Hardware Accelerated Chess Engine

Wai Cheong Tsoi
Department of Electrical and Computer Engineering
University of California, Davis
Davis, California, United States
wctsoi@ucdavis.edu

Kenneth Dano
Department of Electrical and Computer Engineering
University of California, Davis
Davis, California, United States
kddano@ucdavis.edu

Shane Blim
Department of Electrical and Computer Engineering
University of California, Davis
Davis, California, United States
scblim@ucdavis.edu

***Abstract*—This paper covers the overarching goal of hardware acceleration, previous hardware chess engines, and details the FPGA hardware-accelerated design of our chess engine implemented on a Terasic DE1-SoC FPGA board. The hardware design used to speed up pseudo-legal move generation is based on Deep Blue's move generator, integrated into a modified version of Tom's Simple Chess Program (TSCP) hosted on Linux. In addition, success metrics, results, and future work are discussed. For a depth of 1, the pseudo-legal move generation hardware performed slower to TSCP's pure software approach due to data transfer overhead, but much larger improvements are expected as hardware clock frequency is increased and more depths are generated in hardware.**

Keywords—Chess Engine, Hardware Acceleration, Field Programmable Gate Array, System on Chip, Move Generator, FIFO, DE1-SoC, Avalon Bus

I. Introduction

Hardware-accelerated machine learning has been one of the key research areas for the past decade. For example, Microsoft's Project Catapult is focused on FPGA acceleration for cloud computing purposes such as AI and deep learning. In contrast to Microsoft's enterprise-level application, Apple's 2018 A11 Bionic chip is a consumer-level SoC which uses an FPGA for its neural engine. We are now more commonly seeing hardware-accelerated machine learning being implemented in enterprise solutions and consumer products.

With the continuously decreasing development cost of custom hardware and the great potential speedup, hardware acceleration is becoming the norm of hardware development. Combining the benefits of improved performance and reduced power consumption of hardware acceleration with the complexity of a chess engine, this project demonstrates the potential of a future paradigm where custom hardware replaces general-purpose hardware, not only in the field of automation or artificial intelligence, but in every consumer device, especially with the trend of Internet of Things.

Custom hardware of such not only bring benefits in the technical field, by having higher throughput and lower power and latency, but also by using FPGA, be more environmentally sustainable as hardware in FPGA can be updated without physically replacing it. It also reduces the manufacturing cost, when hardware obsolescence does not imply retiring older chips, saving cost in manufacturing the chips and the compatible boards, reducing physical waste while also allowing future devices to have a longer product lifetime. This potentially changes the culture of the society, where tech gadgets that we own will not get replaced every two years for a mere 10% boost in productivity, but getting a hardware and firmware update on our existing solution instead, ultimately reducing the global impact of its life cycle, reducing its disposal and potentially, the environmental impacts from raw material extraction and processing.

Therefore, to embrace the benefits of this paradigm shift, we would like to design a hardware-accelerated machine learning hardware of reduced scale as a case study, in which we chose a chess engine. A chess engine is chosen since it includes the generation of a large search tree that its size increases exponentially, the search and evaluation through the large search tree, and the decision that one move is the best move based on the current board state, thus resembling the basic structure of machine learning.

The design and implementation of a chess engine incorporate knowledge through multiple disciplines. By incorporating the knowledge of computer architecture and embedded systems, we have designed a communication protocol between the software and hardware layer. The hardware design of our move generator uses Verilog to describe the behavior of the hardware to the synthesizer. We have constructed test benches using Verilog to verify the behavior of our hardware modules. We used Linux to host our software written in C to complete the chess engine.

We used the DE1-Soc Development Kit to develop our chess engine. The DE1-SoC is an industry-level development kit built around the Altera System-on-Chip (SoC) FPGA. FPGAs are popular due to their flexibility and thus lower development times, compared to full custom designs. The DE1-SoC provided realistic hardware constraints that professionals experience when prototyping new designs with limited economic resources; hardware limitations directly influenced the design of our system.

II. Problem definition

A. *Pseudo-legal Move Generator*

We define "board state" as the current state of the board i.e. what piece is in each square of the current board. A move generator would scan the board state and generate all possible moves given the current board state. But, according to the rules of chess, a move that results in a check is not considered legal, so if the legality of the moves is not verified, this type of move generator is considered generating pseudo-legal moves.

While legality checking takes a sizeable amount of hardware to achieve, this process prunes out moves that are unfavorable for us, which helps the search algorithm to run more optimally. In contrast, without checking legality in hardware, the move generator that generates multiple depths would suffer significantly in time and space exponentially since moves that are illegal will be searched and generated with its own branches. It is thus very favorable to have hardware legality check implemented to save time at the expense of logic blocks and memory blocks.

B. Hardware Used

A DE1-SoC FPGA board is used for this project. The DE1-SoC contains a Cyclone V FPGA with a dual-core ARM processor. The board is decently capable of supporting our chess engine and a Linux operating system to interface software with our hardware. The DE1-SoC contains 85 thousand logic elements, 4450 Kb of embedded memory, and 64 MB of SDRAM, which is enough for our purposes. There is 1GB of DDR3 SDRAM used by the Linux operating system that is not accessible by hardware.

The scarce amount of logic and memory blocks confined our design to be optimized against the constraint of time and space. If the module is not optimized against time, the performance will decline exponentially as the depth generated is too shallow to compete with others. If the module is not optimized against space, it will not fit into our FPGA. It is crucial to strike the balance between having the least processing time to maximize its performance while designing a module that fits our FPGA.

C. Interfacing with Software

Linux is installed onto the FPGA board to host our software, which checks the legality of moves generated by hardware, evaluates the moves, and searches the tree to determine the best move of a given board state. By using the Qsys system integration tool, our hardware module can be compiled into the Cyclone V and used by our software.

Avalon bus is the interface that is used to communicate between the hardware and the Linux operating system. While the move generator generates the moves in hardware and stored in SDRAM, the Avalon bus transfers the generated move list to our software hosted by Linux for further processing.

Timing the communication between hardware and software presents several challenges. The two platforms need to be able to correctly signal that they are ready for the other to perform a task. In addition, they also need to be able to react to the other signals and respond in a timely fashion so that the system is wasting as little time idle as possible.

III. Related Work

A. Belle

Built in 1977 by Ken Thompson and Joe Condon at Bell labs, Belle was the first chess engine to use hardware to improve performance. Belle handles move generation and evaluation in hardware, but the alpha-beta search is done in software. All of Belle's hardware is contained in a small cabinet.

Belle's chessboard is represented in hardware by an 8x8 grid of squares. The squares are composed of digital logic that records the piece currently occupying the square and transmits and receives moving piece data to and from neighboring squares. There are two levels of control hardware that manage the squares. One level manages 4x4 groups of squares and then the second level manages those groups. Belle's hardware makes moves by first identifying possible victim pieces, and then identifying possible aggressor pieces. The hardware uses a 64-level stack of 64 mask bits to hide squares that have already been checked for the current process. The system prioritizes moves by the most valuable victim and least valuable aggressor.

B. Deep Blue

In 1997 the chess machine Deep Blue beat the world chess champion Gary Kasparov to become the first chess engine to beat a world champion in a match. Deep Blue was designed by Feng-Hsiung Hsu at IBM. Deep Blue's chess hardware is a full-custom VLSI design using 0.6-micron CMOS technology. Full-custom designs take more time and money to design up-front compared to FPGA's but can give you better performance and are cheaper to mass-produce. Deep Blue's hardware is housed in a large, upright cabinet.

Like Belle, Deep Blue's hardware has an 8x8 grid of squares, but the first level of control hardware manages rows of squares instead of 4x4 groups, and then the next level of control manages the row hardware. Deep Blue also has hardware for evaluation of positions and alpha-beta searching. Deep Blue's bit masking is different from Belle's in that instead of saving different levels of mask bits, it calculates masks using the most recently generated move.

IV. Design Methodology

The pseudo-legal move generator hardware consists of 64 square units, 8 column units, a move generator module, and a control module. As this move generator only produces pseudo-legal moves, legality checking hardware is not included. Without the legality check in hardware, the logic blocks utilization was found to be 40%, implying that it is possible for such a module to be implemented in hardware later.

A. Square Unit

64 square units were constructed with respect to the 64 positions on a chessboard. Each square unit contains the information of the piece occupying that square, and each square communicates with the neighboring squares to generate pseudo-legal moves. As the new board state is passed to the squares, the squares calculate which direction to propagate to, and send the information to the neighboring squares, as if the piece is being moved to that square as the next move. It will also send a hold signal to all squares in that line of sight. As a square receives propagation information from neighbors, it calculates whether that move can be made and put into a FIFO to be collected later. Once all possible moves are generated, a done signal is flagged as true to signal the column unit.

A square not only communicates with its eight immediate neighbors, passing pieces as if they were physically moved, but also connects to its eight possible knight movement blocks. Since the knight moves would not propagate after received by its neighbors, after receiving, knight pieces were held until the square finished generating all other moves. This allows propagation of other moves instead of wasting one cycle to consider knight moves before its propagation as well as creating timing problems. Since one cycle is required after holding the read-enable signal to its FIFO, it could be possible that the done signal be flagged one cycle earlier, when the knight move is being evaluated by the square, that still doesn't affect performance nor timing.

As squares can be operated in parallel to other squares, the move generation process takes at most nine cycles to complete. If some squares are done before others, the moves can be transferred to the column unit first asynchronous to other squares. For squares that are empty, the empty signal would be high and signaling the column to ignore this square, and hence saving two cycles per squares ignored.

B. *Encoding*

The positions and the piece information are being passed to neighboring squares in the format of {3 bits of column, 3 bits of row, 3 bits of type}. The 3 bits of column and row denote the origin position of the piece. The remaining 3 bits denote the type of the piece being propagated. One bit of the color of the piece is omitted since only pieces on our side can be propagated.

One move is encoded in the format of {7 bits of flags, 6 bits of origin position, 6 bits of destination position}. The position bits use the same format of {3 bits of column, 3 bits of row}. The 7 bits of flags include a valid bit, and 6 bits of other flags used by the software, including signifying a capture, a castle, et cetera. The valid bit is used since some invalid moves are used as fillers to speed up the process.

Eight moves are passed into the FIFO each time since each square receives propagation information from 8 neighboring squares in one cycle. To speed up and reduce the number of cycles used by the square to write moves, if there are less than 8 moves to be written in that cycle, an invalid move will take its place. Those moves will not be written into the final layer before the software to reduce the delay by the Avalon Bus as the bus is relatively slow.

Castling and *en passant* moves are evaluated and generated in the move generator module to avoid overcomplication of logic in square units, while also utilizing the downtime of waiting for moves from columns to generate these moves. This saves two cycles per board state as well as a lot of redundant logic blocks to be used in the 64 square units.

C. *Column Unit*

A column unit acts as the middleman between the move generator module and the squares, it takes care of breaking down the board state received from the move generator module to send to each square and collects moves from the square to be sent to the move generator module. A column is chosen instead of a row to ensure a relative balance on the number of moves collected from the columns, and thus avoid overloading any FIFO while minimizing the cost of memory blocks in our DE1-SoC board.

Columns, just like the squares, operate in parallel to each other, takes at most 80 cycles to transfer the moves from the FIFOs inside the square to its own FIFO, excluding the overhead from the squares. Further pipelining efforts should consider this module since this module not only consists of wiring between columns and squares, but also complicated logics to gather moves from squares to its own FIFOs. The illegal moves generated by the squares are not purged at this stage to reduce the amount of propagation delay within this module at the expense of using a larger FIFO.

D. *Move Generator*

The move generator module takes the board state and other flags from the control module and pass it to the columns. While waiting for the squares and columns to finish, the move generator module also writes any possible *en passant* moves and castling moves into its FIFO. It acts as the final FIFO to collect moves from all columns and pass it to the control module to be read by the software. Castling and *en passant* moves are generated in parallel to the generation of moves of the squares, utilizing the wasted cycles. This not only saves two cycles but also extra resources to send the flags from software to the specific squares, extra logic that is redundant for most squares, and extra cycles to produce said moves.

The move generator module takes at most 528 cycles to transfer the moves from the FIFOs from the columns to its own FIFO, excluding the overhead from the columns. While having 64 squares directly accessed by the move generator module would reduce the number of cycles needed for retrieving the moves, it also significantly increases the cost for development and verification, whereas its benefits would show as depth increases. In our case, it is a more feasible option to have columns act as a middleman to both interact with neighboring columns and transfer moves from squares to the move generator module.

E. *Memory Allocation*

Each move takes 19 bits and 8 moves are stored in the FIFO in one cycle. The data width of the FIFO must be greater than 152 bits, a 160-bit width was selected for the FIFO because it was the closest width possible when creating the FIFO via the Quartus' IP Catalog. The extra 8 bits are currently unused but could be later used to provide its origin node data for multiple-depth generation and tracking or future debugging purposes.

Each level (square, column, move generator module) has a different number of depths of memory to allow future multiple-depth solution expansion by minimizing the number of memory bits used in lower levels. Theoretically, a square unit would only generate a depth of 8, a column unit of 64, and the move generator module of 514. For our current design, we provided a depth of 16 for a FIFO in a square unit, a depth of 128 for a FIFO in a column unit, and a depth of 1024 for a FIFO in the move generator module. This allowed more memory to be allocated in the control module, and hence potentially more depth of the move tree to be generated.

F. *Control Module*

The control module communicates with software and the move generation hardware via the Avalon bus. The software passes a board state, flags required for special moves like castling and *en passant*, and a start signal via the control module to the lower level modules. When control receives a done signal from move generation, it reads data from the FIFO and checks it for valid moves which the control module then writes to its own block RAM for software to read. When the control module has read all the moves from the FIFO, it writes the number of moves to be read to a special address in the RAM. When all the moves from the FIFO have been read, control signals the software that it is done.

A 256 Kib RAM is allocated to the control module, allowing a move tree of maximum depth of 3 and a branching factor of 150 to be stored. This implies that the move generator might not be able to store all the possible moves of depth 3 on average due to memory constraints. This greatly reduces the capability of the move generator, but could be solved either by having a larger FPGA of larger memory, or review the architecture of the Avalon Memory Mapped devices and consider whether it's possible to have part of the moves transferred to another block memory while more moves are generated.

G. *Hardware / Software Interface*

Hardware and software are connected using the Avalon Memory Mapped (Avalon-MM) interface. Hardware and software read from and write to a set of registers at specific addresses in memory to communicate. In this configuration, the hard processor system (software) acts as the master device and the hardware (our Verilog component) acts as the slave device. The software uses the interface to schedule read and write operations. To begin the process of move generation, the software writes board state information to a set of addresses in RAM designated as software to hardware data, and then writes a start signal to the control address. When move generation is finished, hardware writes a done signal to the control address.

Table 1 describes the memory addresses of our slave Verilog component i.e. the pseudo-legal move generator. The Avalon bus data width is 32-bits, and so is all the data at each slave address. The slave address goes up to 8191 (13-bits) in order to fully address the 256 Kib RAM.

For future expansion of multiple-depth move generation, it is possible to have an agreed handshake to specify the range of memory that the current depth has reached while continuously generating the next depth. This will allow fast and slow evaluation modes mimicking that of in Deep Blue.

Table 1: Pseudo-Legal Move Generator Address Descriptions

Slave Address	Description
0	Control bits: {unused[28:0], reset, done, start}
2-9	Current board state/position
1, 10-15	Unused
16	Number of moves generated
17-8191	Move stack generated by hardware

H. *Software*

We downloaded a lightweight Linux Ubuntu image onto the microSD card used on the DE1-SoC board. C code is used for the software engine and is used to access hardware memory addresses. We used Tom Kerrigan's Simple Chess Program (TSCP) as the base for our engine. We replaced the C code pseudo-legal move generator with our hardware pseudo-legal move generator that is written in Verilog.

Figure 1 is an overview of how our chess engine works. In software, it first takes in a 256-bit representation of the current chessboard state and writes it to regs in hardware. Then the software signals the hardware to generate pseudo-legal moves and waits until all moves are generated for the current board state. Hardware signals software when it is done, and the software reads all the moves. The think & search step first determines move legality and then evaluates the move based on the resulting board state. Evaluating a board means to assign a numerical rating to the board state. The value of the rating shows how advantageous/disadvantageous the board state is for the light (white) pieces. To find the optimal move, our engine needs to generate a tree of legal moves and apply the min-max algorithm with alpha-beta pruning. To grow the tree, the software will request moves from hardware whenever it needs to.

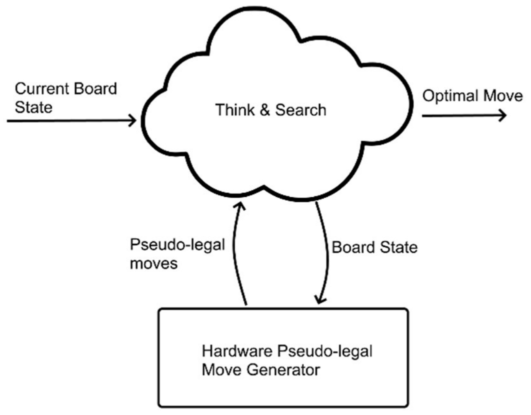

Figure 1: Our Chess Engine's Move Generation Process

The pseudo-legal move generator only generates moves for light pieces. Therefore, if moves need to be generated for dark pieces, we translate the dark pieces into light pieces when writing the board state to hardware. When hardware returns the stack of pseudo-legal moves, we translate the moves to match with the dark piece locations.

V. VERIFICATION METHODOLOGY

Given the complexity of our design, our team created test benches for simulation and dummy hardware to functionally verify each component, as well as scripts to generate tests for some components.

A. *Square Unit*

To verify the functionality of the square unit individually we wrote testbench code in Quartus and then simulated it in Modelsim. The testbench code sets the piece to be tested occupying the square as well as the inputs coming in from neigboring squares. Then it toggles the reset signal on and off, and waits five cycles for the square to write any applicaple moves to the FIFO. After that it sets the read enable signal high and waits one clock cycle so that the output of the FIFO can be analyzed, as well as the directional outputs of the square.

B. *Column Unit*

The column unit mainly sends the board state into the 8 squares it includes and retrieves the moves from it, and then promptly streams the move out from its FIFO when requested. Its wiring also includes hold signals between the squares and to other columns, further complicating the tests required to verify the functionality of the module.

The test bench hosts the column units to mainly test the timing of the signals since the module mainly transfers signals and does not include lots of complicated circuitry unlike the square module. Therefore, the test bench mostly focuses on the accuracy of the wiring and the timing of the data transfer. Each hold signals are set to test the connection into each square inside the column, and modified squares that triggers each bit of each hold signals that connects out from the column unit to the test bench. Modified squares are also used to send out a move from each square to the column to verify the integrity of the write and its timing.

C. *Move Generator*

The move generator module in principle has a similar behavior as a column module, where it also routes signals with minimal logic between columns, retrieves the moves from the columns and stream it to the control module when requested. Thus, the testing procedure is similar to of the column unit, except due to the complexity of the column modules, no dummy column modules are created for the purpose of verification. Instead, the move generator is verified after the verification of the column unit. By utilizing the ability to descend to the child modules in ModelSim, it is simpler to verify the design of the move generator module while also saving time.

A Python script is created to generate the bit-string of a certain board state such that the test bench could read the file and send the signal to the move generator module without changing and recompiling the design for every test. The script is also useful since the formatting of the board state is not intuitive and it is easier to convert a board state to its corresponding bit-string using a script than physically typing all 64 pieces of the board, including empty pieces, per board.

All moves generated are printed in the console along with the total number of moves as a signature. The signature serves as the first check since if there is a mismatch in the total number of moves generated, it is likely that either the logic of the square unit or the timing of the column or the move generator module is skewed. But, if the signature matches, the individual moves need to be verified that it matches with every possible pseudo-legal move that can be generated for the given board state.

D. *Control*

The control module is the top-level module of our pseudo-legal move generator and has the Avalon bus interface. Our testbench for control is very similar to how software uses control, and thus the pseudo-legal move generator. The main difference from our control testbench and software writing/reading to control is that our control testbench needs to simulate the Avalon bus signals.

To test control, the testbench writes a board state to addresses 2-9, writes the start bit high at address 0, and then cycles the clock 600 times to let the move generation finish. The generated waveforms are inspected for accuracy. A dummy move generator module is used for the verification of the control module because of its complexity. Meanwhile, the verification of the control module only consists of the correctness of the moves extracted and written into the RAM, the state machine that handles this process, and the timing of the process. Thus, the dummy move generator module would always check the integrity of the input signals, initialize by writing pre-defined moves into its own FIFO, set a done signal, and wait for the control module to extract the pre-defined moves. The test bench will then read the RAM to ensure all pre-defined moves are written into the RAM without the illegal moves that are also included in the test.

VI. EVALUATION METHODOLOGY

To evaluate the performance of our hardware pseudo-legal move generator, we created a test in C code that loads a pre-defined board state and records the time it takes to generate the moves and write them into memory. This is repeated 1000 times and then the average is calculated. This test function also measures the time it takes to transfer data from hardware to software, and vice versa, in order to determine the overhead.

To evaluate the performance of our engine as a whole, we created a test in C code that loads a pre-defined board state and records the time it takes for the engine to produce the final move decision. By measuring the execution time of the software only engine and our engine, we can calculate the speedup. This is repeated 1000 times and then the average and standard deviation are calculated. Any speedup would be a success in terms of hardware acceleration.

We only replaced the pseudo-legal move generator of the engine; therefore, our engine does not search any deeper or have any smarter search algorithm. Our engine should reach the same solution as the software only engine, except faster, so we can compare the final move decision between the two engines to test for accuracy.

The next metric would be how well our engine can play chess. We know that given enough time, our engine and the software only engine will both produce the same move. Does that mean our engine will never perform better than the software only engine? No, if there is a sufficient time constraint to make a move, then our engine should be able to reach search deeper and thus produce a better move on average. There are two options to measure how well our engine can play chess.

1. Play hundreds of bullet chess matches online and see how our engine is ranked.
2. Play hundreds of bullet chess matches against the software only engine and calculate its win/loss ratio.

Bullet chess matches are chess matches where each person has less than three minutes of turn time for the entire game. For our purposes, we would likely need to have one-minute games with zero time added each turn (increment). For option 1, it would take too much time to determine performance because we would have to play enough games to accurately rank both our engine and the software only engine. Option 1 also has network overhead that would slow down the number of games played per hour. Option 2 is our best bet for determining whether our engine can perform better than the software only engine, albeit under a tight time constraint. If our engine can achieve more wins than losses against the software only engine, that would be a success.

Another way to evaluate our engine is to use TSCP's benchmark function. This function loads in the board position from move 17 of Bobby Fischer vs. J. Sherwin, New Jersey State Open Championship, 9/2/1957. The function then searches fiveply three times and calculates the search rate of nodes per second from the best time.

VII. Results and Discussion

A. Design Accomplishments and Shortcomings

A pseudo-legal move generator of one layer of depth was created. Due to time constraints, we did not implement move legality checking in hardware nor a hardware search function. One layer of depth is generated due to time constraints, but it can be changed to search for a variable depth with extra hardware in the move generator module.

Memory constraints also impeded our design flow and forced us to optimize our modules before adding more functionality in hardware. The main optimization we did was create different size FIFO modules for the square unit, column unit, and the move generator module.

B. Hardware Resource Usage

Table 2 summarizes our designs total resource usage. Our current implementation consumed 19% of the total 4450 Kb embedded memory of the FPGA. But for a depth of 4 and a branching factor of 20, it will exceed the 4450 Kb embedded memory of the FPGA, where a different solution should be proposed should we have a depth of greater than 3, which is highly feasible in speedup but also costly in design time.

39% of the logic blocks are utilized for our pseudo-legal move generator. It is expected that the legality checking hardware will be a modified version of our pseudo-legal move generator that have every piece attack our king after the move. Note that since the action is similar to generating another depth, another possibility of such could be directly generating the next depth with two copies of the same move generator and add logic in the control module to remove moves that are determined to be illegal. But this approach will result in an extra FIFO in the control module to store all moves required to be checked before it is finalized and stored in the RAM, utilizing more of the already scarce memory resource.

Another method for checking legality would be utilizing the evaluation function to give a score of negative infinity when it results in the king in check. This integrates the legality checking into the evaluation function during the search, which could simplify the datapath and process in the expense of a larger branching factor, longer time required for the search, and possibly a lower maximum frequency due to the complicated logic present in the evaluation function.

Table 2: Resource Usage Summary

Logic Utilization (ALMs)	12,281
Combinational ALUT Usage for Logic	19,563
Dedicated Logic Registers	9,191
Total Block Memory Bits	789,598
Total PLLs/DLLs	2/1
Maximum Fan-Out	18,440
Total Fan-Out	255,588
Average Fan-Out	6.19

C. Hardware Clock Frequency

The maximum clock frequency is only 129.5 MHz according to Quartus' compilation report. It is obvious that there is a lot of long logic sequences in our component. For example, the reset signal that is sent to the legal move generator is then sent to fanout to 8 columns, which is then also sent to 8 of its square units, and then sent to the FIFO of the square unit, also taking a significant amount of time to reset the FIFO unit. Another example would be when a piece is transferred to the neighboring square, the signal is passed through a lot of gates and cascades until it reaches the data input of the FIFO, which hurts the latency between flip-flops. Because of the complexity of the circuit, there exists a lot of long wires and huge logic blocks with lots of gates that cascade, significantly hurting the possible maximum clock frequency.

Our team has already added registers between most long path wires to decrease the latency of a long wire like the reset signal, but while it is possible to further decrease the latency within pipeline stages by adding flip-flops within a logic, it would also take a significant amount of time to verify the design, time we did not have. Besides, a higher clock frequency does not imply a shorter overall latency as the circuit now takes more cycles to finish.

D. Board Positions

We used chess board positions from [3] to test our engine, Figure 2. [3] also has the number of nodes at each depth for each board position, which proved helpful for testing the accuracy of our hardware pseudo-legal move generator.

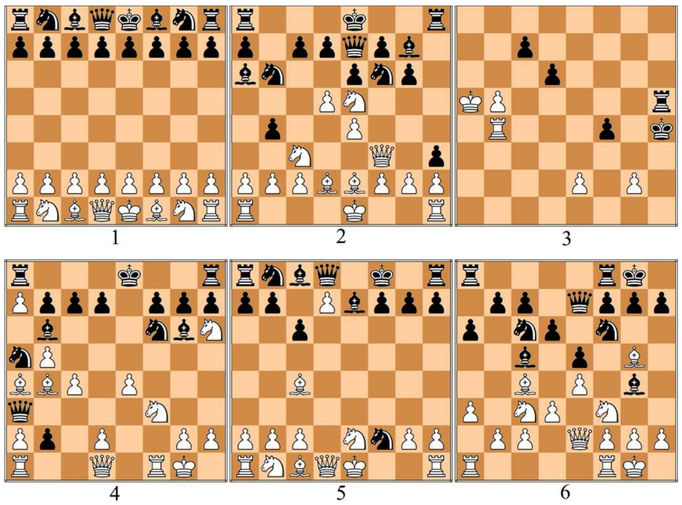

Figure 2: Board Positions Used for Testing

E. Evaluation Results

We managed to get results for all evaluation methods described in Section V except for the engine vs. engine bullet chess results. We encountered difficulties when running the software only engine and our engine at the same time. We disabled pondering so that the engines only "think" during its turn, but it still does not work.

Tables 3 and 4 show the results of our test function that measured the time it takes to generate pseudo-legal moves for each of the 6 board positions in Figure 2. Note that Table 4 breaks up the total time it takes to generate the moves into subcomponents: the time spent moving data from software to hardware, the time spent moving data from hardware to software, and the time spent waiting for hardware to finish generating moves. The data transfers from software to hardware, and vice versa, are the overhead time that our engine has that the software engine does not. From the data, our generator is slower even when removing this overhead, although, not by a huge margin slower. We missed out on easy optimizations that would improve our current design, optimizations such as running the hardware at a higher frequency or using burst reads on the Avalon bus. Bigger design changes such as returning multiple move depths would have a massive impact on average move generation times.

Table 3: Software Pseudo-Legal Move Generator Measurements

Board Position	Total Avg Time (ms)
1	0.004099
2	0.005146
3	0.002430
4	0.004769
5	0.004726
6	0.004925

Table 4: Hardware Pseudo-Legal Move Generator Measurements

Board Position	Total Avg Time (ms)	SW to HW Avg (ms)	HW to SW Avg (ms)	HW Time (ms)
1	0.026423	0.003263	0.015818	0.007342
2	0.051687	0.003185	0.035240	0.013262
3	0.021393	0.003177	0.012373	0.005843
4	0.043841	0.003203	0.029285	0.011353
5	0.045578	0.003213	0.030667	0.011698
6	0.051690	0.003103	0.035212	0.013375

Tables 5 and 6 show the results of our testing function that measured the time it took for each engine to generate the optimal move (move decision) for each of the 6 board positions in Figure 2. We expected our engine to make the same move as the software only version, and it did. We expected our engine to be faster, but it is not. As said previously, our current design has a lot of overhead, which is likely the reason for these large measurements.

Table 7 shows the results of TSCP's benchmark function when applied to the software-only engine and our hardware integrated engine. The results agree with our other tests. Our engine has a node per second search rate 19.48% slower than the software only engine.

Table 5: Software Only Move Decision Measurements

Board Position	Average (ms)	Standard Deviation (ms)	Move Made
1	32.088420	0.112060	e2e4
2	478.249954	19.771978	d5e6
3	3.553162	0.068635	h1h4
4	617.345712	0.853014	c4c5
5	112.105319	0.393748	d7c8
6	255.071943	0.540552	c3d5

Table 6: Hardware Only Move Decision Measurements

Board Position	Average (ms)	Standard Deviation (ms)	Move Made
1	65.330337	0.139295	e2e4
2	708.241818	1.422432	d5e6
3	4.738120	0.033988	h1h4
4	829.155684	1.301215	c4c5
5	171.825185	0.226932	d7c8
6	329.327749	0.589217	c3d5

Table 7: TSCP Benchmark Results

	Nodes	Best time (ms)	Nodes per second
Software Only	550778	4096	134467
Hardware Only	510398	4714	108272

VIII. FUTURE WORK

The chess hardware operates independently of software; therefore, the hardware modules can be further optimized to minimize the latency between flip-flops, such that they could operate at a higher clock frequency than the software. Adding a separate higher frequency clock for hardware would offer significant speedup with minimal change of the design. But, it also needs to be considered that the more pipeline stages could yield diminishing returns as the more pipeline stages are added, the more cycles is required for data to propagate. Besides, hardware verification of timing cost a lot of time, which is a serious tradeoff to be considered.

Depending on the optimization progress, the move generator can be modified to generate more depths in order to minimize a lot of the delay caused by the Avalon Bus by having to perform fewer transactions. However, the maximum number of depths that are searchable in hardware may be limited by the memory of the FPGA board. Upgrading to a larger FPGA could allow the hardware to search even more depths and further speed up the system. However, a larger FPGA also comes with an initial cost when purchasing the board, as well as usually a higher power requirement, which hurts the sustainability of the end product developed.

Additional hardware could be added to do evaluation of board states for the search in hardware instead of software. This would require significantly more hardware but would offer a significant speedup. Utilizing parallelism, the evaluation process could be done faster in hardware than in software, and later in the process hardware would have to pass fewer moves to the software, because the large list of pseudo-legal moves will have already been slimmed down. It is estimated that if the legality check module is of similar size as the move generation circuit that the overall design could fit into our current FPGA board. However, including the multiple-depth capability also requires more logic blocks and memory blocks for the state machines and logics, as well as the move list and undo list.

Checking move legality in hardware instead of software can speed up the move generating process. It can either be implemented with an entire module for checking, or by searching one more depth and pruning the nodes if found to be leading to a check by the opponent. This is also limited by the number of logic blocks present in the FPGA board.

A search algorithm can be implemented in hardware, and with the help of alpha-beta pruning and other techniques, search for the best move in hardware. This completely reduces the load of software and potentially provide a huge speedup. However, the search hardware cost a lot of logic and memory blocks, implies that it is unlikely to be implemented in our current FPGA board.

REFERENCES

[1] Feng-Hsiung Hsu, "Chess hardware in deep blue," in *Computing in Science & Engineering*, vol. 8, no. 1, pp. 50-60, Jan.-Feb. 2006, doi: 10.1109/MCSE.2006.2.

[2] M. Boule and Z. Zilic, "An FPGA based move generator for the game of chess," *Proceedings of the IEEE 2002 Custom Integrated Circuits Conference (Cat. No.02CH37285)*, Orlando, FL, USA, 2002, pp. 71-74, doi: 10.1109/CICC.2002.1012769.

[3] “Perft Results - Chessprogramming wiki,” *ChessProgrammingWiki.* Accessed: June 9, 2020. [Online]. Available: https://www.chessprogramming.org/Perft_Results.

[4] "Bell chess-playing computer | Mastering the Game | Computer History Museum," *ComputerHistory.* Accessed: June 3, 2020. [Online]. Available: https://www.computerhistory.org/chess/stl-43305190f1a84.

[5] “The day a computer beat a chess world champion, 1997 - Rare Historical Photos,” *RareHistoricalPhotos.* Accessed: June 3, 2020. [Online]. Available: https://rarehistoricalphotos.com/kasparov-deep-blue-1997/.