Operating Systems

Project Checkpoint 2

111062109 黃庭曜

Table of Contents

§1 Screenshots for compilation **2**

§1.1 Command “make clean” 2

§1.2 Command “make” 2

§2 Screenshots and explanation **2**

§2.1 ThreadCreate 2

§2.2 Producer 7

§2.3 Consumer 9

§2.4 Interrupt 11

§1 Screenshots for compilation

§1.1 Command "`make clean`"

```
huangtingyao@crazytingyao ppc2 % make clean
rm *.hex *.ihx *.lnk *.lst *.map *.mem *.rel *.rst *.sym *.asm *.lk
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

§1.2 Command "`make`"

```
huangtingyao@crazytingyao ppc2 % make
sdcc -c  testpreempt.c
testpreempt.c:62: warning 158: overflow in implicit constant conversion
sdcc -c  preemptive.c
preemptive.c:206: warning 85: in function ThreadCreate unreferenced function argument : 'fp'
preemptive.c:264: warning 158: overflow in implicit constant conversion
sdcc  -o testpreempt.hex testpreempt.rel preemptive.rel
```

Fig. 2: "`make`" command execution result

§2 Screenshots and explanation

§2.1 ThreadCreate

There are a total of two `ThreadCreate` calls. One is in the `Bootstrap` function, and the other is in the `main` function. The `Bootstrap` function creates a thread for the `main` function, and the `main` function creates a thread for the `Producer`. Besides, the `main` function will run the `Consumer`'s code directly, so the `Consumer` doesn't require an extra `ThreadCreate` call.

```
      Value  Global                          Global Defined In Module
      -----  -------------------------------  ------------------------
C:    00000014  _Producer                      testpreempt
C:    00000041  _Consumer                      testpreempt
C:    00000071  _main                          testpreempt
C:    0000007D  __sdcc_gsinit_startup          testpreempt
C:    00000081  __mcs51_genRAMCLEAR            testpreempt
C:    00000082  __mcs51_genXINIT               testpreempt
C:    00000083  __mcs51_genXRAMCLEAR           testpreempt
C:    00000084  _timer0_ISR                    testpreempt
C:    00000088  _Bootstrap                     preemptive
C:    000000AE  _myTimer0Handler               preemptive
C:    00000103  _ThreadCreate                  preemptive
C:    0000017E  _ThreadYield                   preemptive
C:    000001D9  _ThreadExit                    preemptive
```

Fig. 3: Function address list

The address list of all functions is shown in **Fig. 3** above. The `ThreadCreate` function is located at address 0x103. The screenshot before the first `ThreadCreate` call is shown in **Fig. 4**, where the `Bootstrap` function shown in **Fig. 5** starts at address 0x88 and calls the `ThreadCreate` function at address 0x96.

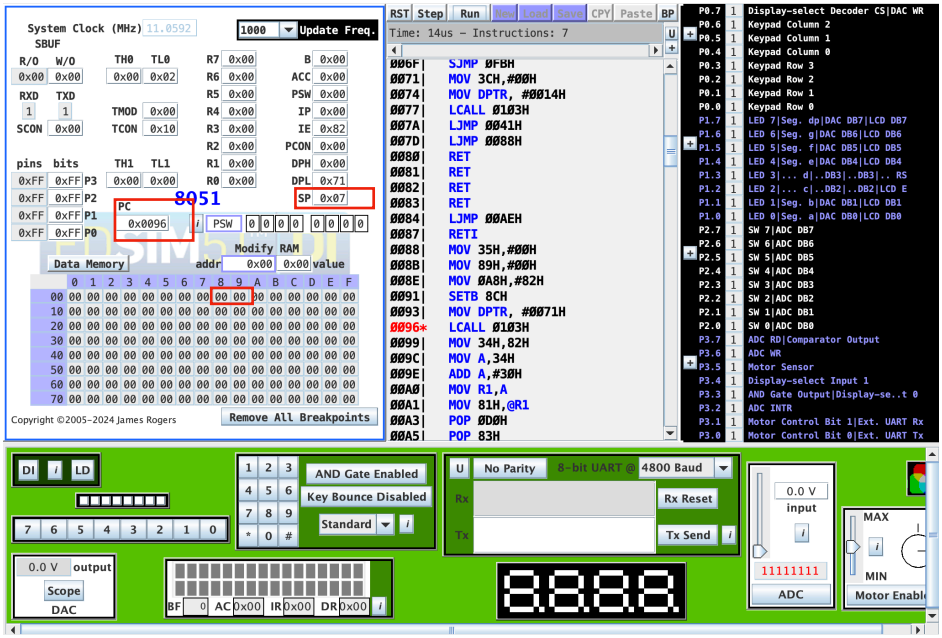

Fig. 4: Screenshot before first `ThreadCreate` function call

```
void Bootstrap(void) {
    threadMask = 0;
    TMOD = 0; // timer 0 mode 0
    // TH0 = 0xD8;
    // TL0 = 0x00;
    IE = 0x82; // enable timer 0 interrupt,
    TR0 = 1; // start running timer0
    currentThread = ThreadCreate(main);
    RESTORESTATE;
}
```

Fig. 5: Bootstrap function

After jumping to the `ThreadCreate` function by the instruction LCALL, we expect the next program counter of the current function (0x0099) to be saved to the stack. Since the stack pointer (`SP`) is now pointing to 0x07, the lower address of the program counter (0x99) will be pushed to data memory address 0x08, and the higher address of the program counter (0x00) will be pushed to data memory address 0x09. Then, the stack pointer will be refreshed to 0x09. The screenshot after finishing the first call of the `ThreadCreate` function is shown in **Fig. 6**.

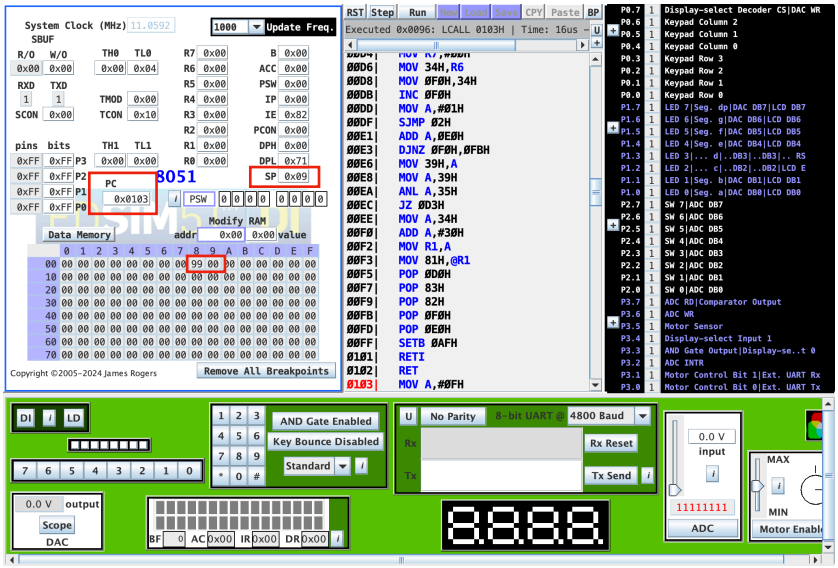

Fig. 6: Screenshot after first `ThreadCreate` function call

The `ThreadCreate` function first finds an unused thread by looking through the threadmask. For the main function, since all the threads are unused, Thread 0 will be assigned to it. Then, calculate the new stack pointer location $(0x3F)_{(2)}$, push the previous `DPL` and $\texttt{DPH}_{(3)}$, initialize `ACC, B, DPL, DPH` to $0_{(4)}$, and push it to the new $\text{stack}_{(5)}$. This is followed by setting the program status word (`PSW`) to `newthread` << 3 and pushing it to the new $\text{stack}_{(6)}$. This allows us to use `RESTORESTATE` to pop all `main` function's information immediately. The expected result is that address 0x40 will store 0x56 and 0x41 will store 0x00, which is the lower byte and the higher byte of the main function's address value, respectively, and addresses 0x42 to 0x46 will all be 0. (`newthread` = 0 for the first round) The corresponding codes $(1) - (6)$ are shown in **Fig. 7**, and the result is shown in **Fig. 8**.

```
ThreadID ThreadCreate(FunctionPtr fp){
    /*
    EA = 0;
    if(threadMask == 0x0F){
        return -1;
    }
    for(i = 0; i != MAXTHREADS; i++){
        // find a thread ID that is not in use
        temp = 1;
        temp <<= i;
        if(!(threadMask & temp)){                    (1)
            // take it, update the bit mask
            threadMask |= temp;
            newThread = i;
            break;
        }
    }
    //save the current SP in a temporary
    tempSP = SP;
    //calculate the starting stack location for new thread
    //set SP to the starting location for the new thread
    SP = (0x3F) + newThread * (0x10);                (2)
    //push the return address fp
    __asm
        PUSH DPL
        PUSH DPH                                     (3)
    __endasm;
    // initialize the registers to 0
    __asm
        ANL A, #0
        PUSH ACC
        PUSH ACC
        PUSH ACC                                     (4)
        PUSH ACC
    __endasm;
    // push PSW
    PSW = (newThread << 3);                          (5)
    __asm
        PUSH PSW                                     (6)
    __endasm;
    // write the current stack pointer to the saved stack pointer array for this newly created thread ID
    savedSP[newThread] = SP;
    // set SP to the saved SP in step c
    SP = tempSP;
    EA = 1;
    // return the newly created thread ID
    return newThread;
}
```

Fig. 7: Code for `ThreadCreate()`

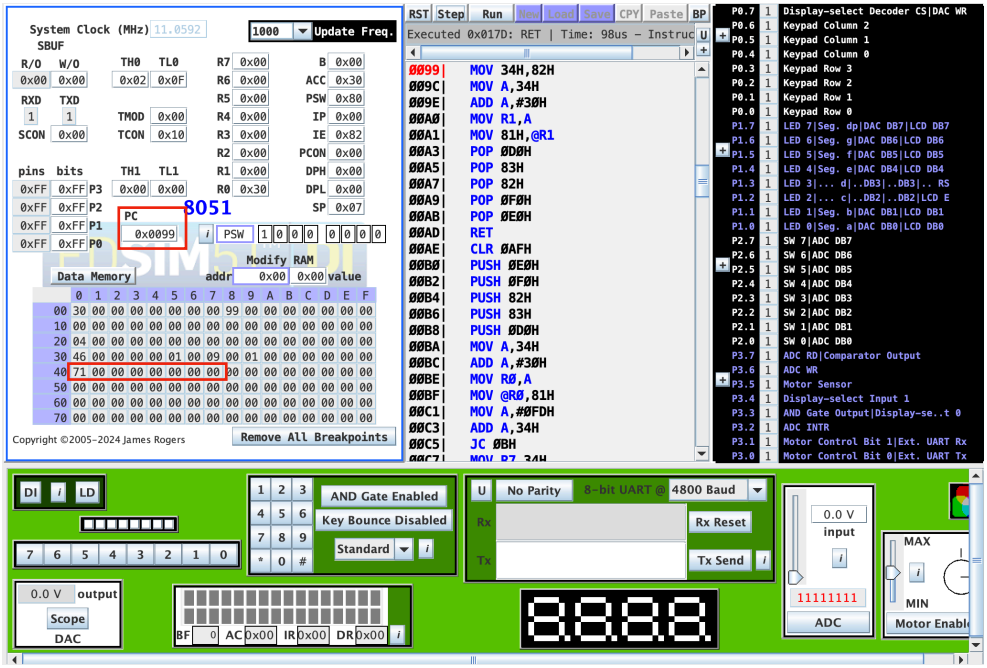

Fig. 8: Screenshot after returning from first `ThreadCreate` call

The screenshot before the second `ThreadCreate` call is shown in **Fig. 9**, where the `main` function starts at address 0x71 and calls the `ThreadCreate` function at address 0x77.

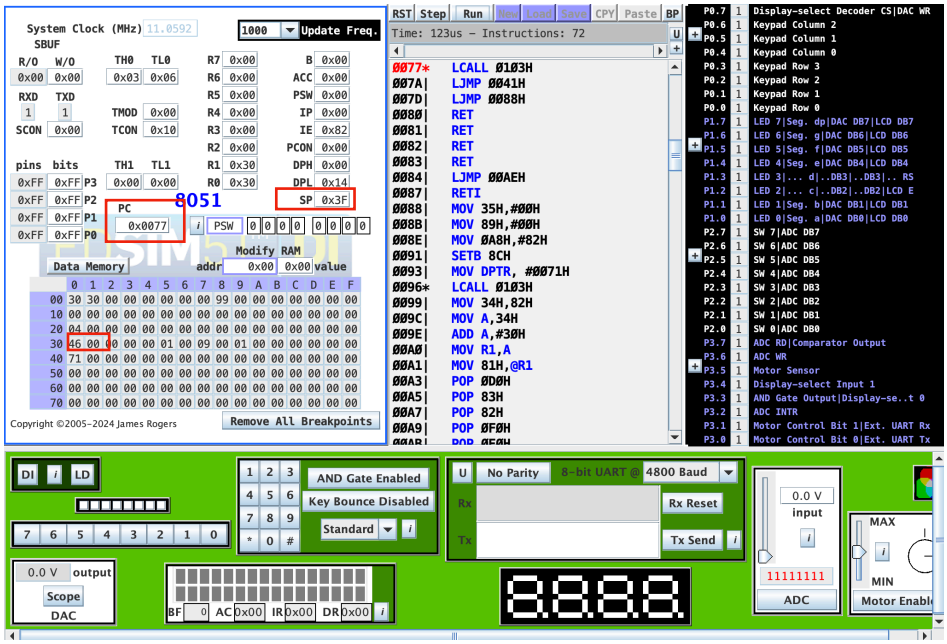

Fig. 9: Screenshot before second `ThreadCreate` function call

Similarly, after jumping to the `ThreadCreate` function by the instruction LCALL, we expect the next program counter of the current function (0x007A) to be saved to the stack. Since the stack pointer is now pointing to 0x3F, the lower address of the program counter (0x7A) will be pushed to data memory address 0x40, and the higher address of the program counter (0x00) will be pushed to data memory address 0x41. Then, the stack pointer will be refreshed to 0x41. The screenshot after the second call of `ThreadCreate` is shown in **Fig. 10**.

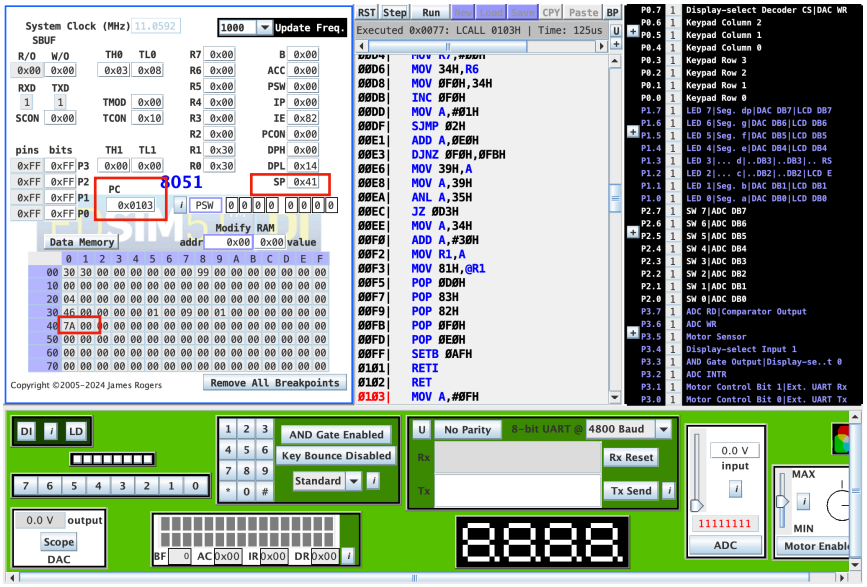

Fig. 10: Screenshot after the second `ThreadCreate` call

For the `Producer` function, since Thread 0 is used by `main` function, Thread 1 will be assigned to it. Then, calculate the new stack pointer location (0x4F), initialize `ACC, B, DPL, DPH` to 0, and push it to the new stack. The `PSW` will be 0x09 this time. (The last bit is the parity bit) Then, push it to the new stack. The expected result is that address 0x50 will store 0x14 and 0x51 will store 0x00, which is the lower and the higher byte of the `Producer`'s address value, addresses 0x52 to 0x55 will all be 0, and address 0x56 will be 0x09. The result is shown in **Fig. 11**.

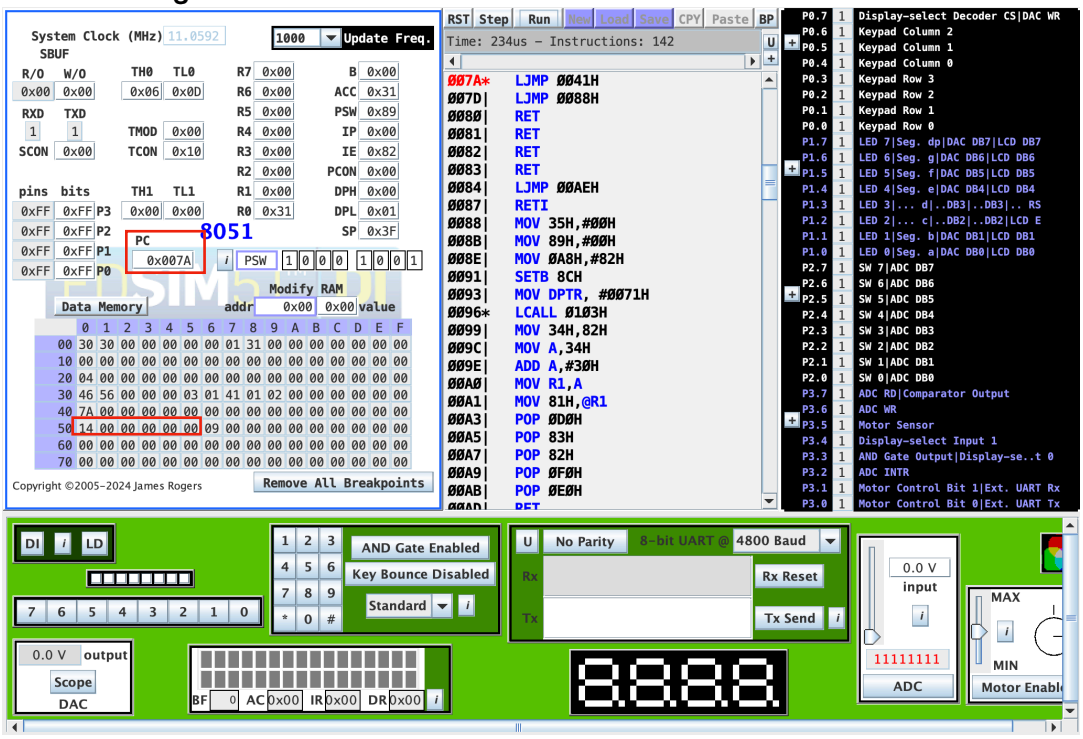

Fig. 11: Screenshot after returning from second `ThreadCreate` call

§2.2 Producer

The `Producer`'s code address is between 0x14 and 0x3F. The screenshot for the first `Producer` call is shown in **Fig. 12**.

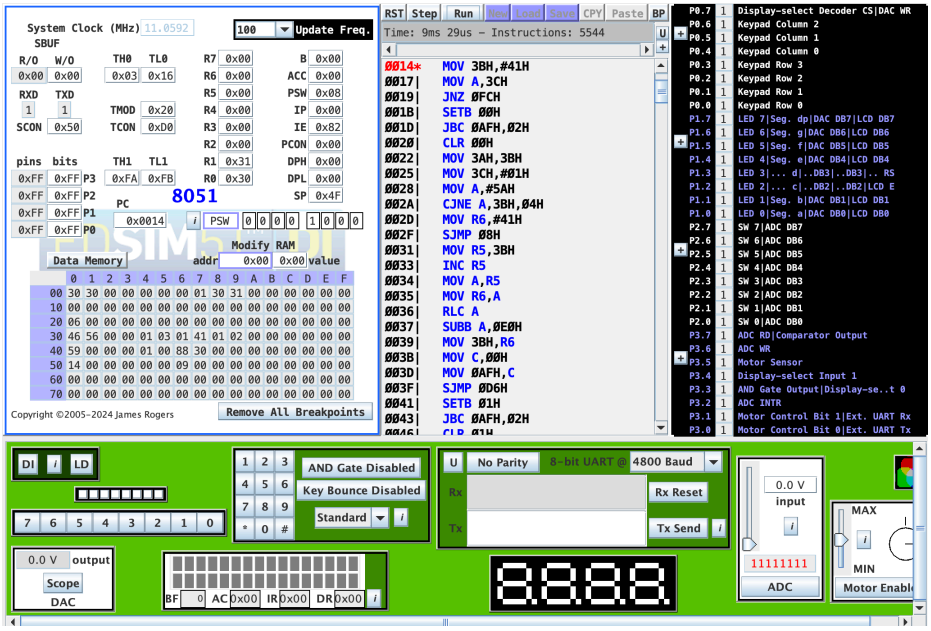

Fig. 12: Screenshot for first `Producer` call

In the first round, the `Producer` will set the initial `currentChar` variable (located at 0x3B) to the character 'A' $(0x41)_{(1)}$ and then enter an outer infinite loop. Since the buffer is empty, the `Producer` will push the `currentChar`'s value to the `buffer` (located at 0x3A), set the variable `bufferFull` (located at 0x3C) to $1_{(2)}$, and increment the `currentChar` variable (to $0x42)_{(3)}$. After that, the `Producer` will be stuck in a loop, waiting for the `Consumer` to pop the buffer. The screenshot before the first `Producer` call was preempted is shown in **Fig. 13**.

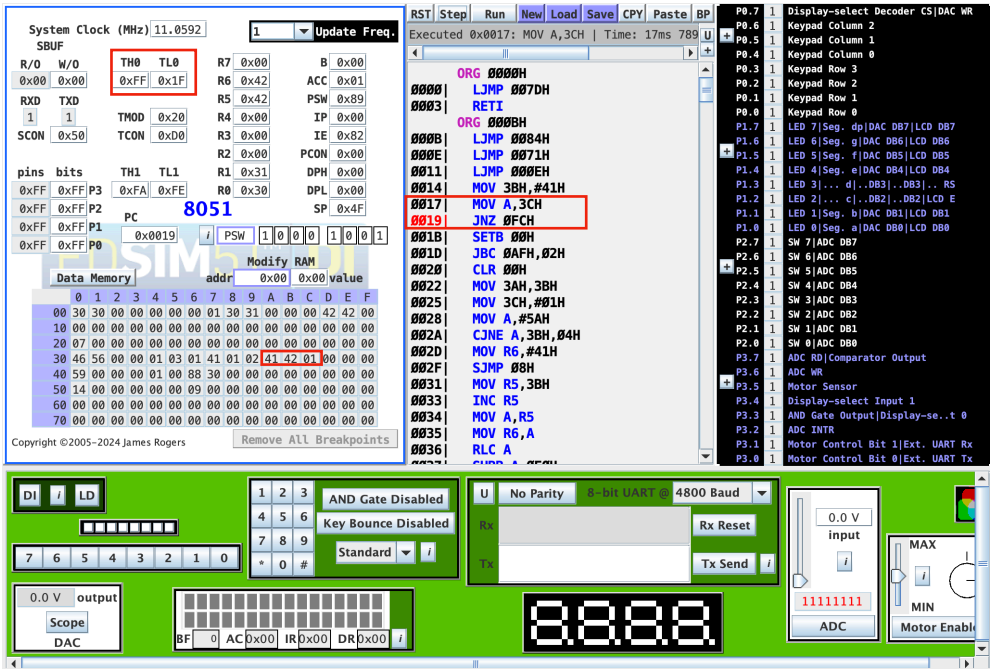

Fig. 13: Screenshot before first `Producer` call was preempted

The screenshot after the first `Producer` call was preempted is shown in **Fig. 14**.

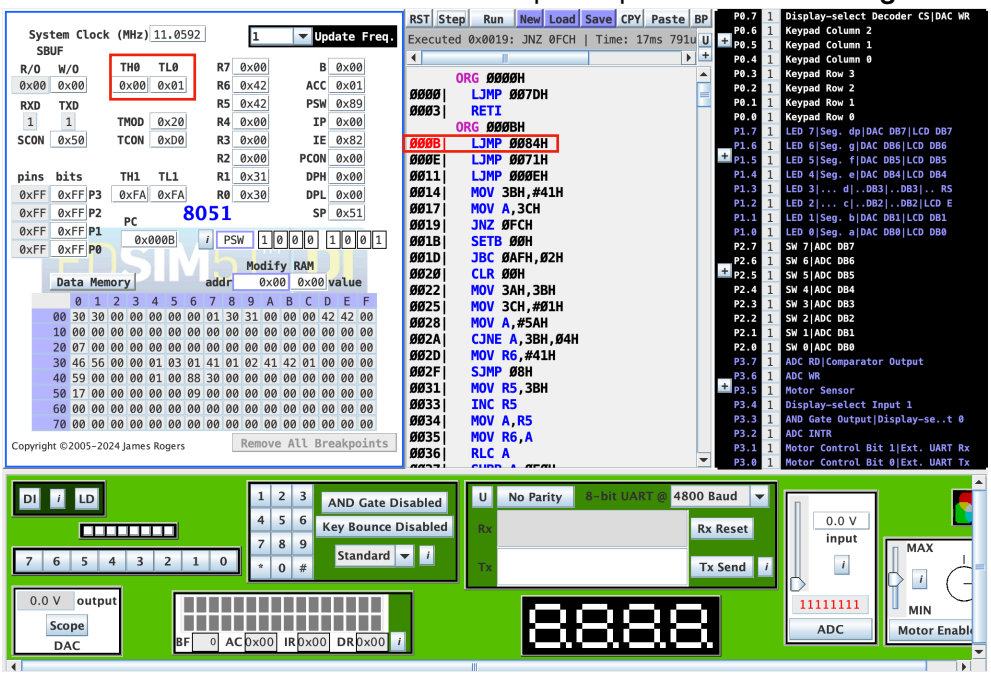

Fig. 14: Screenshot after first `Producer` call was preempted

The code of `Producer` is shown in **Fig. 15**.

```
void Producer(void)
{
    currentChar = 'A'; (1)
    while (1){
        while (bufferFull);
        __critical{
            buffer = currentChar;
            bufferFull = 1;
        }                                   (2)
        currentChar = (currentChar == 'Z') ? 'A' : currentChar + 1;   (3)
    }
}
```

Fig. 15: Code for `Producer`

In future rounds, if the buffer isn't empty (0x3C == 1), the `Producer` will continue to oscillate between the code addresses 0x17 and 0x19. Otherwise, proceed with the same instructions as the first round in the outer infinite loop and return to the oscillating part until it has been preempted.

§2.3 Consumer

The `Consumer`'s code address is between 0x41 and 0x6F. The screenshot for the first `Consumer` call is shown in **Fig. 16**.

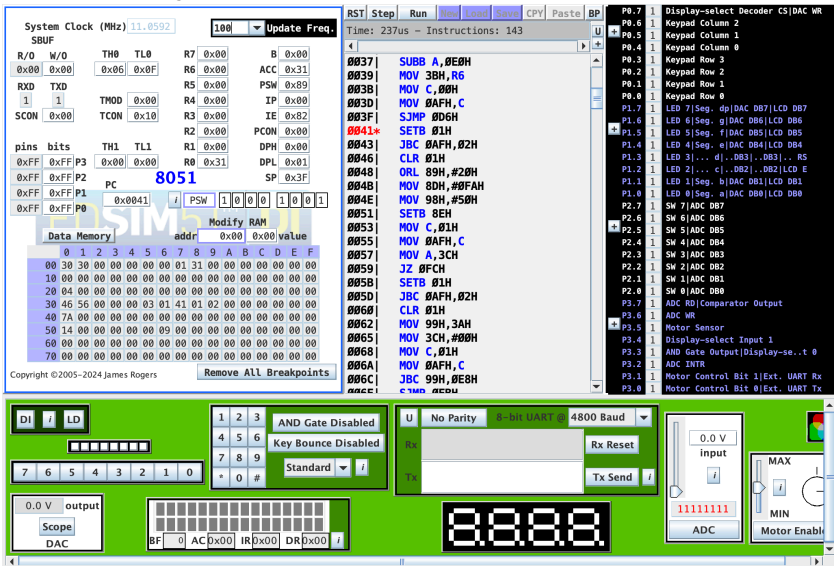

Fig. 16: Screenshot for first `Consumer` call

In the first round, the `Consumer` will initialize `Tx` for polling$_{(1)}$ and then enter the outer infinite loop. Since I run the `Consumer` first instead of the `Producer`, the first round is trapped in the waiting while–loop (addresses 0x57 & 0x59) immediately till preempted without any extra instructions$_{(2)}$. For the following rounds, if the buffer isn't empty (0x3C == 1), the `Consumer` will save the buffer's character to SBUF (located in address 0x99 and shown in the top left corner W/O slot in Edsim) and set the variable `bufferFull` back to 0$_{(3)}$. SBUF will let the `Consumer` send the character to Rx. If the transmission is completed, the `TI` flag will be pulled to 1, allowing the next character to be sent to SBUF. If not, the `Consumer` waits in the while–loop (addresses 0x6C & 0x6F) until the `TI` flag is pulled to 1 or be preempted$_{(4)}$. The `TI` flag is required to be pulled down manually$_{(5)}$, or we can't tell if the next character completes its transmission. The code is shown in **Fig. 17** and the screenshots for oscillating between 0x57 & 0x59 and 0x6C & 0x6F are shown in **Fig. 18 & 19**, respectively.

```
void Consumer(void)
{
    __critical{
        TMOD |= 0x20;
        TH1 = -6;
        SCON = 0x50;
        TR1 = 1;
    }                                   (1)
    while (1){
        while (!bufferFull);            (2)
        __critical{
            SBUF = buffer;
            bufferFull = 0;             (3)
        }
        while (!TI);                    (4)
        TI = 0;                         (5)
    }
}
```

Fig. 17: Code for `Consumer`

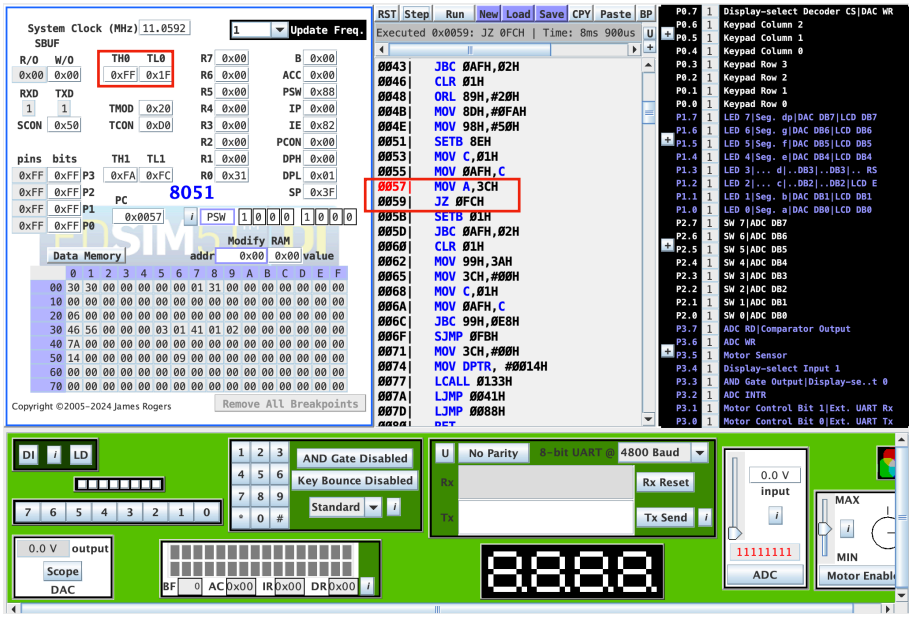

Fig. 18: `Consumer` oscillating between 0x57 & 0x59

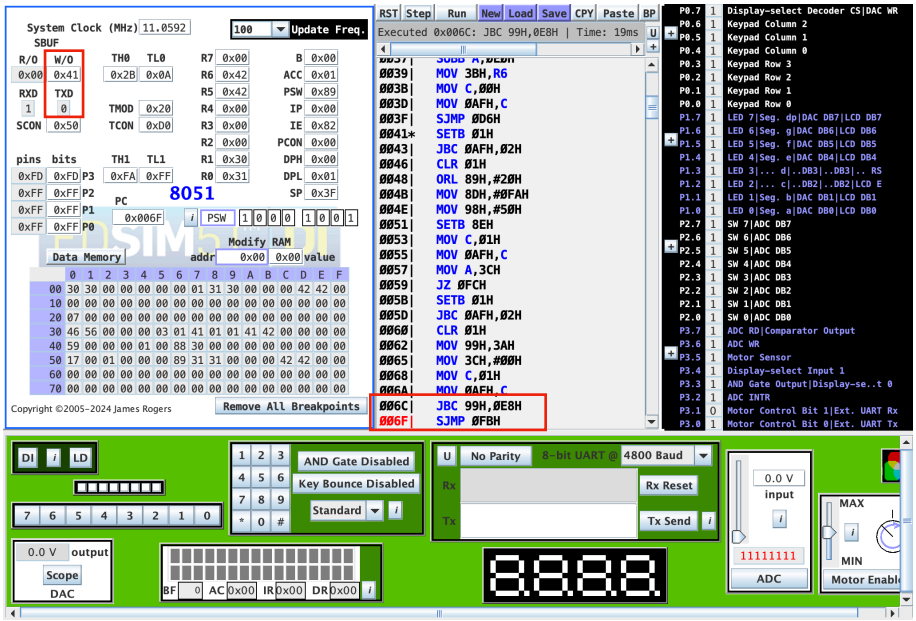

Fig. 19: `Consumer` oscillating between 0x6C & 0x6F

§2.4 Interrupt

Since we are using the timer interrupt with mode 0, which is a 2^{13} cycle timer. When TH0 = 0xFF and TL0 = 0x1F (Timer = 0x1FFF), the timer interrupt will trigger at the next cycle (Timer = 0x0000). From **Fig. 21 & 22**, we can observe that the code jumps to the interrupt vector and further jumps to the ISR (located at 0x84, code shown in **Fig. 20**).

```
void timer0_ISR(void) __interrupt(1) {
        __asm
            ljmp _myTimer0Handler
        __endasm;
}
```

Fig. 20: ISR code

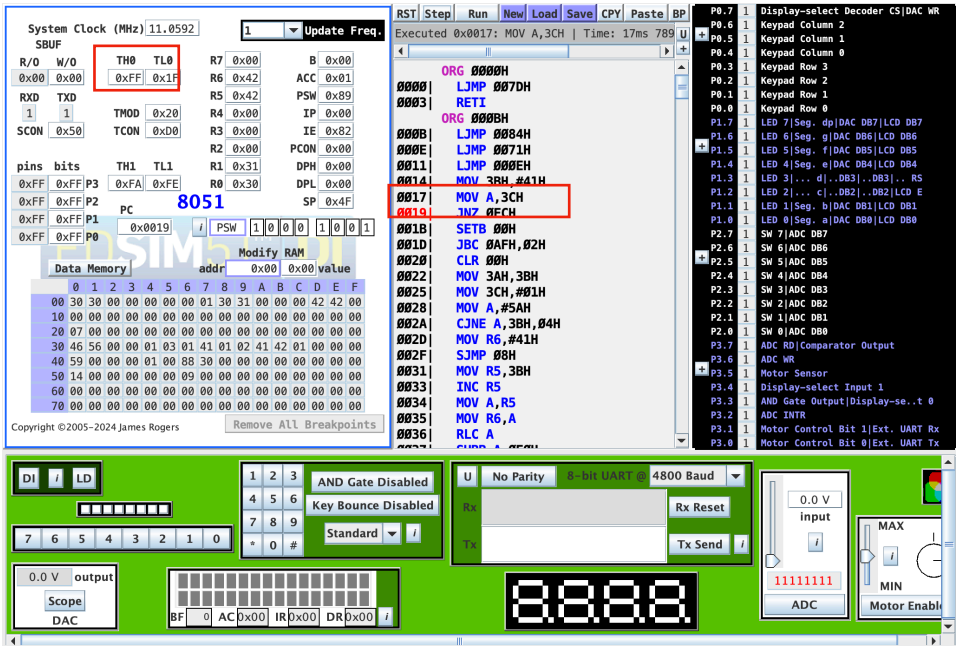

Fig. 21: Screenshot before first `Producer` call was preempted

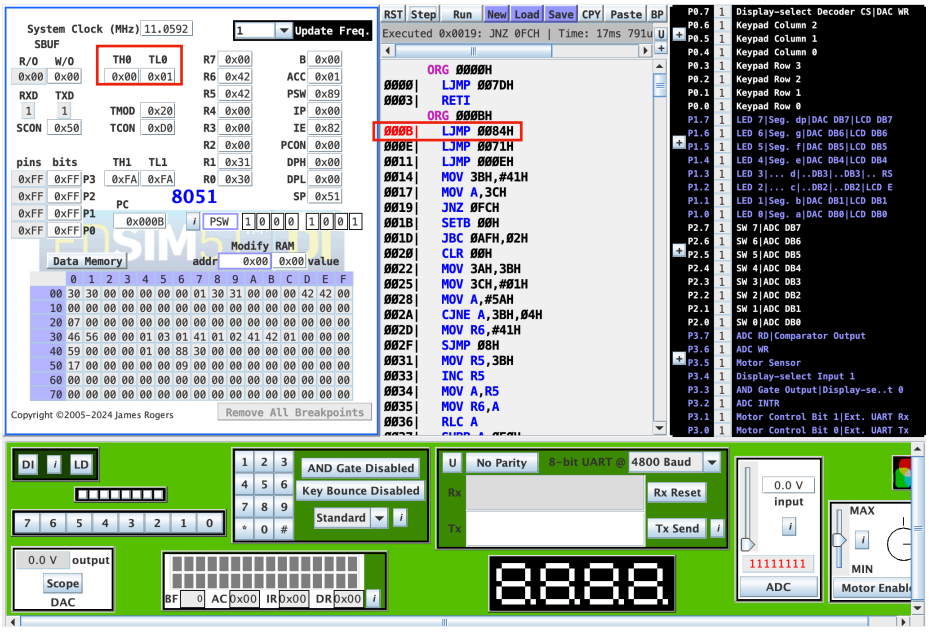

Fig. 22: Screenshot after first `Producer` call was preempted

Timer interrupt will use Round-Robin scheduling to let the next thread run. The details are similar to `ThreadYield()` in the previous checkpoint. The code is shown in **Fig. 23**.

```
void myTimer0Handler(void) {
    EA = 0; // don't do __critial
    SAVESTATE;
    do{
        currentThread = (currentThread < 3) ? currentThread + 1 : 0;
        temp = 1 << currentThread;
        if (threadMask & temp){
            break;
        }
    } while (1);
    RESTORESTATE;
    EA = 1;
    __asm
    RETI
    __endasm;
}
```

Fig. 23: Timer Handler code

After running for some time, the result is shown in **Fig. 24**.

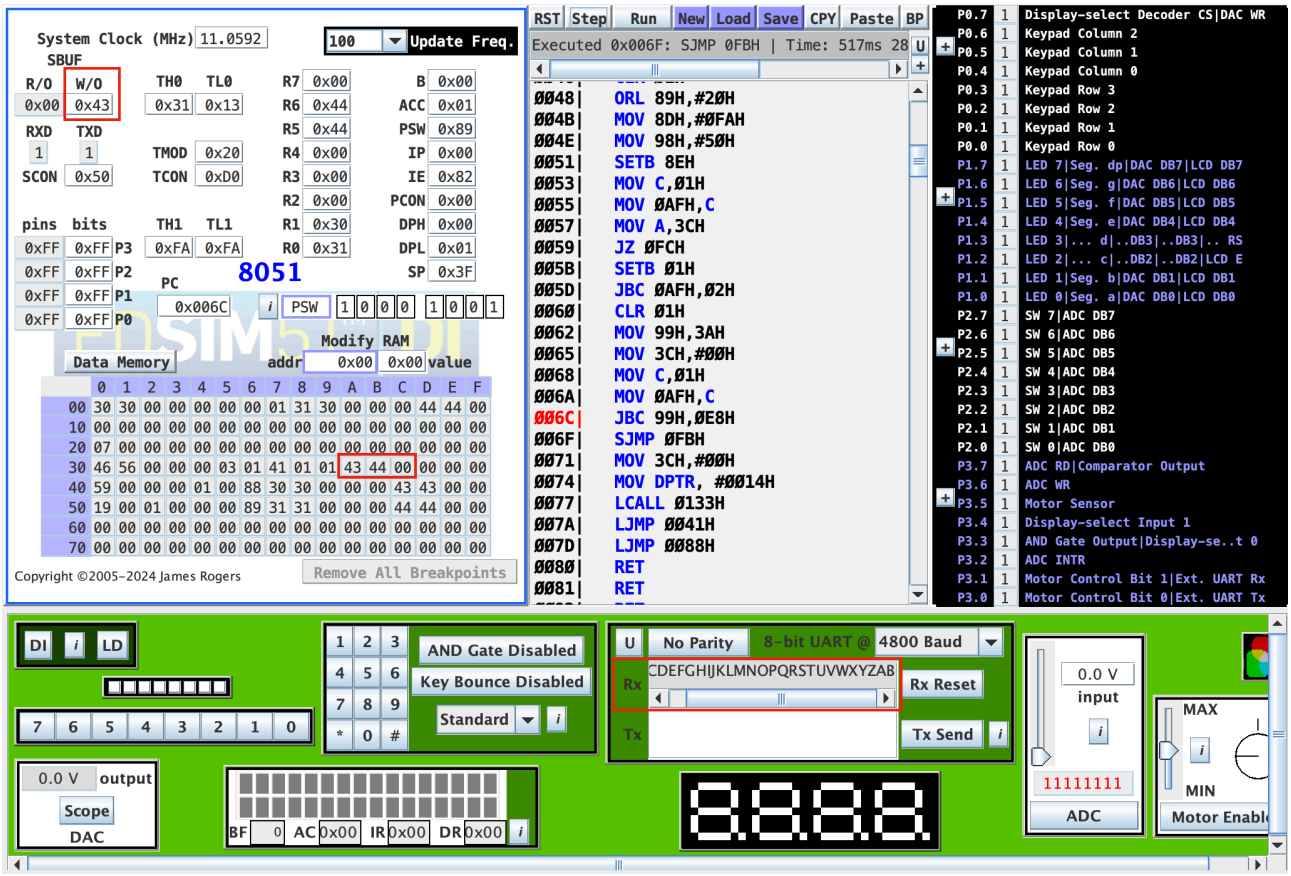

Fig. 24: Screenshot for after executing for some time

From `RX`, we can observe that the characters successfully reset to 'A' when the previous character is 'Z.' The `SBUF` waits to write 'C' to RX; the buffer is empty (0x3C), meaning the `consumer` has taken the character 'C' (at 0x3A) in the buffer. The overall result is correct.