River Chess Engine Final Report

1st Arjun Barrett
Department of EECS
Massachusetts Institute of Technology
Cambridge, MA, USA
arjunjb@mit.edu

1st Dylan Isaac
Department of EECS
Massachusetts Institute of Technology
Cambridge, MA, USA
disaac@mit.edu

Abstract—**We introduce River, a chess engine designed to run on an FPGA, utilizing hardware-level parallelism for a high-throughput move search implementation based upon an iteratvely deepened, alpha-beta-pruned implementation of negamax optimization with quiescence search. Our project led to an engine that is capable of playing FIDE-style chess at an ∼2200 Elo level. The successful implementation of River demonstrates that hardware-based chess computation is a viable (and significantly more efficient) alternative to software-based approaches.**

The source repository for our project, which include all C and SystemVerilog code for our software model and hardware design, is available on [redacted]

I. Motivation

Traditionally, chess engines have been implemented in software, mainly utilizing multithreaded CPU execution for move search and evaluation. While these software implementations are often easy to implement, they can be computationally expensive and power-hungry. This is especially true when multiple CPU operations are needed to emulate a single fundamental chess board operation, such as for sliding piece move generation.

FPGAs provide the unique ability to design optimized gate structures for particular bit manipulations that may not be possible to perform efficiently in software. Moreover, FPGAs expose massive parallelism to the programmer in contrast to the relatively small amount of instruction-level parallelism and multicore parallelism possible on most CPUs. As computer chess is a task that involves both obscure bit manipulation routines and can be highly parallelized, it is a prime candidate for implementation in hardware rather than software. If designed correctly, a hardware implementation of a chess engine has the potential to be significantly more power & time-efficient than its software counterpart. We introduce River, a highly pipelined, hardware-based, streaming chess engine for FPGAs that was designed based on these principles.

II. System Overview

We introduced the initial design for River in our earlier preliminary report. Our system architecture has not fundamentally changed since then; the full architectural schematic is shown in Figure 1. However, after working through concrete implementations of several of the features we planned to implement, we now have the final design capable of running on an Artix-7 FPGA. Below we discuss our design and some of the issues we came across.

III. Software model (Arjun)

We have implemented a functional, complete software model for the project in C. RiverSW was written carefully to mimic the approach we would ultimately use to design River in hardware. It creates no dynamic allocations, uses no large-scale memories, and implements all logic in terms of bitwise operations and small lookup tables.

RiverSW is technically a chess engine in its own right, but it is extremely slow (and therefore weak in Elo). However, this is largely an artifact of the Verilog-esque coding style it employs, and performance increases by several orders of magnitude once the logic is translated into hardware. For instance, various complex, multi-cycle operations in RiverSW (e.g. Hyperbola Quintessence move generation) have been translated into pure combinational logic in River. Additionally, costly branching logic present within RiverSW are removed entirely in River.

The implementation of RiverSW proved to be a larger time investment than we initially anticipated; however, it paid major dividends in ensuring the correctness of our hardware modules. By nature, chess programming involves handling an abnormally large number of edge cases; a software model gave us the ability to find, target, and fix edge-case bugs much more quickly. RiverSW helped us quickly discover specific input positions causing bugs in our hardware implementation and debug those specific edge cases in simulation. This proved to be crucial because running an exhaustive search for bugs in simulation would have been nearly impossible; we found many bugs that only occurred a few times in hundreds of millions of nodes in our game search tree, and having an exact software replica helped us narrow down the search for badly behaving positions very quickly. The hardware-friendly coding style of RiverSW also transformed much of the module writing process into a translation exercise from C to Verilog, which dramatically accelerated our development speed.

IV. Move executor (Dylan)

The move executor is used by both the UCI module and engine coordinator; as such, it has strict requirements in terms of throughput and latency. To meet these requirements, we opted for a larger, highly parallelized design that can execute a move onto a board within a single cycle. Additionally, great care has been taken to implement true FIDE chess rules, including edge cases such as en-passant pawn captures, castling, and piece underpromotions. It has been largely based

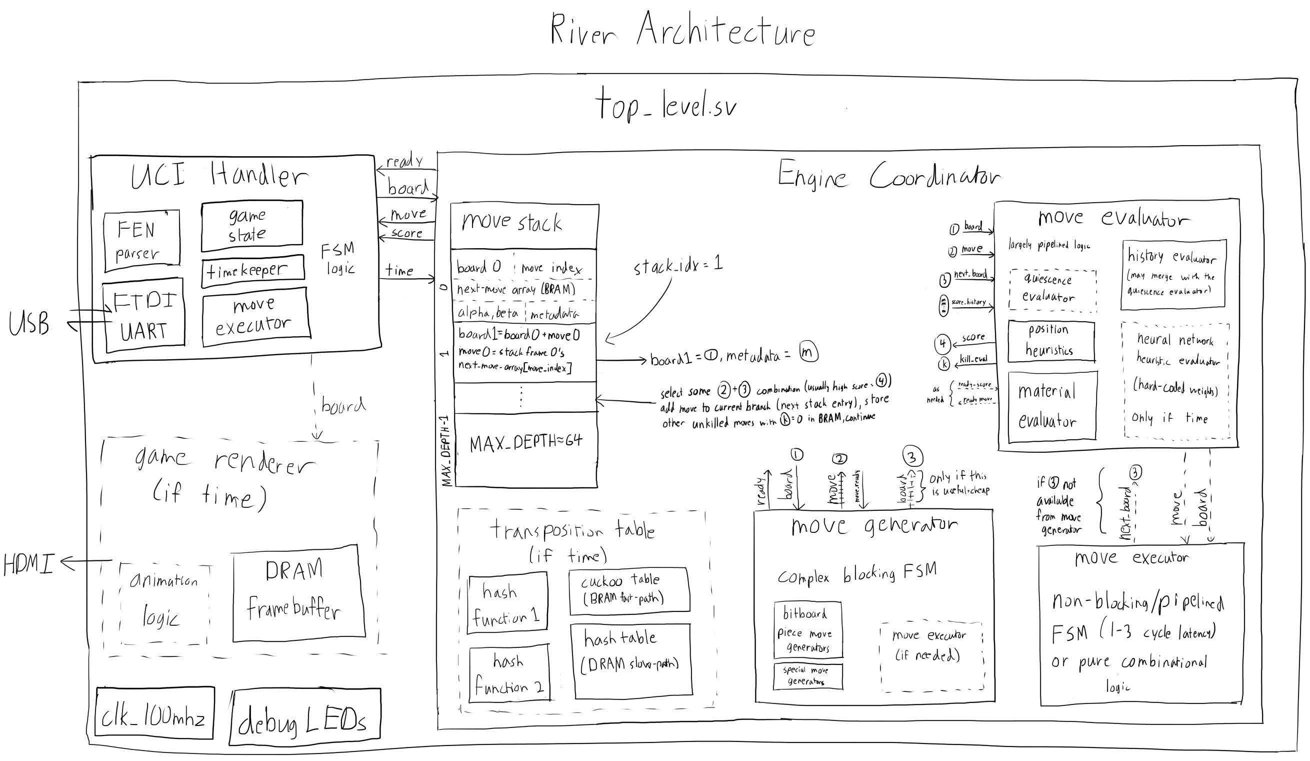


Fig. 1. Initial system-level block diagram. The primary changes between this diagram and our final design were (1) pipelining optimizations in the engine coordinator, (2) a removal of score history from the move evaluator in favor of quiescence search, and (3) a captures-only input flag for our move generator to accelerate quiescence search.

on RiverSW's move executor, but is modified to eliminate branching and conditional execution through extensive use of 64-bit masks.

We opted to represent the board with a one-hot encoding for most of the pieces, as doing so allowed us to accept a small increase in board size for much lower latency. This representation allowed for most of the engine logic to be done in parallel using bitwise operations in the move executor and generator.

V. Move generator (Arjun)

Move generation is a key step in any chess engine and is often tricky to implement correctly. Although we initially anticipated that the verification of this module would take a large portion of our total time, we found that in practice the move generation was easy to get right with a verifiable reference model (i.e. RiverSW).

We implemented move generation using a blocking finite state machine that takes in a position and generates all legal moves from that position, one at a time. This may initially seem like a poor choice in an otherwise highly pipelined system, but because each position generates upwards of 30-50 moves, this architecture is not a bottleneck.

The specific implementation involves combining the results of individual move generators for each piece, using count-trailing-zeros hardware to generate move structures from bitboards using combinational logic. The module updates individual piece substructure states as needed depending upon the piece move generator it uses on any given cycle to generate all of the legal moves from a position.

We originally designed the move generator as we had originally intended (one move per cycle of throughput, with a blocking FSM on the input position). However, this proved to create a long critical path in our design. In order to relax our timing constraints, we opted to change this into a 2-stage pipeline with a throughput close to, but not quite equal to, 1 move per cycle.

One optimization in the hardware implementation that wasn't present in our software model was a "captures-only" mode for the quiescence search, which helps dramatically reduce the latency of evaluating positions deep in the search tree. On average, this improved the performance of our overall system in its search to a fixed depth by a factor of 2.

The move generator makes use of multiple-bit tricks to extract what moves are possible for each piece. Due to the complexity of the module (600 lines of dense SystemVerilog code), RiverSW proved to be an excellent investment for the design of the move generator.

VI. Move evaluator (Arjun)

We have decided not to opt for an advanced neural move evaluator and instead have focused on a simple combination of position and material heuristics that are very easy to implement and compute. Similar to the move generator, the move evaluator benefited greatly from the development of RiverSW. We may add some extra complexity to the move evaluator as follow-up work for the project; however, we have found while playing against the hardware model that the evaluator works quite well as is.

The move evaluator also detects illegal moves and checkmate (which we can treat identically to a "very bad" evaluation), as this can reduce the size of our search space dramatically without too much effort. It is also a necessary addition to avoid worst-case behavior of our engine generating illegal moves (though this cannot happen with a search depth of more than 2 in general, as both of these situations have a highly unfavorable terminal node evaluation that will be skipped).

Another improvement we utilized based on the output of the move evaluator such that we have "best-first" move ordering. This allows our engine to generate beta cutoffs for the alpha-beta search more quickly, which can reduce the search space by multiple orders of magnitude. In RiverSW, we opted to use quick-sort over a very approximate move evaluator. When translating this into hardware, we opted for a simple single-cycle streaming parallel insertion sorter over true static evaluation results; this yields both a better move ordering and a faster output. Since we receive moves over time, we could amortize the work of sorting across the move generation process, allowing us to access a fully sorted list of moves only a few cycles after we finish move generation.

VII. Engine coordinator (Arjun)

The engine coordinator takes the role of the top-level from lab. It handles communication between the UCI module , execute, generation, evaluator, move-stack, and sorters. Additionally, it holds logic that doesn't fit neatly into any of the previously mentioned modules. The engine coordinator also contains an FSM.
The FSM contains states:

- READY
- NEXT
- GENERATING
- WRITEBACK
- FINISH

We encode the current state of the engine in the position stack, which is a large component of the engine coordinator that stores alphas, betas, scores, and generated move information for each node in the path of the current depth-first search. Most of this information is simply stored in registers, and we impose a limit of 64 on the maximum depth of the DFS to store this in fixed-size hardware. However, we store the move lists in BRAM because they can grow to be quite large (there could theoretiaclly be up to about 220 legal moves in each of the 64 nodes of the moves stack, and each move is 15 bits in our representation). We also limit the number of moves we consider at each depth to the 64 moves yielding the best static evaluations; this is much more than the maximum number of legal moves in any position in a chess game (typically 40-50), so it's a very good approximation that saves a lot of BRAM in most cases.

The FSM in the coordinator is the heart of the module. The engine starts in the READY state and accepts "go" commands from the UCI module to begin a move search on a new board. At any given search depth (including the start at depth 0), the engine starts in the NEXT state, verifies it isn't a terminal node, and then enters the GENERATING state. It waits in the GENERATING state as the move generator sequentially generates every move in a board. As moves are produced, they are executed by the move executor, and the output positions are evaluated to generate a move key. This key is used to sort the moves using an insertion sort algorithm. After sorting completes, the engine enters the WRITEBACK state, and the moves are stored into a subsection of a 60 kilobit BRAM based that corresponds to the current depth in the move stack. We recursively iterate on the next move until we reach a maximum target depth; at this point, we extend the search by searching only for loud moves (moves that capture pieces or move out of check; this is the quiescence search). After evaluating all moves in a position, we enter the FINISH state and return to the next shallower depth; providing the evaluation of the current position to the parent; this is either the static evaluation of the position for a terminal quiescent node, or otherwise the minimax evaluation for the subtree. The parent ingests the evaluation to update its own minimax state (alpha, beta, and score), inducing a beta cutoff if needed, and continues to the next node in its subtree (or returns to the parent if all its children have been evaluated).

Once we reach the root node in the FINISH state, we have executed minimax for the entire tree for some target depth; we choose to iteratively deepen (increment the target depth) or return the best move found depending upon move time constraints.

VIII. Interfacing with UCI (Dylan)

UCI is a standard command-line protocol that allows chess engines to communicate with game boards and other chess programs via a standardized text interface. Although it is primarily designed for software-based chess engines, it allows compatibility with hardware-based engines like River without modifications. We implement support for the UCI text protocol directly within our FPGA to make our engine more self-contained, rather than depending upon a complex software adapter.

Since sequential character-by-character command-line inputs are not automatically compatible with FPGAs' design philosophy of extreme parallelization, we've decided to contain the UCI protocol to a singular FSM, which provides parallel inputs and outputs to the rest of our design.

To avoid making our UCI implementation too complex, we only implement the `debug`, `position`, `go`, `uci`, `move`, `bestmove`, and `info` commands. However, the current subset is sufficient to interface neatly with a wide variety of chess gameboards including Lichess.

Since the UCI module was creating at a relatively early point in the project, it became extremely beneficial for test-benching the rest of the modules. By using UCI to load arbitrary boards, it was a useful tool in sim to test move executor and generator.

A. Physical Communication

In order to support communication with a diverse set of systems, we decided to first have an in-software program collect UCI commands from an arbitrary source. We then have this program forward commands to and from our FPGA over USB.

On the hardware side, to support USB communication, we use our FPGA's FTDI USB chip. This chip is used to translate between UART and USB communication, simplifying the design of the module.

B. Architecture

The UCI module consists of:

- Read FSM
  - READY
  - DEBUG
  - POSITION_BOARD_TYPE
  - POSITION_NEXT
  - POSITION_MOVES
  - TRASH
- Write FSM
  - READY_OUT
  - INFO
  - BEST_MOVE
- `charbuff` - character buffer for managing in-coming commands
- `best_move_buffer` - character buffer for managing out-going best_move commands
- `info_in_buffer` - character buffer for managing out-going info commands
- Internal board representation
- Move executer
- I/O ports for reading and writing individual characters to the UART/USB interface
- I/O ports for communicating with the rest of the design using the AXI protocol.

The bulk of the complexity of the UCI module comes from the two FSMs. The two FSMs decide how to use the listed assets based on their current state.

Each state within the read FSM represents a relevant position within the process of reading a particular command.

Conversely, each state within the write FSM represents a relevant position within the process of sending a particular command.

IX. Evaluation

There are two primary perspectives from which we can evaluate the success of our experiment with implementing a chess engine in hardware: (1) the strength of the engine, and (2) the improvement in efficiency versus a software implementation.

Unfortunately, it is difficult to exactly determine River's Elo due to differences in measurement techniques between different sources, time controls, and other factors. However, from our own play against the engine, and from approximate performance statistics such as average centipawn loss per move, we estimate its Elo to be between 2000 and 2200 (i.e. master level). This is not as strong as state-of-the-art software engines can achieve; this is not because River is poorly written, but rather that River uses no transposition table or NNUE due to their complexity and difficulty to pipeline. It nonetheless competes with significantly more complex software engines due to its massively better throughput, even on much less advanced hardware.

We managed to reach almost all of our ideal goals, which was exciting considering the complexity of the project. The only ideal goal we didn't hit was the parallelized DFS stack. After running into a slice-count bottleneck, we learned that effectively doubling most of our design wouldn't be feasible. However, this doubling wasn't required to reach our ideal Elo. Additionally, we reached the stretch-goal of playing against the FPGA online through Lichess. Overall, the project managed to achieve what we set out for.

Most of our modules have a throughput and latency that we are happy with. The UCI, move executor, sorter, and engine coordinator all work on a 100MHz clock. Additionally, for these modules, we can accept an input every cycle at 100MHz.

However, we needed to drop our clock frequency down to 50MHz, although in theory we could have increased it to 60MHz. Our move evaluator module in one of the steps requires adding 64 integers, as such this became our main source of propagation delay. Although we already made the evaluator a two-stage pipeline, due to time, we weren't able to make it deep enough to increase our frequency back to 100MHz.

It's a common theme that the move evaluator was the most difficult module of our project. Halfway through development, we ended up running out of slices on our FPGA, overshooting by only around 900. After extensive testing and re-building, we found that the main culprit was the evaluator. Since the evaluator was integral to the performance of our project, we opted to transition from using the Spartan 7 to the Artix 7. Our final build on the Artix 7 utilizes 77% of the boards slices, 50% of the board LUTs, and 60 Kbits of BRAM. Additionally, if we were to further pursue the project by adding a transposition table, we would also need to use the board's DRAM.

Another issue we came across appeared to be a PLL instability issue. Although our engine worked 100% of the time in sim, it only worked 90% of the time in hardware. Since it seemed to be almost random when it happened, and it only occurred when sending quick successive commands to the UCI, we assumed it was a bug in the code of uci_handler. However, after debugging the system using the info command supported by the uci module, we learned that random bits in our board representation were being flipped. Due to the nature and number of bits being flipped, it wasn't possible for it to be a logical error in our code. As such, we decided

it was an issue with clocking. To solve this, in the interest of time, we added a delay in our forwarding script to prevent commands from being sent too quickly.
A python script we used to print out the corrupt board representations is shown in Fig 2.

X. Future Work

Due to time constraints, there were some features we chose not to implement, or chose not to include in the final build.

One such feature was full support for parsing go commands in uci. In reality, a go command can be formatted as ”go winc 100 wtime 200 btime 300 binc 400.” Although we had a full hardware implementation of uci capable of parsing this command, due to the PLL issue, we ran out of time to test the command in hardware (although it worked in sim). We’d like to push the full parser for go commands into our code post-submission.

Additionally, we would like to increase our clock speed from 50MHz to 100MHz. This isn’t too difficult of a task, since all we need to do is add more pipeline stages to move evaluator.

One stretch goal that was too ambitious for our project was the addition of a transposition table. In future work, we would like to implement a transposition table in DRAM. Although the latency for DRAM is quite high, it could decrease our search time dramatically by reducing redundant work.

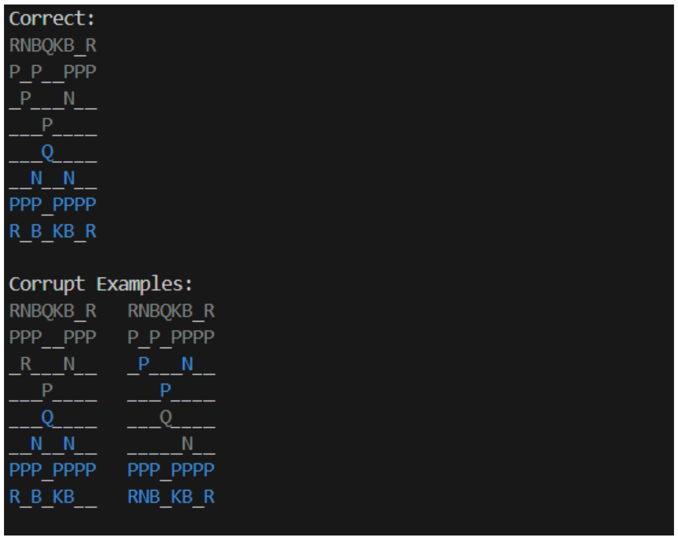
XI. Contribution

A. Dylan Isaac

In terms of SystemVerilog code, mainly focused on programming UCI Handler and Move Executor. Spent remaining time using UCI to test bench the system’s remaining modules in Python.

B. Arjun Barett

Created initial software model of River in C. Used software model to inform the design of each module. In SystemVerilog, programmed Move Generator, Move Evaluator, and Engine Coordinator.



```
Correct:
RNBQKB_R
P_P__PPP
_P___N__
___P____
___Q____
__N__N__
PPP_PPPP
R_B_KB_R

Corrupt Examples:
RNBQKB_R   RNBQKB_R
PPP__PPP   P_P_PPPP
_R___N__   _P___N__
___P____   ___P____
___Q____   ___Q____
__N__N__   _____N__
PPP_PPPP   PPP_PPPP
R_B_KB__   RNB_KB_R
```

Fig. 2. Corrupted Board representation likely from PLL instability