A RISC-V computer system design*

Jun-Cheng Xiong

December 26, 2023

Abstract

In this project, a simple RISC-V computer system is implemented on Digilent Nexys A7-100T FPGA board. The compter system is composed of: a CPU, a data memory, an instruction memory, a vga screen, a keyboard, and a PWM audio output. The CPU is a 5-stage pipeline processor, which can execute 32-bit RISC-V instructions. The system interface is a terminal, which can execute basic commands and run two software.

Keywords: RISC-V, FPGA, 5-stage pipeline, computer system

*The latest version can be found at `https://github.com/dream-tentacle/digital-logic-paper`

Contents

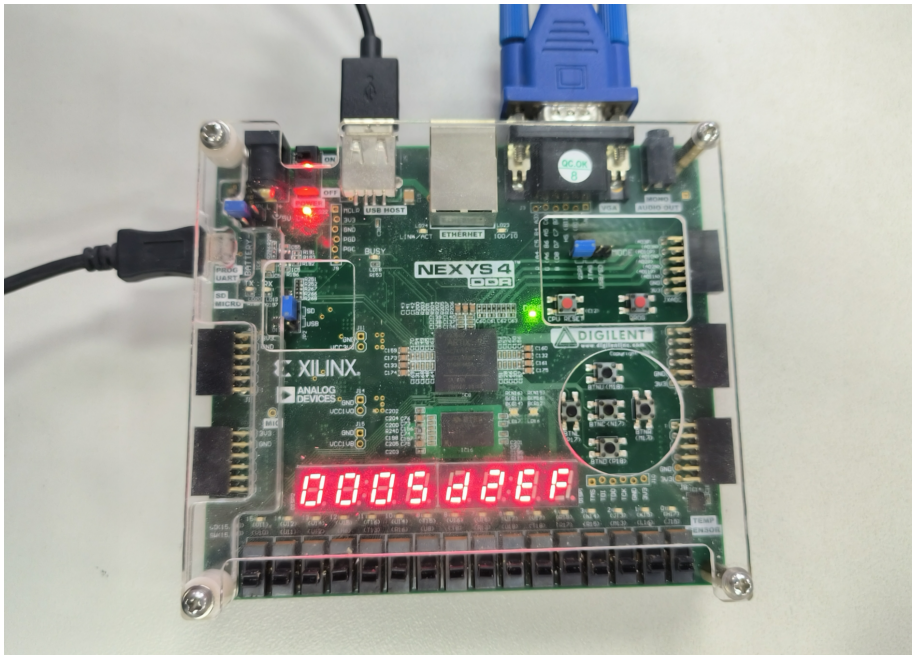

Figure 1: Digilent Nexys A7-100T FPGA board

1 Introduction

1.1 Overall introduction

In the realm of digital systems design, the creation of a fully functional computer system on a Field-Programmable Gate Array (FPGA) stands as a testament to the designer's knowledge of digital systems. This project centers around the development of a simple yet robust RISC-V computer system, implemented on the Digilent Nexys A7-100T FPGA board. At the heart of this project lies the CPU, which is a 5-stage pipeline processor that can execute 32-bit RISC-V instructions. The computer system designed for this project includes essential components such as a CPU, a data memory, an instruction memory, a VGA screen, a keyboard, and a PWM audio output. The user interacts with the system through a terminal, which supports several commands and can run three software - Snake, Piano, and Conway's Game of Life.
This report is divided into two parts. The first segment delves into the hardware design, concentrating on the intricacies of the 5-stage pipeline CPU, memory management, and all other devices. The second part shifts focus to the software design, encompassing the implementation of the terminal, the engaging Snake game, the simple yet functional Piano game, and the Conway's Game of Life.

1.2 FPGA

The FPGA board for this project is Digilent Nexys A7-100T, a circuit design and implementation platform for classroom use. For more information, please refer to `https://digilent.com/reference/programmable-logic/nexys-a7/start`.

1.3 RISC-V

RISC-V is an open-source instruction set architecture (ISA) based on reduced instruction set computer (RISC) principles. It is a standard ISA designed to be simple, extensible, and easy to implement. In this project, I implemented a 32-bit RISC-V CPU, which can execute all 37 base instructions. For more information about RISC-V itself, please refer to `https://riscv.org/`.

2 Hardware

2.1 5-stage pipeline CPU

The 5-stage pipeline divides every instruction into 5 stages, and executes them in parallel, which can greatly improve the speed of the CPU. The 5 stages are: instruction fetch (IF), instruction decode (ID), execute (EX), memory access (MEM), and write back (WB). The pipeline is shown in Figure 2. The cycle of clocks begins at the falling edge and ends at the next falling edge. The CPU clock is used in both the rising edge and the falling edge as follows. The regester file is written in the rising edge but read with logic assignment (thus output changes at any time). The instruction memory is read in the rising edge. The data memory (including other I/O devices) is read in the rising edge and written in the falling edge. The PC is updated in the falling edge. The pipeline registers are written and read in the falling edge.

Instruction fetch (IF)

This stage fetches the instruction from the instruction memory. The PC is the program counter, which stores the address of the next instruction. Ususally, the PC is updated by adding 4 to itself, and this is calculated by a specialized add unit. At the end of the IF stage, the current PC and the instruction are stored in the pipeline register IF/ID.

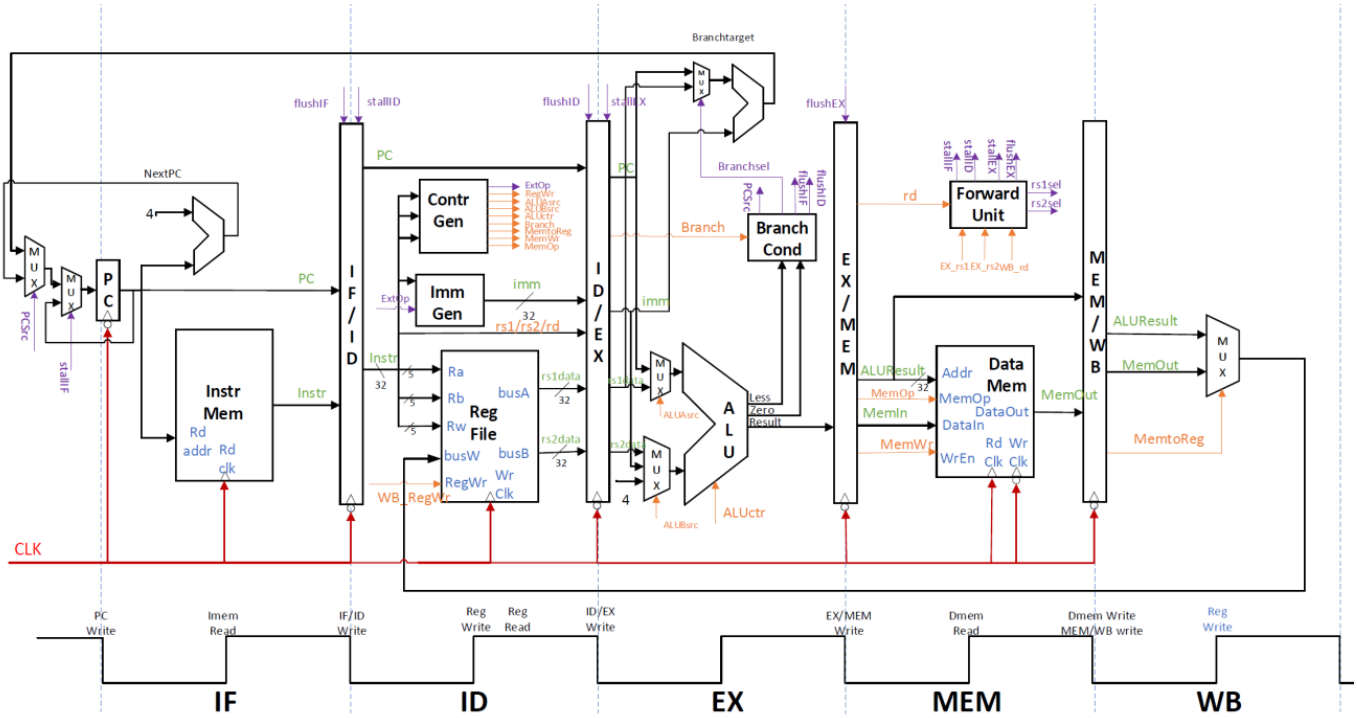

Figure 2

Instruction decode (ID)

This stage decodes the instruction and reads the register file. The register file has two read ports, which can read two registers at the same time. Other control signals are also generated in this stage. At the end of the ID stage, the PC, the regester information and the control signals are stored in the pipeline register ID/EX.

Execute (EX)

This stage executes the calculation. According to the result of ALU and the control signals, a signal for branching is generated. If it is going to branch, the IF and ID stages will be flushed, and the PC will be changed. At the end of the EX stage, the PC, the result of ALU, the regester information and the control signals are stored in the pipeline register EX/MEM.

Memory access (MEM)

This stage not only reads and writes the memory, but also checks if there is a hazard, which will be introduced later. At the end of the MEM stage, the PC, the result of ALU, the regester information and the control signals are stored in the pipeline register MEM/WB.

Write back (WB)

This stage writes the result of ALU or the memory to the register file. Because the register file is written in the rising edge, the written value can be used in the ID stage of the same clock cycle, which is called forwarding.

Hazard solution

There are three types of hazards: data hazard, control hazard and structural hazard. The structural hazard doesn't exist in this project because we don't use any hardware resource in two stages at the

same time. The control hazard is solved by flushing the IF and ID stages, which we have already mentioned. The data hazard is solved by forwarding.
There are three types of data hazard: RAW (read after write), WAR (write after read) and WAW (write after write). Only the RAW appears in this design. Again, there are three types of RAW: WB-ID, WB-EX, and MEM-EX. The WB-ID is mentioned in the WB stage introduction. To solve the WB-EX hazard, we use a forwarding unit to forward the result of ALU or the memory to the EX stage. The forwarding unit checks whether one of the register data in the EX comes from the writing register. If so, the forwarding unit will change the data to the writing data. The MEM-EX hazard is also solved by the forwarding unit. The forwarding unit checks whether one of the register data in the EX comes from the writing register and the instruction is a load instruction that reads the memory and writes the register. If so, the forwarding unit will change the data to the writing data.

2.2 Memory management

The CPU uses byte addressing, and access all other devices through memory-mapped I/O. The memory address is 32 bits wide, and the first 12 bits are used to select the device. The address map is shown in Table 1.

Address range	Device
0x00000000 - 0x000FFFFF	Instruction memory
0x00100000 - 0x001FFFFF	Data memory
0x00200000 - 0x002FFFFF	VGA screen
0x00300000, 0x00300004	Keyboard
0x00400000, 0x00400004	LED
0x00500000, 0x00500004	timer
0x00600000	deprecated
0x00700000	deprecated
0x00800000, 0x00800004	PWM audio output

Table 1: Address Map

2.3 Instruction and data memory

The instruction and data memory are both implemented using block RAM (BRAM). The instruction memory is read-only, and the data memory is read-write. Both of them are 1MB in size. The instruction address must be aligned to 4 bytes, while the data address can be any byte address.

2.4 VGA screen (Write-only)

The resolution is 640x480, and the color depth is 12 bits. Because the system is totally based on a terminal and the games are using characters as the basic unit, the screen is divided into 80x30 characters of 8x16 pixels. This allows us only store 80x30 characters' ascii code.
To accelerate the screen access speed, the screen is implemented with an one-dimensional array instead of a two-dimensional array. Considering the line and column size, 11-7 bits of the address are used to represent the line number, and 6-0 bits are used to represent the column number($2^5 = 32, 2^7 = 128$). The screen has a base address of 0x00200000, and is accessed with 0x00200000 + offset.

2.5 Keyboard (Read-only)

The input signals of the keyboard are PS2_CLK and PS2_DATA. With a keyboard signal processor, every byte of scan code can be generated from a stream of PS2_DATA. The scan code is then stored

in a circular buffer of 16 bytes. The buffer provides the CPU with a “new_key” signal, which is high when the buffer is not empty. The CPU can read “new_key” through the address 0x00300000. When the CPU reads the address 0x00300004, the buffer will pop a byte of scan code.
The driver of the keyboard is implemented in the software part. Basically, the driver is a finite state machine (Figure 3), which can be divided into 4 states: KEY_DOWN, KEY_UP, LONG_KEY_DOWN, and LONG_KEY_UP. The “LONG_” prefix in states refers to the two-bytes scan code which mainly start with 0xE0. The break code of a key is 0xF0 succeeded by its make code, thus the KEY_UP and LONG_KEY_UP states are used to detect the break code.
The driver is implemented with a C function, which has only one state change every time it is called. There are three global values, “key_ready”, “two_byte_code” and “key_up”, which are signals for applications and are set according to the state. When the state machine finds that it ends a key press, it sets “key_ready” to 1, and returns (the lowest byte of) the scan code. Otherwise, it returns 0. The other two-byte scan codes are not used in this project, so we just ignore them.

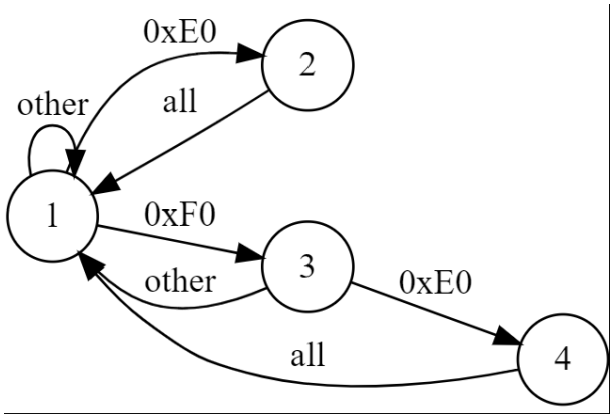

Figure 3: Keyboard driver state machine
1: KEY_DOWN, 2: LONG_KEY_DOWN,
3: KEY_UP, 4: LONG_KEY_UP

When a keycode is gotten, the software part will get the ascii code for it through an array. If the key is a signal key, it will be proceeded correspondingly.

2.6 PWM audio output (Write-only)

The audio pitch is based on the frequency of the PWM signal. There is a count value and a max value. Thw PWM signal is high when the count value is less than half of the max value, and low otherwise. The count value is increased by 1 every clock cycle and set back to 0 when it exceeds the max value. Thus, the pitch can be adjusted by changing the max value.
The audio has two memory addresses, 0x00800000 and 0x00800004. The first address is used to set the max value, and the second address is used to turn on/off the audio.

2.7 Timer (Read-only)

There are two memory addresses, 0x00500000 and 0x00500004, corresponding to the millisecond counter and the second counter. Four counters are used in the timer component: clock counter, millisecond counter 1, millisecond counter 2, and second counter. The clock frequency is 100MHz. When the clock counter reaches 10^5, both the millisecond counters are increased by 1, and the clock counter is set back to 0. When the millisecond counter 1 reaches 1000, the second counter is increased by 1, and the millisecond counter 1 is set back to 0. The address 0x00500000 is used to read the millisecond counter 2, and the address 0x00500004 is used to read the second counter.

3 Software

3.1 Terminal

The terminal is the interface between the user and the system. The function of the terminal calls the keyboard driver function only once every time it is called. If the keyboard driver returns 0, the terminal will return immediately. Otherwise, the terminal will process the scan code and execute the corresponding command. When the system is turned on, it enters a while loop, which calls the terminal function without stopping.
The terminal has different types of responses to different key presses:

- **key up**: When the global variable "key_up" is set to 1, the terminal checks if the key is "shift", "ctrl", or "alt". If so, it sets the corresponding global flag variable to 0. Otherwise, it does nothing.
- **key down**: When the global variable "key_up" is set to 0:
  - **signal keys**: If the key is "shift", "ctrl", or "alt", the terminal sets the corresponding global flag variable to 1.
  - **backspace**: If the key is "backspace", the terminal deletes the last character in the buffer and put a space in the screen.
  - **enter**: If the key is "enter", the terminal executes the command in the buffer. If the command is not recognized, the terminal does nothing.
  - **caps lock**: If the key is "caps lock", the terminal toggles the "caps_lock" global flag variable.

3.2 Help

Typing "help" and enter gives you the usage of different commands (Figure 4). The simple commands

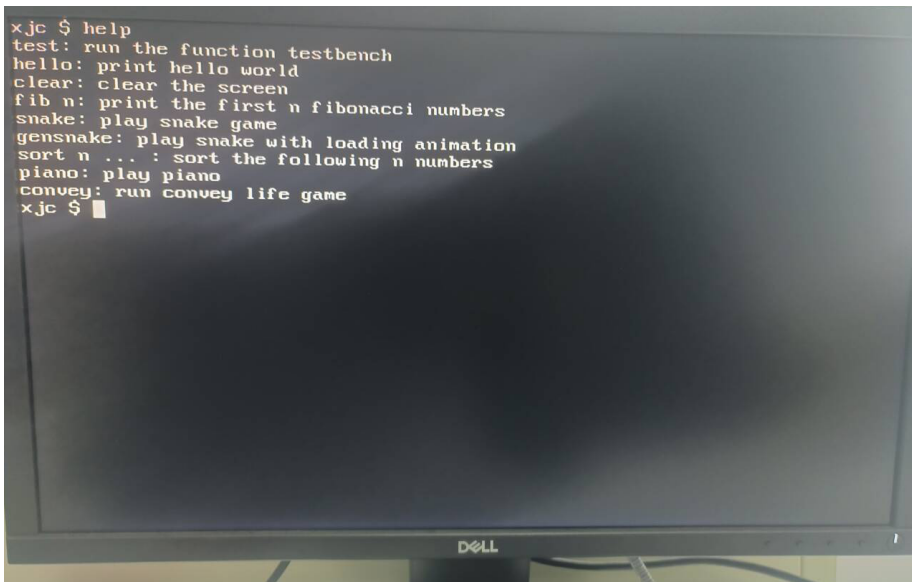

Figure 4: Help

as "fib", "hello" and "sort" is shown in Figure 5.

3.3 Structural Hazard Test

The test command runs a program testing whether the control hazard is solved. There are 5 types of testing (code in Appendix). The result is shown in Figure 6.

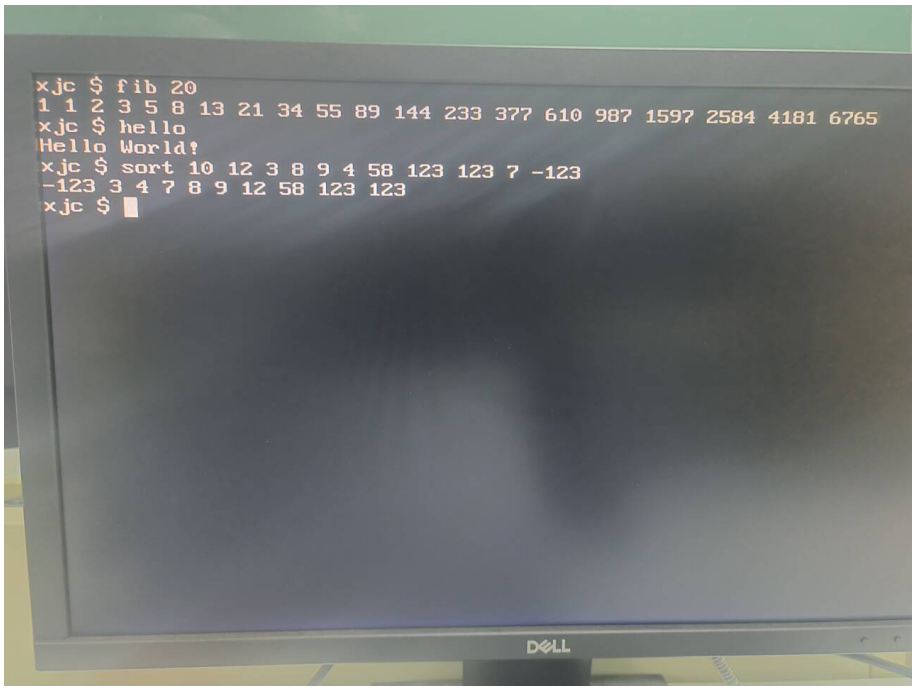

Figure 5: Simple commands

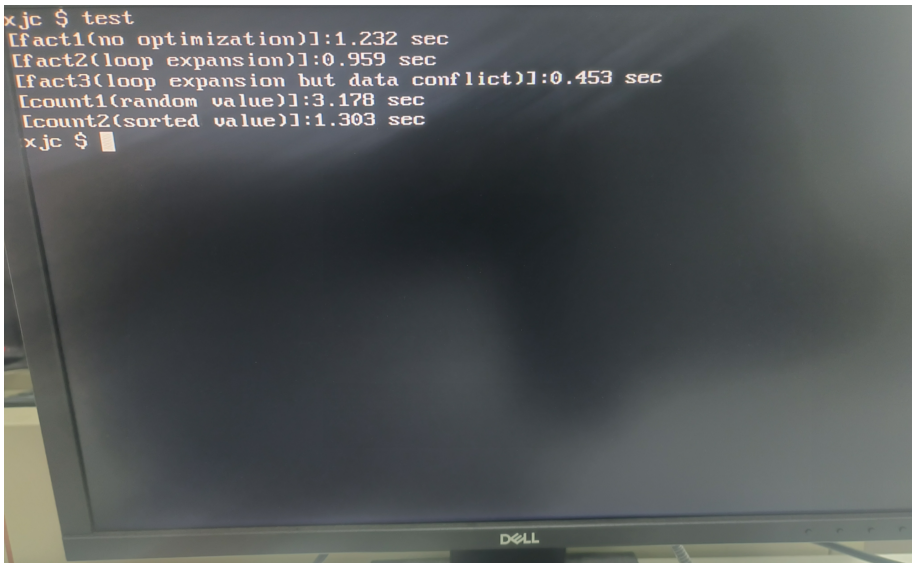

Figure 6: Test result

3.4 Snake

The snake game looks like the classic snake game on Nokia phones. The snake can move in four directions, and it will die if it hits the wall or itself. The snake will grow longer when it eats food. There are 3 different levels in the game. The speed of the snake increases as the level increases. When the “snake” command is executed in the terminal, a pop-up window will appear (Figure 7), and the player can choose the level by pressing 1, 2, or 3. The game will start after the player chooses the level. The player can press “q” to quit the game and return to the terminal.
The game looks like Figure 8. The snake is composed of squares, and the food is a circle. The score is shown in the top left corner. The player can press “w”, “a”, “s”, “d” to control the snake. The player can press “q” to quit the game and return to the terminal. When the game ends, a pop-up window will appear, reads “YOU LOSE” in ascii art and the score (Figure 9).

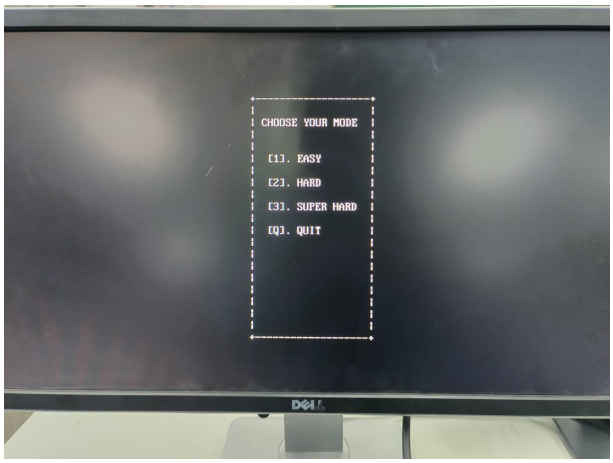

Figure 7: Snake game level selection

Figure 8: Snake game

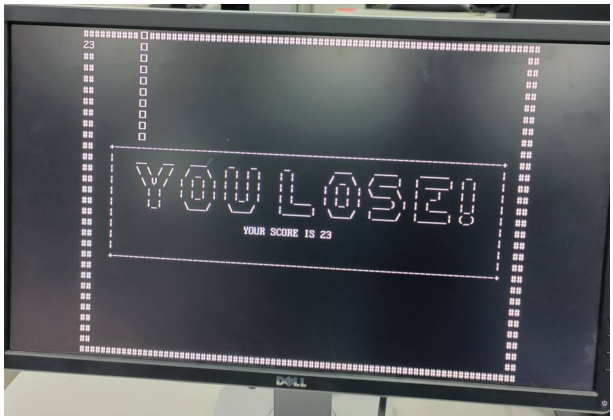

Figure 9: Snake game over

When using the command “gensnake”, a loading animation will appear (Figure 10).

3.5 Piano

The piano game is just a simple piano. The player can press the keys: `, 0-9, -, = and backspace to play. The shift (ctrl) key can increase (decrease) the pitch by one octave. The player can press “q” to quit the game and return to the terminal.
The game looks like Figure 11. The numbers under the piano keys tell the player how to play the piano. The player can press “q” to quit the game and return to the terminal.

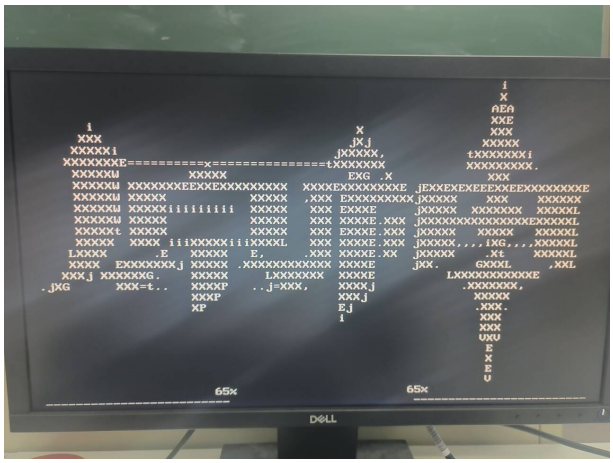

Figure 10: Loading animation

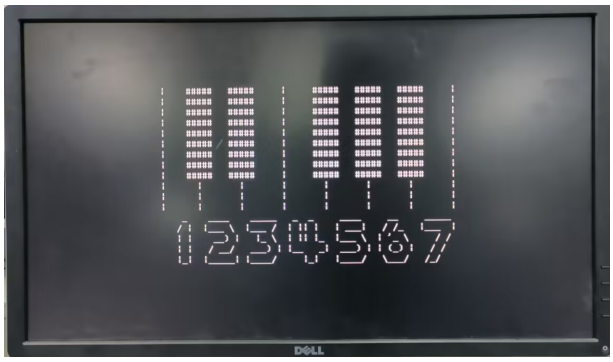

Figure 11: Piano game

3.6 Conway

The convey is the well-known experiment of Conway's Game of Life. The game implements the rules of Conway's Game of Life. The player can press "q" to quit the game and return to the terminal. Figure 12 shows the game when it starts, and Figure 13 shows the stable state of the game.

Figure 12: Conway's Game of Life

Figure 13: Conway's Game of Life

4 appendix

4.1 test

```
static __attribute__((noinline)) void fact1() {
    volatile unsigned fact __attribute__((unused)) = 1;
    for (unsigned i = 1; i < FACT_N; i++)
    fact *= i;
}
__attribute__((noinline)) void fact2() {
    volatile unsigned fact0 __attribute__((unused)) = 1;
    volatile unsigned fact1 __attribute__((unused)) = 1;
    volatile unsigned fact2 __attribute__((unused)) = 1;
    volatile unsigned fact3 __attribute__((unused)) = 1;
    for (unsigned i = 1; i < FACT_N; i += 4) {
    fact0 *= i;
    fact1 *= i + 1;
    fact2 *= i + 2;
    fact3 *= i + 3;
    }
}
__attribute__((noinline)) void fact3() {
    volatile unsigned fact __attribute__((unused)) = 1;
    for (unsigned i = 1; i < FACT_N; i += 4) {
    fact *= i;
    fact *= i + 1;
    fact *= i + 2;
    fact *= i + 3;
    }
}
static __attribute__((noinline)) void count1() {
    int T = TIMES;
    while (T--) {
        for (int i = 0; i < SIZE; i++) {
        value[i] = rand();
        }
        volatile int cnt __attribute__((unused)) = 0;
        for (int i = 0; i < SIZE; i++)
        if (value[i] > 500)
```

```
36              cnt++;
37      }
38  }
39  static __attribute__((noinline)) void count2() {
40      int T = TIMES;
41      while (T--) {
42          for (int i = 0; i < SIZE; i++) {
43          value[i] = i;
44          }
45          volatile int cnt __attribute__((unused)) = 0;
46          for (int i = 0; i < SIZE; i++)
47          if (value[i] > 500)
48              cnt++;
49      }
50  }
```