Code explanation

Testcoop.c:

```
__data __at (0x3A) char isEmpty;
__data __at (0x3B) char buffer;
__data __at (0x3C) char nextChar;
```

Global Variables:

These global variables manage the shared state between the producer and consumer:

- **isEmpty:** Indicates whether the buffer is empty (1 for empty, 0 otherwise).
- **buffer**: Holds the current character being shared between the producer and consumer.
- **nextChar:** Tracks the next character that producer will produce (cycles through 'A' to 'Z').

```
void Producer(void)
{
    /*
     * [TODO]
     * initialize producer data structure, and then enter
     * an infinite loop (does not return)
     */
    nextChar = 'A';
    while (1)
    {
        /* [TODO]
         * wait for the buffer to be available,
         * and then write the new data into the buffer */
        while (!isEmpty) {
            ThreadYield();  // Wait until buffer is empty
        }
        buffer = nextChar;
        isEmpty = !isEmpty;
        nextChar = (nextChar == 'Z') ? 'A' : nextChar + 1;  // Cycle through 'A'-'Z'
        ThreadYield();  // Yield control
    }
}
```

The Producer generates characters in a loop:

- Waits for the buffer to become empty. (the while loop)
- Writes the current character (nextChar) into the buffer.
- Marks the buffer as full by toggling isEmpty. (isEmpty = !isEmpty)
- Updates nextChar to the next character in the sequence ('A'-'Z').
- Yields control to allow other threads to execute.

```c
void Consumer(void)
{
    /*
     * [TODO]
     * initialize Tx for polling
     */
    SCON = 0x50;  // Serial mode 1 (8-bit UART) and enable receiver
    TMOD = 0x20; // Timer 1 in mode 2 (8-bit auto-reload)
    TH1 = (char)-6;   // Load value for 9600 baud rate (assuming 11.0592 MHz clock)
    TR1 = 1;      // Start Timer 1
    TI = 1;
    while (1)
    {
        /*
         * [TODO]
         * wait for new data from producer
         */
        while (isEmpty) {
            ThreadYield();  // Wait for new data
        }
        /*
         * [TODO]
         * write data to serial port Tx,
         * poll for Tx to finish writing (TI),
         * then clear the flag
         */
        while (!TI);  // Wait for transmission to complete
        SBUF = buffer;  // Send data to serial port
        TI = 0;  // Clear transmission flag
        isEmpty = !isEmpty;
        ThreadYield();
    }
}
```

The Consumer continuously retrieves characters from the buffer:

- Initialize Tx for polling and configures the serial port for communication at 4800 baud.
- Waits for data in the buffer (while loop, isEmpty == 0).
- Wait for transmission to complete
- Transmits the character in the buffer via the serial port.
- Marks the buffer as empty by toggling isEmpty. (isEmpty = !isEmpty)
- Yields control to other threads.

```c
void main(void)
{
    /*
     * [TODO]
     * initialize globals
     */
    isEmpty = 1;  // Initialize buffer as empty
    buffer = ' ';
    /*
     * [TODO]
     * set up Producer and Consumer.
     * Because both are infinite loops, there is no loop
     * in this function and no return.
     */
    ThreadCreate(Producer);
    Consumer();
}
```

The main function:

- Initializes the global variables (isEmpty and buffer).
- Creates a separate thread for the Producer using ThreadCreate.
- Runs the Consumer in the main thread.

cooperative.c:

Global Variables Declaration:

savedSP[MAXTHREADS]:

- This array holds the saved stack pointers for each thread. Each thread will have a unique stack pointer value saved here.

threadBitmap

- A bitmap where each bit represents the state of a thread (active/inactive). If a bit is 1, the corresponding thread is active.

bitmapCheck

- A temporary variable used to check the availability of thread slots in the bitmap.

newThreadID

- Holds the ID of a newly created thread.

currThread

- The ID of the currently running thread.

newThreadStack

- Stores the starting stack pointer for a newly created thread.

prevSP

- Temporarily stores the previous value of the stack pointer during thread creation.

RegBank

- Represents the register bank to use for the newly created thread. Each thread is associated with a unique register bank for its local registers.

For savestate and restorestate's implementation are explained in the code (comments):

```
#define SAVESTATE \
  { __asm \
    push ACC          /* Save accumulator register */ \
    push B            /* Save B register */ \
    push DPL          /* Save Data Pointer Low byte */ \
    push DPH          /* Save Data Pointer High byte */ \
    push PSW          /* Save Processor Status Word */ \
  __endasm; \
  } \
  { __asm \
    /* Save the current stack pointer (SP) for the current thread */ \
    mov a, _currThread   /* Load current thread index into A */ \
    add a, #_savedSP     /* Add address of savedSP to A */ \
    mov r0, a            /* Store the address of savedSP[currThread] in r0 */ \
    mov @r0, _SP         /* Store current SP into savedSP[currThread]; savedSP[currThread] = _SP */ \
  __endasm; \
  }
```

```
#define RESTORESTATE \
  { __asm \
    /* Load the stack pointer (SP) for the current thread */ \
    mov a, _currThread   /* Load current thread index into A */ \
    add a, #_savedSP     /* Add address of savedSP to A */ \
    mov r1, a            /* Store result in r1 */ \
    mov _SP, @r1         /* Load savedSP[currThread] into SP; _SP = savedSP[currThread]; */ \
  __endasm; \
  } \
  { __asm \
    /* Restore the saved registers */ \
    pop PSW              /* Restore Processor Status Word */ \
    pop DPH              /* Restore Data Pointer High byte */ \
    pop DPL              /* Restore Data Pointer Low byte */ \
    pop B                /* Restore B register */ \
    pop ACC              /* Restore accumulator register */ \
  __endasm; \
  }
```

Open a file

For Bootstrap function (I follow TODO):

- Thread Initialization: Initializes the thread bitmap to 0 (no threads active, 0x00).
- Create main Thread: Calls ThreadCreate to create a thread for the main() function and sets currThread to its ID.
- Restore Context: Restores the context for the main thread using RESTORESTATE and starts executing main().

For ThreadCreate (I follow TODO and already put the comments, but I would like to explain a bit how I implement those TODO):

- First, I check whether we have reach the max #thread or not by checking whether my threadbitmap is equal to 15 or not (15 = 1111 in binary, since each bit indicate each thread and if all of them are all '1' then that means we have reach the max #thread)

- Then I run for loop to update the bit mask (threadBitmap) by checking the next available thread slot by find the bit in ‘threadBitmap that has a value of ‘0’, then change that bit to ‘1’ later on. This will update the bit mask and ensure we have swap the thread.
- Since each stack is 16 bytes apart, I calculate the starting stack location for new thread by add the value of 3, shift left the thread ID by 4 bits and OR with 0xF, by doing this I can ensure that the stack pointer starts at safe spot.
- Then I Save the current stack pointer and set up the new thread's stack using the temp stack pointer.
- Then I push the return address (fp) for the new thread onto the stack by pushing DPL and DPH.
- Then initialize the registers to 0 and push them four times (ACC, B, DPL, DPH)
- Then I push PSW by calculating the correct memory bank for each thread, using Regbank
- Then I write the current stack pointer to the saved stack, r estore the original stack pointer, andr eturn the newly created thread ID

ThreadYield:

- **Save Current State:** Saves the context of the current thread using SAVESTATE.
- **Round-Robin Scheduling:** Iterates through the threads in a round-robin fashion to find the next active thread. Updates currThread to the ID of the next active thread.
- **Restore Next Thread's State:** Restores the state of the next thread using RESTORESTATE.

ThreadExit:

- **Mark Thread Inactive:** Clears the corresponding bit in threadBitmap for the current thread.
- **Find Next Thread:** Searches for the next valid thread using the bitmap. This ensures the system does not halt as long as at least one thread is active.
- **Restore Next Thread's State:** Uses RESTORESTATE to switch to the next thread.

Screenshots for compilation

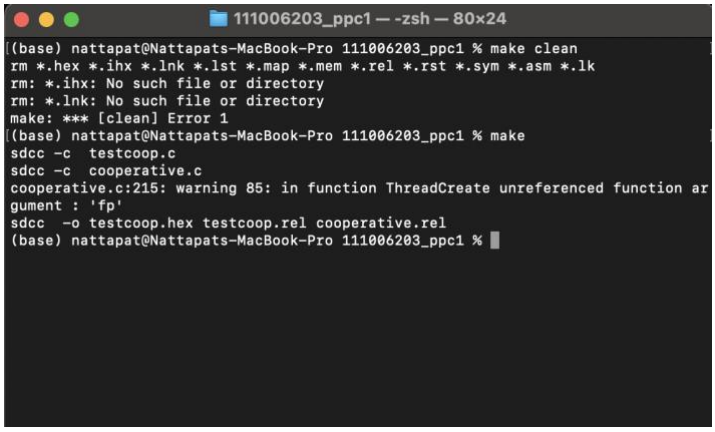

Figure 1 Screenshot for compilation

Screenshots and explanation

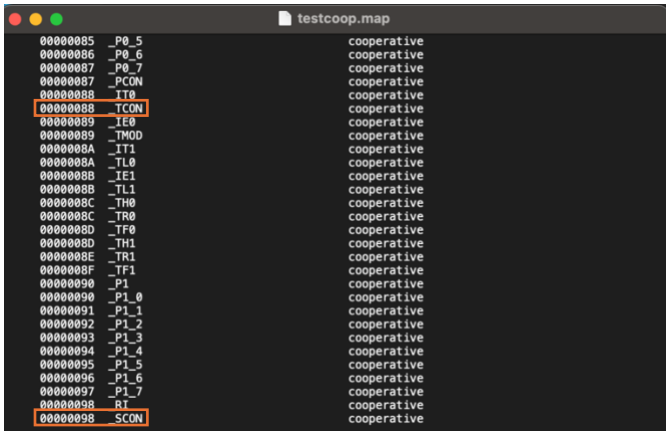

Figure 2 testcoop.map (will be used for explanation in the following section)

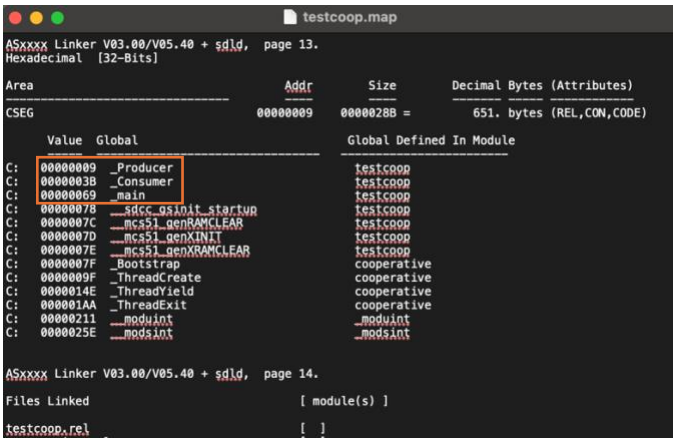

Figure 3 testcoop.map (will be used for explanation in the following section)

Before each ThreadCreate call

- ThreadCreate(main), Address of main is 0x69, as shown in figure 3

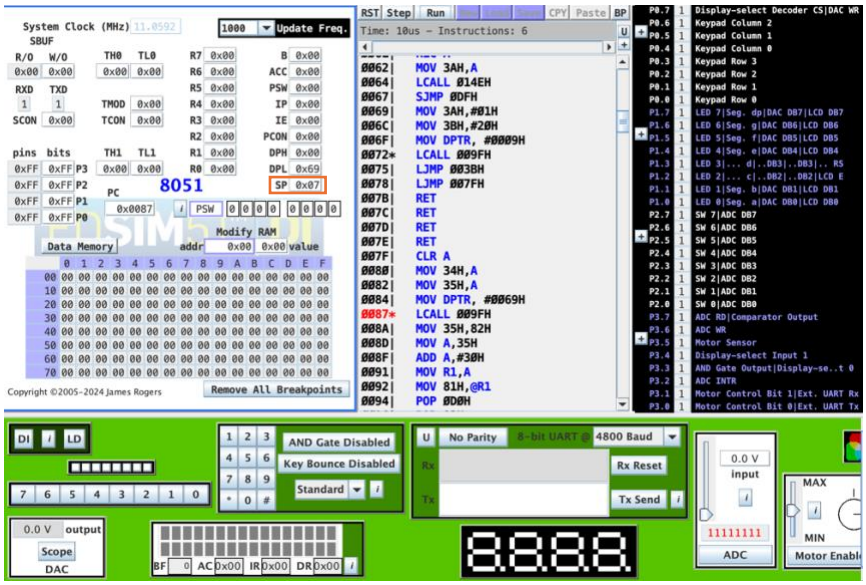

Figure 4 ThreadCreate(main)

When perform LCALL, the DPTR with the value 0x69 (which is the address of the main) the return address is pushed into the stack 2 bytes which leads into the changes of SP from 0x07 -> 0x09.

- ThreadCreate(Producer), Address of producer is 0x09, as shown in figure 3

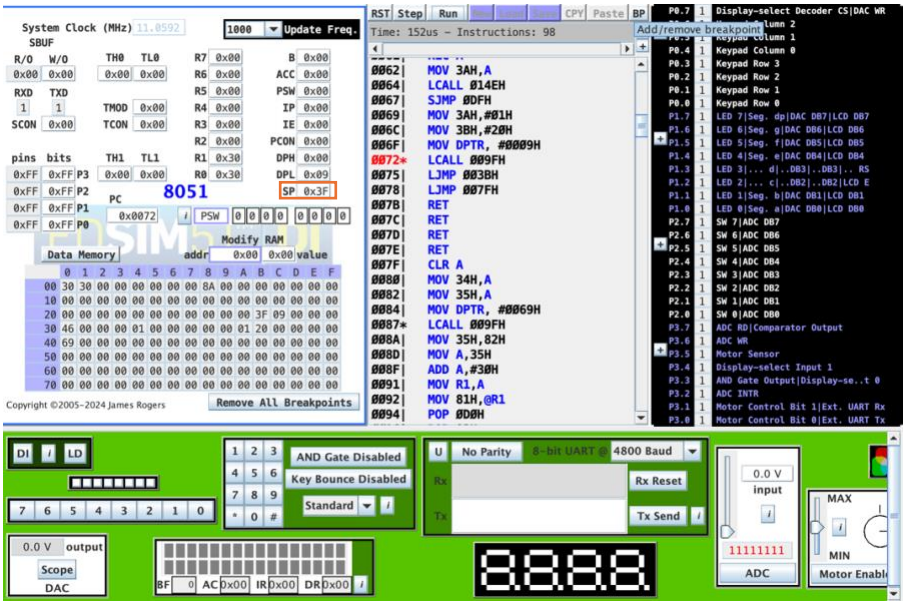

Figure 5 ThreadCreate(Producer)

When perform LCALL, the DPTR with the value 0x09 (which is the address of the producer), the return address is pushed into the stack 2 bytes which leads into the changes of SP from 0x3F -> 0x41.

When the producer is running

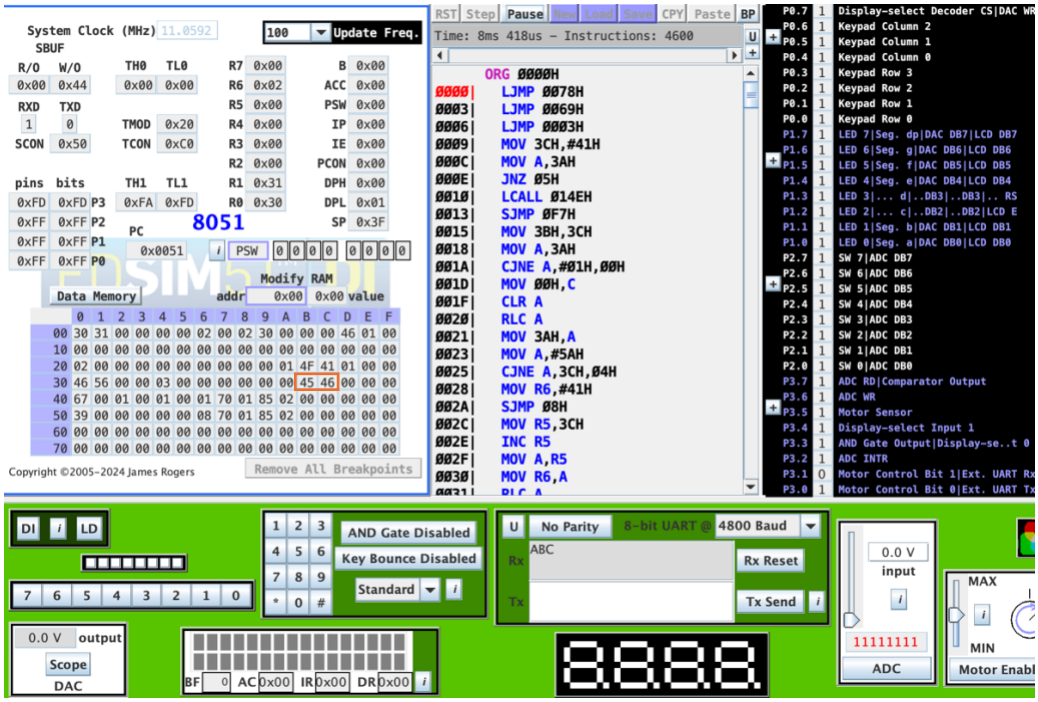

Figure 6 Producer (running)

As I have set *nextChar* (indicate which character will be transmit into SBUF) and *buffer* (represent shared buffer) in the address of 0x3B and 0x3C, respectively. Since producer is the one who generates a character and keep it in the shared buffer, therefore if the values in those addresses change that means the producer is running. (And it's changing over time)

When the consumer is running

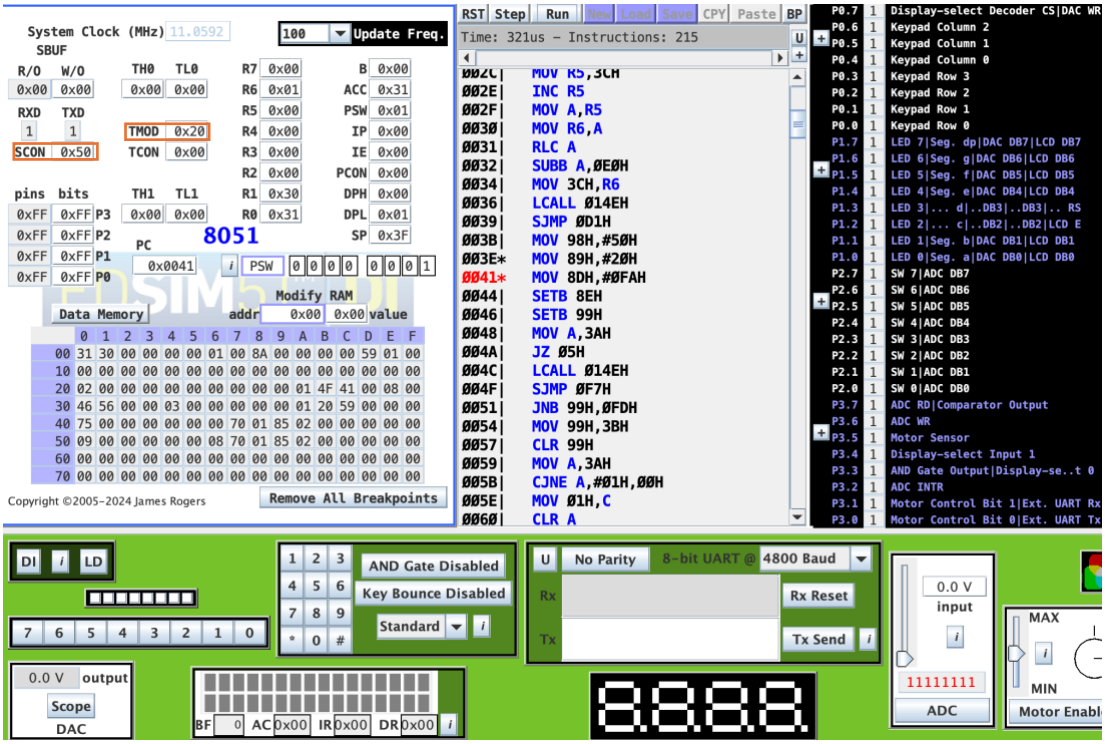

Figure 7 Consumer (running)

When the consumer is running, the value of TMOD and SCON will be changed to what we set in the code, which is 0x20 and 0x50, respectively.