107062381 楊孝偉

Checkpoint 3 Report

Testpreempt.c

In this file, I will two variables from the previous checkpoints: buffer and token. Buffer is used for the shared buffer between the producer and consumer. Token is used to generate characters from A to Z for the producer. There are also other additional variables : head and tail to keep track of the circular queue, mutex, full and empty for the semaphore variables.

In the main function, Token is initialized to A. Main will spawn consumer thread, while main itself calls producer. This is done to not waste main's thread, more economical. We also need to create three new semaphores : full, mutex and empty. Set full to 0, mutex to 1, and empty to 3 (since we have 3 deep buffer). I followed the materials taught during the lectures. Then we set head and tail to 0 to indicate that both of them point to the initial position of the 3 deep buffer.

In the producer function, before it enters the infinite loop, I initialized the token to 'A' first, to make sure that token is really initialized to 'A'. After it enters the loop, it will perform semaphorewaitbody for empty and mutex to check whether there is still an empty space in the buffer and whether it's the producer turn. Buffer will take the value in token, then update the value in token. When buffer is taking the value of token, it should be in critical section since SDCC suggests that you can surround the code fragment where they access the shared variable, in this case it's buffer, to ensure that the shared variables are accessed atomically. Updating the tail value should also be inside the critical section since if it's not, then interrupt might interrupt it before tails could update its index position and this might cause an unwanted error. Then signal the mutex to indicate that it's the consumer turn and signal full to indicate that buffer has been fully filled.

In the consumer function, before it enters the infinite loop, we initialize Tx for polling. After it enters the loop, it will semaphorewaitbody the full and mutex semaphores to check whether the data in buffer have been consumed and it's the consumer turn. SBUF(reading or writing register) will take the value in buffer, then we update the value of head, then we signal the mutex and empty semaphores for the producer, then we check if Tx is busy (serial port hasn't finished writing it yet), then we keep polling again. Finally, we reset It to 0 again.

Preemptive.c

In this file, I use quite a lot of variables, I will explain some later as I explain the functions. Sp[4] is used for the saved stack pointers. Cur_thread is for the current thread ID. Bitmap is used to determine which thread ID is a valid thread.

First, we define SAVESTATE. It is a C macro for saving the context of the current thread and it is written in inlined assembly. First we push the ACC, B register, DPTR, and PSW onto stack. Then we save the stack pointer for the current thread into the saved stack pointers array as indexed by the current thread ID.

Next, we define a C macro for restoring the context of the current thread by basically doing reverse the operation of SAVESTATE.

107062381 楊孝偉

Bootstrap is the start-up code to set up and run the first thread. First we initialize the bitmap to 0000 (which is 0 in decimal value), which means that all of the threads are available for use. Then we create a thread for main and set current thread to this thread ID and restore its context with RESTORESTATE so that it starts running the main function. Stack was set up by the threadcreate(main). Stack 0 now has the return address of main.

Next is the thread create function to create a thread data structure so it is ready to be restored. First we check to see whether we have not reached the maximum number of threads. Maximum number of threads is reached if the bitmap value is equal to 15 (which is 1111 in binary value), because we will set the bit value to 1 if a thread is used. If so, we will return -1 which is not a valid thread ID. Otherwise, we will find a thread ID that is not in use and grab it. I did this by first initializing a variable called mark to 1 (this means that mark will have the value of 0001 in binary initially). Then it will enter the while loop. First it will look for which thread is available by keep shifting the mark to the left by 1 bit position. If there is an available thread, then bitmap & mark value will be 0, then we simply set the new thread accordingly. Then we update the bitmap value to indicate that the selected thread is now used by XOR-ing the bitmap with mark. Then we calculate the starting stack location for new thread. We will check if new thread is 0, 1, 2 or 3. If new thread is 0, we set the address to 0x3F as the hardware stack in 8051 is pre increment. If new thread is 1, 2 and 3, we set the address to 0x4F, 0x5F, 0x6F respectively. Then we save the current SP in a temporary and set SP to the starting location for the new thread, which is address basically. Then we push the return address of fp onto stack by pushing DPL and DPH as in SDCC convention, 2-byte ptr is passed in DPTR, but push instruction can only push it as two separate registers, DPL and DPH. Then we initialize the registers to 0. So we assign ACC to 0 by ANL, an instruction to perform a bitwise logical AND operation between the specified operands and stores the result in the destination operand (ACC), and push it four times for ACC, B, DPL and DPH. Then we need to push PSW registers. RS1 and RS0 is used for register bank selection. Therefore, we can fill RS1 and RS0 by shifting the new thread ID by 3 as RS1 and RS0 is 4 and 3 bits away from the least significant bit. Then we push the PSW. Then we can write the current stack pointer to the saved stack pointer array, SP will take the previously saved SP, and return this newly created thread ID.

Thread Yield function is called by a running thread to yield control to the other thread. First it will find the next thread that can run and set the current thread ID to it. Then if the bitwise AND value of bitmap and the current thread ID is not equal to 0, we break from the while loop. This means that thread can run, so we break from the loop. If the bitwise AND value of that thread ID and bitmask is equal to 0, this means that thread ID is still not active, then the while loop will continue to find the next thread that can run. It is guaranteed that at least one thread is active, so this loop will always terminate.

I did not fill the Thread Exit function as this function is not used anywhere.

myTimer0Handler function to be the ISR for Timer0 to serve the purpose of preemption. I used the code from ThreadYield, added the RETI assembly instruction, saved the content of R0 to R7 to registers that have been saved by RESTORESTATE. This is done to preserve the value of R0 to R7.

Preemptive.h

I added a semaphore API to this file.

The SemaphoreCreate function the semaphore s to the value n.

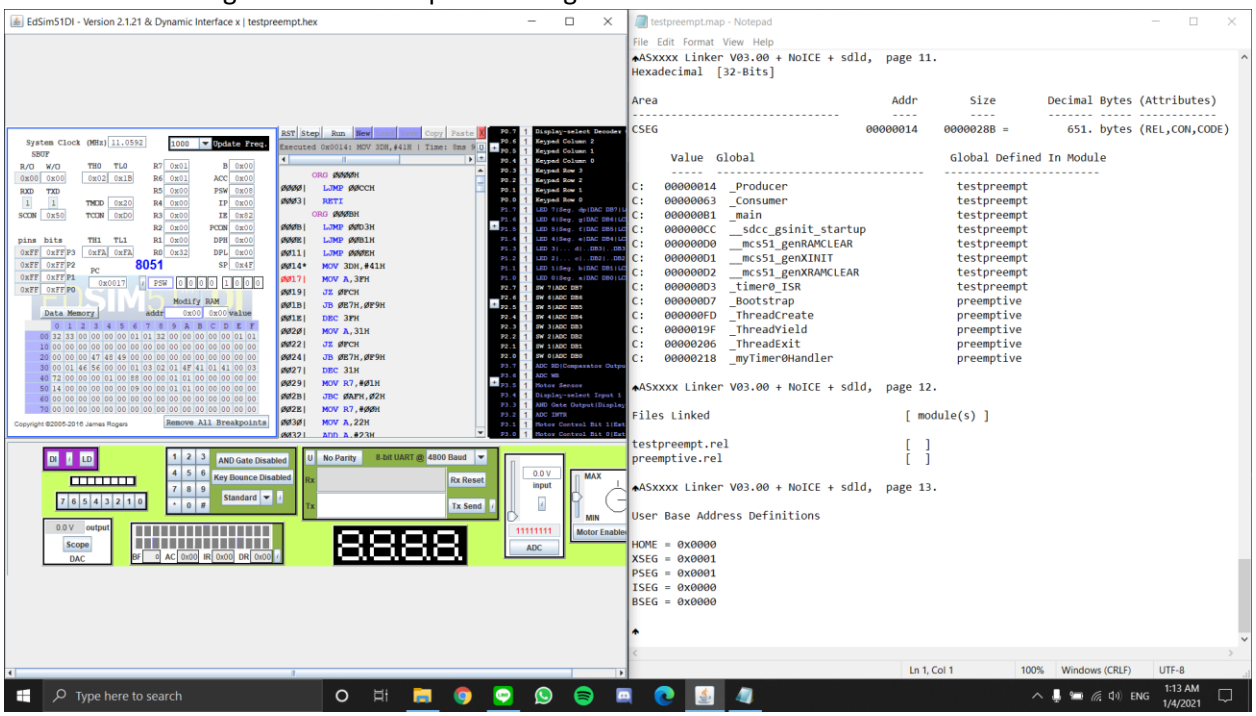
107062381 楊孝偉

The SemaphoreSignal only increment the value in the semaphore variable.

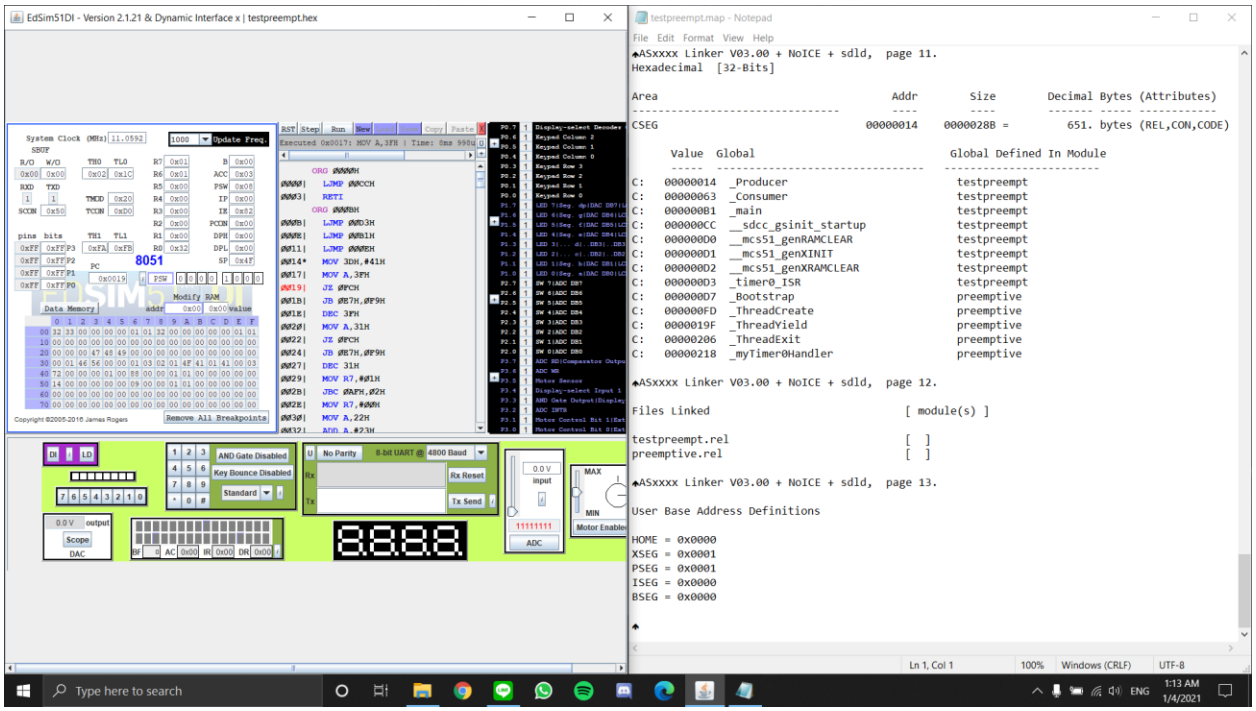
The SemaphoreWait basically reads the value of S into ACC then checks if ACC is <= 0 o not.

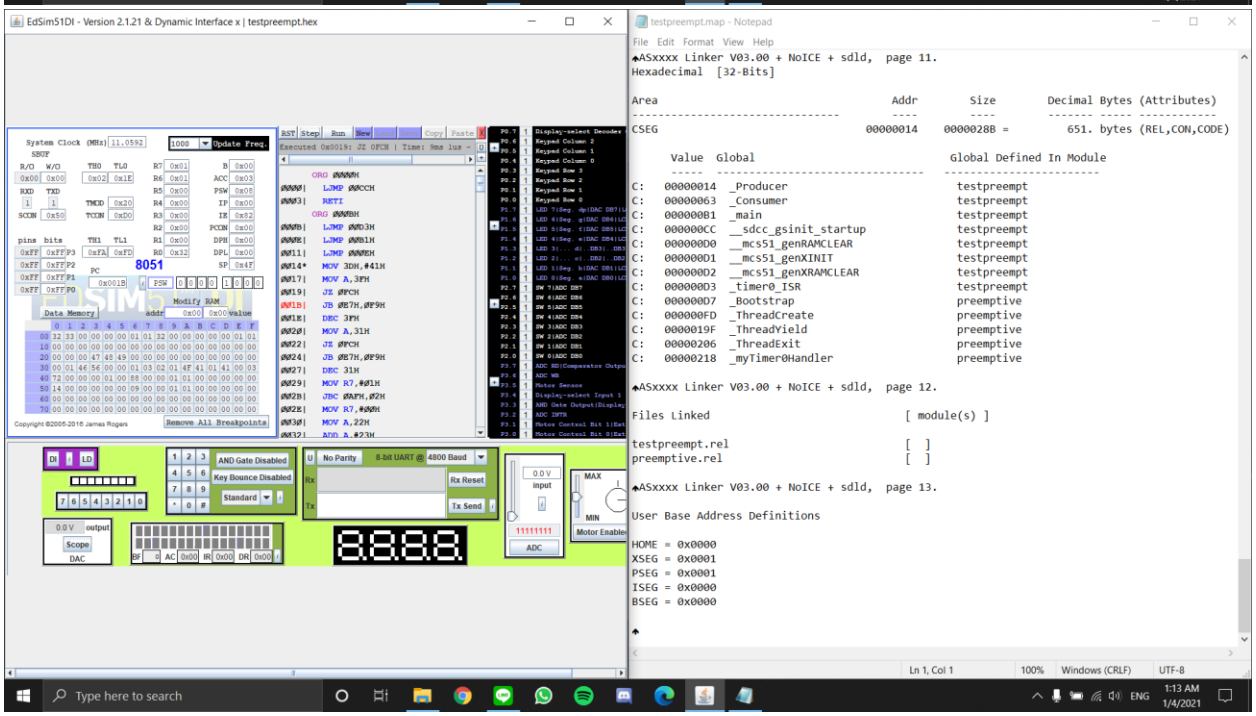
Breakpoints

1. Producer is running and show semaphore changes

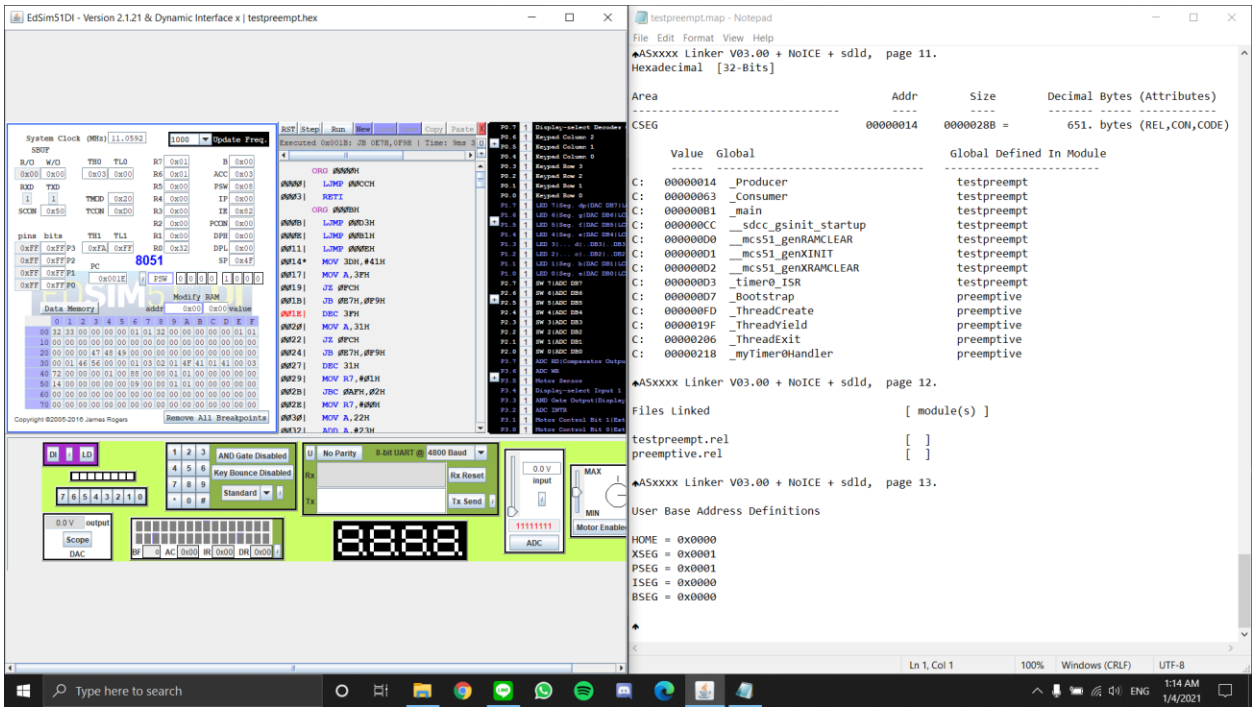


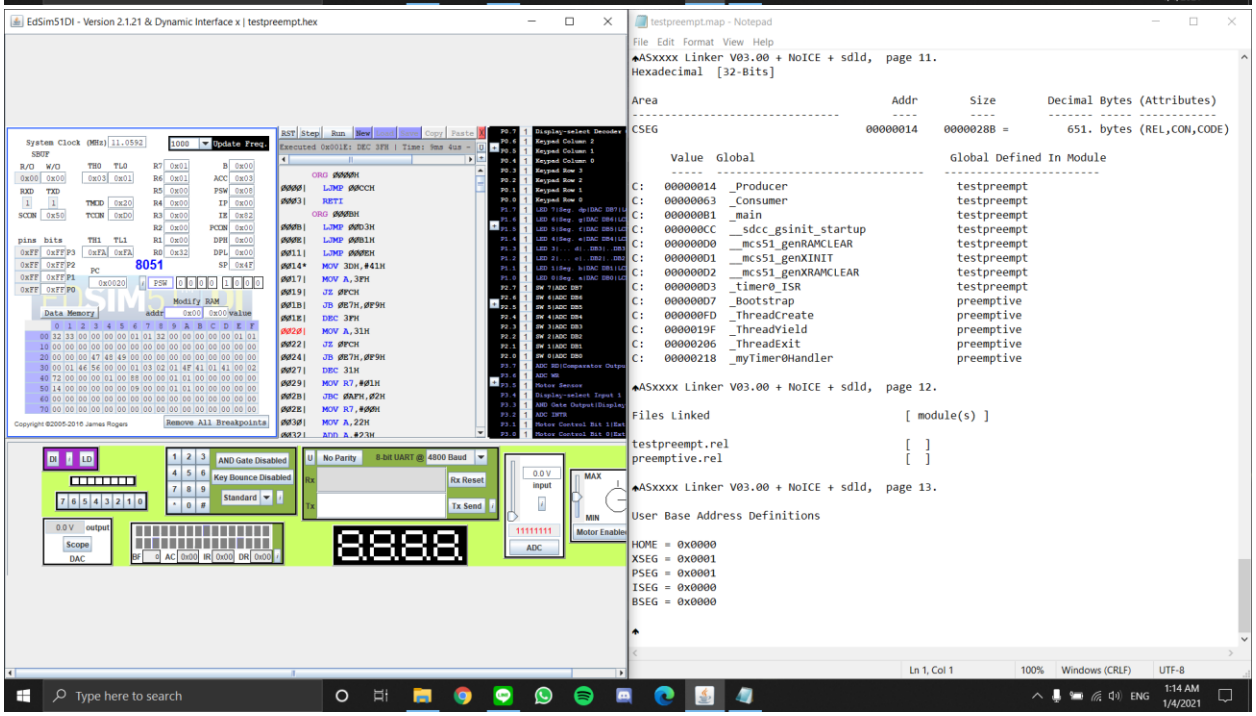
107062381 楊孝偉



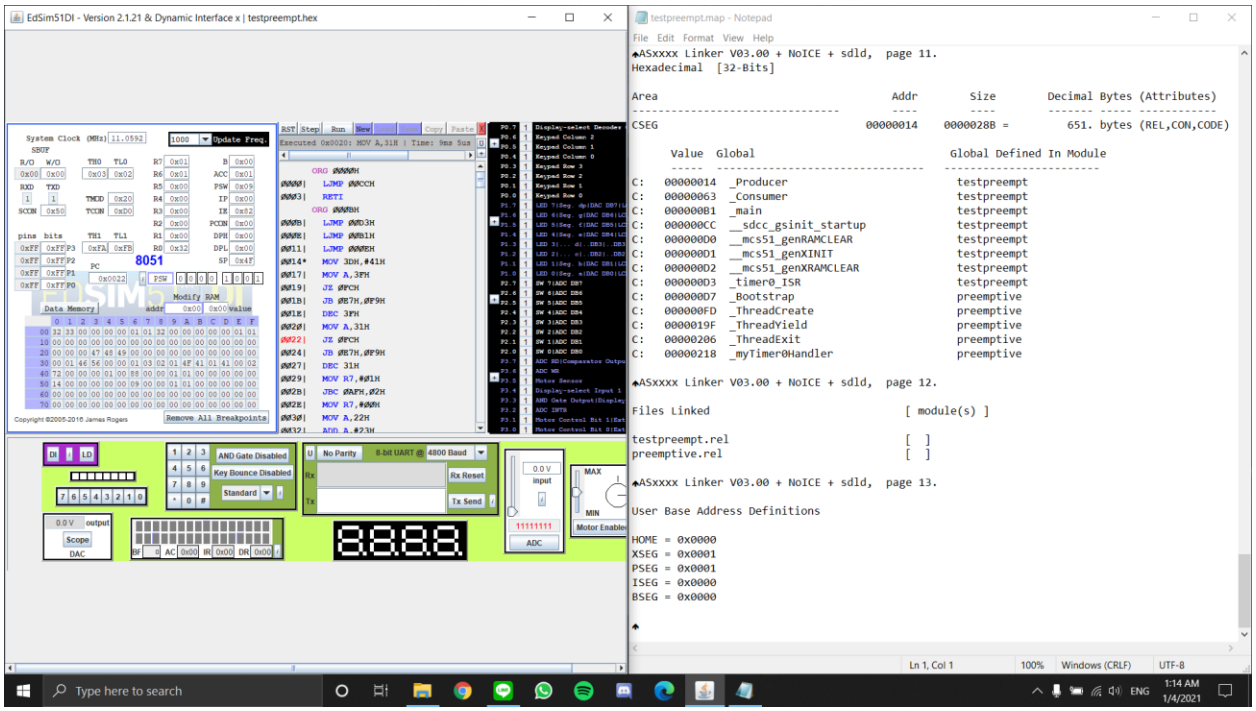


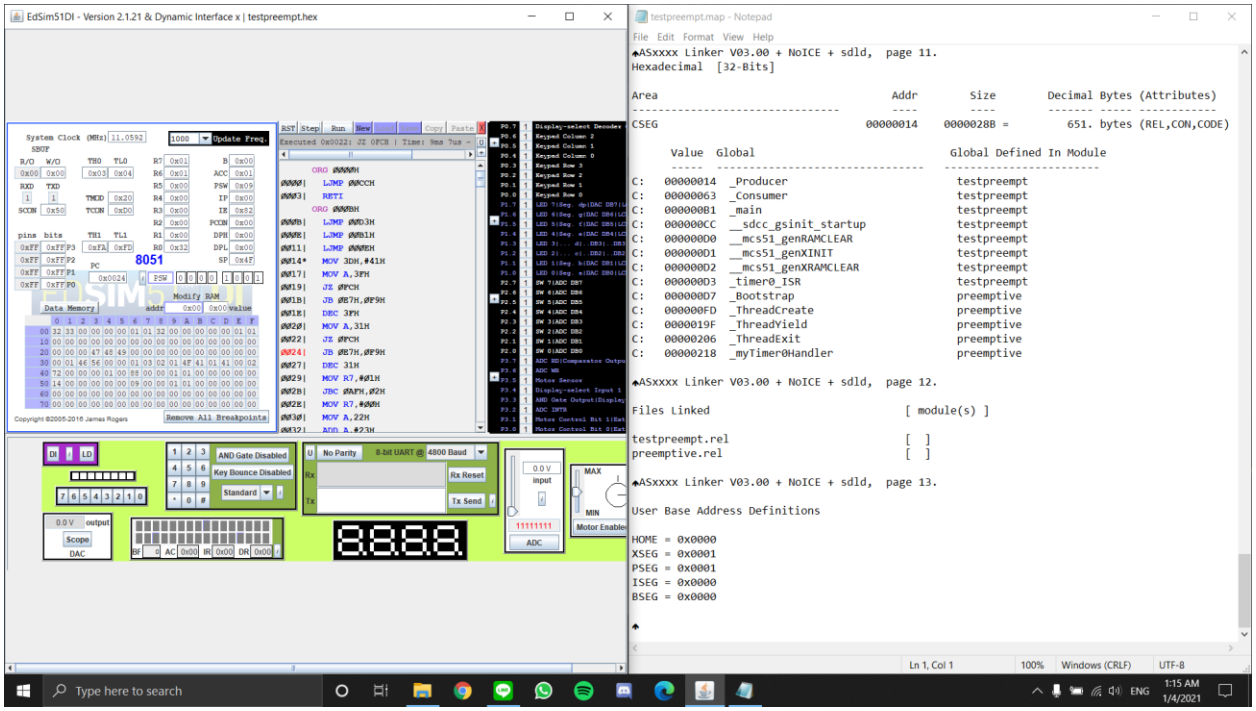
107062381 楊孝偉



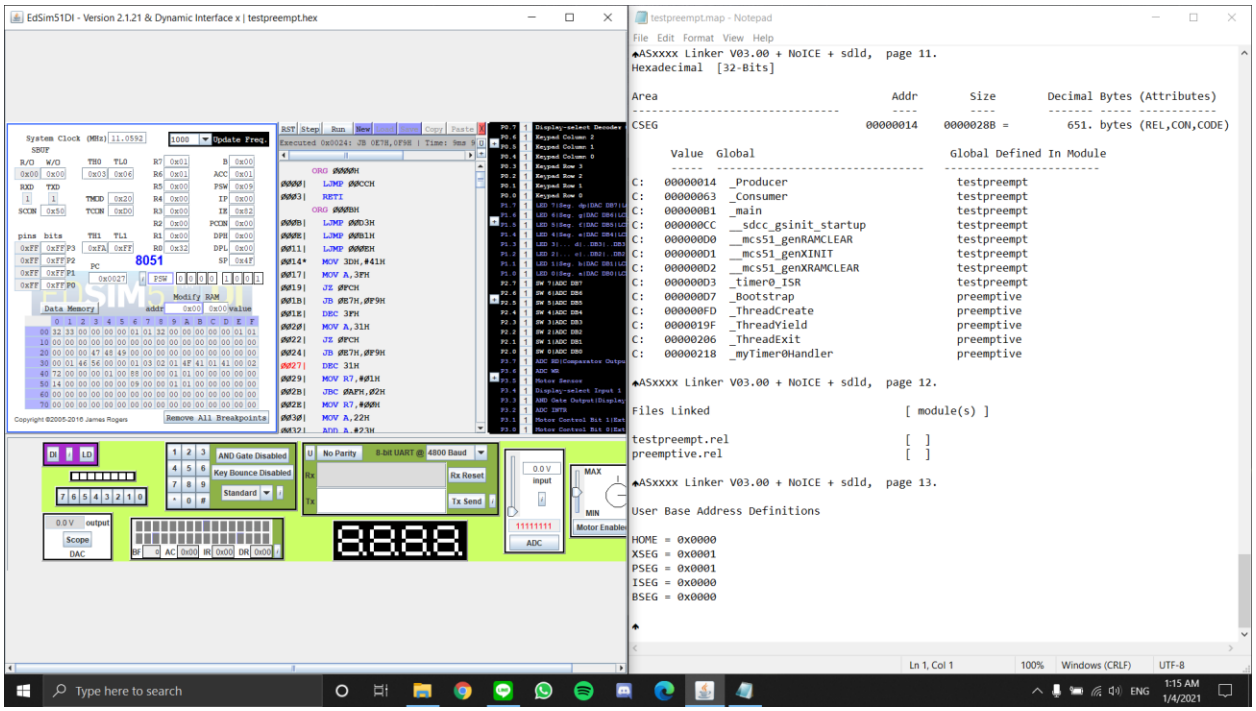


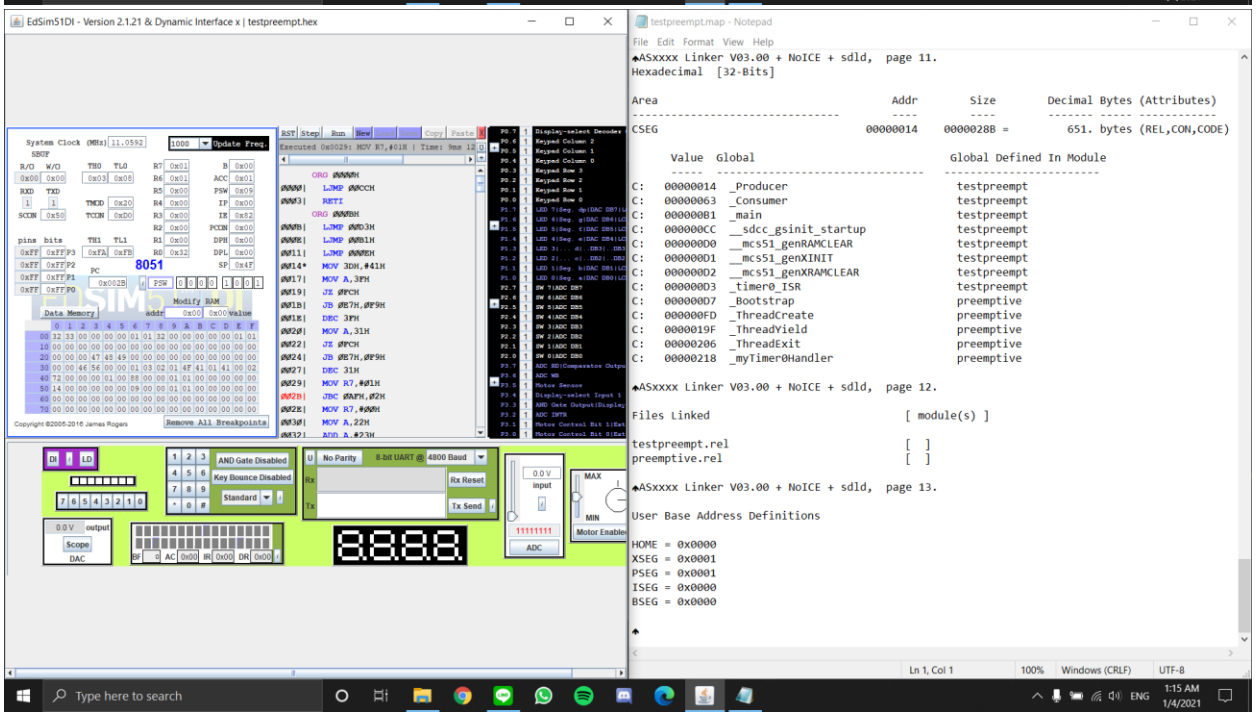
107062381 楊孝偉



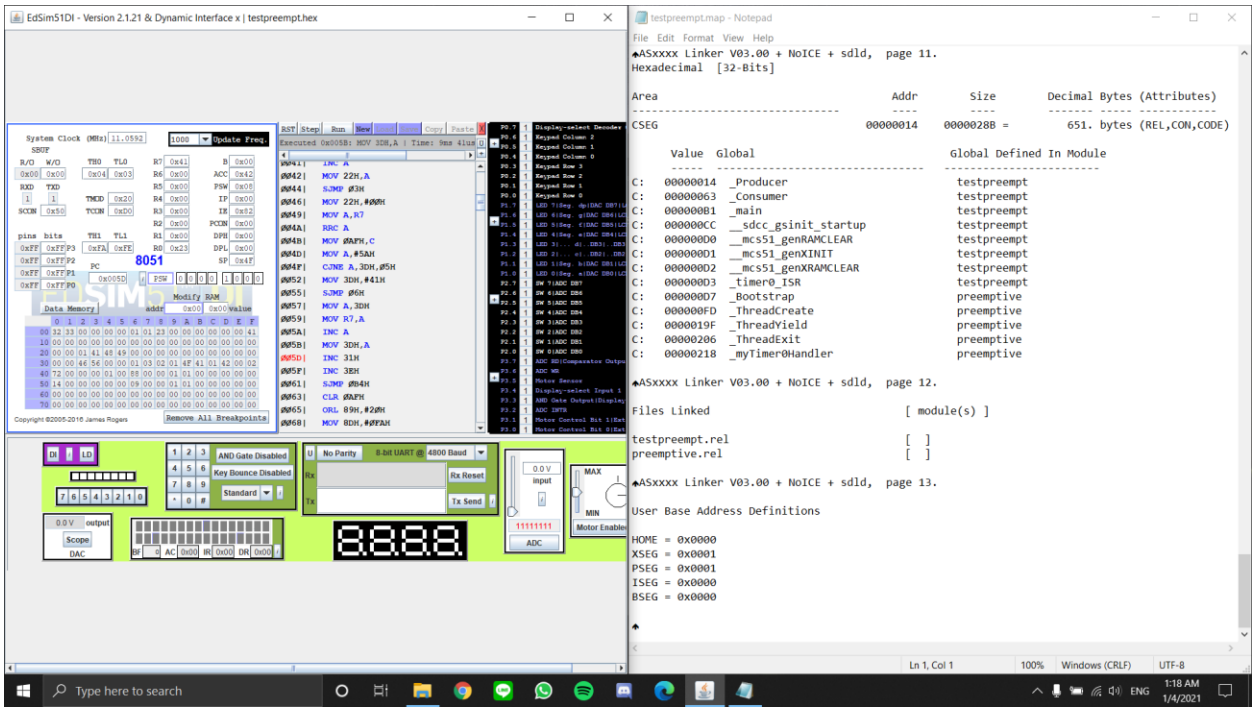


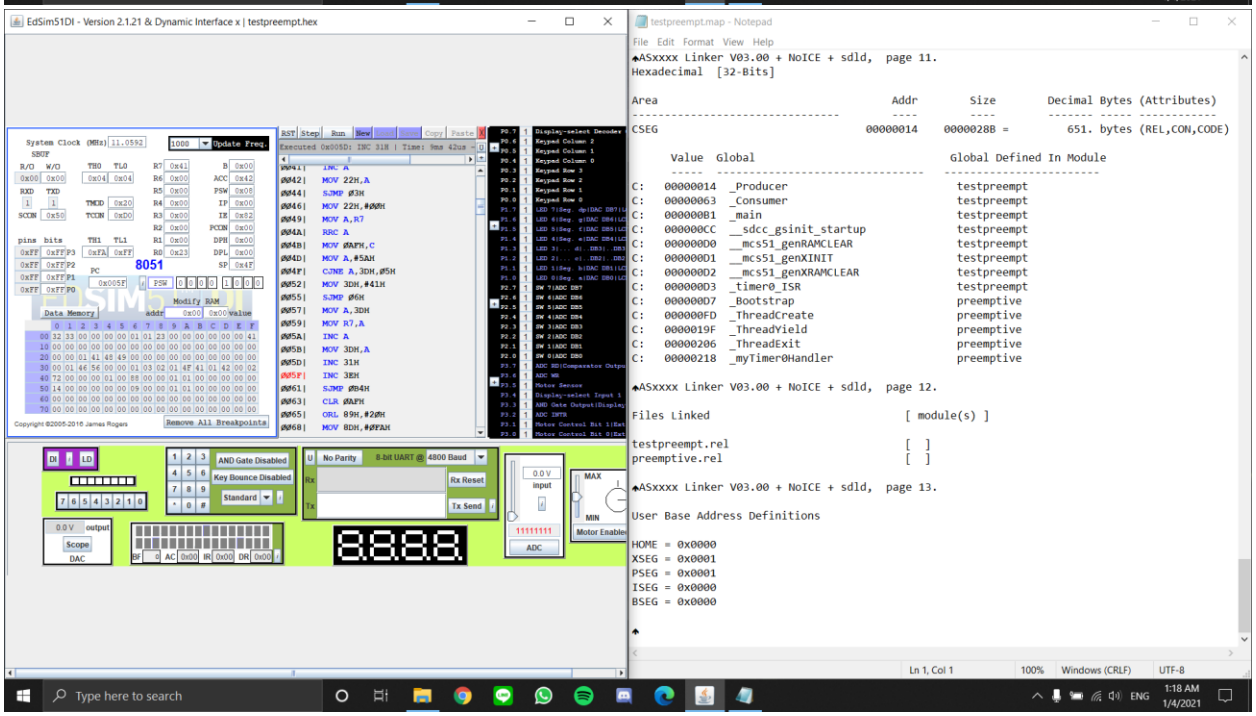
107062381 楊孝偉





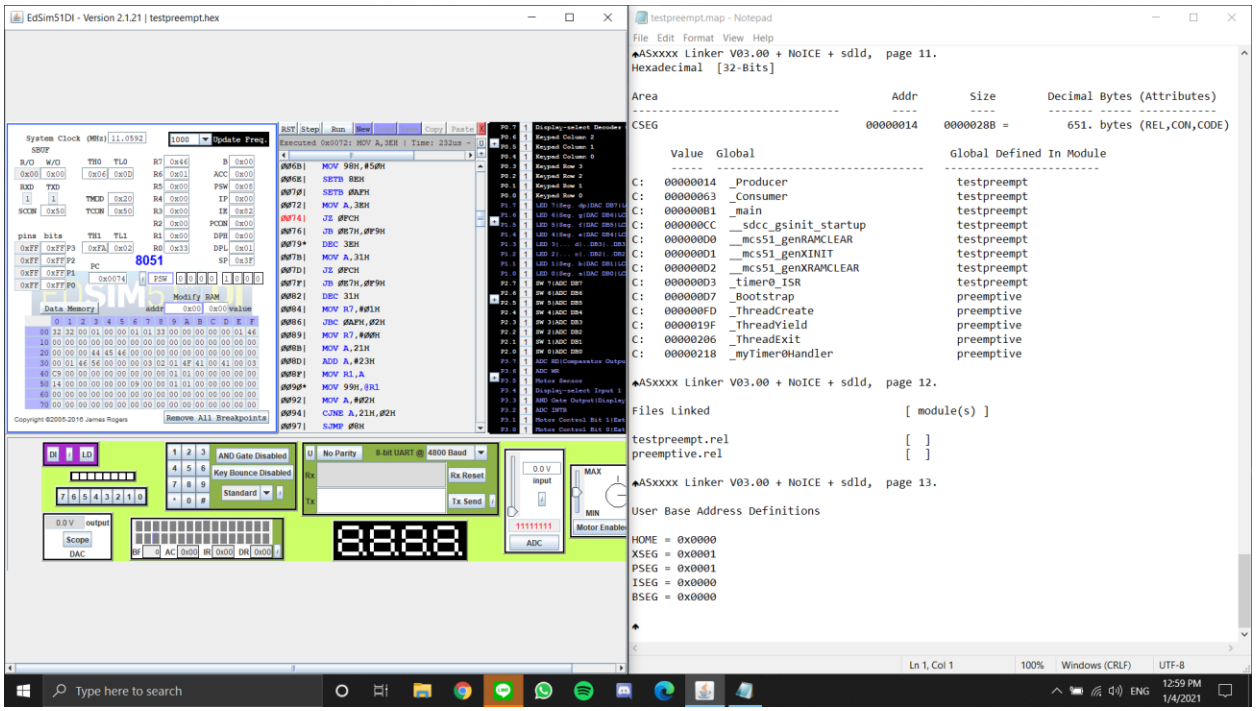
107062381 楊孝偉

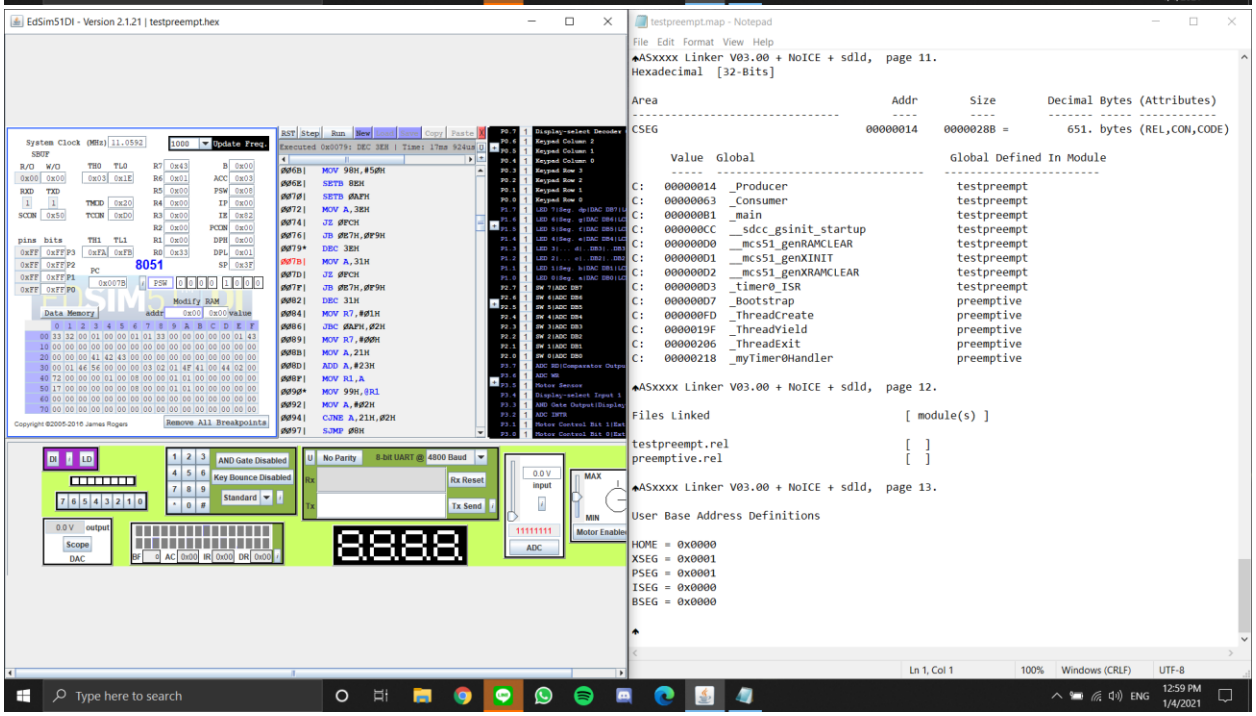




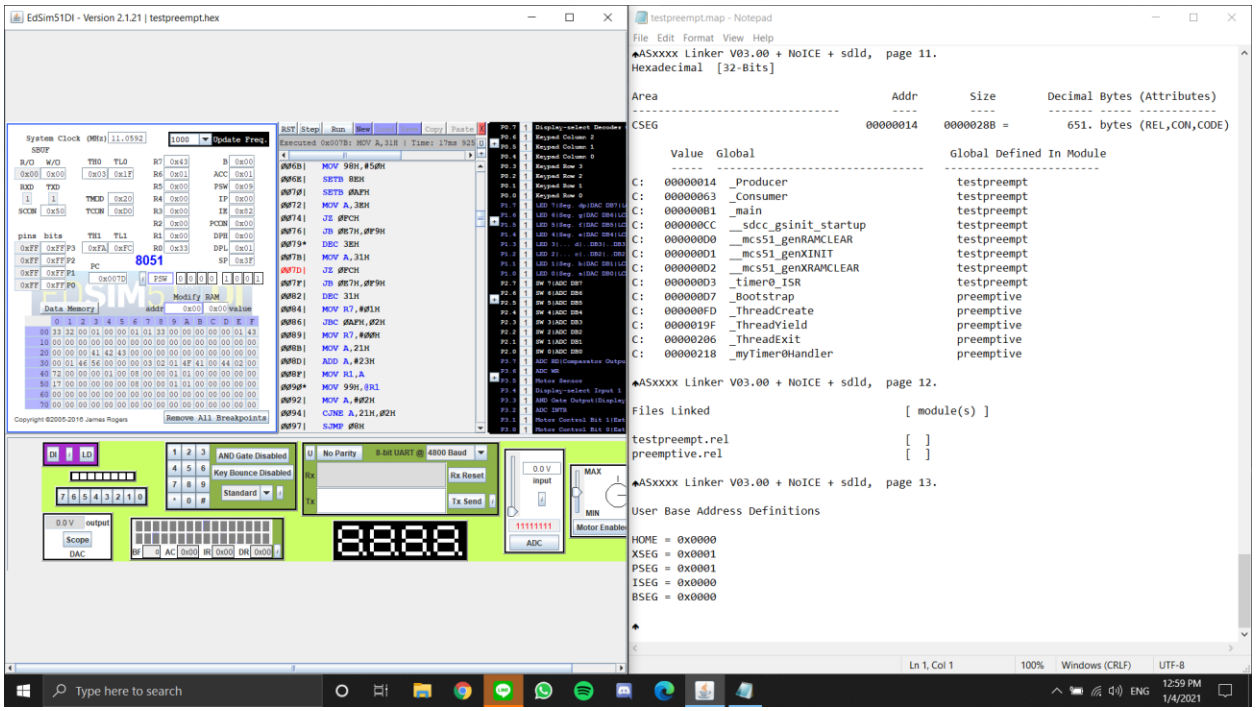
2. Consumer is running and show semaphore changes

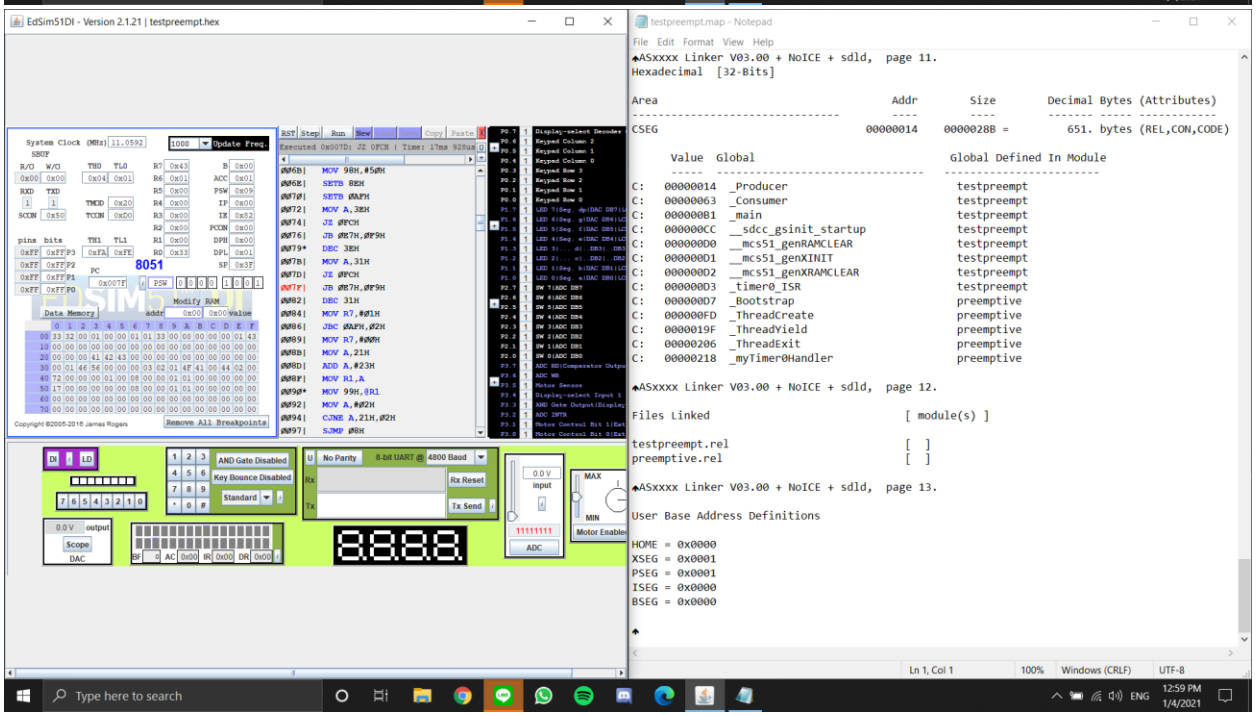
107062381 楊孝偉



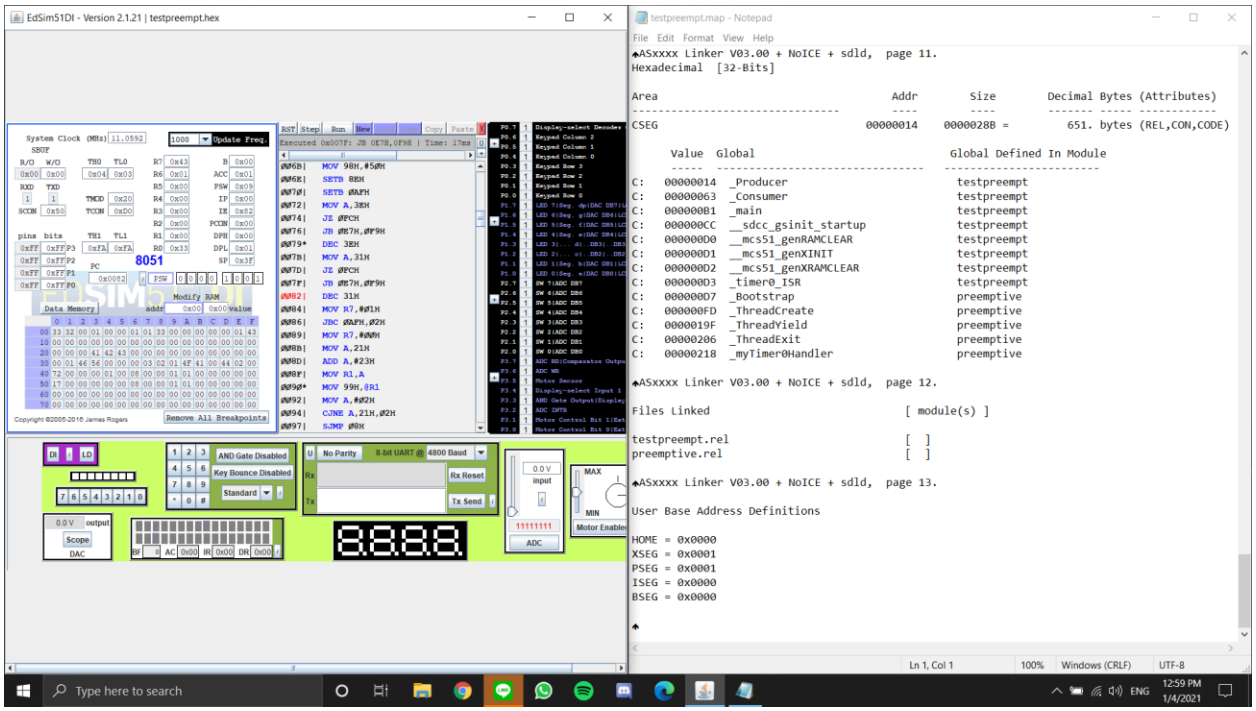


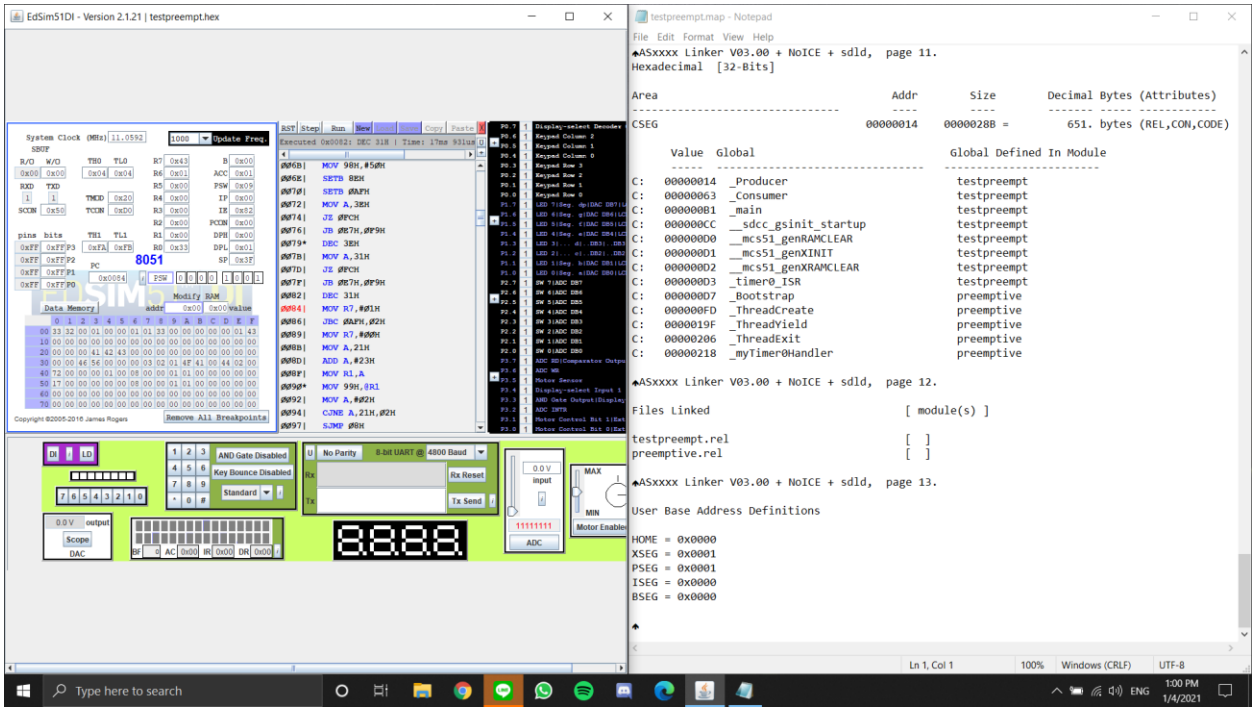
107062381 楊孝偉





107062381 楊孝偉





107062381 楊孝偉