ParaDPLL FPGA SAT Solver

Gordon Lichtstein
Dept. of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA, USA
glicht@mit.edu

Spencer Pogorzelski
Dept. of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA, USA
spogo@mit.edu

***Abstract*—ParaDPLL is a hardware boolean satisfiability (SAT) solver implementing the DPLL algorithm on an FPGA with a Python interface. Combining a modular and readable FSM-based design with parameterizable SystemVerilog and key optimizations to address memory limitations of FPGA SAT solvers (xor-sum clause signature and transposed representations), ParaDPLL provides a scalable baseline for future hardware accelerated SAT solvers.**

***Index Terms*—Boolean satisfiability (SAT), DPLL algorithm, FPGA acceleration**

I. System Overview

Our system, ParaDPLL, implements a hardware boolean satisfiability solver based on the Davis-Putnam-Logemann-Loveland (DPLL) algorithm. SAT is incredibly important across CS as it was one of the earliest NP-Complete problems identified, and as such a wide variety of difficult and useful problems can be reduced to instances of SAT. Thus optimized SAT solvers are useful for a wide variety of problems, ranging from graph coloring and theorem proving to scheduling optimization and Sudoku.

State-of-the-art FPGA-based solvers achieve 2-3x improvements over top CPU solvers, but use HLS systems to speed up development workflows and enable complex optimizations. The primary goal of our system is to provide access to a modular, readable, and somewhat optimized hardware SAT accelerator via a Python interface, potentially serving as a jumping off point for potential future projects. While we don't expect to exceed state-of-the-art solver performance, hand-written solvers like ours could potentially achieve better performance that HLS-developed ones, if the same optimizations are implemented.

The primary challenges we face broadly fit into two categories: memory/data management and complex control flow.

A. Memory and Data Management

The bottleneck of almost all modern hardware SAT solvers is memory access. DPLL implementations track a large amount of state across numerous structures, each access taking multiple cycles. CPU solvers have access to advanced low-latency caches and much higher clock frequencies, so FPGA solvers must make smart memory access decisions to compete.

DPLL is an inherently sequential algorithm with complex control flow including conditionals, loops, and indirection. It's extremely difficult to effectively parallelize computation, so instead we targeted optimizations to speed up sequential logic and reduce memory access counts. To address control flow, we represent each module as a finite state machine (FSM) to simplify implementation, reasoning about the system, and debugging.

At the algorithmic level, choosing appropriate data structures can make a meaningful difference in the number of memory accesses, and thus performance. Instead of iterating over large blocks of memory like standard DPLL implementations do, we implement optimizations to read a single targeted address or store a small list of occurrences. At the beginning of the project we estimated total memory usage to be fairly low (540,000 bits at most for our initial constants) and we knew that memory allocation would be static, so we were able to use BRAM for all memory needs. Synthesis confirmed our initial memory estimations to be accurate.

The design and routing of the BRAM in hardware is also an important consideration. Due to the large amount of state, our implementation creates hundreds of memory access wires that need to be routed to multiple modules. Our system utilizes SystemVerilog's interface abstraction to bundle these wires, alongside a memory multiplexer module that allows separate modules to request and write to the shared state. A similar system is implemented for additional solver state that is stored in registers owned by a state multiplexer.

To transfer data to and from the FPGA, we utilize a custom protocol on top of UART, transferring information similarly to its representation in BRAM to simplify decoding and encoding. UART is a fairly slow serial protocol and increases solver overhead latency, but we were able to vastly increase baud rate to reduce these impacts.

II. High-level Description

Given an instance of a SAT problem in conjunctive normal form (CNF), our system encodes it into its transposed form (TF), sends it over UART, assembles the problem into memory, solves it, and returns over UART whether the problem is satisfiable or unsatisfiable (SAT or UNSAT), and a set of satisfying inputs if it is satisfiable.

On the software side, our Python program takes a CNF SAT instance, performs trivial cleaning (e.g. removing duplicate literals), and generates the TF of that instance, which it sends in bytes over UART. On the FPGA, a UART module chunks the UART stream into bytes to send to protocol_in, which

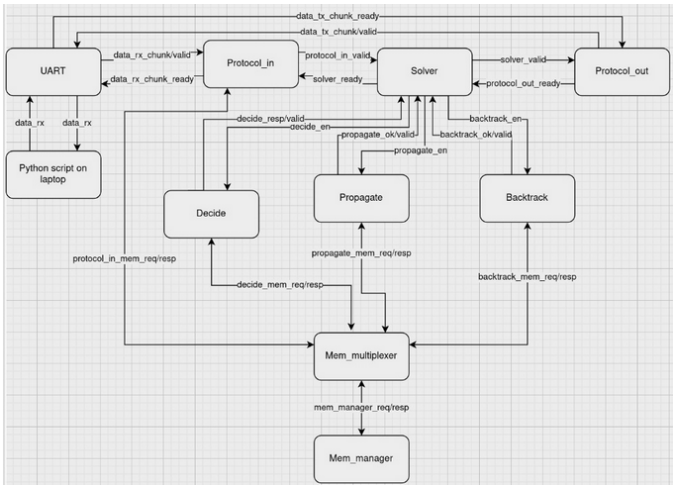

Fig. 1. Block Diagram

populates memory and solver states, and initiates the core solver FSM. The core solver orchestrates solving rounds by activating the decide module to assign a new variable, then the propagate module to propagate the assignment to every clause, and the backtrack module to undo the assignment if a contradiction has been found. This is repeated until all variables are correctly assigned and the instance is marked as SAT, or all possible variable assignments were eliminated and the instance is marked as UNSAT. When all rounds are completed, the core solver initiates protocol_out, which assembles the result into bytes which the UART module sends back to the Python script.

To visually demonstrate this system we implemented a Sudoku solving UI in Python. The user fills the Sudoku board with numbers, selects between the hardware and software solver for testing, then press ”solve”. The board is encoded, sent over UART, and the script decodes the result forming a solved board.

III. Memory Layout and Key Optimizations

A. Literal Representation

By default we store literals as 16 bits, with the most significant bit representing the sign (0 for a variable appearing positively, and 1 for appearing negatively), and the least significant 15 bits storing a variable ID. This allows for dense storage and the xor-clause signature optimization.

B. Memory Layout and SAT Instance Representation

The input format for a SAT problem to the Python program is a CNF file containing a list of clauses, each of which are lists of variables paired with a sign representing whether they appear positively or negatively. We optimized this form out of the FPGA implementation to save BRAM space, and only utilized the transposed form (TF).

The TF stores arrays of every clause index in which a given variable appears positively, and corresponding arrays of clauses in which it appears negatively. These arrays are slices out of a larger clause ID pool, storing the start position in the pool and the number of entries to save space. The main benefit of this form is that the common operation looking up every clause that contains a variable is a fast lookup rather than a slow scan, vastly speeding up propagation and backtracking.

C. XOR-sum Clause Signature

If all variables in a clause are assigned except one, and the corresponding literals all evaluate to false, then the remaining unassigned literal must be true, immediately obtaining the only possible variable value. To take advantage of this, each clause stores a count of unassigned variables and an XOR-sum of all the literal IDs. When we assign a literal, we decrement the unassigned count and xor the literal ID with the clause XOR-sum signature. If the unassigned count is 0 and the clause is unsatisfied, then the current variable assignment is UNSAT and we must unroll, which we can efficiently do by incrementing the unassigned count and xor-ing the literal ID for each variable assignment that must be undone. If the unassigned count is 1 and the clause is yet unsatisfied, the ID of the implied literal is contained in the clause's XOR-sum signature. One tradeoff with implementing this optimization is that we must implement a pre-filtering step in the Python script to ensure that we do not have duplicate literals in any clause, which is important because it maintains the invariant that each assignment can only invalidate at most one literal from any given clause.

IV. Key Module Descriptions

A. Memory Manager

We implement the DPLL algorithm using various fixed-size memory buffers referencing the transposed representation, calculated to store problems up to an upper bound size based on parameterized constants. In the C++ model, this appears as access into arrays, and the SystemVerilog memory manager contains a BRAM storage for each buffer, which accepts a single read/write request and returns a response some number of cycles later. Though the BRAMs have a fixed latency (2 cycles in HIGH PERFORMANCE mode), the memory interface abstracts this away with an AXI-like valid flag, allowing drop-in support for variable latency memory like DRAM. These BRAMs are instantiated with a xilinx_single_port_ram_read_first module, which we modified to accept a custom INIT value as a parameter, used to set default phase saved (last variable assignment) values to 1 instead of the default 0 if we choose. This leads to the solver initially guessing that a variable is True, which is a fair heuristic for many real-world SAT instances (although not for Sudoku), thus speeding up average solver times.

B. Memory and State Multiplexers

Each submodule can make memory requests as if it is directly connected to the memory manager. The memory multiplexer accepts requests from all submodules, forwards one at a time to the memory manager, and returns the response back to the requester. Since the solver only delegates control to

a single submodule at a time, only one request to each buffer will ever arrive at a time and thus a simple fixed priority system is sufficient to never drop requests. This module is implemented with a Python code generator due to the identical logic for each variable. We also generate a simplified "state multiplexer" that performs the same function but for smaller variables, stored in local registers. The state variables can be directly read from an input wire with no latency, and updates are provided by passing in a new value and making a new_valid wire single cycle high.

C. Input Protocol

The input protocol leverages preprocessing on the computer to transmit the problem description with low overhead and in the solver's memory format. The protocol is UART-based, encoding all numbers in big-endian format. It begins by sending two literal header bytes, followed by the number of variables and the number of clauses in two bytes each. Next, it streams into solver memory the four two-byte integer arrays each with one value per variable. These arrays encode the structure transposed representation in memory. The clause index pool is streamed in next. Rather than using its upper bound size, the exact length is known by the solver by adding up all of the previously received positive and negative subarray lengths. SAT instances are encoded and sent using this protocol via the utility Python script.

D. Core Solver

The solver module contains the primary FSM orchestrating the decide, propagate, and backtrack modules. Starting from the IDLE state, it transitions to the DECIDE state when the protocol_in module indicates that the CNF and TF have been loaded into memory. There it enables the decide module and waits for its response. If the decide module returns that all variables have been assigned and the instance is satisfied, then the FSM transitions to the SAT state. If the decide module could not find an unassigned variable and the instance is not satisfied, then the FSM enters the UNSAT state. If the decide module assigned a variable successfully, then the FSM initiates the propagate module and enters the PROPAGATE state. The PROPAGATE state waits for the propagate module to complete. If it identifies a conflict the FSM transitions to the CONFLICT state and initiates the backtrack module to unroll recent assignments, otherwise the FSM initiates the decide module again to select another variable and returns to the DECIDE state. When the FSM enters the CONFLICT state it waits for the backtrack module to return whether it has successfully re-assigned a variable to its negation, in which case the FSM returns to the PROPAGATE stage to propagate this assignment, otherwise the FSM enters the UNSAT state. In the SAT and UNSAT states, protocol_out is initiated, and the FSM returns to IDLE.

E. Enqueue

The enqueue module is a commonly used helper that assigns a variable and returns whether the assignment is consistent (e.g. whether we tried to reassign a literal to its negation). It's instantiated within other modules, using its parent's memory and state interfaces. The module accepts the variable ID to assign, and a value to assign (true or false). When the module receives a request, it reads the current assignment of that variable. If it is unassigned, the module writes to memory

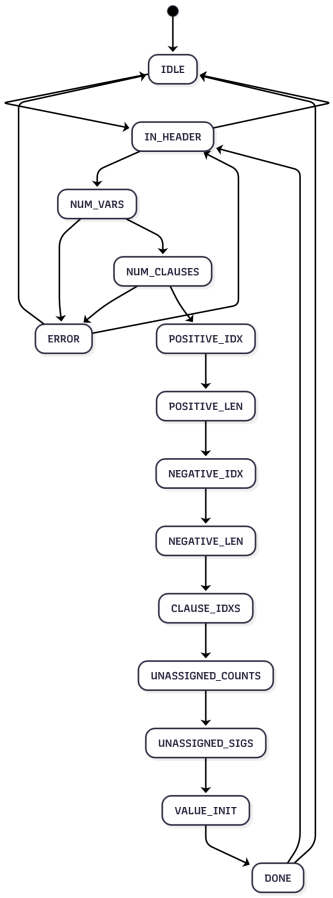

Fig. 2. Input Protocol FSM

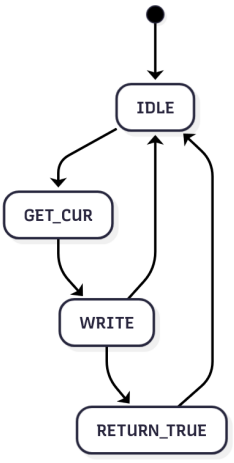

Fig. 3. Enqueue FSM

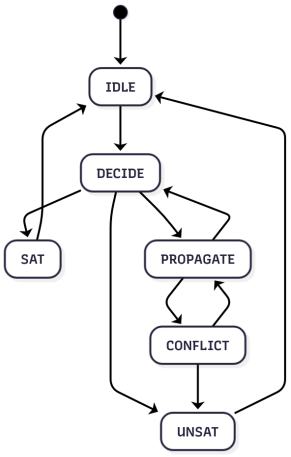

Fig. 4. Core Solver FSM

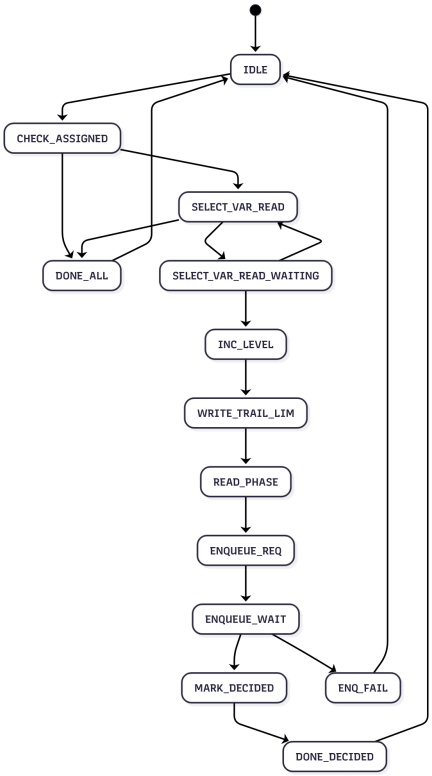

Fig. 5. Decide FSM

the value of the variable, the decision level at which this assignment was made (stored in the state interface), updates the assignment order record (trail), and returns to the parent module that the assignment was consistent. If the variable was already assigned, then the module checks whether the current assignment is the same as the previous assignment, and indicates to the parent module whether the assignment is consistent or inconsistent.

F. Decide

The decide module decides the next variable to assign, and assigns it with the enqueue module. Upon receiving a request from the core solver FSM, the module checks whether all variables have already been assigned, in which case it returns to the core solver that all variables have successfully been assigned and the instance is SAT. If not, it loops over all variables with two FSM states looking for the first unassigned one, and determines its value. When it finds one it initiates the enqueue module on this variable and new value. If enqueue is successful, decide returns the success to the core solver, which transitions to the propagate state. If enqueue found a contradiction, it returns that all variables have been assigned and the instance is unsatisfied.

G. Propagate

The propagate module propagates the most recent variable assignment to all clauses, returning to the core solver whether the assignment is consistent or not. From the trail we retrieve

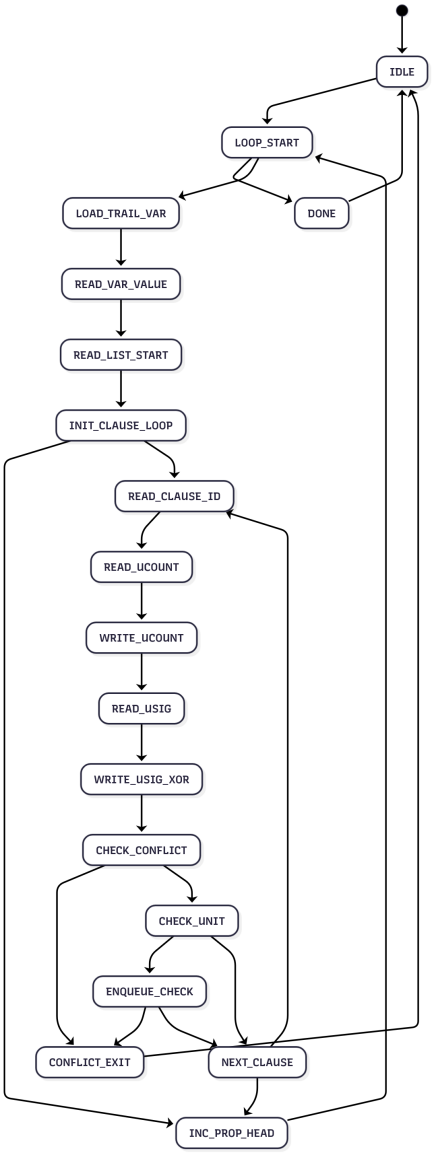

Fig. 6. Propagate FSM

the most recent assigned variable ID, and we request its value from memory. Depending on the variable's value (true of false), we loop over the transposed representation's list of positive or negative clause IDs, reducing the unassigned count, and updating the xor-sum. If the unassigned count is now zero, then we know there must have been a conflict so we return this to the core solver. If the unassigned count is 1 then we may immediately enqueue the implied literal and value, stored in the new xor-sum clause signature, returning to the core FSM the result of enqueue.

H. Backtrack

The backtrack module performs chronological backtracking, starting from the most recent conflicting assignment. It first checks if the conflict occurred at the root level, if so the formula must be UNSAT, which is reported to the core solver

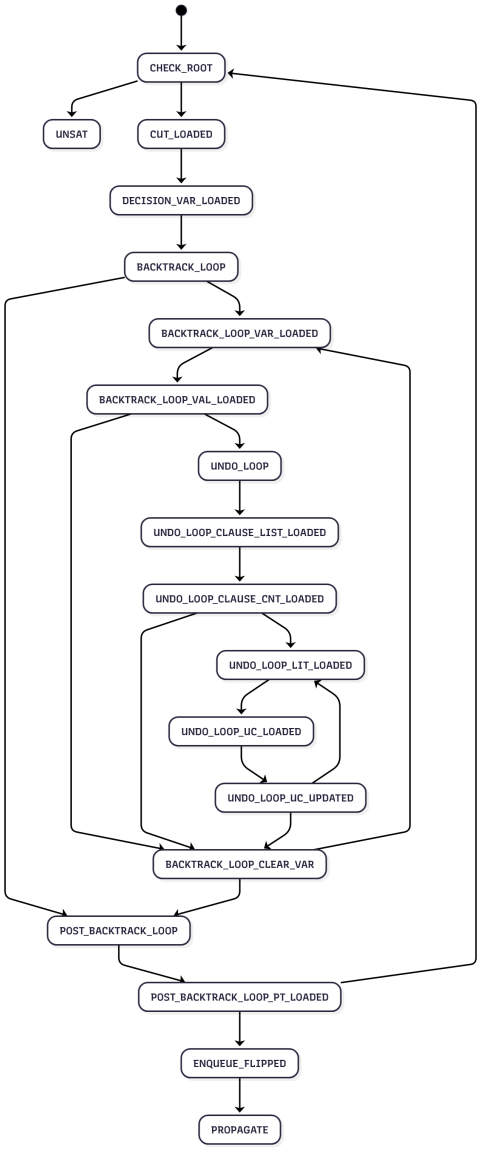

Fig. 7. Backtrack FSM

FSM. Otherwise, we undo the assignments from the trail until the previous decision level has been reached. Backtrack identifies the decision variable at this level. If both assignments for this variable (true and false) have been tried, then backtrack continues unwinding until the next decision level is reached, continuing this process until either the root is reached (in which case the instance is UNSAT), or we identify an assignment for which both phases have not been tried (e.g. a variable has been assigned true but not false). Backtrack then enqueues the literal and returns to the core solver FSM to transition into the propagate stage to propagate this assignment.

I. Output Protocol

The output protocol is initiated by the solver over UART to transmit the result to the computer. It sends a constant header, and a byte representing whether the problem has been satisfied.

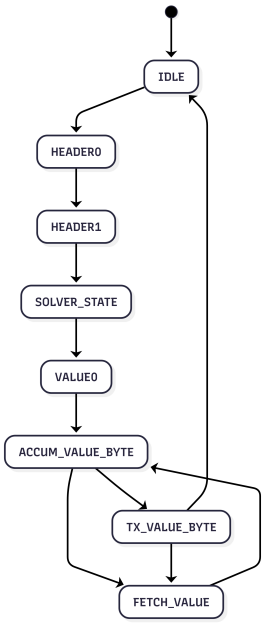

Fig. 8. Output Protocol FSM

TABLE I
BRAM RESOURCE UTILIZATION

Site Type	Used	Fixed	Prohibited	Available	Util%
Block RAM Tile	32.5	0	0	75	43.33
RAMB36/FIFO*	28	0	0	75	37.33
RAMB36E1 only	28				
RAMB18	9	0	0	150	6.00
RAMB18E1 only	9				

If it has been satisfied, it packs 8 variables to a byte, with one bit each to represent whether they are assigned TRUE or FALSE in the satisfying assignment. The Python program interprets this output to reconstruct the satisfying assignment.

V. EVALUATION

A. Commitments, Goals, and Stretch Goals

Our design met our primary commitments. We developed a working single-core FPGA SAT solver solving our current uf20 and uf50 testbenches, including the following functional modules, testbenches individually or together using test_full.py: mem_manager, mem_interface, mem_multiplexer, uart_transmit, uart_receive, protocol_in, protocol_out, solver_core, decide, propagate, backtrack, and enqueue. Our design seems to have a hardware-only bug where it does not properly re-initialize state or memory after one solve. Thus, the design must be reset between each solve to ensure accuracy.

We met one of our two further goals. Our design is parameterizable and modular allowing for easy scaling and minimizing resource utilization. However, our design does not currently include multiple solver cores, as this addition proved too lengthy to incorporate given the project timeline.

TABLE II
LOGIC RESOURCE UTILIZATION

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice LUTs	1266	0	0	32600	3.88
LUT as Logic	1266	0	0	32600	3.88
LUT as Memory	0	0	0	9600	0.00
Slice Registers	1078	0	0	65200	1.65
Register as Flip Flop	1078	0	0	65200	1.65
Register as Latch	0	0	0	65200	0.00
F7 Muxes	0	0	0	16300	0.00
F8 Muxes	0	0	0	8150	0.00

Our design met our stretch goal to solve Sudoku problems on the FPGA. We preprocess a user-specified Sudoku grid from a Python Tkinter UI, generate a SAT instance, send it over UART, retrieve the response, and display it on screen. We also optimized the UART protocol, identifying the highest supported baud rate without introducing errors (which we found to be 921600 baud). As we didn't implement multiple solver cores, we could not meet our stretch goal of splitting large SAT instances into multiple smaller disjoint instances to run in parallel. However this would be an interesting future addition to our project, as the space of joint hardware-software SAT solvers and intelligently splitting SAT instances to optimize for hardware constraints and strengths seems to be relatively unexplored.

B. Performance Evaluation Overview

Our other evaluation criteria include measuring latency, throughput, and memory usage. For our current FSM-based modules, latency is directly related to the maximum number of sequential states an FSM goes through. This varies from a minimum of 3-4 states in enqueue up to potentially hundreds of transitions (depending on solver constants) for propagate which can loop over every clause and contain nesteds loops. Current throughput is simply 1/latency, but multiple solver cores could alter this relationship. Our FSM-based design also incurs occasional minor 1-cycle overheads like padding on memory/state transactions, but this tradeoff is acceptable given that FSMs make the system much easier to implement, testbench, and debug.

C. Latency

The most critical metric for our SAT solver is end-to-end solving latency. We measure the time taken to from transmitting the SAT instance over UART to receiving a response from UART, omitting Python preprocessing time, as our primary goal is measuring the hardware's latency. In most cases the latency is fairly low, almost always taking around 0.04-0.06 for 50 variable instances, and most Sudoku instances taking a little over 1 second, fairly comparable to our software implementation and an unnoticable difference in the Sudoku UI. It appears that UART transmission makes up the vast majority of the latency for these cases, as solving takes only around 30,000 cycles for many 50 variable instances. However, for some Sudoku instances the solver takes much longer (around 10 minutes, while the C++ model takes 1 minute), but still returns a correct solution. It's unclear exactly why this is occurring, as these Sudoku instances are too large to simulate with cocotb without crashing. We currently believe that this is because the solver is fairly unoptimized compared to modern software solvers and cannot cache memory, leading to many incorrect assignment guesses and expensive backtracks.

D. BRAM Memory Usage

Initial memory calculations including CNF, TF, and state information indicated a maximum usage of about 51 18K BRAM blocks for our initial constants (512 vars and 4096 clauses), and after CNF removal optimizations, predicted memory usage fell to 30 18K blocks. By inspecting Vivado logs, we observed that the final implementation for these constants occupied 10 RAMB36 blocks and 9 RAMB18 blocks, equivalent to a total utilization of 29 18K BRAM blocks, almost exactly matching our post-CNF optimization estimation. To support larger SAT instances like those generated from Sudoku boards, we increased the maximum number of variables to 1024 and the maximum number of clauses to 16384, which about doubled memory usage to 28 RAMB36 blocks and 8 RAMB18 blocks, accounting for about 40 percent of memory. This still leaves sufficient space for a potential second solver core.

E. LUT Logic Usage

Another key evaluation metric is LUT usage, representing the proportion of available logic that our design uses, and the scalability of our design. Our design uses a total of around 1200 LUTs out of a total of 32600, representing only about a 4 percent usage. This was to be expected, as our design includes little complex combinational logic and mostly relies on register-heavy sequential logic with relatively simple transition conditions that can easily be condensed into just a few LUTs. This confirms that the primary scaling limitation is memory usage, and that there remains potential for our design to implement complex combinational decision heuristics.

F. Timing

The standard DPLL algorithm utilizes little complex combinational logic that could lead to our design failing to meet timing, and FSMs further buffer timing critical-operations like memory requests. Thus the WNS (worst negative slack) for our design is calculated to be 2.788 nanoseconds.

VI. IMPLEMENTATION INSIGHTS

Throughout the project we made heavy use of SystemVerilog's interface abstraction, which essentially bundles numerous wires together, reducing the number of lines of code in each module dedicated to memory/state management. Without this, each module would contain about hundreds of additional inputs and outputs to drive memory, and the memory manager and multiplexer modules would become more complex. Interfaces also support modports, so a memory request could use mem_interface.mem_req to access a variable index into an array as output, while the memory multiplexer and manager modules can access the same logic as an input, through

mem_interface.mem_resp. However, interfaces are not well supported by iVerilog, and cocotb does not allow testbenches to drive interfaces directly. Thus to testbench modules we had to use local Vivado and write module wrappers that forward interface logics to inputs and outputs cocotb could drive, which was tedious.

Relatedly, there are also hundreds of lines of almost repeated code in the memory and state and multiplexer modules. Interfaces do not meaningfully affect this problem, and writing the repeated code by hand is error-prone. To mitigate this we wrote a series of Python scripts to automatically generate these files, which proved more reliable and easier to write than SystemVerilog macros.

Initially developing a C++ model and software testbench suite greatly sped up hardware implementation. It allowed us to test functionality, optimizations, and easily estimate memory usage before we had functional HDL implementations. After developing the C++ model and rigorously testing it with a framework that accepts SAT instances in a standard file format, we converted each module to a finite state machine representation to more accurately represent the hardware implementation. While these FSMs didn't exactly match the final hardware implementation, they did allow us to test the FSM versions of the C++ modules as we wrote them, also providing a baseline that made it easier to write SystemVerilog FSMs.

The C++ model was an invaluable resource when debugging SystemVerilog implementations, as we could print out when memory or state is modified or read from, and compare it to waveforms. We also developed a tool that tracked key memory and state writes in the C++ model, and matches it up with the same writes generated from a debugging top level module. This allowed us to quickly identify exactly where the first mismatch between C++ and SystemVerilog behavior was, and massively sped up debugging. Additionally, complex modules like decide, propagate, and backtrack are quite difficult to accurately testbench on their own, as their inputs and outputs are very simple and they primarily modify state based on previous state. Thus to testbench accurately, we would have to manually set up state and memory, run the module, then inspect state and memory to ensure it is equivalent to the golden model (an approach we did take to testbench enqueue, protocol_in, and protocol_out). Our test_full.py file and C++ memory and state inspector replace individual testbenches by automatically setting up solver state and memory, thus testing modules in context and on real-world .cnf files. This allows for more accurate and automated testing, reduces testbench errors, and enables a wider variety of tests on complex solver state.

Software loops are easily to visually understand, but they become much more difficult to understand and debug when they are flattened into FSMs. To facilitate debugging and FSM visualization, we developed a small domain specific language to describe FSMs. Given a description of the FSM states and the conditions for moving between states (using an embedded expression language) written in a Python library, the tool generates Mermaid flowcharts like those used in this report. The tool can also generate skeleton SystemVerilog, with the state enum, reset behavior, state transitions, and idle and ready signals, although this functionality wasn't used in this project.

VII. Contributions

We jointly researched prior FPGA SAT solver implementations, decided on the scope and optimizations to include in our project, and created a basic C++ software model. Spencer integrated the transposed representation optimization and expanded our initial testbench framework to include SAT instances loaded from .cnf files downloaded from the SATLIB benchmark. Then we converted this software model to a more hardware-realistic version by making each key module an FSM, with Gordon focusing on the core solver, decide, and propagate modules, and Spencer on backtrack and enqueue.

In the initial hardware translation phase, we each focused on the modules we had made into an FSM as we were more familiar with them. In addition, Spencer designed and implemented the input and output protocols, wrote Python scripts to automatically generate state and memory multiplexers, and first jointly testbenched modules and memory. Gordon wrote the initial memory interface, management, and multiplexer modules.

Final integration and debugging was done together, with Spencer developing tooling to automatically compare simulated memory and state writes to the C++ golden model, and Gordon debugging Vivado synthesis errors and testing the end-to-end system. This allowed us to testbench multiple modules and their complex interactions simultaneously, without having to manually initialize valid memory and states for each test. Spencer wrote initial utility files to convert SAT instances into bytes for the input protocol, and Gordon integrated these changes with previous UART sending/receiving modules and the Sudoku UI.

References

[1] "Boolean satisfiability problem," Wikipedia, The Free Encyclopedia. [Online]. Available: https://en.wikipedia.org/wiki/Boolean_satisfiability_problem

[2] "Conjunctive normal form," Wikipedia, The Free Encyclopedia. [Online]. Available: https://en.wikipedia.org/wiki/Conjunctive_normal_form

[3] M. Lo, M.-C. F. Chang, and J. Cong, "SAT-Accel: A modern SAT solver on a FPGA," in *Proc. FPGA '25: ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, 2025. doi: 10.1145/3706628.3708869. [Online]. Available: https://dl.acm.org/doi/10.1145/3706628.3708869

[4] "FPGA-BASED SAT SOLVER," IEEE Xplore. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4054862

[5] IEEE Standards Association, "IEEE Standard for SystemVerilog—Unified hardware design, specification, and verification language," *IEEE Std 1800-2017*, 2018. [Online]. Available: https://rfsoc.mit.edu/6S965/_static/F24/documentation/1800-2017.pdf

[6] H. H. Hoos and T. Stützle, "SATLIB Benchmark Suite." [Online]. Available: https://www.cs.ubc.ca/~hoos/SATLIB/benchm.html