

# High-Level Modelling and Communication Modeling

**Abstract**— In this document, a complete system with processor, memory, bus and channel is to be designed using system C.

**Keywords**— Memory, Processor, Bus, Channel, FIFO, Multiplier, Intermediate component

## I. INTRODUCTION

If a full system with all components is to be designed, it is difficult to design this system at the RT-level. Therefore, it is reasonable to first design every component and the whole system at a higher level, and only after the successful simulation of this design can every component be designed at the RT-level. In this computer assignment, a full system should be modeled by "bracketing" in systemC.

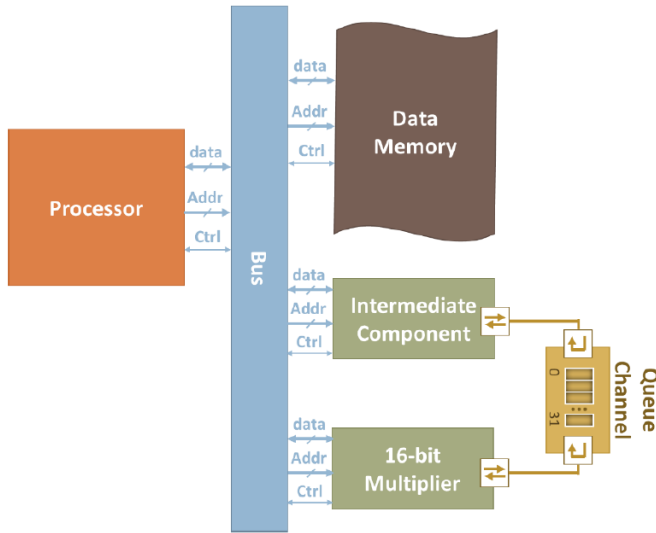


Fig. 1 Overall view of the system

## II. MEMORY

Memory is one of the most important part of the design, because all-data should be read from memory and after all-process results should be saved in the memory. So it's necessary to design and modeled memory first.

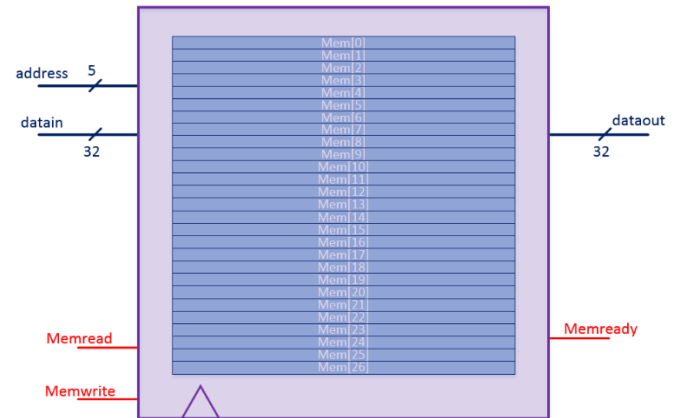


Fig 2. Memory-block schematic

As shown in Fig 1, a memory consists of a few registers. The memory has two input-control signals known as "Memread" and "Memwrite". When Memread is asserted, the memory reads data from the selected register specified by the address and places this data on the "dataout" bus. On the other hand, when Memwrite is asserted, the memory reads data from the "datain" bus and saves this data to the selected register specified by its address. It's worth noting that reading from the memory is asynchronous, whereas writing is synchronized to the clock.

To model this memory in SystemC, the easiest approach is to utilize an array of "sc\_lv". Depending on the control signals, the memory can either read or write data into the array.

To initialize data in the memory, an initialization method needs to be developed. In this computer assignment, the method takes the name of a file and reads the data in that file line by line, and then puts each line of data into the array. Additionally, a function is available for dumping all memory data into a "txt" file. Since this function does not exist in the concurrent body, it is necessary to call it to dump all the data.

After modeling memory in systemC, it's necessary to develop a test-bench to justify functionality of this memory. As shown in figures 3, 4 and 5, the memory is working correctly.

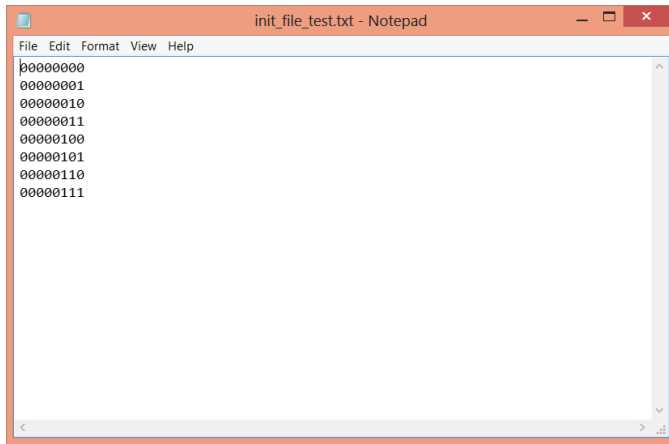


Fig. 3 Initial-file

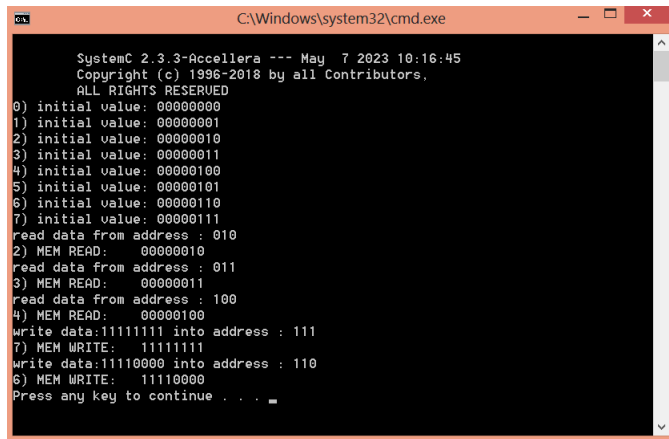


Fig. 4 Test-bench output

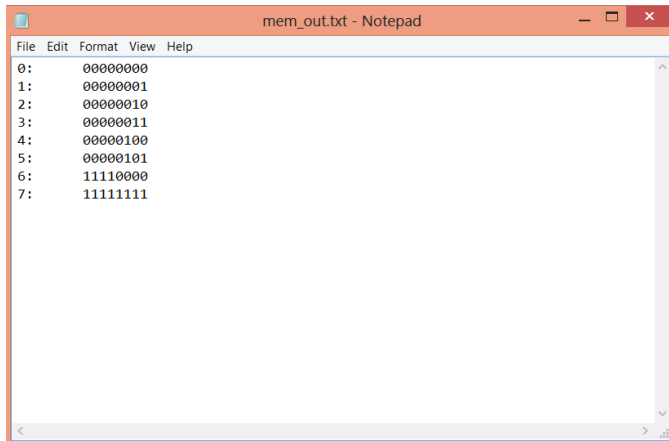


Fig. 5 dumped memory data

### III. BUS

In every high-level system, there should be an arbiter to handle communication between different components. But in this project, "bus" is going to handle the communications.

As shown in Figure 6, the "Bus" in this project is not simply a few wires connecting components to each other. Rather, the "Bus" serves as an abstraction of an arbiter, allowing it to handle communication between components. To achieve this

purpose, when the processor attempts to use the memory by asserting the "memwrite" or "memread" signals, the Bus copies the data and address buses of the processor into the memory input buses. Once the memory operation is complete, the Bus copies the data and ready buses of the memory into the related buses in the processor. The Bus performs the exact same operation for the two other components.

Now it's necessary to develop a test-bench for this design to justify the functionality of this Bus. The result of this test-bench can be seen in Figure 7.

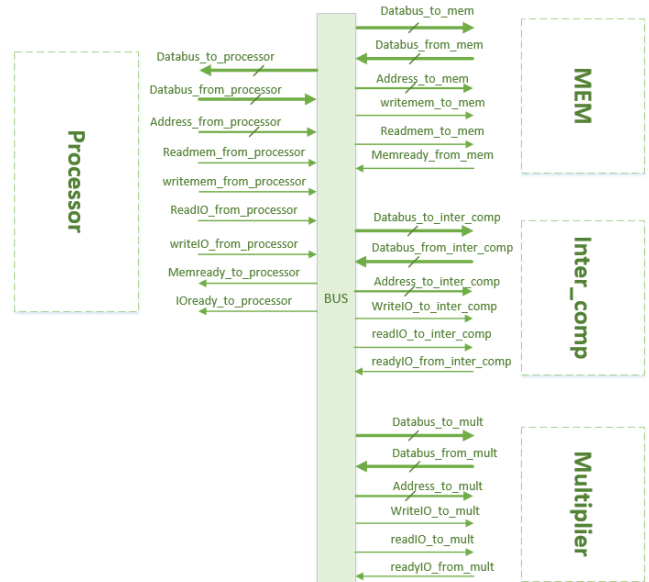


Fig. 6 Bus block diagram

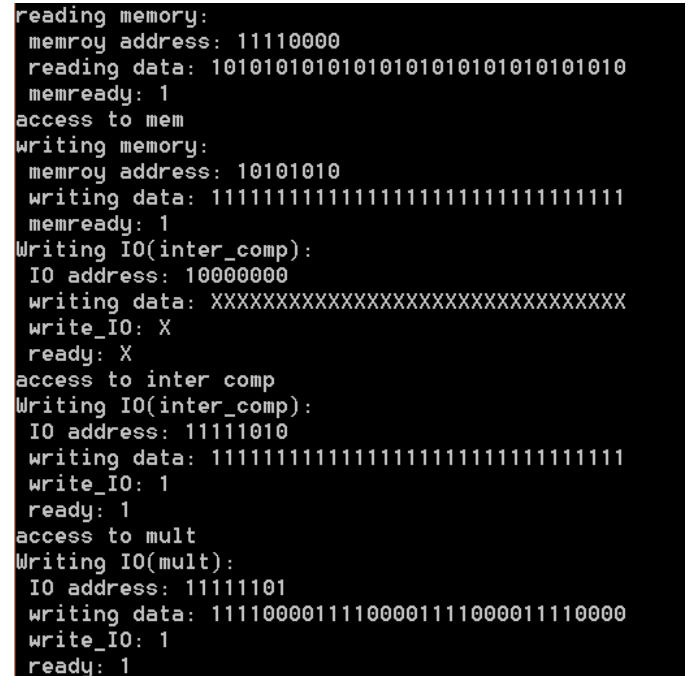


Fig. 7 test-bench result for Bus

#### IV. FIFO CHANNEL

In this computer assignment, an abstract channel is required to connect the multiplier and intermediate component. This FIFO channel should have the ability to store 32 numbers, each consisting of 32 bits. To develop this channel, the interface must be designed first. It should include a put-interface, a get-interface, and a checking-empty-interface. The first two interfaces are explained in the class-slides, but the last interface is new. The checking-empty-interface is added to verify if the FIFO is empty or not. Once the interfaces are developed, the FIFO channel can inherit from all of them. One of the most important point of the design is that the channel must be developed as "blocking". This means that if the FIFO is already full of data, and a method attempts to put a new data into the FIFO, the execution of that method should be halted until another method gets data from the FIFO. Using "sc\_event" is one of the most effective way to achieve this goal. Two "sc\_event", "put\_event" and "get\_event", can be declared. When a method tries to put data in the FIFO while it's full, the method should wait until "get\_event" notifies by get-method. Figure 8 illustrates the FIFO channel and its interfaces.

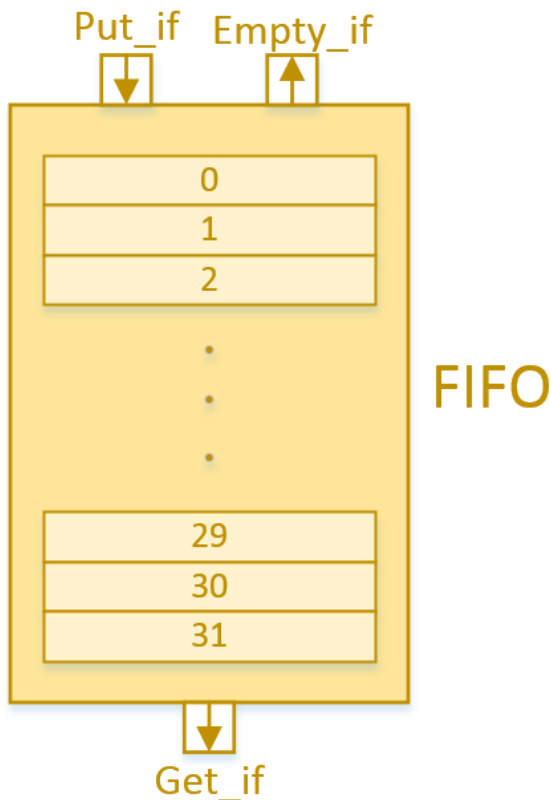


Fig. 8 Block diagram of FIFO channel

As previous parts, a test-bench should be developed to check the functionality of this FIFO channel. The result of this test-bench is shown in Figure 9.

```
new data pushed: 0001
4) is_empty: 0
5) put data: 0010
new data pushed: 0010
one element popped: 0000
6) get data: 0000
one element popped: 0001
7) get data: 0001
8) put data: 1000
new data pushed: 1000
9) put data: 1100
new data pushed: 1100
10) put data: 1110
new data pushed: 1110
one element popped: 0010
11) get data: 0010
one element popped: 1000
12) get data: 1000
one element popped: 1100
13) get data: 1100
14) is_empty: 0
one element popped: 1110
15) get data: 1110
16) is_empty: 1
```

Fig. 9 Test-bench result for FIFO

#### V. INTERMEDIATE COMPONENT AND MULTIPLIER

The intermediate component is an element that can store data in the FIFO channel. However, the main question is how the processor can use it. The intermediate component consists of two 32-bit registers: the "control-register" and the "data-register". When the processor wants to put new data into the FIFO, it must first read the "control-register". The intermediate component checks if the FIFO is empty or not, and if it is empty, it sets the least significant bit of the "control-register" to one. Therefore, if the FIFO is empty, the processor can determine this and load data into the "data-register" of the intermediate component. Every time a new data is loaded into the "data-register", it should be put into the FIFO.

When the processor wants to read from or write data into the intermediate component registers, it should assert the "readIO" or "writeIO" signal. However, what about the address? Every IO component has a particular address that the processor is aware of. So when the processor decides to use a component, it should place the component's address into the address bus. In this project, the address of every component is referred to as the "base-address".

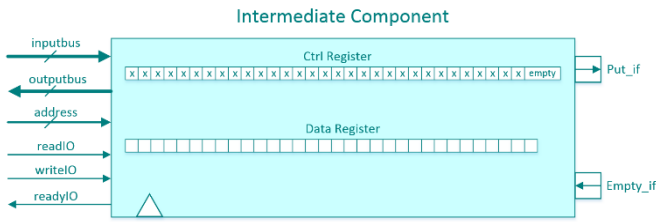


Fig. 10 Intermediate component register accurate view

What about the multiplier? How does it work? The multiplier has the same registers as the intermediate component. First, the processor issues its "writeIO" signal and loads new data into the "ctrl register". This data has two important bits. The least significant bit is the "start" signal. Whenever the start signal becomes one, the multiplier should retrieve one 32-bit number from the FIFO. After that, the multiplier should break this 32-bit number into two 16-bit numbers, multiply them together, and store the result in the "data register". Once the result is stored in the "data register", the "done" signal - the second bit of the "ctrl register" - should be asserted. Meanwhile, the processor reads the "ctrl register" in every clock cycle and waits for the "done" signal to become one. Once the "done" signal becomes one, the processor reads the "data register" value and stores it into the memory.

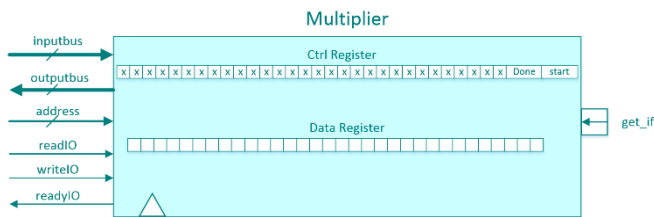


Fig 11. Multiplier register accurate view

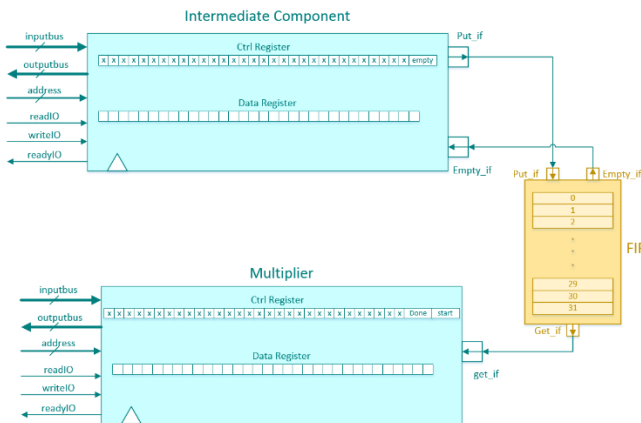


Fig 12. Test-bench of multiplier and intermediate component

For testing these two component- the intermediate component and the multiplier- it's necessary to develop a test-bench in systemC. In this test-bench in addition of these two components, a FIFO channel also should be instantiated. The result of this test-bench is shown in Figures 13, 14 and 15.

```
1) reading ctrl register of inter_comp:
output: XXXXXXXX
2) writing data 00010001 into fifo:
new data pushed: 00010001
ready: 1
3) writing data 00100010 into fifo:
new data pushed: 00100010
ready: 1
4) writing data 00110011 into fifo:
new data pushed: 00110011
ready: 1
5) writing data 01000100 into fifo:
new data pushed: 01000100
ready: 1
6) reading ctrl register of inter_comp:
output: XXXXXXXX
```

Fig 13. The test-bench result of intermediate component

```
7) writing data 00000001 into ctrl register of mult(start multiplication):
one element popped: 00010001
ready: 1
8) reading ctrl register of mult:
output: 00000001
9) reading ctrl register of mult:
output: 00000010
10) reading first result:
output: 00000001
11) writing data 00000001 into ctrl register of mult(start multiplication):
one element popped: 00100010
ready: 1
12) reading ctrl register of mult:
output: 00000001
13) reading ctrl register of mult:
output: 00000010
14) reading first result:
output: 00000100
```

Fig 14. The test-bench result of multiplier (first part)

```
15) writing data 00000001 into ctrl register of mult(start multiplication):
one element popped: 00110011
ready: 1
16) reading ctrl register of mult:
output: 00000001
17) reading ctrl register of mult:
output: 00000010
18) reading first result:
output: 00001001
19) writing data 00000001 into ctrl register of mult(start multiplication):
one element popped: 01000100
ready: 1
20) reading ctrl register of mult:
output: 00000001
21) reading ctrl register of mult:
output: 00000010
22) reading first result:
output: 00010000
```

Fig 15. The test-bench result of multiplier (second part)

## VI. PROCESSOR

In every high-level system, there should be a processor acting as a manager unit that controls the flow of instructions in the system. Processors are elements in the system that can execute different variations of instructions, but in this project, the processor should be modeled by "bracketing". This means that the processor can only execute a specific instruction (control data flow) and no more.

In this computer assignment, the processor should first check if the FIFO is empty or not. To do so, the processor should read the "ctrl register" of the intermediate component. If the channel is empty, the processor should read 32 numbers from memory and send them to the intermediate component in some consecutive clock cycles. The intermediate component stores these data into the FIFO. After the FIFO becomes full, the processor should issue the start signal of the multiplier. To achieve this, the processor should write a data that has the least significant bit set to one. Once the multiplier starts its operation, the processor should wait for the "done" signal to become 1. The "done" signal is the second least significant bit of the multiplier's "ctrl register". When the "done" signal becomes 1,

correctly or not, the memory should be dumped into a file that can display the results.

Figure 20 displays the output file of the memory. The results of the multiplication can be verified with these values.

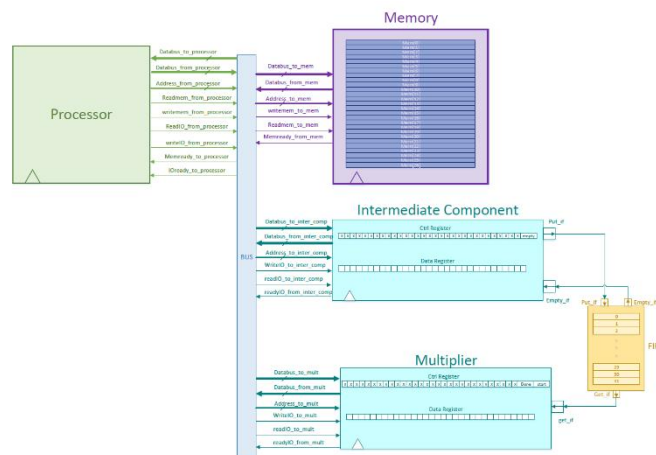
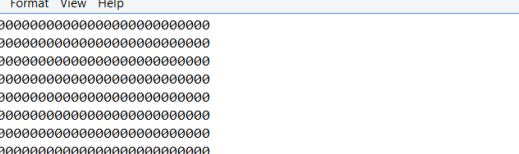


Fig 18. System design block diagram

[illegible]

mem\_out.txt - Notepad

File Edit Format View Help

```

138: 00000000000000000000000000000000
139: 00000000000000000000000000000000
140: 00000000000000000000000000000000
141: 0000000000000000000000000000000001
142: 0000000000000000000000000000000100
143: 00000000000000000000000000000001001
144: 0000000000000000000000000000010000
145: 0000000000000000000000000000011001
146: 00000000000000000000000000000100100
147: 00000000000000000000000000000110001
148: 00000000000000000000000001000000
149: 0000000000000000000000000000000001
150: 0000000000000000000000000000000100
151: 00000000000000000000000000000001001
152: 0000000000000000000000000000010000
153: 0000000000000000000000000000011001
154: 00000000000000000000000000000100100
155: 00000000000000000000000000000110001
156: 00000000000000000000000000000100000
157: 0000000000000000000000000000000001

```

Fig 20. Results of the multiplication that are stored in the memory

In this part, the system, as shown in Figure 18, should be constructed. To achieve this, it is necessary to instantiate each element that has been designed and implemented in SystemC in the previous parts. After instantiation, all elements should be bound together. Then, the system is ready to be tested. The system works automatically, and it just needs simulation time to read numbers from memory, calculate multiplication results, and store them in memory. To verify that the system is working

