Operating Systems

Project Checkpoint 3

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 Address list 2

§2.2 Producer 2

§2.3 Consumer 6

§1 Screenshots for compilation

§1.1 Command "`make clean`"

```
huangtingyao@crazytingyao ppc3 % 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 ppc3 % make
sdcc -c  testpreempt.c
testpreempt.c:66: 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
```

§2 Screenshots and explanation

§2.1 Address list

The address list of all functions is shown in **Fig. 3**.

```
      Value  Global                          Global Defined In Module
      -----  -------------------------------  ------------------------
C:   00000014  _Producer                       testpreempt
C:   00000065  _Consumer                       testpreempt
C:   000000AF  _main                           testpreempt
C:   000000C1  __sdcc_gsinit_startup           testpreempt
C:   000000C5  __mcs51_genRAMCLEAR             testpreempt
C:   000000C6  __mcs51_genXINIT                testpreempt
C:   000000C7  __mcs51_genXRAMCLEAR            testpreempt
C:   000000C8  _timer0_ISR                     testpreempt
C:   000000CC  _Bootstrap                      preemptive
C:   000000F2  _myTimer0Handler                preemptive
C:   00000147  _ThreadCreate                   preemptive
C:   000001C2  _ThreadYield                    preemptive
C:   00000216  _ThreadExit                     preemptive
```

Fig. 3: Function address list

§2.2 Producer

The `Producer`'s code address is between 0x14 and 0x63. The screenshot for the first `Producer` call is shown in **Fig. 4**.

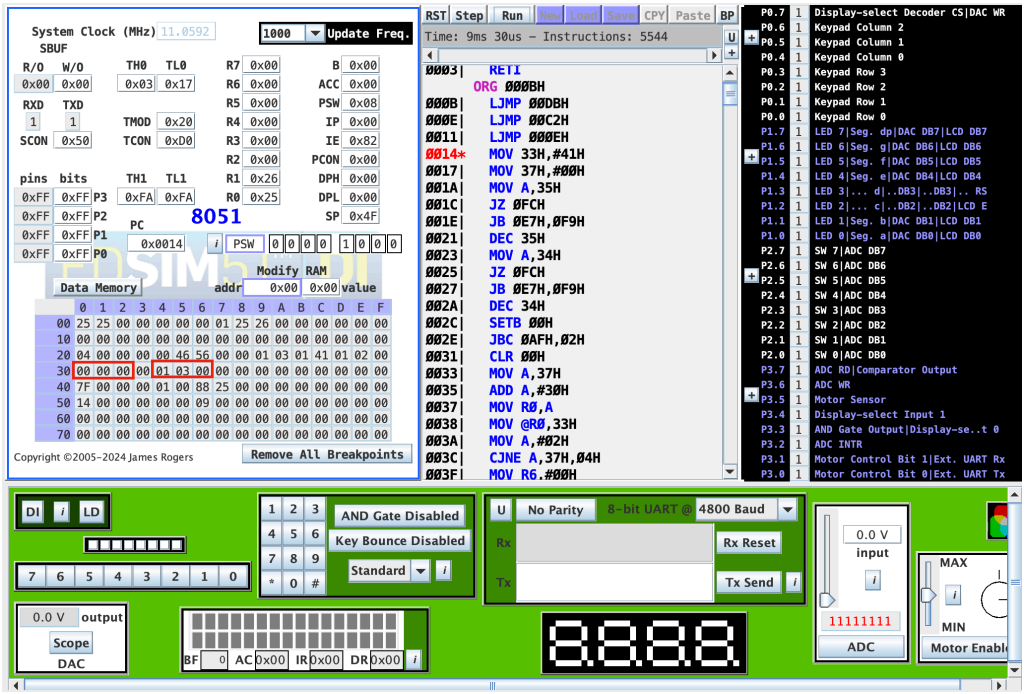

Fig. 4: Screenshot for first `Producer` call

The semaphore values of `mutex`, `empty`, and `full` are stored in 0x34, 0x35, and 0x36. For the first run, since the value of empty is 3, the Producer will enter the critical part through the semaphores$_{(1)}$ and push a character into the buffer$_{(2)}$ (at 0x30, the buffer ranges from 0x30 to 0x32). The semaphore wait code is shown in **Fig. 5**, `Producer`'s code is shown in **Fig. 6**. The screenshot before the `Producer` leaves the critical part is shown in **Fig. 7**.

```
#define SemaphoreWaitBody(S, label) \
    {                               \
        __asm \
            label: \
            MOV A, CNAME(S) \
            JZ label \
            JB ACC.7, label \
            dec  CNAME(S) \
        __endasm;       \
    }
```

Fig. 5: Screenshot of `SemaphoreWaitBody` code

```
#define LABEL(x) x ## $
#define L(x) LABEL(x)

void Producer(void){
    currentChar = 'A';
    producer_ptr = 0;
    while (1){
        SemaphoreWaitBody(empty,  L(__COUNTER__));
        SemaphoreWaitBody(mutex,  L(__COUNTER__));
        buffer[producer_ptr] = currentChar;
        producer_ptr = (producer_ptr == 2) ? 0 : producer_ptr + 1;
        SemaphoreSignal(mutex);
        SemaphoreSignal(full);
        currentChar = (currentChar == 'Z') ? 'A' : currentChar + 1;
    }
}
```

(1) (2) (3)

Fig. 6: Screenshot of `Producer` code

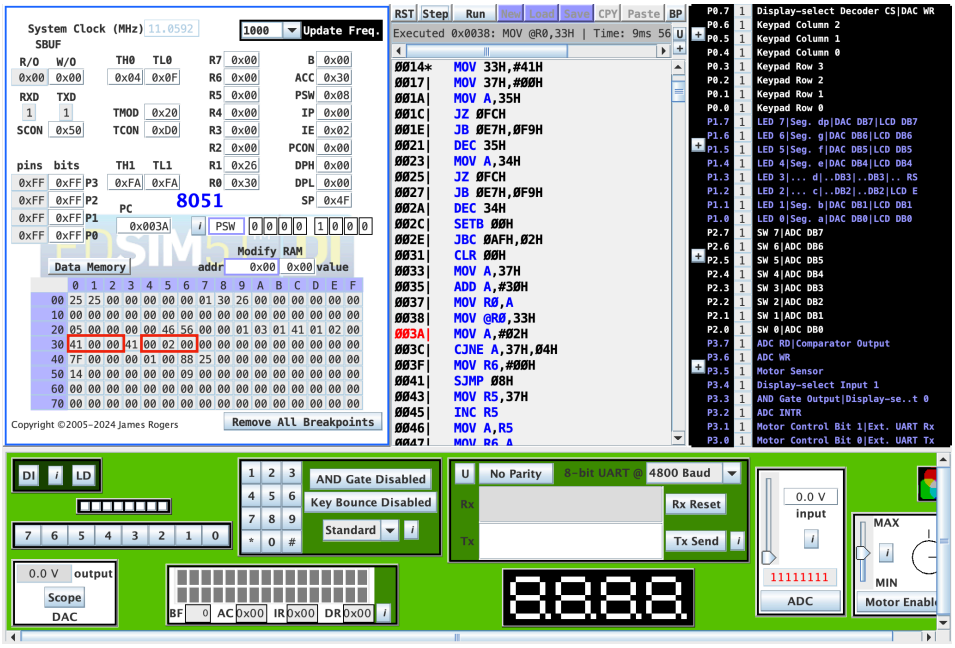

Fig. 7: Screenshot before the `Producer` leaves the critical part

The values of `mutex`, and `full` will be incremented after the `Producer` leaves the critical part$_{(3)}$, which is shown in **Fig. 8**, and the code for semaphore signal is shown in **Fig. 9**.

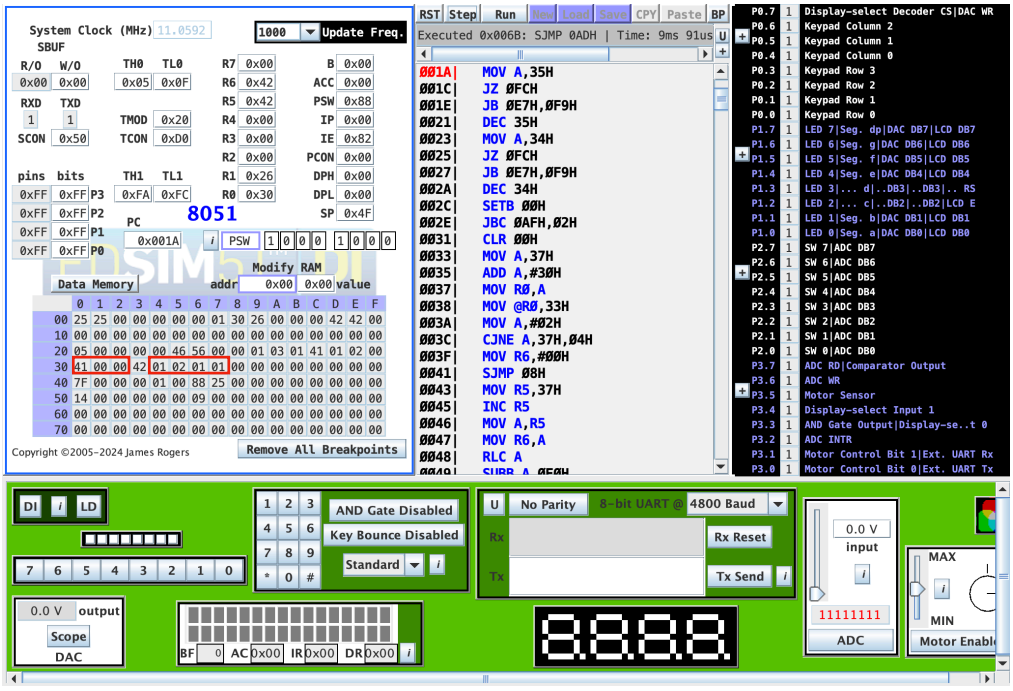

Fig. 8: Screenshot after the `Producer` leaves the critical part

```
#define SemaphoreSignal(s)   \
    {                        \
        __asm                \
        INC CNAME(s)  \
        __endasm;            \
    }
```

Fig. 9: `SemaphoreSignal` code

Since there are three slots in the buffer to fill, the Producer will resume pushing characters until the buffer is full. Then, the `Producer` will be stuck at the semaphore until it has been preempted by the timer. The buffer writing address is saved at address 0x37 and will perform a Round–Robin fashion. The results after the second and third push are shown in **Fig. 10 & 11**, respectively.

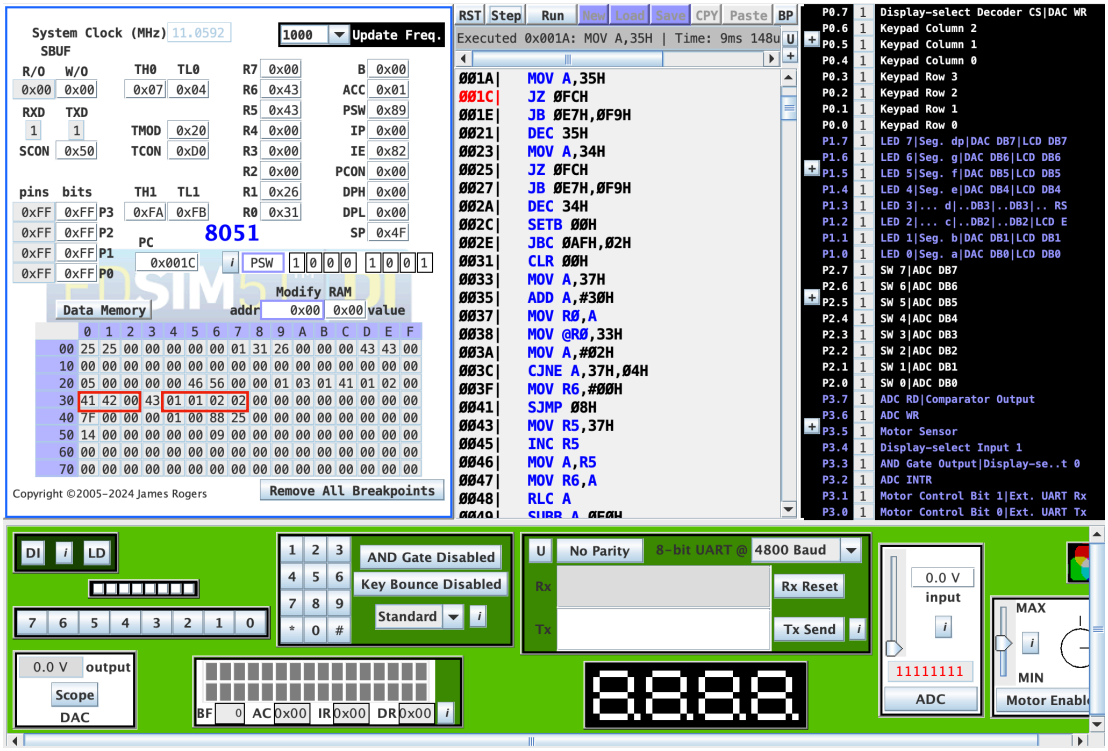

Fig. 10: Screenshot after the `Producer`'s second push

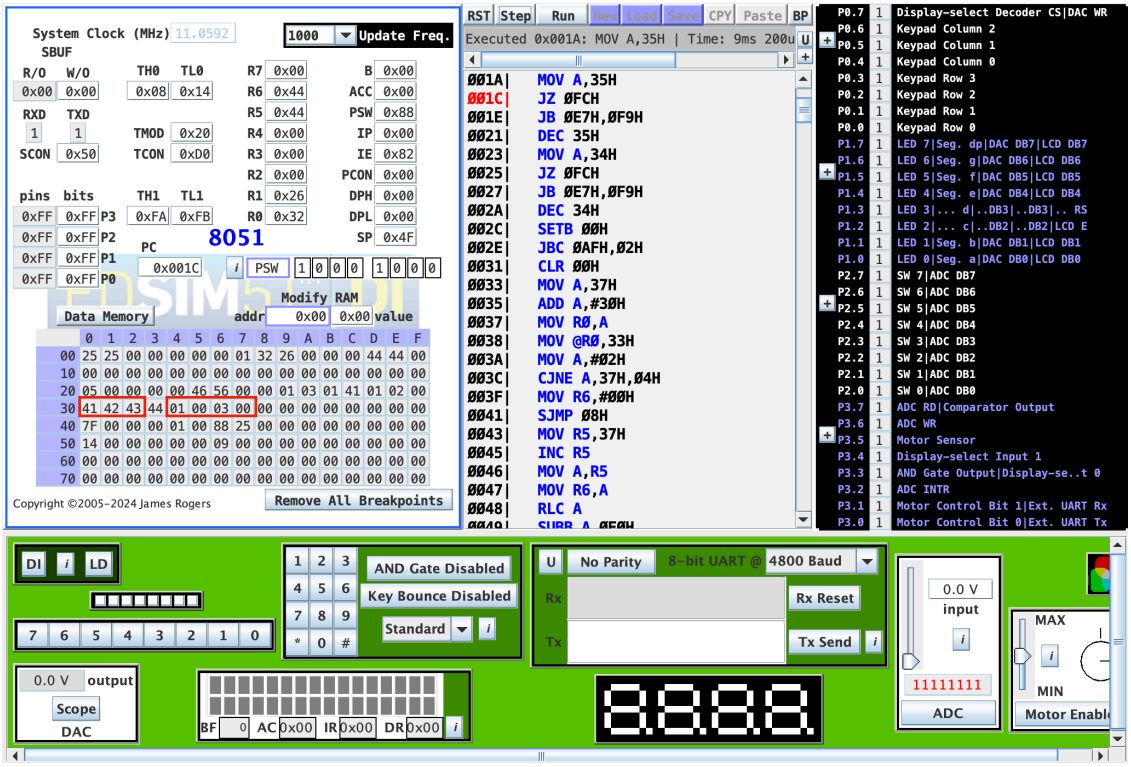

Fig. 11: Screenshot after the `Producer`'s third push

§2.3 Consumer

The `Consumer`'s code address is between 0x65 and 0xAD. The screenshot for the first `Consumer` call is shown in **Fig. 12**.

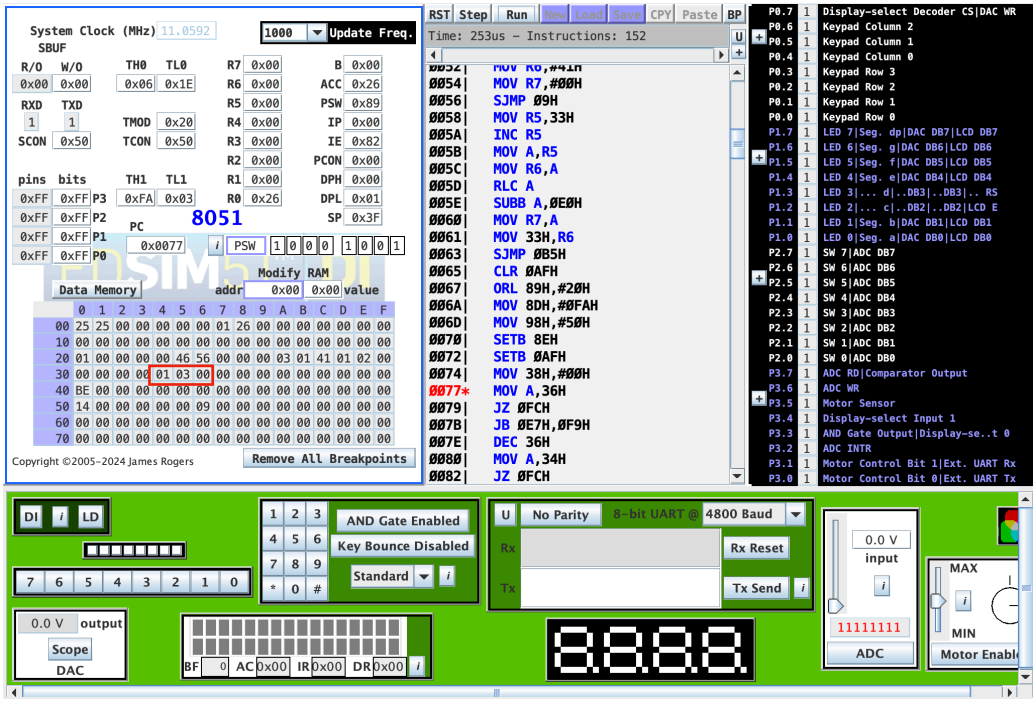

Fig. 12: Screenshot for first `Consumer` call

Since I wrote to run the `Consumer` first instead of the `Producer`, in the first round, the `Consumer` will be stuck at the `full` semaphore$_{(1)}$ (located at 0x36). This round will be preempted by the timer by doing only initialization$_{(2)}$. The screenshot of the code is shown in **Fig. 13**.

```
void Consumer(void)
{
    EA = 0;
    TMOD |= 0x20;
    TH1 = -6;
    SCON = 0x50;
    TR1 = 1;
    EA = 1;
    consumer_ptr = 0;
    while (1){
        SemaphoreWaitBody(full,  L(__COUNTER__));
        SemaphoreWaitBody(mutex,  L(__COUNTER__));
        SBUF = buffer[consumer_ptr];
        consumer_ptr = (consumer_ptr == 2) ? 0 : consumer_ptr + 1;
        SemaphoreSignal(mutex);
        SemaphoreSignal(empty);
        while (!TI);
        TI = 0;
    }
}
```

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

Fig. 13: Screenshot of `Consumer` code

The second `Consumer` call is shown in **Fig. 14.** In this round, the buffer is full of characters, and the `full` semaphore is correctly set to 3. So, the `Consumer` enters the critical part and pops the buffer to the $SBUF_{(3)}$. Then, it signals the mutex and the empty $semaphores_{(4)}$. The result is shown in **Fig. 15**.

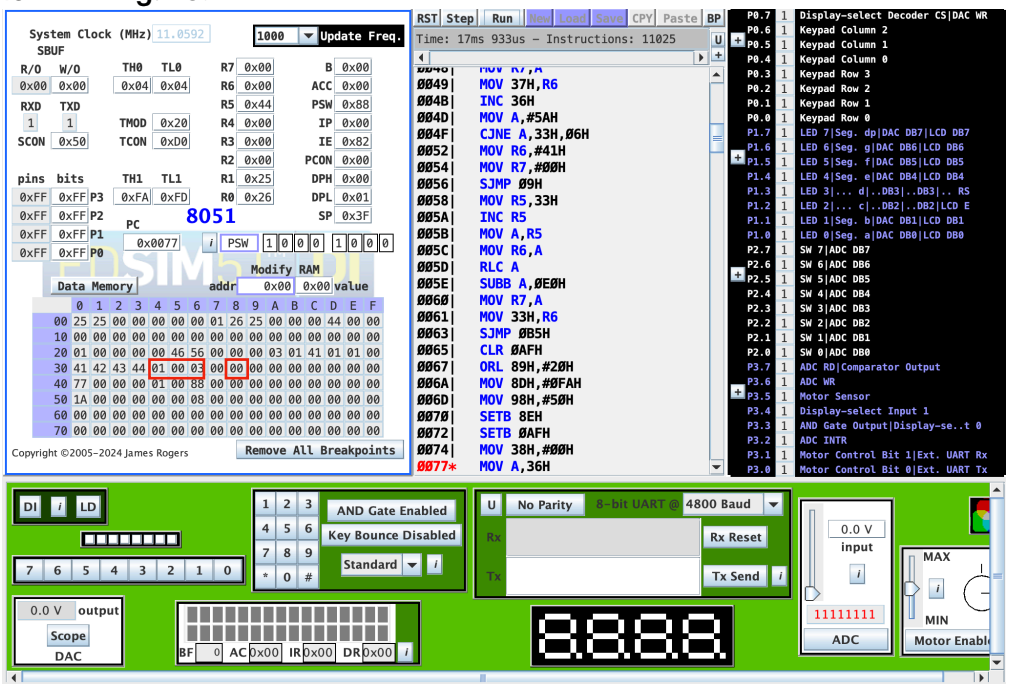

Fig. 14: Screenshot for second `Consumer` call

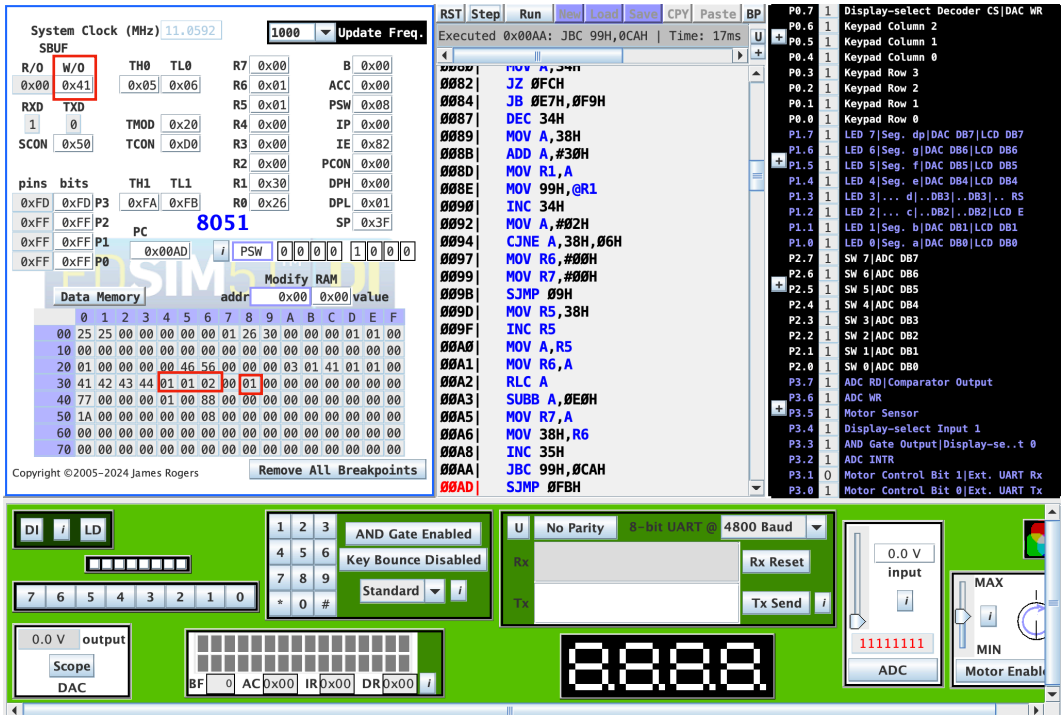

Fig. 15: Screenshot after `Consumer` pushes the SBUF

After leaving the critical section, the `Consumer` starts to wait for the RX transmission to complete. If completed, the `Consumer` pops another item in the buffer. The buffer reading address is saved at address 0x38 and will perform a Round–Robin fashion. The screenshots for popping the second and the third items are shown in **Fig. 16 & 17.**

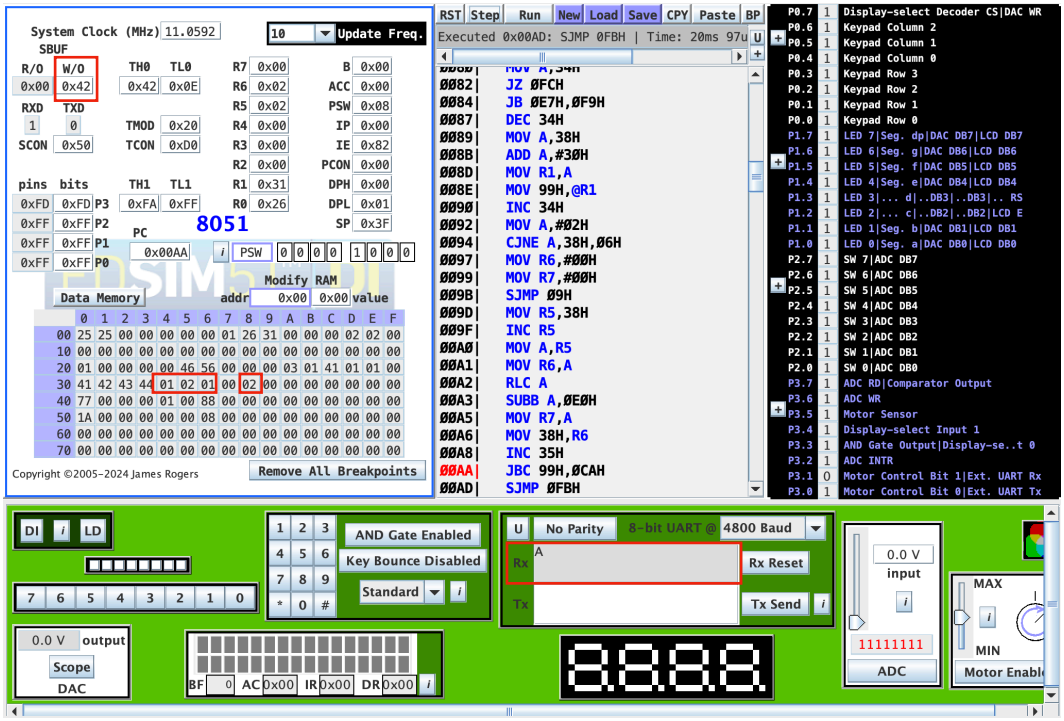

Fig. 16: Screenshot after `Consumer` pops 'B'

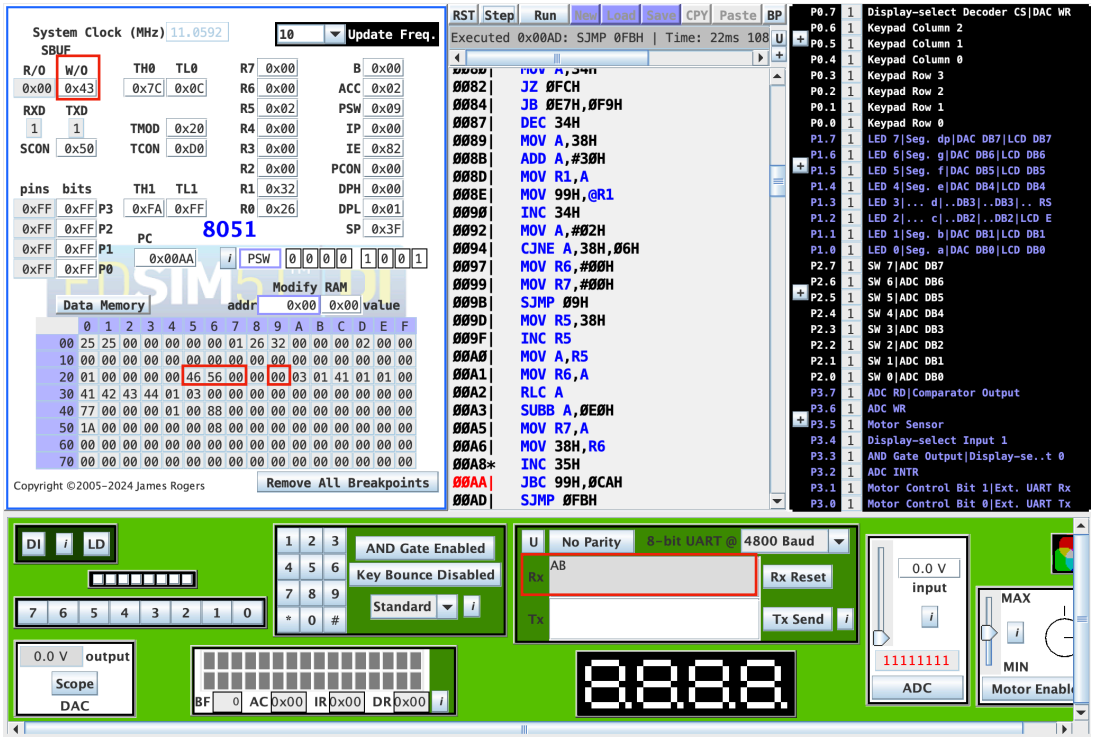

Fig. 17: Screenshot after `Consumer` pops 'C'

Finally, I let the code run for some rounds, and the result is shown in **Fig. 18**.

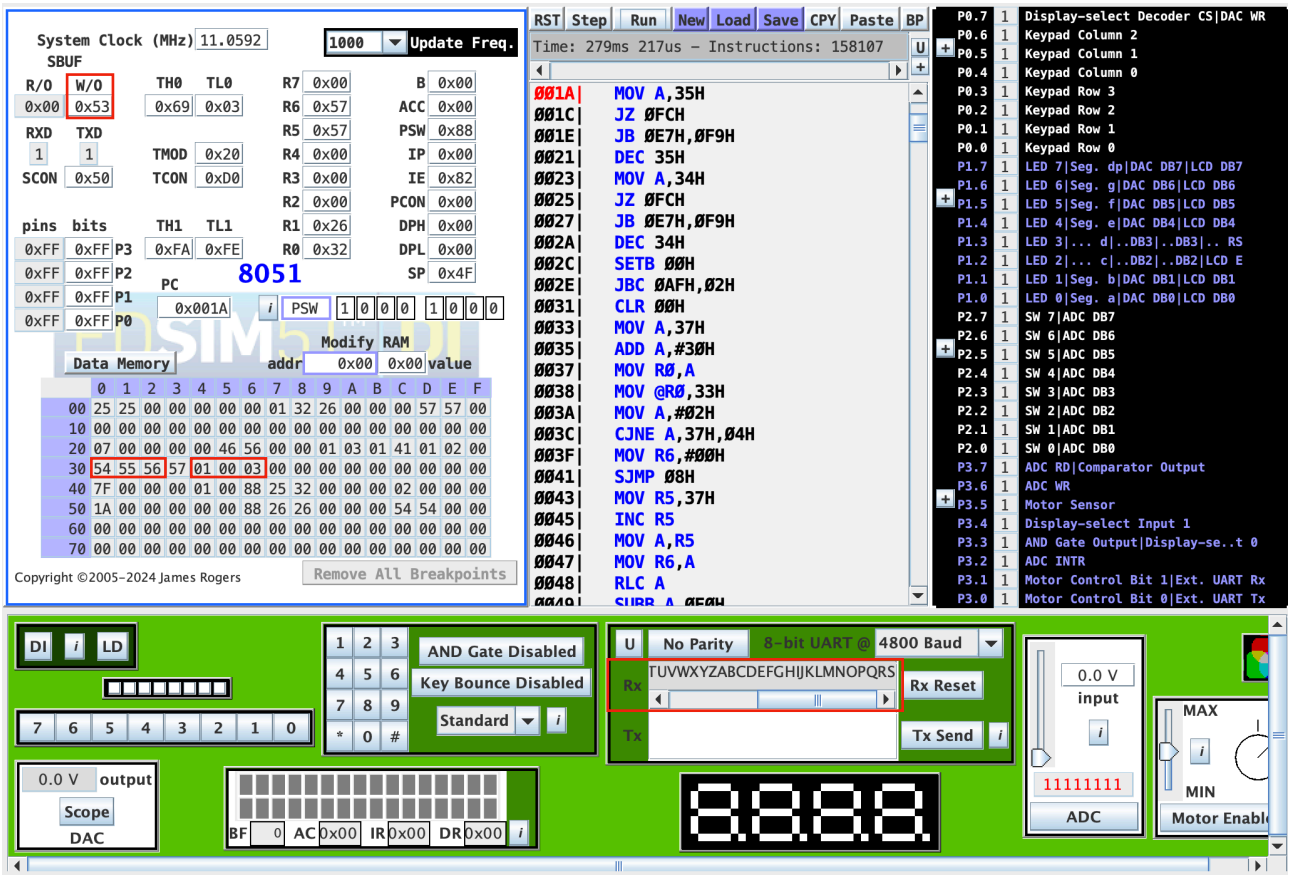

Fig. 18: Execution result of running for a while

We can observe that the code stops at the `Producer`; since the RX has output 'S' (0x53), the buffer can be overwritten with 'T,' 'U,' and 'V.' The `Producer` has completed its writing, so the addresses 0x30 to 0x32 are 'T,' 'U,' and 'V' already, and the semaphores are correctly assigned.