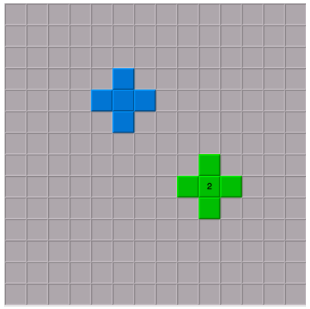
The Liquid Metal Blokus Experiment



Stephen Fink

IBM Research

Liquid Metal Team (IBM T. J. Watson Research Center)

Josh Auerbach

David Bacon

Ioana Baldini

Perry Cheng

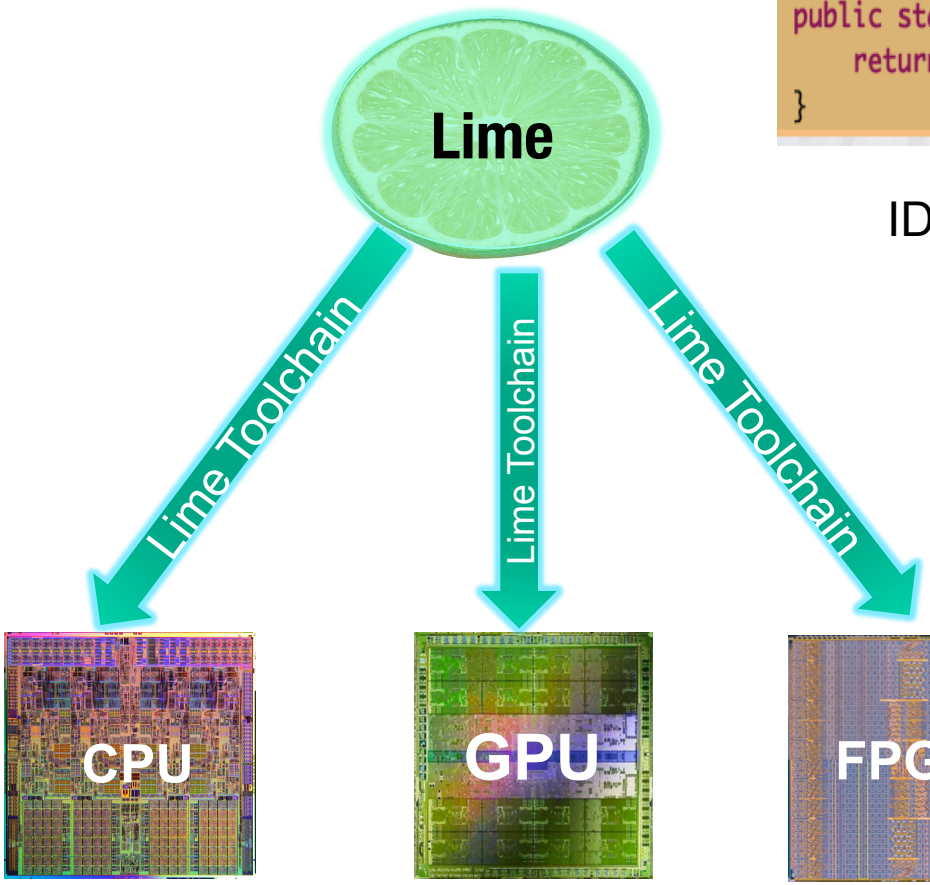
Stephen Fink

Rodric Rabbah

Sunil Shukla

Goal: Make Heterogeneous Platforms Accessible for Mainstream Use

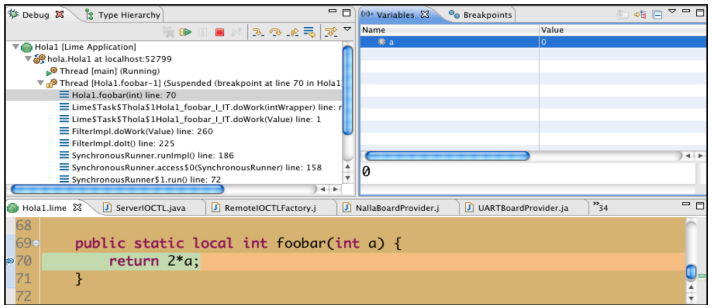
Vision:



High-Level Java-Derived Language

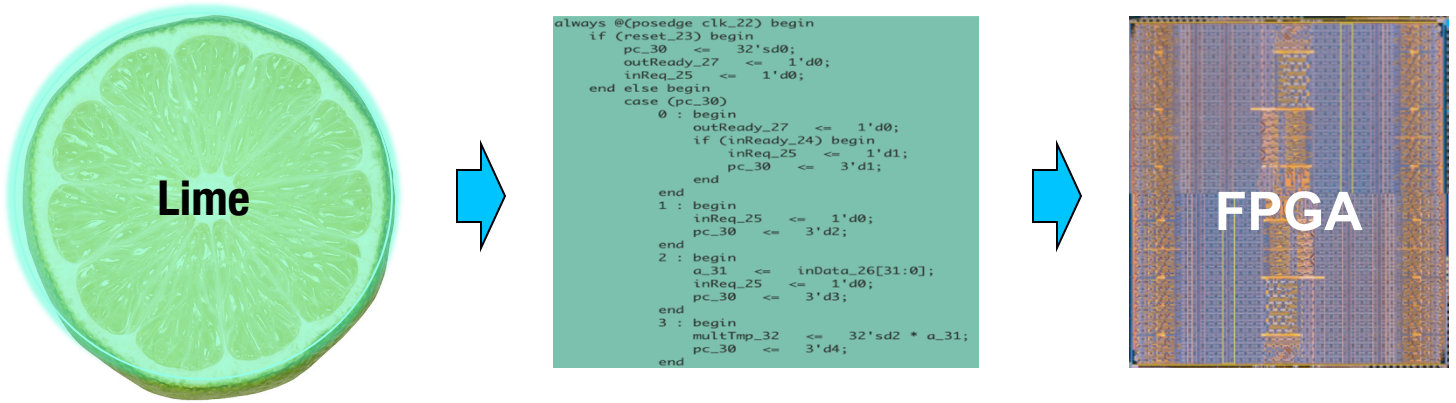
```
public static local int foobar(int a) {
    return 2*a;
}
```

IDE, Compiler, Debugger, Cloud Synthesis



Platform:
Hardware + Runtime

Is High-Level Synthesis (using Lime) viable?



Using a high-level language, can a developer design hardware whose quality matches hardware designed with standard tools?

An Experiment

Blokus Demo

Lime

The Liquid Metal Programming Language

What is Lime?

Lime = Java + Isolation + Abstract Parallelism

Isolation: ability to move computation

- **local** keyword
- Immutable types

Abstract Parallelism: freedom to schedule

- Stream programming model (**task, =>**)
- Data parallel constructs (**@** map, **!** reduce)

The Lime Development Experience

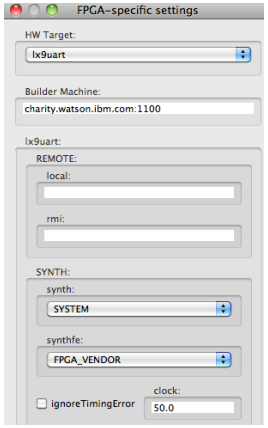
1. Prototype in standard Java

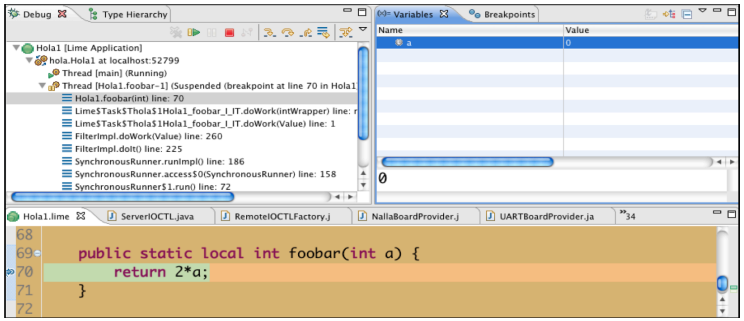
- Lime is a superset of Java
- **Java-like Eclipse IDE (editors, debugger, navigation)**

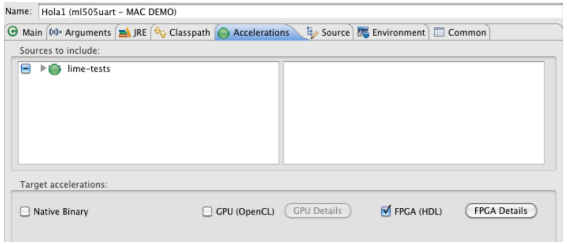
2.Gentle, incremental migration to parallel Lime code

Lime language constructs restrict program to safe parallel structures

- Isolation
- Immutability
- Safe Deterministic Parallelism
- Bounded Space







Implementing Blokus player in Lime

```
public interface Player {

    /**
     * Return some descriptive name for this Player
     */
    public string getName();

    /**
     * Reset the state of the game to g.
     */
    public void reset(Game g);

    /**
     * What is the next move, after the opponent makes move m?
     */
    public Move nextMove(Move m);

}
```

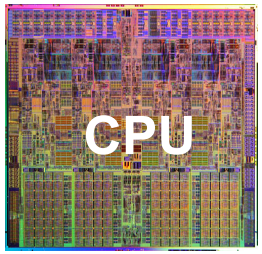
Player is stateful object

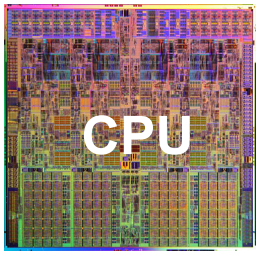
Relocation Brackets **[( )]** indicate tasks that can relocate to device

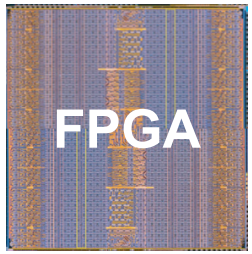
Connect operator **=>** abstracts communication

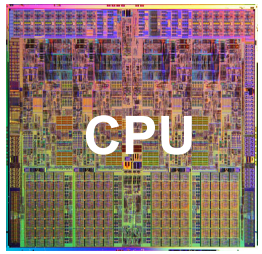
Task graph expression specifies coarse-grain dataflow:

```
source() => ([ task new BlokusPlayer().nextMove ]) =>  sink()
```









Lime for a standalone (headless/no host) deployment

```
([ task receiveMove => task move => task sendMove ])
```

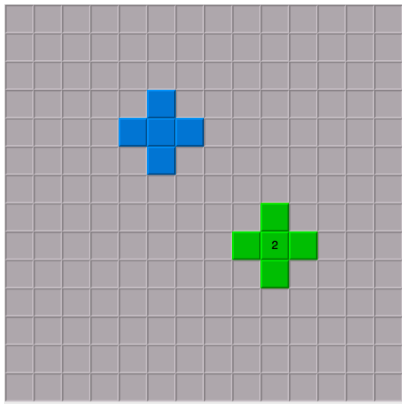
Entire task graph inside relocation brackets

Lime Native Interface

```
import lime.lang.annotations.lni.InputPins;
import lime.lang.annotations.lni.OutputPins;

public class NativeBlokus {

    @InputPins({"opp_move:72", "opp_rdy"})
    @OutputPins({"opp_ack"})
    public static native local byte[[9]] receiveMove();

    @InputPins({"lime_ack"})
    @OutputPins({"lime_move:32", "lime_rdy"})
    public static native local void sendMove(byte[[4]] b);

}
```

Implement “native” methods in VHDL/Verilog/etc.

Blokus AI Algorithm



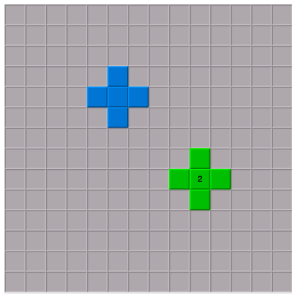
Minimax tree search
- Based on board **evaluation function**

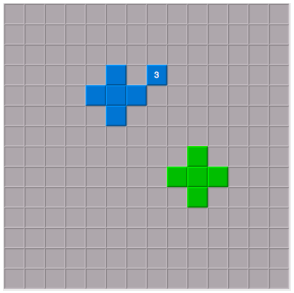
Fixed budget (1 sec response)
- Iteratively deepen tree until budget exhausted

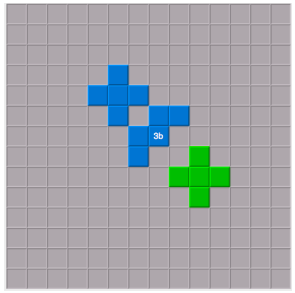
Maximum bounded tree size
- Statically allocated data structures – no dynamic memory allocation

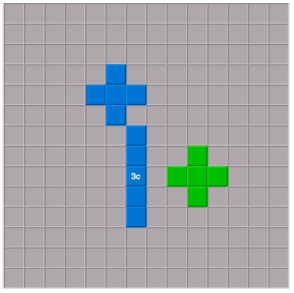
Relatively simple algorithm for software AI players
- Relatively complicated data structures for hardware implementation

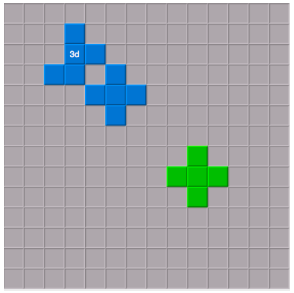
Minimax Step 1: Enumerate all possible moves

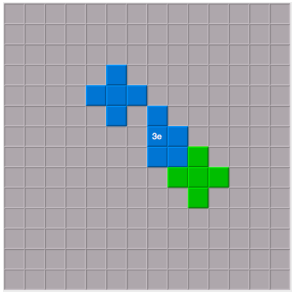






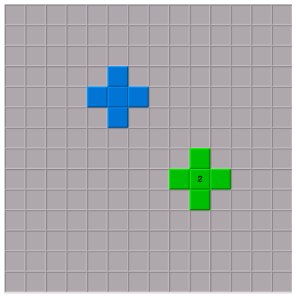


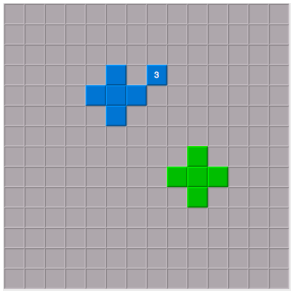




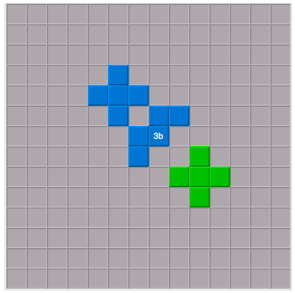
....

Minimax Step 2: Score each move with *evaluation function*

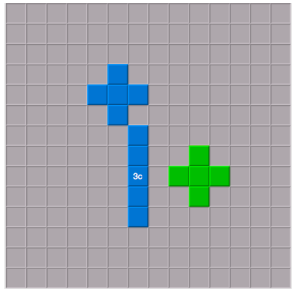




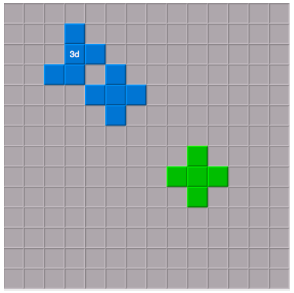
100



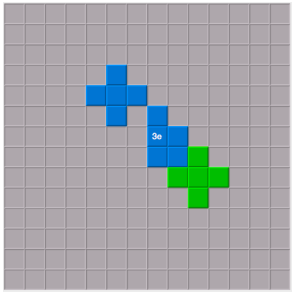
700



600



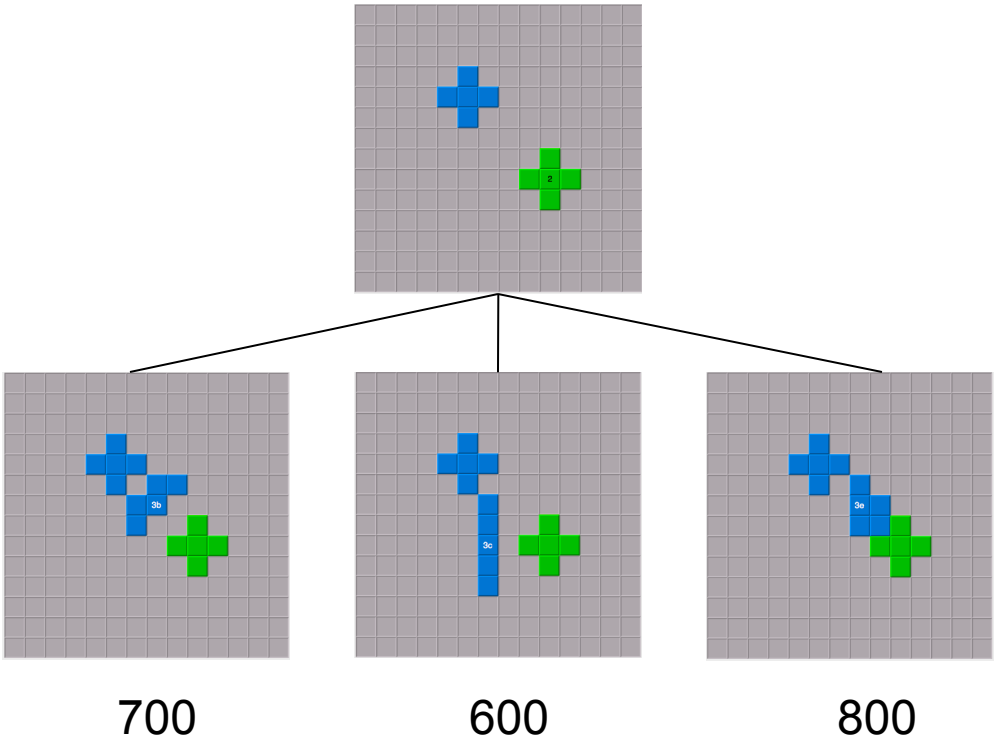
300



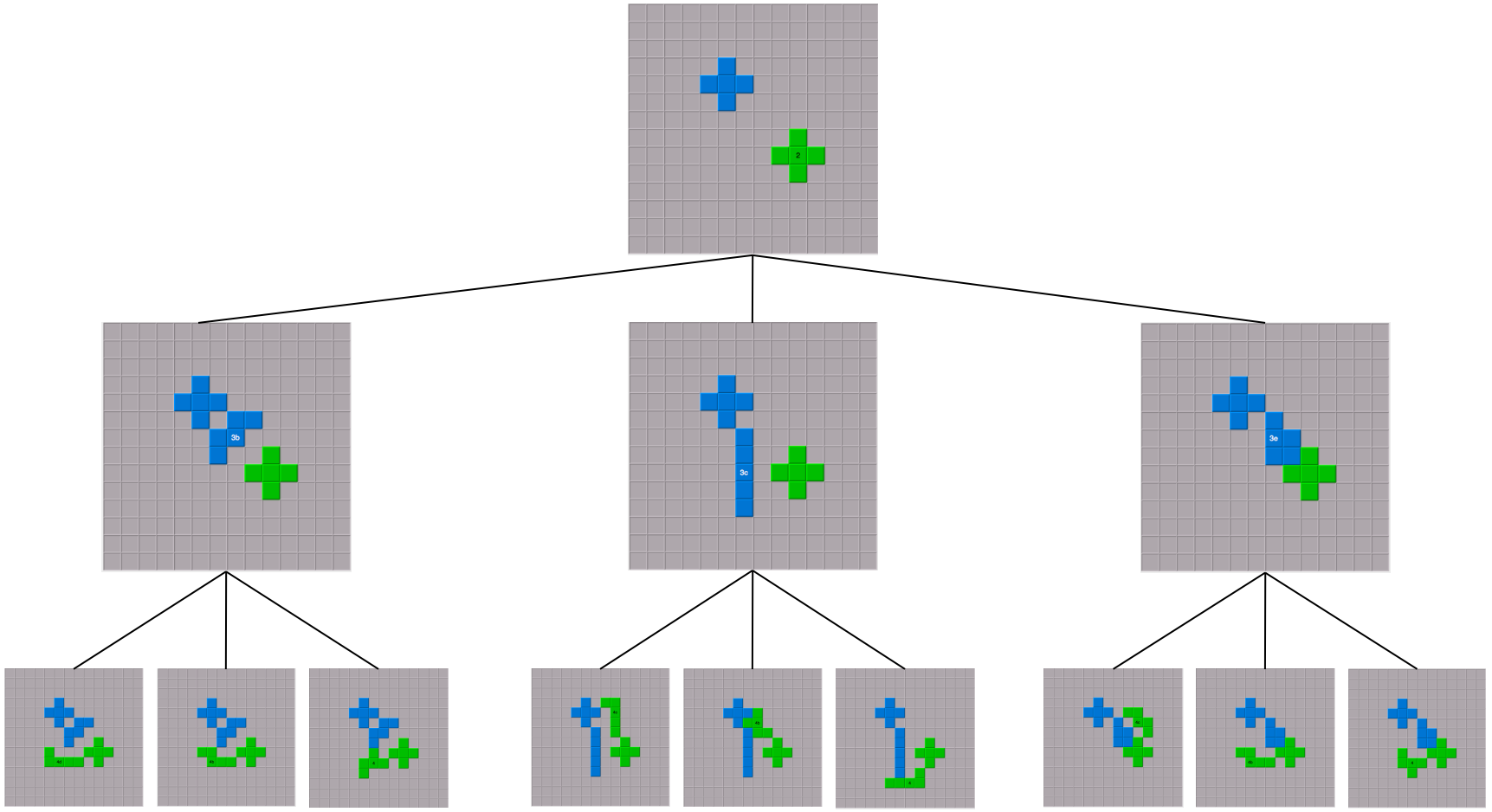
800

....

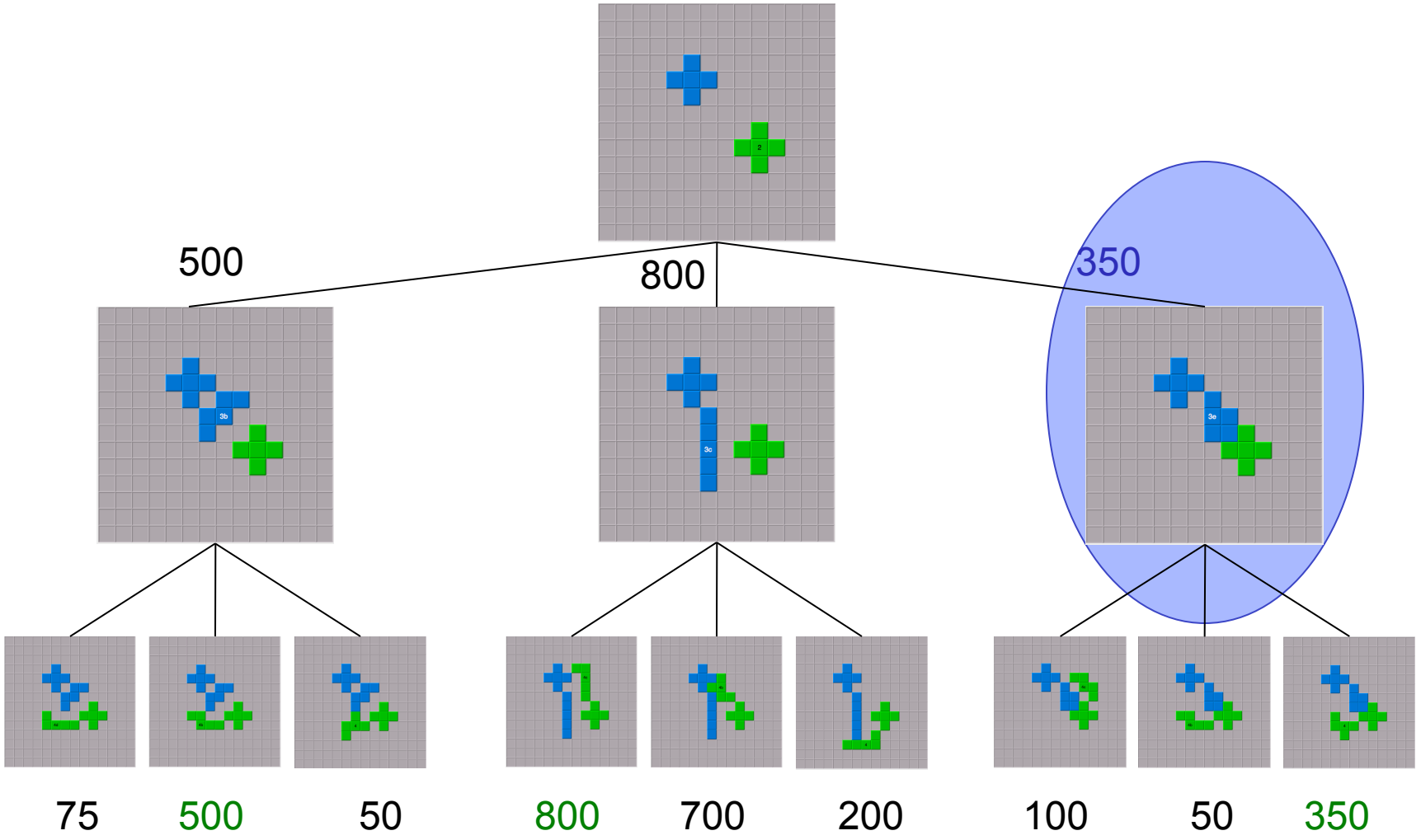
Minimax Step 3: Pick best *k* moves and add to search tree



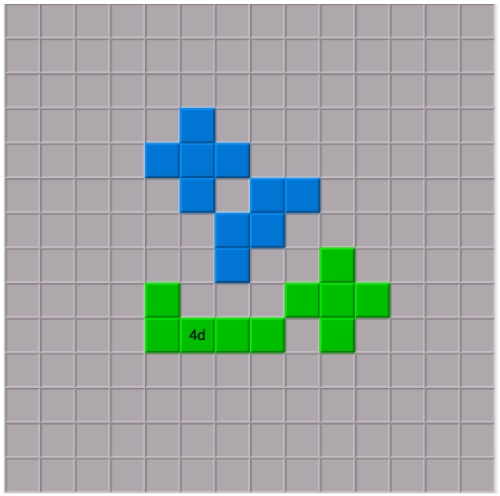
Minimax Step 3: Iterate until budget exhausted

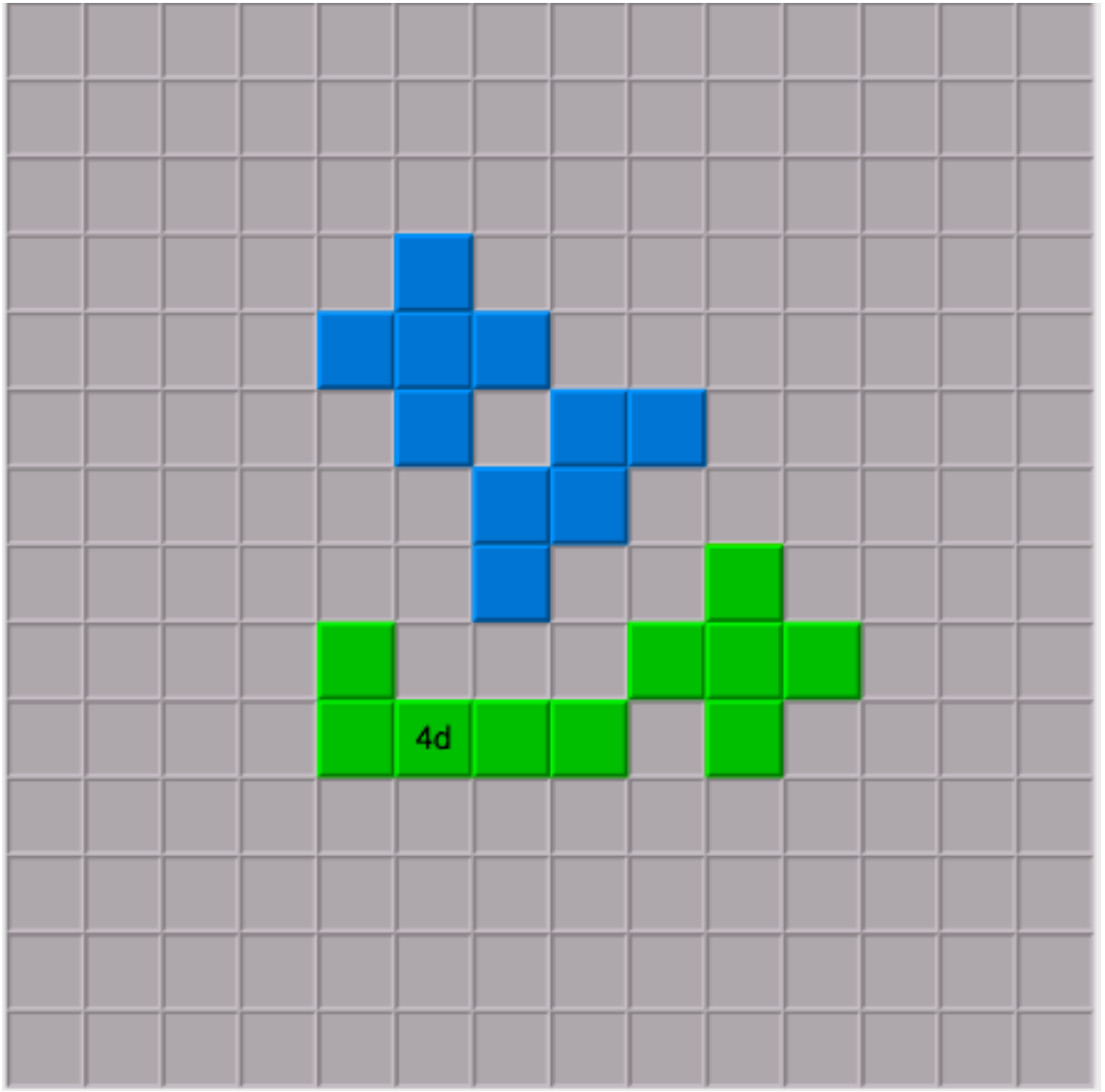


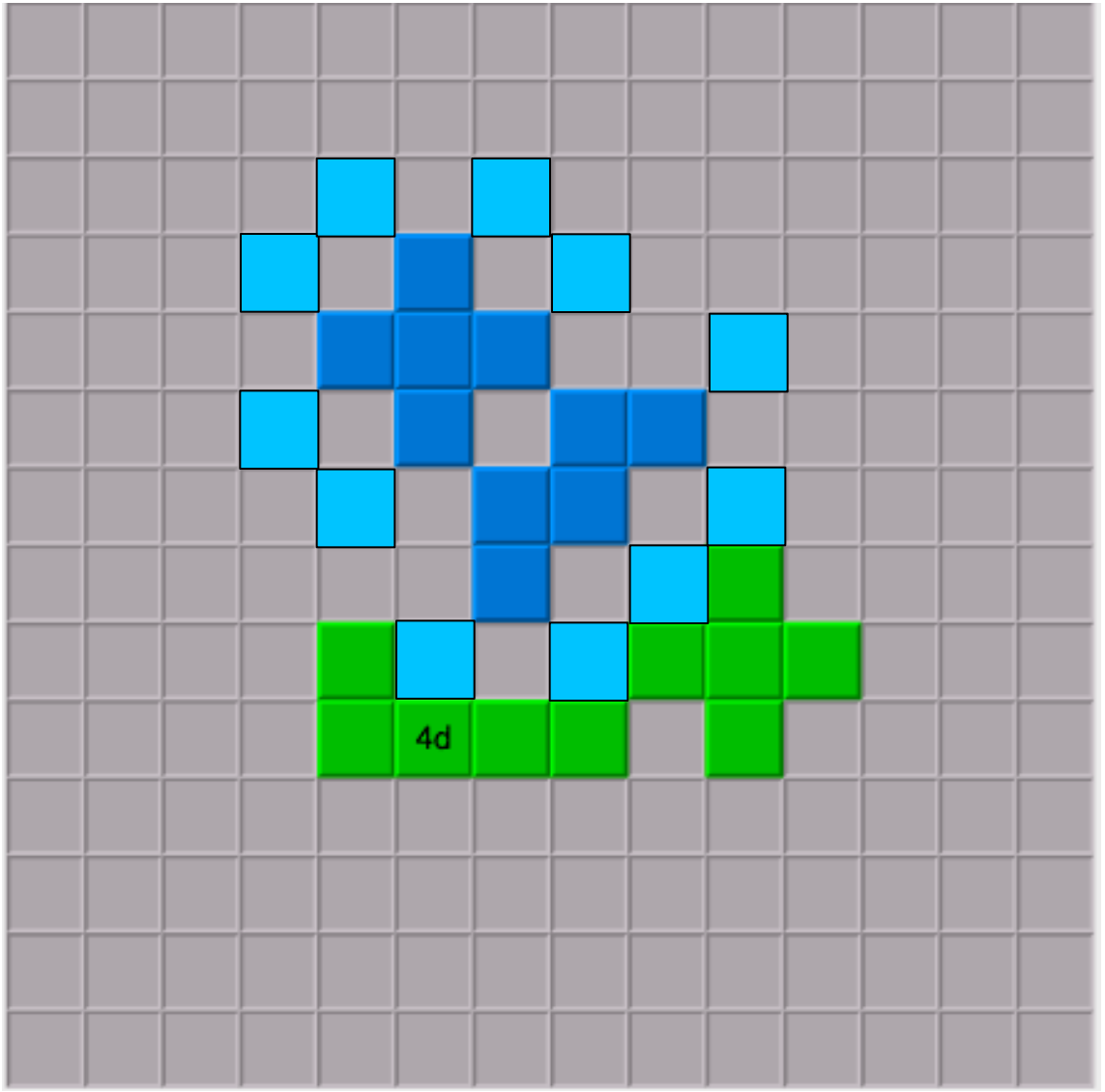
Minimax Final Step: Choose best move via minimax reduction

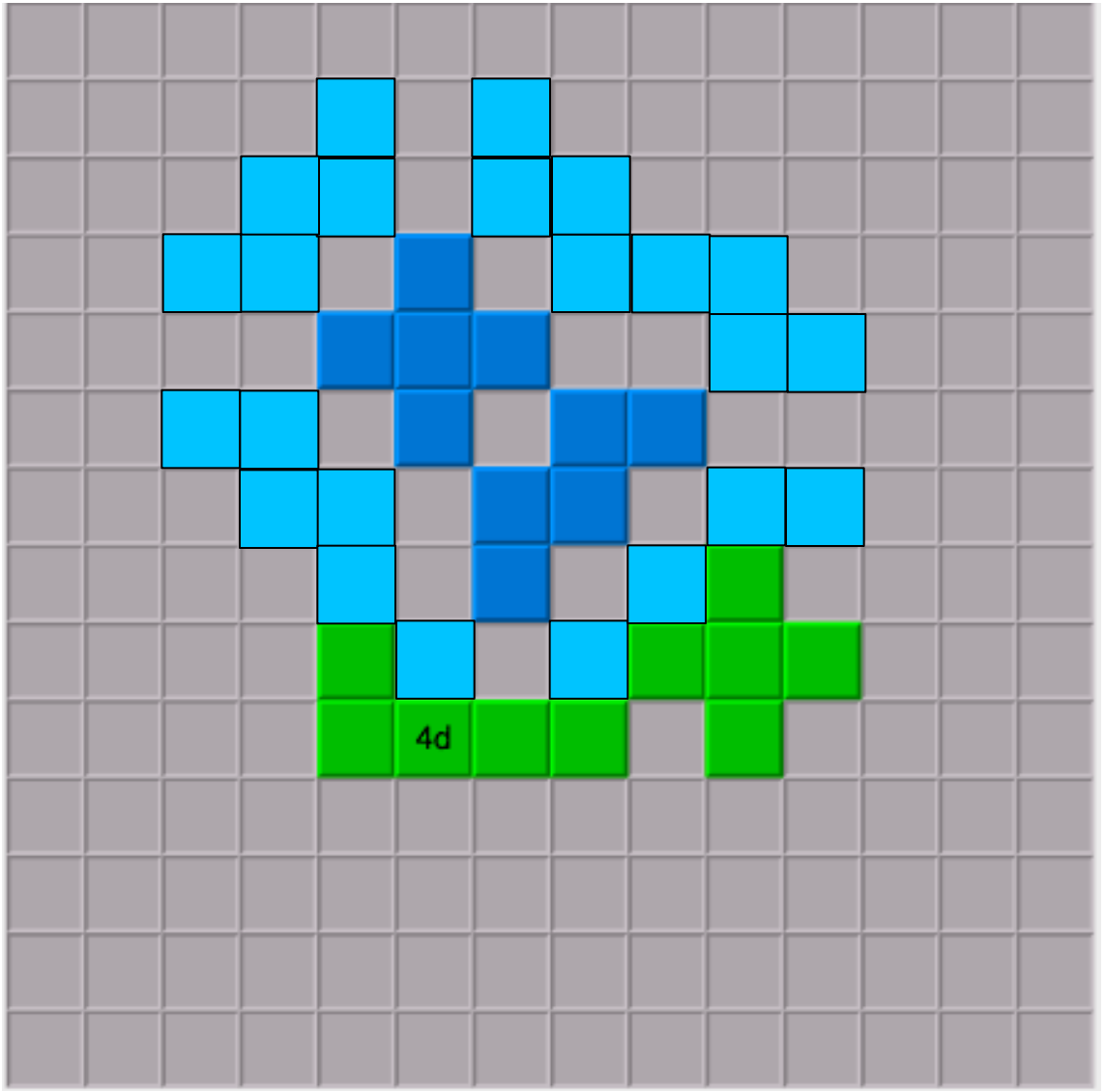


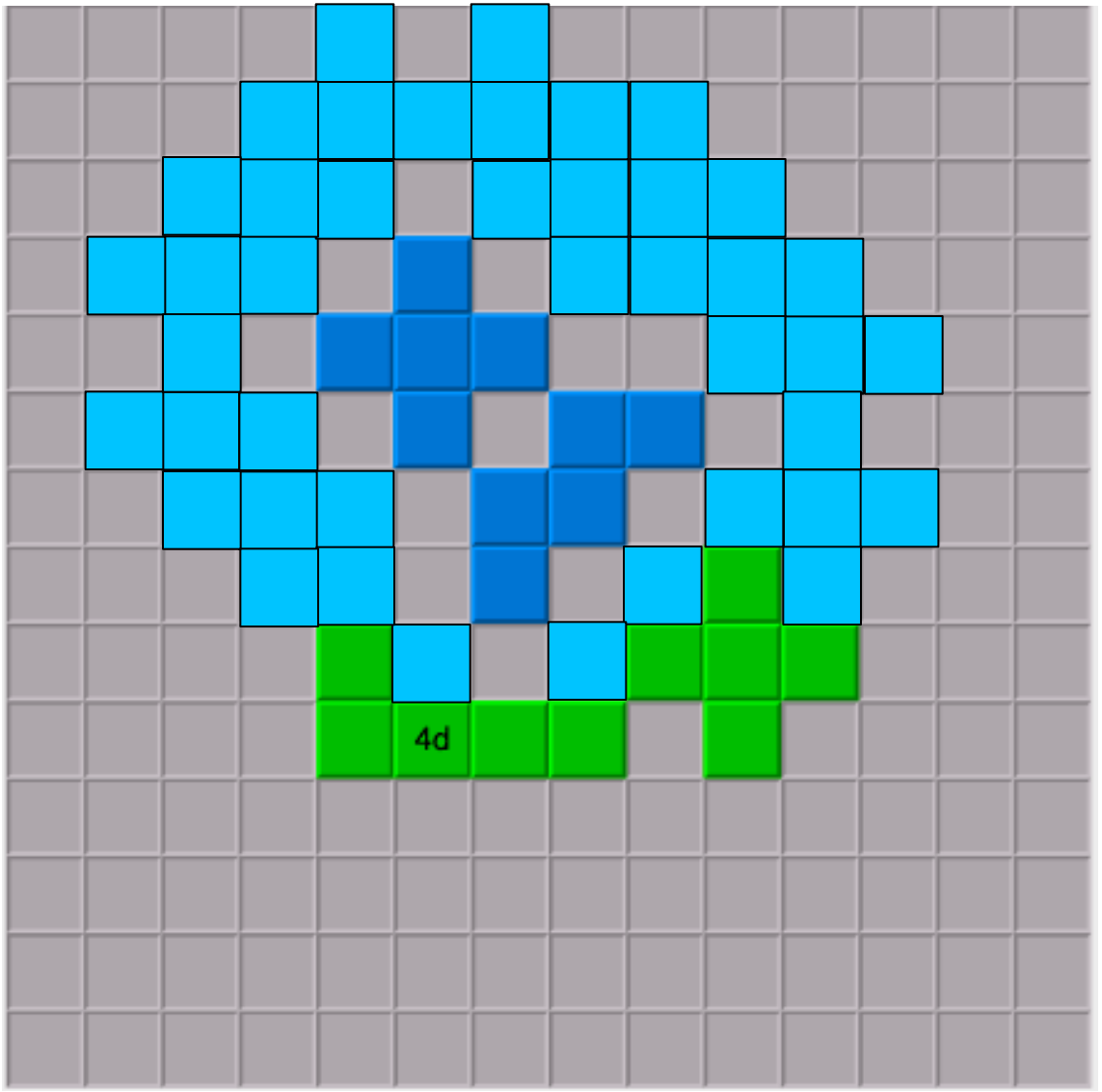
Territory-Based Evaluation Function

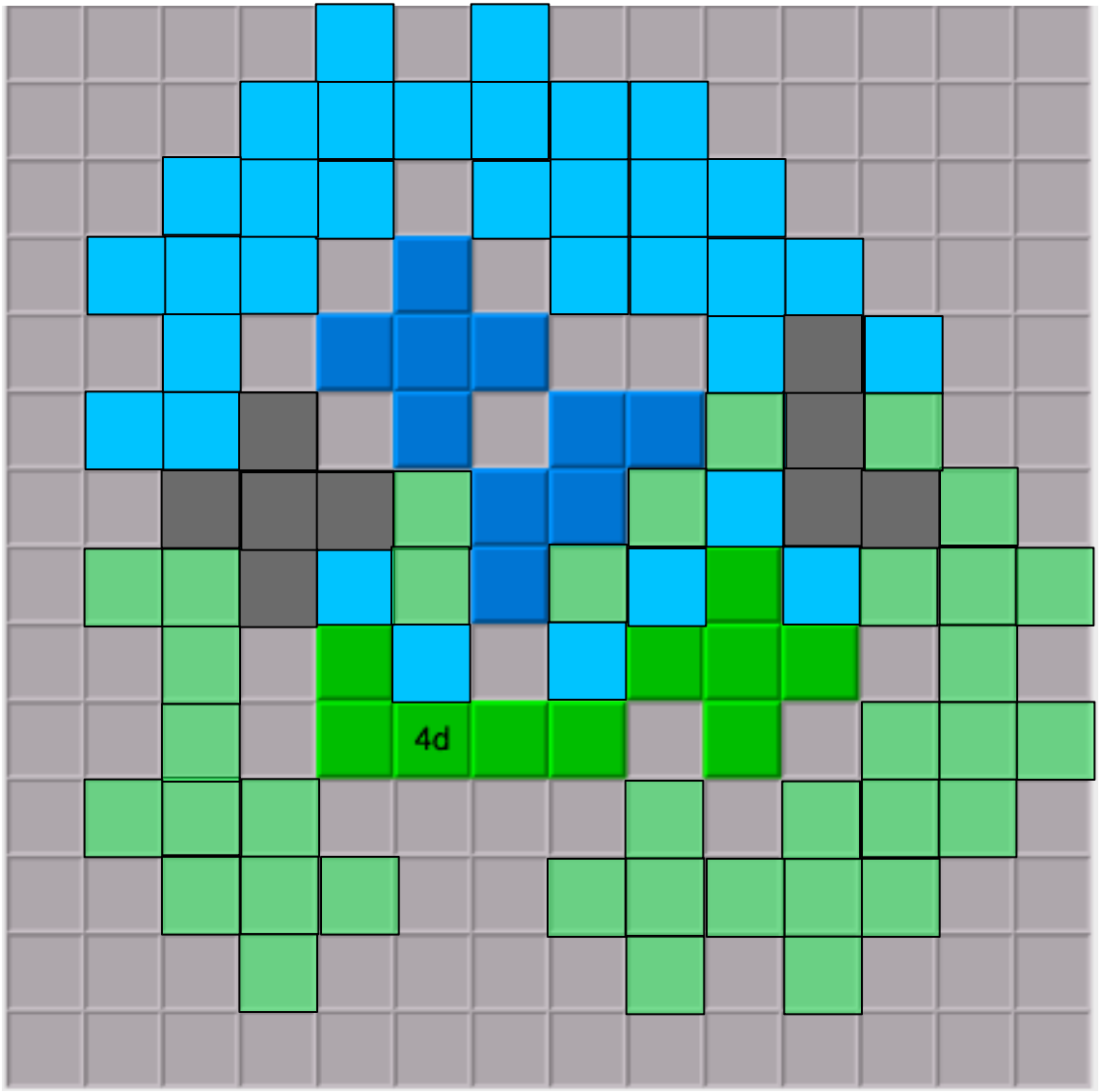
$f($  $)$











Inner loop of evaluation function in Lime

local qualifier guarantees isolation

value classes (programmable primitives)

bounded, immutable arrays

bounded integers

```
public static local Row[[14]] manhattanNeighbors(Row[[14]] g) {

    Row[14] r = new Row[14];
    Row upper = 0;
    Row mid = g[0];
    for (int<14> i){
        Row lower = (i == 13) ? (Row)0n : (Row)g[i+1];

        // manhattan propagation.
        r[i] = (Row)(upper | lower | (mid << 1) | (mid >>> 1));

        // update rows.
        upper = mid;
        mid = lower;
    }

    return new Row[[14]](r);
}
```

Lime type system invariants allow efficient translation inner loop – potentially one cycle

Implementation

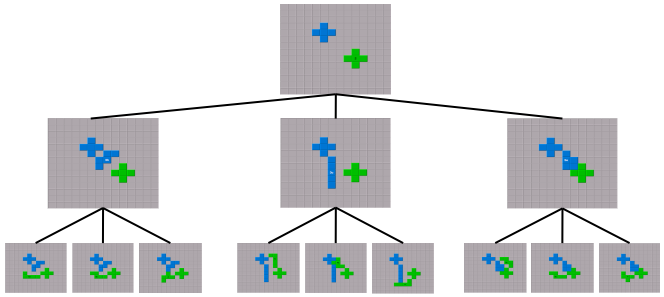
ML505 / Xilinx-Virtex 5 LX50T
7200 slices
48 DSP48E
2160 kB BRAM

Design Stats	
Lime Lines of Code	4231
Lines of Verilog generated	24,657
Lines of hand-written Verilog	186
Frequency	85 MHz
LUTs	15,917 (55.3%)
Flip Flops	11,050 (38.4%)
18kB BRAMs	64 (54.2%)
DSPs	14 (29.2%)

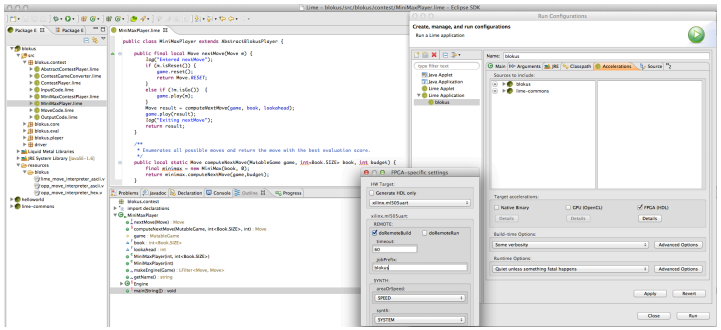
Lessons Learned So Far

AI Search: Ideal domain for high-level synthesis

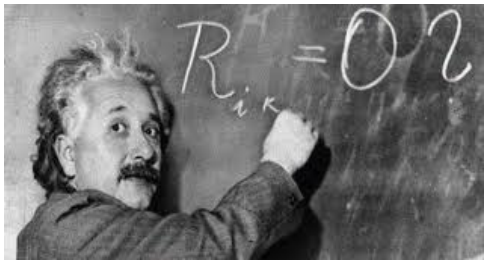
- Algorithmic tuning
- Iterative development cycle
- High computation/communication ratio



Lime development cycle rocks



Software engineer can implement complex hardware in Lime



Using a high-level language, can a developer design hardware whose quality matches hardware designed with standard tools?

Can the Lime team design a Blokus player in Lime that can win the 2013 ICFPT Design Competition?

Tune in early December!