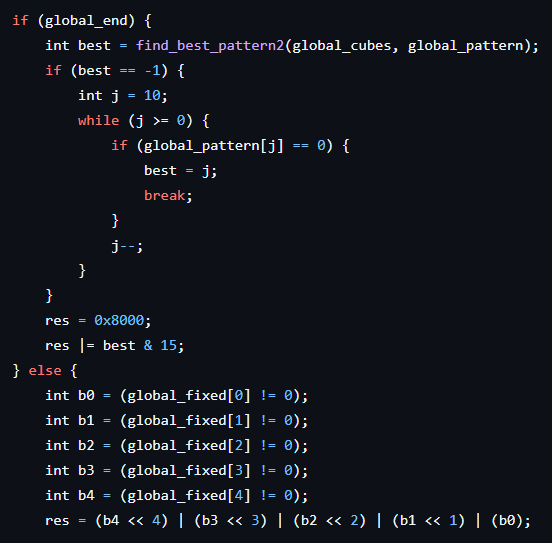
This combination represents five identical cubes with any value. That is, for example, the dice (4, 4, 4, 4, 4) correspond to this combination. Therefore, the corresponding function simply checks that all values of the rolled dice are equal. The other functions work in a similar way.

Next, in the decision function, if one of the conditions matches, the matching function for our pattern is called. Let us again give an example, for Yahtzee. If the previous conditions are met, the dice are fixed using the function to fix Yahtzee:

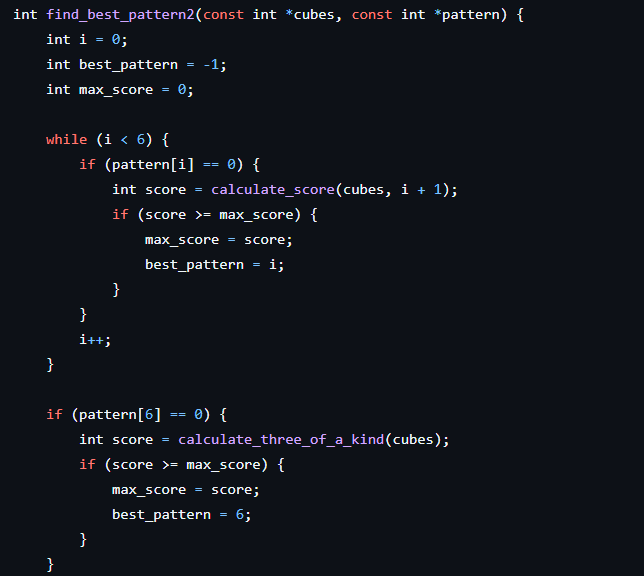


As you can see this function uses another, more general function to fix a given number of most favorable values. In such a function, we determine the most favorable value to fix as follows: if some most frequent value is not already occupied in its template (1-6 templates), we select it. If there is no such value, we fix the most frequent one.

If decision has not found any of the complex combinations, we fix the most favorable value using the find\_best\_value\_to\_fix(counts, pattern) function we discussed earlier.  
Then we go back to the play function.



Now our actions depend on our end game flag. If any complex combination is found and we still fixed, now simply by the resulting points of all patterns we determine which pattern we need to choose. The following function performs the selection of such a pattern:



And so on... How the calculate\_... we have already seen how the calculate\_... functions work.

If the complex combination was not found, we determine what values were changed, pack them into the variable res, by which the corresponding dice will be fixed in Logisim. Then the turn passes to the player.