ECE 385

Fall 2020

Final project

Gold Miner

Name: Xiao Shuhong & Lu Yicheng
Lab Section: D225

Written Description

Description of the overview of the circuit

For our final design, we copy the famous game "Gold Miner", For some one never heard it before, generally it is a typical arcade game that you achieve some target goal within some time and get to next level, otherwise you fail, clearly our goal here is catch gold.

For our design, we finished both one player mode and two player cooperation mode, a mode selection page was designed and also work as our game start page. After mode selected, we get a mission describe page which tell the target score of current level. And also, the game page for our two mode, they are similar, but we add difference between them to make much more fun. Also, we add a settle page to tell our player whether pass or fail, if pass, before go to next game, we design a shop page as well as two kinds of good, strength water which make rope recover faster, and an explosive to destroy anything being catching.

Almost all of our designs are achieved by hardware except the keyboard protocol and gold random initialization. We also design two interfaces, one for keyboard to tell which keycode is press and another store the gold position for next level.

Feature-hook

For our features, the core feature of our design is hook, it contains different kinds of modes: rotation, throw, catch, recover, a state machine is used to control our hook, in fact, we describe hook and string separately, however, as we define them in the hook.sv together, we consider them as one. The first challenge of designing hook is how to describe its rotation, as floating number are unachievable, we are unable to express rotating with a two dimensional matrix, so instead we design a pseudo-rotation, the idea is we divide here totally 180 degree it to several part, and the hook will only fall on these selection direction, as long as we choose larger enough direction, it will much like a rotation, however, adding direction need much work on the basic definition, so consider both functionality and complexity, we design 11 different directions.

Then, the lengthening of string also take some time to think, that is, decided which pixels are now belong to strings. Here our solution is we use some linear expression like ax=by to constrain the string pixels and use a unit signal to decide the lengthening in every unit. Also, the recovery is just similar, and the well define of unit signal help us construct a simply quality model that catch a heavy object like big gold or stone will always cause hook recovery slower than catch small object or nothing.

Then, the leaving thing here is how we define catching some object or not. First, the simplest logic here is if hook reach some bottom or right left side, we say it cannot catch and just recovery, than, if the hook pixels get some overlap with the object pixel (not the object background), we say some objects are catching and we recovery with this object. The movement and appearance of objects will individually be decided by object itself and have nothing else to do with hook.

The last thing here to mention is our hook are control by its state machine which we plan to discuss later.

Feature-timer

For every game, we have a timer which will show on the right corner of our game page. If timer goes to zero, we end our game. The task here actually is we should define a front table, for each number or letter, we define it 8x16bit, and store them in the on-chip memory.

Feature-score keeper

The implantation of score keeper is quite similar to timer, we split every single digit and using a front table to display. Also, score keeper will detect which object is now being caught and add the corresponding score.

Feature-objects

Objects here actually I mean gold, diamond and stone, we have at most 14 objects can show in one game, diamond, big golds, middle golds, small golds, big stone and small stones. Score of each object is fixed by keep score while all their positions are randomly decided by our c program.

For the hook catching our object, it should move with hook, this moving process is also defined within object.sv themselves, as well as disappear when hook finish recovery. A state machine also designs for every object, describe their behaviors of stay, move and disappear, the detail we be discussed later.

Feature-option switch (shop and start page)

To make our game more alive as well as adding some details, we design a simple option animation both on the start page to choose mode and shop page to choose goods to buy. Initially when we enter the page, the left most choice is selected and blink, player can change to right or left using keycode “a” and “d”. For the shop, after some good are bought, blink will stop and the icon just disappear. A restoration of the original game in shop is, if player do not buy any good and go to next game, the boss will show bad face, otherwise keeping happy face.

Feature-game props

For time limitation, finally we finish the designing of two game props: energy water and explosive. After bought an explosive, a small icon will show on the game page, player can press ”w” to use it, a bomb animation is added. For energy water, it makes hook recovery a little bit faster in the next game.

Feature-single player mode and double player mode

To make more fun, we design two different mode for single player and double player, single player mode is just what describe above and for double mode, there are some differences: a individual score keeper will show the score of each player together with total score; and to make game much easy, here we update score not when player successfully recovery hook and get objects, but as long as hook touch some object, it will disappear and update score, also a continuous bomb animation is add along the path of hook.

Description of the general flow of the circuit

For our design, most parts are finished inside with system Verilog; our circuit are divided into four part: To VGA interface, To CY7C67200 interface, To SDRAM interface and some test overflow.

For VGA part, we input a 50Mhz clock. RGB color, VGA clock, horizontal and vertical refresh signals will be output to an external monitor. Inside VGA Controller, we defined the basic parameter of display, flip and update pixels from left to right, from top to bottom, and the flash rate is pre-decided, then for every current pixel, a color mapper decided which color it should be now and output to monitor to display. For VGA， we do not have any test signal or waveform as we can see directly from our screen whether the current display is what we want or anything wrong.

For CY7C67200 part, it is the interface we get data flow with our keyboard, data will go through as input, it is designed just like a tristate buffer to avoid data collision. For test, we also connect the keycode to our FPGA board hex, so we can check when we press some key.

For SDRAM interface, it is designed for NOIS II software, here it is actually a large register file to store the data we need like what we do in lab 9, for our design, it is the random object position needed for every round of game. We can directly visit this register file and get the position data we want.

For other test, we use the LEDs and hex on our FPGA, to output and monitor those signal we are debugging.

Module Description

module: Game_start.sv

input:

Clk, reset, left, right, space, display_Game_start,

[9:0] DrawX, DrawY,

output:

[3:0]Game_start_index, singlebutton_index, doublebutton_index, compbutton_index,

[1:0]game_mode,

is_game_start, is_singlebutton, is_doublebutton, is_compbutton,

description:

Clk, when it is high, code block inside always_ff changes;

reset, used to initialize.

DrawX and DrawY, denote the current pixel to draw, with DrawX <=640, DrawY<=480.

left, right, space, these are there control signal to choose mode and select.

display_Game_start, tell color mapper to show start page when high.

is_xxx is high, denote in the start page, current pixel denote by DrawX and DrawY belong to icon xxx and xxx_index store the index of RGB from its individual palette.

game_mode choose whether go to single player or double player mode, also we designed for a double player competition mode, but leave unfinish.

purpose:

This module is the start page of our game, player choose which mode they are going to play here.

module: Target.sv

input:

Clk, reset, display_target,

[9:0] DrawX, DrawY,

output:

[19:0] target_score,

[3:0] target_index,

is_targetscore,is_target

description:

all basic signal like clk, reset, DrawX and DrawY, xxx_index and is_xxx, display_xxx are the same as describe above, to simplify, we do not repeat it again here.

target_score denote the lowest score should be achieve for this round to pass game

purpose:

This module show player their target for this round, will appear at the beginning of every game round.

module: big_gold.sv, middle_gold.sv, small_gold.sv, diamond.sv, bigstone.sv, smallstone.sv

input:

Clk, reset,

[9:0] DrawX, DrawY, goldx, goldy, taily, tailx,

[3:0] R_mode, big_gold_index (middle_gold/ small_gold/ diamond/ bigstone/ smallstone),

[2:0]state_out,

is_explode,

output:

[3:0] big_gold_index (middle_gold/ small_gold/ diamond/ bigstone/ smallstone),

[2:0]State_gold_out,

is_big_gold(middle_gold/ small_gold/ diamond/ bigstone/ smallstone), destroy, is_catch,

description:

all basic signal like clk, reset, DrawX and DrawY, xxx_index and is_xxx, display_xxx are the same as describe above, to simplify, we do not repeat it again here.

goldx and goldy denote the place position of the object on the screen.

tailx and taily denote the position of hook, used here to check whether this object has been caught by hook.

R_mode actually come from hook.sv, denote which direction hook is, used here to show the recovery direction of object after being catching.

state_out also come from hook.sv, denote the state of hook’ s state machine, used here to help decide whether explosive has been used and change the state of object itself together with is_explode.

State_gold_out is the state of object itself, output as a test signal to debug the object behavior.

destroy denote whether this object should disappear on the screen, the possibility of disappearance are: being catch, being explode or not being generated.

purpose:

These modules are the objects for catching, they are almost the same. In our game, whether some objects are caught or not actually decided by object itself (not hook), because hook has already got lots of function and hard to debug, also it much harder to let hook decide from more than 10 objects which has been caught than let every single object decides whether itself is catching.

module: hook.sv/ hook_left.sv/ hook_right.sv

input:

Clk, reset, left, extend, is_catch1-14, destroy1-14, display_random

[9:0] DrawX, DrawY,

output:

[3:0] hook_index, R_mode,

[9:0] string_taily,string_tailx,

[2:0]state_out,

is_hook, is_string, is_back, is_bound,

description:

R_mode denote the direction of hook, continuously change during rotate and fix when throw and recovery.

Extend set to high to throw the hook, controlled by keyboard.

string_taily and string_tailx deote the position of the tail of string as well as the hook.

state_out denote the state for hook, output to object modules.

is_back raise to high when hook is at state rotate.

is_bound raise to high when hook need to recovery from throwing.

purpose:

hook module is designed for single player mode while hook_left and hook_left used for double player, denoted two hooks for right player and left player. They are just the same, except the initial position of hook.

module: big_gold_double.sv, middle_gold_double.sv, small_gold_double.sv, diamond_double.sv, bigstone_double.sv, smallstone_double.sv

input:

Clk, reset,

[9:0] DrawX, DrawY, goldx, goldy, tailyl, tailxl, tailyr, tailxr,

[3:0] R_model, R_moder, big_gold_index (middle_gold/ small_gold/ diamond/ bigstone/ smallstone),

[2:0]state_outr, state_outl,

is_explodel, is_exploder,

output:

[3:0] big_gold_index (middle_gold/ small_gold/ diamond/ bigstone/ smallstone),

[2:0]State_gold_out,

is_big_gold(middle_gold/ small_gold/ diamond/ bigstone/ smallstone), destroyl, destroyr, is_catchl, is_catchr

description:

all basic signal like clk, reset, DrawX and DrawY, xxx_index and is_xxx, display_xxx are the same as describe above, to simplify, we do not repeat it again here.

goldx and goldy denote the place position of the object on the screen.

tailx and taily denote the position of hook, used here to check whether this object has been caught by hook.

R_mode actually come from hook.sv, denote which direction hook is, used here to show the recovery direction of object after being catching.

state_out also come from hook.sv, denote the state of hook’ s state machine, used here to help decide whether explosive has been used and change the state of object itself together with is_explode.

State_gold_out is the state of object itself, output as a test signal to debug the object behavior.

destroy denote whether this object should disappear on the screen, the possibility of disappearance are: being catch, being explode or not being generated.

purpose:

This is the group of modules of objects used in double player mode, it is quite similar to the single player object module. However, to make much fun, we changed a little bit about it behavior through changing its state machine, that now as long as hook touch these objects, they will just disappear while not follow the hook’ s movement, it is just like our player are not dig gold but destroy gold for fun.

module: background_game.sv, background_game_double.sv

input:

Clk, reset,

[9:0] DrawX, DrawY,

output:

is_background_game_double, （ is_background_game ）， is_player1,is_player2

[3:0] bgdouble_index, player1_index, player2_index,

description:

is_background_game_double（is_background_game）are fixed to high as whenever our control state machine decides to begin game, the background will show.

is_playerx raise to high when current position belongs to player icon.

purpose:

These two modules draw the background of our game, as well as the player icon. For single player, there will be only one player to show instead of two.

module: KeepSocre_doubule.sv/ KeepSocre.sv

input:

Clk, reset,

[9:0] DrawX, DrawY,

[19:0] target_score,

[2:0]state_outl, state_outr，state_outl)

catchl1-14, catchr1-14, (catch1-14)，time_end，display_random， display_double（display_single)

output:

is_score_double（is_score)，show_fail

description:

target score denotes the score should achieve to pass current game

state_out come from hook.sv, help decide whether some objects disappear because of successfully catching or explode.

catch1-14 is the series of signal come from each object, for high means the corresponding object is catching, different score should be added after successfully catching due to catch signal.

is_score denote whether the current pixel belong to score

time_end come form timer, raise to high when timer reaches 0 form 50

show_fail raise when time end, and current score smaller than our target score.

purpose:

These two modules update score and check whether player pass or fail at the end of game. The difference between single and double player mode is, at double player mode, we maintain three scores show on the screen, one individual score for each player and one total score, which is complicated than single player mode.

module: shop.sv

input:

Clk, reset, left, right, space, display_shop,

[9:0] DrawX, DrawY,

output:

[3:0]shop_index, bomb_index, water_index, next_index ,boss_index, badboss_index,

is_shop,is_bomb,is_water,is_next,is_boss,is_badboss,bomb_num,water_num,

description:

left, right, space are used to select different goods and buy.

bomb_num and water_num raise to1 denote player has bought our explosive or water from shop, so in next game, players are able to use them.

purpose:

This module performs as a shop, which we complement a selection animation like we choose game mode at start page. Addition, we say each time players can buy each good at most one, so after they buy, the corresponding good will stop blinking and disappear. And if player do not by anything, a bad face boss will show up.

module: shop.sv

input:

Clk, reset,

[9:0] DrawX, DrawY,

output:

[3:0] background_shop_index

purpose:

This module constructs the background of shop, actually we can generate it just into above shop.sv.

module: fail_or_pass.sv
input:
Clk, reset, show_fail, time_end,
[9:0] DrawX, DrawY,
output:
[3:0] fail_or_pass_index
is_failorpass, show_shop
description:
show_fail come from scorekeeper, high denote player do not pass game and we should show lose for him.
time_end high denotes game end
show_shop high denotes next shop page should be displayed.
purpose:
This is another static page after each round ends, we get two possible case, if player get enough score and pass, we show congratulation, otherwise show lose.

module: VGA_controller.sv
input:
Clk, reset, VGA_CLK
output:
[9:0] DrawX, DrawY,
VGA_HS，VGA_VS, VGA_BLANK_N，VGA_SYNC_N,
purpose:
This module controls the pixel update, output the current update pixel position with x and y.

module: HexDriver.sv
input:
[3:0] In0,
output:
[6:0] Out0
purpose:
This module is used for debug, we can display some signal on FPGA with

hex or LEDs through this hexdriver.

module: color_mapper.sv
input:
Clk, is_xx, is_explode_show
[3:0]xx_index, explode_index
[9:0] DrawX, DrawY,
output:
[7:0] VGA_R, VGA_G, VGA_B
description:
is_xx denote all feature need to draw like hook, golds, stone and so on.
is_explode_show high for an explosive animation to show.
VGA_R, VGA_G, VGA_B denote the color should display at current pixel.
purpose:
This is our color mapper for whole game, decide the color of each pixel. Individual palettes are defined for each feature here, using each index to trace.

module: Timer.sv
input:
Clk,reset,display_random,display_single,display_double
[9:0] DrawX, DrawY,
output:
is_timer, time_end,
[5:0] timer
description:
display_random high denote new game will start at next clock, so timer reset to 50, this signal just works as another weak reset.
display_single or display_double raise to high means game start, so timer begin to decrease, otherwise, keep 0 or 50
time_end raise to high when timer become 0, output to score keeper.
purpose:
This is the timer for whole game to control the time for each round, game stop exactly when time up.

module: keycode_manager.sv
input:
[31:0] keycode, keycode0, keycode1, keycode2,

output:

player1_want_catch, player2_want_catch, left, right, enter, space, upper, upper2

description:

Four keycode signals denote we can use at most four keyboard buttons at same time, each store one.

player1_want_catch raise when press “s”, player2_want_catch for “PgDn”, left for “a”, right for “d”, enter for “enter”, space for “space”, upper for “w”, upper2 for “PgUp”.

purpose:

This module transforms keyboard signal to the signal we use in other control modules.

module: random_variable_processor.sv

input:

Clk

[31:0] position0-36

output:

[9:0] goldx0-18， goldy0-18,

description:

position series denote the randomly decided position information from c code.

goldx and goldy is the corresponding position of gold, stone on screen,

purpose:

This module gets the position initialized form c code and transforms it to actual position on screen.

module: game_controller.sv

input:

Clk, reset, time_end2_single, show_fail_single, time_end2_double, how_fail_double, enter

[1:0] game_mode,

output:

display_game_start, display_random, display_target, display_single, display_double, display_fail, display_pass, display_shop

description:

time_end denote game end

show_fail raise when player does not reach the target score.

display_game_start high to show start page.

display_random high only for one clock to prepare for objects initialization.

display_target high to show target page

display_single, display_double high due to game mode choice

display_fail, display_pass high due to player reach target score or not.

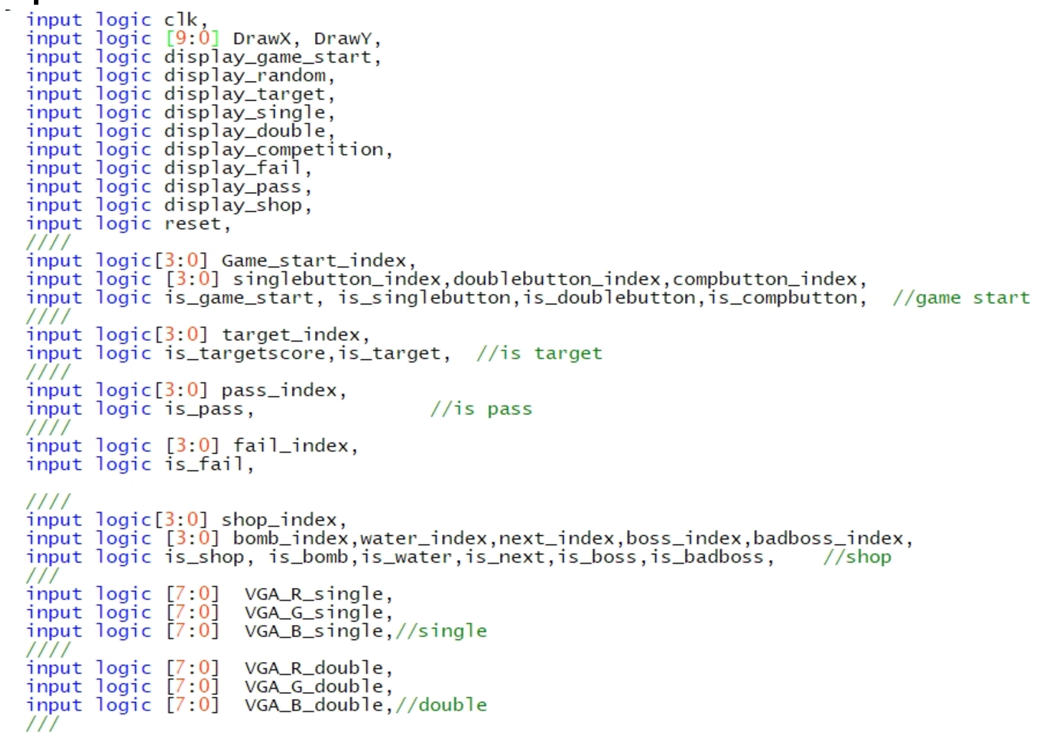
display_shop to show show page

purpose:

This module describes the whole logic of our game, which use a state machine to control state change from start game, choose mode, see target, game, settle and shop. The detailed state machine description will be discussed later.

module: color_mapper_manager.sv

input:



```
input logic clk,
input logic [9:0] DrawX, DrawY,
input logic display_game_start,
input logic display_random,
input logic display_target,
input logic display_single,
input logic display_double,
input logic display_competition,
input logic display_fail,
input logic display_pass,
input logic display_shop,
input logic reset,
////
input logic[3:0] Game_start_index,
input logic [3:0] singlebutton_index,doublebutton_index,compbutton_index,
input logic is_game_start, is_singlebutton,is_doublebutton,is_compbutton,  //game start
////
input logic[3:0] target_index,
input logic is_targetscore,is_target,  //is target
////
input logic[3:0] pass_index,
input logic is_pass,              //is pass
////
input logic [3:0] fail_index,
input logic is_fail,

////
input logic[3:0] shop_index,
input logic [3:0] bomb_index,water_index,next_index,boss_index,badboss_index,
input logic is_shop, is_bomb,is_water,is_next,is_boss,is_badboss,    //shop
///
input logic [7:0]  VGA_R_single,
input logic [7:0]  VGA_G_single,
input logic [7:0]  VGA_B_single,//single
////
input logic [7:0]  VGA_R_double,
input logic [7:0]  VGA_G_double,
input logic [7:0]  VGA_B_double,//double
///
```

output:

[7:0] VGA_R, VGA_G, VGA_B

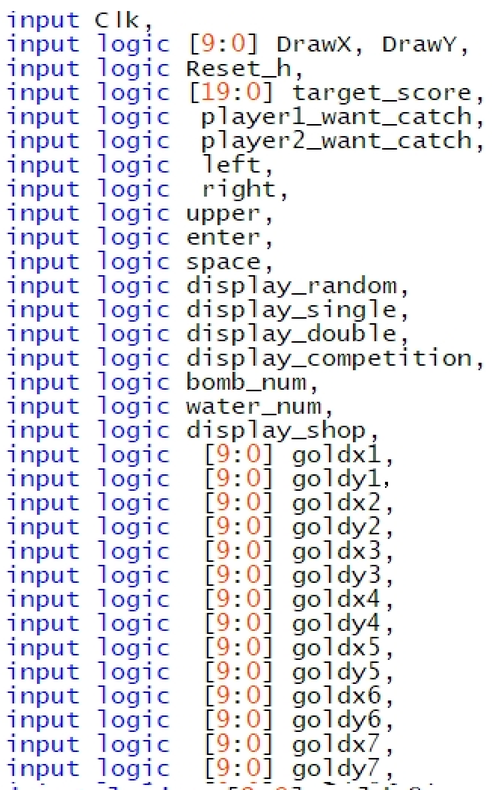
display_shop to show show page

purpose:

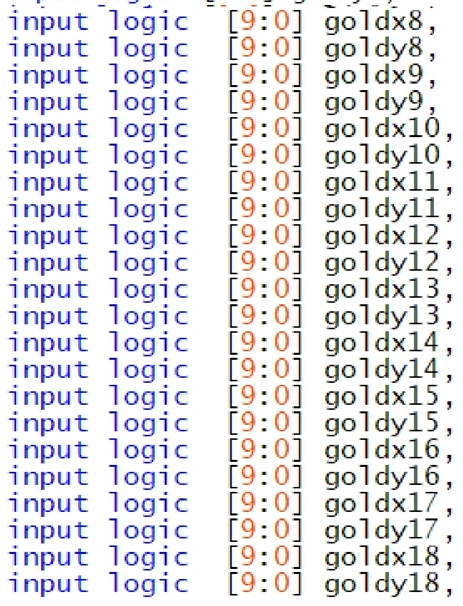
This is the color mapper for single.sv and double.sv to generate background figure, score, timer, hook, string and objects advance to decrease the complexity of the total color mapper.

module: single.sv/ double.sv

input:

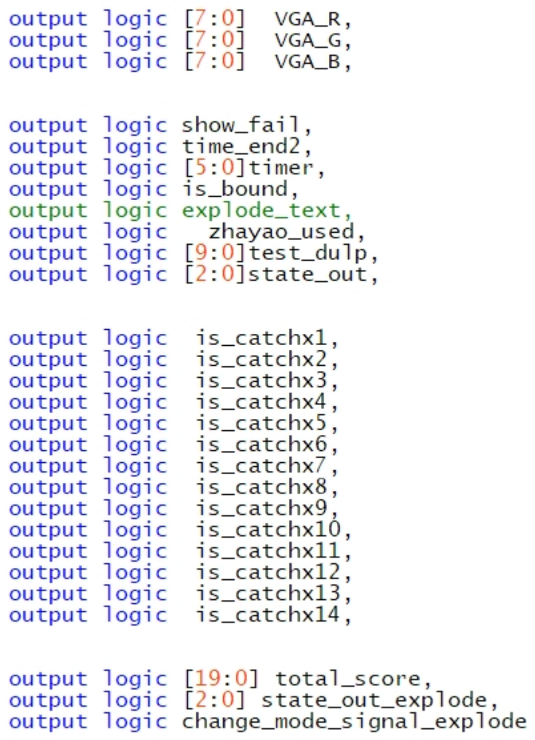


```
input Clk,
input logic [9:0] DrawX, DrawY,
input logic Reset_h,
input logic [19:0] target_score,
input logic  player1_want_catch,
input logic  player2_want_catch,
input logic  left,
input logic  right,
input logic upper,
input logic enter,
input logic space,
input logic display_random,
input logic display_single,
input logic display_double,
input logic display_competition,
input logic bomb_num,
input logic water_num,
input logic display_shop,
input logic  [9:0] goldx1,
input logic  [9:0] goldy1,
input logic  [9:0] goldx2,
input logic  [9:0] goldy2,
input logic  [9:0] goldx3,
input logic  [9:0] goldy3,
input logic  [9:0] goldx4,
input logic  [9:0] goldy4,
input logic  [9:0] goldx5,
input logic  [9:0] goldy5,
input logic  [9:0] goldx6,
input logic  [9:0] goldy6,
input logic  [9:0] goldx7,
input logic  [9:0] goldy7,
```



```
input logic  [9:0] goldx8,
input logic  [9:0] goldy8,
input logic  [9:0] goldx9,
input logic  [9:0] goldy9,
input logic  [9:0] goldx10,
input logic  [9:0] goldy10,
input logic  [9:0] goldx11,
input logic  [9:0] goldy11,
input logic  [9:0] goldx12,
input logic  [9:0] goldy12,
input logic  [9:0] goldx13,
input logic  [9:0] goldy13,
input logic  [9:0] goldx14,
input logic  [9:0] goldy14,
input logic  [9:0] goldx15,
input logic  [9:0] goldy15,
input logic  [9:0] goldx16,
input logic  [9:0] goldy16,
input logic  [9:0] goldx17,
input logic  [9:0] goldy17,
input logic  [9:0] goldx18,
input logic  [9:0] goldy18,
```

output:



```
output logic [7:0]  VGA_R,
output logic [7:0]  VGA_G,
output logic [7:0]  VGA_B,

output logic show_fail,
output logic time_end2,
output logic [5:0]timer,
output logic is_bound,
output logic explode_text,
output logic   zhayao_used,
output logic [9:0]test_dulp,
output logic [2:0]state_out,

output logic  is_catchx1,
output logic  is_catchx2,
output logic  is_catchx3,
output logic  is_catchx4,
output logic  is_catchx5,
output logic  is_catchx6,
output logic  is_catchx7,
output logic  is_catchx8,
output logic  is_catchx9,
output logic  is_catchx10,
output logic  is_catchx11,
output logic  is_catchx12,
output logic  is_catchx13,
output logic  is_catchx14,

output logic [19:0] total_score,
output logic [2:0] state_out_explode,
output logic change_mode_signal_explode
```

description:

all the input and output signal come from individual module like gold.sv, timer.sv and so on. So we do not repeat them here again.

purpose:

This module generates all the feature in single player game mode, include hook, objects, timer, score keeper and so on. For double, they are quite similar, instead of using some feature twice compare with single like hook.

module: Pass.sv

input:

Clk, reset, display_pass,

[9:0] DrawX, DrawY,

output:

[3:0] pass_index, is_pass

purpose:

This module shows pass game background when game end and player reach our target score.

module: Fail.sv

input:

Clk, reset, display_fail,

[9:0] DrawX, DrawY,

output:

[3:0] fail_index, is_ fail

purpose:

This module shows fail game background when game end and player does not reach our target score.

module: show_explode.sv

input:

Clk, reset, is_explode, display_random,

[9:0] DrawX, DrawY, taily,tailx,

output:

[3:0] explode_index,

[2:0] state_out_explode,

is_explode_show,

purpose:

This module construct the animation of explosive.

Design Procedure / State Diagram / Simulation Waveform

0. basic information

Our project does not contain other' s works.

For input: we use keyboard as input. We support four keycodes a t a time.

For state machine: Our project is based on state machines. We use state machines in all modules.

For sprites: As for us, one sprite is accompanied with one module, one block of memory. For examples, big gold has its own picture, its own module, its own in chip memory block.

We did lots of background studies. For examples, we learned the design patterns, the way to rotate, the way to judge whether a single point is in the region or not by using cross product.

Please see the block diagram part to find out how different objectives are linked together to form a complete project.

1. Key idea:

(1) object-oriented programming, we consider a module as a class. So, hook, gold, stone are just objects and we create a module for each of them.

(2) Program to an interface, not an implementation. We always strive for loosely coupled designs.

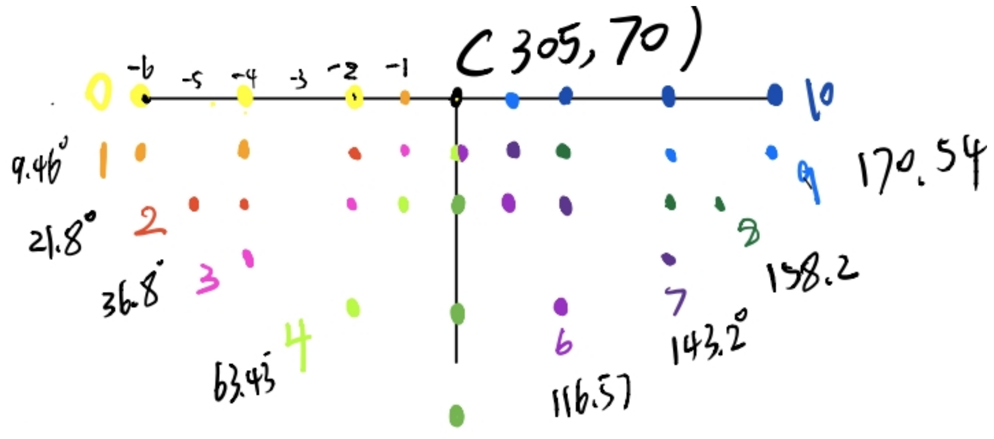
(3) Using "Observer Pattern". When hook changes its state (from shrink to rotate for example), it will send the change signal to all its Observers (like

gold, stone, diamond). Hook doesn't need to know the concrete class of the observer, what it does, or anything else about it. Hook only contain a list of observer's signals.

(4) Using "Command Pattern". We create a module called "show_explode". That is, we encapsulate method invocation. In the single mode, when player press "W", it means player wants to below up the stone. However, in the competition mode, "W" may mean totally different things. By using Command Pattern, we can easily change the real actions. (Actually, since we only have limited time, we have not finished the competition mode).

(5) We make great effort to follow the rule: Classes should be open for extension but closed for modification. At first, we only consider the single mode. So gold, stone objects only need to care about one hook. However, in the Two-Player mode, gold, stone objects need to care about two hooks, and once caught by one hook, it should ignore another hook. This sounds a huge modification and may need huge time to test the new module. As a result, we want to add something instead of modification. Our method is that we can create a big module that contains two original gold instances in same position. One only cares about the left hook, another only cares about the right hook. Once a gold is caught, another gold disappears immediately (This method is great but need more memory to store the image since we do not share image for different golds).

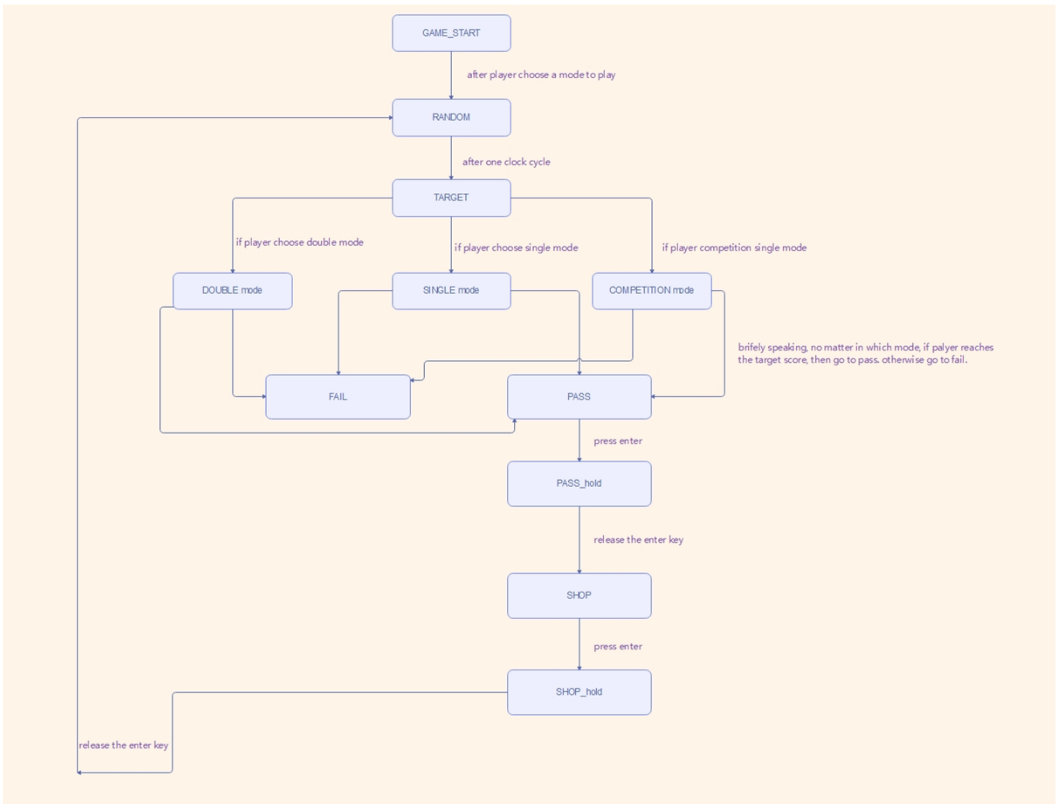
(6) Using 11 states (each state has four dot) to simulate the rotation of the hook. Below is the picture. Each color represent one states. When the hook is in extension. We just calculate if (drawx, drawy) is on the one of the lines between the dots and the origin points.



2. structure of the project:

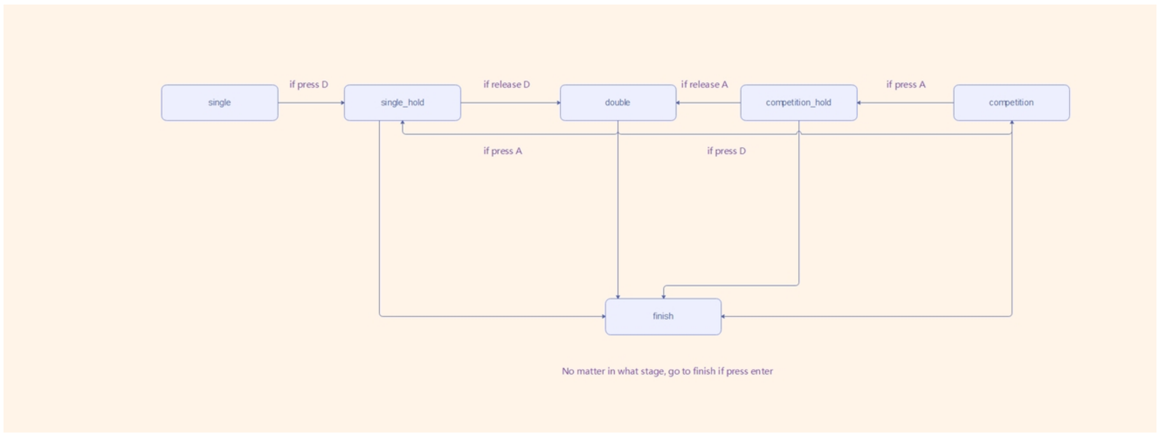
(1) The biggest state machine controls the scene of the game. It is in the module game controller.

1. GAME_START shows the start interface and player can choose the playing mode by using keyboard.
2. RANDOM shows nothing but to give a random location to every gold, stone, diamond.
3. TARGET shows the target score that the players need to achieve to continue playing.
4. In SINGLE, DOUBLE, COMPETITION mode, players can control the hooks by using keyboard and use the explosive and energy water bought from the shop to help them to win.
5. FAIL shows the fails scene.
6. PASS shows the PASS message and will go to shop scene after you press enter. Here we have to make a PASS_HOLD. This is because players may press the enter key for many clock cycles, so we have to consider "press and release" as a signal to change the states.
7. shop shows the shop scene. Player here can buy explosive and energy water. After player press and release enter, we will go to random stage to have next game level.


GAME_START
after player choose a mode to play
RANDOM
after one clock cycle
TARGET
if player choose double mode
if player choose single mode
if player competition single mode
DOUBLE mode
SINGLE mode
COMPETITION mode
brifely speaking, no matter in which mode, if palyer reaches the target score, then go to pass. otherwise go to fail.
FAIL
PASS
press enter
PASS_hold
release the enter key
SHOP
press enter
SHOP_hold
release the enter key

(2) state machine of Game_start.

single,double, competition means we are current point at the single, double,



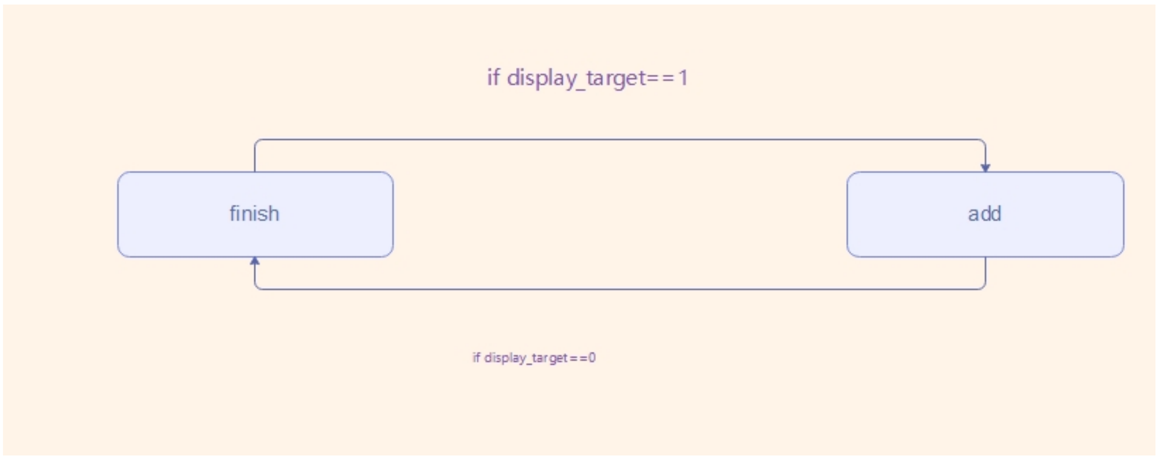
competition button, the button will flash if you point at it. We need single hold; competition hold states because we can only change the state if player press and release the key.

(3) state machine of Target.

Target will receive the signal from game controller. If game controller wants the Target instance to show the score, then we just do the calculation:

target score=80*level*level

and then give all the message to color mapper manager.



(4) SINGLE,DOUBLE,COMPETITION mode -------the main part

Here is how we achieve:

We create three modules called SINGLE,DOUBLE,COMPETITION. Each module has its own color mapper, its own gold instances, its own stone instances. To make a long story short, these three modules are completely independent. Each of them can run as a independent game since they have their own color mapper, their own on chip memory. What we do is we create a game controller and a color mapper manager to handle the three modules and their local color mapper. We will discuses how these components linked together in the Block diagram part.

(5) state machine of Hook in the SINGLE, DOUBLE, COMPETITION mode

module hook is quite complex. It contains two state machines.

The bigger one is:

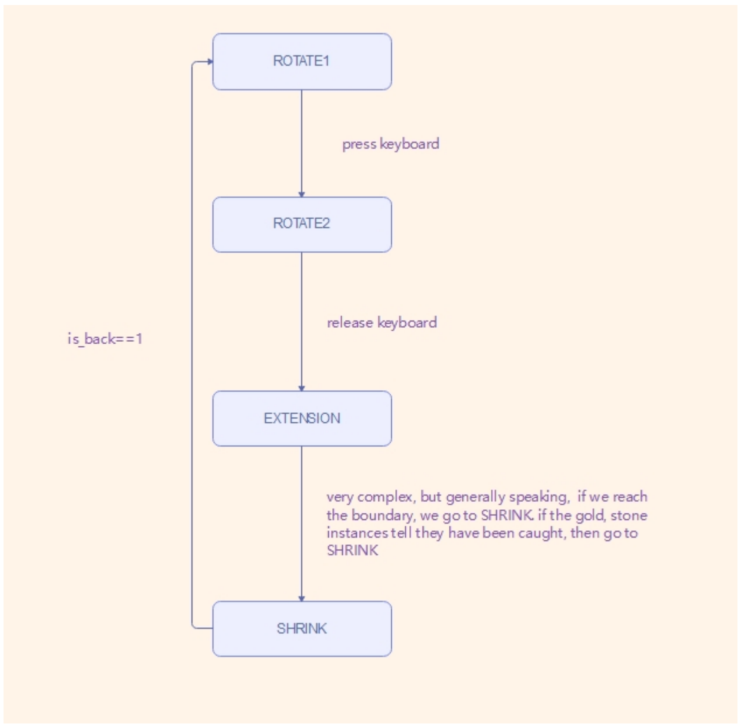
ROTATE1: meaning hook is rotating

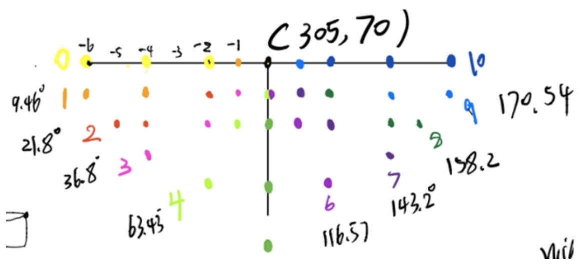
ROTATE2：meaning hook is rotating，but players have already pressed the key.

EXTENSION: hook start to extent. Players have already released the key.

SHRINK: hook begin to shrink. Notice that it is gold instances that tell the hook that it has caught the gold. The difficulty here is that for different golds or stones, hook should go back in different speed. For example, if hook catches big gold, it should have a relatively low speed. We achieve this by maintain a global counter and a local counter. local counter will plus one as soon as the clock changes. global counter will change if local counter reaches threshold. For different golds and stones, we have different threshold.

The smaller state machine is r_mode. It simulates the rotation. Every 0.2 seconds, it will change to the next r_mode. As a result, the hook will look like it is rotating.


ROTATE1
press keyboard
ROTATE2
release keyboard
is_back==1
EXTENSION
very complex, but generally speaking, if we reach the boundary, we go to SHRINK. if the gold, stone instances tell they have been caught, then go to SHRINK
SHRINK



(6) state machine of gold in the SINGLE, DOUBLE, COMPETITION mode

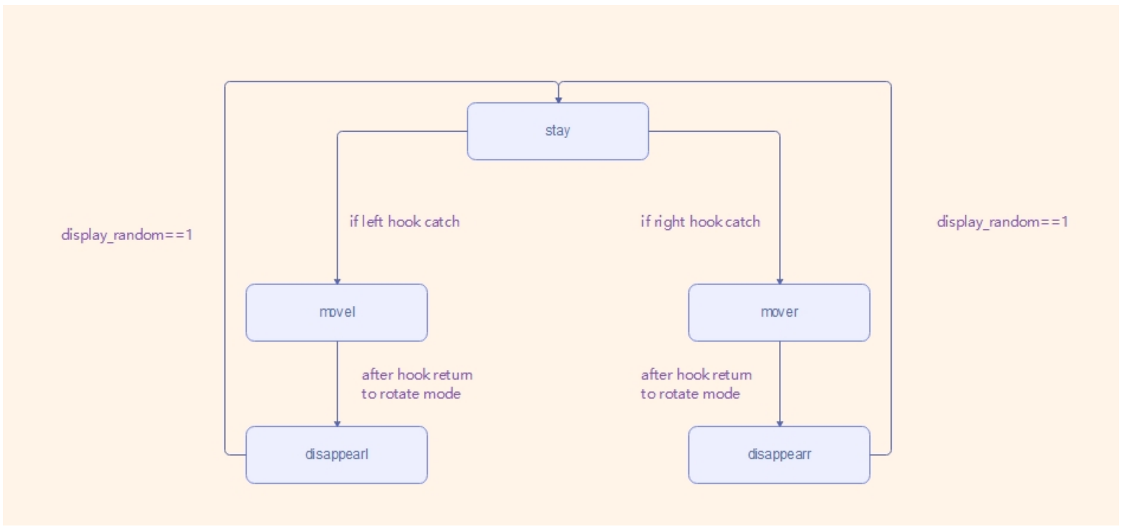
We have big gold module, middle gold module, small gold module, big stone module……

They are actually the same. So, we will only discuss big gold module as an example.

stay: big gold does nothing

movel && mover: big gold is caught by hook and will move with the same speed of hook. Notice that the judgement of whether caught is done in gold.sv. The movement of hook and gold is independent.

Disappear&&disappear: tell the local color mapper to not show the gold. If display_random==1, we should reset all. This is because display_random tells us we are going to go to next game level.



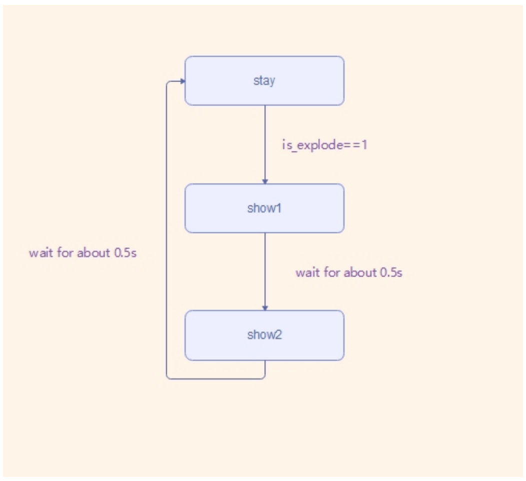
(7) state machine of show_explode in the SINGLE,DOUBLE,COMPETITION mode

Notice that we create a module for showing the explode. In this way, this part is independent to golds or stones. Actually, it only gets the location of hook. And show exploded image at hook' s location.

Stay: show nothing

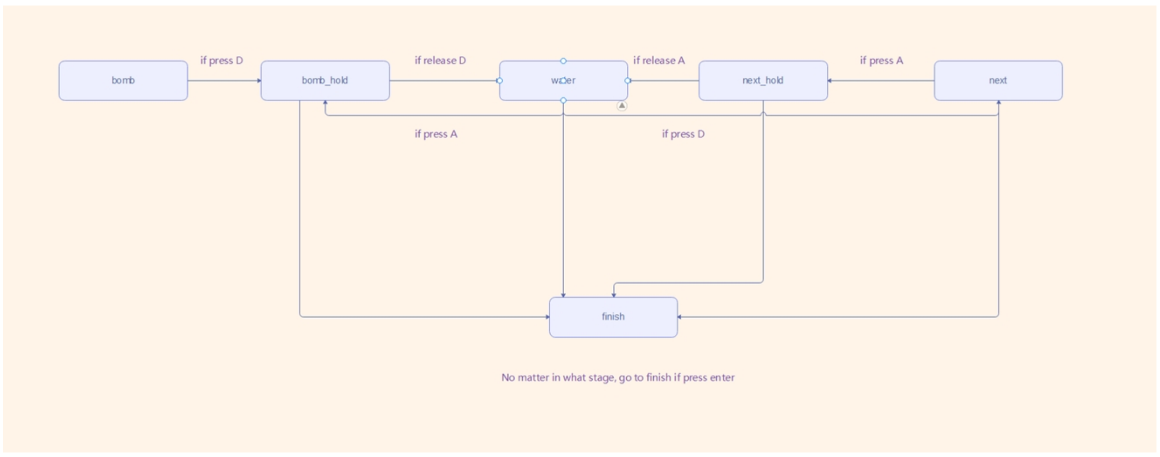
show1: show the small image. It is the initial stage of the explosion

show2: show the big image. It is the peak of the explosion

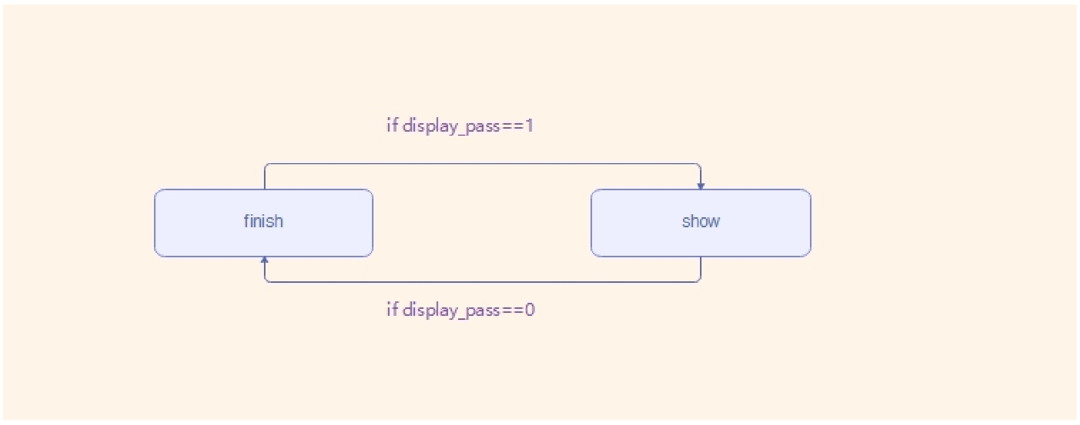


(8) state machine of shop.

bomb, water, next means we are current point at the bomb button, water button, next level button, the button will flash if you point at it. We need bomb hold, next hold states because we can only change the state if player press and release the key. Notice that you can buy the bomb and energy water by press space.



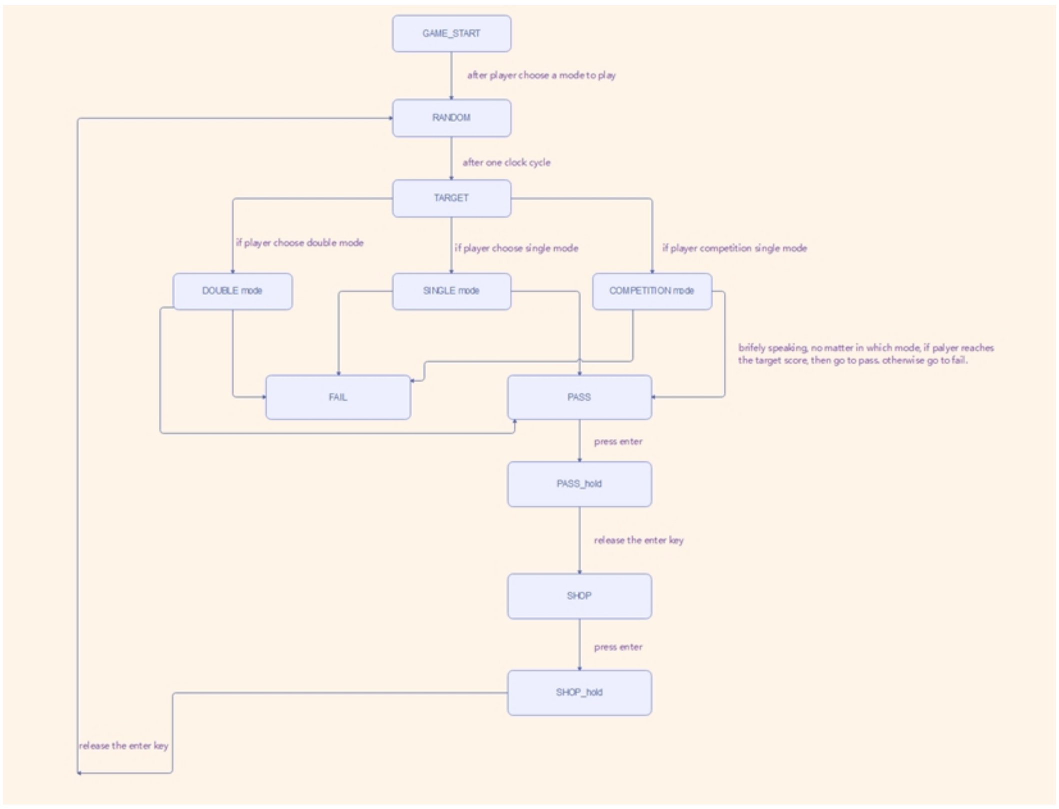
Pass module is quite simple, it is controlled by game controller. When game controller goes to pass state, our pass instance will go to show state. Otherwise, just stay in the finish state.



(9) structure of four color mappers && state machine of color mapper

We call the biggest color mapper as manager. The three-color mappers in SINGLE, DOUBLE, COMPETITION as local color mappers.

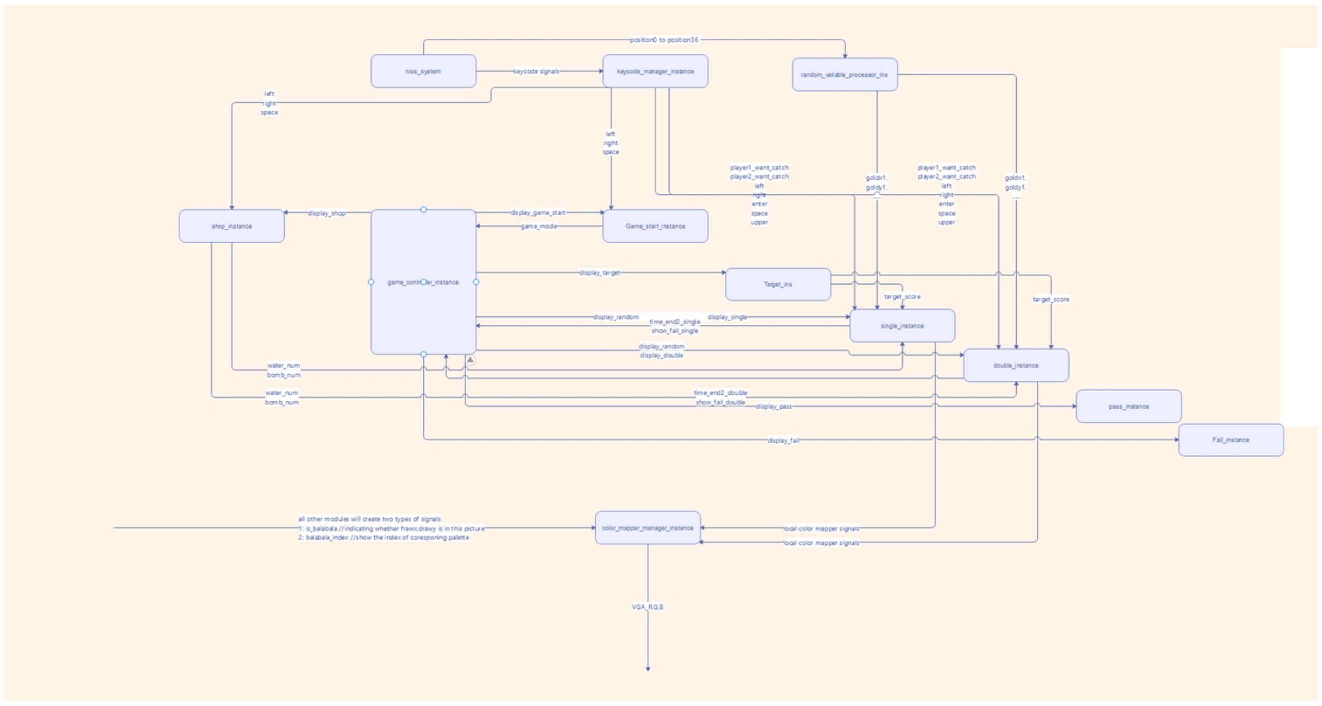
Below is the state machine of manager. It is the same as game controller. That is, different sate will change the way that manager works. Manager will do nothing but output the signal from local color mapper if the manager is in SINGLE,DOUBLE,COMPETITION state.


GAME_START
after player choose a mode to play
RANDOM
after one clock cycle
TARGET
if player choose double mode
if player choose single mode
if player competition single mode
DOUBLE mode
SINGLE mode
COMPETITION mode
brifely speaking, no matter in which mode, if palyer reaches the target score, then go to pass. otherwise go to fail.
FAIL
PASS
press enter
PASS_hold
release the enter key
SHOP
press enter
SHOP_hold
release the enter key

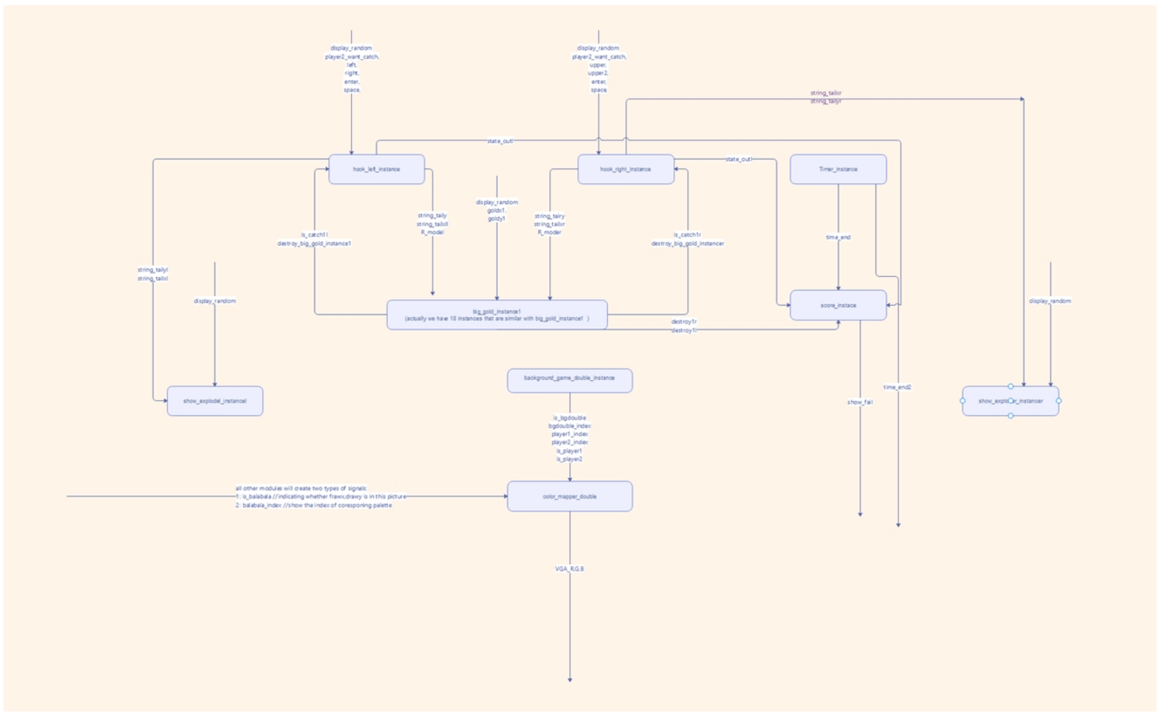
Block Diagram

In order to make everything clear, we will only show one gold instance in the block diagram. And we will ignore hpi_io, vga_clk, vga_controller modules since they are discussed in lab8. We will also ignore DrawX, DrawY since they are widely used in every module.

Below is the top-level block diagram (SINGLE,DOUBLE are huge modules which will show in other picture, COMPETITION module has not been done due to the limited time.):



Below is the block diagram for DOUBLE (diagram for SINGLE is very similar to diagram for DOUBLE, so we will just show one of them).



C Code

Compared to lab8, we only did a little modification to support four keycodes. Here is what we added:

```
            // The first two keycodes are stored in 0x051E. Other keycodes are in

        // subsequent addresses.


        keycode = UsbRead(0x051e);

        keycode2 = UsbRead(0x0520);


        printf("\nfirst keycode values are %04x\n",keycode<<24>>24);

        printf("\nsecond two keycode values are %04x\n",keycode>>8);

        // We only need the first keycode, which is at the lower byte of keycode.

        // Send the keycode to hardware via PIO.

        *keycode_base =  keycode<<24>>24 ;//<<16) + keycode & 0xff;

        *keycode_base0=keycode>>8;

        *keycode_base1=keycode2>>8;

        *keycode_base2=keycode2<<24>>24;


        usleep(200);//usleep(5000);

        usb_ctl_val = UsbRead(ctl_reg);





            *position_0 =rand(  );

            *position_1 = rand(  );

            *position_2 =rand(  );

            *position_3 = rand(  );

            *position_4 = rand(  );

            *position_5 = rand(  );

            *position_6 = rand(  );

            *position_7 = rand(  );

            *position_8 = rand(  );

            *position_9 = rand(  );
            *position_10 = rand(  );

            *position_11= rand(  );

            *position_12= rand(  );

            *position_13= rand(  );

            *position_14= rand(  );

            *position_15= rand(  );

            *position_16= rand(  );

            *position_17= rand(  );

            *position_18= rand(  );

            *position_19= rand(  );

            *position_20= rand(  );


            *position_21 =rand(  );

            *position_22 = rand(  );

            *position_23 =rand(  );

            *position_24 = rand(  );

            *position_25 = rand(  );

            *position_26 = rand(  );


            *position_27 = rand(  );

            *position_28 = rand(  );

            *position_29 = rand(  );

            *position_30 = rand(  );

            *position_31 = rand(  );

            *position_32= rand(  );

            *position_33= rand(  );

            *position_34= rand(  );

            *position_35= rand(  );

            *position_36= rand(  );
```

Design Statistics and Discussion

LUT	56572
DSP	0
Total RAM block bits	0
FLIP-FLOP	8193
Frequency	125.6 MHz
Static Power	108.36 mW
Dynamic Power	0.80 mW
Total Power	204.80 mW

Conclusion

For this final project, we spend more than 100 hours to finish and get a good not bad demo presentation. Actually, Gold Miner is much harder to construct than we ever thought so that we have to decrease some feature planed before as mouse and sound. The logic of game is quite clear, we catch gold and get score, but implementation is rather complexity and long. First, before start we thought to use rotation matrix to describe rotate, however float time are not allowed here in system Verilog, so we have to make a pseudo-rotate instead. The rotate direction is a balance of code length and game experience, for more rotate direction, our hook can reach more space on the screen, but code will be quite longer.

Also, we realize the perniciousness of bad organization of code, at beginning, when we want some more feature, we just add always_ff or always_comb at the end of what we finish last time. However, as logic become more complicated, it is hard to debug, so we decided to rewrite some of code with different state machine. Through this project, we realize the good behavior of state machine and get more familiar with it.

Also, there are some left pitiful thing, such as try to work on mouse and sound. Also, we firstly design 3 different mode, but finally finished 2 and leaving the third mode on start page empty.

Generally, we successfully finish this project, and not bad. For we make the game entertaining and realize infinite level. And one improvement can be done is the quality of our figure and animation. With this project, we build a good quality design with FPGA, coding system Verilog, do better debug it, and get some real example of design hardware.