1D Project Group Report

Electronic Game Prototype

Team Number: 14-3

**Instructions*:

1. *Directly work on your report in this Google doc. Do not work elsewhere and paste it here.*
2. *Use the template formatting, if it is a heading (e.g., Heading 1, Heading 2 etc.), format it as such so the Table of Contents will reflect it.*
3. *The point of this Google doc is to be a live document that you continually update as you work on your 1D project. The teaching staff will be taking a look at these documents as you work on them to see how your team is progressing.*

Table of Contents

Introduction

With the advent of widespread mobile device usage in the beginning of the 21st century, “Snake” is a game that has become a commonplace name in our society. It is the well-known video game concept, whereby the participant controls a line with the ability to grow in length from moving over dots. The line length and 4 walls become the main obstacles for the “snake” to eat more of the dots representing food. What lies in the major success of this game genre is the simplicity of its design, which managed to captivate many due to its addictive nature. In this computation structures 1D project, our group has decided to take on the challenge of replicating this classic yet popular conception in our prototype.

Design

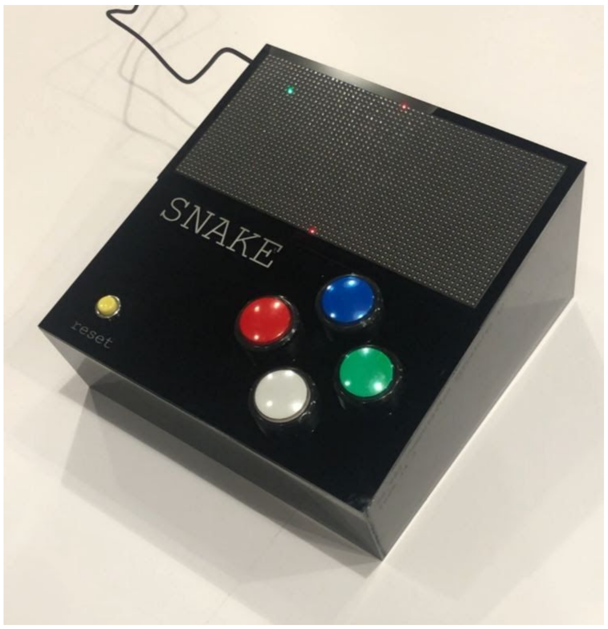
Description of the Game

The solo player attempts to manoeuvre the head of the Snake, with directional buttons, into food pieces randomly generated on the playing field one at a time. Each food item eaten results in the increased length of the Snake, as well as the generation of another food item at another location. Once the snake grows to 10 units long, the player wins. If the Snake collides with itself or the wall, the player loses. The player can press the reset button to start a new game if he wishes to do so.
The game is built on a custom 16-bit CPU and assembly code. The UI is composed of a 32*64 LED matrix, 4 direction buttons and 1 restart button.

Design Inspirations

Game Idea Inspiration:
Many of our group members are filled with nostalgia whenever we think of the game “Snake”. This video game reached viral status in the year of 2000, with the launch of the iconic Nokia 3310. Being one of the most popular mobile devices of all time, Nokia went on to sell 126 million units of that device in merely a year. The snake game resided within that mobile phone, thus it rapidly became a household name globally. For our generation, it was one of the first video games we had contact with; albeit on our parents’ mobile devices:P. Thus, we have decided to recreate that experience in this project.

We find that even if the rules of the snake game are quite simple, the game logics behind it are not that trivial. For example, we need to keep track of the snake’s each body unit and the fruit’s position. We need to check whether the snake is eating itself or colliding with the wall. We have to update the snake’s position and randomly generate new fruit when the previous fruit has been eaten. That’s why we find it a great idea for our 1D project, for it is both entertaining and relatively difficult to implement, which will enable us to score high marks if we can build it successfully.



Hardware Design Inspiration:
Given that this game is in the retro genre, we have decided to make a classic arcade machine themed housing.
Thus, for the choice of material, we have selected glossy black acrylic, to be laser cut and glued to fit together. As for the physical input into the machine, arcade style buttons were bought to fit the theme. We used a 64 by 32 LED matrix as our display.
Most of the effort went into creating a well made housing, precisely fitted to all the various components and pieces for a polished finish.

Test Scenarios

There are several scenarios to test to ensure that our game works as designed. They can be split into the following categories:

Control:

The directional buttons (up, down, left, right) should be responsive, it should feel intuitive and in-place when moving the snake in game. The buttons should move the snake accordingly to what direction it is intended for. Up button should move the snake upwards, down should move the snake downwards, so on and so forth. The directional buttons should only change the direction the snake is moving, instead of resetting the game or anything else.

One particular case should not happen; the snake cannot turn backwards against the direction it was originally moving(for example, if the snake is moving from the left to right, it cannot start to move left immediately after pressing the left button). What should occur is that a collision is detected, hence stopping the game since one of the loss conditions are fulfilled.

The reset button should reset the game whenever it is pressed; nothing else.
Eating mechanic:

Whenever the head of the snake passes through the food item, these should occur:
1) Green food unit on the screen disappears.

2) Red snake grows by one unit.
3) Another green food unit should generate at a random location, where the snake is not located at

Once the snake reaches a length of 10 units, the win condition is fulfilled, and as a result the whole LED matrix turns blank. The game can then be started again by pressing the reset button.

Lose conditions:

There are two loss conditions:

1) Collision with the wall edges of the LED matrix. Whenever the head of the snake passes the boundary of the edge of the LED matrix, the loss condition is triggered, resulting in the game being frozen in that current position; the snake stops moving.
2) Collision with the snake itself. Whenever the head of the snake passes through its own body, the loss condition is triggered, resulting in the game being frozen in that current position; the snake stops moving.

Display:
The LED matrix should display what is intended at all times and for all instances. The snake should only be made up of red LEDs while the food units should be made up of Green. No other parts of the LED matrix other than the snake and foot unit should be lit.

User Manual

The prototype is rather easy and intuitive to use.

Game objectives:
Manoeuvre the snake to reach a length of 10 units without colliding with itself or the walls. The snake grows longer by one unit after the head of the snake passes through a green food unit.

Starting the Game:
Plug the power into the unit to start the game.
Use the directional buttons to change the direction of the snake.
Press the reset button to restart the game; anytime you want.

Budget

Component	Purpose	Quantity	Total Price
32 * 64 LED matrix	Game display	1	24
Big arcade buttons	Game input	4	27.20
Small buttons	Game input	4	14
Acrylic glue	Game enclosure	1	4.20
Wire	Game connections	1	3

Summary

Our game has effectively at applied concepts taught during the
50.002 Computation Structures course, using the Alchitry Au FPGA Board to design a computer architecture to support our designed functionality. In particular, we have managed to **design a CPU of our own** to implement the logic of our digital game, which is a distinguished extra mile we would like to highlight. (Detailed info of game implementation can be found under the appendix)

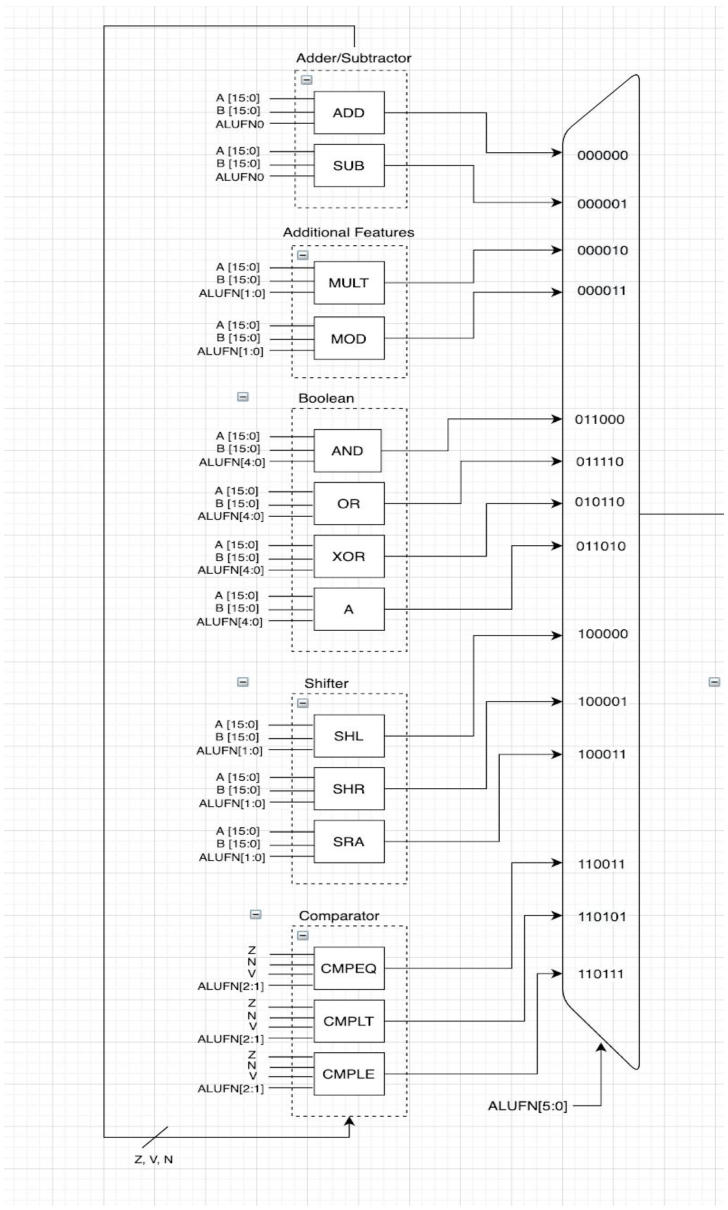
The logic of the game we created, snake, is not as straightforward to implement as many other digital games since the user could arbitrarily move the snake around on the board. Much programming and debugging difficulties are also met due to the sophisticacity of the hardware programming using FPGA. Nonetheless, with a strong perseverance in moving the project forward and a supportive collaboration in our team, all the features are delivered in full functionalities as expected.

References

1. Getting Started With FPGA Part 1: Combinational Logic
2. Getting Started With FPGA Part 2: Sequential Logic and FSM
3. Getting Started With FPGA Part 3: Reset and I/O
4. Getting Good with FPGA: Building the Beta CPU
5. Sample Au projects prepared by Prof. Natalie.
6. BSim Documentation.

Appendix

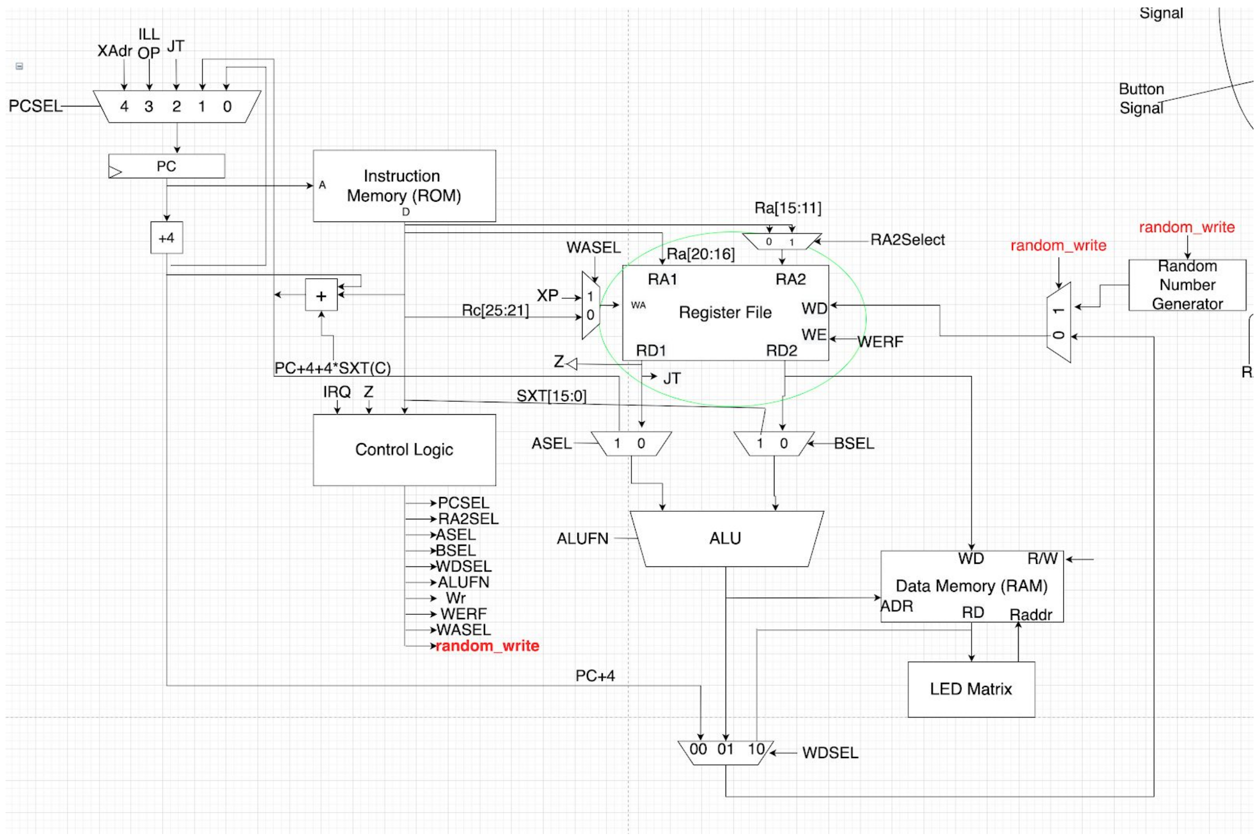
1. ALU Design and Tests



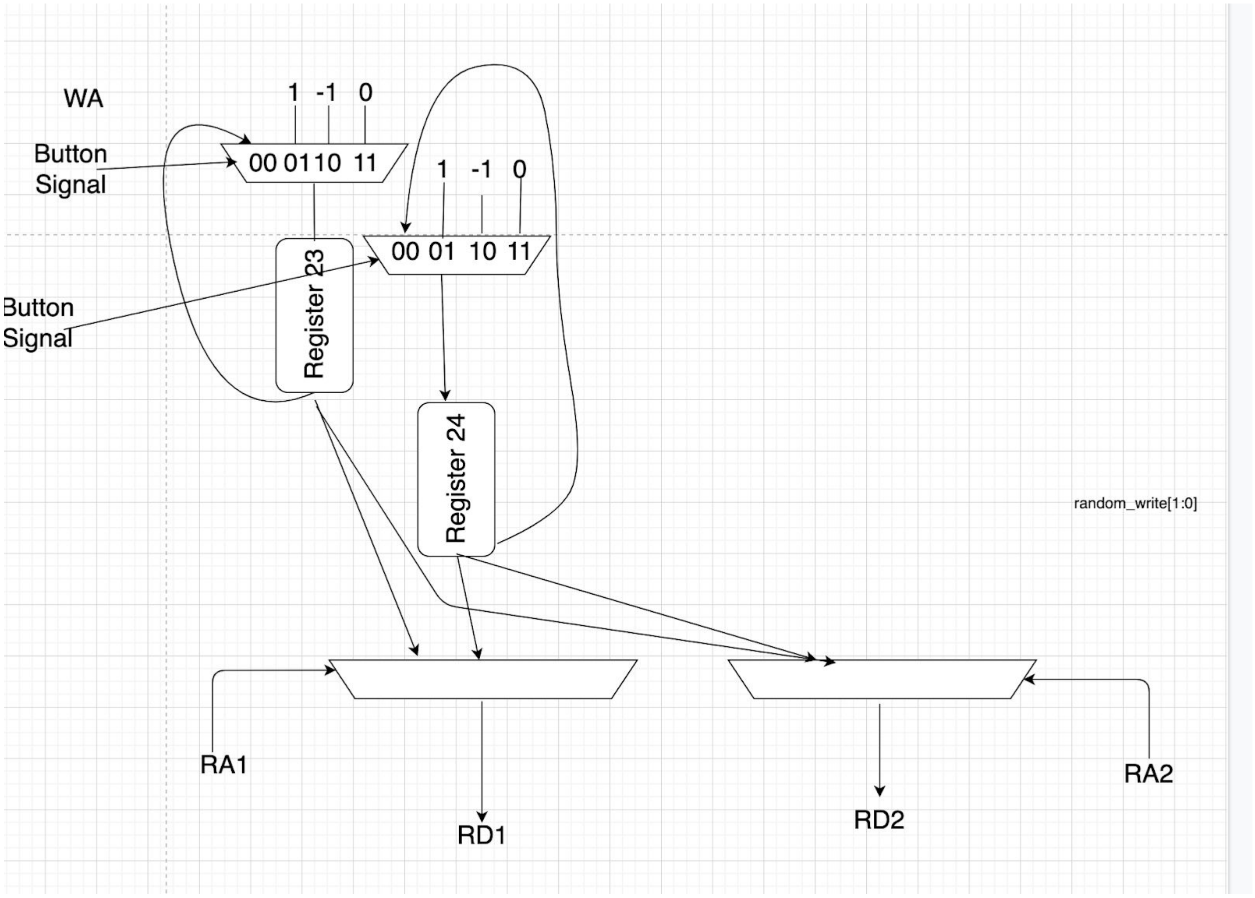
Test cases:

Operation tested	A	B	Expected output
ADD	0xFFFF	0x0004	0x0003
SUB and **n signal**	0xFFFF	0x0004	0xFFFB (n == 1)
MULT	0xFFFF	0x0004	0xFFFC
AND	0xFFFF	0x0004	0X0004
OR	0xFFFF	0x0004	0XFFFF
XOR	0xFFFF	0x0004	0XFFFB
A	0xFFFF	0x0004	0XFFFF
SHL	0xFFFF	0x0004	0XFFF0
SHR	0xFFFF	0x0004	0X0FFF
SRA	0xFFFF	0x0004	0XFFFF
CMPEQ	0xFFFF	0x0004	0X0000
CMPLT	0xFFFF	0x0004	0X0001
CMPLE	0xFFFF	0x0004	0X0001
SUB and **v signal**	0x8000	0x0001	0x0000 (v == 1)
SUB and **z signal**	0x8000	0x8000	0x0000 (z == 1)
MOD	0xFFFF	0x0004	0x0001
Circular Shift Left	0x1234	0x0004	0x2341
Circular Shift Right	0x1234	0x0004	0x4123

2. 16-bit CPU Diagram



Compared to the beta cpu, our 16-bit cpu has an additional random number generator unit, which is responsible for generating random number input to the regfile.
Control signal *random_write* will control whether we feed in a random number to the regfile input. It is used to generate a random fruit position when we need a new fruit.
It is set to 0 by default. It is set to 1 only when a RAND(will be covered in ISA Description) instruction is called.



The value of R23 and R24 (which are used to indicate the current snake's direction) is controlled by our four direction buttons. They are used to capture the player's button press and reflect it on the led matrix.

3. ISA Description

Instructions:

Instruction size: 32
Instruction format:

TYPE 1

OPCODE 6 bits	Rc 5 bits	Ra 5 bits	Rb 5 bits	11 bits unused

TYPE 2

OPCODE 6 bits	Rc 5 bits	Ra 5 bits	16 bits **signed** constant

Instruction sets: almost the same as the beta cpu, except that it does not have the JMP instruction. The OPCODE taken by JMP (011011) in beta is mapped to the RAND instruction, which is a new instruction added in our ISA.

RAND instruction:

RAND(Rc) effect: feed in a random 16-bit number to Rc.
Machine code: 011011 xxxxx 11111 11111 00000000000
(Rc is converted to xxxxx. Ra and Rb are set to 31)
Control signal: the same as ADD, except that *random_write* is set to 1.

Control Signal: almost the same as the beta cpu, except for an additional *random_write* signal, which is set to 1 only when a RAND instruction is called.

Data Memory:

The memory unit has 16*64*6 bits in total and it is 6-bit addressable. Which means the address needed is log(16*64) = 10 bits long. Each 6-bit represents the B0 G0 R0 B1 G1 R1 values used to draw the led matrix. B0 G0 R0 corresponds to the upper half of the matrix and B1 G1 R1 corresponds to the lower half. (This is related to the mechanism of the LED matrix of receiving pixel address and color value). The memory unit is always connected to the led matrix with *matrix_writer.luc*. So writing values to the memory unit will immediately result in color display on the led matrix. Setting every bit to 0 will result in a completely blank LED matrix.

To give an example, if I want to set the pixel located at row 16, col 1 to color red. I need to set mem[0000 000001] = 000 001.

In other words, the data memory is only used to drive the LED matrix. It is not used to store any intermediate data for the programme’s later use.

Instruction Memory:
Instruction Memory is implemented with a ROM instead of a RAM. Because if we use RAM, we still need to create a ROM with all the binary instructions and then write the instructions to ROM.

Regfile:
Regfile is composed of 32 16-bit registers, with R31 always equal to 0.

4. Registers' usage

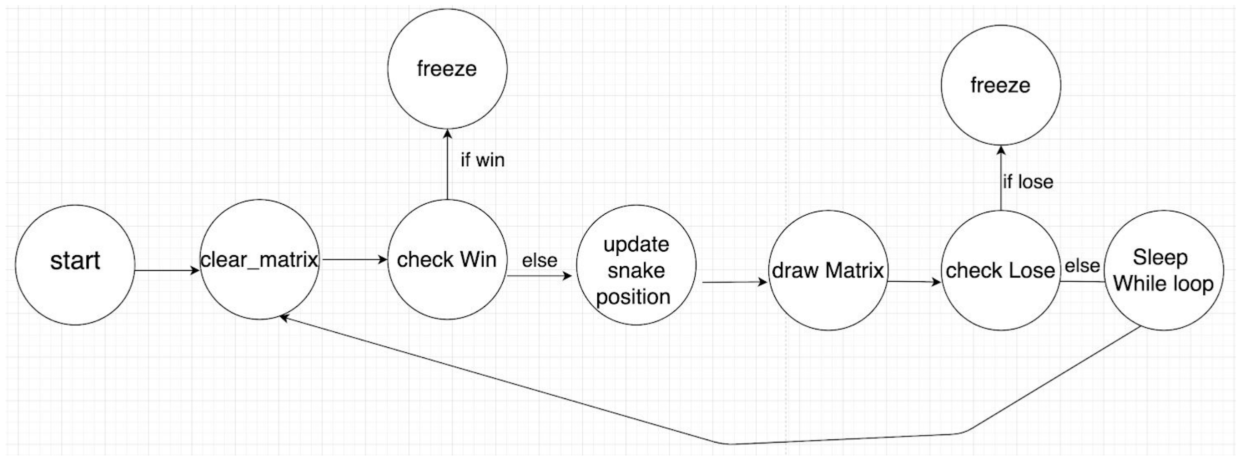
R0: row index of head unit
R1: col index of body unit
R2: row index of body unit 1
R3: col index of body unit 1
R4: row index of body unit 2
R5: col index of body unit 2
So on and so forth till R18 and R19 (body unit 9), so the maximal possible length of the snake is 10, which is also the game's winning condition.

R22: snake's length
R23 and R24: current snake's direction.
 up : R23 = -1, R24 = 0
 down: R23 = 1, R24 = 0
 left: R23 = 0, R24 = -1
 right: R23 = 0, R23 = 1
R25: row index of the fruit
R26: col index of the fruit
R30: snake growing flag:
 R30 == 1 if the snake has eaten a fruit in the last move.
 R30 == 0 if the snake has not eaten a fruit in the last move.
R31 == 0

No fixed use: (used to store intermediate values)
 R20, R21, R27, R28, R29

5. Assembly's Workflow.

The following diagram gives an overview of the assembly code's stages within 1 snake movement.



Whenever the power is connected or the restart button is pressed, the game will start from scratch (the *start* stage), which means all values inside registers are cleared.

Then the programme will clear the led matrix(by clearing the RAM) and check whether the length of the snake satisfies the winning condition. If the winning condition is satisfied, the programme will freeze. Since we have just cleared the matrix, the led matrix will be blank. This is the sign used to tell the player that he has won the game.

If the player has not yet won the game, the programme will update the snake's position stored R0, R1, R2 …. based on the value inside R23 and R24 (the snake's moving direction). Then it will draw the matrix (by storing values in the memory unit) and check if the player has lost the game (if the head has collided with the wall or with the snake's body unit). If the player loses, the programme will freeze. But for this time, since we have just drawn the led matrix, freezing the programme will result in the snake and the fruit just drawn on the screen to be freezed at the same time, which is different from the scenario when the player wins the game and the whole screen becomes blank.

Otherwise, the game will enter the sleep loop, which is a while loop with a counter. The condition set for the counter to exit this while loop is a very large number, so the cpu will be "stuck" here for many clk cycles. This will generate the effect that the snake on the screen will only move for each half second. Besides, the time taken for the programme to run from *clear_matrix* to *draw_matrix* is extremely short. So for out naked eyes, It seems like the LED matrix is never really "cleared".

6. Project Management Log: Team Tasks

Time	Task	People Involved
Week 1 - Week 2	Built the 1-bit full adder for 1D Mini Hardware Project and drew circuit diagram	Hng Yi Yingjie Peiyuan
Week 3 - Week 4	Built the bonus automatic mode on FPGA. Poster and video for submission. Check-off Demo	FPGA auto test: Peiyuan Poster: Ryan Video: Qi Bao
Week 5 - Week 6	Check-off Demo for Bonus	Everyone
Week 7 - Week 8	Implement the FPGA ALU. Brainstorming game ideas.	FPGA ALU: Peiyuan Ideation: Everyone
Week 9 - Week 10	Write our 16-bit CPU Write assembly code to run the snake game Write the assembler to convert assembly code into binary machine code	CPU: Peiyuan, Yingjie Assembly: Peiyuan Assembler: Yingjie
Week 10 - Week 11	1.Assembling the game components and testing (32x64 LED matrix, FPGA) 2.Design and fabrication of game exterior(acrylic and buttons)	1. Peiyuan 2. Hng Yi
Week 12 - Week 13	Check-off Demo Poster, video and report submissions.	Check-off and Report: Everyone Poster: Ryan Video: Qi Bao