Operating Systems

Project Checkpoint 1

111062109 黃庭曜

Table of Contents

§1 Screenshots for compilation

§1.1 Command "`make clean`"

```
huangtingyao@crazytingyao ppc1 % 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 ppc1 % make
sdcc -c  testcoop.c
testcoop.c:61: warning 158: overflow in implicit constant conversion
sdcc -c  cooperative.c
cooperative.c:192: warning 85: in function ThreadCreate unreferenced function argument : 'fp'
cooperative.c:249: warning 158: overflow in implicit constant conversion
sdcc  -o testcoop.hex testcoop.rel cooperative.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:    00000009  _Producer                         testcoop
C:    00000030  _Consumer                         testcoop
C:    00000056  _main                             testcoop
C:    00000062  __sdcc_gsinit_startup             testcoop
C:    00000066  __mcs51_genRAMCLEAR               testcoop
C:    00000067  __mcs51_genXINIT                  testcoop
C:    00000068  __mcs51_genXRAMCLEAR              testcoop
C:    00000069  _Bootstrap                        cooperative
C:    00000087  _ThreadCreate                     cooperative
C:    0000010C  _ThreadYield                      cooperative
C:    00000165  _ThreadExit                       cooperative
```

Fig. 3: Function address list

The address list of all functions is shown in **Fig. 3** above. The `ThreadCreate` function is located at address 0x87. The screenshot before the first `ThreadCreate` call is shown in **Fig. 5**, where the `Bootstrap` function (shown in **Fig. 4**) starts at address 0x69 and calls the `ThreadCreate` function at address 0x6F.

```
void Bootstrap(void){
    threadMask = 0;
    currentThread = ThreadCreate(main);
    RESTORESTATE;
}
```

Fig. 4: Bootstrap function

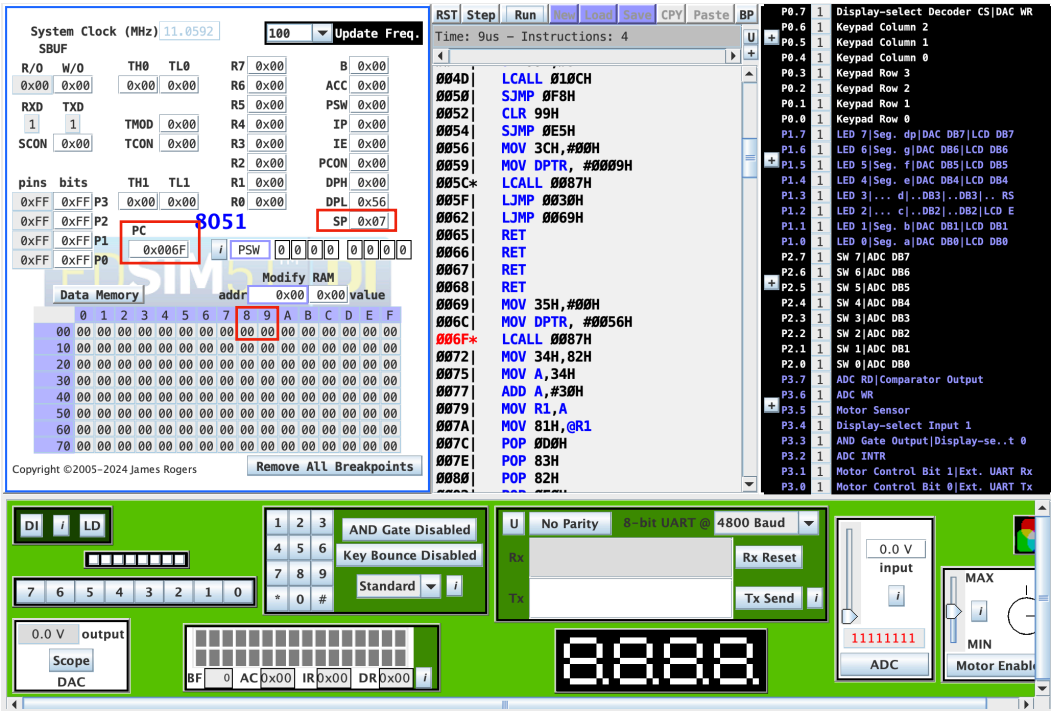

Fig. 5: Screenshot before first `ThreadCreate` function call

After jumping to the `ThreadCreate` function by the instruction LCALL, we expect the next program counter of the current function (0x0072) to be saved to the stack. Since the stack pointer (`SP`) is now pointing to 0x07, the lower address of the program counter (0x72) 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 the first call of the `ThreadCreate` function is shown in **Fig. 6**.

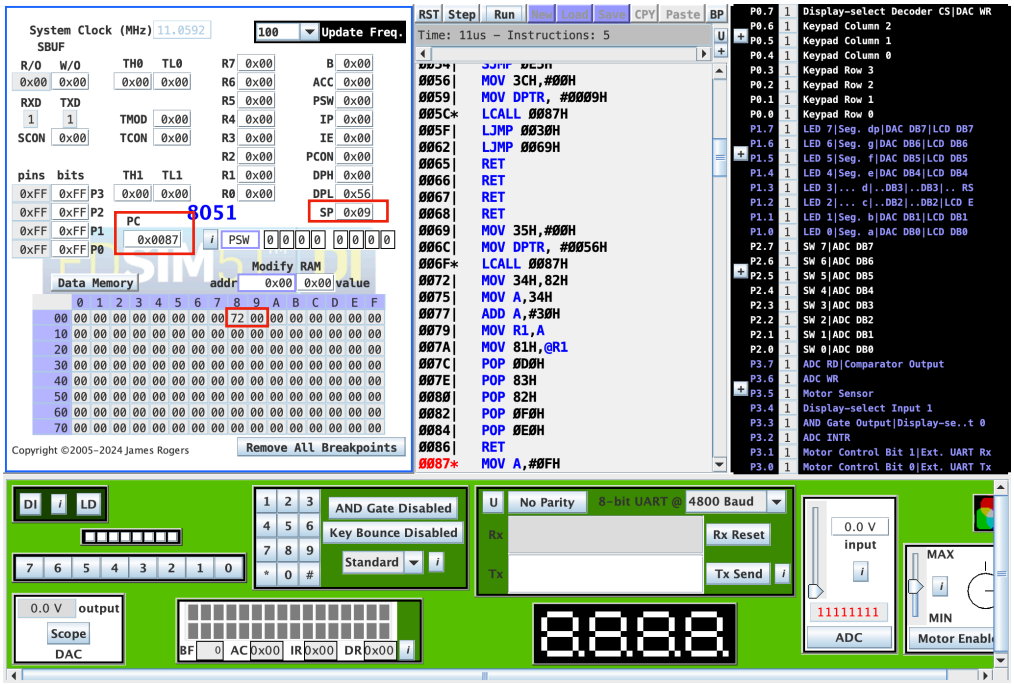

Fig. 6: Screenshot after first `ThreadCreate` function call

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()`

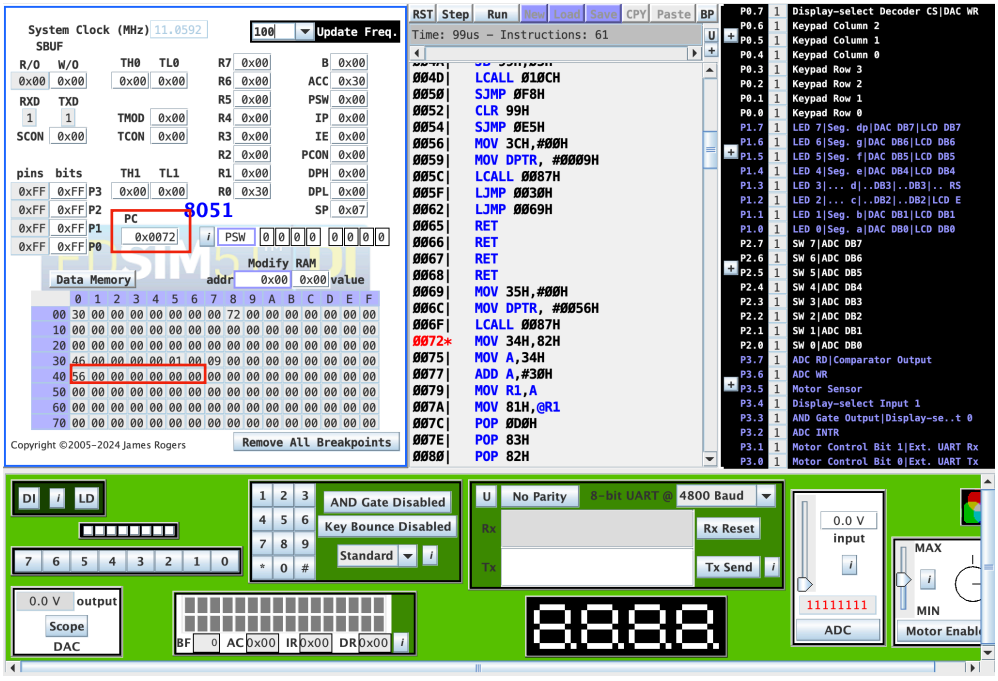

Fig. 8: Screenshot after returning from first `ThreadCreate` call

The screenshot before the second `ThreadCreate` call is shown in **Fig. 9**, and the corresponding code is shown in **Fig. 10**. where the `main` function starts at address 0x56 and calls the `ThreadCreate` function at address 0x5C.

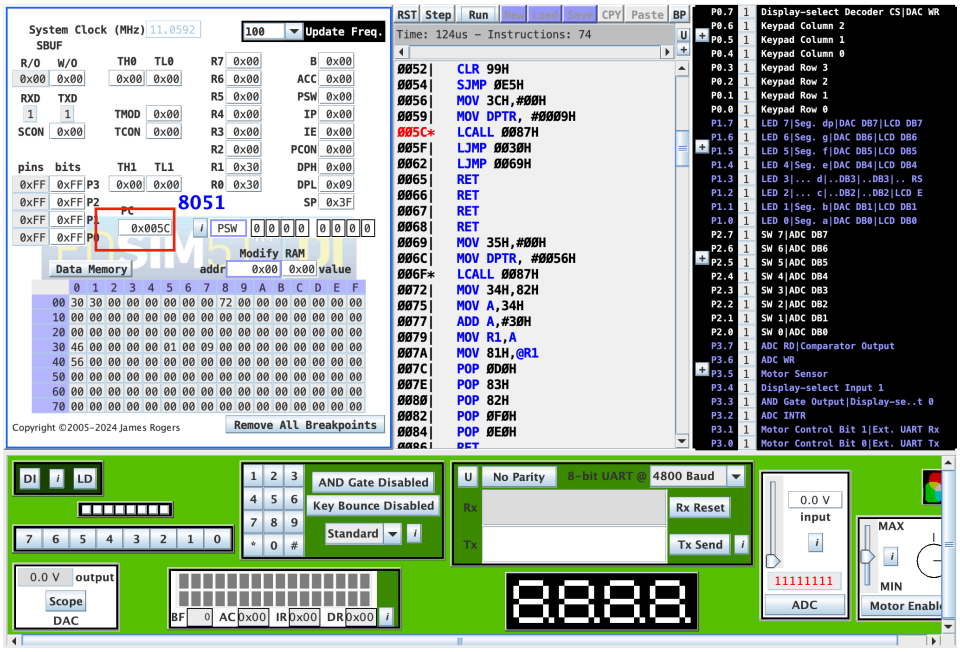

Fig. 9: Screenshot before second `ThreadCreate` function call

```
void main(void)
{
    bufferFull = 0;
    ThreadCreate(Producer);
    Consumer();
}
```

Fig. 10: Screenshot of the main function

Similarly, after jumping to the `ThreadCreate` function by the instruction LCALL, we expect the next program counter of the current function (0x005F) to be saved to the stack. Since the stack pointer is now pointing to 0x3F, the lower address of the program counter (0x5F) 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. 11**.

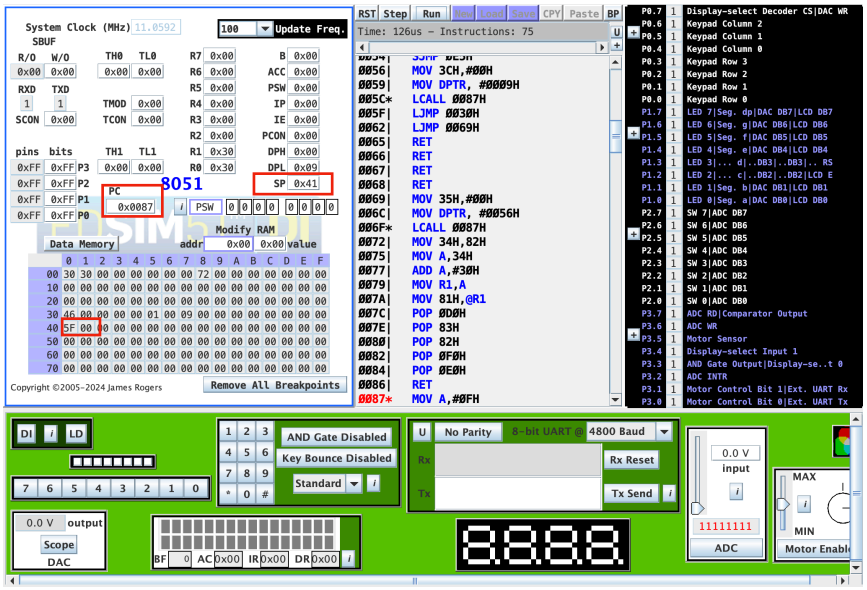

Fig. 11: Screenshot after the second `ThreadCreate` call

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 0x09 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. 12**.

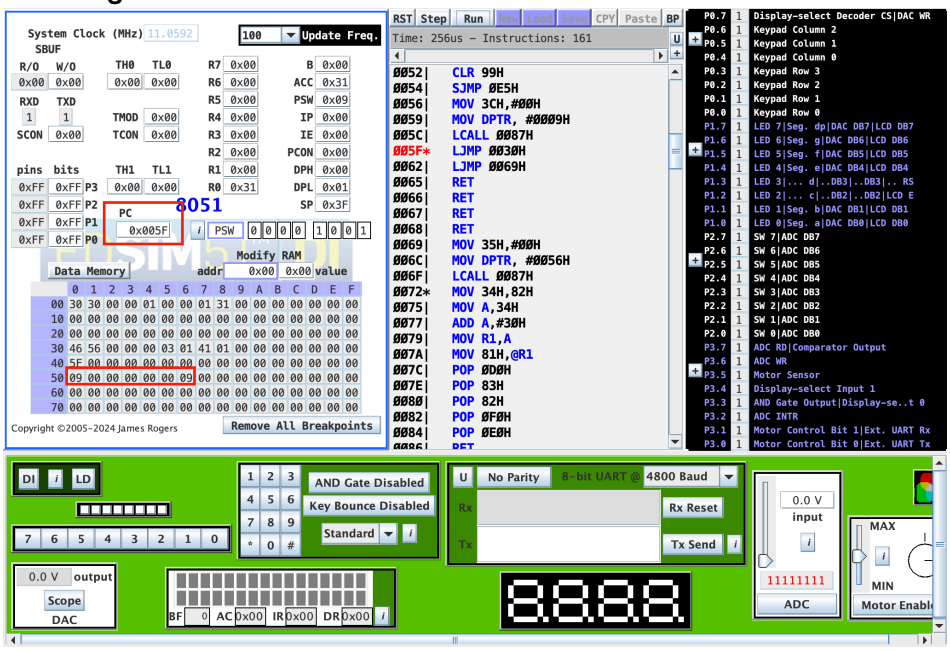

Fig. 12: Screenshot after returning from second `ThreadCreate` call

§2.2 Producer

The `Producer`'s code address is between 0x09 and 0x2E. The screenshot for the first `Producer` call is shown in **Fig. 13**.

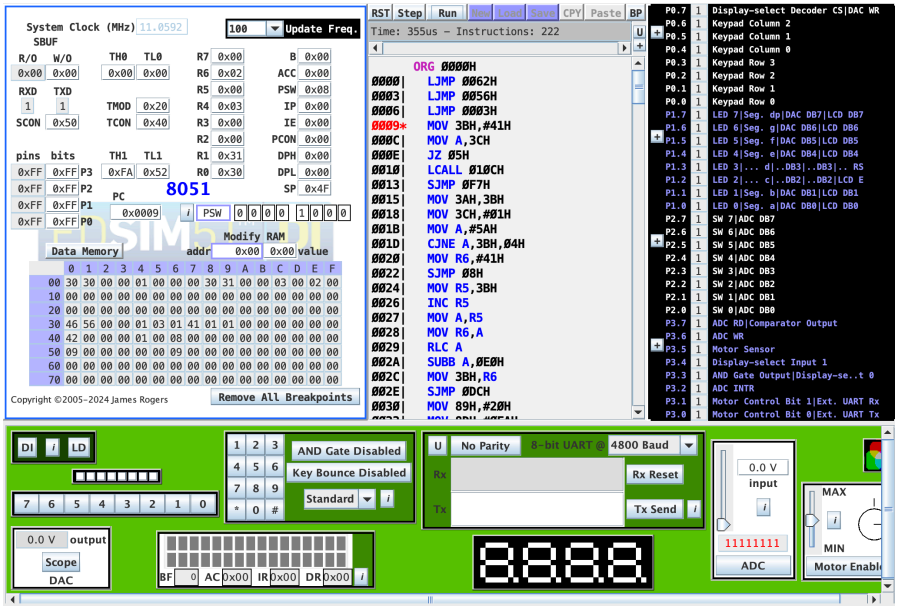

Fig. 13: Screenshot for first `Producer` call

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 infinite loop. Since the buffer is empty, the `Producer` will push the `currentChar`'s value to the `buffer` (located at $0x3A)_{(2)}$, set the variable `bufferFull` (located at 0x3C) to $1_{(3)}$, and increment the `currentChar` variable (to $0x42)_{(4)}$. The screenshot after the first `Producer` is shown in **Fig. 14** and the corresponding code is shown in **Fig. 15**.

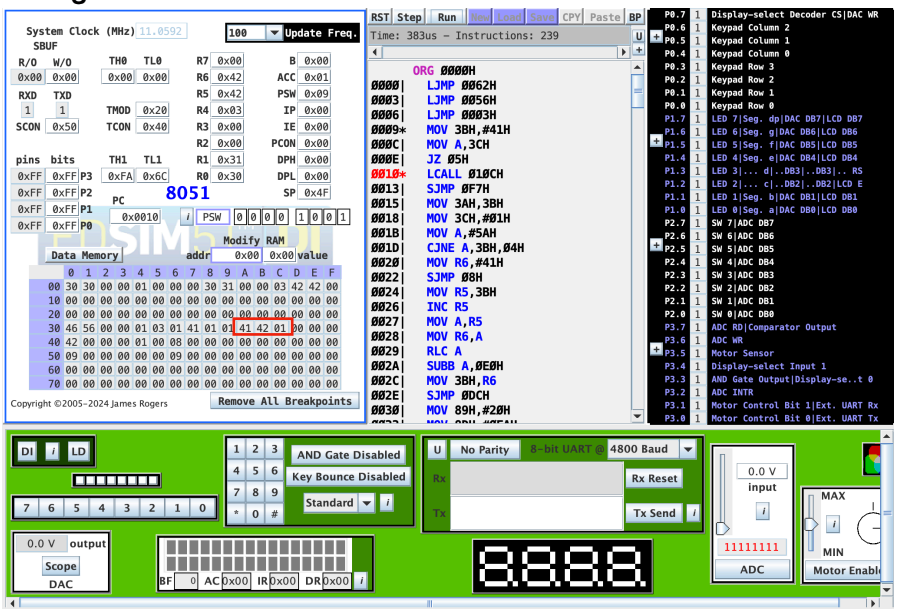

Fig. 14: Screenshot after finishing first `Producer` call

```
void Producer(void)
{
    currentChar = 'A';    (1)
    while (1)
    {
        while (bufferFull){    (5)
            ThreadYield();
        }
        buffer = currentChar;    (2)
        bufferFull = 1;    (3)
        currentChar = (currentChar == 'Z') ? 'A' : currentChar + 1;
    }    (4)
}
```

Fig. 15: Code for `Producer`

In future rounds, if the buffer isn't empty (0x3C == 1), the `Producer` will yield immediately without running any instructions$_{(5)}$. Otherwise, proceed with the same instructions as the first round in the infinite loop.

§2.3 Consumer

The `Consumer`'s code address is between 0x30 and 0x55. The screenshot for the first `Consumer` call is shown in **Fig. 16**.

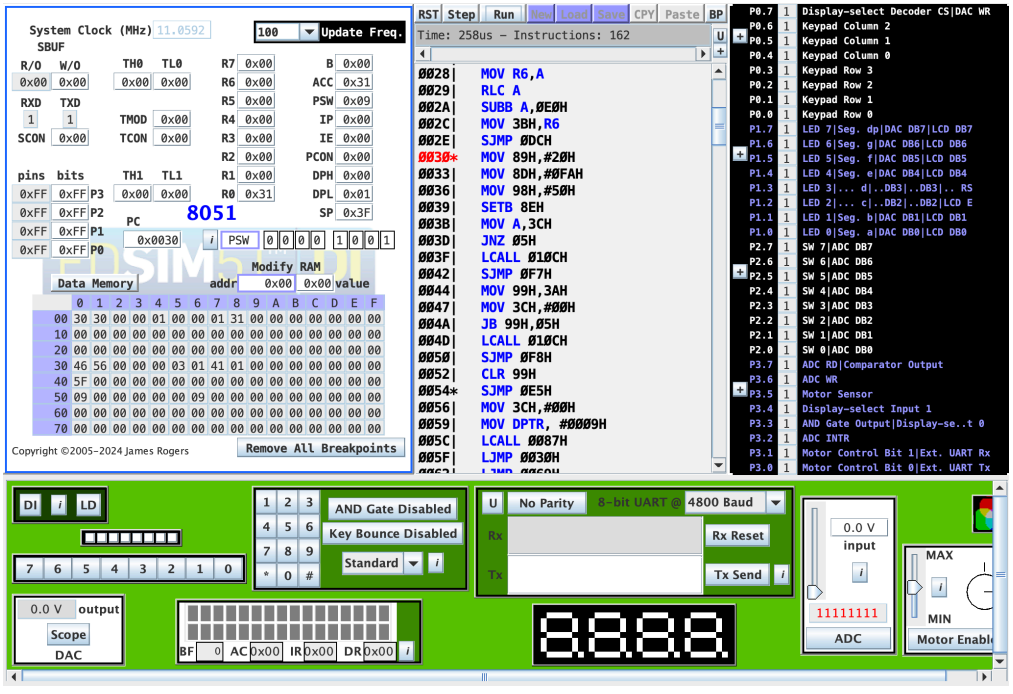

Fig. 16: Screenshot for first `Consumer` call

In the first round, the `Consumer` will initialize `Tx` for polling$_{(1)}$ and then enter an infinite loop. Since I run the `Consumer` first instead of the `Producer`, the first round will yield immediately 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` yields until the `TI` flag is pulled to 1.(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 corresponding codes (1) – (4) are shown in **Fig. 17**.

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

Fig. 17: Code for `Consumer`

The screenshot for running for a while is shown in **Fig. 18**.

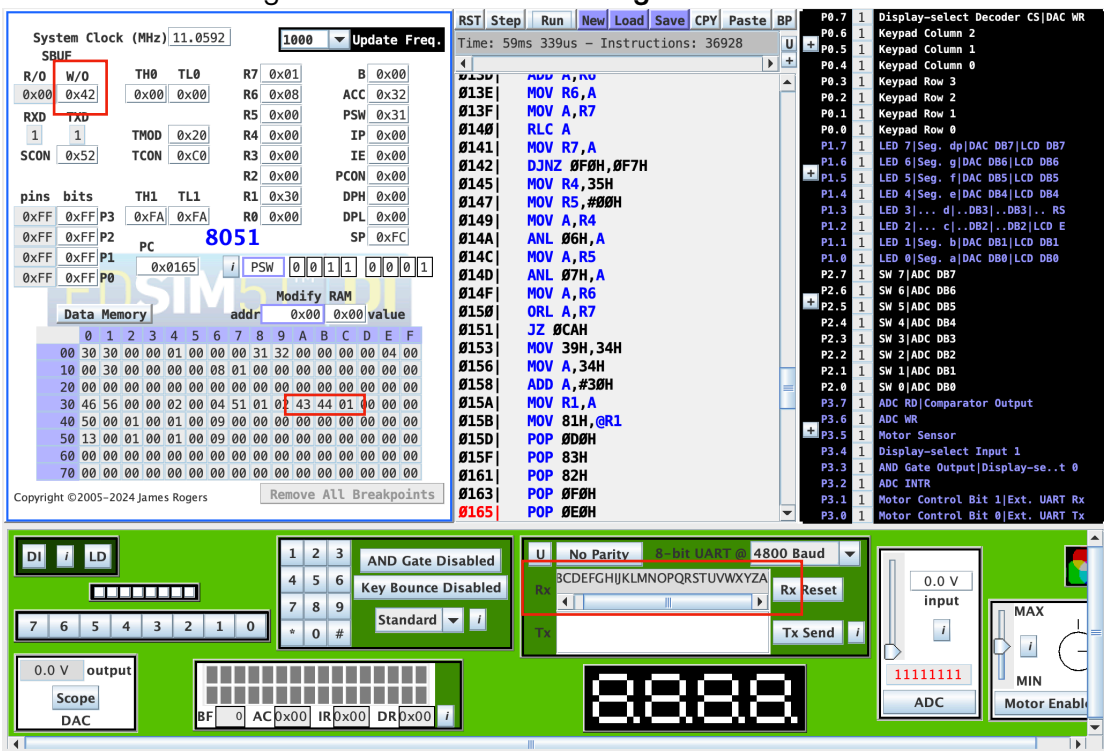

Fig. 18: Screenshot for running 36928 instructions

From `RX`, we can observe that the characters successfully reset to 'A' when the previous character is 'Z.' The `SBUF` is pending to write 'B' to `RX`, and the next character pending to be sent to the `Consumer` is 'C.' The overall result is correct.