ECSE 541 – Assignment 2

Xiangyun Wang
McGill ID: 260771591, Department of Electrical and Computer Engineering, McGill University

***Abstract*—This Assignment implemented a HW/SW system with a memory and arbitrated bus to perform matrix multiplication. System performance with and without hardware acceleration is investigated.**

I. INTRODUCTION

This assignment is aimed to implement a HW/SW partitioned matrix multiplication application using SystemC, and compare the time consumed for the operations with and without hardware acceleration. The entire system design is shown in Figure 1. The *bus* in the middle is used for all the communications of other components. It consists of a master interface and a minion interface, where a master can make request through the *bus* and a minion constantly listens for requests from the *bus*. Once a request is acknowledged, *bus* would only serve for this pair of master and minion. In our system, *SW* can only be master, *HW* can either be master or minion, and *Memory* can only be minion.

At the start of the multiplication process, *SW* will write appropriate command parameters into the low part of *Memory*, which is reserved for I/O communications only. After this, *SW* will make a request to *HW* for doing the matrix multiplication. Once *HW* acknowledged the request, it will start the operation by reading command parameters from *Memory* (input & output memory addresses and matrix size). After getting all the parameters, it will reset the output memory segment with 0s, and then access the two input matrices from *Memory* and save them locally. With the locally saved matrices, *HW* can do multiplications without further doing the time-consuming memory access operation. The result will be saved back into *Memory* once the multiplication is finished. Meanwhile, *SW* will also be notified.

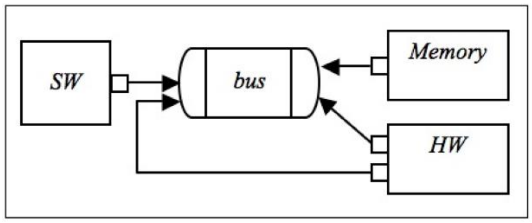

Figure 1 - System Overview

II. SYSTEM IMPLEMENTATION

A. Bus Arbiter

For our system implementation, bus is required to support a round robin arbitration scheme. Therefore, a queue called "*task*" is used to store received requests. A "*current_task*" variable is used to hold the current broadcasting request. If the current request is not acknowledged by a listener within two clock cycles, then this request would be pushed back into the *task* queue, and the next front element in the queue will be popped. However, if the current request is acknowledged, the request would be pushed into another queue called "*ack_task*". This mechanism will be further explained in the *Minion Interface* section.

B. Master Interface

The provided interface skeleton has four functions, which are *Request(), WaitForAcknowledge(), ReadData()* and *WriteData().* For the sake of my own system implementation, I added two extra functions for the interface, *AckFinish()* and *WaitForFinish().*

a) Request (mst_id, addr, op, len)

This function will be called whenever a master wants to communicate with or utilize other components in the system. A "*Task_Info*" struct is created to hold task parameters together. When the function is called, a *Task_Info* object will be constructed with the input parameters, and it will be either passed to *current_task* or pushed into the *task* queue, depending on the number of active requests (requests that are not acknowledged).

b) WaitForAcknowledge (mst_id)

This function will be called by a master immediately after calling *Request()*. It will periodically (1 time / clock cycle) check if the request is acknowledged or not. This is done by checking the front element of the *ack_task* queue. If the front element has the same *mst_id* as the waiting *mst_id*, then it can be known that the request is acknowledged. Then, it will raise the busy flag of bus, save the number of data transfer required and reset the number of actual data transferred. Finally, the acknowledgement result will be returned from the "ack_status" queue. Whenever a request is acknowledged, a flag will be pushed into *ack_status* queue, to indicate if the acknowledgement is true or false. This prevents potential blockings of invalid requests, such as out-of-bound memory access. Further explanation is in the *Acknowledge()* section of Minion Interface.

c) ReadData (&data)

This function is used to access data from the bus data buffer. Whenever there is a request to read from *Memory*, *Memory* will first send the data to the buffer in *bus*, and later redirect these data to the correct user. This function keeps checking the front of the data buffer. If the buffer if empty, wait for another clock cycle; otherwise, read the front data and remove it from the buffer. The number of data transfer done is also recorded, and if the number reached the target transfer number, *bus* will be released.

d) WriteData (data)

This function simply saves write data in the bus data buffer. These data will be later saved into the memory with the *ReceiveWriteData(),* which is further explained in the Minion Interface section.

e) AckFinish(status)

This function is specifically designed for the request made by *SW*. When *SW* asked *HW* to do the multiplication, it needs a way to be notified if the multiplication is finished or not. If the multiplication is finished, HW would call this function with a flag to indicate if the operation is successful or not. This flag would be pushed into a queue buffer called "*finish_status*", and the SW can access this status using *WaitForFinish()*, which will be explained below.

f) WaitForFinish()

This function periodically (1 time per clock cycle) checks the *finish_status* queue. If there is an element in the queue, the status element will be recorded and popped from the queue. In this way, SW can be notified for the finish of the multiplication.

C. Minion Interface

The provided interface skeleton has four functions, which are *Listen()*, *Acknowledge()*, Send*ReadData()* and Receive*WriteData()*. The detailed implementations of them are provided below.

a) Listen (&req_addr, &req_op, &req_len)

This function will be called by a minion. It passes the parameters of the current request to the addresses of the listening parameters.

b) Acknowledge (status)

When a minion called this function, there will be two elements pushed into two different queues. Firstly, current request will be pushed into *ack_task* queue. Then, a flag indicating the acknowledgement result will be pushed into *ack_status* queue. Meanwhile, current request will be changed to an invalid request, where mst_id = 0 and op = NONE are reserved for invalid requests. In this way, this request will not be pushed back into *task* queue for the arbitration.

c) SendReadData (data)

This function simply pushes required data to the bus data buffer whenever a *ReadData()* is called by a master.

d) ReceiveWriteData (&data)

This function is responsible for transferring data from bus data buffer to *Memory*. It periodically checks the data buffer. If there is a data in the front of queue, it will pop the data out and save it to the desired location. Meanwhile, the number of data transfer is updated. If the number of transfers reaches the target, *bus* will be released.

D. Memory

Memory can only be a minion in the entire system. The memory component is initialized in the constructor and a memory access thread is started. The thread listens to the request from bus and determine whether it should acknowledge the request.

It will first determine if the access request is read or write, then checks if the address is within the memory size. After acknowledgement, it will send and receive data according.

E. Hardware

The *HW* component is used to accelerate the matrix multiplication process. There are three blocks of local memories in *HW*, which are used for storing the two input matrices and the output matrix. These local memories dramatically reduced the number of memory access required. Instead of accessing element by element for pure software multiplications, *HW* can do burst transactions to read and store all the data at once.

HW is first in the minion mode, listening to the calculation request from *SW*. Once the request is received, *HW* switches to the master mode, and reads the locations and length of the matrices. Then, the output locations are reset to 0s and the two input matrices are read in and stored in the local memories using burst transactions. With all the input matrices ready, the multiplication can be done with three for-loops. Finally, the output matrix is stored back into *Memory*.

F. Software

The *SW* component is responsible for invoking the matrix multiplication process. There are two functions, *WithHardware()* and *WithoutHardware()*, which can be used according to the simulation requirements. The difference between these two functions is obvious. *WithHardware()* sends the operating command to the memory first, and then makes a calculation request to *HW* to get the hardware acceleration for the matrix multiplication. For the function of *WithoutHardware()*, the input matrices and the output matrix are accessed element by element. For each access, it needs to call *Request()*, *WaitForAcknowledge()* and *ReadData()* or *WriteData()*, which are extremely time consuming processes. A detailed analysis and comparison between the matrix multiplication processes with and without hardware acceleration is provided in the following section.

III. Simulation Result & Analysis

As mentioned in the previous section, two versions of system are implemented, one with the use of hardware acceleration, the other one with purely software operations. The performance improvements are expected from the caches of the *HW* component. Caches greatly reduce the number of memory accesses with burst transactions. With only software, each multiplication between matrix elements requires initiation of a new set of acknowledgement process. According to the assignment instruction, *Request()* needs two clock cycles, and *Acknowledge()* needs one clock cycles. Assume the matrix size is 5. According to the matrix multiplication for-loops provided, there will be 2 memory reads for a single multiplication between elements, and a memory write for every 5 multiplications. For each set of memory access, 3 extra clock cycles will be used compared to *HW* with burst transaction. Therefore, the performance differences per iteration can be calculated as 5^3*3*2+5^2*3 = 825cc. Due to other delays caused by the implementation, the performance delay is expected to be around 1000 clock cycles for each iteration of 5 by 5 matrix multiplication.

All the simulations are the same matrix multiplication looped for 10 times, and the two input matrices are all 1s. As shown in the figures below, Figure 2 and Figure 3 are the

simulation results and time consumed with and without hardware acceleration. The answers are both verified to be correct. The system performance is much better with the hardware acceleration. Each iteration is approximately 1400 cc faster than the system without hardware acceleration. A complete simulation progress with hardware acceleration for one iteration is shown in Figure 4. The requests, acknowledgements and finishes can be clearly seen in the figure.

```
Info: /OSCI/SystemC: Simulation stopped by user.
Golden Answer:
5 5 5 5 5
5 5 5 5 5
5 5 5 5 5
5 5 5 5 5
5 5 5 5 5

Actual Answer:
5 5 5 5 5
5 5 5 5 5
5 5 5 5 5
5 5 5 5 5
5 5 5 5 5

Time Consumed: 12405.426(ns)
Time Consumed: 1861(cc)
xiangyun@Xiangyuns-MacBook-Pro A2 %
```

Figure 2 - Simulation with HW Acceleration

```
Info: /OSCI/SystemC: Simulation stopped by user.
Golden Answer:
5 5 5 5 5
5 5 5 5 5
5 5 5 5 5
5 5 5 5 5
5 5 5 5 5

Actual Answer:
5 5 5 5 5
5 5 5 5 5
5 5 5 5 5
5 5 5 5 5
5 5 5 5 5

Time Consumed: 108529.15(ns)
Time Consumed: 16281(cc)
xiangyun@Xiangyuns-MacBook-Pro A2 %
```

Figure 3 - Simulation without Hardware Acceleration

```
xiangyun@Xiangyuns-MacBook-Pro A2 % ./test mem_init_A2.txt

        SystemC 2.3.2-Accellera --- Nov 17 2020 04:32:52
        Copyright (c) 1996-2017 by all Contributors,
        ALL RIGHTS RESERVED
Initializing memory of size 65536
System Start!
==================LOOP 0==================
New Request ID = 1
Software: Request Write to Memory
Memory: Access Acked
Software: Request Acknowledged
New Request ID = 2
Hardware: Received Requested to Calculate
Hardware: Acked to Calculate
Hardware: Request to Access Command
New Request ID = 3
Memory: Access Acked
Hardware: Acked to Access Command
Hardware: Request to Reset Memory
New Request ID = 4
Memory: Access Acked
Hardware: Acked to Reset Memory
Hardware: Request to Access Matrix A
New Request ID = 5
Memory: Access Acked
Hardware: Acked to Access Matrix A
Hardware: Request to Access Matrix B
New Request ID = 6
Memory: Access Acked
Hardware: Acked to Access Matrix B
Hardware: Request to Access Matrix C
New Request ID = 7
Memory: Access Acked
Hardware: Ack to Access Matrix C
Hardware: Task Finished
Software: Task Finished
```

Figure 4 - Simulation Process with HW Acceleration