ECEN 468   Advanced Logic Design
Department of Electrical and Computer Engineering
Texas A&M University

(Lab exercise created by Jaeyeon Won and Jiang Hu)

Lab 3

Design of System Bus

Purpose:

In this lab, we will design System Bus with SystemC, and will use Vista as a tool for simulation and verification. Also, we will use this module to connect a 256K SRAM and UART which are done in previous labs.

Preparation:

1. Brief introduction to System Bus.

In computer architecture, a bus is a sub-system that transfers data between components inside a computer, or between computers. Early computer buses were literally parallel electrical wires with multiple connections, but the term is now used for any physical arrangement that provides the same logical functionality as a parallel electrical bus. By using system bus, the number of data pins used to connect all devices can be decreased sharply. It is easy to see in case of the system includes multiple CPUs and memory devices.

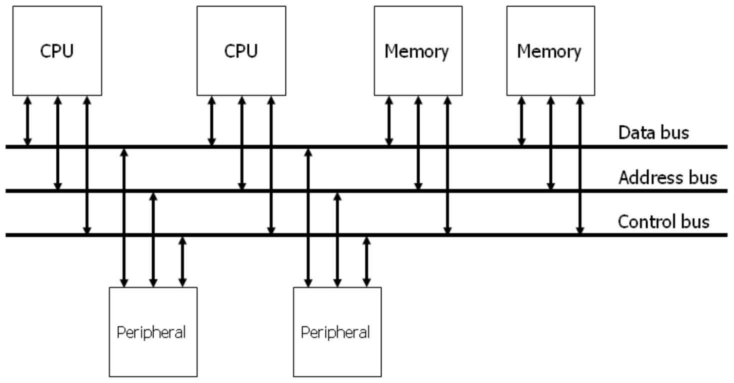

Figure 1   System Bus

An address bus is a computer bus (a series of lines connecting two or more devices) that is used to specify a physical address. When a processor or DMA-enabled device needs to read or write to a memory location, it specifies that memory location on the address bus (the value to be read or written is sent on the data bus). The width of the address bus determines the amount of memory a system can address. A control bus is part of a computer bus, used by CPUs for communicating with other devices within the computer. While the address bus carries the information on which device the CPU is communicating with and the data bus carries the actual data being processed, the control bus carries commands from the CPU and returns status signals from the devices, for example if the data is being read or written to the device the appropriated line (read or write) will be active.

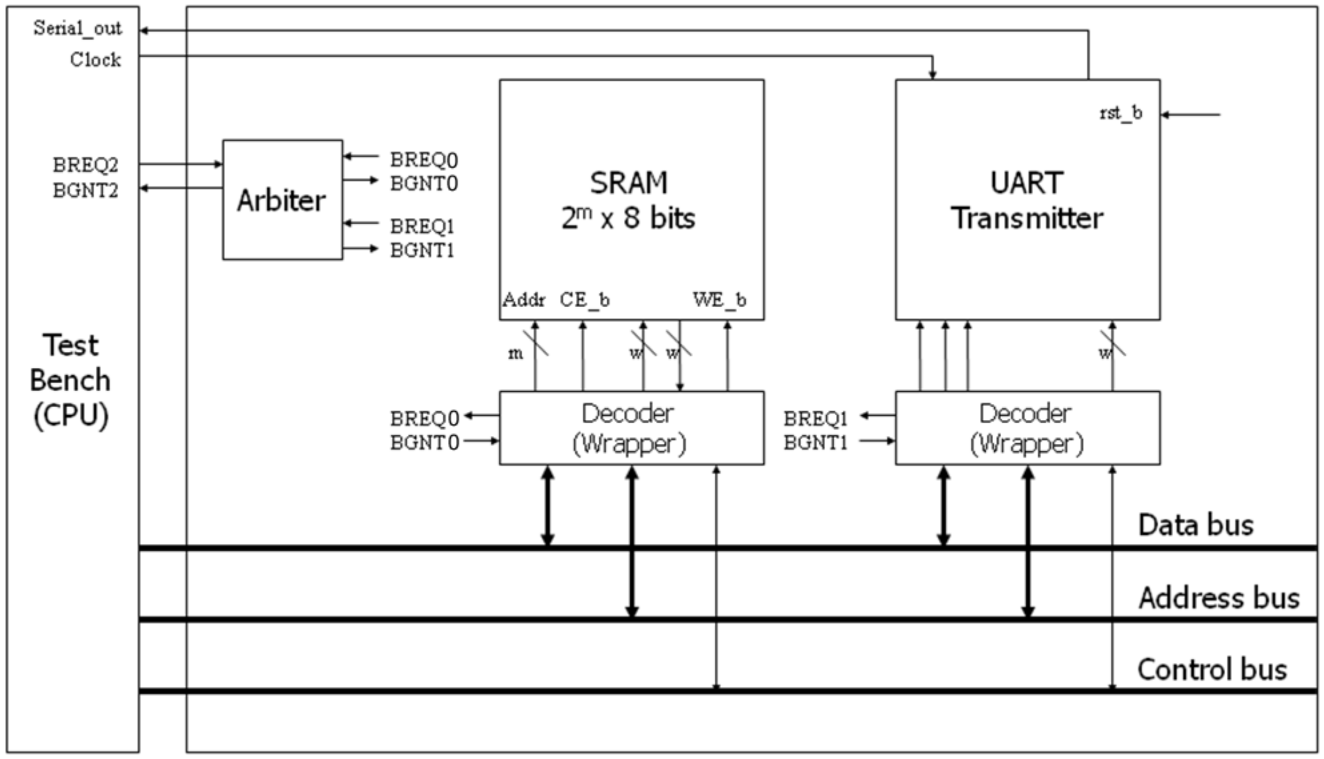

Figure 2 Connections among memory, peripheral and system bus

Whenever a device needs to communicate with another device connected to the main-board, it must do so over the bus. Because the bus is shared amongst all the devices, a method for deciding which device gets to use the bus must be used. The method used to determine who gets access to the bus and when is referred to as bus arbitration. The bus arbitration mechanism is designed so that high priority devices like the processor and RAM get first access to the bus, while other device (disks, video cards, sound cards etc.) get lower priority, and often have to wait to access the bus. There are several modes can used for the arbitration. Among them, we will use centralized fixed-priority arbitration which has simple circuit to implement and shown in Figure 3.

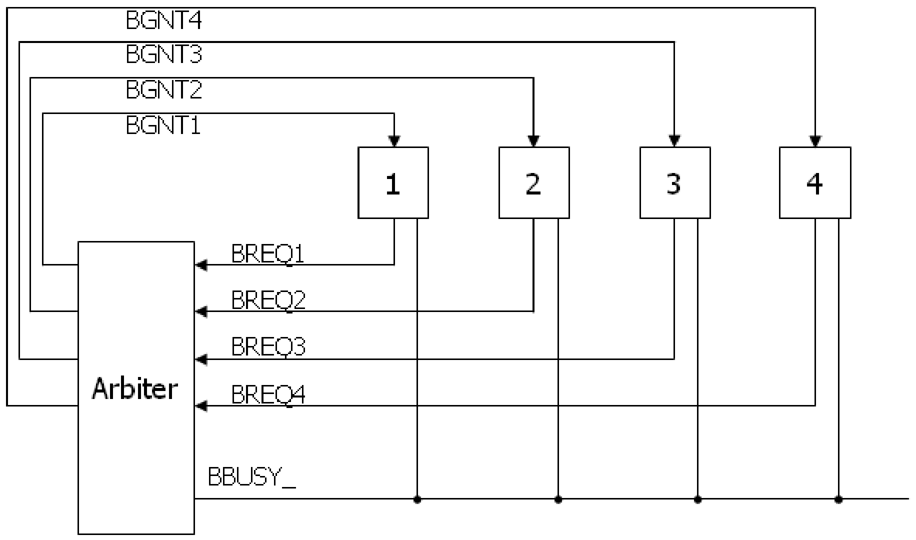

Figure 3 Circuit of Centralized fixed-priority arbitration

In Figure 3, there are three types of signals used. BREQ signals are generated by CPU, memory and peripherals. If they want to use bus to read or write, they must be allocated from the arbiter. To get the right to use the bus, it sends request signal which is BREQ to the arbiter. The arbiter sends grant signals to the device which sent request signals if the bus is free and can be used. If two or more request signals are generated, the arbiter computes their priorities and send only one grant signal to the device has higher priority. If the bus is free, the status of BBUSY_ signal is high (logic 1). If any device gets the right to use the bus, they read data from the bus or write data to the bus while it makes BBUSY_ signal being low state (logic 0). After it completes its operation related to bus, release the BBUSY_ signal which will go high state back. For example, (1) if master 3 sends the request signal (BREQ3) to the arbiter while master 1 is using the bus, (2) the arbiter sends the grant signal (BGNT3). (3) When master 1 completes its operation and make BBUSY_ signal inactive (goes high), (4) master 3 makes BBUSY_ signal active (goes low) and starts to read or write to the bus.

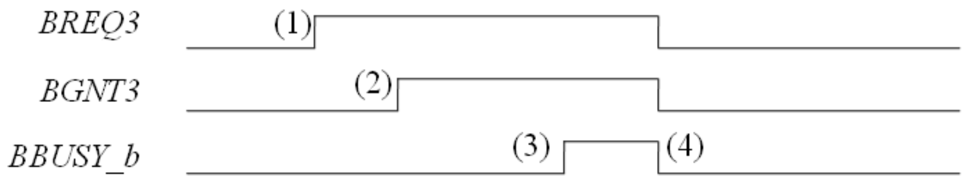

Figure 4 An example of the process for arbitration

We assume that the test-bench has the highest priority and the UART has the lowest priority. **We can also see the module attached between SRAM/UART and system bus. It is called decoder or wrapper. Small size system has a few number of pins to control other devices, but large size system has a lot of pins to control all devices, so the number of control pins must be connected with devices will be huge. Thus, control signals can be also included in the system bus. The width**

of address bus depends on the application we design. We assume that the width of the address bus is 32 bits. The 4 most significant bits are used for identification number of devices. The ID of CPU which is a test-bench in our design is 0011, the ID of SRAM is 0001 and 0010 for UART transmitter. The remaining is used for control signals and address signals for memory. Figure 5 and Figure 6 shows the circuit for decoder and address map.

ID[31:28]	reserved[27:20]	CE_b[19]	WE_b[18]	Addr[17:0]

Figure 5 address map to control memory

ID[31:28]	reserved[27:4]	T_byte[2]	Byte_ready[1]	Load_XMT_datareg[0]

Figure 6 address map to control UART

3. Functional Simulation

Once the design is completed it must be tested. The functionality of the design module can be tested by applying a testbench and checking the results. The testbench module can instantiate the design module and directly drive the signals in the design module. The testbench can be complied along with the design module. At the end of compilation the simulation results will be displayed.

Download 'Lab3_code.tar.gz' file from the lecture website. You will design modules in these files after you decompress the file.

a. Make working folder for this lab.

```
HOME]$ mkdir ECEN468/Lab3/SRC                 (make work folder)
HOME]$ cd ECEN468/Lab3/SRC                    (move to work folder)
Move the file 'Lab3_code.tar.gz' into the working folder
ECEN468/Lab3/SRC]$ tar xvfz Lab3_code.tar.gz  (decompress the file)
```

Please check the files decompressed. It should contain the files below.
SRAM_WRAP.cpp, SRAM_WRAP.h, UART_TXMR_WRAP.cpp, UART_TXMR_WRAP.h
Arbiter.cpp, Arbiter.h, test.h, test.cpp and main.cpp

b. Also, copy SRAM.cpp that you designed from Lab1 to this lab folder.
c. Also, copy UART_XMTR.cpp and UART_XMTR.h that you designed from Lab2 to this lab folder.
d. Insert your code into module files. Your module should satisfy the requirements.
(SRAM_WRAP.cpp, SRAM_WRAP.h, UART_TXMR_WRAP.cpp, UART_TXMR_WRAP.h)
e. You should use the test-bench given.

f. Insert your code into the top module (main.cpp).

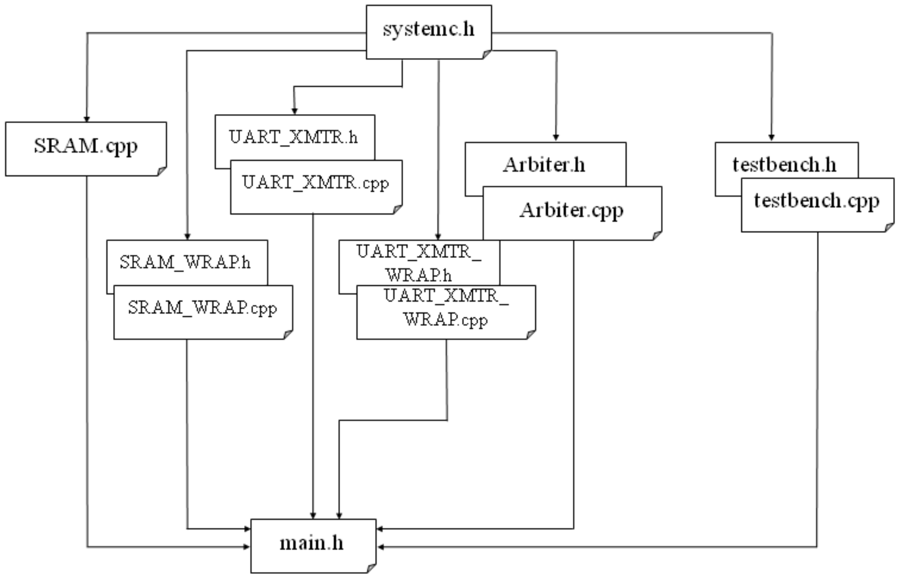

Figure 7 Hierarchy of files used in this lab

Please refer to Lab1 for about how to use simulator.

g. Please show your result to TA within your regular lab hour to get credit for lab evaluation.

4. Tips for simulation

a. Declaration of inout signal and how to control in test-bench
   - Inside SC_MODULE() : sc_inout_rv <# of pins> 'Pin name'
     'Pin name'.write("ZZZZZZZZ"); or 'Pin name'.write(value);

   - Inside sc_main() : sc_signal_rv <# of pins> 'Pin name'

b. If you meet an error which is '(E115) sc_signal<T> cannot have more than one driver:' while you do simulation, write the below command on command prompt.

   ECEN468/Lab3/SRC]$ **Setenv SC_SIGNAL_WRITE_CHECK DISABLE**

c. <u>If you meet type mismatch errors with some signals in SRAM.cpp file. Please modify your SRAM module refering to sample SRAM.cpp file at the end of the manual.</u>

Requirements:

1. Your design modules should satisfy constraints below.
   a. It should control SRAM and UART correctly.
   b. Detailed comments.
   c. Please do not change the names of the signals given and the other functions not mentioned in the report. (You will lose some points if you change any names of the input or output signals.)
   d. You may modify the test-bench given to get the results only if it is resaonable. Please write detailed comments if you have modified.
   e. Late penalty : 20% of total score will be deducted on each subsequent weekday after due date.

2. Submit hardcopy of your report to TA. The report should include contents below.
   a. SRAM_WRAP.cpp (3 points)
   b. SRAM_WRAP.h (3 points)
   c. UART_TXMR_WRAP.cpp (3 points)
   d. UART_TXMR_WRAP.h (3 points)
   e. main.cpp (3 points)
   f. sim.out (test message output) with analysis. (10 points)
   g. Captured waveform with analysis. (10 points)

Note : Please send the files to TA's email address. (5 points)

SRAM_WRAP.cpp, SRAM_WRAP.h, UART_TXMR_WRAP.cpp, UART_TXMR_WRAP.h
Arbiter.cpp, Arbiter.h, test.h, test.cpp, main.cpp, UART_XMTR.cpp,UART_XMTR.h and SRAM.cpp

----

```
SC_MODULE (SRAM) {
  sc_in_rv    < ADDR_WIDTH > Addr    ;
  sc_in       <bool>       bWE    ;
  sc_in       <bool>       bCE    ;
  sc_in_rv    < DATA_WIDTH > InData  ;
  sc_out_rv   < DATA_WIDTH > OutData ;

  // ----- Internal variables -----
  sc_uint <DATA_WIDTH> mem [RAM_DEPTH];
  uint data, adr;

  // ----- Code Starts Here -----
  // Memory Write Block
  // Write Operation : When bWE = 0, bCE = 0
  void write_mem () {
    if (!bCE.read() && !bWE.read()) {
      data=InData.read().to_uint();
      adr=Addr.read().to_uint();
      mem[adr] = data;
    }
  }

  // Memory Read Block
  // Read Operation : When bWE = 1, bCE = 0
  void read_mem () {
    if (!bCE.read() && bWE.read())  {
      adr = Addr.read().to_uint();
      OutData.write(mem[adr]);
}
  }
};
```