Hardware acceleration for General Game Playing using FPGA

(Sprzętowe przyspieszanie General Game Playing przy użyciu FPGA)

Cezary Siwek

Praca magisterska

Promotor: dr Jakub Kowalski

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

3 lutego 2020

Abstract

Writing game agents has always been an important field of Artificial Intelligence research. However, the most successful agents for particular games (like chess) heavily utilize hardcoded human knowledge about the game (like chess openings, optimal search strategies, and heuristic game state evaluation functions). This knowledge can be hardcoded so deeply, that the agent's architecture or other significant components are completely unreusable in the context of other games.

To encourage research in (and to measure the quality of) the general solutions to game-agent related problems, the General Game Playing (GGP) discipline was proposed. In GGP, an agent is expected to accept any game rules expressible by a formal language and learn to play it by itself. The most common example of the GGP domain is Stanford General Game Playing. It uses Game Description Language (GDL) based on the first order logic for expressing game rules.

One popular approach to GGP player construction is the Monte Carlo Tree Search (MCTS) algorithm, which utilizes the random game playouts (game simulations with random moves) to heuristically estimate the value of game state favourness for a given player. As in any other Monte Carlo method, high number of random samples (game simulations in this case) has a crucial influence on the algorithm's performance. The algorithm's component responsible for game simulations is called a reasoner.

The Field Programmable Gate Arrays (FPGAs) are chips, whose logic is designed to be configured after they were manufactured or even embedded in the final product (hence "field"). With them, it is possible to create a circuit that performs certain operations (like image processing or processor emulation) faster or (and) more energy efficiently than it is possible with software.

This work describes the implementation of a system, who given GDL rules, creates a hardware-accelerated reasoner with FPGA and a GGP agent who can use this reasoner to efficiently play the requested game. This thesis discusses multiple iterations of the development of the agent and contains an in-depth analysis of performance across system components and the system as a whole.

Streszczenie

Jedną z istotnych dziedzin badań nad szeroko pojętą Sztuczną Inteligencją jest tworzenie algorytmów zdolnych do grania w gry (agentów). Jednak zazwyczaj najlepsi agenci w konkretnych grach (np. szachach) wykorzystują znaczną ilość wiedzy dziedzinowej definiowanej przez ludzi (jak otwarcia szachowe, optymalne strategie przeszukiwania lub heurystyczne funkcje oceny stanu gry). Na tej wiedzy może być zbudowana zarówno cała architektura, jak i szczegóły implementacyjne agenta. To sprawia, że jego elementów często nie da się wykorzystać w innych zastosowaniach, niż gra w którą ma on grać.

By zachęcić do badanianń nad ogólnymi problemami związanymi z implementacją agentów gier (i żeby móc porównywać jakość tych ogólnych metod), została zaproponowana dyscyplina GGP (ang. General Game Playing). Agent GGP to agent zdolny do grania w dowolne gry, których zasady są wyrażalne w formalnym języku. Przykładową implementacją GGP jest Stanford General Game Playing. Do wyrażania zasad gry używa on opartego na logice pierwszego rzędu GDLa (ang. Game Description Language).

Jedną z najpoplarniejszych metod wykorzystywanych w agentach GGP jest algorytm MCTS (ang. Monte Carlo Tree Search), który wykonuje losowe przebiegi (symulowane gry, w których gracze wykonują losowe ruchy) w celu heurystycznej oceny sytuacji gracza w danym momencie rozgrywki. Jak w innych metodach Monte Carlo, duża liczba losowych próbek (w tym przypadku przebiegów gry) jest kluczowym czynnikiem dla jakości wyników algorytmu. Część agenta odpowiedzialna symulację gry (i w konsekwencji za wykonywanie losowych przebiegów) jest nazywana reasonerem (ang. reasoner).

FPGA (ang. Field Programmable Gate Array) jest cyfrowym układem elektronicznym, którego konfiguracja logiczna jest wprowadzana poza etapem produkcji, lub nawet po tym jak układ jak został już umieszczony w końcowym produkcie. Przy jego pomocy możliwe jest utworzenie układu wykonującego pewne operacje (jak przetwarzanie obrazu lub emulacja procesora) szybciej lub (i) energetycznie oszczędniej, niż jest to możliwe przy pomocy zwykłego oprogramowania.

Praca ta opisuje implementację systemu, który przyjmuje reguły gry w GDLu, a następnie tworzy na ich podstawie sprzętowy reasoner w FPGA oraz zdolnego do gry przy jego użyciu agenta. Praca przedstawia również analizę wydajności zarówno poszczególnych komponentów, jak i całego systemu na przestrzeni wielu iteracji rozwoju projektu.

Contents

1 Introduction **7**

2 Stanford General Game Playing **9**

2.1 General Game Playing 9

2.2 Game Description Language 10

2.3 Classic approaches 12

2.4 Propositional networks 13

2.5 Implementation 13

2.6 Extensions to GDL 15

3 Monte Carlo Tree Search **17**

3.1 Introduction 17

3.2 Implementation 17

3.2.1 Selection and UCB1 18

3.2.2 Expansion 19

3.2.3 Playout 19

3.2.4 Backpropagation 19

3.2.5 Move selection in games with joint actions 20

3.3 Parallel MCTS 20

3.3.1 Leaf parallelization 21

3.3.2 Root parallelization 21

3.3.3 Tree parallelization 21

4 Field Programmable Gate Array **25**

4.1 FPGA architecture 25
4.2 Altera Cyclone V Architecture 26

5 Implementation **31**
5.1 From GDL to Verilog 31
5.2 Reasoner 32
5.2.1 Possible improvements 35
5.3 Java agent 36
5.4 C++ agent 37

6 Experimental results **39**
6.1 Reasoner performance 39
6.2 Parallel MCTS performance 40
6.3 Agent's performance 42
6.4 Discussion 43
6.5 Utilization of the FPGA agent 45

7 Summary **47**

Appendices **49**
A Tictactoe as GDL 51
B Tictactoe as Propnet 52
C Tictactoe as Verilog 53
D Tools used 55

Chapter 1

Introduction

The purpose of this work is to study the feasibility of accelerating General Game Playing with FPGAs. To accomplish this, a complete Stanford GGP agent was developed and compared with the state of the art GGP player.

Chapter 1 describes the GGP discipline with an emphasis on the most popular Stanford's first order logic-based system. It discusses its implementation details and presents the most common approaches in the field.

Chapter 2 presents the Monte Carlo Tree Search (MCTS) algorithm, a method widely used for finding optimal moves in discrete games, and especially in GGP. It discusses the specific variants of the MCTS and presents changes required to arrive at implementation capable of playing Stanford GGP efficiently with hardware reasoner.

Chapter 3 describes the history and high-level construction of FPGAs with emphasis on elements crucial to the performance of hardware reasoner.

Chapter 4 documents the high-level implementation of the system: reasoner construction from game rules, data exchange mechanism between the reasoner and the agent, and two implementations of Stanford GGP agent. At the end of the chapter, possible improvements to the reasoner are listed.

Chapter 5 contains the results of selected empirical experiments performed on particular components of the system and the agent as a whole. It discusses each experiment, in particular, the comparison with the state of the art GGP agent.

The initial version of the system presented in this work was published in *AI 2018: Advances in Artificial Intelligence, Springer* [19].

Chapter 2

Stanford General Game Playing

2.1 General Game Playing

Though research in General Game Playing dates back to at least 1968 [15]. The first practical system for playing general games was the Metagame [14], introduced in 1992 by Barney Pell. Within it, it is possible to describe two-player, symmetrical, perfect information games on a rectangular board. The predicates that express the game dynamics are heavily based on chess. For example, the game definition language has special keywords for defining hopping (knight-like) movement, the piece captures, or promotions. It is therefore only possible to play chess-like board games in Metagame (however, the full rules of chess were actually impossible to express).

Such limitations were partially deliberate, as much of the historical research was focused on generalizations of chess. They were also desirable when Metagame was used for automatic generation on new games, ensuring the resulting games will only be constructed out of concepts familiar to humans.

A much more general system, the Stanford GGP [9][8][11][7], arrived in 2004 and gained so much popularity the term GGP is sometimes used to refer specifically to Stanford GGP. Unlike the Metagame, it does not include concepts from any particular type of game (like board games). Instead, it only relies on the most fundamental game concepts like a player, a move, or a score, while all the game dynamics are defined using the first-order logic. Despite this generality, GGP cannot express games with elements of randomness (like a roll of dice) or imperfect information (like only one player being able to observe a particular set of cards). Such restrictions allow GDL rules to be accepted by logic-based programming languages, like Prolog.

2.2 Game Description Language

GDL is a first-order logic language proposed to represent game rules in Stanford GGP in a compact and modular format. It has two syntaxes: LISP based, and Prolog based. The latter representation will be used throughout this presentation.

A game state in GDL is represented as a set of true facts. Special keywords, described in Tables 2.1 and 2.2, are used to define different game elements and the game dynamics. By processing the GDL rules, a player is able to reconstruct the dynamics of a finite state machine for the game (the game states are the machine's states and players' legal actions in particular states are the machine's transition edges).

Table 2.1: Predicate types of the Game Definition Language. An example consisting a full game is provided in appendix A.

Predicate	Description
role(role)	Defines a playable role within the game, like "white player" in chess: role(white).
base(predicate)	Defines a proposition, that has a truth value in every game state. For example, in chess, `base(control(white))` could define a propsition that is true in states where the white player makes the next move.
init(proposition)	Assigns truth to the base proposition in the first state of the game. For example in chess, we would express the white player having the move in the initial state by predicate: `init(control(white))`
true(base_proposition)	Evalutes a base proposition in a logical formula.
next(base_proposition)	Assigns truth to a proposition in the next game state. If a base proposition was not assigned by the next() predicate in previous state, it is considered false by default. In chess it allows us to express the alternation of moves by the white and the black player: `next(control(while)) :- true(control(black))` `next(control(black)) :- true(control(white))` or `next(whites_move) :- ~true(whites_move)`

Table 2.2: Predicate types of the Game Definition Language.

input(role, action)

Defines a possible set of actions for a particular role. Whenever the action is possible or not in a particluar state is covered by a different predicate type. For example in chess, we might write the following predicate to introduce the castling:

```
input(white, castle_left)
input(white, castle_right)
```

legal(role, action)

Defines if a predefined action is possible from a current game state. Usually it is implied from a logical formula made of base propositions. For example, the following predicate could define avaliability of the right castle to the white player (of course assuming all used base propositions are defined):

```
legal(white, right_castle) :-
        true(control(white)) &
        true(cell(f, 1, free)) &
        true(cell(g, 1, free)) &
        ~true(white_right_rook_touched) &
        ~true(white_king_touched) &
        ~true(check(black, e, 1)) &
        ~true(check(black, f, 1)) &
        ~true(check(black, g, 1))
```

does(role, action)

Indicates whenever a player performed a particular action during the last game state transition. For example, consider possible definition of the white_right_rook_touched predicate:

```
next(white_right_rook_touched) :-
        true(white_right_rook_touched) |
        does(white, right_castle) |
        does(white, move_right_rook(x, y))
```

terminal

If true, indicates that this is the terminal state of the game. For example:

```
terminal :- true(control(player)) &
        ~true(has_move(player))
```

goal(player, score)

If true in terminal state, indicates score for a particular player. For example:

```
goal(white, 100) :- true(check(white, x, y)) &
        true(cell(x, y, black_king))
```

It is, of course, possible to express rules of the same game in many non-trivially different ways. For example, for chess, we might define action space as tuples describing what piece to move, in which direction and how far. For example:

```
legal(white (bishop_black north_east 4))
```

But we might as well define the action space as pairs of positions:

```
legal(white (a1 d4))
```

which would express move of a piece from a1 to d4.

2.3 Classic approaches

Initial approaches to GGP initially tried to employ some mechanism that would produce knowledge for the agent about the game (for example, a function for comparison of states' favourness). This would be achieved for example by data mining and learning (ideally unsupervised), handcrafted game features detection (i.e. the concept of a board), or statistical methods. Such methods are known as *knowledge based*. Examples of successful agents employing such methods are:

- *Cluneplayer* [6] who uses generalized min-max search. Its heuristic evaluation function identifies such metagame-concepts as control (correlated with how much player's decisions can influence the future states), termination (expected game length), termination (expected game length), or payoff. It then tries to correlate such concepts with the state features using statistical methods. The Cluneplayer won the International General Game Playing Competition in 2005.
- *Fluxplayer* [16] attempts to identify semantic structures within the game rules (like a board). It also uses fuzzy logic and graph theory to estimate completeness of the score and terminal predicates in particular states.

In 2006, a Monte Carlo Tree Search (MCTS) algorithm was proposed. Unlike knowledge-based methods, it only relies on simulated games to arrive at the estimation of the state's favourness.

- *CadiaPlayer* [22] uses an open-source, high performance Prolog implementation (YAP-Prolog) as a reasoner. It uses statistical methods to introduce bias towards best promising moves based on historical data. The Similar successful agents were Ary and TurtleBot.

- *Dumalion* [13] performs graph-theory based optimizations to the game rules and thereby arrives at optimal computation routines for particular predicate types. It then expresses those routines in C++ code, which is in turn compiled into a complete reasoner.

2.4 Propositional networks

Because GDL does not support recursion or arithmetic, the set of all propositions must be finite and known before the match starts. Furthermore, the truth value of each proposition (next, legal, or does) is defined by a set of implications, so we may write it as an alternative of all the implications' bodies. The propositional networks are a natural representation of such GDL rules. They are directed graphs where propositions and logical conjunctions are the nodes, while the edges represent the logical dependencies between the nodes. We can interpret such a graph as a stateless (cycle-free) logical circuit, for which the inputs are the truth values of the base propositions (who encode the current game state) and the *does* propositions (who encode the player's actions). The output of this logical circuit are the *next* propositions, who define the next game state, the *legal* propositions, who define next possible actions, and the *terminal* and *score* propositions. If we now connect the *base* propositions with the corresponding *next* propositions trough a synchronous D-FlipFlop latch and feed players the moves at each clock cycle, the circuit will correctly and efficiently compute the complete game playout. In order to switch circuit from one game state to another game state, it is only required to externally override values of the base propositions.

Since their conception, propositional networks are a common choice for building fast reasoners, especially for the MCTS-based agents.

2.5 Implementation

Stanford provides a complete environment for agent development and competitions conducting in Java. Among other components, it contains highly integrated reference implementations of:

- the game server with match scheduling
- simple agents (like a random action agent, or Python agent),
- network communication protocol,
- GDL parser,
- game visualization view.

Game agents are expected to act as HTTP servers, while the actual game server issues match-related commands by sending HTTP requests to participating agents. Agents' answer is carried within the HTTP response. Table 2.3 describes possible commands.

Table 2.3: HTTP-based network protocol of the Stanford GGP.

Request	**Responce**	**Description**
(INFO)	((name <agent_name>) (status <busy | ready>))	Checks if the agent is avaliable for a match.
(START <game_type> <match_id> <role_name> <game_rules> <start_clock> <play_clock>)	<ready | not_ready>	Orders the agent to start a match and provides the game rules expressed in GDL. <start_clock> is a numerical value of time in seconds that the agent has to prepare for the match. The play clock is similar, but it is time the agent has to make each action.
(PLAY <game_type> <match_id> <joint_action>)	<action>	Orders the agent to make an action within the previously defined time constraint. <joint_action> describes previous actions of all players, which enables the agent to obtain new current game state. If this is the first *PLAY* command, the <joint_action> field has the value of *NIL*.
(STOP <game_type> <match_id> <joint_action>)	-	Communicates the last players' actions and indicates the end of the match.

In cases when a player failed to send a move within a game clock time constraint, the connection was lost or a player submitted an incorrect move, a random move is performed by the server for him, and all players are notified about the new joint move. If the agent supports such cases (like the system presented), matches with such situations can be salvaged.

2.6 Extensions to GDL

Several extensions to GDL and Stanford GGP were proposed:

- GDL-II [20] introduces non-determinism and non-perfect information. Random events can be modeled by a player (role) who does random moves. Non-perfect information, on the other hand, requires substantial changes to the game system and a new keyword – "sees". The game server is the only one who knows the full game state, and instead of the joint move, it sends to each player whichever observable event was deduced for this player.
- rtGDL [12] introduces real-time aspects, while maintaining the core structure of the reasoning. It is also compatible with GDL-II.
- GDL-III [21] introduces introspective elements of players' knowledge into the game.

Chapter 3

Monte Carlo Tree Search

3.1 Introduction

MCTS is a simulation-based, heuristic strategy search algorithm [5], originally designed for computer Go [17].

There are three core ideas to MCTS:

- Sampling: in the classical min-max algorithm, the expected outcome for a given player is calculated by a heuristic static evaluation function. However, in 1987, it was proposed to use random playouts. In this method, random moves are performed at each turn from the evaluated state, until the terminal state is reached. Game scores in the terminal state are treated as a random sample of the expected-outcome. This approach is knowledge-free, domain-independent, and as it will turn out, fast to compute.
- Treatment of non-terminal states as a series of multi-arm bandit problems: when looking for the most optimal move for each player, we want to only explore the most promising (or likely) paths in the game tree. Therefore, when traversing the game tree, we need to balance computational resources between exploitation (better estimating the most promising subtrees so far) versus the exploration (looking for new promising subtrees). In MCTS, samples are usually distributed according to the Upper Confidence Bound [4] approach.
- Propagation of results: when we obtain a sample for a particular state, we can treat it as a sample for all the previous states.

3.2 Implementation

At the beginning of every game turn, the game is in a known state, that we will call the current root state. In the first round, this state is just the absolute root

(initial) state of the game. Usually, there is a bijection between MCTS nodes and game states.

The MCTS repeats the following phases until a given search time budget expires.

3.2.1 Selection and UCB1

At the beginning of each iteration, the algorithm traverses the game tree built so far from the current root state of the game. A selection strategy is used to choose which player's action to simulate in each visited node until a state not yet in the tree is reached. One of the most commonly used selection strategies is UCB1, which is also used by the system presented.

During game tree descent, the selection strategy must balance between exploration (sampling of the uncertain game subtrees) and exploitation (sampling of the most promising subtrees). This is analogous to the multiarmed bandit problem, in which players must find the slot machine that maximizes reward, while he simultaneously maximizes the reward gathered so far.

UCB1 works by assigning confidence value for each player p to each of his action $a \in A$, and choosing the action with the highest confidence value. Confidence for action i in a single-threaded implementation is given by the following formula:

$$\frac{v_i}{n_i} + C\sqrt{\frac{ln(N)}{n_i}}$$

where:

v_i is the sum of reward samples of nodes that i leads to,

n_i is the sum of numbers of times the nodes that i leads to were sampled,

C is a tunable exploration/exploitation bias parameter,

N is the number of samples of the currently selected node,

$i \in A_p$ where A_p is a set of all actions for the player p in the currently selected state.

$C = \sqrt{2}$ is theoretically optimal value, provided the rewards are from $[0; 1]$ range.

As mentioned before, in GGP players may do their move simultaneously, so i may lead to multiple nodes. Therefore v_i carries the sum of rewards and n_i sum of the number of samples. In this implementation, each player independently selects a move that maximizes his UBC1 and assumes the other players will do the same.

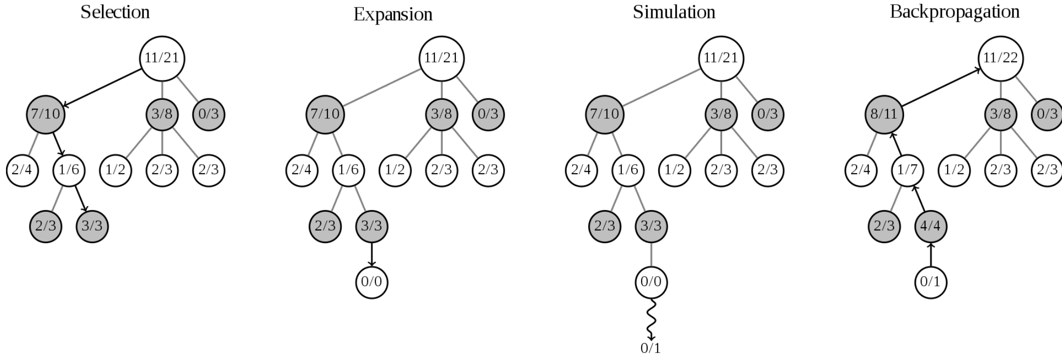


Figure 3.1: One iteration of the MCTS algorithm[1]

3.2.2 Expansion

When selection reaches a state in the simulation that was not part of the tree yet, this state is added to the tree as a new node. In this particular implementation, if a node was never selected, it is considered not-open. When a not-open node is selected for the first time, all of its children are added to the tree as non-open, and the selected node becomes open. In this implementation, actions leading to not-open nodes have assigned $v_i = 1$, $n_i = 2$ during UCB1 calculation. If the state is terminal, the expansion phase is not performed.

3.2.3 Playout

After the node is expanded, its expected outcome value for each player is sampled by performing a before-mentioned random game playout from its game state.

Uniquely to this implementation, the node is sampled multiple times (usually around 20) due to exceptional brute speed of the reasoner and comparatively high communication cost between the reasoner and the MCTS manager (which will be described in detail in the *Implementation* chapter). If the state is terminal, the playout phase is not performed.

3.2.4 Backpropagation

After the playout step is performed on the selected node, the sample s of the expected outcome for each player is propagated to each ancestor of the selected node.
More specifically, for each ancestor of the selected node:

s is added to a value of v_i, where i is an action that leads to the selected node,
N (the total number of the ancestor's samples) is increased by 1.

If the selected state is a terminal one, then s is substituted with the terminal scores.

3.2.5 Move selection in games with joint actions

After the time budget for action is about to end within a predefined safety margin, the loop performing MCTS phases is stopped and the best action for the player p from the current root state is computed. In this implementation, it happens by the selection of action i that maximizes the expected outcome:

$$\frac{v_i}{n_i}$$

where:

v_i is the sum of the reward samples of nodes that i leads to,

n_i is the sum of the numbers of times the nodes that i leads to were sampled.

Another popular best action criteria is a selection of the most sampled action (the action with the highest n_i).

In case of games with players who do their actions simultaneously, it may appear incorrect to sum sampled rewards for all nodes that particular action may result in, as there might be many "weak" actions for the opposing players, that should not be taken into account, as they certainly will not be selected. However, the mean expected outcome is weighted by the number of samples, so the "weak" actions will be heavily undersampled, and will, therefore, have a weak effect on the final expected-outcome value. This observation is also critical to the understanding of the selection step in games with simultaneous actions, and the importance of the C parameter tunning. If C is way too high, all actions are sampled equally and are therefore considered equally probable, even if not promising for a particular player.

Other approaches to games with simultaneous moves were studied by Mandy J. W. Tak et al. [1]

After the most promising move is submitted and the game server responds with the joint action of all players, the node that this joint action leads to becomes a new current root, and all nodes that are not its descendants are deleted from the agent's memory.

3.3 Parallel MCTS

As will be later explained, the main restriction on the presented system's performance is the speed of the search manager, as the reasoner spends much more time waiting for the manager than the other way around.

[1]The 4 steps of a Monte Carlo Search Tree expansion, Wikimedia Commons, Mciura, Dicksonlaw583

One solution to this problem is the parallelization of the search manager, so other thread can do the tree-related operations, while other waits for the reasoner's output. In the case of smaller games, it is also beneficial to employ multiple reasoners, each with an associated worker thread performing the tree operations and thereby maximizing utilization of available CPU power.

As it will be discussed later, the system presented so far achieves precise expected-outcome values on early stages, when the tree is comparatively shallow, but ultimately its main performance bottleneck is due to inability to construct deep-enough trees due to communication overhead between the reasoner and MCTS manager. Therefore, the parallelization method is expected to substantially increase the number of expansions, and thereby the number of total meaningful MCTS iterations.

The following approaches to parallel MCTS were proposed by the Guillaume M.J-B. Chaslot et al. [10]

3.3.1 Leaf parallelization

This is the simplest parallelization to implement, as it requires no synchronization mechanisms. It works by performing multiple simulations in the simulation step, so to achieve more precise samples. Only one thread performs all other MCTS' phases. This approach is therefore not useful for the presented system, as the same effect is already achieved thanks to the reasoner's speed, while the main problem, that is a comparatively low number of expansions remains unaddressed.

3.3.2 Root parallelization

This method works by parallel construction of separate MCTS trees by each thread through the entire turn time. After the time budget has expired, the trees are merged and the expected-outcome averaged out. This method also does not address the main bottleneck of the system. Although the total number of expansions is much higher, the vast majority of expansions are redundantly computed in all threads and the merged tree is not deeper than if it was constructed just by one thread. It only results in lower noise, something that the presented system already achieves with a single thread.

3.3.3 Tree parallelization

The last method is the only one that allows for simultaneous, full MCTS iterations. It was proposed with two key concepts. The first concept is mutex localization: it is proposed to either have a global mutex that locks the entire tree, while many threads perform the playout phase on many different leaves simultaneously, or to have threads perform the complete MCTS iterations and use mutexes on each node

to prevent data corruption. The solution similar to the second one is implemented in the presented work. A mutex lock is performed on every leaf during the expansion phase, in order to avoid data corruption to the container that holds the child nodes. Additionally, every variable in every node that holds the sum of rewards or the number of performed samples is wrapped around a spinlock-based container that guarantees atomic incrementation. As synchronization overhead increases with the size of the contained variable, each scalar value (like the sum of the rewards for a particular player in a particular node) is held in a separate container.

Though this approach guarantees proper state of nodes after the backpropagation phase, it may cause a wrong behavior during the selection phase, as in the situation when one thread performs a selection from a particular node while another thread backpropagates sampling results through the same node. In such a situation, the thread performing the selection may calculate UCB1 value based on rewards incremented for one player, while not incremented for other, and arrive at the result it would not make either before or after complete backpropagation. However, such error will not result in wrong reward values or wrong sampling counter, it only causes not-optimal sampling for one MCTS iteration. If such error occurs in the upper parts of the tree, it will likely be immediately corrected by the UCB1 formula not selecting again the oversampled action during the subsequent MCTS iterations. The likelihood of such an event occurring on lower parts of the tree is low, due to the second concept proposed by Guillaume M.J-B. Chaslot et al., namely the virtual loss.

The virtual loss is a modification of the original UCB1 formula, that decreases confidence value for nodes that were selected by other thread, but were not yet backpropagated by it. The system presented in this thesis, when built with parallelism enabled, applies virtual loss to the UCB1 formula in the following manner:

$$V^{t_i}\left(\frac{v_i}{n_i} + C\sqrt{\frac{ln(N)}{n_i}}\right)$$

4

V is a tunable penalty for choosing action that other threads are working on ,

t_i is the number of threads that are already working on the action i.

In other words: during the selection phase, the virtual loss makes nodes that are being worked on by other threads less attractive. This causes the selecting thread to avoid data-races by to exploring less likely, but still probable actions.

Proper tuning of the V parameter is crucial to the performance of the multi-threaded system. If it is too low, then all threads will follow the same path, thereby increasing the overhead of the spinlock-based synchronization containers and yielding ineffective expansions steps. The performance of such a system may be worse

than a single-threaded one. If it is too high, then all threads will avoid each other at the cost of entering the most unpromising paths, thereby wasting time and producing degenerated sampling distributions, which in turn may cause the move selection phase to overestimate the probability of unpromising actions.

Chapter 4

Field Programmable Gate Array

4.1 FPGA architecture

In 1969, Motorola introduced the X157 chip consisting of 12 logic gates and 30 IO pins. Unlike the previous Integrated Circuits, the logic gates were not permanently connected to the IO pins. Instead, X157 had to be first configured to perform a particular logical function. This marked the beginning of the configurable digital circuits. Trough 70s, Programmable Logic Array and Programmable Array Logic devices emerged, containing multiple types of logic gates and flip-flops. In 1984, Lattice Semiconductor invented GAL (Generic Array Logic) devices, which utilized electrically erasable floating gates to enable configuration the same chip multiple times. It allowed fast prototyping and efficient construction of circuits capable of multiple modes of operation. When GALs became much more complex and started utilizing non-trivial interconnections between layers of logic gates, a new name emerged for such types of devices: CPLD (Complex Programmable Logic Device). CPLDs are actively used and produced to this day.

In the late 80s, in parallel to the development of CPLDs, a new architecture of reconfigurable chips was proposed: FPGA. Instead of CPLD's sea-of-gates approach, it extensively uses lookup-tables and flip-flops inside a well-structured network of identical groups of logic gates (called programmable cells or logic elements).

Figure 4.1 shows a simplified logic element in FPGA used in this presentation. Besides obligatory lookup-table and synchronous flip-flop register, it also features two full adders, enabling efficient embedding of numerical addition within the circuit. Such architecture makes FPGAs more suited to the implementation of large state-machines, while CPLDs may perform better when the application demands high utilization of stateless, combinational logic.

[1] CV-51001: Cyclone V Device Overview, Intel

[2] UG-20152: Introduction to Intel FPGA SDK for OpenCL Standard Edition Best Practices Guide, Intel

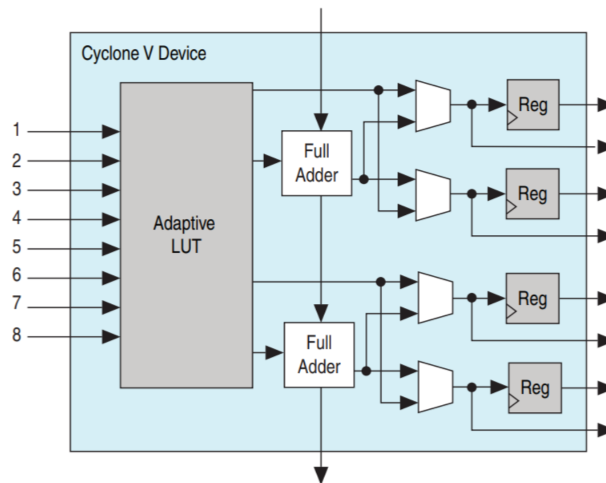


Figure 4.1: Example logic block of the modern FPGA. Clock input to the flip-flop registers is ommited[1].

Figure 4.2 shows modern FPGA construction at a more macroscopic level. It is made of thousands or millions of logic elements within a single FPGA chip. In high-end devices, the programmable interconnect network may contain programmable delay registers. There might also be digital signal processing modules, modules specialized for arithmetic, or memory blocks distributed across the entire device. Modern FPGAs also contain a specialized clock and reset distribution networks, phase-locked loop devices for clock signal generation and two way IO registers.

4.2 Altera Cyclone V Architecture

Besides purely computational operations, some applications (like GGP reasoners) require FPGA to efficiently handle execution of highly sequential control programs (for example TCP/IP stack implementation for communication). Two approaches are common:

- softcore processors: specially designed processor architectures to be efficiently synthesized within the FPGA fabric – for example Intel Nios II.
- hybrid chips with Hard Processor System (HPS) – the silicon fabric has separate spaces for FPGA and the HPS processor, which is designed as a classical integrated circuit. There usually many bridges for efficient communication between those two.

The work presented here uses the second approach with the Altera Cyclone V series SoC: 5CSEMA5F31C6. It is equipped with the two ARM Cortex A9 cores @900MHz and 1GB of RAM.

[3]cv_5v4: Cyclone V Hard Processor System Technical Reference Manual, Intel

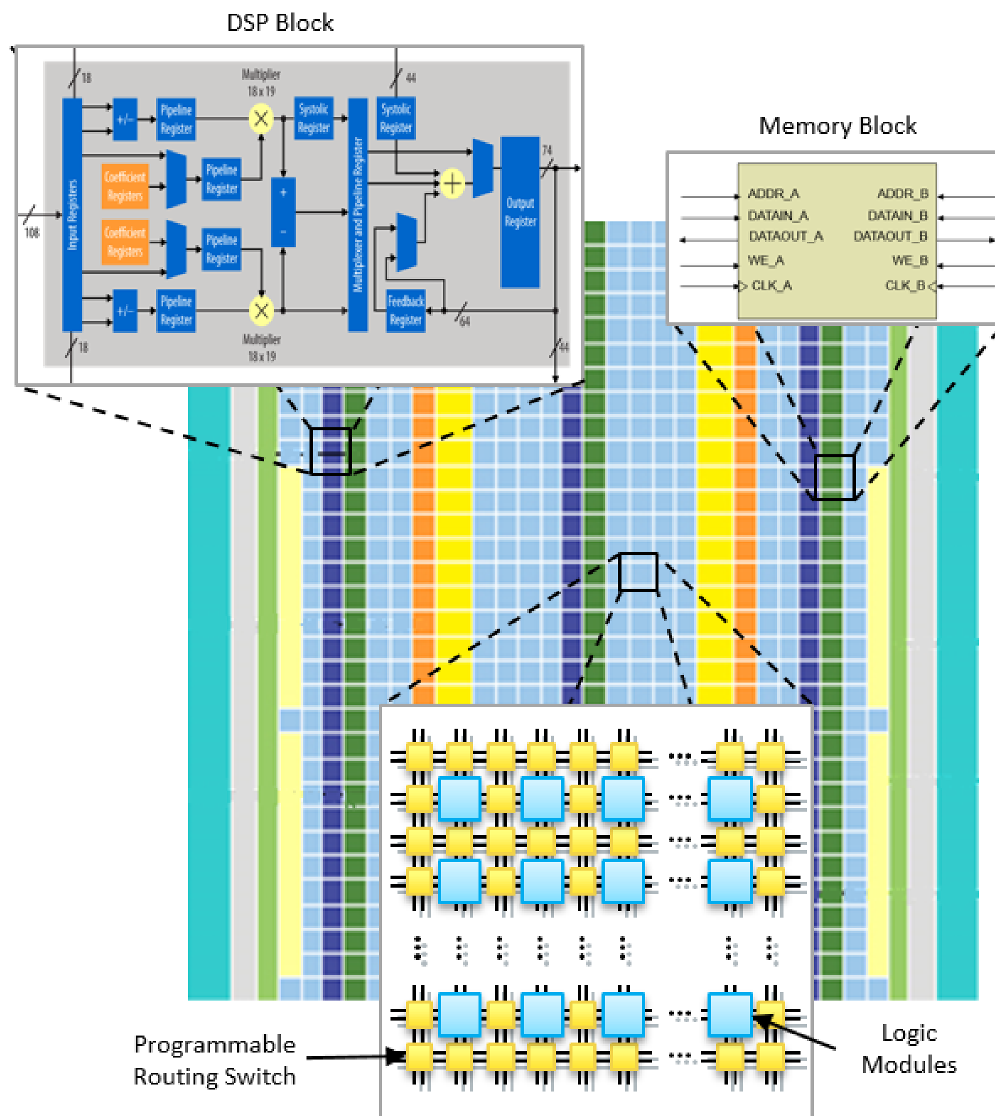


Figure 4.2: Example layout of a modern FPGA fabric[2].

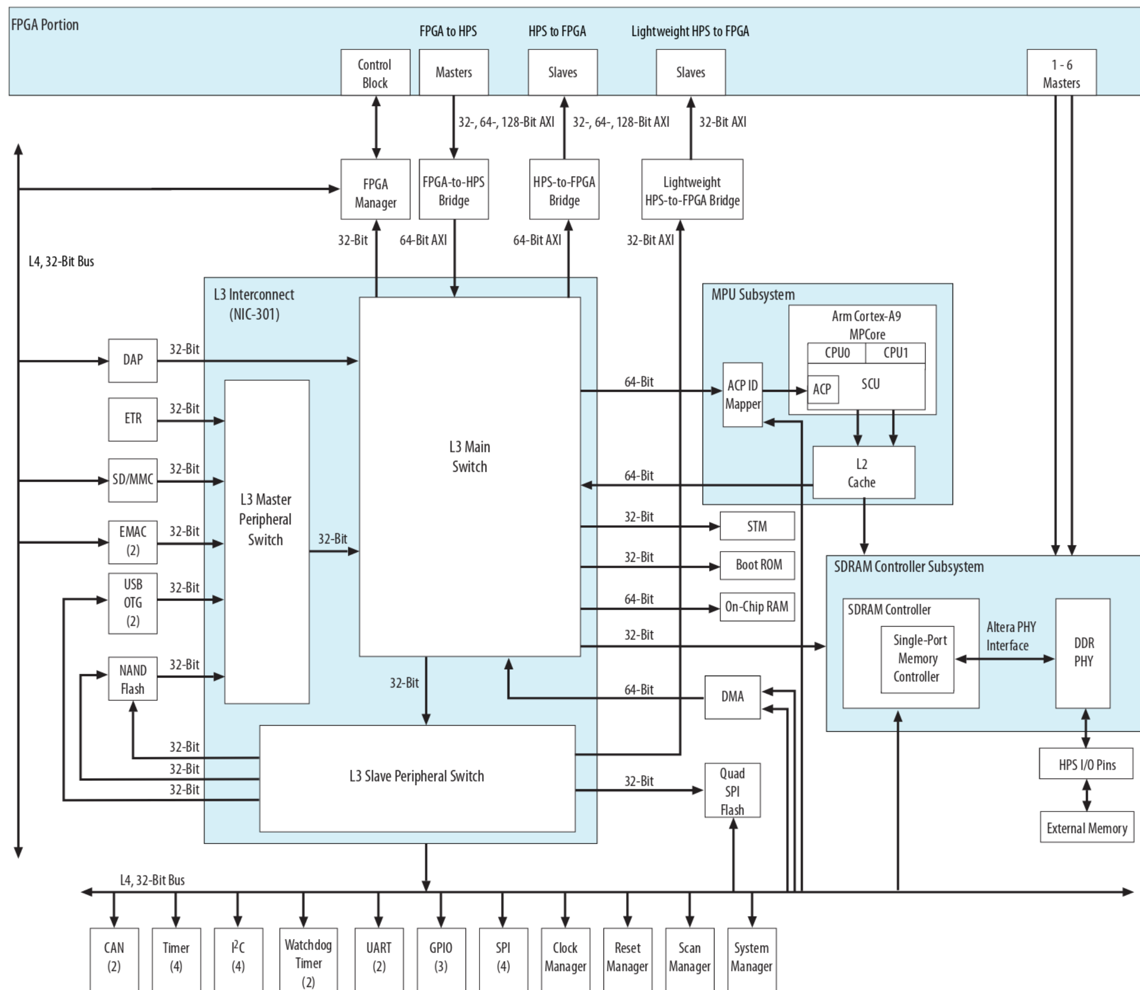


Figure 4.3: Architecture of an HPS-equipped Cyclone V SoC. Due to direct memory access for both the HPS and the FPGA, it is possible to freely assign IO devices to both of these two[3].

Figure 4.3 represents the architectural layout of the SoC. There are visible two types of interconnections between the HPS and the FPGA hardware reasoner:

- HPS-to-FPGA AXI Bridge – high bandwidth interface used for sending relatively large data, like discovered game states with associated scores, legal moves, or playout results,
- Lightweight HPS-to-FPGA Bridge – low latency interface for sending commands from HPS to the reasoners. Note that unlike AXI, Lightweight Bridge is connected to the L3 Peripheral Switch, yet the expected latency is lower than AXI's.

Chapter 5

Implementation

5.1 From GDL to Verilog

The presented system comes with a template project for the Quartus IDE. The only missing part is a definition of the actual propnet structure in propnet.v file. All other logical components of the propnet are prepared as Verilog modules who only need to be instantiated and linked with each other.

The first phase of the game preparation is the processing of the GDL rules to the software propnet form by the code from [18]. Next, a Java program does a BFS traversal of the underlying propnet graph, starting from the *input* propositions. The traversal algorithm instantiates the before-mentioned Verilog modules as it encounters new propnet components, and memorizes them in case of them being referenced by yet-undiscovered components. All components accessible from outside the propnet: *input, legal, transition, goal* are given additional, unique *id* numbers, and are connected to externally defined IO ports of the propnet module:

- *transition* components are grouped and connected to 128-bit read-write context registers, which allows for external game state readout or switch,
- *legal* and *input* components are connected with modules responsible for move randomizations and move enumeration, which allows for the propnet to internally generate correct random moves or iterate over all possible joint moves in a particular game state. Additionally, *id*s of the active *input* components are also exported out of the propnet module, which is used during the joint move enumeration phase. Those *id*s are also written into a separate XML metadata file that associates them with the corresponding GDL propositions. In the later stages, it will allow the game agent to decode binary representations of joint moves coming from the hardware reasoner.
- *goal* components are connected to the *scores* IO port, which is used to readout game results in terminal states.

The first phase is concluded by finding *transition* components associated with the *init* propositions and using their *ids* to derive a binary representation of the root game state. This information is appended to the previously mentioned XML metadata file.

The second phase is a compilation of the reasoner FPGA project. This phase is not automated and might require multiple trials since the success of the compilation depends on two key parameters that need to be estimated. The first is the number of parallel reasoners to synthesize on the FPGA (between 1 and 4 for the used Cyclone V chip). If this number is too high, the resulting structure might be too large for the hardware at hand. Another parameter is the propnet clock frequency. If it will be too high, the synthesis tool will unable to produce a structure for whom signal propagation time across the longest paths in the propnet fits within the clock period. If the compilation and synthesis step succeded, the second phase is concluded by a selection of the appropriate version of the agent program (what will be described in the *C++ Agent* section) and running it on the HPS computer.

5.2 Reasoner

The goal of the system is to implement a reasoner that can be effectively used by the MCTS algorithm. Thus, it needs to perform random simulations from an arbitrary game state to some terminal state, and report players' scores.

The search algorithm works on the integrated HPS computer. For the hardware reasoner to start playouts from a specific node, it has to be given a game state corresponding to this node. Thus, we require from the MCTS tree implementation to store data representing the internal state of the FPGA propnet. The root game state is provided by the before-mentioned XML metadata file, while the subsequent states are provided by the reasoner itself during the MCTS expansion phase.

The FPGA driver code exposes two core functions to the MCTS:

```
list<pair<joint_move_t, variant<score_t, fpga_state_t>>>
        getNextStates(fpga_state_t state)
```

which, for a given FPGA reasoner state, returns a list of all legal joint actions that might happen from it, associated with either the final game score or a next internal reasoner state, depending if a particular action leads to a terminal or a non-terminal state.

```
list<long> getScores(FPGAState state, int n)
```

computes, for each player, the scores obtained during a batch consisting of n random simulations.

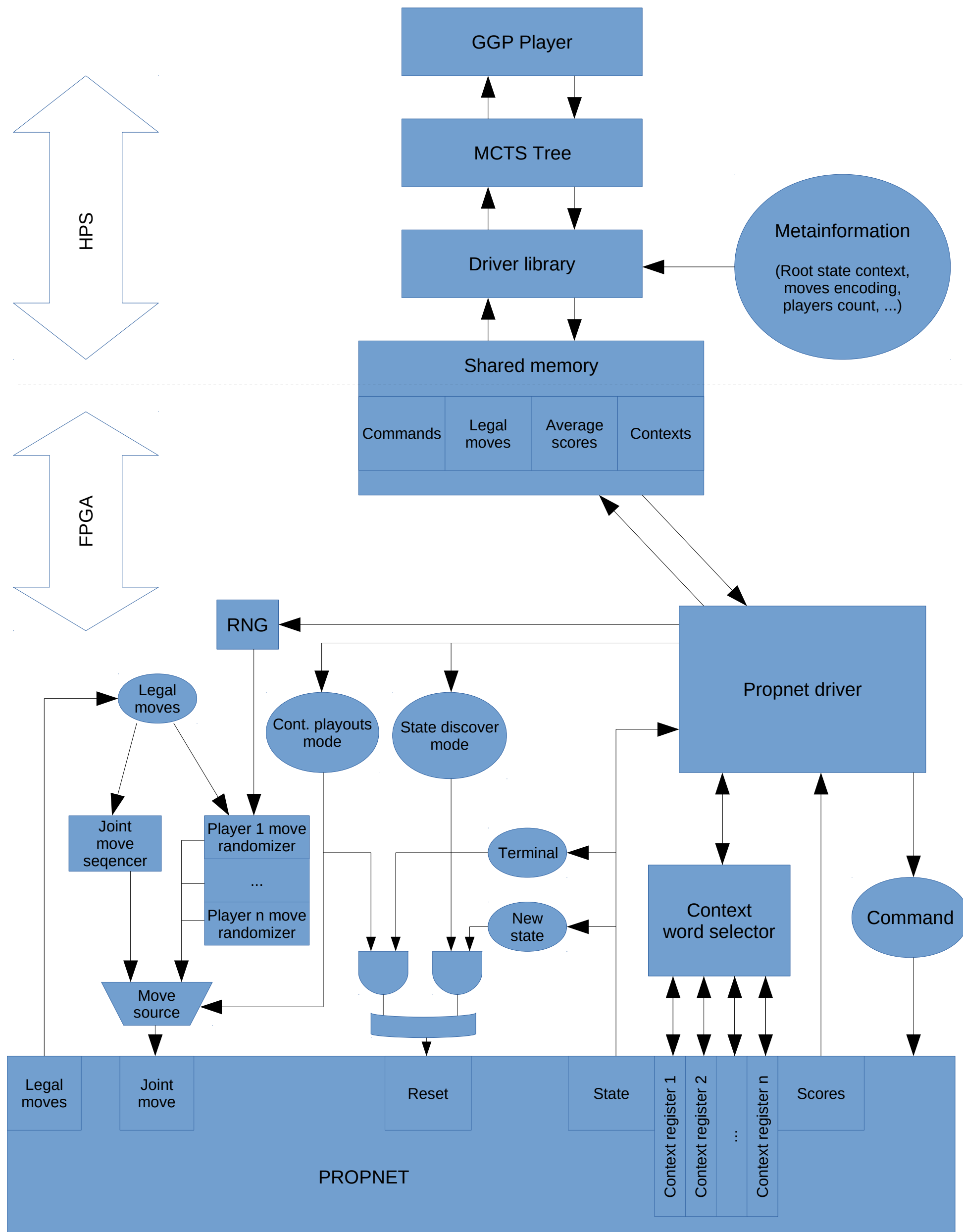


Figure 5.1: Diagram of the complete agent. If mutiple reasoners are used, all FPGA components are copied (including the shared memory).

Calling the reasoner to calculate a single playout, which is standard for software propnets, is very inefficient in our FPGA-based architecture, mainly because of communication costs. In order to reduce the number of read-write cycles, we only provide an interface for scheduling batched playouts. When simulating, MCTS uses the *getScores* function to request a specific number of playouts (it is an MCTS initialization parameter) and backpropagates the summarized scores.

Internally, those functions are realised in three modes of the FPGA reasoner operation: *state discovery*, *state switching*, and *continuous playout* by four essential components:

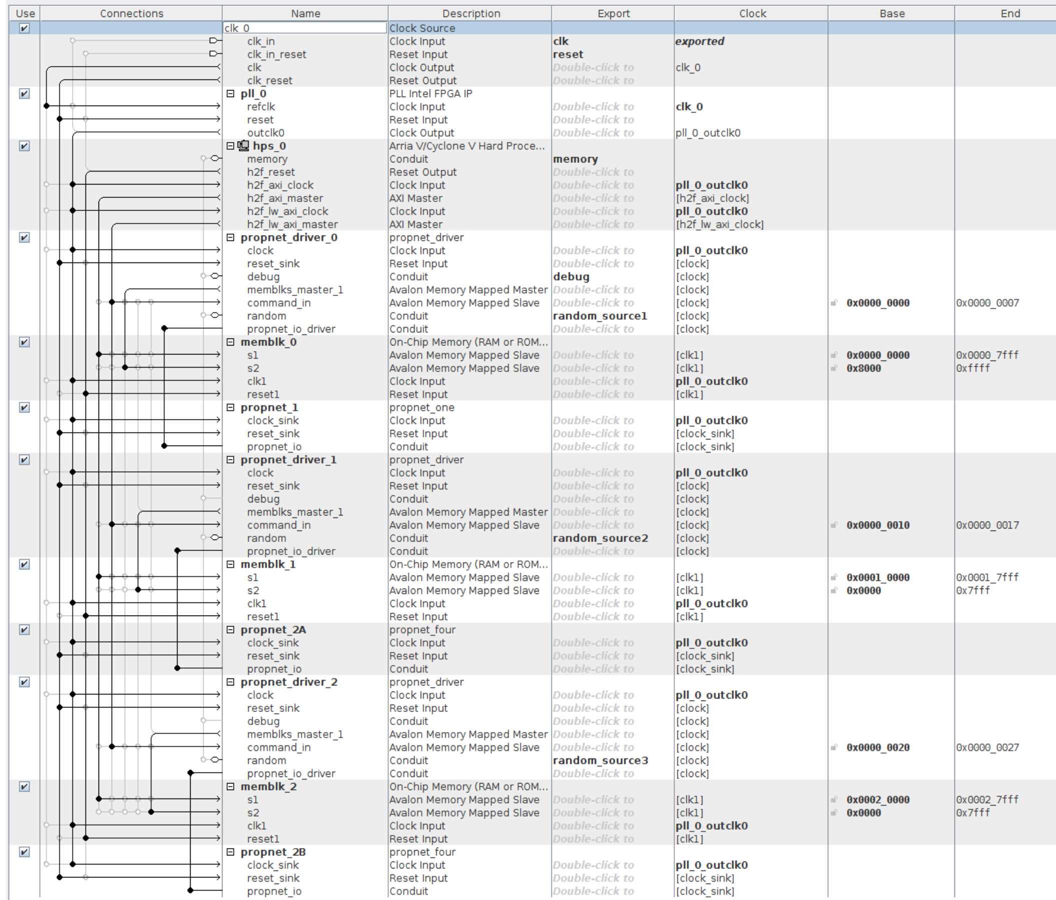
- propnet_driver – one for each reasoner. It is a simple state machine designed to receive commands from the HPS, control the propnet, handle memory operations and synchronize propnet operations with memory operations. If system is to be ported to another FPGA environment, only propnet_driver has to be rewritten.
- propnet – The part containing the game logic. Also contains logic for making continuous playouts (and thereby legal move randomization) and state discovery (and thereby iteration overall legal moves).
- Shared memory – accessed by both the propnet_driver and HPS for data exchange. It is a two-port memory synthesized inside the FPGA and accessed trough L3 AXI bridge by the HPS.
- RNG – one for each reasoner. Provides random numbers for moves randomizations during playouts.

The structure of dataflow between those structures is presented in Figure 5.1.

In the state discovery mode, all legal joint actions are iterated over by the move sequencer. For each joint action, after calculating the next state, the propnet driver starts forwarding context words (state representation) and the corresponding joint move into the shared memory. This can be a multi-step process, as the memory may force a stall of the state transfer until it is ready.

In the context switching mode, the propnet driver queries requested a place in the shared memory for the context words, which are then written to the propnet's context registers.

In the continuous playout mode, the players' actions are continuously taken from the modules generating legal random actions, until a terminal state is reached. When that happens, the propnet module signals scores to the propnet driver and resets the internal propnet to the previously set game state. To ensure the generated actions are uniformly distributed, for each player, we randomize a number i between 0 and the number of his legal actions, and loop through all his actions, reducing i on set bits, until the i-th legal action is found.



Use	Connections	Name	Description	Export	Clock	Base	End
		clk_0	Clock Source				
		clk_in	Clock Input	clk	exported		
		clk_in_reset	Reset Input	reset			
		clk	Clock Output	Double-click to	clk_0		
		clk_reset	Reset Output	Double-click to			
		pll_0	PLL Intel FPGA IP				
		refclk	Clock Input	Double-click to	clk_0		
		reset	Reset Input	Double-click to			
		outclk0	Clock Output	Double-click to	pll_0_outclk0		
		hps_0	Arria V/Cyclone V Hard Proce...				
		memory	Conduit	memory			
		h2f_reset	Reset Output	Double-click to			
		h2f_axi_clock	Clock Input	Double-click to	pll_0_outclk0		
		h2f_axi_master	AXI Master	Double-click to	[h2f_axi_clock]		
		h2f_lw_axi_clock	Clock Input	Double-click to	pll_0_outclk0		
		h2f_lw_axi_master	AXI Master	Double-click to	[h2f_lw_axi_clock]		
		propnet_driver_0	propnet_driver				
		clock	Clock Input	Double-click to	pll_0_outclk0		
		reset_sink	Reset Input	Double-click to	[clock]		
		debug	Conduit	debug	[clock]		
		memblks_master_1	Avalon Memory Mapped Master	Double-click to	[clock]		
		command_in	Avalon Memory Mapped Slave	Double-click to	[clock]	0x0000_0000	0x0000_0007
		random	Conduit	random_source1	[clock]		
		propnet_io_driver	Conduit	Double-click to	[clock]		
		memblk_0	On-Chip Memory (RAM or ROM...				
		s1	Avalon Memory Mapped Slave	Double-click to	[clk1]	0x0000_0000	0x0000_7fff
		s2	Avalon Memory Mapped Slave	Double-click to	[clk1]	0x8000	0xffff
		clk1	Clock Input	Double-click to	pll_0_outclk0		
		reset1	Reset Input	Double-click to	[clk1]		
		propnet_1	propnet_one				
		clock_sink	Clock Input	Double-click to	pll_0_outclk0		
		reset_sink	Reset Input	Double-click to	[clock_sink]		
		propnet_io	Conduit	Double-click to	[clock_sink]		
		propnet_driver_1	propnet_driver				
		clock	Clock Input	Double-click to	pll_0_outclk0		
		reset_sink	Reset Input	Double-click to	[clock]		
		debug	Conduit	Double-click to	[clock]		
		memblks_master_1	Avalon Memory Mapped Master	Double-click to	[clock]		
		command_in	Avalon Memory Mapped Slave	Double-click to	[clock]	0x0000_0010	0x0000_0017
		random	Conduit	random_source2	[clock]		
		propnet_io_driver	Conduit	Double-click to	[clock]		
		memblk_1	On-Chip Memory (RAM or ROM...				
		s1	Avalon Memory Mapped Slave	Double-click to	[clk1]	0x0001_0000	0x0001_7fff
		s2	Avalon Memory Mapped Slave	Double-click to	[clk1]	0x0000	0x7fff
		clk1	Clock Input	Double-click to	pll_0_outclk0		
		reset1	Reset Input	Double-click to	[clk1]		
		propnet_2A	propnet_four				
		clock_sink	Clock Input	Double-click to	pll_0_outclk0		
		reset_sink	Reset Input	Double-click to	[clock_sink]		
		propnet_io	Conduit	Double-click to	[clock_sink]		
		propnet_driver_2	propnet_driver				
		clock	Clock Input	Double-click to	pll_0_outclk0		
		reset_sink	Reset Input	Double-click to	[clock]		
		debug	Conduit	Double-click to	[clock]		
		memblks_master_1	Avalon Memory Mapped Master	Double-click to	[clock]		
		command_in	Avalon Memory Mapped Slave	Double-click to	[clock]	0x0000_0020	0x0000_0027
		random	Conduit	random_source3	[clock]		
		propnet_io_driver	Conduit	Double-click to	[clock]		
		memblk_2	On-Chip Memory (RAM or ROM...				
		s1	Avalon Memory Mapped Slave	Double-click to	[clk1]	0x0002_0000	0x0002_7fff
		s2	Avalon Memory Mapped Slave	Double-click to	[clk1]	0x0000	0x7fff
		clk1	Clock Input	Double-click to	pll_0_outclk0		
		reset1	Reset Input	Double-click to	[clk1]		
		propnet_2B	propnet_four				
		clock_sink	Clock Input	Double-click to	pll_0_outclk0		
		reset_sink	Reset Input	Double-click to	[clock_sink]		
		propnet_io	Conduit	Double-click to	[clock_sink]		

Figure 5.2: System layout in the QSYS tool.

Figure 5.2 represent modules layout in the QSYS tool who allows to design a system from high-level components. The propnet structure is contained in such components (*propnet_1*, *propnet_2A*, *propnet_2B*). All components who contain propnets accept the same signals (*propnet_io*), but are associated with different Verilog files. That allows for trivial control of the number of propnet instances to be embedded in the FPGA – some of the component's implementations are substituted with placeholders.

5.2.1 Possible improvements

Almost all bottlenecks of the system came from a low number of state expansions. This was primarily because of the relatively slow speed of the MCTS manager, and secondly due to latency for complete communication transactions between the reasoners and the MCTS manager. For example, in the first Java-based approach, it was typical for the reasoner to only work for 10% of the time during the gameplay. This caused main focus on optimizations to the software part of the system since no amount of improvements to the reasoner's speed could increase the overall system performance by 10%. Currently, in the multithreaded system, each reasoner typi-

cally works for 50% of the time, and if further improvements to the MCTS manager were made, or the other hardware was considered, the following improvements can also be applied:

- pipelining – propnet structure is mostly made out of simple combinational components, with the only minimal amount of trivial sequential logic. It, therefore, should be feasible and interesting to generate „instruction-level parallelism"-like pipelining, and increase maximum frequency by order of magnitude for some games.
- multiple clocks – currently, as shown in Figure 5.2, all system's components are clocked from a single PLL, whose frequency is limited by the longest combinational path within the propnet. Clock separation between the propnet and the memory-related parts of the *propnet_driver* could shorten propnet's stalls due to the pending memory operations.
- state cache – generate separate memory blocks (in FPGA) for storage of lastly added game states, and refer to them by index attached to the command sent through low-latency *lightweight* bridge. This may eliminate a good portion of the states copied trough the high-latency AXI bridge.
- multiple worker threads per reasoner – currently, the worker thread associated with a particular reasoner is blocked when this reasoner is performing state sampling. It could be beneficial to spend this time on HPS doing backpropagation of the previous sample, and at the same time have a third state selected, so as soon as sampling is completed, the sample backpropagation is enqueued, and the reasoner can immediately start computing the next expansion or the next sampling. This should significantly improve performance in games for whom the required propnet is so large that only one reasoner can be fitted.

5.3 Java agent

The first implementation of the game agent who used this hardware reasoner and was capable of playing full GGP matches was written in Java. It was built on top of the ggp-base project and used custom MCTS implementation provided by Chiara Sironi [18] To communicate with the reasoner, it was using a shared library written in C and accessible to the Java code by the JNI interface.

However, after the initial experimental results, it quickly proved to be the biggest bottleneck of the system. This was mainly because of the limited performance of the HPS computer in conjunction with the computation overhead and nondeterministic garbage collection of the Java Virtual Machine. In practical scenarios (like the Pentago game), the hardware reasoner was spending more than 85%

of total search time in the standby state waiting for the tree search manager to finish processing of the previous data and issue a new command.

5.4 C++ agent

The subsequent agent implementation was done in C++. Using templates, it produces separate implementations for different game parameters (most importantly state sizes), which makes it possible to directly embed the game state and other information directly within the MCTS structures. Agent builds for different game types can be either prepared in advance or automatically compiled when a new match request is received.

This, together with efficient and deterministic deallocation of the nodes and lack of JVM and JNI overhead results in a reduction of reasoner standby time to approximate 40% for Pentago with the same parameters as the Java agent was tested. Moreover, the C++ version of the agent also supports the usage of multiple reasoners in tree-parallelized MCTS.

Chapter 6

Experimental results

6.1 Reasoner performance

Hardware implementation of propositional networks is expected to have orders of magnitude higher performance than a software implementation because of zero computational overhead and simultaneous propagation of signals. The latter is essential, as in hardware, the simulation speed is dictated by the clock frequency, which is in turn constrained by the longest combinational path, not the total number of propnet components.

All the following tests were conducted with TerasIC DE1-SoC board containing the Altera's Cyclone V series SoC: 5CSEMA5F31C6. The GGP player, search algorithm and communication with the reasoner are run by a computer embedded in the before-mentioned SoC with ARM Cortex A9, Dual core @925Mhz with 1 GB RAM, running Debian 9 Strech 32-bit. The FPGA project compilation is performed on Intel Core i5-4670 with 16 GB DDR3 @1600Mhz RAM using Debian 10 64-bit and Intel Quartus Prime Lite Edition 18.0 as FPGA compilation IDE. Software propnets and the GGP-Base Prover are tested on a Linux server consisting of 64 AMD Opteron 6174 2.2-GHz cores and 252 GB RAM.

The first test evaluates the performance of the hardware reasoner against the state-of-the-art software reasoner, and a baseline Stanford GGP reasoner. The test is done by performing a million of random playouts (except for reversi, where 1000 playouts were performed) and measuring:

- agent's initialization time (for FPGA this includes the compilation and circut upload process),
- average reasoner's speed in moves (game state changes) per second,
- correctness: usually one of the players has a little advantage and thereby higher mean score on random playouts. If hardware reasoner is valid, it should

Table 6.1: Comparison of reasoners based on running Flat Monte Carlo algorithm

Game	Speed (avg nodes/sec)			Initialization time		#Propnet	FPGA chip
	FPGA	software	Prover	FPGA (min)	software (sec)	components	utilization
Horseshoe	8,500,000	192,583	3,812	4:20	0.45	350	7%
Connectfour	7,000,000	285,908	561	5:37	0.67	814	12%
Pentago	7,000,000	119,111	342	5:20	2.70	1,291	13%
Joint-connectfour	4,500,000	171,575	270	5:53	1.00	1,614	16%
Breakthrough	1,400,000	38,015	601	12:03	1.35	17,752	72%
Reversi	1,171,875	4,806	19	14:08	23.91	56,014	41%

produce the same score discrepancy as software reasoner, but in much shorter time.

Table 6.1 describes the result of a single hardware reasoner. It clearly shows the massive advantage of the hardware reasoner in performance and its massive disadvantage in initialization time. For all games except Reversi, the improvement factors are between 24.5 (Connect-Four) and 58 (Pentago). For Reversi, which produces the largest propnet among the tested games, FPGA-based reasoner computes states over 290 times faster. This example shows that smaller propnets do not necessarily imply smaller chip utilization.

The initialization time instead of seconds is about 5–6 minutes for small and medium games, and for large propnets it is almost a quarter. Such times exclude GGP players from being ready during their standard initialization clock. This issue is discussed in detail later and possible solutions are presented in the next section.

6.2 Parallel MCTS performance

After the single-threaded implementation was extensively tested against the classical Monte Carlo player bundled with the GGP-project, it was necessary to confirm that multi-threading and other optimizations of the C++ agent do not distort the UCB results. It is also helpful to estimate the multithreading overhead and the virtual loss penalty.

There are following build parameters to the agent build process:

- MCTS parallelization enabled (M) or disabled (S),
- optimizations on (R) or off (D),
- game state size.

For example, MD-8 denotes the parallel agent with optimizations disabled, supporting a game state of the maximum size of eight 32-bit words.

The test was conducted by computing 200 000 MCTS iterations from the root state of the Pentago game with the SD build and setting it as a baseline. Achieved

UCB values for the white player were saved and then recalculated by every configuration with much stricter time available. The multithreaded build was working using three hardware reasoners. In a correctly working system, it was expected that:

- given the same amount of time, the multithreaded build will perform more than 100% MCTS simulations in total, but there will be up to 50% less MCTS iterations per thread due to synchronization overhead and a limited number of two physical cores on the HPS,
- there should be a strong inverse correlation between the number of MCTS simulations and the mean error regardless of the configuration. Given the same amount of simulations, the multithreaded build might perform marginally worse due to the virtual loss penalty,
- optimizations should give a significant improvement in performance.

Table 6.2: Performance comparison of different builds of the MCTS manager. Errors are squared differences in UCT values between the tested configuration and the baseline. The UCT values are from 0-64 range.

Configuration:	**SD-8**	**SD-8**	**MD-8**	**MD-8**	**SD-8**	**MR-8**
Time (s):	**2**	**19**	**2**	**19**	**41**	**7**
MCTS interations:	**1314**	**11742**	**2879**	**25153**	**25255**	**23843**
Run 1 error:	22.96	4.46	11.72	1.65	2.24	3.81
Run 2 error:	22.31	3.46	10.31	2.25	2.25	2.58
Run 3 error:	35.18	6.19	8.77	2.36	1.93	2.11
Run 4 error:	31.34	8.44	6.28	2.85	1.56	2.48
Run 5 error:	17.04	2.99	10.12	2.00	2.32	1.81
Run 6 error:	27.54	2.99	12.38	2.18	2.75	3.46
Mean error:	**26.06**	**4.76**	**9.93**	**2.21**	**2.17**	**2.71**

All those predictions were fulfilled. The performance of a single thread in the multi-propnet build is slower by around 30%. This is because:

- virtual loss computation overhead,
- scores, simulation, and virtual loss must be stored in containers that guarantee atomic operations,
- mutex lock and release is required every expansion,
- in a multithreaded system every time a thread waits for the reasoner, std::thread::yield() function is called, which encourages the system scheduler to reschedule the currently executing thread. In the single-threaded version, on the other hand, the whole HPS processor is stalled until reasoner completes an operation, making possible to continue execution as soon as the reasoner's work is completed.

Game	Initialization time		FPGA		MCTS iterations / second		Score	
	Software (sec)	FPGA (hours)	Reasoners no.	Utilization	Software	FPGA	Software	FPGA
Reversi	23.91	4:51:58	2	73%	43	3207	0	80
Breaktrough	1.35	0:16:11	1	70%	390	1052	20	60
Pentago	2.70	0:16:21	4	81%	2189	6122	23	57
Joint-connectfour	1.00	0:11:04	4	52%	6825	2860	40	40
Joint-connectfour*	1.00	0:07:30	1	25%	6825	1580	58	22
Joint-connectfour**	1.00	0:11:04	4	52%	6825	3698	48	32
Connectfour	0.67	0:09:37	4	31%	14239	9685	58	22

Table 6.3: Results of the matches conducted between the FPGA player and the baseline software implementation. Each game was played 40 times. After each round two points were given to the victorious player or one point to both players in the case of a draft. The FPGA player was set to do 10 random playouts at each simulation phase, except for the test marked with **, when 1 playout was set. The game clock was set to 5 seconds. The distribution of the scores was non-trivial for each game (i.e. no high number of drafts or near-100% correlation between the role and the game result). *MCTS iterations/second* values were sampled during the first step of the match. Joint-connectfour was separately tested with multiple reasoners disabled in the test marked with *.

The virtual loss penalty was found to be insignificant (the difference in mean error between MD-8 and SD-8 runs is very insignificant).

6.3 Agent's performance

To ensure proper fine-tunning of the V and C MCTS parameters, two sessions of trial-and-error tests were conducted. A python script was used to run 14 instances of the FPGA agents, each with different parameters. Then, the same script conducted 20 matches against the software player with a fixed configuration. After coarse values were obtained, the test was rerun for finer adjustment with parameters closely clustered around the previously found optimal value. All tests were conducted on the Joint-connectfour game. The difference in winrate between guessed parameters ($V = 0.85, C = 0.707$) and the found ones ($V = 0.985, C = 0.55$) was around 20%.

Then, the most comprehensive test was run: a session of matches where the FPGA agent played 40 full matches against the state of the art GGP player on 5 different games. The results are presented in Table 6.3.

The last conducted experiment was to measure how the game progress affects the *MCTS iterations per second* discrepancy. When a match is in its final phases, the MCTS subtrees become shallow and devoid of open nodes. The *MCTS iterations per second* then depends less on the reasoner speed, and more on the MCTS routines, who in case of the hardware agent, are executed on the HPS. To measure this effect, an experiment was conducted, in which 560 Joint-connectfour matches were played. Then, the distribution of game lengths was examined and four game lengths, each with more than 15 samples, selected. The samples were then averaged in terms of

MCTS iterations per second at each match step and presented in figure 6.1.

6.4 Discussion

In Table 6.1 we observe that the speed of the agent expressed in the number of full MCTS iterations per second is closely related to the game complexity (expressed in the number of propnet components) for software implementation. For the hardware implementation, however, there is no such correlation. The following reasons might be the cause.

- There are additional game parameters that are not correlated with game complexity but affect the computational work required to complete an MCTS iteration. The average match length is the most notable example. For hardware reasoner, another such crucial parameter is the average number of legal actions and size of the game states, which affect communication overhead.

- As shown in Table 6.1, hardware reasoner's clock is much less correlated with the game complexity than software's nodes per second metric. As a result, the „noise" introduced by the previous point might be higher than whatever correlation remains.
  The principal example is the most complex game tested, the Reversi/Othello, for which the FPGA agent achieves two orders of magnitude advantage in speed and wins every single match.

The experiment clearly shows the massive advantage of the FPGA agent in complex games, as well as its disadvantage in the simple ones. Joint-connectfour is an interesting example for two reasons:

- it is a game whose complexity is just right so the performance of the software and hardware implementations match,
- software reasoner does about twice more MCTS iterations per second, however, the win rate is the same for both agents. It is important to remember that the FPGA agent does 10 game playouts at every simulation step, while the software version does only one. This significantly decreases the noise of the simulation results and may balance out less MCTS iterations. This is further indicated by results of the match marked with *, where the number of playouts was set 1, which increased the number of MCTS iterations but reduced overall score, confirming the positive effect of batching playouts.

As shown in Table 6.1, compilation time for a single Reversi hardware reasoner is about 12 minutes. However, during testing of the full agent who used two such

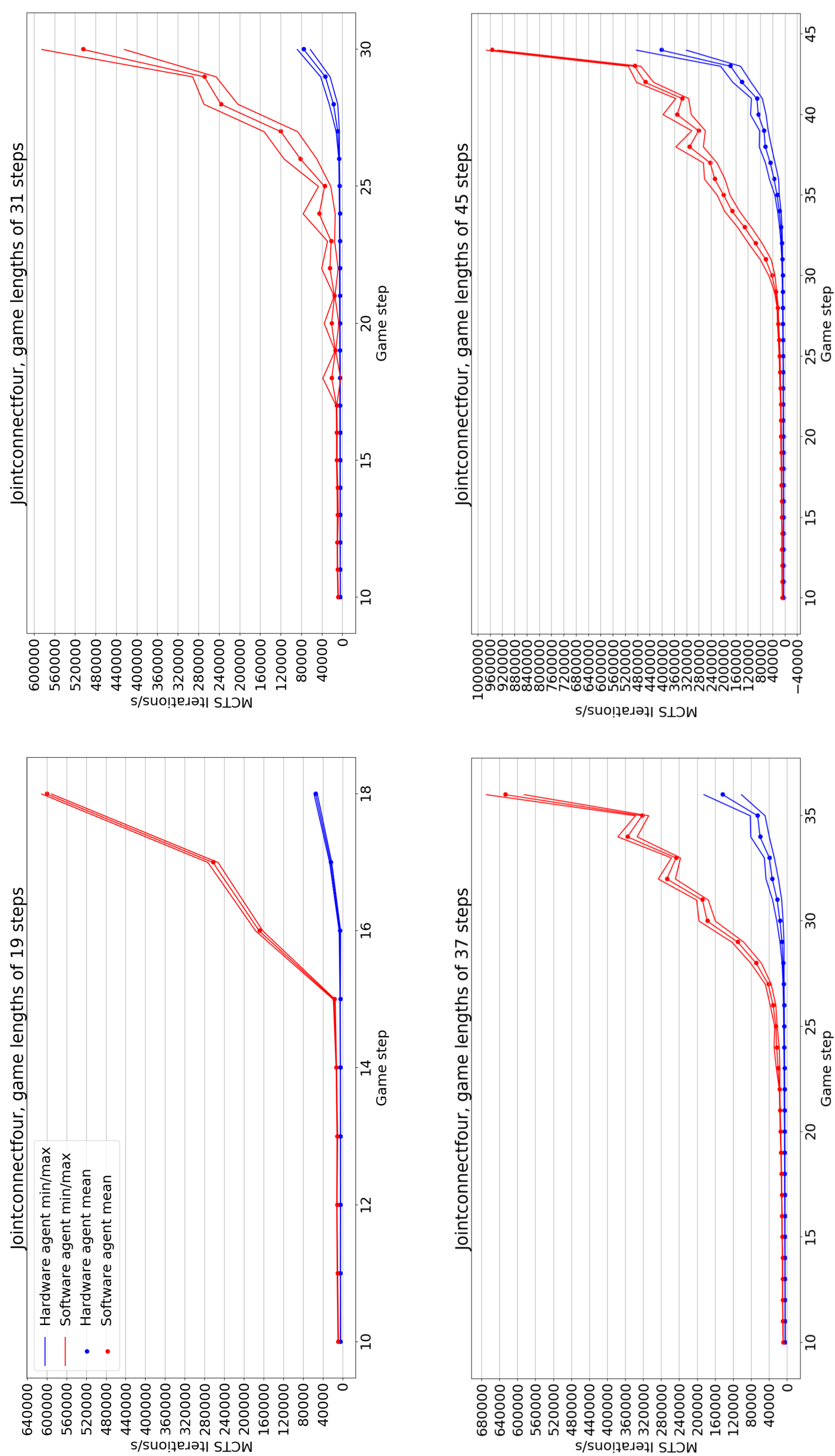


Figure 6.1: Comparisons of numbers of *MCTS iterations / second* for software and hardware players at different game phases.

reasoners, the compilation time increased to almost 5 hours. Reasons for such a dramatic increase in compilation time are unclear, as it does not happen in other games. For example, it takes about 5 minutes to compile a system with a single Pentago reasoner, and it only takes about 16 minutes to compile a system with 4 such reasoners. Moreover, in the case of 4 reasoners Pentago system, the reported FPGA chip utilization is higher than 2 reasoners Reversi system, yet the compilation time is an order of magnitude lower.

Figure 6.1 shows that discrepancy between software and hardware agent only gets higher as the game in Joint-connectfour progresses (from twice the value to order of magnitude for certain game lengths). This does not imply the implementation of the MCTS in the hardware agent performs worse, because the HPS has much less computing power than Opteron 6174, who runs the software agent (i.e. 48-core Opteron 6174 achieved 27241 score in Geekbench 2 benchmark [2], while the HPS only scored 872 [3]).

6.5 Utilization of the FPGA agent

The current implementation extensively uses Intel's Quartus Prime and QSYS features, which are not trivial to automatize. For two player games, the current system needs a human to estimate the number of parallel reasoners and rerun the compilation if the estimation was too high. If the game has more than two players, a manual change of widths of the buses and memory layouts is required in the QSYS environment. However, those interventions do not require a human to make any decision process that cannot be clearly and algorithmically written. They are only due to how Intel's FPGA development tools practically work and can all be automated in principle.

Long initialization time is the main reason why FPGA player cannot compete in normal GGP competitions. With industry-standard software, the compilation times are about an order of magnitude longer than usual GGP limits, even for small and medium games.

There are however other scenarios, where long compilation time can be outweighed by the huge performance boost of the presented system. Consider MCTS research, where there might be a need for an experiment that evaluates a selection strategy by performing a million of MCTS iterations in a game as complex as Reversi. Furthermore, assume the reasoners are already built since this is not the first experiment on Reversi. Conducting such an experiment would take 6.5 hours with the current state of the art GGP player, while the hardware agent presented here would only need 5 minutes (according to results from table 6.3). Even if the reasoner had to be built, the hardware agent would have still have completed the experiment 1.5 hours sooner.

Chapter 7

Summary

The purpose of the study was fulfilled. The dynamic construction of the GGP reasoner was found possible for the small and medium-large game sizes on relatively cheap hardware ($155 in 2019). Resulting reasoner is orders of magnitude faster than its state of the art software counterpart. The larger the game is, the higher is the relative discrepancy between those reasoners, provided the propnet structure will fit into the FPGA chip.

While the game size limit of the presented process can be mitigated with better hardware, the construction time of the reasoner is orders of magnitude longer than for software and beyond usual limits of the GGP competitions, even for the smallest games.

When coupled with the full agent running on the non-FPGA processor and tested in the real GGP match, the reasoner is underutilized in all tested scenarios and therefore agent loses much of its reasoning speed advantage on the tested hardware. This problem was partially mitigated by arriving at second, highly optimized MCTS implementation, using multithreading and batching certain operations. Given enough initialization time, the resulting agent wins against the state of the art software agent running on a high-end computer in medium-large games each time, and most of the times in medium games. Its performance matches the software agent in Joint-connectfour and it performs worse in smaller games.

The study finds the hardware acceleration is unfeasible for GGP competitions but potentially very useful for MCTS and GGP research, where it can reduce the time of conducting certain experiments by two orders of magnitude.

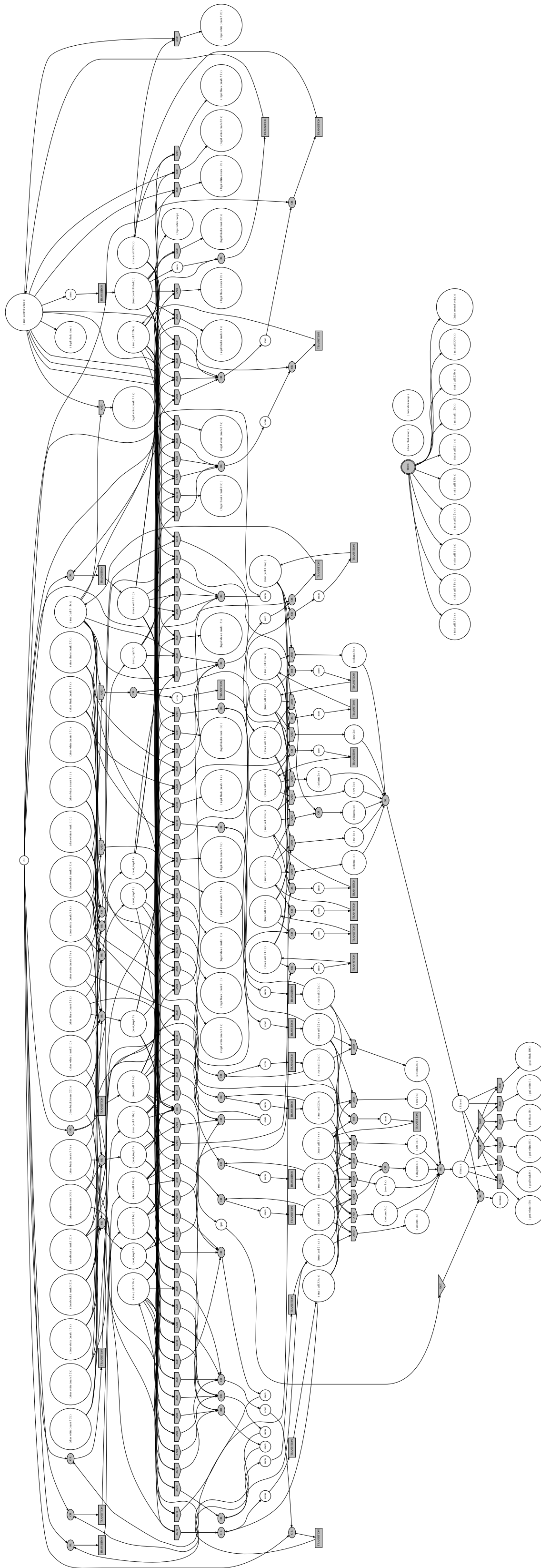
Appendices

A Tictactoe as GDL

```
role(white)
role(black)

base(cell(M,N,x)) :- index(M) & index(N)
base(cell(M,N,o)) :- index(M) & index(N)
base(cell(M,N,b)) :- index(M) & index(N)

base(control(white))
base(control(black))

input(R,mark(M,N)) :- role(R) &
        index(M) & index(N)
input(R, noop) :- role(R)

index(1)
index(2)
index(3)

init(cell(1,1,b))
init(cell(1,2,b))
init(cell(1,3,b))
init(cell(2,1,b))
init(cell(2,2,b))
init(cell(2,3,b))
init(cell(3,1,b))
init(cell(3,2,b))
init(cell(3,3,b))
init(control(white))

legal(W,mark(X,Y)) :-
  true(cell(X,Y,b)) &
  true(control(W))

legal(white,noop) :-
  true(control(black))

legal(black,noop) :-
  true(control(white))


next(cell(M,N,x)) :-
  does(white,mark(M,N)) &
  true(cell(M,N,b))

next(cell(M,N,o)) :-
  does(black,mark(M,N)) &
  true(cell(M,N,b))

next(cell(M,N,W)) :-
  true(cell(M,N,W)) &
  distinct(W,b)

next(cell(M,N,b)) :-
  does(W,mark(J,K))
  true(cell(M,N,b)) &
          distinct(M,J)

next(cell(M,N,b)) :-
  does(W,mark(J,K))
  true(cell(M,N,b)) &
          distinct(N,K)

next(control(white)) :-
  true(control(black))

next(control(black)) :-
      true(control(white))

goal(white,100) :- line(x) & ~line(o)
goal(white,50) :- ~line(x) & ~line(o)
goal(white,0) :- ~line(x) & line(o)

goal(black,100) :- ~line(x) & line(o)
goal(black,50) :- ~line(x) & ~line(o)
goal(black,0) :- line(x) & ~line(o)

line(Z) :- row(M,Z)
line(Z) :- column(M,Z)
line(Z) :- diagonal(Z)

row(M,Z) :-
      true(cell(M,1,Z)) &
      true(cell(M,2,Z)) &
      true(cell(M,3,Z))

column(M,Z) :-
      true(cell(1,N,Z)) &
      true(cell(2,N,Z)) &
      true(cell(3,N,Z))

diagonal(Z) :-
      true(cell(1,1,Z)) &
      true(cell(2,2,Z)) &
      true(cell(3,3,Z)) &

diagonal(Z) :-
      true(cell(1,3,Z)) &
      true(cell(2,2,Z)) &
      true(cell(3,1,Z)) &

terminal :- line(x)
terminal :- line(o)
terminal :- ~open

open :- true(cell(M,N,b))
```

Source: Stanford's Public Course on GGP

B Tictactoe as Propnet



C Tictactoe as Verilog

```
    module propnet (
            input  wire         clock_sink_clk,   // clock_sink.clk
            input  wire         reset_sink_reset, // reset_sink.reset
            input  wire [15:0]  command,          // propnet_io.command
            output wire [127:0] context_out,      //           .context_out
            output wire         terminal,         //           .terminal
            output wire [31:0]  scores,           //           .scores
            input  wire [15:0]  random_source,    //           .random_source
            output wire [127:0] debug,            //           .debug
            input  wire [127:0] context_in,       //           .context_in
            output wire [31:0]  players_moves,    //           .players_moves
            output wire [1:0]   context_valid,    //           .context_valid
            output wire [15:0]  context_size      //           .context_size
    );

assign scores[15:0] = terminal ? player0_score : 0;
assign scores[31:16] = terminal ? player1_score : 0;

wire[15:0] player0_seqmove;
wire[15:0] player1_seqmove;
wire[15:0] player0_randmove;
wire[15:0] player1_randmove;

wire[7:0] context_request = command[15:8];

wire [15:0] player0_move = command[2] ? player0_seqmove : player0_randmove;
wire [15:0] player1_move = command[2] ? player1_seqmove : player1_randmove;

wire soft_rst = command[1] || terminal;

wire clk = clock_sink_clk;
wire rst = reset_sink_reset;
wire protected_step = command[0] && !terminal && (!context_valid[0]);
    wire context_valid_seq;
    reg context_valid_reg;

assign active_player_move = player0_move;

reg step_done;
assign context_valid[0] = step_done && command[2];

assign players_moves[15:0] = player0_move;
assign players_moves[31:16] = player1_move;

always @(posedge clk or posedge reset_sink_reset)
begin
      if(reset_sink_reset)
      begin
                step_done <= 0;
      end
      else
      begin
            if(soft_rst)
                    step_done <= 0;
                            else
                                    step_done <= step_done | command[0];
      end
end

    wire [28:0] current_context;
    wire [28:0] return_context;

    pn_muxer128 #(1) context_muxer(current_context, context_request, context_out);
    pn_demuxer128 #(1) context_demuxer(clk, context_in, context_request, command[3], return_context);

    wire [9:0] player1_move_onehot;
    wire [9:0] legals1;
    wire [9:0] player0_move_onehot;
    wire [9:0] legals0;
    pn_move_selector #(10) move_selector1(legals1, random_source, player1_randmove);
    pn_onehot #(10) move_mux1(player1_move, player1_move_onehot);

    pn_move_selector #(10) move_selector0(legals0, random_source, player0_randmove);
    pn_onehot #(10) move_mux0(player0_move, player0_move_onehot);

    pn_move_sequencer #(10, 10) sequencer(clock_sink_clk, reset_sink_reset, command[2], command[0], legals0, legals1, player0_seqmove, player1_seqmove, context_valid[1]);

    assign context_size = 1;
    wire[15:0] player1_score = (goals_1 * 0) + (goals_3 * 100) + (goals_4 * 50) + 0;
    wire[15:0] player0_score = (goals_0 * 100) + (goals_2 * 0) + (goals_5 * 50) + 0;
    assign current_context[0] = output_98;
    assign current_context[1] = output_105;
    assign current_context[2] = output_122;
    assign current_context[3] = output_128;
    assign current_context[4] = output_138;
    assign current_context[5] = output_143;
    assign current_context[6] = output_148;
    assign current_context[7] = output_153;
    assign current_context[8] = output_158;
    assign current_context[9] = output_185;
    assign current_context[10] = output_195;
    assign current_context[11] = output_202;
    assign current_context[12] = output_206;
    assign current_context[13] = output_211;
    assign current_context[14] = output_217;
    assign current_context[15] = output_221;
    assign current_context[16] = output_225;
    assign current_context[17] = output_229;
    assign current_context[18] = output_233;
    assign current_context[19] = output_236;
    assign current_context[20] = output_240;
    assign current_context[21] = output_243;
    assign current_context[22] = output_245;
    assign current_context[23] = output_246;
    assign current_context[24] = output_247;
    assign current_context[25] = output_248;
    assign current_context[26] = output_249;
    assign current_context[27] = output_262;
    assign current_context[28] = output_265;
    wire output_0;
    wire output_1;
    wire output_2;
    wire output_3;
    wire output_4;
    wire output_5;
    wire output_6;
    wire output_7;
    wire output_8;
    wire output_9;
    wire output_10;
    wire output_11;
    wire output_12;
    wire output_13;
    wire output_14;
    wire output_15;
    wire output_16;
    wire output_17;
    wire output_18;
    wire output_19;
    wire [5:0] input_20;
    wire output_20;
    wire [5:0] input_21;
    wire output_21;
    wire [1:0] input_22;
```

```
        wire output_22;
        wire [1:0] input_23;
        [...]
        assign input_20[0] = output_3;  assign input_20[1] = output_17; assign input_20[2] = output_12; assign input_20[3] = output_0;  assign input_20[4] = output_19;
        assign input_21[0] = output_11; assign input_21[1] = output_14; assign input_21[2] = output_2;  assign input_21[3] = output_0;  assign input_21[4] = output_4;
        assign input_22[0] = output_0;  assign input_22[1] = output_47;
        assign input_23[0] = output_49; assign input_23[1] = output_1;
        assign input_24[0] = output_8;  assign input_24[1] = output_11; assign input_24[2] = output_2;  assign input_24[3] = output_15; assign input_24[4] = output_18;
        assign input_25[0] = output_5;  assign input_25[1] = output_17; assign input_25[2] = output_15; assign input_25[3] = output_1;  assign input_25[4] = output_7;
        assign input_26[0] = output_2;  assign input_26[1] = output_53;
        assign input_27[0] = output_3;  assign input_27[1] = output_55;
        assign input_28[0] = output_16; assign input_28[1] = output_6;  assign input_28[2] = output_8;  assign input_28[3] = output_3;  assign input_28[4] = output_12;
        assign input_29[0] = output_6;  assign input_29[1] = output_16; assign input_29[2] = output_5;  assign input_29[3] = output_14; assign input_29[4] = output_4;
        assign input_30[0] = output_59; assign input_30[1] = output_4;
        assign input_304[0] = output_300;

        [...]

        pn_input #(0) component_9 (output_9, player0_move_onehot[9]);
        pn_input #(0) component_10 (output_10, player1_move_onehot[0]);
        pn_input #(0) component_11 (output_11, player1_move_onehot[1]);
        pn_input #(0) component_12 (output_12, player1_move_onehot[2]);
        pn_input #(0) component_13 (output_13, player1_move_onehot[3]);
        pn_input #(0) component_14 (output_14, player1_move_onehot[4]);
        pn_input #(0) component_15 (output_15, player1_move_onehot[5]);
        pn_input #(0) component_16 (output_16, player1_move_onehot[6]);
        pn_input #(0) component_17 (output_17, player1_move_onehot[7]);
        pn_input #(0) component_18 (output_18, player1_move_onehot[8]);
        pn_input #(0) component_19 (output_19, player1_move_onehot[9]);
        pn_or #(6) component_20 (input_20, output_20);
        pn_or #(6) component_21 (input_21, output_21);
        pn_and #(2) component_22 (input_22, output_22);
        pn_and #(2) component_23 (input_23, output_23);
        pn_or #(6) component_24 (input_24, output_24);
        pn_or #(6) component_25 (input_25, output_25);
        pn_and #(2) component_26 (input_26, output_26);
        pn_and #(2) component_27 (input_27, output_27);
        pn_or #(6) component_28 (input_28, output_28);
        pn_or #(6) component_29 (input_29, output_29);
        pn_and #(2) component_30 (input_30, output_30);
        pn_and #(2) component_31 (input_31, output_31);
        pn_and #(2) component_32 (input_32, output_32);
        pn_and #(2) component_33 (input_33, output_33);
        pn_and #(2) component_34 (input_34, output_34);
        pn_and #(2) component_35 (input_35, output_35);
        pn_and #(2) component_36 (input_36, output_36);
        pn_and #(2) component_37 (input_37, output_37);
        pn_and #(2) component_38 (input_38, output_38);
        pn_and #(2) component_39 (input_39, output_39);
        pn_and #(2) component_40 (input_40, output_40);
        pn_and #(2) component_41 (input_41, output_41);
        pn_and #(2) component_42 (input_42, output_42);
        pn_and #(2) component_43 (input_43, output_43);
        pn_proposition #(1) component_44 (input_44, output_44);
        pn_proposition #(1) component_45 (input_45, output_45);
        pn_or #(2) component_46 (input_46, output_46);
        pn_proposition #(1) component_47 (input_47, output_47);
        pn_or #(2) component_48 (input_48, output_48);
        pn_proposition #(1) component_49 (input_49, output_49);
        pn_proposition #(1) component_50 (input_50, output_50);
        pn_proposition #(1) component_51 (input_51, output_51);
        pn_or #(2) component_52 (input_52, output_52);
        pn_proposition #(1) component_53 (input_53, output_53);
        pn_or #(2) component_54 (input_54, output_54);
        pn_proposition #(1) component_55 (input_55, output_55);
        pn_proposition #(1) component_56 (input_56, output_56);
        pn_proposition #(1) component_57 (input_57, output_57);
        pn_or #(2) component_58 (input_58, output_58);
        pn_proposition #(1) component_59 (input_59, output_59);
        pn_or #(2) component_60 (input_60, output_60);
        pn_proposition #(1) component_61 (input_61, output_61);
        pn_or #(2) component_62 (input_62, output_62);
        pn_proposition #(1) component_63 (input_63, output_63);
        pn_or #(2) component_64 (input_64, output_64);
        pn_proposition #(1) component_65 (input_65, output_65);
        pn_or #(2) component_66 (input_66, output_66);
        pn_proposition #(1) component_67 (input_67, output_67);
        pn_or #(2) component_68 (input_68, output_68);
        pn_or #(2) component_69 (input_69, output_69);

        [...]

        pn_and #(2) component_88 (input_88, output_88);
        pn_proposition #(1) component_89 (input_89, output_89);
        pn_proposition #(1) component_90 (input_90, output_90);
        pn_and #(2) component_91 (input_91, output_91);
        pn_and #(2) component_92 (input_92, output_92);
        pn_and #(2) component_93 (input_93, output_93);
        pn_and #(2) component_94 (input_94, output_94);
        pn_and #(2) component_95 (input_95, output_95);
        pn_and #(2) component_96 (input_96, output_96);
        pn_or #(9) component_97 (input_97, output_97);
        pn_transition #(1) component_98 (input_98, output_98, protected_step, return_context[0], rst, soft_rst, clk);
        pn_proposition #(1) component_99 (input_99, output_99);
        pn_proposition #(1) component_100 (input_100, output_100);
        pn_and #(2) component_101 (input_101, output_101);
        pn_and #(2) component_102 (input_102, output_102);
        pn_and #(2) component_103 (input_103, output_103);
        pn_and #(2) component_104 (input_104, output_104);
        pn_transition #(1) component_105 (input_105, output_105, protected_step, return_context[1], rst, soft_rst, clk);
        pn_and #(2) component_106 (input_106, output_106);
        pn_and #(2) component_107 (input_107, output_107);
        pn_and #(2) component_108 (input_108, output_108);
        pn_and #(2) component_109 (input_109, output_109);
        pn_and #(2) component_110 (input_110, output_110);
        pn_and #(2) component_111 (input_111, output_111);
        pn_and #(2) component_112 (input_112, output_112);
        pn_and #(2) component_113 (input_113, output_113);
        pn_and #(2) component_114 (input_114, output_114);
        pn_and #(2) component_115 (input_115, output_115);
        pn_proposition #(1) component_116 (input_116, output_116);
        pn_proposition #(1) component_117 (input_117, output_117);
        pn_and #(2) component_118 (input_118, output_118);
        pn_and #(2) component_119 (input_119, output_119);
        pn_and #(2) component_120 (input_120, output_120);
        pn_and #(2) component_121 (input_121, output_121);
        pn_transition #(1) component_122 (input_122, output_122, protected_step, return_context[2], rst, soft_rst, clk);
        pn_proposition #(1) component_123 (input_123, output_123);
        pn_proposition #(1) component_124 (input_124, output_124);
        pn_and #(2) component_125 (input_125, output_125);
        pn_and #(2) component_126 (input_126, output_126);
        pn_and #(2) component_127 (input_127, output_127);
        pn_transition #(1) component_128 (input_128, output_128, protected_step, return_context[3], rst, soft_rst, clk);
        pn_and #(2) component_129 (input_129, output_129);

        [...]

endmodule
```

D Tools used

Tool name	**Usage**
GNU/Linux	Operating systems for the development machine and the HPS
Intel Quartus 17	IDE for the compilation of the FPGA project
g++	C++ cross compiler
gdb, gdbserver	C++ cross-debugger
cmake	Build system
Poco	C++ networking library
pugixml	C++ XML library
Visual Studio Code	C++ and Verilog IDE
GGP project	Reference implementation of the Stanford GGP components
Icarus Verilog	Verilog code simulation
GTK Wave	Visualisation of Verilog code simulation
git	Version control
bash	Agent and debugger launch scrips
python	gdb scripting
ssh	HPS access

Bibliography

[1] Mandy J. W. Tak, Marc Lanctot, and Mark H. M. Winands . Monte Carlo Tree Search variants for simultaneous move games . *IEEE Conference on Computational Intelligence and Games*, 2014.

[2] Geekbench 2. Test of a 48-core Opteron 6174 system. http://browser.geekbench.com/geekbench2/353033.

[3] Geekbench 2. Test of the Cyclone V Hard Processor System. http://browser.geekbench.com/geekbench2/2678890.

[4] Peter Auer. Using Confidence Bounds for Exploitation-Exploration Trade-offs. *Journal of Machine Learning Research*, 3:397–422, 2002.

[5] C. B. Browne, E Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1), 2012.

[6] James Clune. Heuristic evaluation functions for general game playing. In *Proceedings of the 22Nd National Conference on Artificial Intelligence - Volume 2*, AAAI'07, pages 1134–1139. AAAI Press, 2007.

[7] M. Genesereth. *Stanford University General Game Playing Coursera online course*. 2014.

[8] M. Genesereth and Y. Björnsson. *The International General Game Playing Competition*, volume 34, pages 107–111. 2013.

[9] M. Genesereth and M. Thielscher. *General Game Playing*. Morgan & Claypool, 2014.

[10] Guillaume M.J-B. Chaslot, Mark H.M. Winands, and H. Jaap van den Herik. Parallel Monte-Carlo Tree Search. *International Conference on Computers and Games*, pages 60–71, 2008.

[11] M. Gunther and S. Schiffel. *Dresden General Game Playing Server*. 2013.

[12] Kowalski J. Game Description Language for Real-time Games. *Proceedings of the IJCAI-15 Workshop on General Game Playing (GIGA'15)*, 2015.

[13] J. Kowalski and M. Szykuła. Game Description Language Compiler Construction. In *AI 2013: Advances in Artificial Intelligence*, volume 8272 of *LNCS*, pages 234–245. 2013.

[14] B. Pell. *METAGAME in Symmetric Chess-Like Games*. 1992.

[15] J. Pitrat. *A general game playing program*, pages 125–155. 1971.

[16] Stephan Schiffel and Michael Thielscher. M.: Fluxplayer: A successful general game player. In *In: Proceedings of the AAAI National Conference on Artificial Intelligence*, pages 1191–1196. AAAI Press, 2007.

[17] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529:484–503, 2016.

[18] Chiara F. Sironi and Mark H. M. Winands. Optimizing propositional networks. In *Computer Games*, volume 705 of *CCIS*, pages 133–151. 2017.

[19] Cezary Siwek, Jakub Kowalski, Chiara Sironi, and Mark Winands. *Implementing Propositional Networks on FPGA*, pages 133–145. Springer, 12 2018.

[20] M. Thielscher. GDL-II. *Künstliche Intelligenz*, 25, 2011.

[21] M. Thielscher. GDL-III: A Description Language for Epistemic General Game Playing. *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence (IJCAI-17)*, 2011.

[22] Hilmar Finnsson Yngvi Björnsson. Cadiaplayer: A simulation-based general game player. In *IEEE Transactions on Computational Intelligence and AI in Games*, volume 4-15, pages 4–15. 2009.