ECSE 541: Assignment 2 Report

Negin Firouzian
Department of Electrical and Computer Engineering
McGill University
negin.firouzian@mail.mcgill.ca

I. Bus Implementation

The bus's primary responsibility is to control the data flow from incoming requests of masters and deliver them accurately to the target minions while keeping fairness in arbitration. The architecture of the bus consist of the methods that are inherited from *bus_master_if* and *bus_minion_if*, in addition to an arbiter function that controls the master's access to the bus. Bus ports can be used by multiple master threads, therefore an arbitration scheme should be used to manage the requests coming to the bus. For synchronizing each of these threads, *sc_event* variables are used. A thread can be triggered or resumed on an event's occurrence, or it can notify an event to trigger other waiting threads.

Since each request coming from the masters has some attributes including, master ID, address, option and length of burst transaction, a new data type, called *Bus_request*, is defined, which is an object consisting of the aforementioned attributes, in addition to a counter that keeps track of burst reads/writes until the current moment. This scheme facilitates the process of arbitration and keeping the status of requests in each stage.

A. Bus Arbitration

For implementing the Round Rubin scheduling, std::queue class is used. Each incoming request is stored as a *Bus_request* object and pushed into the queue. The bus arbiter checks whether the queue is empty or not, and in the latter case, the first request in the queue will be popped out and set as the current request. Consequently, the *Listen* thread will be triggered, and the current request's information will be sent to the minions. If the target minion responds to the request by calling *Acknowledge*, then the *WaitForAcknowledge* thread will be triggered and returns true in the master interface. In that stage, the master is authorized to call the *ReadData* or *WriteData* methods of the interface.

For reading transactions, when the data is ready by minion, the *SendReadData* thread in the minion interface notifies the *ReadData* in the master interface and waits for *data_valid* event from *ReadData* in order to read the next data (in case of burst). Additionally, the *ReadData* thread keeps track of the number of reading transactions and declares the end of the transaction by notifying the *busUnlock* event. In writing transaction, the *WriteData* thread in master interface notifies the *ReceiveWriteData* in the minion interface, when writes a valid data on the buffer, and then waits for the receive event from the *ReceiveWriteData* thread in order to write the next data (in case of burst). This time *ReceiveWriteData* thread keeps track of burst transactions by incrementing a counter, and declares the end of the transaction by notifying the *busUnlock* event.

Meanwhile, new requests may be called by other masters, and they will be pushed in the queue. However, to stop interruption of the current request, the bus arbiter does not proceed to the next loop to pop a new request until the bus is unlocked; in fact, the arbiter waits for the *busUnlock* event to be triggered. The *busUnlock* is triggered when the transaction of the current request, whether burst or single, is finished and the bus is released from master. This event will be asserted at the end of *ReceiveWriteData* and *ReadData* when the counter attribute of the request is equal to the length of the request, which means the burst or single transaction is finished. When the bus transitions to unlock state again, the first request in the queue would be served.

II. Performance Modeling

To implement the approximate time/cycle accurate system model, I have used some wait statements in different bus module methods. These timing annotations are fine-grained and placed in *arbiter*, *ReceiveWriteData*, *ReadData* and *Acknowledge* methods of bus module. In addition, all of the modules are connected to a clock operating at 150 MHz.

For profiling the performance of this system, I have calculated the cycle counts of each loop. This calculation is done by capturing the time stamp before and after the target loop and subtracting the captured times to compute the total elapsed time in each loop. By dividing the total elapsed time by the clock cycle width, the total number of cycles will be calculated. For the final report of system performance, the mean of each loop cycles is calculated over the 1000 iterations of the outermost loop in order to have more accurate estimations.

III. Results

After time annotating and completing the design, it is necessary to test the performance of the model and compare it against other possible options; in other words, perform the design space exploration to find the optimal design decisions. In this project, three main design models were compared together in terms of elapsed clock cycles. In the following sections, each of these models will be described briefly.

TABLE I: Results of profiling three proposed models.

Models	Total HW/SW cycles	Init i loop	Init j loop	Mul i loop	Mul j loop	Mul k loop	SW waiting for HW
Pipelined HW/SW Partitioned	228865	4333	666	44000	7332	6200	180531
Simple HW/SW Partitioned	400329	4333	666	395996	65999	9889	10444 (36 execs)
SW only	572276	4333	666	567941	94656	15775	-

A. Software only

This model is used as a simple baseline to compare against other accelerated choices. All of the calculations and operations were performed purely on software, and one cycle delay is considered for each operation.

B. Simple Hardware/Software Partitioned

In this model, initialization loops is performed by software, and burst transaction is used to set the matrix cell to zero. In the calculation loop for each i and j, a request is made to hardware to calculate the c[i][j]. This request is a burst transaction with size of 2 and sends the address of a and b to the hardware. Hardware then starts multiplying the 1*k matrices of a and b in parallel and writes the results in the memory. After calculating each cell of the c matrix, hardware sets the *done_flag* to inform the software that the calculation is done. Since the software is a master, cannot listen to hardware; therefore, software should check the *done_flag* constantly to acknowledge the end of calculation of c[i][j] cell. When the calculation of c[i][j] cell is finished, the software proceeds to the following loop and requests the calculation of the next cell in the c matrix.

C. Pipelined Hardware/Software Partitioned

This model is pretty similar to the previous model. The only difference is that software continuously sends the calculation request to hardware and does not wait for the hardware calculations in the i and j loops. Instead, after the end of i and j loops and after sending all the calculation requests, the software starts checking the *done_flag*, which this time indicates the end of calculation for the whole matrix. To implement this model, a queue object is used in hardware to buffer the incoming requests. In that way, hardware minion listens to requests from software, and as long as the first request comes, hardware master will be notified and starts the calculations, while at the same time hardware minion is listening to the upcoming requests and saving them in the hardware queue. This approach will result in some levels of parallelism. As mentioned above, after sending the continuous calculation request, software should constantly check the *done_flag* to see if the calculation of matrix c is finished. This results in sending constant requests and increased traffic in the bus, while at the same time hardware is accessing the bus to read/write values on/from memory. To decrease the bus traffic and increase the performance of the system, specific intervals were set, and software checks the *done_flag* in these intervals. Choosing an optimal value for the intervals was a design decision. For finding the optimal interval value in design space, human-performed Local Search algorithm was used! The result of this exploration is available in Table II. According to Table II, the optimal decision for this model is intervals of 15 clock cycles for checking the *done_flag*.

TABLE II: Design space exploration results for optimal intervals of checking the *done_flag*.

Intervals	Total HW/SW cycles	Mul k loop	SW waits for HW
No interval	361441	9889	313107
5 cycles	243664	6617	195331
10 cycles	238309	6308	189974
14 cycles	230310	6216	181975
15 cycles	228865	6200	180531
16 cycles	238864	6200	190530
17 cycles	234754	6185	186419
20 cycles	231976	6139	183642
30 cycles	232310	6093	183975
40 cycles	230976	6062	182642
50 cycles	239975	6046	191641
100 cycles	272083	6015	223749
1000 cycles	789476	5985	741141

Table I shows the results of profiling these three models. Compared to the baseline model, Simple HW/SW Partitioned resulted in a significant improvement in the system's performance. This performance improvement is because of partitioning the multiplication loop into the hardware, which calculates six parallel multiplications in each operation. In this model, the most time was spent in the k loop and consequently in i and j loops of multiplication as well as the time spent by the software when waits for hardware to finish the calculation. Checking the *done_flag* takes 10444 cycles for calculating each cell of the c matrix, which will be executed 36 times for the whole matrix. The pipelined model performed better than the other two models and is considered as a final best model. In this model, both i and j multiplication loops were performed relatively fast compared to the last models. This is because the software did not wait for hardware in the loops and sends the requests continuously. The time spent on checking the *done_flag* is the most significant in this model. However, this time, checking the flag is done only once compared to the last model that checks the flag for each of 36 cells of the c matrix.

It can be concluded that partitioning the multiplication in hardware and adding a buffer at hardware to form some kind of pipeline increases the parallelism and consequently boosts the system performance.