Operating Systems

Project Checkpoint 5

111062109 黃庭曜

Table of Contents

§1 Screenshots for compilation

§1.1 Command "`make clean`"

Part 1:

```
huangtingyao@crazytingyao ppc5_1 % make clean
rm *.hex *.ihx *.lnk *.lst *.map *.mem *.rel *.rst *.sym
rm: *.ihx: No such file or directory
rm: *.lnk: No such file or directory
make: *** [clean] Error 1
```

Fig. 1: "`make clean`" command execution result for part 1

Part 2:

```
huangtingyao@crazytingyao ppc5 % make clean
rm *.hex *.ihx *.lnk *.lst *.map *.mem *.rel *.rst *.sym *.asm
rm: *.ihx: No such file or directory
rm: *.lnk: No such file or directory
make: *** [clean] Error 1
```

Fig. 2: "`make clean`" command execution result for part 2

§1.2 Command "`make`"

Part 1:

```
huangtingyao@crazytingyao ppc5_1 % make
sdcc -c  --model-small testlcd.c
sdcc -c  --model-small preemptive.c
preemptive.c:214: warning 85: in function ThreadCreate unreferenced function argument : 'fp'
preemptive.c:272: warning 158: overflow in implicit constant conversion
sdcc -c  --model-small lcdlib.c
lcdlib.c:76: warning 85: in function delay unreferenced function argument : 'n'
sdcc -c  --model-small buttonlib.c
sdcc -c  --model-small keylib.c
sdcc  -o testlcd.hex testlcd.rel preemptive.rel lcdlib.rel buttonlib.rel keylib.rel
```

Fig. 3: "`make`" command execution result for part 1

Part 2:

```
huangtingyao@crazytingyao ppc5 % make
sdcc -c  --model-small dino.c
sdcc -c  --model-small preemptive.c
preemptive.c:240: warning 85: in function ThreadCreate unreferenced function argument : 'fp'
preemptive.c:333: warning 158: overflow in implicit constant conversion
sdcc -c  --model-small lcdlib.c
lcdlib.c:80: warning 85: in function delay unreferenced function argument : 'n'
sdcc -c  --model-small keylib.c
sdcc  -o dino.hex dino.rel preemptive.rel lcdlib.rel keylib.rel
```

Fig. 4: "`make`" command execution result for part 2

§2 Peripheral devices

§2.1 Address list

The address list of all functions is shown in **Fig. 5**.

	Value	Global	Global Defined In Module
C:	00000014	_Producer1	testlcd
C:	00000068	_Producer2	testlcd
C:	000000BC	_Consumer	testlcd
C:	000000FE	_main	testlcd
C:	0000011F	__sdcc_gsinit_startup	testlcd
C:	00000123	__mcs51_genRAMCLEAR	testlcd
C:	00000124	__mcs51_genXINIT	testlcd
C:	00000125	__mcs51_genXRAMCLEAR	testlcd
C:	00000126	_timer0_ISR	testlcd
C:	0000012A	_Bootstrap	preemptive
C:	00000153	_myTimer0Handler	preemptive
C:	000001DC	_ThreadCreate	preemptive
C:	00000250	_ThreadYield	preemptive
C:	000002A4	_ThreadExit	preemptive
C:	00000301	_LCD_ready	lcdlib
C:	00000305	_LCD_Init	lcdlib
C:	00000318	_LCD_IRWrite	lcdlib
C:	0000033A	_LCD_functionSet	lcdlib
C:	00000364	_LCD_write_char	lcdlib
C:	0000038E	_LCD_write_string	lcdlib
C:	000003C3	_delay	lcdlib
C:	000003C7	_AnyButtonPressed	buttonlib
C:	000003D9	_ButtonToChar	buttonlib
C:	00000465	_Init_Keypad	keylib
C:	0000046B	_AnyKeyPressed	keylib
C:	00000478	_KeyToChar	keylib
C:	000004F4	__gptrget	_gptrget

Fig. 5: Function address list for part 1

§2.2 Producer1 (Button)

The `Producer1`'s code address is between 0x14 and 0x66. The screenshot for the first `Producer1` call is shown in **Fig. 6**.

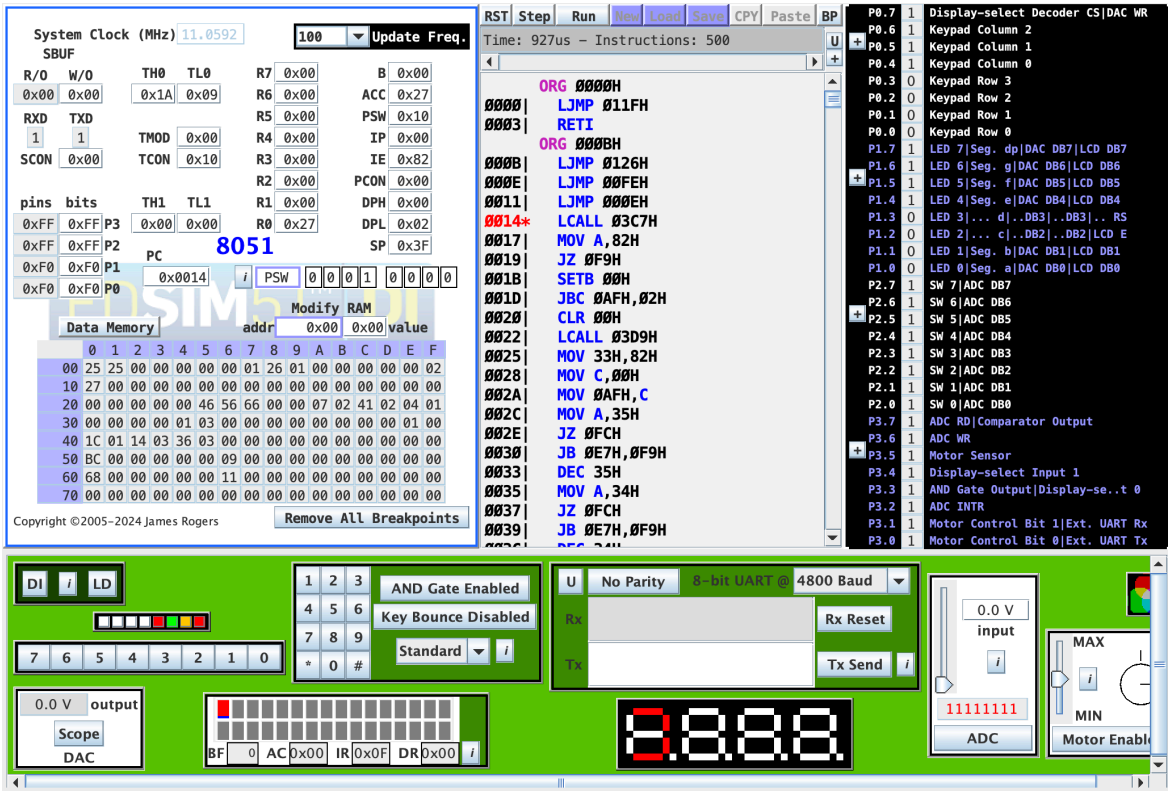

Fig. 6: Screenshot for the first `Producer1` call

The `Producer1` waits for any button to be pressed by using the `AnyButtonPressed` function$_{(1)}$. If a button is pressed, the button number will be recorded by the current button parameter via `ButtonToChar` function$_{(2)}$. The buffer part remains the same as checkpoint 3 & 4. Before recording the next button, we have to wait for the user to release all pressed buttons.$_{(3)}$ The code is shown in **Fig. 7**. An example after pressing button 4 is shown in **Fig. 8**.

```
#define LABEL(x) x ## $
#define L(x) LABEL(x)
void Producer1(void)
{
    while (1){
        while(!AnyButtonPressed());          (1)
        __critical{
            currentButton = ButtonToChar();  (2)
        }
        SemaphoreWaitBody(empty,  L(__COUNTER__));
        SemaphoreWaitBody(mutex,  L(__COUNTER__));
        buffer[producer_ptr] = currentButton;
        producer_ptr = (producer_ptr == 2) ? 0 : producer_ptr + 1;
        SemaphoreSignal(mutex);
        SemaphoreSignal(full);
        while (AnyButtonPressed());          (3)
    }
}
```

Fig. 7: `Producer1` code

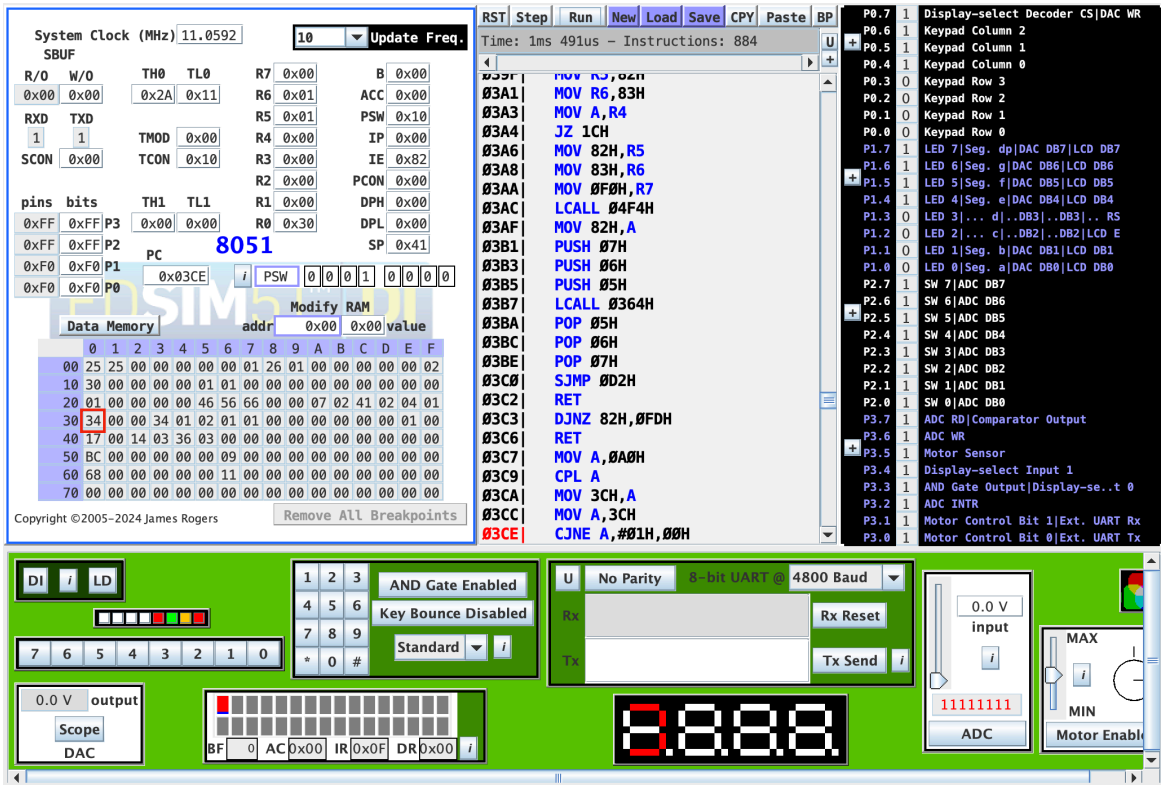

Fig. 8: Screenshot for pressing button 4 in the `Producer1` call

The buffer at 0x30 writes the ASCII code of the number 4, which is 0x34.

§2.3 Producer2 (Keypad)

The `Producer2`'s code address is between 0x68 and 0xBA. The screenshot for the first `Producer2` call is shown in **Fig. 9**.

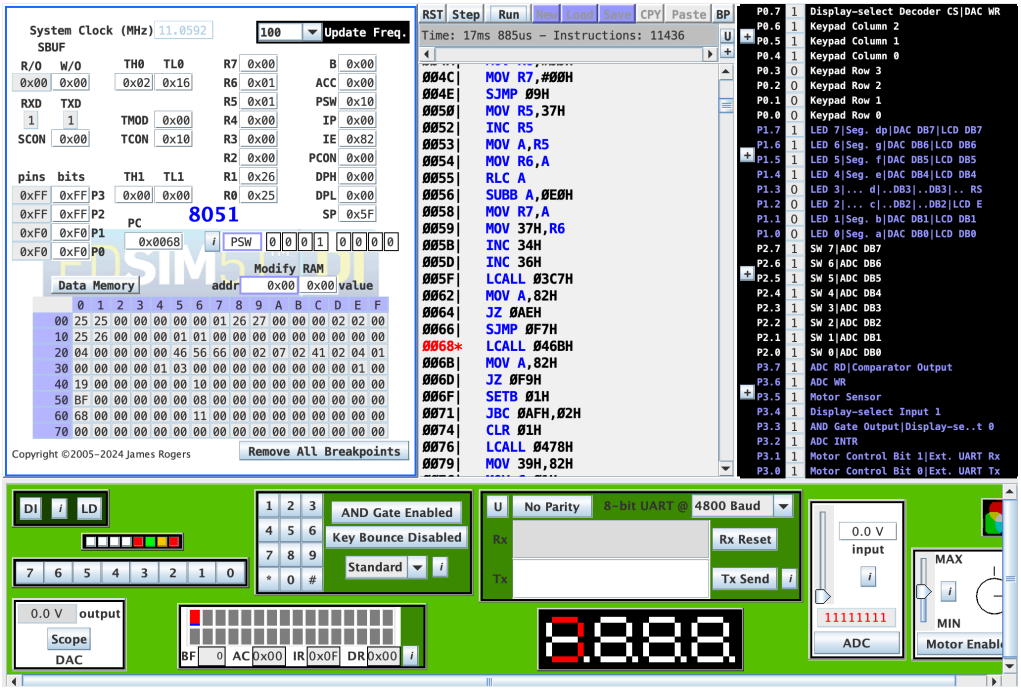

Fig. 9: Screenshot for the first `Producer2` call

In `Producer2`, if we press a button on the keypad, the corresponding number will be written to the buffer. For example, after pressing keypad number 9, the result is shown in **Fig. 10**.

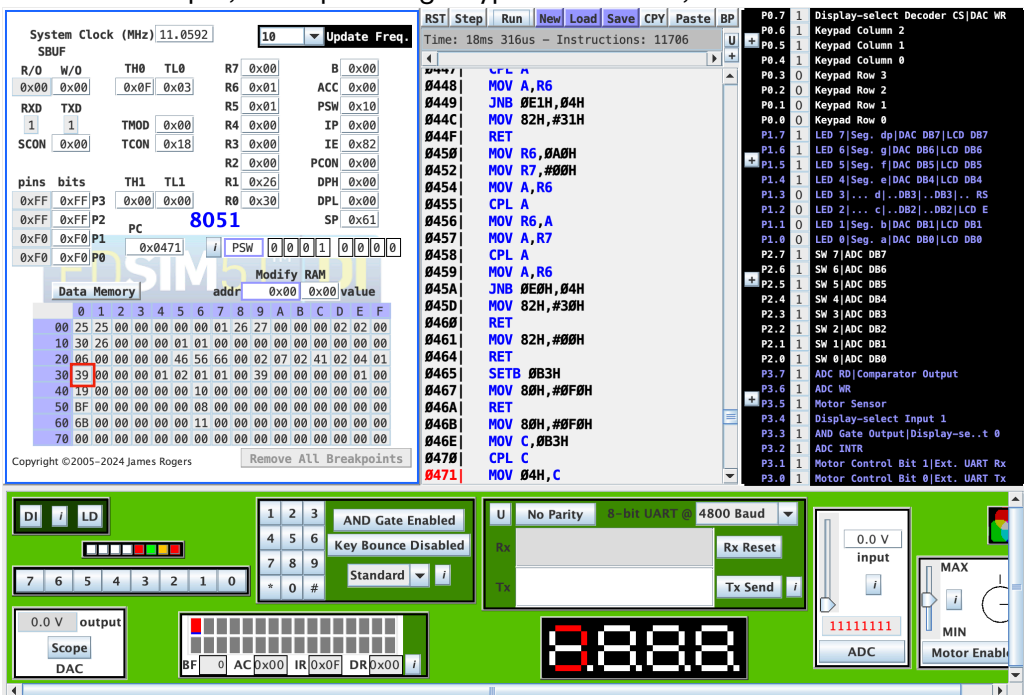

Fig. 10: Screenshot for pressing button 9 in the `Producer2` call

The buffer at 0x30 writes the ASCII code of the number 9, which is 0x39.
The code is similar to `Producer2` and is shown in **Fig. 11**.

```
void Producer2(void)
{
    while (1){
        while (!AnyKeyPressed());
        __critical{
            currentKey = KeyToChar();
        }
        SemaphoreWaitBody(empty,  L(__COUNTER__));
        SemaphoreWaitBody(mutex,  L(__COUNTER__));
        buffer[producer_ptr] = currentKey;
        producer_ptr = (producer_ptr == 2) ? 0 : producer_ptr + 1;
        SemaphoreSignal(mutex);
        SemaphoreSignal(full);
        while (AnyKeyPressed());
    }
}
```

Fig. 11: `Producer2` code

§2.4 Consumer (LCD)

The `Consumer`'s code address is between 0xBC and 0xFC. The screenshot for the first `Consumer` call is shown in **Fig. 12**.

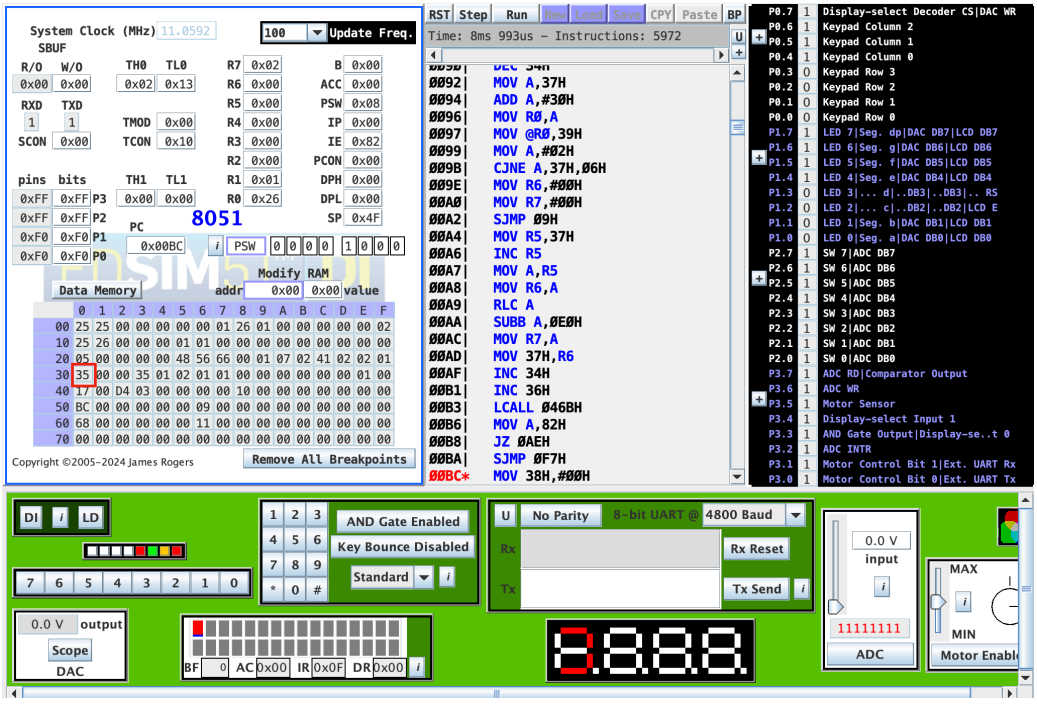

Fig. 12: Screenshot for the first `Consumer` call

Currently, a button 5 is pressed before, and the buffer now contains 0x35. The `Consumer` will write the number to the LCD. The result is shown In **Fig. 13**.

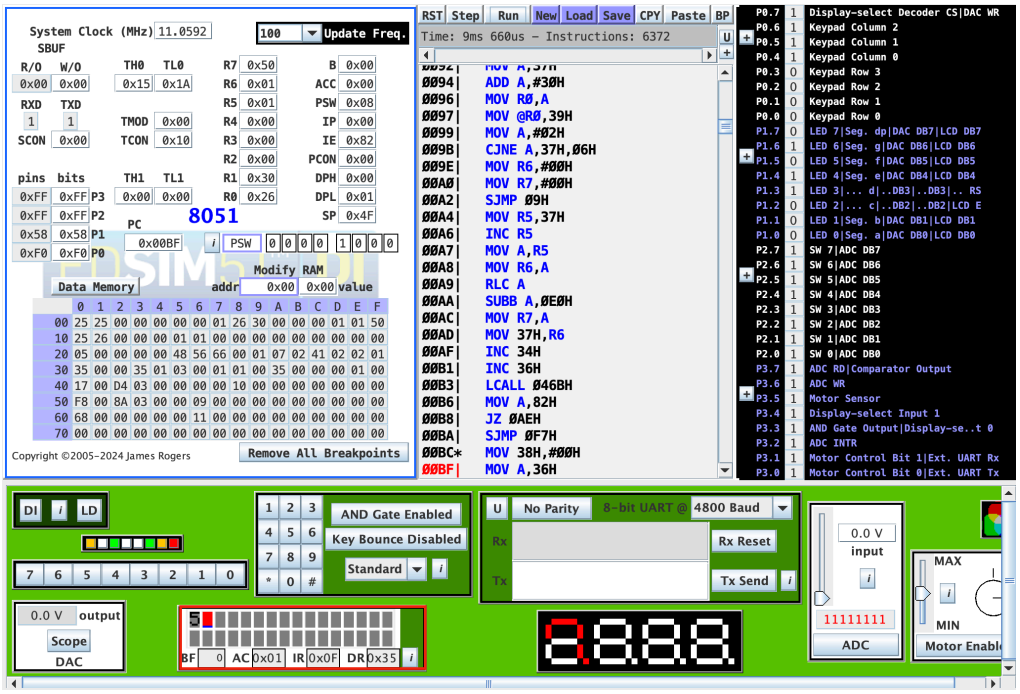

Fig. 13: Screenshot for writing 5 in the `Consumer` call

The code is shown in **Fig. 14**. The code is similar to the original `Consumer` in checkpoints 3 & 4. Only writing to SBUF is replaced by the `LCD_write_char` function, and waiting TI changes to waiting for `LCD_ready`.

```
void Consumer(void)
{
    consumer_ptr = 0;
    while (1){
        SemaphoreWaitBody(full,  L(__COUNTER__));
        SemaphoreWaitBody(mutex,  L(__COUNTER__));
        pop = buffer[consumer_ptr];
        SemaphoreSignal(mutex);
        SemaphoreSignal(empty);
        consumer_ptr = (consumer_ptr == 2) ? 0 : consumer_ptr + 1;
        LCD_write_char(pop);
        while (!LCD_ready());
    }
}
```

Fig. 14: Code for `Consumer`

§3 Dinosaur game

§3.1 Address List

The address list of all functions is shown in **Fig. 15**.

```
      Value  Global                          Global Defined In Module
      -----  -------------------------------  ------------------------
C:    00000014  _render_task                   dino
C:    00000162  _keypad_ctrl                   dino
C:    000001E3  _game_ctrl                     dino
C:    000002DA  _main                          dino
C:    0000035C  __sdcc_gsinit_startup          dino
C:    00000360  __mcs51_genRAMCLEAR            dino
C:    00000361  __mcs51_genXINIT               dino
C:    00000362  __mcs51_genXRAMCLEAR           dino
C:    00000363  _timer0_ISR                    dino
C:    00000367  _Bootstrap                     preemptive
C:    00000385  _myTimer0Handler               preemptive
C:    000003DF  _ThreadCreate                  preemptive
C:    0000045A  _ThreadYield                   preemptive
C:    000004B3  _ThreadExit                    preemptive
C:    00000522  _ThreadReset                   preemptive
C:    00000526  _LCD_ready                     lcdlib
C:    0000052A  _LCD_Init                      lcdlib
C:    0000058B  _LCD_IRWrite                   lcdlib
C:    000005AD  _LCD_functionSet               lcdlib
C:    000005D7  _LCD_write_char                lcdlib
C:    00000601  _LCD_write_string              lcdlib
C:    00000636  _delay                         lcdlib
C:    0000063A  _LCD_set_symbol                lcdlib
C:    0000074B  _Init_Keypad                   keylib
C:    00000751  _AnyKeyPressed                 keylib
C:    0000075E  _KeyToChar                     keylib
C:    000007DA  __moduint                      _moduint
C:    00000827  __divuint                      _divuint
C:    00000850  __gptrget                      _gptrget
C:    0000086C  __modsint                      _modsint
C:    000008A2  __divsint                      _divsint
```

Fig. 15: Function address list for part 2

§3.2 Main Function

The `main`'s code address is between 0x2DA and 0x35A. In `main` function, I performed keypad and LCD initialization first$_{(1)}$. Then, an infinite loop is entered. The map, score, dinosaur location, etc., are initialized in the loop$_{(2)}$. Then, the user is requested to enter a number followed by a pound sign indicating the stage number$_{(3)}$. After that, it creates the `render_task` thread and the `keypad_ctrl` thread$_{(4)}$. It then runs the `game_ctrl` thread$_{(5)}$. The infinite loop supports infinite plays; the details will be explained in §3.5. The code is shown in **Fig. 16**.

```
void main(void)
{
    Init_Keypad();
    LCD_Init();
    LCD_entryModeSet(1, 1);
    LCD_displayOnOffControl(1, 0, 0);
    do{
        map_up[0] = 0x21;
        map_up[1] = 0xC4;
        map_down[0] = 0x08;
        map_down[1] = 0x11;
        EA = 0;
        time = 0;
        lose = 0;
        points = 0;
        dinosaur_loc = 0;
        while(1){
            while(!AnyKeyPressed());
            currentKey = KeyToChar();
            while(AnyKeyPressed());
            if(currentKey == '#'){
                if(time == 0) continue;
                else break;
            }
            if(currentKey == '*'){
                continue;
            }
            time = (currentKey - '0');
        }
        ThreadCreate(render_task);
        ThreadCreate(keypad_ctrl);
        TMOD = 0; // timer 0 mode 0
        TH0 = (time << 4);
        TL0 = 0x00;
        IE = 0x82; // enable timer 0 interrupt,
        TR0 = 1; // start running timer0
        game_ctrl();
        EA = 0;
        ThreadReset();
    }while(1);
}
```

(1) (2) (3) (4) (5)

Fig. 16: Code of `main` function

§3.3 Render Task Thread

The `render_task`'s code address is between 0x14 and 0x15F. The screenshot for the first `render_task` call is shown in **Fig. 17**.

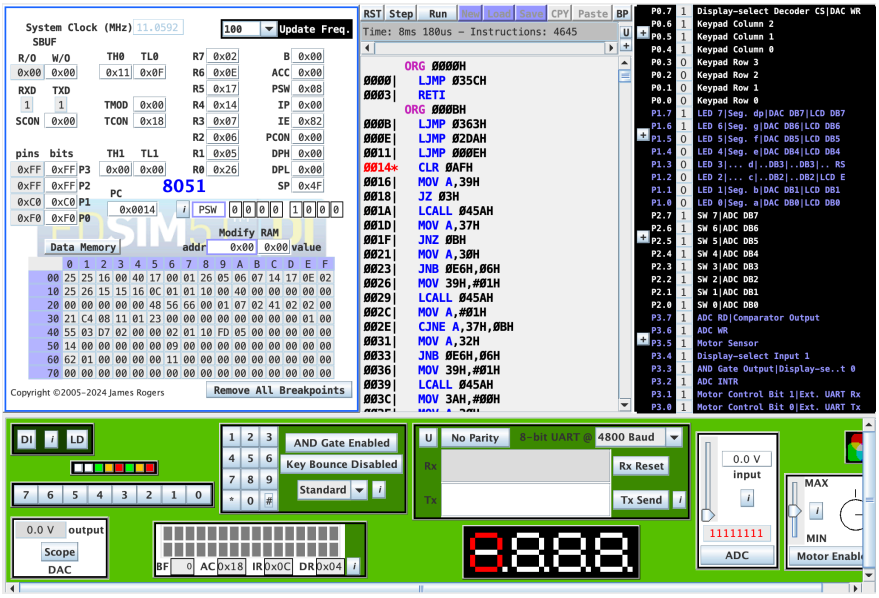

Fig. 17: Screenshot for the first `render_task` call

The `render_task` itself is an infinite loop. It will first check if the dinosaur will collide with the upcoming cacti. If it will, the `render_task` will set `lose` to 1 and call `ThreadYield()` immediately$_{(1)}$. If not, the map will perform a left shift$_{(2)}$, and the map will be drawn on the LCD$_{(3)}$. If a cactus is shifted to the end, meaning the dinosaur dodged it successfully, it increments the score. After drawing, it calls `ThreadYield()` immediately$_{(4)}$. The screenshot before it calls `ThreadYield()` is shown in **Fig. 18**, and the code is shown in **Fig. 19**.

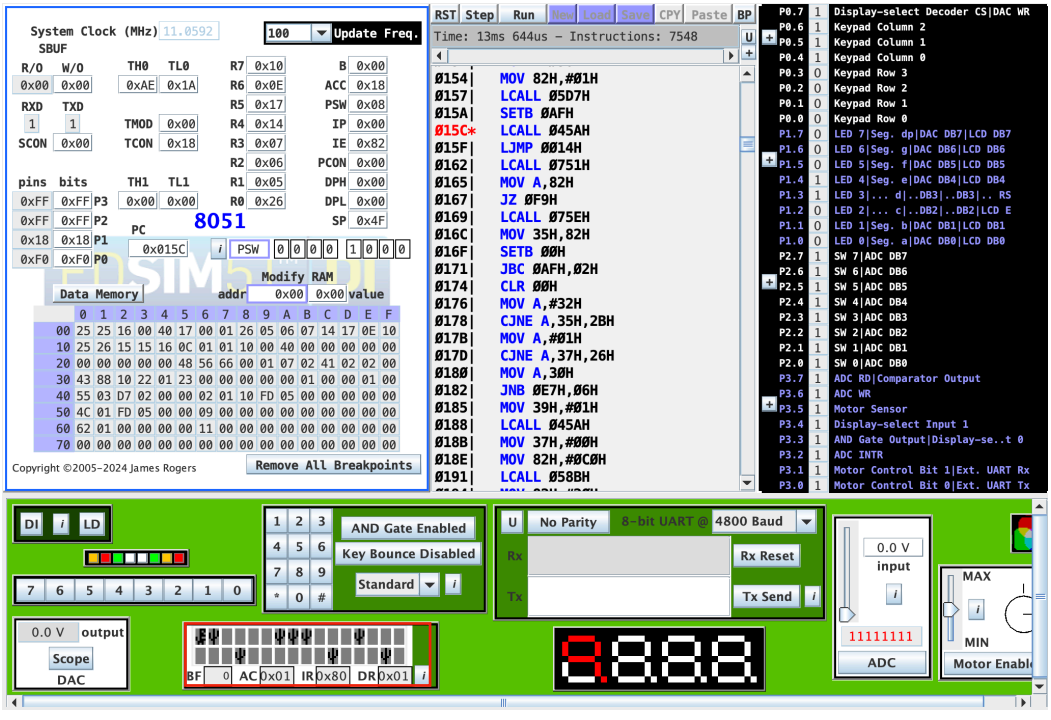

Fig. 18: Screenshot before first `render_task` calls `ThreadYield()`

```c
void render_task(void){
    while(1){
        EA = 0;
        if(lose) ThreadYield();
        if(dinosaur_loc == 0){
            if(map_up[0] & 0x40){ lose = 1; ThreadYield(); }
        }
        if(dinosaur_loc == 1){
            if(map_down[0] & 0x40){ lose = 1; ThreadYield();}
        }
        temp_shift = 0;
        if(map_up[0] & 0x80) temp_shift = 1;
        map_up[0] <<= 1;
        if(map_up[1] & 0x80) map_up[0]++;
        map_up[1] <<= 1;
        if(temp_shift){ points++; map_up[1]++; }
        temp_shift = 0;
        if(map_down[0] & 0x80) temp_shift = 1;
        map_down[0] <<= 1;
        if(map_down[1] & 0x80) map_down[0]++;
        map_down[1] <<= 1;
        if(temp_shift){ points++; map_down[1]++; }
        LCD_cursorGoTo(0, 0);
        temp_print = 0x80;
        while(temp_print){
            if(map_up[0] & temp_print) LCD_write_char('\2');
            else LCD_write_char(' ');
            temp_print >>= 1;
        }
        temp_print = 0x80;
        while(temp_print != 1){
            if(map_up[1] & temp_print) LCD_write_char('\2');
            else LCD_write_char(' ');
            temp_print >>= 1;
        }
        LCD_cursorGoTo(1, 0);
        temp_print = 0x80;
        while(temp_print){
            if(map_down[0] & temp_print) LCD_write_char('\2');
            else LCD_write_char(' ');
            temp_print >>= 1;
        }
        temp_print = 0x80;
        while(temp_print != 1){
            if(map_down[1] & temp_print) LCD_write_char('\2');
            else LCD_write_char(' ');
            temp_print >>= 1;
        }

        if(dinosaur_loc == 0){
            LCD_cursorGoTo(0, 0);
            LCD_write_char('\1');
        }
        else{
            LCD_cursorGoTo(1, 0);
            LCD_write_char('\1');
        }
        EA = 1;
        ThreadYield();
    }
}
```

(1)

(2)

(3)

(4)

Fig. 19: Code for `render_task`

An example of the dinosaur failing to dodge a cactus is shown in **Fig. 20**.

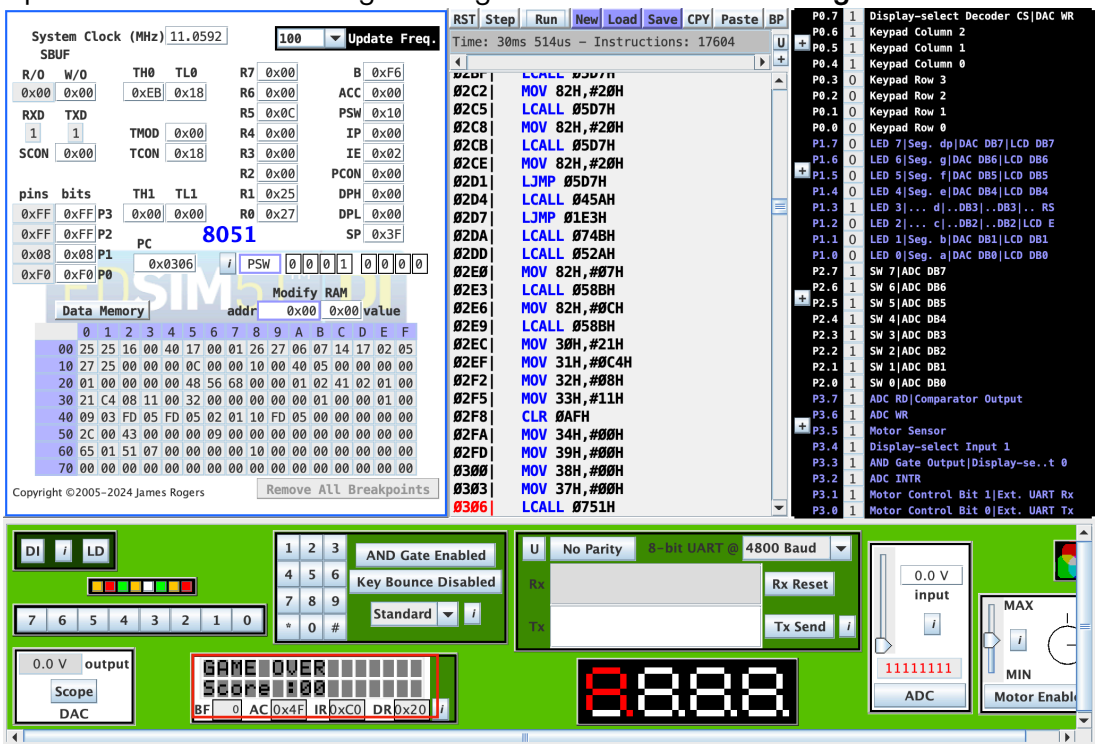

Fig. 20: Screenshot after the dinosaur fails to dodge a cactus

§3.4 Keypad Control Thread

The `keypad_ctrl`'s code address is between 0x162 and 0x1E0. The screenshot for the first `keypad_ctrl` call is shown in **Fig. 21**.

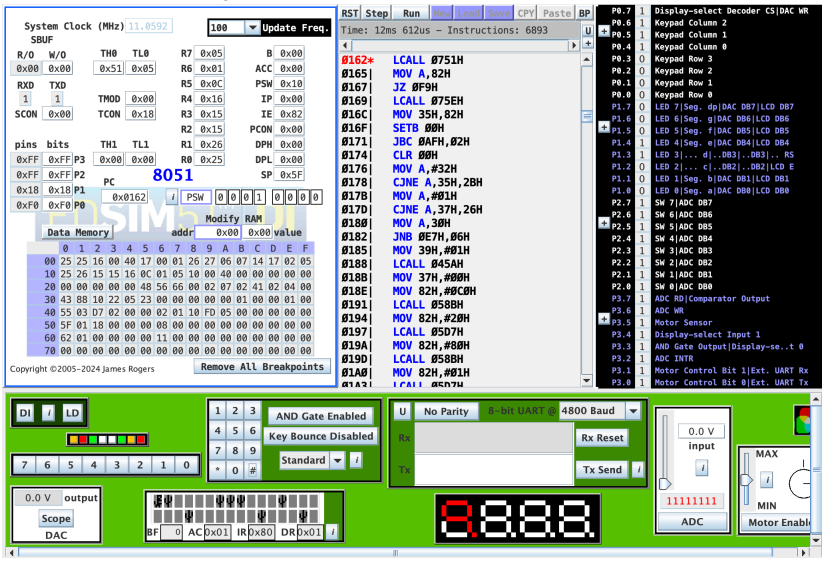

Fig. 21: Screenshot for the first `keypad_ctrl` call

The `keypad_ctrl` itself is also an infinite loop. It catches the keypad input "2" or "8" to determine where the dinosaur should be located.(1) It also checks if the dinosaur will hit a cactus. If it does, the thread will set `lose` to 0 and call `ThreadYield()`.(2) If not, it will proceed with the movement.(3) The code is shown in **Fig. 22**, and an example movement result is shown in **Fig. 23**.

```
void keypad_ctrl(void)
{
    while (1){
        while (!AnyKeyPressed());
        currentKey = KeyToChar();
        __critical{
(1)         if(currentKey == '2'){
                if(dinosaur_loc == 1){
(2)                 if(map_up[0] & 0x80){
                        lose = 1;
                        ThreadYield();
                    }
(3)                 dinosaur_loc = 0;
                    LCD_cursorGoTo(1, 0);
                    LCD_write_char(' ');
                    LCD_cursorGoTo(0, 0);
                    LCD_write_char('\1');

                }
            }
(1)         if(currentKey == '8'){
                if(dinosaur_loc == 0){
(2)                 if(map_down[0] & 0x80){
                        lose = 1;
                        ThreadYield();
                    }
(3)                 dinosaur_loc = 1;
                    LCD_cursorGoTo(0, 0);
                    LCD_write_char(' ');
                    LCD_cursorGoTo(1, 0);
                    LCD_write_char('\1');

                }
            }
        }
        while (AnyKeyPressed());
    }
}
```

Fig. 22: Code for `keypad_ctrl`

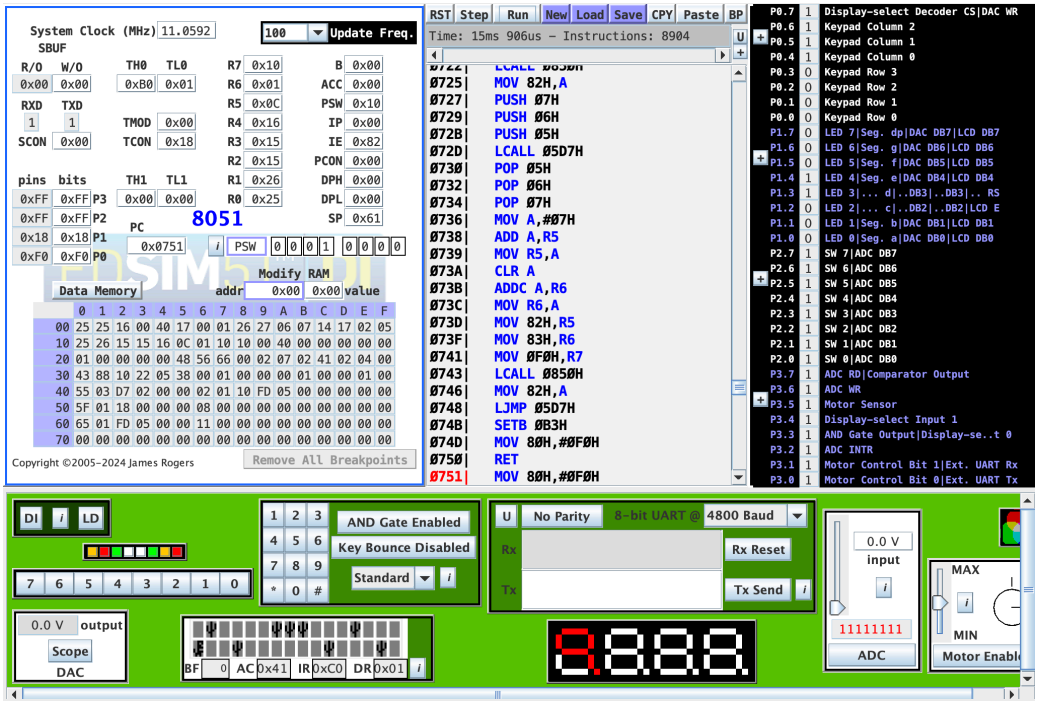

Fig. 23: Screenshot of the dinosaur moving downwards

An example of hitting a cactus is shown in **Fig. 24 & 25**.

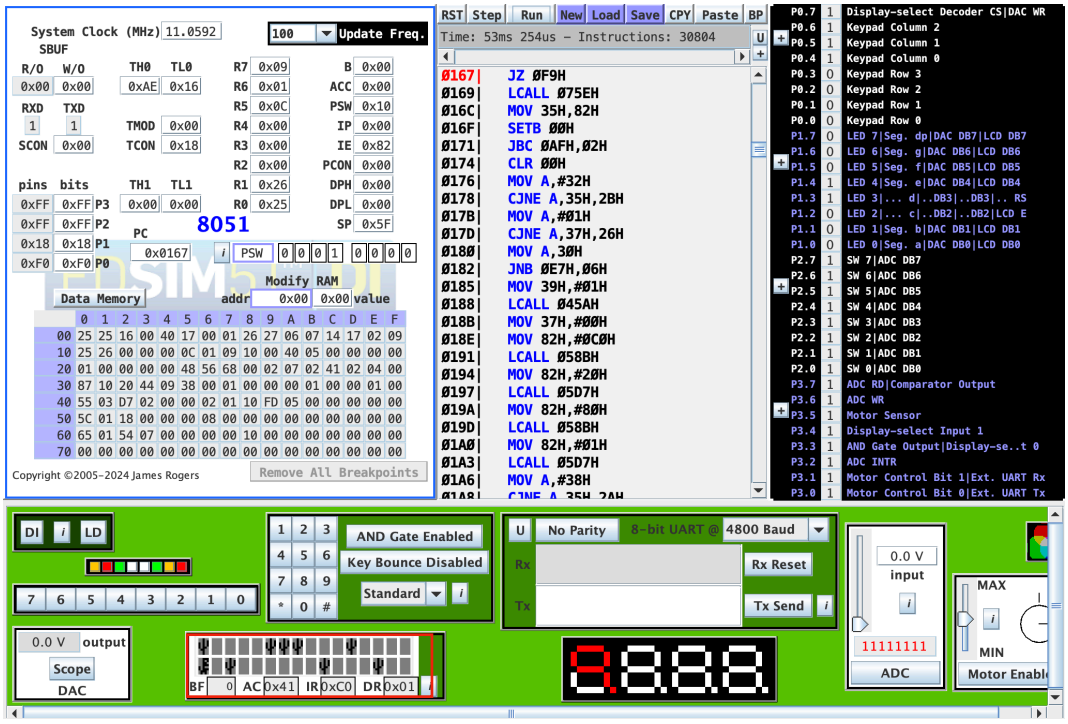

Fig. 24: Screenshot before the dinosaur moves upwards

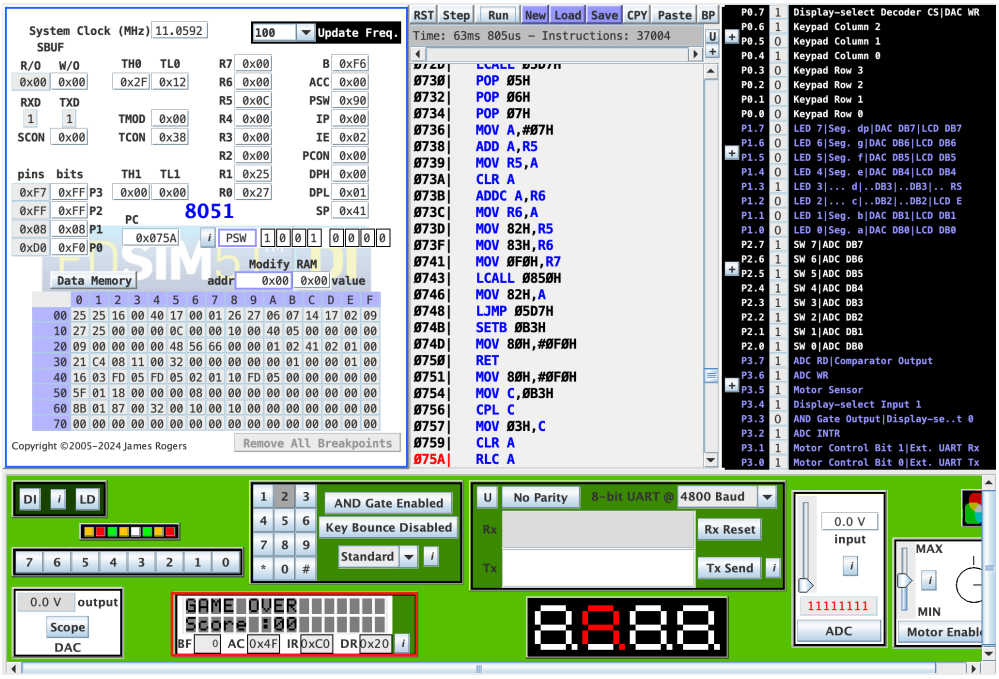

Fig. 25: Screenshot after the dinosaur moves upwards

§3.5 Game Control Thread

The `game_ctrl`'s code address is between 0x1E3 and 0x2D7. The screenshot for the first `game_ctrl` call is shown in **Fig. 26**.

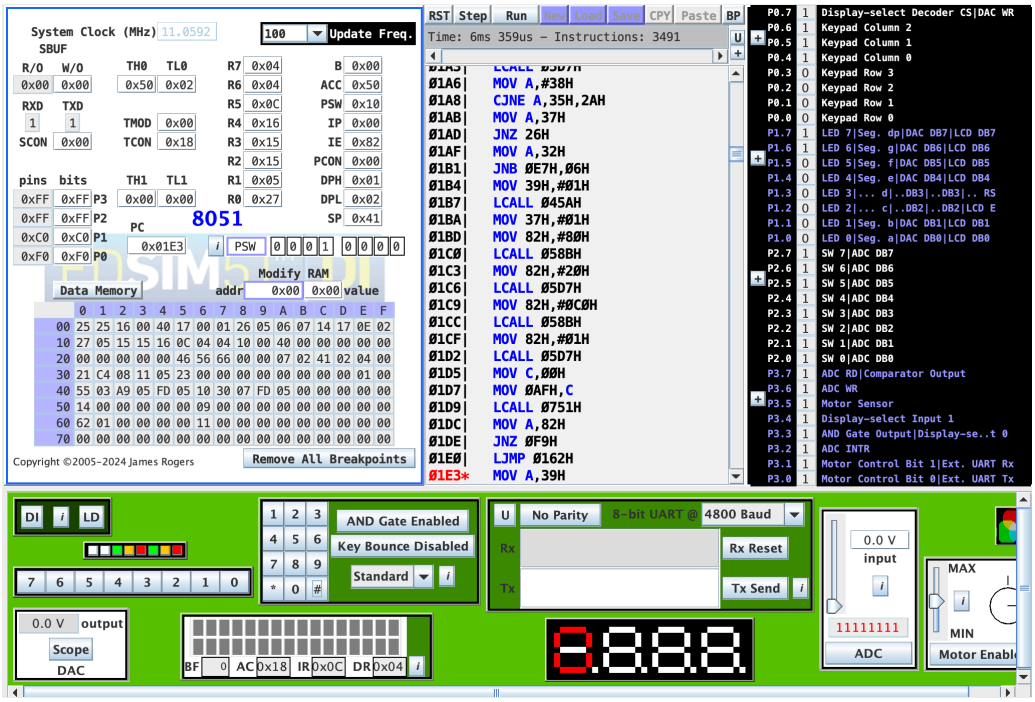

Fig. 26: Screenshot for the first `game_crtl` call

The `game_ctrl` shares the same thread with the main function. The game control does nothing till the end of the game$_{(1)}$. When `lose` = 1, it will display the score on the screen$_{(2)}$, and return to the main function$_{(3)}$. The main function will close the other two threads and allow the user to play for the next round. The code is shown in **Fig. 27.**

```
void game_ctrl(void)
Click to collapse the range.
    /*
     * [TODO]
     * initialize Tx for polling
     */
    while (1){
        if(lose){
            EA = 0;
            LCD_cursorGoTo(0, 0);
            LCD_write_char('G');
            LCD_write_char('A');
            LCD_write_char('M');
            LCD_write_char('E');
            LCD_write_char(' ');
            LCD_write_char('O');
            LCD_write_char('V');
            LCD_write_char('E');
            LCD_write_char('R');
            LCD_write_char(' ');
            LCD_write_char(' ');
            LCD_write_char(' ');
            LCD_write_char(' ');
            LCD_write_char(' ');
            LCD_write_char(' ');
            LCD_cursorGoTo(1, 0);
            LCD_write_char('S');
            LCD_write_char('c');
            LCD_write_char('o');
            LCD_write_char('r');
            LCD_write_char('e');
            LCD_write_char(' ');
            LCD_write_char(':');
            LCD_write_char(points/10 + '0');
            LCD_write_char(points%10 + '0');
            LCD_write_char(' ');
            LCD_write_char(' ');
            LCD_write_char(' ');
            LCD_write_char(' ');
            LCD_write_char(' ');
            LCD_write_char(' ');
            return;
        }
        ThreadYield();
    }
}
```

(2) (3) (1)

Fig. 27: Code for `game_ctrl`

§3.6 Race condition

In my design, the only race condition will occur in the `keypad_ctrl` thread. If the user presses a valid button just before the timer interrupts, the display might display two dinosaurs. To solve this, I made the part that deals with a pressed button a critical part. This solves the race condition.

§3.7 Questions and Answers

1. What data type do you use for the map?

I used four characters named `map_up[0]`, `map_up[1]`, `map_down[0]`, `map_down[1]` to save the map. To extract bits, AND & shift operations can be used. This allows me to use only four characters instead of 32 to save the cactus map.

2. How do you generate a new cactus?

A predefined cactus map is used. Each render only performs a circular left shift. As long as the predefined cactus map is valid and well-designed, the map will always be valid.