ECE #720 **SystemC Transaction Level Modeling Report**

Zhiping Wang | Unity ID: zwang79 | Student ID: 200265085

12/9/2019

1. Introduction

With the demand for less-time-consuming and lower-cost design and verification method, transaction level modeling (TLM) emerges as an abstract or simplified method for register transfer level (RTL) modeling. In this project, I modeled the system shown in Figure 1 with SystemC TLM 2.0 modules. This report demonstrates basic design thoughts and obstacles that I have encountered. Design structure is explained module by module in Section 2. Remaining issue is discussed in Section 3. Section 4 concludes the project.

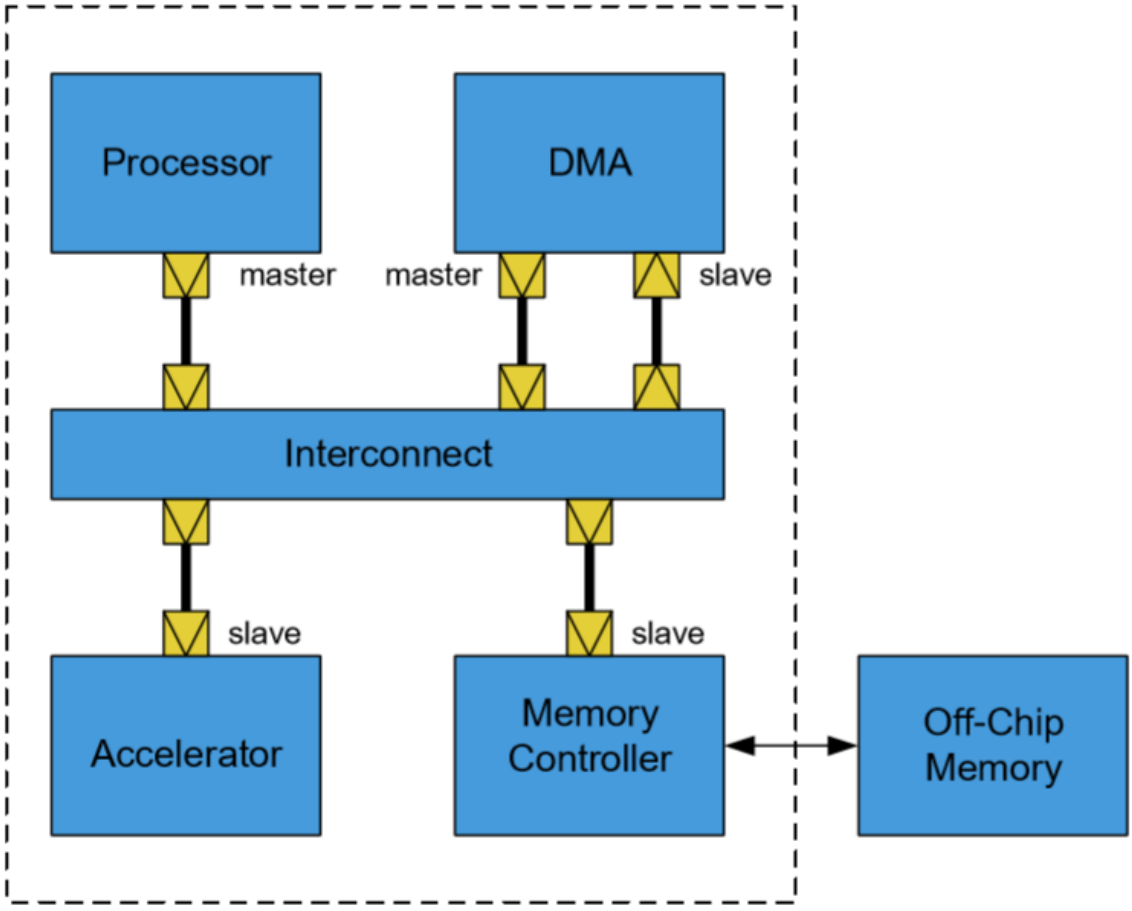

Figure 1. System overview

2. Design Structure

Design consists of 5 modules: Processor, Bus, Memory, DMA, Accelerator. Bus module serves as the interconnect which models Advanced High-performance Bus (AHB). Processor can communicate with the rest 3 modules through Bus module. Typically, Processor writes to or read from the memory, src, dest, length registers in DMA, accum register in Accelerator.

```
private:

SimpleBusLT<2, 3> bus; // AHB bus
mem mem0; // Memory
dma dma0; // DMA
accelerator accelerator0; // Accelerator
stub stub0; // Processor
```

Figure 2. Design structure

2.1. Processor and Bus

Processor and Bus use stub and SimpleBusLT module respectively, I made few changes to them, except for some display improvements. I assume all transactions on Bus is 32-bit wide, which means all READ and WRITE operations initiated by Processor and DMA are in burst fashion and each burst contains 4 bytes of data.

2.2. Memory

Memory module basically remains the same. I added two major features to it: row access recorder and latencies.

2.2.1. Memory Row Access Recorder

Figure 3 demonstrates the address mapping in this design. When Processor and DMA communicate with memory, the row they are accessing could be either off or active, this may cause a difference in terms of latency. As a result, in order to model realistic RTL behavior, it is necessary to monitor which rows are currently active.

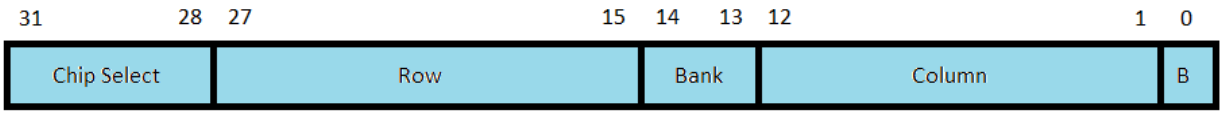

Figure 3. Address Mapping

What I did is to use STL container (list), Figure 4 to store row number that has been accessed before. Whenever a new row is being accessed, program will search in the list to check if this row is currently active or not (Figure 5).

```
list<sc_dt::uint64> row_access_recorder; // monitors which rows are active
```

Figure 4. List for active rows

```
bool mem::row_active(sc_dt::uint64 row)
{

  return (find(row_access_recorder.begin(), row_access_recorder.end(), row) != row_access_recorder.end());

}
```

Figure 5. Search for active row in the list

Since the memory size is limited to 4KB, according to address mapping in Figure 3, all the memory accesses will hit the same row. This makes this recorder redundant because only the first access will hit an inactive row.

2.2.2. Memory Latencies

As I mentioned before, since all the accesses will hit the same row. The only command latency will be RCD. Moreover, burst latency and CAS latency will be introduced (Figure 6). All accesses will wait for CAS and burst latencies, if the row being accessed is inactive, the operation will wait for RCD as well.

```
// since memory size is limited to 4KB, these are the only latency values we need
sc_core::sc_time burst(10,sc_core::SC_NS);
sc_core::sc_time CAS(20,sc_core::SC_NS);
sc_core::sc_time RCD(20,sc_core::SC_NS);
```

Figure 6. Latencies in Memory

2.3. DMA

```
#include "tlm.h"
#include "tlm_utils/simple_target_socket.h"
#include "tlm_utils/simple_initiator_socket.h"

class dma: public sc_core::sc_module
{
  public:

  SC_HAS_PROCESS(dma);

  dma( sc_core::sc_module_name module_name );

  tlm_utils::simple_initiator_socket<dma> dma_initiator;
  tlm_utils::simple_target_socket<dma>  dma_target;

  private:

  unsigned char src[4];
  unsigned char dest[4];
  unsigned char len[4];

  void dma_b_transport
  ( tlm::tlm_generic_payload &gp, sc_core::sc_time &delay );

};
```

In a typical system operation, Processor will first write to DMA, then DMA will then communicate with Memory and Accelerator. This means DMA module should be both target and initiator (Figure 7). As a target, DMA should be able to receive READ and WRITE commands from Processor. Meanwhile, as an initiator, DMA should be able to call b_transport function in order to READ from Memory and WRITE to Accelerator.

Figure 7. dma.h

2.3.1. DMA Design Idea

Since DMA have both initiator and target sockets, my dma.cpp basically imitates SimpleBusLT.h. As is shown in Figure 7, function dma_b_transport responds to READ and WRITE access from Processor. Detailed operations in this function is explained as follows:

1. DMA receives WRITE operation to source and destination registers.
2. When DMA receives a WRITE operation to length register, after responding with a TLM_OK_RESPONSE back to Processor, DMA will initiate b_transport to Memory to READ data from source memory address (Figure 8).

```
12820 ns top.Proc WRITE len:0x4 addr:0x10000008 data:0x00000008
12840 ns top.DMA  is being WRITE len:0x4 addr:0x8 data:0x00000008
Writing to DMA Length Register triggers DMA initiator operations...
12840 ns top.DMA  is READING from Mem len:0x4 addr:0xd00
```

Figure 8. DMA b_transport to Memory

3. After Memory responds with TLM_OK_RESPONSE which indicates that DMA READ finished, DMA will initiate b_transport to Accelerator to WRITE data to destination register. Then it displays a completed message when TLM_OK_RESPONSE is received back from Accelerator (Figure 9).

```
12840 ns top.Mem  is accessing an ACTIVE row (latency = Burst *8 + CAS)
12940 ns top.Mem  READ len:0x20 addr:0xd00 data:0x3c6b56fcbf12ad2ebd66e43b3c9c92de3f35ff3f3d31098dbf3e86833ed880d8
12940 ns top.DMA  READ completed data:0x3c6b56fcbf12ad2ebd66e43b3c9c92de3f35ff3f3d31098dbf3e86833ed880d8
12940 ns top.DMA  is WRITING to Acce len:0x4 addr:0x20000010 data:0x3c6b56fcbf12ad2ebd66e43b3c9c92de3f35ff3f3d31098dbf3e86833ed880d8
13030 ns top.Acce WRITE len:0x20 addr:0x10 data:0x3c6b56fcbf12ad2ebd66e43b3c9c92de3f35ff3f3d31098dbf3e86833ed880d8
13030 ns top.DMA  WRITE completed
```

Figure 9. DMA b_transport to Accelerator

2.3.2. DMA Latencies

I add two kinds of delays in DMA module, as in Figure 10.

dma_delay refers to delay within DMA that is required to satisfy setup and hold constraints for registers. Also, burst delay is included because all the communications are in burst fashion.

```
sc_core::sc_time dma_delay(10,sc_core::SC_NS);
sc_core::sc_time burst(10,sc_core::SC_NS);
```

Figure 10. DMA delays

2.4. Accelerator

Accelerator is similar to Memory module in terms of structure. It contains a target socket that can respond to READ and WRITE requests from Processor and DMA. Accumulation block is also modeled.

2.4.1. Accelerator Design Idea

When Accelerator receives a READ or WRITE request, it will first conduct address decode in order to find out which register is being accessed (Figure 11).

```
address_mask = address & 0xFF;


if (address_mask == 0x00) { //  writing to accumulator
```

Figure 11. address decoder

When multiplier register (MULT) is being written, accumulation operation will be triggered. In order to simulate RTL design, I assume that the accumulation operation will be parallel with MULT WRITE operation. In other words, they are two stages of a pipeline. As is shown in the red box in Figure 12, Accumulation for WEIGHT[7] * MULT[7] starts right after WRITE operation of MUTL[7] is done (at 13310ns). After all WRITE and Accumulation operations are done, Accelerator will send TLM_OK_RESPONSE back to initiator module.

```
13300 ns top.Acce WRITE len:0x20 addr:0x30
DMA Writing to MULT triggers accumulation...
13310 ns top.Acce Write MULT[7] = 0xc29d1108 Accumulating WEIGHT[7] * MULT[7]...
13320 ns top.Acce Write MULT[6] = 0xc18e8bc0 Accumulating WEIGHT[6] * MULT[6]...
13330 ns top.Acce Write MULT[5] = 0x427d12b8 Accumulating WEIGHT[5] * MULT[5]...
13340 ns top.Acce Write MULT[4] = 0x42c9ae6c Accumulating WEIGHT[4] * MULT[4]...
13350 ns top.Acce Write MULT[3] = 0xc29da168 Accumulating WEIGHT[3] * MULT[3]...
13360 ns top.Acce Write MULT[2] = 0xc2b07874 Accumulating WEIGHT[2] * MULT[2]...
13370 ns top.Acce Write MULT[1] = 0x420d85e8 Accumulating WEIGHT[1] * MULT[1]...
13380 ns top.Acce Write MULT[0] = 0x42aee168 Accumulating WEIGHT[0] * MULT[0]...
13385 ns Multiplier writing finished, accumulation done.
```

Figure 12. MULT WRITE and Accumulation

2.4.2. Accelerator Latencies

There are 3 kinds of delays in Accelerator as in Figure 13:

```
sc_core::sc_time accelerator_delay(10,sc_core::SC_NS);
sc_core::sc_time burst(10,sc_core::SC_NS);
sc_core::sc_time accum_block_delay(5,sc_core::SC_NS);
```

Figure 13. Accelerator Delays

accelerator_delay stands for register operation delay. accum_block_delay refers to delay caused by accumulation operation. burst delay is also included.

One thing that needs to be emphasized is that I assume Accumulation delay is less than a clock cycle (burst delay), which means it will not be “visible” except for the last Accumulation operation. This is demonstrated in the blue box in Figure 12.

2.4.2. Accelerator Floating Point Calculation

WEIGHT and MULT are transferred as char, Accumulation, however, is in floating point format. As a result, I need to convert a byte array into single precision floating point number before Accumulation and change it back for the convenience of bus transaction. In order to implement this, I introduced two functions in Accelerator (Figure 14).

```
void accelerator::Float_to_Byte(float f, unsigned char byte[]) {
  FloatLongType fl;
  fl.fdata = f;
  byte[0] = (unsigned char)fl.ldata;
  byte[1] = (unsigned char)(fl.ldata >> 8);
  byte[2] = (unsigned char)(fl.ldata >> 16);
  byte[3] = (unsigned char)(fl.ldata >> 24);
}

void accelerator::Byte_to_Float(float *f, unsigned char byte[]) {
  FloatLongType fl;
  fl.ldata = 0;
  fl.ldata = byte[3];
  fl.ldata=(fl.ldata << 8) | byte[2];
  fl.ldata=(fl.ldata << 8) | byte[1];
  fl.ldata=(fl.ldata << 8) | byte[0];
  *f=fl.fdata;
}
```

Figure 13. Conversion Functions

3. Remaining Issue

Although most of the runs works correctly for my design, I cannot fix one issue.

Look at our validation input file, 8 weights and 8 multipliers are accumulated in the Accelerator. If we change the order of weights and multipliers accordingly (Figure 14), accumulation result is supposed to be the same. However, difference occurs.

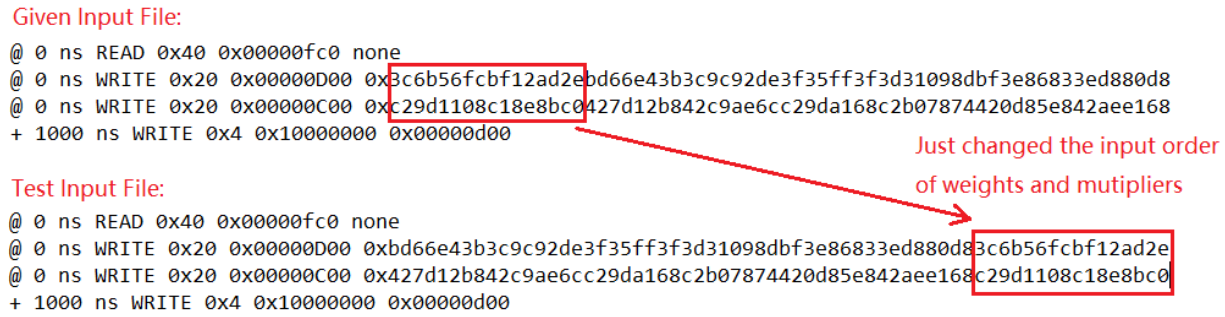

Figure 14. Original and Test Input

I added some debugging outputs in Figure 15 and 16. As we can see, the outputs of different input files indicate same floating-point calculation results (accum_f = -41.7606). However, after conversion, Accumulations differs at last bit. I am not sure where does this come from, but I guess it is because of the floating-point mechanism in C++.

```
w * m = -26.3318
accum_f = -78.7354
accum = 0xc29d7881
13380 ns top.Acce Write MULT[0] = 0x42aee168 Accunmulating

w * m = 36.9748
accum_f = -41.7606
accum = 0xc2270ace
13385 ns Multiplier writing finished, accumulation done.
13385 ns top.DMA  WRITE completed
13405 ns top.Proc READ len:0x4 addr:0x20000000
13425 ns top.Acce READ len:0x4 addr:0x0 data:0xc2270ace
13425 ns top.Proc READ completed data:0xc2270ace
13425 ns top.Proc Completed_
```

Figure 15. Output for Given Input File

```
w * m = -1.12805
accum_f = -51.9696
accum = 0xc24fe0e0
3890 ns top.Acce Write MULT[0] = 0xc18e8bc0 Accunmulatin

w * m = 10.209
accum_f = -41.7606
accum = 0xc2270acf
3895 ns Multiplier writing finished, accumulation done.
3895 ns top.DMA  WRITE completed
3915 ns top.Proc READ len:0x4 addr:0x20000000
3935 ns top.Acce READ len:0x4 addr:0x0 data:0xc2270acf
3935 ns top.Proc READ completed data:0xc2270acf
WARNING: READ data differs from expected data
3935 ns top.Proc Completed _
```

Figure 16. Output for Test Input File

4. Conclusion

The results for validation run are correct.

The most difficult part of this project for me is the floating-point calculation and DMA module.

Even though, we do not have the RTL design to do the comparison, I have the idea that TLM does speeds up the simulation. Since TLM uses function calls instead of signals and wires.