oPOSsum: Network-on-Chip based SAT Solver Accelerator

ECE 427 Project Proposal

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 propgation (BCP), as it is the bottleneck in traditional SAT solving algorithms.**

II. Introduction

Boolean Satisfiability (SAT) is the problem of finding a set of boolean inputs which finds an assignment of values to the variables in a Product-of-Sums (POS) boolean expression that makes the expression evaluate to true. It has many real-world applications, including but not limited to: hardware verification, logic synthesis, software verification, and even applications as simple as scheduling (classes, sports tournaments, etc). Previous works have implemented software solvers based on the Davis–Putnam–Logemann–Loveland (DPLL) algorithm, but many works thereafter have shown that most of the computation time (80-90%) [1] is spent in doing Boolean Constraint Propagation (BCP). BCP essentially updates which variables still need to be assigned in all clauses still not satisfied (i.e., evaluating to true) after one variable gets assigned a certain value.

We propose an architecture for accelerating BCP which parallelizes updating a variable with evaluation of the clause. Our architecture is based on HW-BCP, a design proposed by Park et al. [1]. However, since we are not able to expose the CAM/SRAM bitlines like they did in the paper, we modified our functional units to only use D flip-flops for storing the clauses, variable values, and polarities.

That said, we leave ourselves open to certain hardware extensions that would make it easy to add clause learning, one of the modern techniques used to accelerate SAT.

III. Architecture

A. Network On Chip

After considering various Network On Chip architectures, we are leaning towards a combination of the Mesh and Torus frameworks, using a flow control digit or FLIT as a unit of communication.

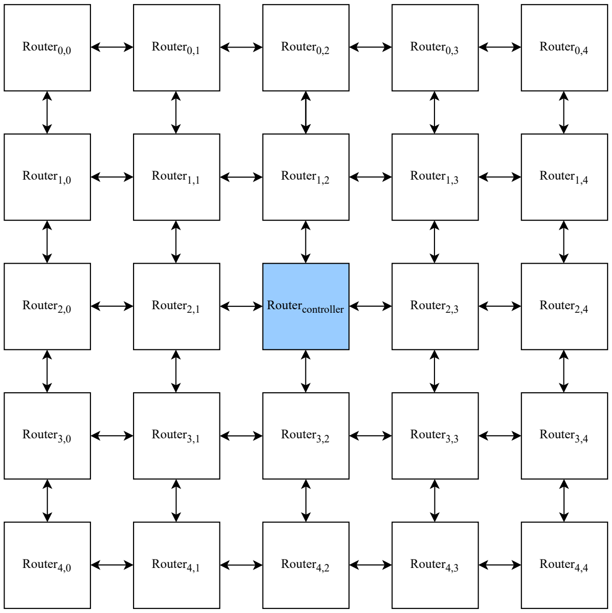

Fig. 1. Top Level Block Diagram

There are 3 types of flits, HEAD, BODY and TAIL. HEAD stores the routing information and destination address, BODY stores the actual data to be transmitted, and TAIL is responsible for closing the virtual channels behind it.

1) Switch Boxes: Each BCP unit is connected to a single switch, and each switch is connected to four neighboring switches [2]. The mesh topology services many operations, including but not limited to the initial distribution of clauses to BCP units from the controller, the propogation of contradictions back to the controller, as well as the standard BCP protocol.

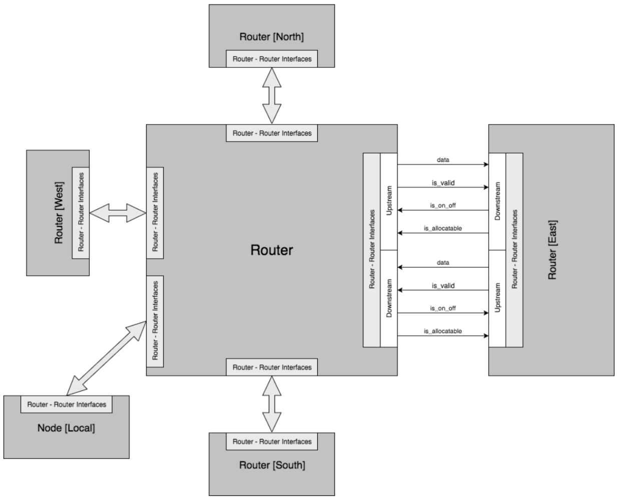
Router [North]
Router - Router Interfaces
Router - Router Interfaces
Router [West]
Router - Router Interfaces
Router - Router Interfaces
Router - Router Interfaces
Router
Router - Router Interfaces
Upstream
Downstream
data
is_valid
is_on_off
is_allocatable
data
is_valid
is_on_off
is_allocatable
Downstream
Upstream
Router - Router Interfaces
Router [East]
Router - Router Interfaces
Router - Router Interfaces
Node [Local]
Router - Router Interfaces
Router [South]

Figure 2: Detailed view of the external interface

Fig. 2. Switch Block Diagram

Let us denote flits and switches previously along a message's path as upstream, and any flits and switches that are further along the path as downstream.

To account for contention within the network-on-chip fabric, each of the five upstream connections (each cardinal direction and the local resource port) have a dedicated circular buffer, as shown in Figure 2. The routing unit then determines which upstream switch is allowed to communicate with which downstream switch, including accounting for verifying if a switch is busy with other communications.

Compared to standard networking protocols like TCP, such network-on-chip practices are unable to allow for dropped packages. Therefore to account for potential head-of-line blocking and multiple streams of data that must not interact with each other, we propose the use of the aforementioned virtual channel routing [3], shown in greater detail in Figure 3.

B. BCP Functional Units

1) Boolean Constraint Propagation (BCP): In software SAT solvers, Boolean Constraint Propagation (BCP) is a crucial step in determining whether or not a Product-of-Sums (POS) boolean expression is satisfiable (i.e., ∃ some assignment of values to variables in the expression which makes the expression evaluate to true). BCP focuses on evaluating the result of one sum (clause) in the product (expression), and the search is pruned/finished based on the following three conditions after assigning a selected variable a certain value:

1) Satsified: the assignment results in the clause evaluating to 1.
2) Conflict: the assignment results in a variable being in two states at once (i.e., the variable was already assigned a value in a previous iteration).
3) Unit Clause: the assignment still results in the clause evaluating to 0, but there remains only a single unassigned variable in the clause which can make it evaluate to 1 if assigned a certain value.

Then, after the clause is evaluated with the new assignment, the result must be propagated to all of the other clauses in the expression to update the search.

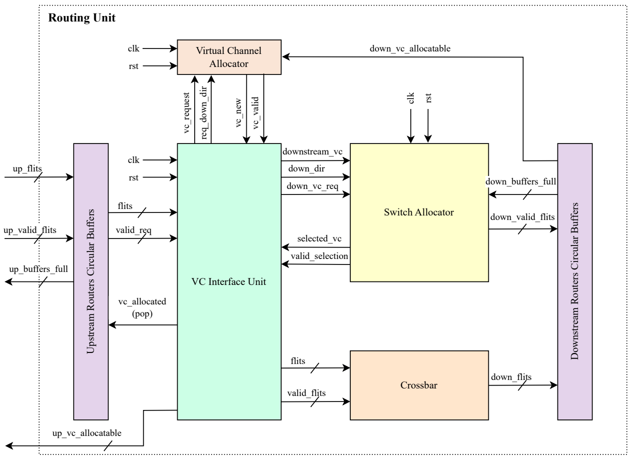

Fig. 3. In-Depth Routing Unit

As previously mentioned, the functional units are mainly focused on parallelizing Boolean Constraint Propagation (BCP) and are based on the design implemented by Park et al. [1]. For the clause unit, We initially looked at the design proposed by Zhu et al. [4], but quickly determined based on area estimations that their design was not as scalable for our area constraint as we hoped. The HW-BCP clause units use a custom architecture which grants a very high degree of parallelism. The main feature which results in a very high degree of parallelism is their custom CAM-SRAM fusion which connects the CAM matchline to the SRAM wordline, where the CAM stores the variable ID, and the SRAM stores the value assigned to the variable. In doing so, this allows the variable lookup and update to happen simultaneously.

However, since we cannot expose the wordlines on the SRAM IP we are provided, we opted to use a D flip-flop oriented design for our clause units. We store the variable ID into a clause-length array of var_id-width registers if that variable appears in the clause. Then, we use another clause-length array of 2-bit registers to store the variable value (0, 1, or ”x” for unassigned variables), and another clause-length array of FFs to store each variable's polarity (whether or not it appears negated in the expression, with a 1 denoting a non-negated variable).

Each BCP Function Unit is organized as a bank of clause units. Each clause unit is responsible for locally holding clause information, accepting new state, and emitting new state.

These clause units are then organized into banks to share hardware resources when interfacing with the NoC.

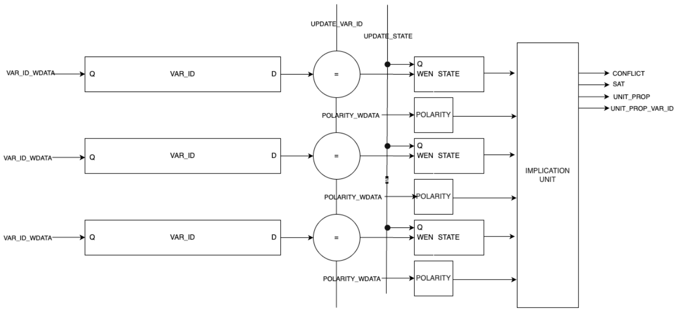

Fig. 4. Clause Unit

In the Clause Unit, a variable ID is given as input, as well as a desired value to assign to it. Then, the unit will evaluate the previously mentioned conditions as follows:

1) Satsified: the assignment is forwarded into the evaluation logic, and if the evaluation logic returns a 1 with this new assignment, the clause is satsified, and asserts the satisfied signal.
2) Conflict: using the inputted variable ID, the new assignment is checked against the previously assigned value by indexing into the variable value FFs; if it is found that the assignment conflicts with the previous assignment, then the BCP unit asserts the conflict signal.
3) Unit Clause: within the BCP unit, two counters are used to track the number of variables present in the clause and the number of assigned variables in the clause. If the number of variables assigned + 2 (because we are forwarding the new assignment) is equal to the number of variables present in the clause and the clause is still not yet satisfied, then the BCP unit asserts the unit clause signal. A flag is also raised in case the NoC is busy, such that the clause may be propagated later.

Wesley has already written some RTL logic for the Clause Units and synthesized it using the FreePDK45 standard cell library, so we have some area estimates based on the number of variables the Clause unit supports:

1) 10-bit variable ID, 5-variable clauses: 641 μm^2
2) 10-bit variable ID, 500-variable clauses: 58272 μm^2

At the bank level, a set of Clause Units share the update lines, and a decoder and encoder. The decoder is responsible for taking commands of the NoC and translating them to actions within the Clause Units. All the implications generated by the Implication Unit are then arbitrated and encoded such that any Unit Clauses can propagate their implications through the encoder.

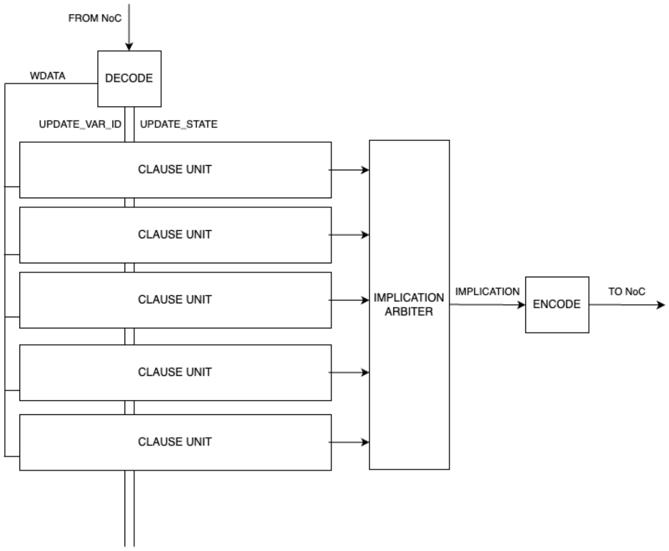

Fig. 5. Clause Bank

With this we estimate the area that a 512 variable problem with 512 clauses with 5 variables per clause could fit comfortably at around 0.328 mm^2.

If our banks wanted to support clause learning, another module would be included in the implication arbiter that tracks the implication level, a mechanism used to tag implications with the order in which they were generated. This tagging can then be used in the controller to isolate the most important implications and write a new clause representing said implications.

In addition to accommodating clause learning, which is further described in the controller section, clause banks will additionally propagate an implication level with each implication message. Given that in a 2D mesh, propagations can happen asynchronously and in parallel, it is important to retain logical order. This is achieved with the implication level, where greater implication levels logically follow those with lower implication levels. Each clause unit stores the implication level as a state. The state is updated to max(current implication level, incoming implication level) and will produce implications at current implication level + 1.

C. Controller

To run the DPLL algorithm, we will need a controller to generate the clauses and distribute them to the various BCP FUs. We have designed a state machine controller consisting of only crucial components of the algorithm: Decision, implication and rollback. This allocating more area for clauses in the form of functional units.

Along with controller-logic, we are currently in the process of deciding storage for the implication and decision stack, both key components in the controller. On the one hand SRAMs are much smaller in area, however they also come with the penalties of a 2 cycle write in comparison with a traditional flip flop.

The basic software model for DPLL has been written, and the scaling laws for how much memory a certain problem requires have also been calculated.

In addition to the basic DPLL algorithm, we are currently considering including the hallmark Clause Learning Technique that modern software solvers employ. This involves reconstructing the implication tree and applying the BFS algorithm, continually searching the tree until the BFS queue only contains one element. This is called the 1 UIP cut of the implication graph. This ensures the shortest possible clause that can be added to the BCP that making it impossible to return to the unfruitful segment of the DPLL tree.

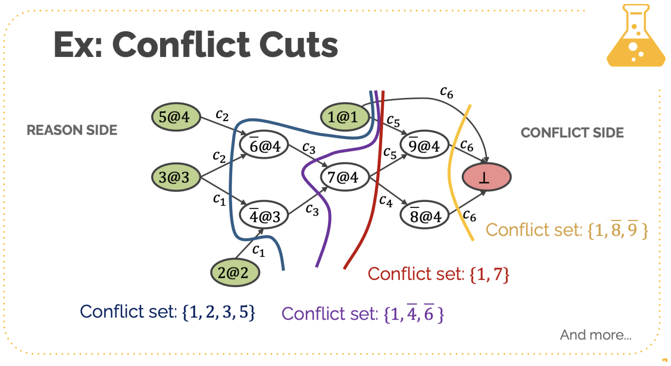

Fig. 6. Clause Learning Graph Example

The controller modification will require states that involve, the BFS and an additional memory mechanism to store the graph edges in SRAM. Additionally, once the analysis has been completed, the controller will create a program load to insert the new clause, until the memories saturate with clauses.

IV. Original Timeline

A. *Algorithm Analysis (Late May)*

Identify key markers to accelerate and improve algorithm performance via software modeling.

B. *Architectural Design (Early July)*

Finalize architectural design, behavioral and functional specifications

C. *RTL Design & Verification (July)*

Finish first pass Behavioral RTL for individual components, NoC, Functional Units start verification for functional correctness and coverage.

D. *RTL Finalization and Synthesis (Early August)*

Addition of critical DTF components, area and timing analysis and correction.

E. *Floorplanning and IP Analysis (August)*

Determine geometric locations of SRAMs, CPU, and Functional Units.

F. *Physical Layout (September)*

Automating Physical design flows and setting up toolchains.

V. Current State

Completed:

- Algorithm Analysis (Base Design, Clause Learning, NoC)
- Architecture (Base Design, Clause Learning, NoC)
- RTL Design (Base Design)

In Progress:

- RTL Design (Clause Learning, NoC)
- RTL Verif (Base Design, Clause Learning, NoC)

Yet to Begin:

- RTL Finalization and Synthesis
- Floorplanning and IP Analysis
- Physical Layout

Comments: We are somewhat behind schedule, given our original timeline, as we have not completed our entire RTL design during the summer. That said we were able to complete our software modeling and locked our feature set, as well as completed the RTL for our base design. We had overestimated the time we had during the summer, given that many teammates were working full-time with many overtime is somecases. That said we are still confident that we will finish our feature set by the GDS deadline, as seen in our week by week schedule.

VI. Updated Week-by-Week Timeline

- 8/25: Base Design RTL
- 9/1: Base Design Verif, Clause Learning RTL
- 9/15: Clause Learning Verif, NoC RTL
- 9/22: NoC Verif, Full Integration
- 9/29: Full Verif, Synth, IP Selection
- 10/6: Floorplanning
- 10/13: FU Organization Planning
- 10/20: P&R Scripts for FUs
- 10/27: P&R Scripts for the Controller
- 11/3: P&R Scripts for I/O
- 11/17: Final Check

VII. Pre-Silicon Verification

A. *Toolchain*

For verification, we have completed an initial round of unit testing for the controller and functional units. Upon completion of a simple bus and arbiter, we will integrate the modules and form an integrated testbench to verify system functionality.

Our verification plan comprises two levels of checkers - a software checker to verify outcome correctness, and a Systemverilog checker to verify cycle latency of transactions.

B. Approach

Developing a verification testbench and software model will happen in tandem with RTL, with a different team member in charge of the two tasks to ensure the same ”bug” is not implemented in both the RTL and checker. The strategy would be to first develop verification components for the leaf nodes in our architecture, i.e. the Clause Units, followed by the Clause Bank Interface (and Implication Arbiter), followed by the routing unit for our NOC. This is because the child node components would have to be fully functional before we can have a testbench to test the parent nodes.

C. Clause

The Clause design is relatively straightforward - a small number of variables, with input/output wires to receive and propagate implications, output signal to indicate it has become a ”unit” Clause, and general load/store functionality. It would also need to send and receive implications. Hence, we should be able to achieve 100 percent functional coverage through a randomized testbench and software model checker, without needing many targeted tests.

D. Boolean-Constraint-Propagation Controller

The controller contains an elaborate FSM, and hence would require targetted testing. Features of the controller to test would be the receiving of implications or conflicts, emitting a decision, as well as issuing rollback commands. While it may be impossible to get a complete coverage crossing all features, we should be able to get full coverage individually on the state/FSM, the external interface to the switch, and the implication levels encountered.

E. System Architecture

For the router, we need to use UVM’s virtual sequencer feature in order to coordinate activity at different switches, in order to exhaust important scenarios. Some scenarios that we would have to test in verification are

- simultaneous insertion of flits with similar destination addresses, to flood a virtual channel from a single source/stall some flits
- simultaneous insertion of flits with multiple addresses, to open as many virtual channels as possible from a single source
- forcing many virtual channels to send to the same destination, forcing the switch allocator to arbitrate
- injecting malformed packets to test error checking at Switch and BCP level

Besides basic field coverage on the interfaces, we would also want to monitor the usage of the mesh vs torus networks, as well as delay coverage on traffic through the switches to ensure congestion/starvation does not cause the design to fail.

Due to the complexity of the network design, we will require several ”stimulus modes” in our randomized test bench in order to hit some edge cases in our coverage.

VIII. Post-Silicon Validation

For Post-Silicon Validation, debug and tuning, we will be communicating with the chip via a JTAG interface. This interface will allow us to send various test messages to the system under test (SUT). Furthermore, this interface will allow us to read the different system registers that we have.

To build the JTAG interface, we will need to develop the main TAP controller that is responsible for receiving test data in (TDI) and sending test data out (TDO). The TAP controller will consist of the control logic of our custom JTAG FSM, and the SERDES required to process data coming in and data being sent out.

For DFT, we will implement the following methods to ensure the individual IP blocks are working correctly, and to ensure that the interactions between the IP blocks on the SUT is accurate:

1) Scan Chain: Allows for register values to be exposed and read correctly
2) NoC BIST: Enables us to detect faults in our routing networks

A. Scan Chain

A scan chain is used to efficiently transmit the values of system registers to an external debug tool. To enable the scan chain, we convert all flip flops and registers to scan FFs and scan registers. To do so, we need to add a 2:1 mux to control whether the input data is scan input or a data input. Furthermore, we will need to add a system wide scan enable signal that will be the select bit for the 2:1 mux. Additionally, we will need to switch the clock signal frequency so that no data is lost when transmitting through the TAP controller and PCB trace to the hardware debug tool. This is to take into account the signal propogation delay being longer on the PCB traces due to higher impedances

B. NoC BIST

The NoC is the most critical IP in our system as it is the main method of communication between the various BCP FUs and Control Unit. As we will have a number of routers on the network, implementing a BIST for all the routers will enable us to detect faults quicker on our chips and also give us a deeper insight on what directed tests we should run to further pinpoint any issues should they arise. At a high level, there are 2 kinds of faults we will be concerned with on the NoC.

1) Routing faults: Since most of the routing in our NoC is a broadcast signal, the biggest cause for concern here is that all inputs from any ports (NSEW) get broadcasted to all other ports correctly. To test this, we will develop a BIST with a pattern generator which will generate random data packets that will be sent as stimuli to the FIFO input queues. The test controller will be responsible for routing these stimuli to the different input ports. Lastly, a pattern analyser will gather the outputs of the output ports to ensure that all unique test packets have been received at all output ports

2) Network Interface faults: NI faults occur when data from the BCP FUs (resources) are not packetized correctly for transport on the NoC mesh. In our design, there are a number of data fields that need to be sent on the network such a the var_id, var_status, implication, conflict etc. We need to ensure that the correct data is being sent and received on the network as incorrect packetization will lead to a failure of the algorithm. To achieve this, a pattern generator + analyser will be tagged to either side of the resource port. This allows us to create a stimulus packet on the input side and ensure that it is correctly packetized at the output of the input queue. Additionally, we will create a stimulus network packet on the output side of the resource port. This packet will then be decoded on the output of the output queue and ensure it is correctly depacketized.

The 2 DFT mechanisms will enable us to run deeper tests by running the pattern generation over multiple routers and sending them around for a pre-determined number of cycles depending on the longest path latency. Using the scanchain, we will be able to read the final states of the router switches allowing us to run system level tests on the NoC

IX. Pin-Out

Power

- 2x VDD
- 2x VSS

Regular I/O

- 1x Is_SAT
- 2x var_status
- 10x var_id[9:0]

Debug I/O

- JTAG interface[1]
  - 1x TCK
  - 1x TMS
  - 1x TDI
  - 1x TDO
- 24x Reserved

X. Team Qualifications

1) Hrishi - First-Year MS Student
   - Relevant Coursework: ECE 411, ECE 425, ECE 483, ECE 342, ECE 408
   - Doing machine learning systems research advised by Professor Nam Sung Kim
   - Industry: Interning at Astera Labs in Design Verification, writing UVM testing for fabric switches and CXL memory controllers.
   - Designed HW-SW in-network compression protocol for LLM training, wrote RTL for vector database accelerator, and developed peer-to-peer SNIC to accelerator data transfer scheme for research
   - Role: Algorithm, Architecture, RTL, PD
2) Wesley - Second-Year MS Student
   - Relevant Coursework: ECE 511, CS 533, ECE 411, ECE 425, ECE 498 SJP, ECE 462
   - Doing networking systems research advised by Professor Nam Sung Kim
   - Industry: Embedded SWE Intern @ Motorola Solutions, SoC DV Intern @ Qualcomm, Post-Si Validation Intern @ Arm, System IP Engineer Intern @ Arm
   - Wrote and verified RTL for a FU inside a behavioral HDL model of the FHE Accelerator as part of the semester-long course project for ECE 498 SJP
   - Role: RTL, Verification, PD
3) Parithimaal - Senior in Computer Engineering
   - Relevant Coursework: ECE411, ECE498NSU, ECE342, CS426 (concurrent), ECE425 (audit)
   - GPU DV Intern @ Apple, writing UVM testbenches and software models in systemverilog for graphics caches
   - Have experience writing RTL design and UVM testbenches with both Cadence and Synopsys tools
   - ECE385 and ECE391 Course Assistant
   - Verification, RTL
4) Aaditya - Senior in Computer Engineering
   - Relevant Coursework: ECE 482, ECE 411, ECE464, ECE342, CS426 (concurrent)
   - SoC RTL Design Intern @ Rivos; Power Management Architecture Validation Intern @ AMD working on power management architecture for Zen
   - Developed JTAG scheme for power management controllers, found solution to silicon design bug, and developed ways to reduce sleep state latency
   - PCB design and interconnect debug experience through eco illini supermileage
   - Arch, RTL, PD, Bring-Up
5) Cher Rui - Senior in Computer Engineering
   - Relevant Coursework: ECE 411, ECE 425, ECE 391
   - Experience: ECE 411 and ECE 391 Course Assistant
   - Role: Architecture, RTL, PD
6) Kai - Junior in Computer Engineering
   - Relevant Coursework: ECE 411, ECE 425
   - Doing research on AI in nutrition, advised by Volodymyr Kindratenko
   - Designed a 3D rendering accelerator for ECE 385 final project
   - Role: Algorithm, RTL, PD

References

[1] S. Park, J.-W. Nam, and S. K. Gupta, "Hw-bcp: A custom hardware accelerator for sat suitable for single chip implementation for large benchmarks," in *Proceedings of the 26th Asia and South Pacific Design Automation Conference*, ser. ASPDAC '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 29–34. [Online]. Available: https://doi.org/10.1145/3394885.3431413

[1] The TDI and TDO widths may increase depending on the number of reserved pins not used for other debug

[2] S. Kumar, A. Jantsch, J.-P. Soininen, M. Forsell, M. Millberg, J. Oberg, K. Tiensyrja, and A. Hemani, “A network on chip architecture and design methodology,” in *Proceedings IEEE Computer Society Annual Symposium on VLSI. New Paradigms for VLSI Systems Design. ISVLSI 2002*, 2002, pp. 117–124.

[3] A. Galimberti, F. Testa, and A. Zeni, “Noc router,” in *Embedded Systems*. Politecnico di Milano, 2016.

[4] C. Zhu, A. C. Rucker, Y. Wang, and W. J. Dally, “Satin: Hardware for boolean satisfiability inference,” 2023. [Online]. Available: https://arxiv.org/abs/2303.02588