Accelerated Matrix Processor[AMP00] Memory Subsystem

William Wong
Purdue SoCET
Purdue University
West Lafayette, United States
wong371@purdue.edu

Varun Vaidyanathan
Purdue SoCET
Purdue University
West Lafayette, United States
vvaidya@purdue.edu

Yue Yin
Purdue SoCET
Purdue University
West Lafayette, United States
yin230@purdue.edu

***Abstract*—We have designed a memory subsystem for a systolic array, serving as the central hub for interconnectivity between the instruction set architecture (ISA), systolic array, and scheduling subsystems. Our approach involves developing an architecture that includes an instruction cache (I-cache) for fetching instructions, a data cache (D-cache) for scalar values, a memory arbiter to manage data flow, and a scratchpad to house memory banks and handle outputs to the systolic array. To validate our design, we will test it using software-controlled main memory, using C++ and utilizing Verilator's DPI-C interface to connect this main memory with our Verilog modules. The I-cache and D-cache are made to be reconfigurable depending on any necessary block organization. The I-cache will be optimized to handle instruction retrieval, while the D-cache will be configured to store any scalar values parsed from the scheduler core. The memory arbiter will direct the flow of data by giving first the scratchpad, then the D-cache, and finally the I-cache priority to main memory. This scratchpad will serve as an intermediary, providing two buses to transmit matrix inputs and weights into a multiplexer feeding into the systolic array. The scratchpad is implemented as a split transaction software-managed cache to be able to handle multiple requests from different sources simultaneously.**

***Index Terms*—Scratchpad, Caches, Banking, GEMM, Memory.**

I. Background

Neural networks, a subset of artificial intelligence (AI), have become more prevalent recently due to rapid advancements in the field. A neural network is a computer system that replicates the functions of a human brain such as learning, recognizing, and problem-solving. An example of an advancement would be the use of a neural network to recognize and classify numbers from handwritten digits. However, these operations are computationally expensive and need specialized hardware because the process includes performing a large number of matrix functions. Out of the many available architectures for artificial intelligence hardware acceleration, a systolic array is one of the best architectures for matrix tasks because it exploits data reuse when conducting general-purpose matrix multiplication (GEMM) operations. These matrices would need to be stored somewhere because the operation will consistently reuse previous matrices; otherwise, the system will constantly be recalculating the matrices that were already computed previously. The general architecture of the AMP00 will consist of a scheduler core that acts like a datapath to parse RISC-V instructions and to queue them up, the systolic array for matrix computations, and the memory subsystem, which will store results of computations.

II. Problem Definition

One of the main issues with memory in computer architecture is the cost. If the capacity of memory gets increased, then the cost of accessing and storing also increases. This may seem fine at first, but not for AI workloads which consist of computations that require large transactions at a fast rate. Additionally, a system would consist of multiple systolic arrays for AI workloads, causing the cost of memory transactions to increase exponentially. We would need a design that will enable us to simulate main memory that is able to store these matrices needed for AI workloads, as well as, intermediate modules to handle these memory transactions for matrices, scalar numbers, and instructions.

III. Proposed Architecture

To combat this giant cost of using memory for AI hardware architecture, we propose caches as an intermediary between the datapath and main memory and a software-controlled cache as an intermediary between the systolic and main memory. These intermediaries all exploit the idea of locality in order to significantly reduce the cost of transactions from/to main memory. Figure 1 shows an example of an address

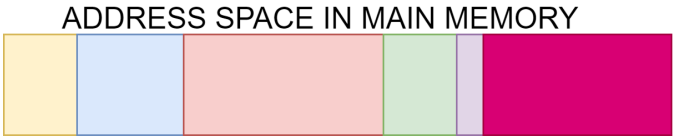

Fig. 1. Example of Address Space in Main Memory

space in main memory, each section being separated by a different color. In this context, each colored slice represents which parts of the memory a datapath would use during a certain moment of time. The intermediaries that we create will exploit two types of localities as described in Figure 2 by knowing the fact that each address space in main memory will store computations and instructions that are related to each other. The main idea for exploiting locality is that a datapath

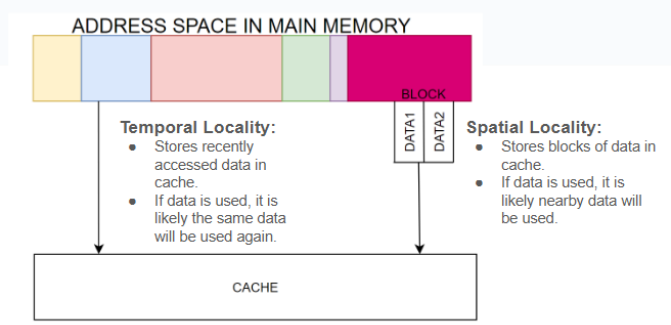

Fig. 2. Locality Examples

would not need to do memory transactions with the entire main memory causing the system to go through unnecessary parts of memory instead of the sections we want. Thus, we will create an I-cache to handle the slices in memory for instructions, a D-cache to handle the slices in memory for scalar values, and a scratchpad to handle the slices in memory where we store the matrices. In addition to the I-cache, D-

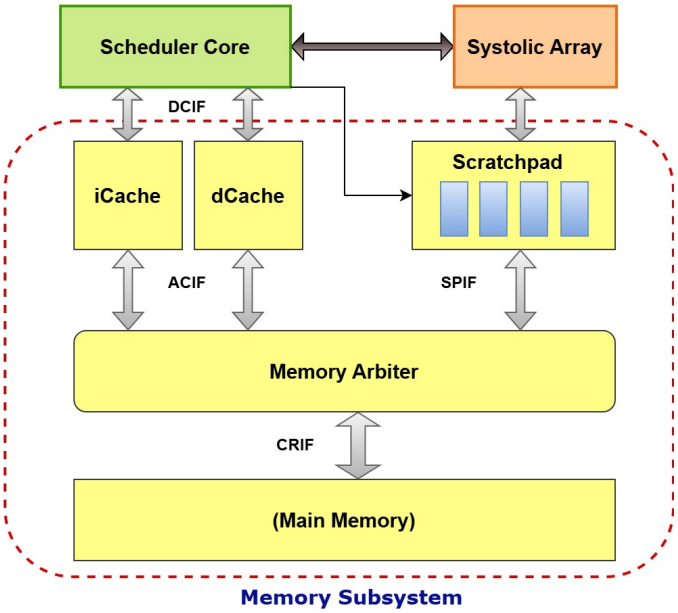

Fig. 3. AMP00 Top-Level Diagram

cache, and the scratchpad, we need to create a memory arbiter to handle which out of the three intermediaries gets access to main memory because in our initial design for AMP00, memory accesses are sequential and can't be accessed by multiple channels. Our overall top-level diagram can be seen in Figure 3 where ACIF, SPIF, and CRIF are just arbitrarily named Verilog interfaces that connect the signals between each module. Another benefit that comes from these intermediaries is the placement of these modules. In addition with locality, by placing the caches closer to the scheduler core and the scratchpad closer to the systolic array, the memory transactions will be faster as seen in Figure 4. Although we lose storage

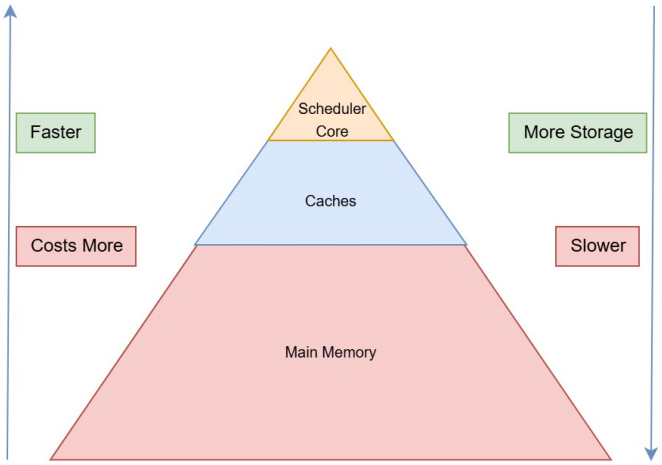

Fig. 4. Memory Hierarchy Diagram

the higher we go up in the hierarchy, that is leveraged by the fact that the address space in main memory is in chunks and those benefits were described in the locality example above.

IV. Design Constraints and Opportunities

A. Caches and Memory Arbiter

The main constraint for arbitrating the memory accesses between the scratchpad and caches is the inherent blocking architecture of our design for the initial iteration of AMP. Because there isn't any coherence between the modules, we can't access main memory at the same time using multiple modules. However, what we give up in functionality and throughput, we make up in convenience and area since we wouldn't need a state machine to track coherence and less registers to store those signals for coherence.

B. Main Memory

The major constraint for memory is that it will be too slow if we write the memory as a huge register file in system verilog. The time to simulate and synthesize will be significant. Since our plan is to synthesize in FPGA before taping out, it would be impossible to synthesize such a huge main memory in FPGA. There is not enough transistors in FPGA to do such a synthesis. So it is an area constraint. Another constraint we are facing is that the assembler written by the ISA team is not working so we have to figure a way to generate the meminit.hex file for the main memory.

C. Scratchpad

One of the most critical constraints in the scratchpad design is ensuring fast and efficient data transfer to the systolic array. Since the systolic array takes a large number of cycles to compute a single matrix multiplication, any fixed-cycle overhead introduced by the scratchpad must be minimized to avoid becoming a performance bottleneck.

Each matrix tile processed by the array is a 4×4 block occupying 32 bytes. To maintain high throughput, the scratchpad

must hold many such tiles simultaneously, enabling quick, low-latency access to the required data during computation. This requirement drives the need for a relatively large scratchpad size.

To optimize for capacity and access efficiency, the scratchpad is implemented as a software-controlled, untagged cache, effectively functioning as a matrix register file. Unlike hardware-managed caches, which require tag arrays to track data validity and location, the software-controlled design eliminates this overhead. Tag arrays can consume significant memory, which is especially costly in large SRAM banks. By removing them, we maximize the amount of SRAM available for actual matrix data, aligning with the goal of supporting high-throughput matrix operations.

This approach trades automatic data movement and coherence management for programmer or compiler-level control, which is acceptable in this context due to the predictable access patterns of matrix multiplication and the performance-critical nature of the systolic array.

V. Solution Overview

A. Instruction Cache

The instruction cache is one of the two caches we have in our design that sits in between the scheduler core and main memory. It mainly exploits temporal locality by grabbing the instruction that was least recently used and storing it inside of itself using a tagging system. As mentioned above, to utilize the fast speed of a cache we would want the cache to be smaller than the main memory, hence the tagging system and only replacing instructions stored in the I-cache that were least recently used. If an instruction is found inside of the I-cache, then grabbing that instruction would be quick. However, when an instruction is seen for the first time, then it would take the whole time of grabbing the instruction from main memory and then grabbing the instruction from the I-cache. It's best to remember that in our workload with the systolic array, matrix operations are looped, and seeing repeat instructions will overshadow first-time instructions.

B. Data Cache

The instruction cache is the second of the two caches we have in our design that sits in between the scheduler core and main memory. It exploits both temporal locality by grabbing the scalar data that was least recently used and storing it inside of itself using a tagging system and spatial locality to grab scalar data that is next to the one we already grabbed. Likewise with the I-cache, to utilize the fast speed of a cache we would want the cache to be smaller than the main memory, hence the tagging system and only replacing instructions stored in the D-cache that were least recently used. Also, if it is the first time the D-cache has seen scalar data, it would wait the whole time of fetching the scalar data from main memory and then from D-cache to the scheduler core. However, we are further limiting this occurrence with the addition of spatial locality because we are leveraging the idea that the instruction assembly files that we received have stored consecutive scalar computations next to each other.

C. Memory Arbiter

The memory arbiter's goal is to limit who gets access to main memory because we lack any coherence in our design because of our constraints. The memory arbiter will have a priority system where it will first give access to scratchpad operations, then to D-cache operations, and then to I-cache operations. We would want our priority in this order because scratchpad operations are the deepest in the instruction pipe and without this priority, we are faced with a memory deadlock, where the next scratchpad requests will constantly have to wait for any of the caches to finish and potentially never executing.

D. Scratchpad

The scratchpad memory plays a crucial role in enhancing performance by providing low-latency, high-bandwidth access to frequently used data. Unlike traditional cache hierarchies, scratchpads are software-managed, allowing more deterministic control over data movement. This makes them especially effective in systolic array architectures, where predictable and repeated access patterns benefit from fast, localized storage. Scratchpads provide fast access to matrix tiles which means that the processor won't need to access main memory as often. By minimizing off-chip memory accesses, scratchpads contribute to improved overall efficiency in neural network computations. Given that the systolic array requires numerous cycles to complete a matrix multiplication, it is essential that the scratchpad provides a steady stream of data without introducing latency that could bottleneck performance.

The scratchpad is designed as a software-controlled, untagged memory array rather than a traditional hardware-managed cache. This design choice avoids the need for tag arrays, which are costly in terms of hardware resources and SRAM space. By eliminating tag storage, a greater portion of the available SRAM can be dedicated to actual matrix data, significantly improving effective storage capacity. Since data access patterns in matrix operations are known and predictable, software management of memory placement is both feasible and efficient in this context.

The scratchpad stores up to 64 4×4 matrix tiles, each occupying 32 bytes, resulting in a total scratchpad size of 2 KB. These matrices are organized across four banks, with each bank holding 16 matrices, as shown in Figure 5. This banking structure allows for up to four simultaneous memory reads, while also enabling concurrent read and write operations to support parallel data movement and computation. During a matrix multiplication instruction, the compute unit may require up to three matrices simultaneously, for the input, weights, and partial sums, while a DRAM store operation may require one additional matrix, making multi-bank access crucial to avoid contention. These bank accesses are determined by the programmer which gives them the ability to maximize the parallelization.

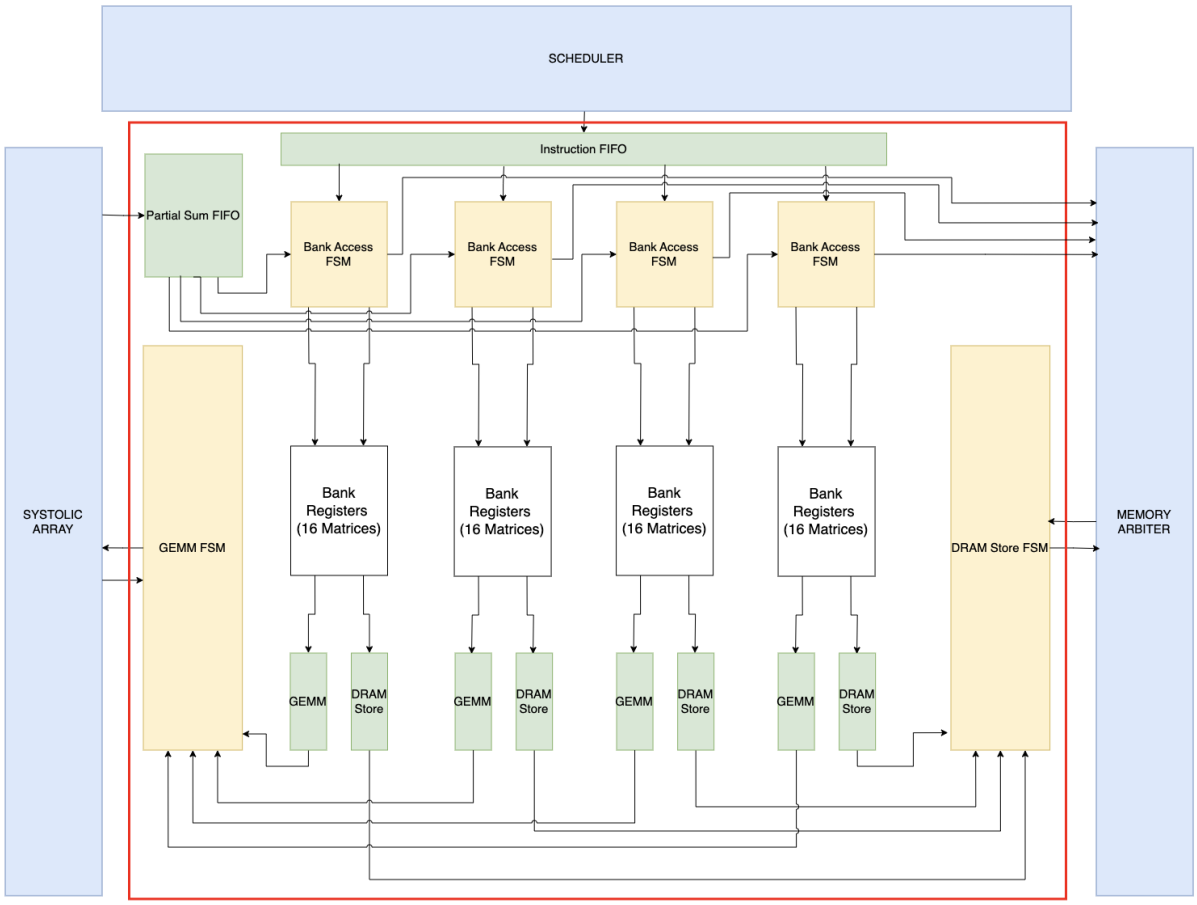

Fig. 5. Scratchpad Architecture Diagram. FSM modules are colored in yellow and FIFO modules are colored in green.

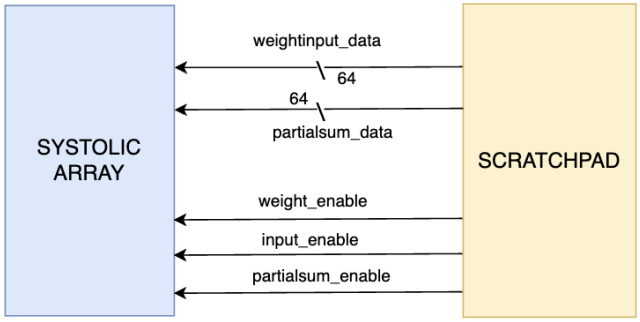

Fig. 6. Scratchpad to Systolic Array Interface

Matrix data is transferred between the scratchpad and the systolic array row by row. Since each 4×4 matrix consists of four rows, it takes four cycles to fully send or receive a matrix. To support this operation efficiently, the architecture uses two 64-bit buses connecting the scratchpad to the systolic array. In the common case (when the weight matrix is already pre-loaded into the systolic array), these buses are used to send input and partial sum matrices simultaneously. In a less frequent scenario, both the weight and input matrices may need to share the same bus. This multiplexing introduces a minor performance trade-off but results in a significant area savings, making it a worthwhile optimization in terms of silicon footprint.

Overall, this scratchpad design balances speed, capacity, and hardware efficiency, ensuring that the systolic array remains well-fed with data while maintaining a compact and area-efficient memory subsystem.

E. Main Memory

Our high level solution for main memory is to use the DPI-C interface that allows the interchange between Verilog and C++ to create a C++ file-based main memory and can read and write from meminit.hex that should be created by the assembler.

VI. Technical Implementation Details

A. Cache Organization

The I-cache and D-cache starts out with 1kb in size. The number of frames is the cache is the cache size divided by

the block size which is a word (32 bits). For a directly-mapped cache, the number of sets is the same as the number of frames. For a 2-way set associative cache, the number of sets is the number of frames divided by the associativity. Now,

$$\begin{aligned}
&\text{CS} = \text{Cache Size}\\
&\text{A} = \text{Associativity}\\
&\text{BS} = \text{Block Size}\\
&\text{Frames (F)} = \frac{\text{CS}}{\text{BS}}\\
&\text{Sets (N)} = \frac{\text{F}}{\text{A}}\\
&\text{Address Bits (AB)} = 32\,\text{bits}\\
&\text{Block Offset (BO)} = \log_2(\text{BS})\\
&\text{Index Bits (IB)} = \log_2(\text{N})\\
&\text{Tag Bits (TB)} = \text{AB} - \text{IB} - \text{BO}
\end{aligned}$$

Fig. 7. Cache Organization Calculations

to parse an address that the scheduler core gives to us we need to categorize the tag bits, index bits, block offset, and byte offset for the I-cache and the D-cache. We know that the dependencies of the caches are from its size and its association. Using the equations from Figure 7, we are able to make an I-cache and D-cache suitable for any size.

B. Instruction Cache

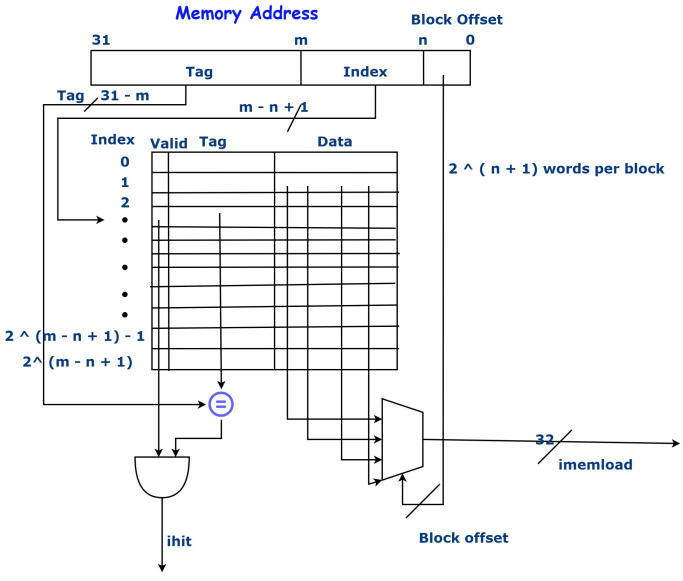

Fig. 8. RISC-V I-cache Microarchitecture

Our instruction cache (I-cache) is set up as a one-way, directly mapped cache because access patterns to instructions are usually linear and easy to predict. This means that the processor will get the next few instructions from memory after this one. This makes it less likely that there will be a cache conflict. A direct-mapped I-cache has simple indexing and replacement logic that makes accessing it the fastest possible. This is important for keeping the pipeline full and maintaining high instruction flow. Unlike data caches, instruction code is generally read-only and doesn't change much while it's being executed. This means that performance loss due to conflict or capacity misses isn't as likely. So, a 1-way plan for I-cache is a good compromise between ease of use, speed, and a high enough hit rate for most workloads.

The flow of how I-cache works is that when the scheduler core wants to read an instruction, it will first check the cache instead of going directly to the memory. When the cache sees imemREN and imemaddr the I-cache knows that the CPU wants to read instruction from it. It will decode the instruction address from the scheduler core into tag bits, index bits, block offset and byte offset. Block offset is used when there are more than 1 words in each index. By knowing the index bits, it can go check the valid bit in the cache table to see if that row is valid or not. If it is valid, and the tag stored in cache matches the decoded tag, then it will be a cache hit. Cache hits are processed immediately and will pass the instruction data by imemload back to the scheduler core and a ihit to tell the scheduler the data is ready. However, if neither of the conditions above are matched (invalid bit in that index and tag doesn't match), then the I-cache needs to send a request to the memory arbiter. It will send the read enable signal passed from the scheduler core to the memory arbiter to tell it that it wants memory from main memory. Then it will give the memory

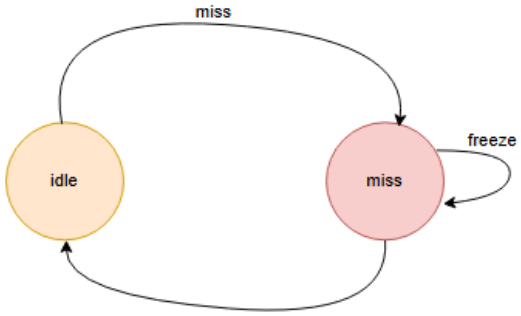

Fig. 9. I-cache FSM

arbiter the instruction address to tell the memory arbiter the address to pull the instruction from. When the data is ready, the memory will set the iwait bit to low so that the I-cache knows the data in iload from the memory arbiter is valid and ready to be stored into the I-cache.

Figure 9 exemplifies the simplicity of the I-cache internal flow. From the idle state when an instruction request arrives, we can output the instruction almost immediately from the same state if it was already in our cache, but if it's a miss, then it will take some extra time to go to the miss state, fetch the instruction from main memory, and come back to idle to output the instruction to the scheduler core.

C. Data Cache

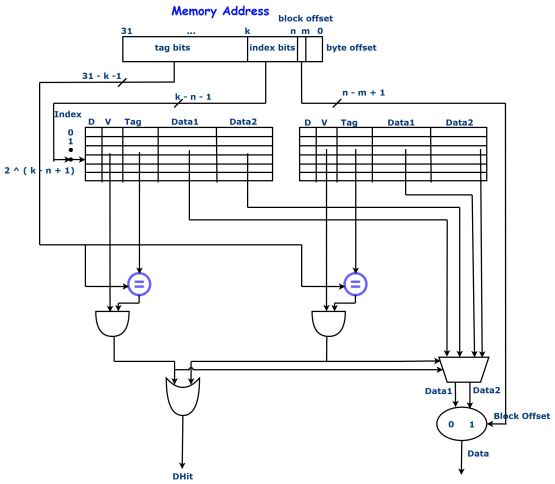

Fig. 10. RISC-V D-cache Microarchitecture

Our D-cache is a two-way set-associative cache, and it works faster than a one-way, directly mapped cache because it has a lot fewer conflict misses. This happens when multiple memory blocks map to the same cache line and keep pushing each other out. The two-way design gives higher hit rates because each block can be kept in one of two lines within a set. This is especially helpful for our workloads that use loops or other repetitive access patterns. Multiple tag comparisons and replacement logic (e.g., LRU) make it a bit more complicated and slow to access, but it hits a good balance between performance and cost.

The flow of how D-cache works is when the scheduler core wants to store or load a scalar value in memory, and likewise, it will first check if the cache holds that value needed before pulling from main memory. The two signals that the scheduler core will send us if they need memory are dmemREN for reading a value in memory and dmemWEN for writing a value in memory. The D-cache internal flow is similar to the I-cache

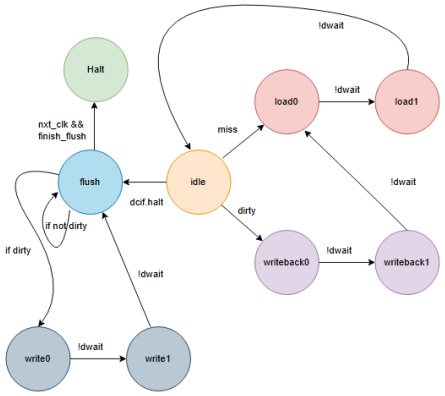

Fig. 11. D-cache FSM

internal flow, but with a few additions. Previously, in Figure 9, when dealing with a miss, we had to pull in values from main memory before returning the instruction. Likewise, in Figure 11, when handling a miss, we would have to pull values from main memory and store them into the cache first before returning. An additional feature the D-cache has is having a dirty bit inside each of its cache frames. The reason for this is to track the writes the D-cache is doing. Remember that I-cache is read-only, but D-cache does both reads and writes, making it destructive. Thus, we would need a state to write back this dirty data. Also in Figure 11, we need a state to flush out all of the dirty data left over when the scheduler core queues a halt because we want our final memory to be up to date. The reason why the states are doubled up: load1, writeback1, and write1, is because the D-cache contains two words of data in a single block as described in our cache organization above. By having two words, the D-cache takes advantage of spatial locality and grabs data next to the one the scheduler core needs.

D. Memory Arbiter

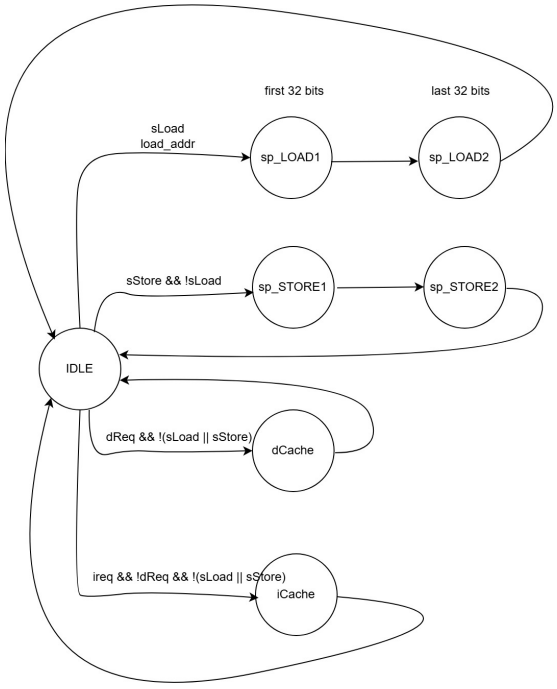

Fig. 12. Old Memory Arbiter FSM

The memory arbiter that we have created is basic in the way that it doesn't implement coherence so accesses are sequential and can't be parallelized. An example of the flow can be found in Figure 12, where the priority can clearly be seen by the control signals leading to spLOAD1 are only dependent on its own related handshake signal, however, to get into the iCache state the scratchpad and dcache handshake signals need to be low. A major problem that arose while implementing the testbench for the full AMP00 was that the scheduler immediately release enable signals for the dcache or icache when a hit signal was released from their respective cache.

This means that there is a chance down the pipe that both states will have the perfect timing to both request access from memory at the same time causing one to not load any values for their cache. A solution to this can be seen in Figure 13,

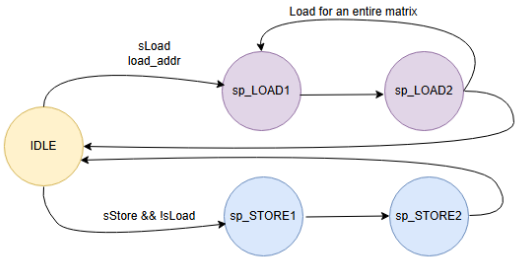

Fig. 13. New Memory Arbiter FSM

where we can transfer the I-cache and D-cache states to the Idle state and arbitrate access there combinationally, so one cache can never access memory at the same time as another.

E. Main Memory

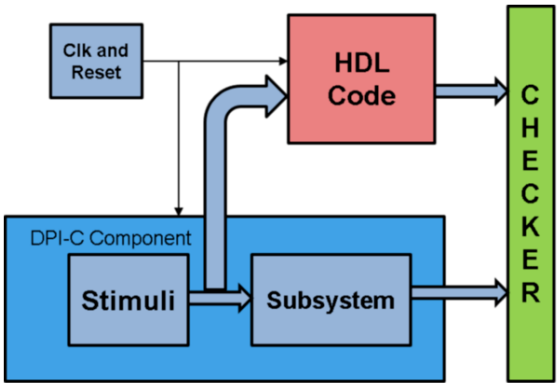

Fig. 14. SystemVerilog DPI test bench for verifying HDL with large datasets, adapted from [1].

The main memory model for this project is written in C++, and the DPI-C (Direct Programming Interface for C) connects it to the SystemVerilog testbench. This setup allows you to use QuestaSim to simulate memory access quickly while the design under test (DUT) is being functionally tested. The memory model reads from a file called `meminit.hex` that has already been set up, acts out read and write actions in memory, and communicates with SystemVerilog through a shared object file called `memory.so`.

The first step in memory flow integration is the hex file generation. The instruction and data memory will be initialized using a hex file called `meminit.hex`. This hex file will be generated from a compiler or assembler targeting our custom RISC-V 32 instruction set. This custom set is based on the current RISC-V instruction set plus some custom instructions used for matrix load and store operations. See details of this instruction set in the ISA team report.

The next step is the memory model implementation. There are four functions in `memory.cpp`, including `mem_init`, `mem_read`, `mem_write`, and `mem_save`. The `mem_init` function will load the `meminit.hex` into a `std::vector` in C++.

The reason for choosing `vector` instead of other container types is because the container choice impacts performance, memory layout, and access time—important considerations in a memory model. Using `vector` implies a sequential memory with indexed access, which closely resembles how real memory works. It simplifies simulation of word-addressed memory with $\mathcal{O}(1)$ read/write operations.

A second reason is scalability and flexibility. The `vector` container can dynamically grow to adapt to any address range defined by the hex file, unlike fixed-size arrays. Compared to the `map` container, which has $\mathcal{O}(\log n)$ overhead, `vector` provides constant-time access, which is critical in simulations involving frequent memory access. The `vector` also consumes less memory in dense address spaces.

The functions `mem_read` and `mem_write` handle memory operations at the specified address. Each function takes an `address` and `data` argument, allowing read or write access to the simulated memory model. The `mem_save` function writes the modified memory contents back to the `meminit.hex` file. This final step enables easier debugging and result comparison by preserving memory state changes across simulation runs.

All of these functions use the `svBitVecVal` type from the `svdpi.h` header to exchange data between C++ and SystemVerilog.

The C++ file is compiled into a shared library object file (`memory.so`) using the following command:

```
/package/eda/mg/questa2021.4/questasim/\
gcc-7.4.0-linux_x86_64/bin/g++ -shared -fPIC \
-I/package/eda/mg/questa2021.4/questasim/include \
-o memory.so memory.cpp
```

In the SystemVerilog testbench, the C++ functions are imported using DPI-C syntax:

```
\begin{verbatim}
import "DPI-C" function void mem_init();
import "DPI-C" function void
mem_read (input bit [31:0] address, output bit [31:0] data);
import "DPI-C" function void
mem_write(input bit [31:0] address, input bit [31:0] data);
import "DPI-C" function void mem_save();
```

The SystemVerilog testbench (`example_mem_tb.sv`) calls these functions directly to simulate memory accesses. Since C++ is not event-driven, all C++ memory functions return instantly with no simulated delay.

F. Scratchpad

As previously discussed, the scratchpad is implemented as a 2 KB software-controlled memory organized into four banks, enabling multiple simultaneous read and write operations. This banking strategy is essential for sustaining high throughput, especially when multiple matrix tiles need to be accessed concurrently by the compute units. However, while banking

improves parallelism, it also introduces significant complexity in arbitrating access to the scratchpad. Managing concurrent access across banks, especially under high instruction throughput, requires careful coordination to avoid conflicts and stalls. To address this, a set of dedicated FIFOs and FSMs were implemented in parallel. The FIFOs temporarily queue incoming read and write requests, while the FSMs orchestrate the timing and order of these operations based on bank availability, instruction type, and priority.

This hardware arbitration logic ensures that instructions arriving from the scheduler core are handled efficiently, without contention or delays that could compromise system performance. Most importantly, it guarantees that the systolic array remains fully utilized, with a continuous stream of data, thus preventing pipeline stalls due to memory access latency. By tightly coupling the scratchpad's access control with the instruction flow, the architecture achieves both high data bandwidth and predictable execution timing, which are critical for performance in matrix-heavy workloads. All of the following descriptions will reference blocks shown in the scratchpad architecture diagram in Figure 5.

Before the scratchpad can begin executing operations, it must first receive instructions from the scheduler core. These instructions are enqueued into a dedicated instruction FIFO. This buffering mechanism allows the scratchpad to manage instruction throughput. The scratchpad supports three primary instruction types: a matrix load, a matrix store, and a matrix multiply. A matrix load transfers a matrix tile from main memory into the scratchpad. A matrix store writes a matrix tile from the scratchpad back to main memory. A matrix multiply sends matrix tiles from the scratchpad to the systolic array for computation.

A key feature of the architecture is that the scratchpad can handle one instruction of each type in parallel. This means that a load, store, and matrix multiply can all be in progress simultaneously, utilizing separate logic paths and independent resources. This concurrency maximizes throughput and ensures that no single operation blocks others, thereby improving overall system performance. Once an instruction completes, the scratchpad sends a handshake signal back to the scheduler core. This signal serves as an acknowledgment that the corresponding instruction type has been processed and that the FIFO now has space to accept another instruction of the same type. This handshake mechanism ensures synchronization and flow control and prevents the scheduler from sending dependent instructions at the same time.

There are four identical instances of the Bank Access FSM, each responsible for managing one of the four scratchpad banks. These FSMs continuously monitor the instruction FIFO to detect when a new instruction has been issued. Upon the arrival of a new instruction, each FSM checks whether the matrix operand involved is stored in its corresponding bank. If so, the FSM engages its arbitration logic to schedule access. The arbitration process ensures that multiple instructions targeting the same bank are serviced in an orderly and conflict-free manner. Each FSM operates independently but follows

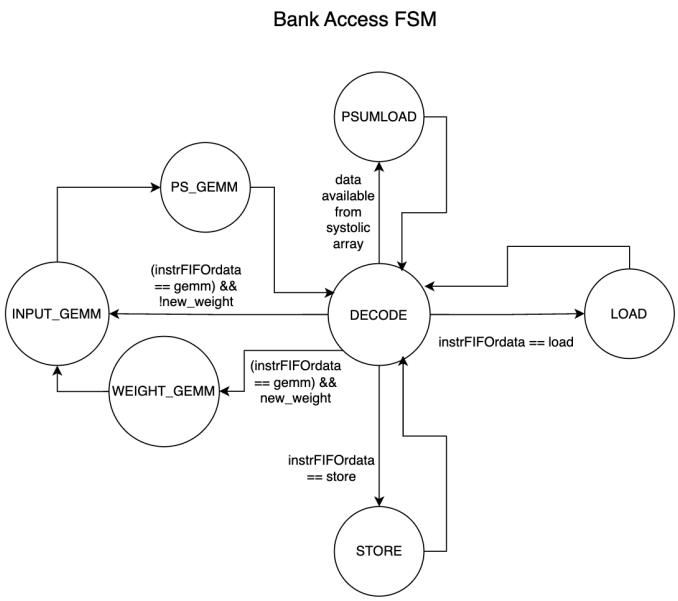

Fig. 15. Bank Access FSM

a uniform arbitration policy, as illustrated in Figure 15. This figure outlines the decision-making logic used to prioritize and sequence operations, allowing the scratchpad to maintain high throughput while ensuring correct and synchronized memory access across all banks. In the DECODE state, priority is given to the PSUMLOAD state, which is responsible for loading the output of the systolic array, stored in the Partial Sum FIFO, back into the scratchpad. This prioritization ensures that matrix multiplication operations complete as quickly as possible, enabling the scratchpad to promptly send a handshake signal back to the scheduler core. By accelerating this, the design maximizes the utilization of the systolic array and minimizes stalling. Additionally, once all four Bank Access FSMs have scanned the current instruction and incremented the instruction FIFO, the FSMs can proceed to the next instruction independently, even if one or more FSMs are still completing operations from the previous instruction. This decoupled execution model allows multiple instructions to be in-flight within the scratchpad simultaneously, increasing instruction-level parallelism and overall throughput.

Matrix register writes, such as storing partial sum outputs from the systolic array or loading data from DRAM, are handled directly by the Bank Access FSMs. These write operations are straightforward because they involve placing data into the scratchpad's matrix register banks, with minimal control overhead. Read operations, on the other hand, require more complex handling. These include matrix store instructions, which read data to be sent back to DRAM, and matrix multiplication instructions, which read data to be forwarded to the systolic array. When a read is initiated, the Bank Access FSM issues the request to the appropriate bank and appends a set of encoding bits along with it. These bits carry essential metadata that informs the bank logic about how

to handle the outgoing data. Specifically, the encoding bits indicate whether the matrix being read is destined for DRAM or for the systolic array. In addition, if the data is intended for the systolic array, the bits also identify the matrix type which distinguishes between weight matrices, input matrices, and partial sums. This classification ensures that the downstream logic can correctly interpret the role of the matrix in the GEMM pipeline.

As a result of this tagging mechanism, the scratchpad bank logic knows exactly where to route the output row. Rows meant for DRAM are directed to the DRAM Store FIFO, while those intended for computation are sent to the GEMM FIFO. This approach allows for flexible, efficient data movement and decouples the read process from the instruction source, enabling high-throughput matrix operations without unnecessary stalls or ambiguity.

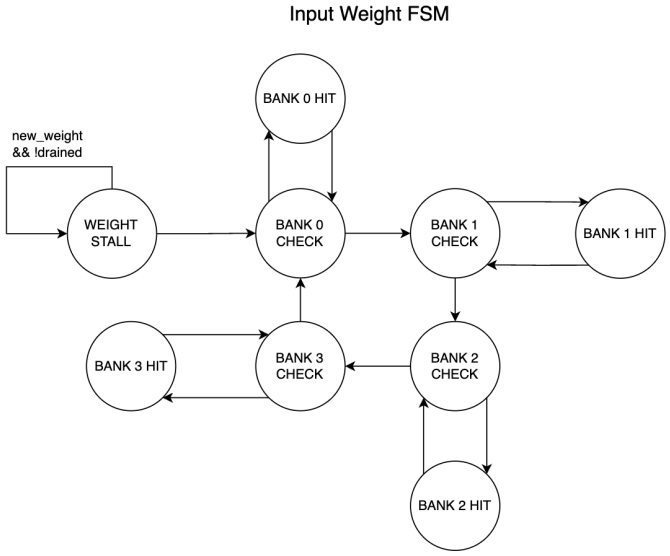

Fig. 16. Input/Weight Bus FSM

For any DRAM store operations, data movement from the scratchpad to main memory is managed by the DRAM Store FSM, which functions as a static priority arbiter. This design is appropriate for the current iteration of the scratchpad architecture because only one store instruction is allowed to be in flight at any given time. As a result, there is no need for dynamic arbitration or complex scheduling mechanisms. The DRAM Store FSM continuously monitors the DRAM Store FIFOs, each of which buffers row data originating from the scratchpad banks. When the FSM detects that data is being pushed into any of the FIFOs, it begins pulling rows from the active FIFO. These rows are then sent to the DRAM interface for writing to main memory.

Finally, for GEMM operations, the GEMM FSM is responsible for orchestrating the most critical data transfers in the scratchpad system. This FSM handles the coordination of matrix data sent to the systolic array, leveraging two dedicated 64-bit buses: one for input and weight matrices, and another

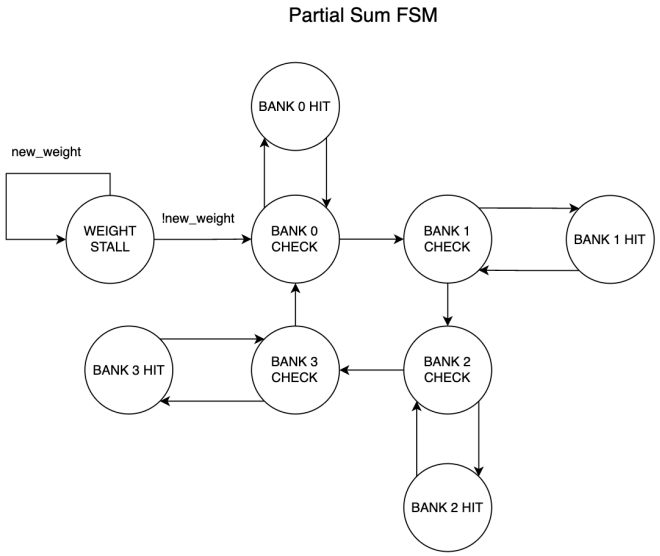

Fig. 17. Partial Sum Bus FSM

for partial sum matrices. Each of these buses is paired with its own dedicated FSM to arbitrate which matrix rows are transmitted and when.

Due to the weight-stationary design of the systolic array, it is essential that weight matrices are preloaded before any input or partial sum data is sent. To enforce this constraint, the scheduler core sends a special ”new weight” flag along with any GEMM instruction that requires a new weight matrix to be loaded into the array. Upon receiving this flag, the scratchpad registers a signal indicating that a new weight is being loaded.

This registered signal forces the Partial Sum Bus FSM to stall, preventing it from sending any partial sum rows to the systolic array until the weight matrix has been completely transmitted. Meanwhile, the Input/Weight Bus FSM actively scans the GEMM FIFOs to determine whether the required weight matrix is available. If found, it immediately begins transmitting the weight matrix rows to the systolic array at maximum throughput. Once all rows of the weight matrix have been delivered, the “new weight” signal is unregistered, allowing the Partial Sum Bus FSM to resume operation. This careful coordination ensures that the systolic array receives its inputs in the correct order and can begin computation without error.

It is important to note that this weight-loading protocol is a relatively rare occurrence, owing to the high reuse factor of weight matrices in most workloads. In the more common case, only input and partial sum matrices need to be streamed into the systolic array. When no new weight is required, both the Input/Weight and Partial Sum FSMs operate in a round-robin arbitration mode, checking the FIFOs and dispatching matrix rows as soon as the bus becomes available. This arbitration strategy ensures fairness and keeps both buses fully utilized under typical operating conditions, maintaining high throughput for the GEMM pipeline.

VII. Evaluation and analysis

A. Caches and Memory Arbiter

We couldn't get synthesis results for the instruction and data caches because of time or toolchain issues, but a qualitative analysis of the design choices gives us an idea of how well they should work and what the area will be like. The instruction cache was made to be a direct-mapped cache with little associativity so that it would be fast and easy to use. This is because instruction access patterns are usually reliable and low-latency fetches work best. The data cache, on the other hand, was made as a 2-way set-associative cache to find a good mix between hit rate and hardware complexity. This is because data access patterns aren't always regular in a direct-mapped design, which can lead to more conflict misses. Theoretically, the instruction cache should take up less space and have less latency, while the data cache should cost a little more but work better under normal tasks. To be sure of these predictions, more research with combined data would have to be done. Overall, both caches are made to be parameterizable so that future iterations of the AMP can utilize memory load/stores for instructions and scalar values of any size optimally.

The memory arbiter was heavily focused to load/store memory from a DPI-C C++ main memory module. In future AMP iterations with DDR4, DDR5, ETC, simulations the memory arbiter would need to be replaced with a memory controller with their respective dram. However, this module is able to simulate any DRAM, provided timings for loads/stores are given by a third-party module like Ramulator.

B. Main Memory

After simulate the main memory in QuestaSim, we can see that the data will come out instantly from the main memory with no delay. There are three reasons behind that. The first reason is that C++ is not event-driven like Verilog. Because in Verilog we can do #delays or `@posedge clk` to create some delays. While in C++, unless we want to add a delay mechanism, all the computation will happen immediately in zero simulation time. And our goal in this phase is just to use the main memory to make sure all the parts that will interact with it (I-cache, D-cache, memory arbiter, and scratchpad) have the correct functionality, so there is no point in adding extra delays.

The second reason is that the DPI-C interface does not advance simulation time. When SystemVerilog calls a C++ function by DPI-C, it will stop the Verilog simulation and executes the C++ code immediately and resume the simulation at the same timestamp.

The third reason is that the C++ memory right now are just returning data from a vector with no modeled wait or access time.

C. Scratchpad

The scratchpad architecture was designed to ensure that the systolic array remains fully utilized with minimal stalls due to memory latency or bandwidth constraints. The final implementation successfully supports high-throughput matrix operations by leveraging a software-controlled, untagged, 4-bank scratchpad architecture with support for parallel reads and writes. With each bank capable of servicing a matrix read or write independently, up to four simultaneous accesses can occur, reducing contention and improving overall bandwidth. During typical GEMM operations, input and partial sum matrices are streamed concurrently through two dedicated 64-bit buses. This dual-bus design ensures that the systolic array receives a new row every cycle after initialization, keeping compute resources fully active. During testing, the banked scratchpad architecture demonstrated significant advantages in throughput and responsiveness. Additionally, the dual-bus design performed as expected since the FSMs were able to juggle the weight pre-loading effectively.

Under standard matrix multiplication workloads the scratchpad architecture performed exceptionally well. The ability to store and reuse weight matrices locally reduced DRAM bandwidth pressure and allowed the systolic array to remain fully utilized. However, one notable caveat is that the effectiveness of the scratchpad depends on intelligent bank access scheduling. Since each bank operates independently, it is up to the programmer or compiler to ensure that concurrent instructions are distributed across different banks to avoid conflicts. When all four banks were utilized effectively, the scratchpad exhibited strong multitasking capabilities, handling simultaneous matrix loads, stores, and GEMM operations with minimal contention. This highlights the importance of software-awareness in managing data locality and instruction scheduling, which, when done properly, can unlock the full performance potential of the banked design.

Despite its strengths, the current design has a few notable limitations. First, the static-priority DRAM Store FSM supports only one in-flight store at a time. While sufficient for this iteration, this becomes a performance bottleneck under workloads with frequent output writes, especially if partial sum reuse is low. Future versions could implement a multi-instruction store queue and a round-robin or dynamic arbitration mechanism to address this. In addition to this, the scratchpad only supports one in-flight load at a time. This is also bottlenecking the design since the scheduler needs to schedule up to 3 loads before a GEMM instruction. By implementing a system to track different loads, the scratchpad could perform much better.

Additionally, although the software-controlled, untagged cache structure maximizes SRAM usage, it places a significant burden on the compiler or runtime system to manage data movement explicitly. If the compiler decides not to utilize the banked nature of scratchpad, a large chunk of the scratchpad remains inactive. The architecture also assumes a weight-stationary compute model, which simplifies certain optimizations but also reduces flexibility. For layers or workloads with rapidly changing weights, the cost of loading new weights could become more pronounced.

The scratchpad architecture was synthesized using MITLL's 90nm Silicon-on-Insulator process, achieving a clock frequency of 696 MHz. The final synthesized design occupied

an area of 941,580 μm², making it a compact yet capable component relative to the scale of the overall system. Power analysis revealed a total consumption of approximately 386.27 mW, which is reasonable given the high-speed operation and the support for multiple simultaneous accesses across four banks. These results validate the scratchpad as a high-performance and area-efficient memory subsystem, well-suited for tightly coupled accelerators like systolic arrays, particularly in compute-intensive matrix multiplication workloads.

VIII. Lessons learned and future work

A. Caches and Memory Arbiter

Designing the caches and memory arbiter relied heavily on the structure of our main memory. Main memory was an important foundation for this project and more resources should've been allocated to getting main memory working. Memory arbitration is an interesting concept during this project because the arbiter relied on the processes and flow of all the modules. For example, an instruction would pass through the scheduler core, into the I-cache, and then down to the arbiter. A matrix would pass through the systolic array, into scratchpad, and then down to the arbiter. When one a transaction didn't output the correct values it was easy to pinpoint where the flow might've failed because the handshake signals between every module in the AMP00 were so concrete. We would know if the flow stopped before or after the memory arbiter.

Some future work are needed for caches. First of all, all the dpi-c-related code should be removed from the source module code because they are not synthesizable. So the memory arbiter shouldn't directly DPI-C functions like mem-read and mem-write directly. Instead, these function should be used in the testbench because we don't need to synthesize the testbench anyway. The easy fix to that is replace all the dpi-c functions with some signal that can be connected to the DUT. The DUT will input the data read from hex file and output the data to the hex file. Secondly, even if the caches doesn't use the DPI-C functions, they are still not synthesizable for some reason. Due to the time constraint, they can't be done when this report is due.

B. Main Memory

As we mentioned above, right now our main memory written in C++ has no delay mechanism and is purely a tool for simulation. There are some more things we can do about it in the future. The first step is maybe we can add some delays say 300 ms in C++, to make it look like a real main memory. The second step is to use Ramulator as a tool to create the behavior as a real DDR main memory. However, the ramulator itself doesn't contain the functionality of storing bits and is only used for tell the main memory when the memory is ready (which simulates how DDR main memory works in the real world). So we need a wrapper and unique config file to use it, which the Ramulator team is working on it right now and hopefully will be implemented in AMP01. Also, because of the area constraints in the FPGA, we need to buy some real main memory to hold the data if we want to make it taped out.

C. Scratchpad

Designing the scratchpad memory system brought several challenges and important insights. One of the main difficulties encountered was managing the complexity introduced by the multi-banked architecture. While having four independent banks significantly boosted performance by enabling simultaneous access, it also created arbitration and coordination challenges. Early versions of the system suffered from over-serialization of independent instructions. These problems were ultimately resolved by implementing robust, independent FSMs and using FIFO-based buffering to decouple instruction flow from memory access. This design allowed operations to proceed in parallel without bottlenecking the systolic array.

Another key takeaway was the critical role of software-hardware co-design. Because the scratchpad operates as a software-controlled, untagged memory, it relies heavily on the programmer or compiler to manage data placement and avoid conflicts. This approach reduced hardware complexity and saved area by eliminating the need for tag arrays, but it increased the burden on software to ensure correctness and efficiency. As a result, any future improvements should include better tooling or compiler support to automate or guide memory management decisions.

Additionally, we learned that while the DRAM store path seemed non-critical at first glance, its single-instruction throughput could become a limiting factor in some workloads. Similarly, the current design allows only one load instruction from DRAM to be in flight at a time, which is a performance bottleneck since multiple loads need to be performed for each GEMM instruction. Enabling multiple concurrent load instructions would improve data movement efficiency and better match the parallel nature of the compute pipeline. In the same vein, supporting multiple GEMM instructions in flight could significantly enhance system throughput by ensuring the systolic array remains continuously fed with data. Currently, if one GEMM instruction is stalled waiting for a matrix load or bus arbitration, the entire compute pipeline may idle. Allowing multiple GEMM instructions to proceed simultaneously would help hide these latencies and maintain a steady flow of data into the systolic array.

Looking ahead, future versions of the scratchpad could benefit from supporting both multiple in-flight load and store instructions, as well as concurrent GEMM instruction handling. More flexible arbitration schemes and the addition of lightweight metadata or tags could reduce programming complexity without incurring the full overhead of a traditional cache. Furthermore, evaluating how this architecture performs under more complex workloads is a key direction.

References

[1] MathWorks, "Verify HDL Design with Large Dataset Using SystemVerilog DPI Test Bench," *MathWorks Documentation*, https://www.mathworks.com/help/hdlcoder/ug/verify-hdl-design-with-large-dataset-using-systemverilog-dpi-test-bench.html. [Accessed: May 3, 2025].