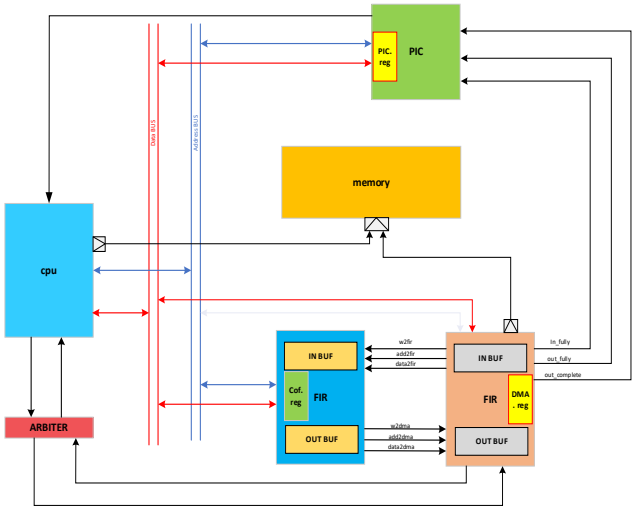
CA4 VHDL(Memory Map SystemC version)

Amirhossein ilkhani
school of Electrical Engineering
University of Tehran
Tehran, Iran
hossein.ilkhani@ut.ac.ir

*Abstract*__ ***SystemC is a set of C++ classes and macros which provide an event-driven simulation kernel in C++, together with signals, events, and synchronization primitives, deliberately mimicking the hardware description languages VHDL and Verilog. The goal of this assignment is to use systemC for mimicking the soc which was described with HDL in the last assignment. Also, the systemC bracketing model is used for CPU and TLM for communicating MEMORY with CPU and DMA-controller***.

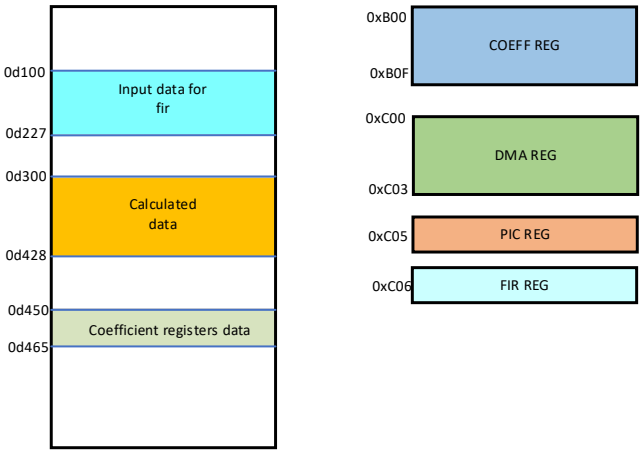
A. SOC OVERVIEW

Our system includes the main processor (PUNEH), FIR, DMA controller, Arbiter, and Programmable interrupt controller. The processor fills the coefficient registers and then set the DMA for reading the related data and puts them in the buffer of the DMA. After this with an in fully interrupt signal processor realizes now the fir is ready to start the calculation with the data we buffered in DMA. When the calculation is finished and the calculated data was written in the buffers, the out full interrupt is set for interrupting PUNEH to save the data in the ram. After finishing this operation out_complete interrupts PUNEH to clear the configuration registers of DMA. All these actions are done by using SystemC and in the next parts, the design of these components will be described in SystemC and illustrated.



B.MEMORY MAP

Because of using the bracketing model for CPU there is no need for instruction data in memory for fetching but the timing is important which is regarded in CPU modeling, so the memory is used just for data. Each data location and memory map are illustrated below:



C. PIC

a programmable interrupt controller (PIC) is an integrated circuit that helps a microprocessor (or CPU) handle interrupt requests (IRQ) coming from multiple different sources. In this PIC; there are 3 interrupts which are in_fully and out_fully and out_complete and the return address for each interrupt is 0x900, 0x910, 0x920 respectively. The SystemC code of this part is:

```
    SC_CTOR(PIC)
    {

        SC_THREAD(operation)
            //sensitive<<clk.pos();
            sensitive << AddrBus << infully << outfully << out_complete;
    }
    void operation();
};

void PIC::operation()
{
    while (true)
    {
            if(infully == '1'){
                interrupt = SC_LOGIC_1;
                if (AddrBus->read() == "0x0c05")
                    DataBus = IR1ISR;
            }
            else if(outfully == '1'){
                interrupt = SC_LOGIC_1;
                if (AddrBus->read() == "0x0c05")
                    DataBus = IR2ISR;
            }
            else if(out_complete == '1'){
                interrupt = SC_LOGIC_1;
                if (AddrBus->read() == "0x0c05")
                    DataBus = IR3ISR;
            }
            else{
                interrupt = SC_LOGIC_0;
                //DataBus = "0x0000";
            }
        wait();
    }
}
```

D. ARBITER

the bus arbiter is a device used in a multi-master bus system to decide which bus master will be allowed to control the bus for each bus cycle. Arbiter is used every time the DMA or CPU wants to take the bus. The DAM has a higher priority. The SystemC code of this part is:

```
    SC_CTOR(Arbiter)
    {
        SC_THREAD(operation)
            sensitive << dma_req<<cpu_req;
    }
    void operation();
};
void Arbiter::operation()
{
    int dma_en = 0;
    int cpu_en = 0;
    while (true)
    {
        if(dma_req == '0'){
            dma_gnt = sc_logic_0;
            dma_en = 0;
        }
        if(cpu_req == '0'){
            cpu_gnt = sc_logic_0;
            cpu_en = 0;
        }
        if(dma_req == '1'){
            if(cpu_en == 0){
                dma_gnt = sc_logic_1;
                dma_en = 1;
            }
            else{
                dma_gnt = sc_logic_0;
                dma_en = 0;
            }
        }
        if(cpu_req == '1'){
            if(dma_en == 0){
                cpu_gnt = sc_logic_1;
                cpu_en = 1;
            }
            else{
                cpu_gnt = sc_logic_0;
                cpu_en = 0;
            }
```

E. FIR

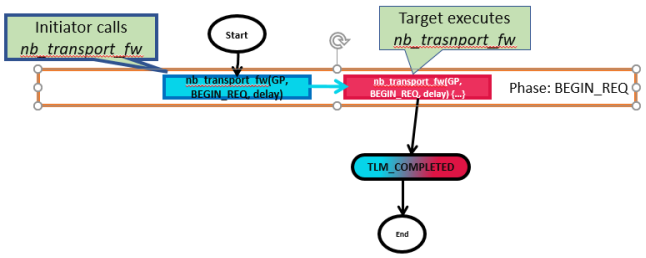
This fir unit has 16 coefficient registers and a 128bit input buffer and output buffer. The coefficient registers are filled by the CPU and when the FIR is called for starting(AddrBus = 0x0C06) the FIR starts calculating data and putting them in the output buffer. The SystemC code of this part is:

```
int coeff_reg[16];
int inBuf[128];
int outBuf[128];

SC_CTOR(FIR)
{

    SC_THREAD(coeff_reg_op)
        sensitive << clk.pos();

    SC_THREAD(start_op)
        sensitive << clk.pos();

    SC_THREAD(fir_op)
        sensitive << start.posedge_event();
```

```
void FIR::fir_op()
{
    while (true)
    {
        if(start == '1'){
            temp = 0;
            result = 0;
            numofdata = Ndata;
            cout<< "fir calculation was started @"<<sc_time_stamp()<< endl;

            for(int i = 0; i < numofdata; i++){
                for(int j = 0 ; j < coeff_reg[0] ; j++){
                    cout<<"coeff_reg= "<<coeff_reg[j+1]<<"\tinBuf= "<<inBuf[i+j]<<endl;
                    temp = coeff_reg[j+1]*inBuf[i+j];
                    result = result + temp;
                    wait(clk->posedge_event());
                }
                cout<<"result= "<<result<<"\tcalculated @"<<sc_time_stamp()<< endl;
                outBuf[i] = result;
                temp = 0;
                result = 0;
            }

            w2dma = sc_logic_1;
            for(int i = 0; i < 128; i++){
                    add2dma = i;
                    data2dma = outBuf[i];
                    wait(0,SC_PS);
                }
            w2dma = sc_logic_0;
            cout<< "fir calculation was finished @"<<sc_time_stamp()<< endl;
            }
        wait();
    }
}

void FIR::start_op()
{
    while (true)
    {
        if(AddrBus->read() == "0x0C06"){
            start = sc_logic_1;
        }
        else{
            start = sc_logic_0;
        }
        wait();
    }
}
```

F.TLM

The DMA and CPU use "one cycle nonblocking TLM" for reading and writing in memory. The protocol is illustrated below:



Base code for TLM-reading which is used in DMA and CPU is :

```
tlm::tlm_phase forwardPhase;
sc_time processTime; // Processing time of initiator prior to call

processTime = sc_time(0, SC_PS);

    tlm::tlm_command cmd = tlm::TLM_READ_COMMAND;

    nBlockWriteRead->set_command(cmd);
    nBlockWriteRead->set_address(Add);
    nBlockWriteRead->set_data_ptr((unsigned char*)data);
    nBlockWriteRead->set_data_length(1);
    nBlockWriteRead->set_streaming_width(1);
    nBlockWriteRead->set_byte_enable_ptr(0);
    nBlockWriteRead->set_dmi_allowed(false);
    nBlockWriteRead->set_response_status(tlm::TLM_INCOMPLETE_RESPONSE);

    forwardPhase = tlm::BEGIN_REQ;

    cout << "CPU side  " << (cmd ? 'W' : 'R') << ", @" << Add << " data:";
    sc_lv<8> vv;
    cout << data[0] << " ";
    cout << " @time " << sc_time_stamp() << " delay=" << processTime << '\n';

    tlm::tlm_sync_enum returnStatus;
    returnStatus = memWRSocket->
        nb_transport_fw(*nBlockWriteRead, forwardPhase, processTime);

    int* ptr = reinterpret_cast<int*>(nBlockWriteRead->get_data_ptr());
    cout<< "readData ="<<(*ptr)<<endl;
    return (*ptr);
}
```

Also, the base code for TLM-writing is :

```
tlm::tlm_phase forwardPhase;
sc_time processTime; // Processing time of initiator prior to call

processTime = sc_time(0, SC_PS);

    tlm::tlm_command cmd = tlm::TLM_WRITE_COMMAND;

    nBlockWriteRead->set_command(cmd);
    nBlockWriteRead->set_address(Add);
    nBlockWriteRead->set_data_ptr((unsigned char*)putdata);
    nBlockWriteRead->set_data_length(1);
    nBlockWriteRead->set_streaming_width(1);
    nBlockWriteRead->set_byte_enable_ptr(0);
    nBlockWriteRead->set_dmi_allowed(false);
    nBlockWriteRead->set_response_status(tlm::TLM_INCOMPLETE_RESPONSE);

    forwardPhase = tlm::BEGIN_REQ;

    cout << "CPU side  " << (cmd ? 'W' : 'R') << ", @" << Add << " data:";
    sc_lv<8> vv;
    cout << putdata << " ";
    cout << " @time " << sc_time_stamp() << " delay=" << processTime << '\n';

    tlm::tlm_sync_enum returnStatus;
    returnStatus = memWRSocket->
        nb_transport_fw(*nBlockWriteRead, forwardPhase, processTime);

    cpu_req = sc_logic_0;
    wait(clk->posedge_event());
}
```

Both of these codes are in the initiator and call the "nb_tansport_fw", on other hand, the target(memory) executes the "nb_tansport_fw", which is in memory and is described here:

```
class memoryUnit : public sc_module {
public:
    tlm_utils::simple_target_socket<memoryUnit, 32> memSocket;
    tlm_utils::simple_target_socket<memoryUnit, 32> dmaSocket;

    static const int SIZE = 512;

    SC_CTOR(memoryUnit) : memSocket("cpu_side_socket"),dmaSocket("dma_side_socket") {
        memSocket.register_nb_transport_fw(this, &memoryUnit::nb_transport_fw);
        dmaSocket.register_nb_transport_fw(this, &memoryUnit::nb_transport_fw);
```

Two sockets are defined for CPU and DMA and the "nb_tansport_fw" is registered in memory.

```
virtual tlm::tlm_sync_enum nb_transport_fw(tlm::tlm_generic_payload&,
    tlm::tlm_phase&, sc_time&);
int memArray[SIZE];


tlm::tlm_sync_enum memoryUnit::nb_transport_fw
(tlm::tlm_generic_payload& receivedTrans,
    tlm::tlm_phase& phase, sc_time& delay) {

    tlm::tlm_command cmd = receivedTrans.get_command();
    uint64           adr = receivedTrans.get_address();
    unsigned char* ptr = receivedTrans.get_data_ptr();
    unsigned int     len = receivedTrans.get_data_length();
    unsigned char* byt = receivedTrans.get_byte_enable_ptr();
    unsigned int     wid = receivedTrans.get_streaming_width();

    if (byt != 0) {
        receivedTrans.set_response_status(tlm::TLM_BYTE_ENABLE_ERROR_RESPONSE);
        return tlm::TLM_COMPLETED;
    }
    if (len > 5 || wid < len) {
        receivedTrans.set_response_status(tlm::TLM_BURST_ERROR_RESPONSE);
        return tlm::TLM_COMPLETED;
    }
    unsigned int i;
    if (cmd == tlm::TLM_READ_COMMAND)
        for (i = 0; i < len; i = i + 1) {
            *(ptr + i) = *((unsigned char*)(memArray + adr + i));
        }
    else if (cmd == tlm::TLM_WRITE_COMMAND)
        for (i = 0; i < len; i = i + 1) {
            *((unsigned char*)(memArray + adr + i)) = *(ptr + i);
            cout<<"memArray["<< adr + i<<"]= "<<memArray[adr + i]<<endl;
        }

    receivedTrans.set_response_status(tlm::TLM_OK_RESPONSE);
    delay = delay + sc_time(0, SC_NS);
    return tlm::TLM_COMPLETED;

}
```

G.DMA

DMA Controller is a hardware device that allows I/O devices to directly access memory with less participation from the processor. This DMA has three registers which are control-register, Address-register, and number-register. The Address of this register is "0xC00", "0xC01" and "0xC02" respectively. The DMA communicates with memory using TLM as described previously.

```
SC_THREAD(memRead);
    sensitive << read_flag;

SC_THREAD(memWrite);
    sensitive << write_flag;

SC_THREAD(register_op);
    sensitive << Addrbus;
```

When the control register is filled for reading the read_flag is set for reading from the memory using TLM. Also When the control register is filled for writing the write_flag is set for writing to the memory using TLM. the codes of this action are:

```
void DMA1::register_op()
{
    while (true)
    {
        if (Addrbus->read() == "0x0C00") {
            control_reg = DataBus;
            wait(clk->posedge_event());
            if(control_reg.read() == "0x0003"){
                read_flag = SC_LOGIC_1;
            }
            else if(control_reg.read() == "0x0005"){
                write_flag = sc_logic_1;
                out_fully_flag = sc_logic_0;
            }
            else if(control_reg.read() == "0x0000"){
                read_flag = sc_logic_0;
                write_flag = sc_logic_0;
                in_fully_flag = sc_logic_0;
                out_fully_flag = sc_logic_0;
                out_complete_flag = sc_logic_0;
            }
        }
        else if (Addrbus->read() == "0x0C01") {
            addr_reg = DataBus;
            wait(clk->posedge_event());
        }
        else if (Addrbus->read() == "0x0C02") {
            number_reg = DataBus;
            wait(clk->posedge_event());
        }
        else if (Addrbus->read() == "0x0C03") {
            reserved_reg = DataBus;
            wait(clk->posedge_event());
        }

        wait();
    }
}
```

The writing and reading were described in the TLM part.

H.CPU

CPU has two important parts, first, instructions which are modeled by SystemC bracketing, and ISR part which executes the instructions according to interrupt. All the timing including fetching, bus requesting, and executions are noticed in this code, for example, the instructions for reading coefficients from memory and writing them in FIR are:

```
for(int i = 450; i < 466; i++){

    cpu_req = sc_logic_1;
    wait(cpu_gnt->posedge_event());
    wait(clk->posedge_event());//read gnt
    wait(clk->posedge_event());//fetch
    cpu_req = sc_logic_1;
    wait(clk->posedge_event());//instruction
    temp[0] = memread(i);
    cpu_req = sc_logic_0;
    wait(clk->posedge_event());


    cpu_req = sc_logic_1;
    wait(cpu_gnt->posedge_event());
    wait(clk->posedge_event());//read gnt
    wait(clk->posedge_event());//fetch
    Addrbus = sc_lv<16>(coeffAddres.read().to_uint() + (i-450));
    DataBus = sc_lv<16>(temp[0]);
    cpu_req = sc_logic_1;
    wait(clk->posedge_event());//instruction
    cpu_req = sc_logic_0;
    Addrbus = "xxxxxxxxxxxxxxx";
    DataBus = "xxxxxxxxxxxxxxx";
    wait(clk->posedge_event());
}
```

After receiving an interruption, the PIC is called for the ISR address according to the interruption. After this CPU executes the needed instruction. again All the timing including fetching, bus requesting, and executions are noticed in this code:

```
if(interrupt == '1'){

    cout<< "interrupt was recieved@"<<sc_time_stamp()<< endl;

    cpu_req = sc_logic_1;
    wait(cpu_gnt->posedge_event());
    wait(clk->posedge_event());//read gnt
    wait(clk->posedge_event());//fetch
    Addrbus = "0x0C05"; //PIC REG
    DataBus = "0x0000"; //
    cpu_req = sc_logic_1;
    wait(clk->posedge_event());//instruction
    cpu_req = sc_logic_0;
    wait(clk->posedge_event());

    if(DataBus->read() == "0x0900"){
        cout<< "in_fully was recieved@"<<sc_time_stamp()<< endl;
        cpu_req = sc_logic_1;
        wait(cpu_gnt->posedge_event());
        wait(clk->posedge_event());//read gnt
        Addrbus = "0x0C06";
        cpu_req = sc_logic_1;
        wait(clk->posedge_event());//instruction
        cpu_req = sc_logic_0;
        Addrbus = "xxxxxxxxxxxxxxx";
        DataBus = "xxxxxxxxxxxxxxx";
        wait(clk->posedge_event());
```

The writing and reading were described in the TLM part.

I.SIMULATION

For simulating this SystemC modeling we must consider to VCD file and console window simultaneously. In the VCD all the signals and timing are obvious but in the console window, the TLM actions with their timing are declared. You can see the TLM details after running the code:

```
CPU side  R, @450 data:0  @time 7 ns delay=0 s
readData =3
CPU side  R, @451 data:3  @time 23 ns delay=0 s
readData =1
CPU side  R, @452 data:1  @time 39 ns delay=0 s
readData =2
CPU side  R, @453 data:2  @time 55 ns delay=0 s
readData =3
CPU side  R, @454 data:3  @time 71 ns delay=0 s
readData =4
CPU side  R, @455 data:4  @time 87 ns delay=0 s
readData =5
CPU side  R, @456 data:5  @time 103 ns delay=0 s
readData =6
CPU side  R, @457 data:6  @time 119 ns delay=0 s
readData =7
CPU side  R, @458 data:7  @time 135 ns delay=0 s
readData =8
CPU side  R, @459 data:8  @time 151 ns delay=0 s
readData =9
CPU side  R, @460 data:9  @time 167 ns delay=0 s
readData =10
CPU side  R, @461 data:10  @time 183 ns delay=0 s
readData =11
CPU side  R, @462 data:11  @time 199 ns delay=0 s
readData =12
CPU side  R, @463 data:12  @time 215 ns delay=0 s
readData =13
CPU side  R, @464 data:13  @time 231 ns delay=0 s
```

Also for watching signals open "memory.vcd" in the VCD viewer.

J.CONCLUSION

SystemC can help increase productivity by offering a more abstract way of defining a system than is available using hardware description languages (HDLs) such as Verilog and VHDL. The abstract nature of SystemC enables systems to be specified independent of their implementation. This gives designers the freedom to explore various architectural approaches and to make trade-offs between software and hardware before committing to an implementation path for each functional block.