# Final Hardware Design Project

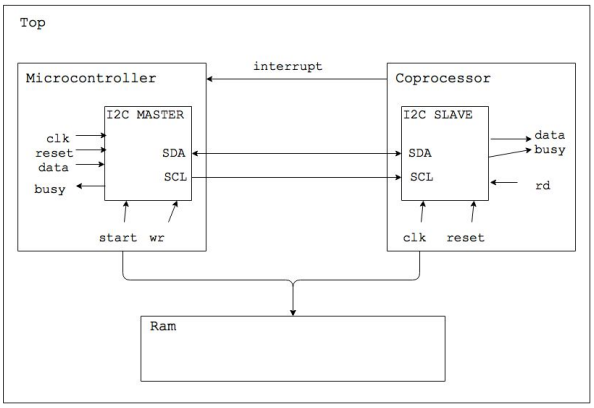
Sandro Yu, *Student, UC San Diego* and Kevin Thai, Student, UC San Diego

## I. INTRODUCTION

The purpose of our project was to create a hardware implementation of two connected processors. The first processors is a microcontroller that contains the main function and will mimic a processor receiving a function call from a program. The second processor will be an application specific coprocessor that implements a matrix multiplication of two large matrices. The two processors are connected using an inter-integrated circuit serial interface where the master I2C is on the microcontroller and the slave I2C is on the coprocessor. Both processors also have access to a shared random-access memory. When a function call occurs, the microcontroller will store both input matrices in the RAM and pass the addresses of the storage locations through the I2C bus to the coprocessor along with the location of where the resulting matrix should be stored in RAM and number of columns in the second matrix. The first matrix passed is expected to be of size 1080x1920 while the second matrix is of size 1920xP where P can be any positive value up to 1080.

## II. IMPLEMENTATION DESCRIPTION WITH DIAGRAMS

Our design is a system that contains many modules connected together. At the highest level is the top module. This module has no inputs and outputs. Within the top module, there is a microcontroller module and coprocessor module. These modules can communicate with each other through an I2C bus, which contains two wires, SDA and SCL. There also resides a shared memory ram module to store the matrices. Within the microcontroller, there is an I2C master module. Within the coprocessor, there is an I2C slave module. All of these modules work together to perform the task of matrix multiplication.




### I2C COMPONENT DESCRIPTION WITH DIAGRAMS

#### A. Description

The communication portion of this hardware implementation consisted of an I2C bus. The I2C bus was implemented in this design using two modules, an I2C master and an I2C slave. In the process of designing the I2C interface, we knew that master only had to talk to one slave device, the coprocessor. Therefore, we decided not to implement the ability for the master I2C to specify a slave address. Additionally, we also knew that we wanted the master I2C to write to a single register in the slave device. We therefore decided to also not implement the ability to specify a memory address within the slave device. This allows the I2C to transfer data sooner. We also knew that the slave only had to receive data, so we decided not to implement the ability for the slave to send data for the master device.

.

B. Design and Implementation

Both the master and slave I2Cs were implemented with a finite state machine. Additionally, both devices are driven by the same clock that originates from the top module. This clock is used to generate two other clocks, bus_clk and data_clk, which will be used to run the master and slave I2Cs.

The master I2C contains a total of five states: IDLE, START_WRITE, WRITE_DATA, ACK1, and STOP. At system reset, the master begins at the IDLE state. Once the microcontroller asserts both the start and wr signals, the master I2C will go to the START_WRITE state. This is where is initiates the start sequence by bringing SDA down while SCL is high. At the next cycle, the master begins writing the data that is present in its data register. This stage takes multiple cycles in order to write all data. After eight bits of data have been written to the slave device, the master I2C goes to the ACK1 state. If there is still more data to be written, the master I2C goes back to the WRITE_DATA state. This loop continues until all data has been written, which will cause the master I2C to go to the STOP state. During the stop state, the master will initiate the stop sequence, and then return to the IDLE state at the next clock cycle.

The slave I2C contains a total of three states: IDLE, RECEIVE_DATA, and ACK1. At system reset, the slave I2C starts at the IDLE state. This is where it is constantly checking for the start sequence sent by the master I2C. Sampling registers clocked by the bus_clk are used to sample the values of SCL and SDA to check for either a start or stop sequence. Once the start sequence is detected, the slave I2C will go to the RECEIVE_DATA state. At this state, it will sample the SDA signal and write the value to the data register. At this state, it is also checking for the stop condition and will return to the IDLE state if it is detected. Otherwise, the slave I2C will then go the ACK1 state, where it will assert a logic '0' through the SDA to the master I2C. Next, it will return to the RECEIVE_DATA state. This loop continues until the slave I2C detects the stop condition while in the RECEIVE_DATA state, during which it will return to the IDLE state.

## MATRIX MULTIPLY COPROCESSOR WITH DIAGRAMS

A. Description

The matrix multiply coprocessor multiplies two given matrices and stores the result in RAM. The first matrix must be of size 1080x1920 and the second matrix must be of size 1920xP where P can be any positive number up to 1080. The coprocessor requires four arguments which it will receive through its I2C bus: the address of the first matrix, the address of the second matrix and the address of where to store the resulting matrix, and the value of P.

B. Implementation and Desgn

Our implementation of the coprocessor is based on the assumption that the circuit size is irrelevant. This allowed us to explore certain ideas that would normally not be considered due to sizing considerations. We started our design with the following algorithm that is normally used in a software implementation of a x by y matrix A and y by z matrix B:

```
For i =  1 to x
  For j = 1 to z
    Sum = 0
    For k = 1 to y
      Sum = sum + A_ik * B_kj
    C[i][j] = sum
```

From this we "unravelled" the loops as much as possible while keeping in consideration that z (which is our P) is not a constant. The result was

For j = 1 to z

$C_{1j} = A_{11}B_{1j} + A_{12}B_{2j} + \ldots + A_{1y}B_{yj}$

$C_{2j} = A_{21}B_{1j} + A_{22}B_{2j} + \ldots + A_{2y}B_{yj}$

…

$C_{xj} = A_{x1}B_{1j} + A_{x2}B_{2j} + \ldots + A_{xy}B_{yj}$

The hardware implementation of this algorithm would require the use of x*y multipliers and x*y-x adders and it would take z clock cycles to complete the calculations. For our specific implementation, this would take 2,073,600 multipliers and 2,072,520 adders with a runtime of P cycles. Since we did not have to consider any physical size restrictions, this number of multipliers and adders is acceptable. However, the critical path of this circuit would have to go through one multiplier, 2,072,520 adders, and the storage element. This would result in a significantly increased clock period. Since the clock for the the coprocess is shared with the entire circuit entity, this would slow down the entire circuit as a whole and the improved number of clock cycles would not be worth the increase in clock period, especially if more functions were to be added to the circuit.

We decided to "re-ravell" one of the loop, which resulted in the following algorithm:

For j = 1 to z
$C_{1j} = 0$
…
$C_{xj} = 0$
For k = 1 to y
$C_{1j} = C_{1j} + A_{1k} B_{kj}$
$C_{2j} = C_{2j} + A_{2k} B_{kj}$
…
$C_{xj} = C_{1j} A_{xk} B_{kj}$

The hardware implementation of this new algorithm would require x multipliers and x adders and it would take y*z clock cycles to complete. This translates to 1080 multipliers and 1920 adders with a runtime of 1920*P clock cycles for our implementation. We chose to implement this specific algorithm since its critical path only goes through one multiplier, one adder, and one storage unit, which should decrease the clock period by a significant amount; thus, it will not negatively affect the rest of the circuit.

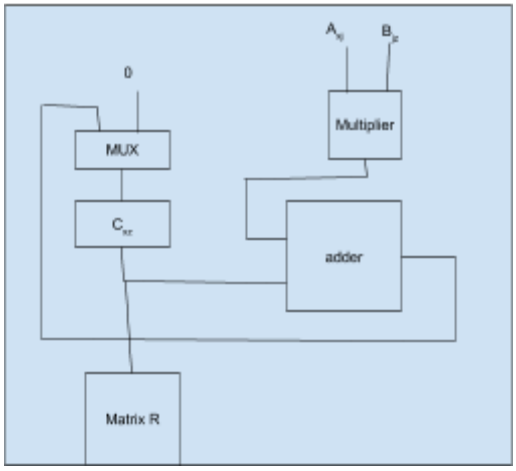



Fig 1. Example of the data path for calculation $C_{xz}$

Other than the multiplication algorithm, we also had to how to retrieve data from and store data to the RAM as well as how much of the data to store in the coprocessor at any given time. We decided to allocate space on the coprocessor to store all three matrices since there was no size restrictions as stated previously. This would minimize the amount of loading from RAM we would have to do since we only have to load each matrix element once whereas not store entire matrices would require multiple loads.

For our state machine, we chose to load all the matrix values for both matrices before performing any multiplication. It also waits until all calculations are done before storing the resulting matrix into the hardware implementation should run faster compared to software at the cost of less flexibility and more RAM. This implementation has a runtime of 1920*P cycles for calculations, 1080*1920+1920*P+1 cycles for reading from RAM, and 1080*P cycles for writing to RAM.

C. Miscellaneous

Our implementation does not check for valid RAM addresses so incorrect addresses (such as those where memory spaces overlap) may result in unknown behavior. In addition, the RAM addressing require a 23 bit input address but our I2C bus only transfers 8 bits, thus our implementation assumes that only the 8 most significant bits are transferred and will pad the 15 least significant bits with zeros when accessing memory.

D. Testing

Unfortunately, due to our liberal use of physical components, we were unable to run our coprocessor implementation on Modelsim. Every attempt returns an error indicating that it is unable to allocated the necessary amount of memory for the implementation.

## III. PERFORMANCE COMPARISON OF HARDWARE-ASSISTED VS. SOFTWARE ONLY

During this project, we also wrote a program in Java to run the matrix multiplication algorithm on two matrices of size 1080x1920 and 1920xP. In the hardware implementation, we used the value 3 for P. After running the software implementation with the value 3 for P, we got the following time results:

| | |
|---|---|
| Real | 0m 0.362s |
| User | 0m 0.281s |
| Sys | 0m 0.039s |

Unfortunately, we were not able to get any performance information from the hardware implementation because we were not able to run our our coprocessor implementation on Modelsim.

## IV. Conclusion

We were able to successfully implement two circuits connected through an I2C bus. From the I2C implementation, we learned how to utilize a two wires to send data between two entities. Being able to communicate using a small amount of wires provides designers with more flexibility with part placement as it will use up less materials and will be easier to position the wires on a circuit.

In theory, our coprocessor should correctly implement matrix multiplication. By implementing an application specific processor, we were able to compare how a hardware implementation compares with a software implementation. We analyzed that a

physical space