oPOSsum: Network-on-Chip based SAT Solver Accelerator

ECE 427 Final Report

Kai Karadi
Electrical and Computer Engineering
University of Illinois
Champaign, United States
kaick2@illinois.edu

Hrishi Shah
Electrical and Computer Engineering
University of Illinois
Champaign, United States
hrishis2@illinois.edu

Wesley Wu
Electrical and Computer Engineering
University of Illinois
Champaign, United States
wwu70@illinois.edu

Parithimaal Karmehan
Electrical and Computer Engineering
University of Illinois
Champaign, United States
pk38@illinois.edu

Aaditya Kothary
Electrical and Computer Engineering
University of Illinois
Champaign, United States
kothary3@illinois.edu

Cher Rui Tan
Electrical and Computer Engineering
University of Illinois
Champaign, United States
cherrui2@iliinois.edu

I. Abstract

***Abstract*—This proposal presents the design and implementation of an SAT solver accelerator, which mainly focuses on Boolean constraint propagation (BCP), as it is the bottleneck in traditional SAT solving algorithms. Many complex problems can be reduced to Boolean satisfiability problems, including formal (hardware) verification, software code coverage, and even cryptography. We would also like to point out that it is sometimes also used in specialized compilers, e.g., that of the GRAPE chip from the Fall 2024 semester of ECE 498HK, which uses satisfiability constraints to map kernels onto a CGRA unit. This report includes a project summary, a post-silicon validation plan, individual contributions, challenges faced along the way, and finally design documentation and testing instructions.**

II. Project Summary

A. Main Features and Functionality

The chip is a Boolean Satisfiability (SAT) Solver that supports problem sizes of 128 variable indices and 224 three-variable clauses.

Boolean Satisfiability is the process of taking a Boolean expression in conjunctive normal form, or product of sums, and finding a set of Boolean assignments that make the expression true. It is an NP-complete problem, with the current best algorithms running in exponential time in the worst case.

Problems can be loaded into via SerDes, and the satisfiability evaluation, as well as the corresponding variable assignment, will also be output via SerDes. The chip also supports external observability of internal structures via SerDes, as well as three specialized pins for displaying the controller state, both of which can be enabled via our FFC (Filter-For-Complete) pin.

B. Quantitative Results

Using post-place-and-route (post-PNR) simulation, we evaluated the chip operating at 200 MHz on a subset of benchmarks drawn from the 2025 SAT Solving Competition with the filter for complete on (FFC 1) and off (FFC 0). In addition, we provide a comparison against Glucose [1], a widely adopted state-of-the-art software SAT solver, serving as a representative industry baseline. Glucose was run using the single thread configuration with all modern heuristics, including clause learning, on an Apple M2 Pro chip, plugging in, with low CPU utilization and low memory utilization for other applications, with multiple previous runs for cache efficiency. This comparison enables a quantitative assessment of the proposed accelerator's performance characteristics relative to a mature software implementation. Moreover, the results, seen in 1, include relative speed-ups between the three configurations. On average, we outperform glucose on our test set with a geometric mean of a 2.59x (3.66 arithmetic mean) Speed Up for FFC 1, and a 2.41x (4.43 arithmetic mean) Speed Up for FFC 0. That said, the deviation within speed up is quite high, ranging from 11x to 0.07x. We comment on this within the Challenges Speed Up section.

C. High-Level Architecture

Our chip is comprised of three major architectural elements: the controller, the Network-on-Chip (NoC), and the clause banks, which contain banks of functional units (FUs). See 2

1) Controller: The controller not only orchestrates program loading into the clause banks, but is also responsible for initial variable assignments, implication generation, and backtracking in the case of conflicting assignments/implications. Its backbone is an FSM that has states corresponding to each of the aforementioned responsibilities, and uses an SRAM to store implications. In the late stages of our RTL design cycle, we also stored variable counters, using the frequency counts of each variable to optimize initial variable assignment, greatly accelerating the process of determining satisfiability.

AIM-50 Test Case	FFC 1 (ms)	FFC 0 (ms)	Glucose (ms)	SUp F1/F0	SUp F1/G	SUp F0/G
aim-50-16-yes1-1	0.55	0.87	1.44	0.63	2.62	1.66
aim-50-16-yes1-2	0.13	0.13	1.34	0.98	10.45	10.29
aim-50-16-yes1-3	1.23	0.47	1.50	2.62	1.22	3.19
aim-50-16-yes1-4	0.14	0.18	1.26	0.76	9.07	6.88
aim-50-16-no-1	2.46	2.29	1.80	1.07	0.73	0.79
aim-50-16-no-2	1.67	3.41	1.53	0.49	0.91	0.45
aim-50-16-no-3	18.57	3.41	1.32	5.44	0.07	0.39
aim-50-16-no-4	0.20	0.30	1.41	0.67	7.08	4.75
aim-50-20-yes1-1	0.39	0.55	1.33	0.71	3.38	2.42
aim-50-20-yes1-2	0.16	0.21	1.83	0.79	11.10	8.74
aim-50-20-yes1-3	0.36	0.42	1.94	0.87	5.37	4.68
aim-50-20-yes1-4	0.38	0.38	1.76	1.02	4.58	4.69
aim-50-20-no-1	0.31	0.47	1.28	0.66	4.06	2.69
aim-50-20-no-2	2.68	2.89	1.30	0.93	0.48	0.45
aim-50-20-no-3	0.39	0.62	1.52	0.63	3.87	2.44
aim-50-20-no-4	0.23	0.34	1.38	0.69	5.88	4.03

Fig. 1. Performance results for AIM-50 test cases, showing speed-up factors across different filter configurations. FFC 1 and 0 are the oPOSsum simulation times on the filter for complete on and off settings. SUp is speed up between different configurations F1(FFC 1), F0(FFC 0), G(Glucose)

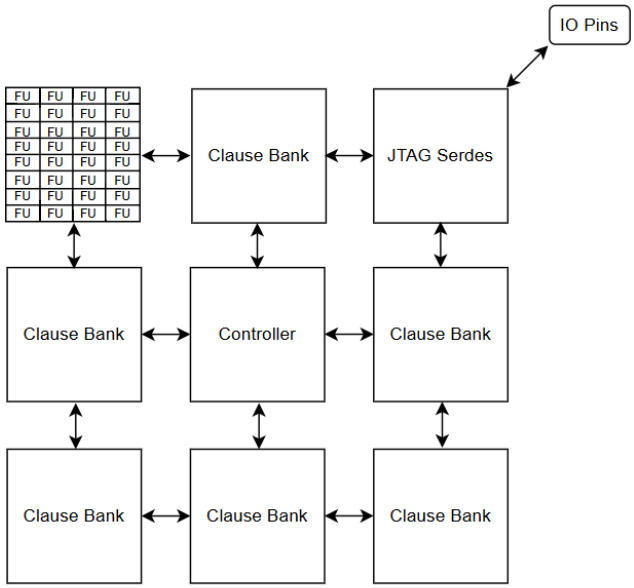

Fig. 2. High-level architecture

2) Network-on-Chip: The NoC is the chosen medium of communication across the chip, as it provides better scalability than a common bus-based architecture. This chip uses a 2-dimensional 3x3 mesh routing structure, where each router is connected to either a clause bank, the controller, or the SerDes interface. Our design only works with single-flit packets and uses dimension-order routing, thus reducing the logic for (re)routing and the possibility of deadlocks. Whilst this approach creates a lot of congestion on the east-west links, we made this design tradeoff as the total number of packets sent out per cycle is low in our design. A router contains 5 ports, 4 of which are used to communicate with other routers, as well as a local port to interface with a clause bank, controller, or SerDes.

3) Clause Banks: The clause banks are composed of 32 functional units, with each functional unit storing a three-variable clause. We chose to implement clause banks to reduce stress on the network by reducing network traffic, while at the same time allowing us to increase the density of the functional units without increasing the size of the mesh. The clause banks perform ordered program loading, with each functional unit filling up first before loading in the next, etc. When a functional unit generates an implication or conflict flit, it raises a request to the bank arbiter, which performs round-robin arbitration to send out flits one by one.

a) Functional Units: Each of the functional units stores one three-variable clause, including the variables' polarities (negated or non-negated) as well as their assigned state (true or false), and computes in two clock cycles any implications (i.e., all other variables assigned values opposite of their polarities with one remaining unassigned variable) or conflicts (implications or assignments which differ from previous implications or assignments to the same variable). It raises a request whenever either an implication or a conflict is generated, and only de-asserts it when the request is serviced by the network. To take flits in from the network, it scans for a valid signal asserted by the network, and is configured to always assert its ready line back to the network to reduce congestion. These valid-ready and req-grant interactions are explained in further detail in section VI-C.

D. Physical Design

Our chip layout is shown in Figure 3. We have included an SRAM near the top-left corner for controller functionality. For clearer viewing, we have also included a screenshot of

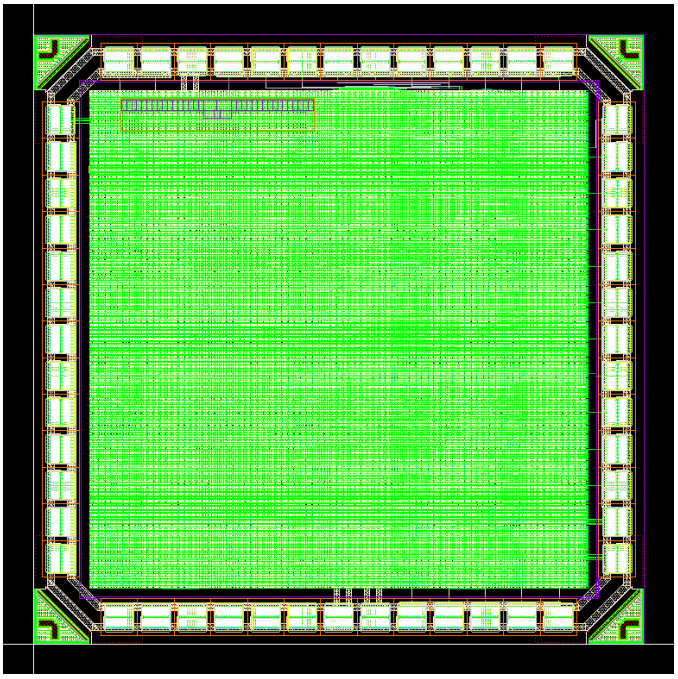

Fig. 3. Chip layout

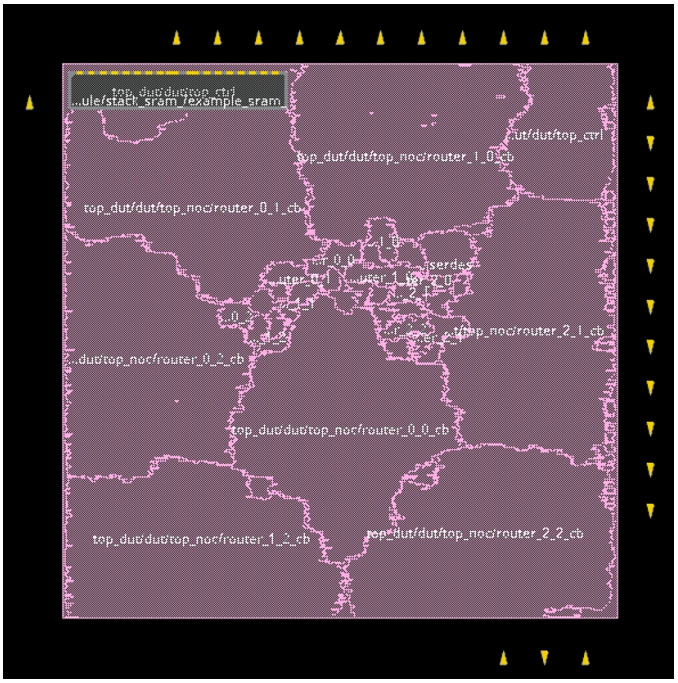

Fig. 4. Amoeba view of layout

our chip's amoeba view in Innovus (see figure 4) showing our area breakdown by module. As you can see, the majority of our area is taken up by the clause banks (the modules suffixed with `_cb`), then followed by the controller (`top_ctrl` at the top right), then the SRAM at the top left. In the center is the JTAG serdes RTL module, and then the rest of the area is consumed by the NoC routers, which are much smaller in comparison to the clause banks. We attempted to do guided PnR, but found that with our area constraints, the automatic PnR worked much better, hence the random structure of the clause banks and NoC.

E. Statistics:

Ultimately, the core area size we used was 815μm x 815μm, and the pre-route, pre-density filler area density was 73.4%. Our chip has 3 VDD/VSS connection pairs to help maintain stable power delivery. The power draw for a single SAT run was 254.7937 mW. We note that this power is somewhat extreme and include a discussion on causes and remedies in the challenges section.

1) Pins:

- 3 VDD
- 3 VSS
- 1 CLK
- 1 MOSI_CONTROL
- 8 MOSI
- FPGA_READY
- ASIC_READY
- 1 MISO_CONTROL
- 8 MISO
- 1 FILTER_FOR_COMPLETE
- 1 SYNC
- 3 CONTROLLER_STATE

The operation of these pins on the chip when interfaced with an FPGA is described in further detail in section VI-F, along with a diagram of the pinout, which is shown in figure 9.

F. Design Choices and Justification

As mentioned previously, we chose a NoC-based architecture as the timing constraints and area scale better compared to a common bus-based architecture. However, the time taken to solve a program increases exponentially with the dimensions of the mesh. To mitigate this, we use clause banks, which would provide some of the benefits of having a bus, while still being a scalable design. We explored square and rectangular meshes, from 3x3 to 5x5 network size, but empirically found that 3x3 was best for the problem sizes that we were targeting.

Since one of our main goals with the NoC was to increase our frequency, we had to decide how many buffers we needed for receiving and transmitting packets. Initially, we decided on having 4 rx buffers and 2 tx buffers as we believed there would be more implications being generated per unit time than implications being sent out. This would enable the network to handle more incoming packets while the switch works on switching the packets. However, this approach added a lot

of flip-flops, which created zones that were hard to route through. Furthermore, functional units were always able to receive incoming flits without having to send out flits first; this allowed for a gather and scatter architecture, which benefited from high network congestion. As such, we decided to reduce the amount of buffering to 2 registers for any pipeline. When testing different configurations, we found that having 2 tx buffers and 0 rx buffers allowed for the best completion time, as it allows packets to be routed first before waiting, creating more packets that are ready to be sent out per cycle.

We added a SerDes module to minimize the number of pins needed to send and receive flits into our design, be it for program loading, displaying a solution, or monitoring network traffic. This allowed us to vary design parameters, which would change the bit-width of flits on the network, to identify an optimal configuration without having to worry about pin constraints.

As previously mentioned, we also decided to implement clause banks with their own internal arbitration in order to reduce network traffic and, at the same time, allow us to incorporate more clauses into the problem without having to scale the mesh size as much. The reason we chose 32 clauses per bank is to best fit the problem sizes we were with the area constraint — we found via a parameter sweep that a combination of larger mesh sizes and smaller bank sizes would not fit within our die area, and we eventually found a 3x3 network with 32 clauses per bank to contain the highest number of clauses within the $1mm^2$ die area.

To accelerate the solving process, we introduced a decision heap to guide variable selection within the decision tree. The initial decision heuristic prioritized variables based on their frequency of occurrence in the SAT instance. Variables with higher occurrence counts are more likely to participate in constraints and, consequently, to induce conflicts. Prioritizing these variables increases the likelihood that conflicts are exposed earlier in the search, reducing unnecessary exploration of less constrained regions of the decision space.

Building upon this static heuristic, we further incorporated a conflict-driven heuristic that dynamically prioritizes variables frequently involved in conflicts. While the original occurrence-based heuristic aimed to surface conflicts early, incorporating conflict frequency enables the solver to adapt its decision strategy during runtime. By explicitly favoring variables that have historically contributed to conflicts, the solver more rapidly converges toward critical decision points, leading to earlier backtracking and improved pruning of the search space.

With these optimizations applied, we observe a performance improvement of approximately 2× when using the occurrence-based heuristic alone, and up to 20× when augmenting it with the conflict-driven heuristic.

Furthermore, after refactoring our design, we managed to integrate the Decision stack and Implication stack into one data structure, allowing us to save half the area of SRAMs. We sized the new Decision_Implication stack considering the number of variables present in the maximum problem size supported by the accelerator. Since every variable could only exist on the stack, as a decision or implication, we noted that the maximum size of the Decision_Implication stack should be equal to the maximum number of variables in the Maximum problem size supported. Since the number of variables in a SAT Problem is bounded by the number of clauses multiplied by the number of variables in a clause, we noted a maximum of 672 variables. However, in such a case, there is a trivial solution to such a problem. We instead considered the more realistic ratio of common 3-SAT Problems and decided on a 1:2 ratio of the number of variables to the number of clauses.

G. Verification and Test Strategy

1) Verification: Our verification approach was ground-up, meaning we focused on unit-level verification first, then worked up to integration testing.

Unit-level verification was completed for our Functional Units, Clause Banks, and Network on Chips using a combination of random-constrained System Verilog test benches, golden models verifying the correctness of behavior, and directed tests.

This was followed by integration with the controller and the function units using a bus-based architecture. This served as our testing for the controller, which was more difficult to write unit tests for, given the complexities of generating network traffic that was consistent with real-world workloads. Hence, we opted to use a simple integration test to generate network traffic to test both the controller and the interfaces between the two systems. Integration tests for the bus were based on comparing to a Python golden model, which simulated a controller and bus interactions. This was then followed by a regression run on a subset of the AIM testsuite.

This was followed by integration and testing with the Network on Chip. This posed another challenge as writing a golden model for a Network on Chip is incredibly challenging due to the chaotic nature of SAT solving (for more, see the challenges section). Hence, we verified using the regression suite, verifying SAT outputs for true solutions.

Subsequently, IO and DFT were integrated, which was tested in a similar way to the regressions. For DFT, both filters for complete 1 and 0 were tested on full regressions.

2) DFT: We have 3 main elements of our design, which were influenced by the DFT doctrine.

The first is programmability. In addition to the minimum required commands to run a SAT problem, we included network commands that allow the user to program the sync time, the time required for no packets to be on the network before a propagation sequence is deemed complete. We also included clear and updated commands in the clause banks, allowing for more manual control over the status of the clauses.

The second is observability. We mapped some of our extra pins to important internal signals, which can aid in debugging and testing during bring-up. These include a controller state signal reflecting the FSM state of the controller, a sync pulse signal tracking when the aforementioned propagation sequence is deemed complete. The final observability we included was the Filter-For-Complete pin. When this pin is asserted only

program complete packets (packets returning the results of the processor) are outputted. While it is de-asserted, all network packets are outputted, allowing for full network visibility, which will aid heavily in debugging.

The last is a debug mode. By allowing the controller to be turned off, and using the programmability and observability to send and receive network commands mimicking the controller's behavior from an external device, allows our team to fully replace the controller if we find a bug within it. This also allows for some interesting possibilities, such as clause learning, which will be explored in the challenges section.

H. Comparison with Proposal

Compared with our initial proposal, we achieved our goals of implementing a NoC-based architecture for accelerating SAT problems. Our design is faster than the software Glucose SAT Solver for satisfiable problems that have a problem size of around 50 variables and 80 clauses.

However, we were unable to implement clause learning within the hardware architecture - a crucial optimization that most software solvers have, and a feature we need in order to accelerate our chip for unsatisfiable problems and problems with a larger number of variables, i.e., a larger search tree. Nevertheless, our design still supports an "extension" where clause learning can be done off-chip, and implications or conflicts can be sent via SerDes into the chip mid-problem. This will be an area of exploration during the bring-up phase and will be further explored in the challenges section.

In addition to clause learning there was a set of smaller DFT features we were unable to implement. The first was a scan chain, which would have been the ultimate DFT tool. We were able to use the Synopsys tools to generate a scan chain, but were unable to work through certain interactions it had with the post-place and route simulation. In addition, we did not have a NoC built in self self-test, due to time constraints and it being a lower priority. That said, we believe the DFT we did manage to include will provide us with some buffer if bugs are discovered.

III. Post-Silicon Validation Plan

For bring-up, we will first design a PCB housing that allows us to connect the chip to an FPGA, which will have a port for testing power delivery before implementing the FPGA driver design. The FPGA will act as the driver for program loading and result collection, and the FPGA design will be based upon our top-level post-PnR testbench, which we will modify to contain purely synthesizable SystemVerilog and use a 200 MHz clock on the FPGA.

In addition, we would like to demonstrate a small but working clause learning example. More information in section V-D.

Our expected timeline for post-silicon validation/bring-up is as follows:

- January: complete FPGA design and PCB design in parallel.
- February: This is approximately when we should receive the chips, and by the end of the month, we aim to have the chip soldered to the PCB and basic power tests completed.
- March: after verifying power is correctly being delivered to the chip, we will run basic FPGA tests using some of the AIM suite, which we've been using for pre-silicon functional verification. In parallel with the other operations during the first 3 months, we would delegate one team member to work on a clause learning test bench on the FPGA, which should be completed by the end of March.
- April: if we find our taped-out chip to be functionally correct, we will do our best to implement clause learning via hardware-software co-design, but more importantly, we will close out the basic functionality validation and potentially record a demo.
- May: if there are any remaining tasks (e.g., clause learning), we will aim to wrap those up, or accomplish what we can before finals.

The team members we expect to support bring up are Cher Rui (PCB design and Implementation), Parithimaal (FPGA RTL design), Kai (FPGA RTL design - Clause Learning), Hrishi (PCB design and Implementation), and Aaditya (PCB design and Implementation).

IV. Individual Contributions

A. Cher Rui

I designed and verified the baseline controller, generated SRAMs for the implication stack, and implemented a Decision Heap with dual heuristics to improve decision efficiency and overall system performance. Additionally, I contributed to the integration level verification of the BUS architecture to ensure proper communication and synchronization between the controller and functional units. I also designed the SerDes module with Parithimaal. After doing initial place-and-route (PnR) for the controller and figuring out the ideal SRAMs placement, I developed design-for-test (DFT) infrastructure to support a reprogrammable controller architecture with visibility into program execution.

B. Parithimaal

I did integration testing and verification on the bus-based DUT as well as the NoC-based DUT, writing parametrizable testbenches in both cases to support variable architecture parameters and solve different CNF problems - identifying some controller and functional unit bugs in the process. I designed the SerDes module together with Cher Rui. I designed the DFT features, added RTL for the IO pads, and wrote the testbench for post-PNR verification, including timing constraints. I helped Wesley and Kai with executing the PNR flow for the final iterations of the design, fixing DRC violations in the process. Lastly, I formed a comprehensive test suite from online SAT-solving competition databases.

C. Hrishi

The majority of my work revolved around the design of our NoC topology, with early explorations on different configurations, sync signals, routing schemes, and their tradeoffs over the summer with Aadi. I created an early software model using graph traversal algorithms to determine congestion points of a proposed network. I verified the network independently and then worked towards an integrated design with our 2D mesh topology, baseline controller, and clause banks that combine the individual efforts of the team. Through this process, I worked to verify basic functionality of the three components to discover and triage bugs related to improper rollback for the controller, dropped packets, etc. After we were confident that the behavior of our integrated design was expected, we placed and routed. From there, I worked towards simulating our chip, verifying behavior, and triaging bugs that were a product of undefined logic in our actual netlist vs. our synthesized netlist.

D. Wesley

I designed and helped verify the functional units and resolved edge-case bugs that arose in both top-level and unit-level verification. After verifying the functional units work correctly on a bus-based architecture, I designed and helped verify the clause banks in preparation for the final integration, which is our NoC-based architecture. Once we finished that, I helped set up the PnR scripting with pins and SRAM placed and also parameterized it so that we can have multiple per-module runs going at the same time, and also performed some PnR runs myself on the top-level design. In addition, I helped close out PD sign-off with Aaditya, Parithimaal, and Kai by connecting the datapath to the IO ring and running DRC/LVS checks and the final density filler, and along the way, I had to make a modification to the replace.py script.

E. Kai

At the beginning of the semester, the main task I was focused on was the feasibility and prototyping of Clause Learning. The work involved reviewing the literature on the mechanics of clause learning, reading source code for state-of-the-art software SAT solvers, and novel hardware adaptations of clause learning techniques in BCP accelerators. After review and high-level design, I moved on to prototyping the design in non-synthesizable form, System Verilog. There were many challenges in this process, eventually resulting in a need to pivot into helping with the verification efforts for the regression testing, functional units, and clause banks. The challenges and design decisions made for the clause learning will be expanded upon in the challenges section.

As for the work in the verification, this consisted of writing a Python model to provide a ground truth for our Base-Integration regression testing. This involved modeling the Base-Integration on a transaction level in Python. In addition, I wrote an in-depth verification of both Functional Units and Clause Banks. This involved simulating random network transactions, comparing behavior with a golden System Verilog software model, and confirming the existence of mathematically derived invariants.

For the latter half of the semester, my contributions centered mostly around physical implementation, including place and route, and post. The place and route involved following many of Wesley's leads, given his early work on place and route, while testing different configurations and ironing out certain DRC and LVS issues that arose from the place and route scripts.

For example, I orchestrated multiple place and route runs with different frequencies, problem sizes, network sizes, and input-output delays to find configurations that would place without violations and with higher densities. I also worked through certain DRC and LVS violations we faced that were associated with the placement of the metal nine stripes and the connection between the SRAMs and power. Alongside Hrishi, I also physically implemented certain guided place runs where modules were placed on a grid, though we opted to use the auto place and route, given its increased density.

I also contributed to some post-place and route implementation (importing files, adding rings, DRC, and LVS), though many of these were done with our entire team in person.

Finally, alongside Parithimaal, I helped design, verify, and implement our DFT features. These came relatively late in our design process and involved designing and testing all the DFT features described in section II-G2. These involved added observability, programmability, and a custom debug mode.

F. Aaditya

I was mainly focused on designing the on-chip communication protocols to allow the FUs and the controller to communicate with each other. Initially, this was the simple bus using priority-based arbitration that we used to prove the functional verification of the Functional units and controller. After which, I started designing the NoC and the routers that use a 2-D mesh topology and dimension order routing and round robin arbitration within the routers, allowing for more network throughput. I also worked on verification of the NoC wth Hrishi across various benchmarks to ensure the NoC correctly handles the packets being sent out. I also worked on verifying and determining bugs in the post-PnR simulation of the accelerator, as there were some x-propagation issues that were not caught in the earlier behavioural simulations, many of which boiled down to typos in the signalling we had used. Lastly, I also worked on PD with Wesley, helping him in fixing DRC/LVS violations and updating the ioring to fix the wbdmy DRC violations.

V. Challenges

A. Controller

During the controller design process, a major challenge was emulating recursive behavior using a hardware implication stack. Traversal of the SAT decision tree requires both speculative decision making and rollback on conflict, introducing significant control complexity. To manage this, we initially separated decision-making logic from rollback

handling by maintaining independent decision and implication stacks. However, due to strict area constraints, these structures were later merged, which introduced subtle corner-case bugs and increased verification complexity.

Additionally, the inherently recursive nature of the SAT solving process made naïve exploration of the decision tree impractical. Our initial approach drew inspiration from software-based SAT solvers, particularly through attempts to implement clause learning. However, the complexity of graph construction and state management in hardware made this approach infeasible within the given design constraints. This realization motivated the development of simpler yet effective heuristics, implemented through the Decision Heap, to accelerate convergence and enable feasible problem completion.

Ultimately, this experience reinforced the importance of approaching acceleration from a hardware-first perspective rather than directly translating software techniques.

B. BCP Functional Units

Many bugs were uncovered during top-level verification with the controller connected via a bus-based architecture, and most of them related to implication generation logic and the associated counters used in determining implications. From this verification effort, we learned the valuable lesson that even if we think we've unit tested each module, it's still possible there are edge cases we haven't uncovered yet.

Additionally, there was a misunderstanding of the expected behavior of the FU interacting with the network. By clearing up these misunderstandings, we strengthened our communication skills.

C. Mesh-Bank-Controller Integration

During our design process, we decided to move with the final integrated design in mind as a way to think about how our individual pieces interact. We communicated frequently on this end, but one of the challenges in our design process was trying to integrate components too quickly. To try and test for end-to-end functionality, we visited the idea of integration very frequently. However, this led to some bugs bleeding through to our integrated designs that might have been caught with thorough unit-level verification. While easy to resolve when discovered, most of our time during the debugging process of these issues was spent solely on figuring out what was going wrong.

While verifying our final top-level, which is the NoC-based architecture with a controller and clause banks, we found an interesting edge case behavior relating to output flit arbitration. Originally, we tried out a clause bank design that would loop back any implication flit generated from other functional units back into the input crossbar of the clause bank to speed up implication propagation to the local clause bank. While the design worked at the unit-level, we found it inadvertently causes congestion on the network since the loop-back is prioritized over incoming network traffic, so any traffic being sent to the local clause bank would be held up if too many output flits were generated, which fills up the network transmission buffers and causes **deadlock**. To relieve this deadlock, we opted instead to remove the clause bank loop-back and implement the loop-back at the router level, i.e., local banks would get their own flits back after passing through the router. In this way, we ensure the problem state propagates through all the functional units in a clause bank. Similarly, we found via parameter sweep that having the send/receive buffers in the NoC routers was more optimal, so we decided to remove the send/receive buffers in the clause bank implementation. In resolving this edge-case behavior, we further enhanced our communication skills, as the discussion involved lots of whiteboarding to make sure everyone was on the same page.

A lesson learned from this challenge would be to have comprehensive unit-level verification from the beginning that truly explores the coverage space of each unit extensively. In scenarios where it is important to note how different pieces interact, perhaps prioritizing smaller integrations that combine maybe two to three modules may help us model behavior as it interacts with pieces. Such a modular verification process may have helped us find bugs much earlier.

D. Clause Learning and Algorithm

Clause learning has been somewhat of a white whale for our team, and we would like to take this section to explain it in detail. In addition, this section serves to explain the chaotic nature of the SAT Solving Algorithm and how that contributes to difficulty in transaction-level verification.

Before beginning, we would like to define clauses and provide a high-level explanation of the algorithm used within our solver.

Clauses are defined to be the set of variables used within a single conjunction in conjunctive normal form.

As for the algorithms, we will build up from a naive brute force solver to the techniques modern state-of-the-art solvers such as MiniSat and its descendants use [2]. Currently, there is no known algorithm that is better than exponential time in the worst case. The most naive algorithm is a simple DFS tree search that backtracks every time a clause is found to be false. A simple improvement is the DPLL [3] algorithm, which is a tree search algorithm. One of the improvements DPLL includes is when a unit clause is found, IE, a clause that simplifies to one entry due to all other literals in the clause being false, it is implied that said literal must be true. By globally propagating this message, the DFS can be solved quickly. This set of implications also essentially creates another DFS search, where implications are found, propagated, which can, in turn, derive more implications, causing a minor tree search within every step of the DPLL major tree search. This is the algorithm used with our current chips, passing the implication messages on the network on the chip. MiniSAT and its modern solver descendants, such as Glucose [1], apply a whole plethora of heuristics on top of the DPLL core. One of the most critical including clause learning, where certain searches in the tree are used to create new clauses, which constrains the search space. Another is selecting more active variables, variables that show up in more

clauses and participate more in the implications. This is one of the heuristics we have implemented on our chip.

It turns out that the nature of this clause learning process makes it quite difficult to implement within hardware when used in conjunction with a network on a chip.

The first issue is the complexity associated with clause learning. Because clauses must be dynamically allocated and reallocated during the algorithm, complex allocation mechanisms and algorithms need to be run on the controller. Many papers, such as Satin [4], offload this complex logic to a CPU, but we wanted to use a controller given the smaller area and the lack of IO overhead with an external CPU.

The second issue is a difficulty in transaction-level verification due to the chaotic nature of SAT on a Network on Chip. This also impacted our verification of the top-level design, where we do not have a transaction-level golden model, only an output golden model. The reason is as follows. Because our algorithm is a modified DPLL search and has a major and minor pass of DFS, the choice of which branch is explore first is critically important. When the choice of the branches is simple, it is easy to design a golden model that can model every transaction. When the DFS branch is chaotic, meaning it depends on small changes within the design, a transaction-level model must be able to model every small interaction within the chip to get the same branches. This is somewhat problematic as the golden model would only be slightly more abstract than the RTL, making the risk of conceptual bugs in both high. This is also associated with a higher time required to develop the golden model. It turns out that a network on a chip and heap variable sorting are exactly the mechanisms that make both the major and minor DFS branch selection chaotic. Depending on which packet arrived earlier, one part of the implication tree may be explored before another. This causes small changes in packet timing and network topology, causing potentially massive effects on timing. One race condition might quickly find a dead end, causing a quick backtrack, while another sequence of packets may cause an unfruitful branch to be explored for a long time. In addition, using the heap to select for more common variables is also contingent on the implications seen on the network, making the tree search in the major DFS also chaotic. This double chaos makes writing a transactional model very difficult. As such, we opted for end-to-end testing for our final top-level verification.

Because of these two major reasons, we were not able to implement clause learning in software. But given how important it was in our proposal, we were able to include a debug mode that allows network commands to be sent and received by an external CPU (FPGA). This allows for the complex allocation logic to be written in software on an FPGA soft or hard core, and mapped to clause clears and clause writes commands that allow for learned clauses to be put on the device.

Though extensive testing of this feature has not been completed due to time constraints, we were able to make smaller initial tests that show the feasibility and confirm the ability to extract all events correctly and program clauses. We will be exploring clause learning during bring-up.

E. Speed Up

In this section, we would like to highlight some of the challenges we had associated with the speed-up. The major issue is that, though on average we are faster, these results are quite inconsistent, ranging from 11x to 0.07x, as shown in figure 1.

When analyzing which test cases did well and which did poorly for oPOSsum, we see that oPOSsum does quite well on yes (satisfiable, geometric mean speed up of 4.79x) cases and does poorly on no (unsatisfiable, geometric mean speed up of 1.65x) cases.

The reasoning for this is the fundamental difference in the algorithms used between Glucose and oPOSsum presented in the previous section. Because Glucose has clause learning, it fares far better in cases where learned clauses can quickly cut down search space, while oPOSsum must search the entire space with minimal cutting (only DPLL implications). On the other hand, yes, cases where the entire tree does not have to be searched, oPOSsum is faster because of its raw speed through the tree.

Again, the challenges with introducing clause learning and other advanced heuristics in hardware limited our speedup. That said, with the opportunity of an external CPU available, we believe we can overcome this challenge and see the no speed up get closer to the yes speed up.

The other challenge we note in relation to speeding up is the SUp F1/F0 columns, which represent the speed of the filter for complete on relative to the filter for complete off. The geometric mean for this speed up is 0.93. This was somewhat counter to our initial theoretical understanding. The intention was that the filter for complete being on would have a speed up as the overhead of having to transmit all network packets to the IO was removed. That said, we see somewhat of the opposite. This was a challenge as it did not meet our theoretical understanding. That said, after some digging, we strongly believe the cause is due to the chaotic nature of our implemented algorithm presented in the previous section. By changing what is sent out to the IO, there is a small difference in network behavior, which could lead to large but unpredictable changes in the speed. This is what we see with some being a decrease in performance, but others providing larger speed-ups. In general, this result is somewhat good for our team as it means the overhead of transiting network data out to IO is not as extreme as we thought, allowing for clause learning to be used in debug mode with little overhead.

F. Power

The final challenge, and the one we could have improved the most on, was our power consumption. Our power consumption is quite high. The breakdown of power is as follows: 8.6 mW Internal Power, 243.5 mW of Switching Power, and 2.6 pW of Leakage Power. Given the high Switching Power, we plan to remedy power issues with lower clock frequencies or a more intensive thermal solution. The reason for the high power is

due to an expensive clock tree, which was due to synthesizing at a higher frequency.

VI. Design Details

A. High-Level Diagram

See Figure 2

B. Controller

The controller orchestrates the backtracking algorithm, maintaining the program's state throughout execution. It makes choices through the decision module and synchronizes with functional units as they produce implications. When a conflict occurs, the controller backtracks up the decision tree, ensuring all functional units revert accordingly.

The main components of the controller include: The Decision_Implication Stack, the Decision Heap, and a finite state machine.

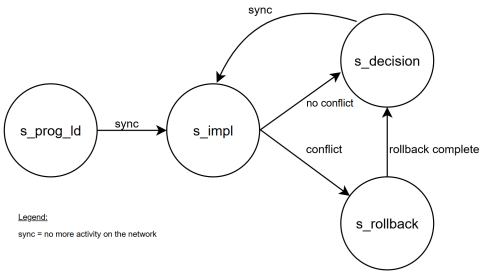

Fig. 5. Controller FSM

The **Decision_Implication Stack** is implemented as an SRAM array, records all implications propagated across the network that must be reverted during rollback. An implication is pushed onto the stack whenever the controller is in the Implication state and receives a packet with an implication control word. Entries are only popped during rollback operations. Furthermore, whenever a new decision is made, the decision is pushed onto the stack, where, during a conflict, the corresponding entry is popped, and its inverse is used to facilitate rollback.

The **Decision Heap**, implemented as a shift register, maintains a pseudo-most-frequent-variable list across the entire SAT problem. It is constructed and updated during the Program Load phase to guide future decision-making. Furthermore, if the conflict heuristic is enabled via DFT functionality, the decision heap will update during runtime when receiving conflicts for added acceleration.

The finite state machine consists of 4 main states: Program Load, Decision, Implication, and Rollback.

The **Program Load state** focuses solely on constructing the pseudo-heap. The architecture includes an SRAM that tracks how many times each variable is encountered, alongside a register list that stores the most frequently seen variables. Since loading occurs on a variable-by-variable basis, the count for each variable can be incremented and compared every cycle. The updated count is then compared with the variable at the head of the list. If the new variable's count exceeds that of the head, the list shifts downward, placing the new variable at the top. Otherwise, the list remains unchanged. This mechanism effectively maintains a pseudo "most-seen" list, ensuring that the most frequently accessed variables remain on it—thereby improving the decision-making heuristic.

The **Implication state** enables functional units to propagate variable assignments and store their resulting implications within the controller for potential rollback. All implications are broadcast across the network, received by the controller, and recorded in the implication stack. Any conflicts that arise are detected and subsequently resolved during the rollback phase.

The **Decision state** is entered when no further implications remain on the network, and a new variable assignment must be selected. The decision is chosen based on the variable's frequency within the overall SAT problem or the frequency of conflicts with that variable, depending on the DFT settings during program loading, ensuring that neither a decision nor an implication has already been made for that variable. Once selected, the decision is broadcast to the network, and its assignment is pushed onto the decision stack. In the event of a conflict, the system backtracks through this decision stack to explore the alternate branch of the decision tree.

The **Rollback state** is triggered when a conflict is detected during the Implication state. In this phase, all implications derived from the most recent decision must be undone. The controller sequentially pops implications from the implication stack and broadcasts rollback signals across the network until it reaches the last recorded decision point. The controller orchestrates the backtracking algorithm, maintaining the program's state throughout execution. It makes choices through the decision module and synchronizes with functional units as they produce implications. When a conflict occurs, the controller backtracks up the decision tree, ensuring all functional units revert accordingly.

C. BCP Functional Units

The BCP functional units, pictured in Figure 6, store one clause each, and only process information from the data word within the flits taken in from the network, with flits being defined as a packed struct, as shown below:

```
typedef struct packed {
    logic [HEADER_WIDTH-1:0] header;
    ctrl_w_t ctrl_w;
    data_w_t data_w;
} flit_t;
```

Listing 1. Flit SystemVerilog struct definition

Additionally, the data_w (data word) is a union of structs, with the interpretation differing based on the ctrl_w (control word), and the union and struct definitions are shown below:

```
typedef union packed {
    struct packed {
        logic [VAR_ID_WIDTH-1:0] var_id;
```

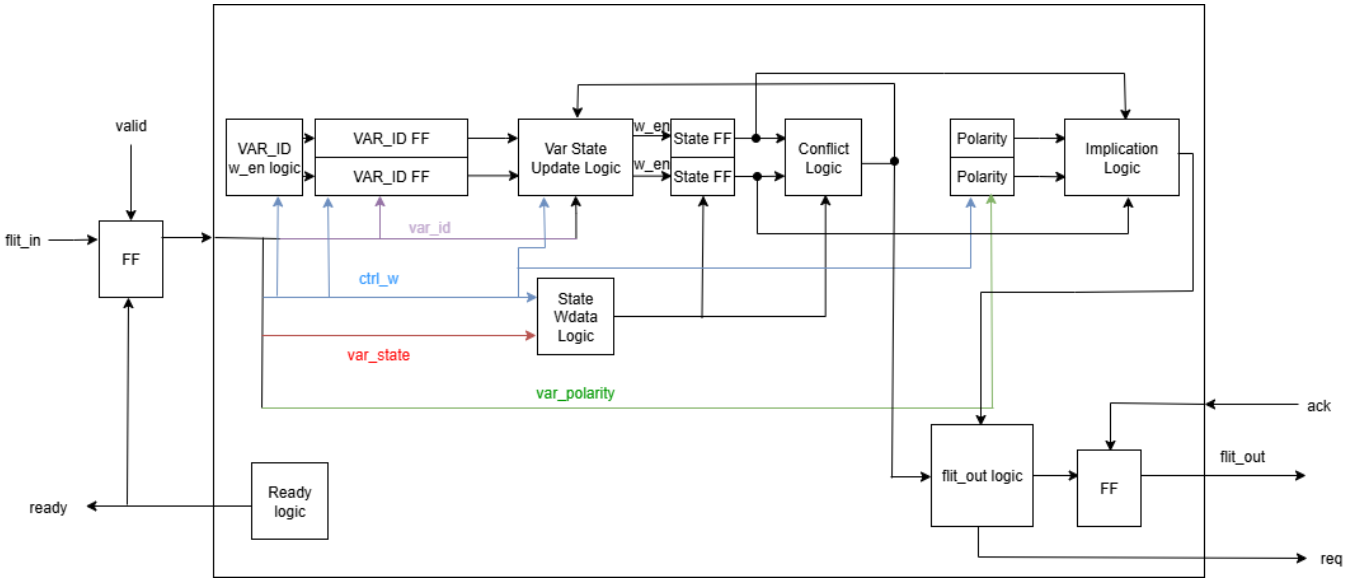

Fig. 6. BCP FU

```
        logic polarity;
        logic valid_prog_load;
    } prog_load_t;

    struct packed {
        logic [VAR_ID_WIDTH-1:0] var_id;
        logic [1:0] state;
    } decision_t;

    struct packed {
        logic [VAR_ID_WIDTH-1:0] var_id;
        logic [1:0] reserved;
    } rollback_t;
} data_w_t;

typedef enum logic [NUM_CNTRL_WORDS_B
    -1:0] {
    prog_load =      0,
    conflict =       1,
    decision =       2,
    implication =    3,
    rollback =       4
} ctrl_w_t;
```

Listing 2. SystemVerilog struct definitions for data words and control words

If the control word is a prog_load, the FU reads the fields from the prog_load_t struct within the union; if the control word is a decision or implication, the FU reads the fields from the decision_t struct within the union; if the control word is a rollback, the FU reads the fields from the rollback_t struct within the union. The conflict control word is only used to propagate info about conflicts to the network and is not used as part of the FU's processing.

The FU stores critical information about the clauses: variable IDs, the state of the variables, and the polarity of the variables (negated or non-negated in the clause, represented as 0 or 1, respectively). The variable states are encoded as 2 bits, with each state representing the following:

1) 00: variable assigned 0/false
2) 01: variable assigned 1/true
3) 10: variable unassigned, post-rollback/post-prog_load
4) 11: variable unassigned, post-reset

To interact with the network, the FU uses both a valid-ready and req-grant handshake. When the FU is ready to accept a flit from the network, it will assert its ready line, and if the network asserts and sends a flit with a valid asserted, then it will update the FF storing the current flit being processed by the FU. Similarly, if the FU needs to send out an implication or conflict, it will signal it wants to put a flit on the network by asserting its req line, and will only de-assert its req line once the network has given it a grant (re-named to ack in the design for simplicity). The valid-ready and req-grant handshakes ensure that the FU correctly receives and sends flits to the network via a credit-based system.

Upon taking in a flit, it will first see if it is a prog_load flit, in which case it will assign one row of clause information in the corresponding FFs (var_id, state, polarity). Additionally, it will look at the valid_prog_load field to see if the prog_load flit is valid, as in some cases an "invalid" prog_load flit is sent to indicate a clause with less than three variables. Then, the FU checks if an assignment is to be made and whether or not a conflict occurs due to a prior decision/implication. If so, it will output a conflict flit with the variable ID on which the conflict occurs. If, after processing a valid assignment, the FU sees that all but one variable are assigned with states opposing the variables' polarities, an implication occurs, as the last must be set to match its polarity, and the FU will output an implication with the corresponding variable ID to be set, as well as the state it should be set to. Otherwise, the

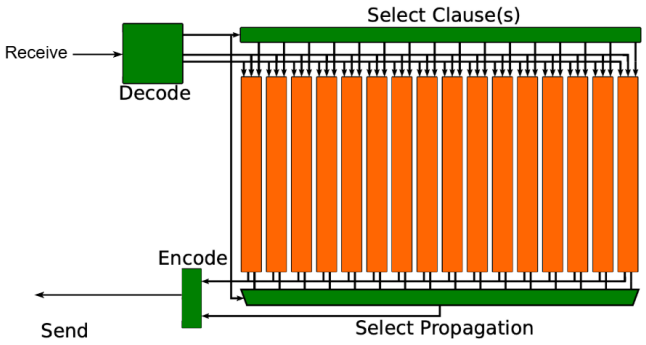

Fig. 7. Clause Banks, with μarch based on [4]

FU will process the assignment and simply update the state FFs accordingly.

D. Clause Banks

We decided to also implement a clause bank (shown in figure 7) to reduce pressure on the NoC. It essentially is just a wrapper containing multiple BCP FUs, and internally performs arbitration to determine which FU's flit to send out to the network. The clause bank is based upon that found in Satin [4]; however, we ended up removing the local send/receive buffers and the loop-back, for reasons previously mentioned in section V-C.

E. Mesh Network

The mesh network provides the structure for communication to happen across the chip. The network uses a 2D-mesh topology to connect the clause banks to each other and to the central controller. It also connects it to our SerDes port for debugging and disseminating the program load instructions to the clause banks.

The routing network uses the following interface to send data from a transmitting port to a receiving port.

```
interface sw_channel();
    import sw_network_types::*;

    bit ready;
    bit valid;
    sw_flit_t flit;

    modport tx (
        output valid, flit,
        input ready
    );
    modport rx (
        output ready,
        input valid, flit
    );
endinterface : sw_channel
```

Listing 3. SystemVerilog network interface definition

The Ready signal lets the transmitting port know that the receiving port is ready to receive data, and the Valid signal lets the receiving port know that the transmitting port is sending valid data. Flit is the data type of our network packet, which contains source and destination headers, and the data regarding variables generated by the FUs.

The mesh network is anchored by our routers, which have 5 ports (North, East, South, West, and Local), where NESW connect a router to the rest of the network, and Local connects the router to the clause bank, controller, or SerDes module.

The routers enable us to send flits from one port to a subset of all ports or all other ports, depending on whether the flit is a directed message or a broadcast message. The router consists of receive and transmit buffers for all ports, routing computation modules for all ports, a switch allocator that arbitrates between requesting ports, and a crossbar that directs packets from a requesting port to its destination ports.

The receive and transmit buffers are simply circular buffers using a FIFO policy with read and write pointers. The read and write pointers have a parity bit, which allows the hardware to determine when the FIFO is full or empty.

For route computation, we settled on using dimension order (XY) routing as it was a simple routing mechanism that keeps computation costs small, allowing us to run at a higher clock frequency without pipelining in the router. XY routing works by routing all packets in the east-west direction first till the desired columns are reached, before routing them in the north-south direction till the desired row and column are reached.

To compute the routing for flits being broadcast, we do the following:

- If a broadcast is being received on the East or West ports, we route it to all other ports, including local ports
- If a broadcast is being received on the North or South ports, we route them to the reciprocal NS port and the local port
- If a broadcast is being sent from the local port, we route it to all other ports

Based on the nature of our SAT problems, we believe that the number of transactions on the network will not be extremely high, and thus, the perceived exertion of the east-west links will be minimized.

The 5 route computation modules output 5-bit vectors that are then used in our switch allocator to determine which requesting port is granted. The switch allocator takes in these requesting vectors, and then checks whether all transmitting buffers are not full for the port that currently holds priority. If all transmitting buffers are able to take in flits, it grants the requesting port and moves the priority to the next port in order of NESWL. In doing so, we implement a round robin arbiter to ensure none of the ports are getting starved. In the case where the port holding priority does not meet the criteria, the switch allocator checks the next port in order and moves the port priority to the next port after the winning requestor.

The switch allocator outputs a final routing vector and grant vector that is used by the crossbar to determine which

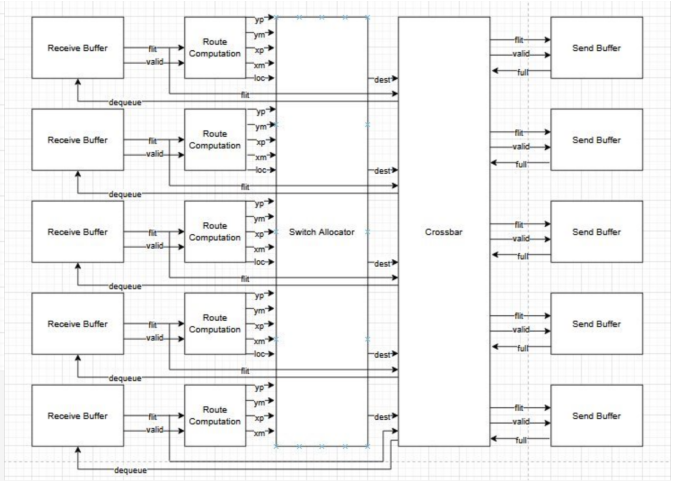

Fig. 8. Router control logic top level

requesting ports' flit gets routed to which destination ports transmit buffers.

F. SerDes

The SerDes module sits between the IO pins (those suffixed with _F in figure 9) and a designated router in the NoC. It is used for loading a CNF problem onto the chip at the start of the program (program loading), as well as reading out the satisfiability result of the problem at the end of the program. When DFT features are enabled, these pins will also be used to broadcast every flit that is on the network, for debugging purposes. The module contains two independent channels, Tx and Rx, for transmitting flits to and receiving flits from the FPGA, respectively.

```
module jtag_dut_serdes (
    input logic SCLK,
    input logic CS,
    input logic [7:0] MOSI, // Command or
        Data
    output logic [7:0] MISO, // Contains
        Flit Data
    input logic MOSI_CTRL, // assert when
        MOSI has valid information
    output logic MISO_CTRL, // assert
        when MISO has valid information
    output logic ASIC_RDY,
    input logic FPGA_RDY,
    input logic FFC_DUT, // enable/
        disable DFT features
    input logic rst,

    sw_channel.tx    jtag_tx, // to the
        router
    sw_channel.rx    jtag_rx // from the
        router
```

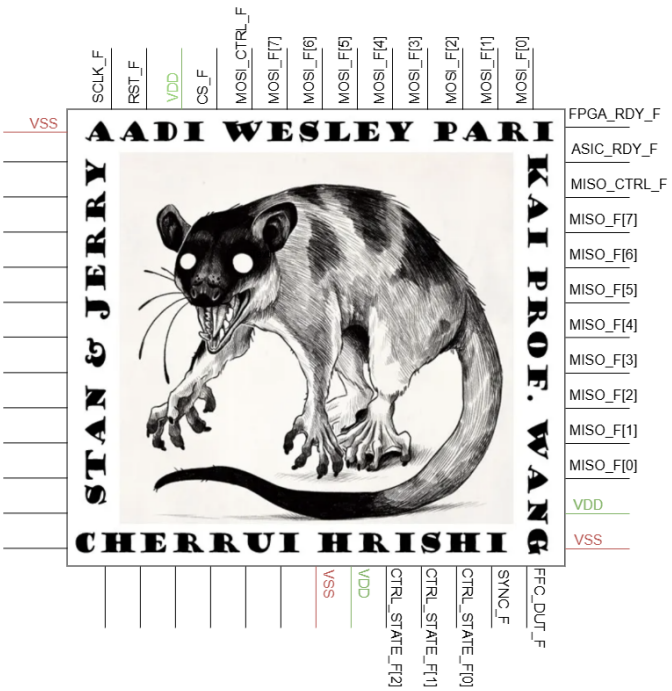

Fig. 9. Chip pinout diagram, with AP art based on [5]

```
);
```

Listing 4. SystemVerilog IO for the SerDes module

The MISO/MOSI width in the above code block highlights how data is transferred between the FPGA and chip a byte at a time, with ready signals to backpressure transmission from either side when necessary.

The receive channel takes input from the FPGA and dictates the program loading. It follows a SPI-like format where an arbitrary-sized flit is loaded one byte at a time, and each byte load requires a CMD-DATA format similar to SPI. Once all the bytes have been loaded into the SERDES module, a SEND command is given, upon which the module sends the flit into the NoC, either to the controller or to an appropriate functional unit for program loading. To ensure flexibility in constructing the flit, this module is designed such that each byte load can be done from an arbitrary offset from the LSB of the flit, and can optionally write less than 8 bits should the flit bit-width not be a perfect multiple of 8.

The transmit channel takes input from the design and sends it to the FPGA, one byte at a time. The FSM for this is much simpler, as it is assumed that the FPGA is always ready to receive a byte at any arbitrary cycle, and the reconstruction into a flit can be done in software. The diagram below shows the states used for each channel in the SerDes module.

```
typedef enum logic[7:0]{
 START_B,
 END_B,
 DATA_R,
```

```
 CLEAR,
 FPGA_TO_ASIC
} reciever_state_t;

typedef enum {
    IDLE,
    SEND_FLIT
} transmission_state_t;
```

Listing 5. SystemVerilog struct definitions for receiver and transmit channel states

VII. Testing Instructions

A. Final Integration Verif

To verify the final design with our 2D mesh topology, clause banks, and the controller, you can follow the following steps:

```
cd /groups/ece427-group2/FINAL/rtl/sim
./run_integrated.sh {FOLDER_NAME}
./run_integrated_post.sh {FOLDER_NAME}
```

The script run_integrated.sh is for the pre PnR test bench and run_integrated_post.sh is for the post PnR test bench.

The output is piped to a user-specified directory, where a separate file is generated for each test case in the regression suite. If the resulting assignment satisfies the SAT problem, the message "SAT Solution is correct" is printed and followed by a value of 1. Otherwise, a value of 0 is printed, along with the specific clause that violates satisfiability. This status message appears at the end of every output file.

It is worth noting that verifying a SAT solution is computationally inexpensive and is performed in constant time, within the SystemVerilog testbench located at /groups/ece427-group2/FINAL/rtl/hvl/top_tb_io_jtag.sv. The testbench checks each clause for satisfiability using the following code snippet where cnf_matrix is the initial encoded SAT problem while sat_solution is the output from the accelerator:

```
begin
    int sat_solution_correct = 1;
    int var_id = 0;
    bit satisfied;
    int clause_unsat[3];
    int var_id_clause[3];
    foreach(cnf_matrix[clause_is]) begin
      $display("clause %0d problem: %0d,
          %0d, %0d", clause_is, cnf_matrix
          [clause_is][2], cnf_matrix[
          clause_is][1], cnf_matrix[
          clause_is][0]);
      var_id_clause[0] = cnf_matrix[
          clause_is][0] > 0 ? cnf_matrix[
          clause_is][0] : -cnf_matrix[
          clause_is][0];
      var_id_clause[1] = cnf_matrix[
          clause_is][1] > 0 ? cnf_matrix[
          clause_is][1] : -cnf_matrix[
          clause_is][1];
      var_id_clause[2] = cnf_matrix[
          clause_is][2] > 0 ? cnf_matrix[
          clause_is][2] : -cnf_matrix[
          clause_is][2];
      var_id_clause[0] -= 1;
      var_id_clause[1] -= 1;
      var_id_clause[2] -= 1;
      $display("clause %0d assignment: %0
          d, %0d, %0d", clause_is,
          sat_solution[var_id_clause[2]],
          sat_solution[var_id_clause[1]],
          sat_solution[var_id_clause[0]]);
      $display("----");
      satisfied = 0;
      for(int i = 0; i < 3; i++) begin
        if(cnf_matrix[clause_is][i] == 0)
            continue;
        var_id = (cnf_matrix[clause_is][i
            ] > 0) ? cnf_matrix[clause_is
            ][i] : -cnf_matrix[clause_is][
            i];
        var_id -= 1;
        if((cnf_matrix[clause_is][i] > 0)
            == sat_solution[var_id])
            begin
          satisfied = 1;
        end
      end
      clause_unsat = cnf_matrix[clause_is
          ];
      if(!satisfied) begin
        $display("clause %0d not
            satisfied: %0d, %0d, %0d",
            clause_is, clause_unsat[2],
            clause_unsat[1], clause_unsat
            [0]);
        sat_solution_correct = 0;
      end
    end
    $display("SAT Solution is correct: %0
        d", sat_solution_correct);
end
```

For power simply take the dump.fsdb generated from any of the previous runs in the

```
/groups/ece427-group2/FINAL/rtl/
    sim/sim
```

directory and use the following commands.

```
fsdb2saif -licqueue dump.fsdb -s
    top_tb_post/top_top_dut
```

Place the generated .saif renamed to dump.fsdb.saif file into

```
/groups/ece427-group2/FINAL/rtl/synth
```

as well as the .ddc file in

```
/groups/ece427-group2/FINAL/rtl/param\
    _synth/synout/synth.ddc
```

Then in the

```
/groups/ece427-group2/FINAL/rtl/synth
```

directory run the following:

```
make power_vcs
```

VIII. Acknowledgements

We would like to thank Apple for sponsoring this class, and TSMC for allowing us to use their 65nm process node for tapeout. We would also like to thank the team at Muse Semiconductors for organizing our tapeout logistics. Last, but certainly not least, we would like to thank the University of Illinois Urbana-Champaign for running this class, Professor Dong Kai Wang and the TAs, Stanley Wu and Jerry-Tianchen Wang for their unwavering support throughout the semester, and our Apple mentor, Evan Lissoos, who provided valuable insights during our RTL/architectural development cycle.

References

[1] G. Audemard and L. Simon, "On the Glucose SAT solver," *International Journal on Artificial Intelligence Tools (IJAIT)*, vol. 27, no. 1, pp. 1–25, 2018.

[2] N. Eén and N. Sörensson, "An extensible sat-solver," in *Theory and Applications of Satisfiability Testing* (E. Giunchiglia and A. Tacchella, eds.), (Berlin, Heidelberg), pp. 502–518, Springer Berlin Heidelberg, 2004.

[3] M. Davis and H. Putnam, "A computing procedure for quantification theory," *J. ACM*, vol. 7, p. 201–215, July 1960.

[4] C. Zhu, A. C. Rucker, Y. Wang, and W. J. Dally, "Satin: Hardware for boolean satisfiability inference," 2023.

[5] E. Lovejoy, "Yapok or water opossum study." Tumblr, Dec. 22, 2017. [Online]. Available: https://www.tumblr.com/evanlovejoy/168863318666/yapok-or-water-opossum-study-look-at-those-crazy. [Accessed: Dec. 21, 2025].