A Clash Course in Solving Sudoku (Functional Pearl)

Gergő Érdi*
Independent
Singapore, Singapore
gergo@erdi.hu

Abstract

Clash is a compiler from Haskell to hardware description. We explore a *Haskell-first* approach to hardware design by building an FPGA Sudoku solver based on a well-known software implementation, showing the step-by-step process of adapting it to hardware. The final code still exhibits the benefits of Haskell's powerful tools for abstraction.

CCS Concepts: • **Hardware → Hardware description languages and compilation**; • **Software and its engineering → Functional languages**.

Keywords: Haskell, Clash, FPGA, RTL, Sudoku, functional programming, functional pearl

ACM Reference Format:
Gergő Érdi. 2025. A Clash Course in Solving Sudoku (Functional Pearl). In *Proceedings of the 18th ACM SIGPLAN International Haskell Symposium (Haskell '25), October 12–18, 2025, Singapore, Singapore.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3759164.3759345

1 Introduction

Digital electronic circuits can be used to build generic computational devices that are then capable of running programs. To solve some problem with such a device, we take some pre-existing hardware, and write software that uses the computational resources of that hardware. The hardware is designed without any reference to the particular problem we're solving.

Alternatively, we can start from scratch, and design and implement an application-specific circuit for our problem. This way, we can fit the hardware design to the problem, achieving better performance and efficiency. However, this is usually regarded as a costly endeavor, both in terms of development effort and in manufacturing cost.

Field-programmable gate arrays (FPGAs) provide a cheap and quick-turnaround way of developing application-specific hardware, since the components of the FPGA can be wired together in a configurable way based on a circuit description that is uploaded electronically. They are, effectively, a chip fab on the desktop.

That takes care of the manufacturing cost and time, but what about development effort? In this paper, we claim that the abstractions afforded by functional programming are useful tools in managing circuit complexity the same way they have been proven useful in software development.

We demonstrate this claim by building a high-performance solver circuit[1] for a well-known combinatorial puzzle: Sudoku. After reviewing the task ahead of us in section 2, we start in earnest in section 3 by applying standard techniques of pure functional programming: we design representations of Sudoku cells and grids that facilitate an efficient and elegant implementation of pruning.

In the second half we turn towards hardware: in sections 4 to 6 we adapt this design to the constraints of finite hardware, and sections 7 and 8 round off the implementation and discuss actual hardware deployment. We conclude with comments towards possible improvements and the experience of using Clash in sections 9 and 10.

2 Preliminaries

2.1 Sudoku

Sudoku is a combinatorial puzzle game originally played on a nine-by-nine grid partitioned into three-by-three boxes, each three by three in size. Given nine distinct symbols, the player is given a partial assignment of symbols to grid cells (a *problem*), and their task is to to extend it into a complete assignment that is *consistent*, i.e. the nine cells of each *group* (a *row*, *column*, or three-by-three *box*) contain the full set of all nine symbols.

To talk about the complexity of Sudoku, we need to generalize it by some notion of *size*. Since each row, column, and box needs to have the same area for the consistency constraint to make sense, once the box size is fixed, the full grid size also becomes fixed. This makes the original Sudoku the $(3, 3)$-Sudoku. Figure 2 shows the $(3, 4)$ and the $(2, 6)$-Sudoku structures, illustrating that while both have a grid

*Views expressed in this paper do not necessarily represent the views of current or previous employers of the author.

Haskell '25, Singapore, Singapore

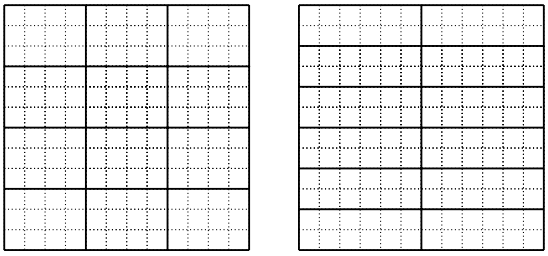
ACM ISBN 979-8-4007-2147-2/25/10
https://doi.org/10.1145/3759164.3759345

[1]The complete source code of the solver is available at https://github.com/gergoerdi/clash-sudoku

	2		9		8			
8	7				1		5	4
5		6	4				1	
		2					9	5
9	4					8		
	8				4	5		3
1	3		2				8	6
			3		7		2	

4	2	1	9	5	8	6	3	7
8	7	3	6	2	1	9	5	4
5	9	6	4	7	3	2	1	8
3	1	2	8	4	6	7	9	5
7	6	8	5	1	9	3	4	2
9	4	5	7	3	2	8	6	1
2	8	9	1	6	4	5	7	3
1	3	7	2	9	5	4	8	6
6	5	4	3	8	7	1	2	9

Figure 1. A standard 9×9 $(3, 3)$-Sudoku board with an example problem and one of its solutions



(a) $(3, 4)$-Sudoku board **(b)** $(2, 6)$-Sudoku board

Figure 2. Two different structures of Sudoku boards with size 12×12

size of 12×12, and each group contains 12 cells for both sizes, the underlying constraint system is completely different.

For the rest of this paper, we will denote $N = nm$ as the size of an (n, m)-Sudoku where the exact shape doesn't matter.

Under this generalization, the problem of Sudoku solving is NP-complete [15]. It stands to reason, then, to base our design on backtracking and a collection of heuristics.

2.2 FPGA design basics

An FPGA consists of a large number of uniform elements called logic blocks, a network of signal wires between these blocks, and switching boxes that route these signals according to the configuration. Inside each logic block, we find a fixed-size lookup table with a small handful of input and output lines, connected to a register. There are also a small number of specialized elements like RAM, clock signal generators, or complex digital signal processor blocks; these connect to the same routing network as the regular components.

FPGA designs are usually described at the register transfer level (RTL), and then automated tools refine that design into a complete configuration for a given device, filling each lookup table and switching each router. In the RTL model, the design is described as a network of arbitrary-width registers, synchronized via shared clock lines. Connections between the registers are so-called *combinational circuits*: clockless, and thus stateless circuits whose output at any given moment is a pure function of their current input.

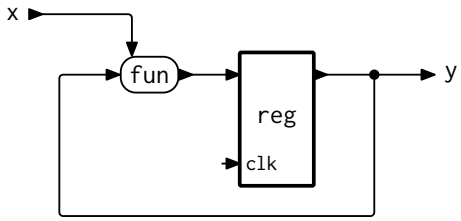


Figure 3. Register transfer-level component

2.3 Clash: Haskell for Hardware

Clash [1, 6] is a Haskell compiler based on GHC [14] that emits register-transfer level hardware descriptions instead of executable programs. The circuits represented by these descriptions can then be realized on an FPGA or in the form of a bespoke integrated circuit.

Clash compiles pure Haskell functions into combinational circuits, for example the following function describes a circuit consisting of one 8-bit adder and one 8-bit `AND` gate:

```
combine :: Unsigned 8 → Unsigned 8 → Unsigned 8
combine x y = (x + y) .&. x
```

By their very nature, combinational circuits are stateless and correspond to the behaviour of RTL blocks within a single clock cycle. We will use only these in section 3, but by the end of the section we run into physical limitations that necessitate using smaller circuits over several clock cycles.

To describe stateful circuits, we need the ability to insert registers. Unlike so-called *high-level synthesis* languages, Clash never changes the synchronous structure of the circuit: registers are never inferred, only inserted explicitly by the programmer using the `register` combinator.

```
type Signal :: Domain → Type → Type
register :: a → Signal dom a → Signal dom a
```

We build RTL networks by putting combinational circuits between `registers`, using the applicative functor interface [9]:

```
instance Applicative (Signal dom)
```

The `Signal` type constructor is tagged at the type level with a clock domain. This ensures that the applicative combinators cannot be used to accidentally cross clock domains, which can lead to nondeterministically inconsistent circuit behaviour. In our circuit, we will only use a single clock domain.

Signals commute with product types via the `Bundle` typeclass and its `bundle` and `unbundle` methods. For example, for 2-tuples we have:

```
bundle   @(a, b) ::
  (Signal dom a, Signal dom b) → Signal dom (a, b)
unbundle @(a, b) ::
  Signal dom (a, b) → (Signal dom a, Signal dom b)
```

These operations are merely structural, i.e. they correspond to just different ways of grouping the same signal lines. As such, they don't yield any circuitry.

With this toolkit, the elementary RTL from Figure 3 can be described in Clash in the following way, using the ⟨$⟩ and ⟨*⟩ combinators of the `Applicative` typeclass, with recursion guarded by `register` to implement the feedback (`y0` is the initial value of the register upon reset):

```
y :: Signal dom (Unsigned 5)
y = register y0 (fun ⟨$⟩ x ⟨*⟩ y)
```

The type of the contents of the `y` signal is specified as `Unsigned 5`. Since we are describing fully customized circuits, we can use arbitrary sizes that would be exotic in a software context. Clash provides types like `Unsigned`, `Signed`, or `BitVector` that are indexed by type-level natural numbers [17] describing their exact width. Similarly, the `Index n` type can be used to represent a natural number between 0 and $n-1$ (known as `Fin n` in some dependently typed languages) using $\lceil \log_2 n \rceil$ bits.

3 A pure solver

In this section we first adapt Bird's well-known, pure Haskell solver [2] to a hardware-friendly representation of Sudoku boards and then analyze its viability as a hardware circuit.

We start with the following basic structure of the solver:

```
type Sudoku n m = Grid n m (Cell n m)

expand :: Sudoku n m → [Sudoku n m]   -- defined later

sudoku :: Sudoku n m → Maybe (Sudoku n m)
sudoku grid
    | blocked  = empty
    | complete = pure grid
    | changed  = sudoku pruned
    | otherwise =
      asum [sudoku grid' | grid' ← expand grid]
  where
    -- blocked, complete, changed, and
    -- pruned defined later in this section
```

The only difference compared to Bird's solver are the `Grid` and `Cell` types, which are indexed by the size of the Sudoku board.

3.1 Board representation

As is usual with functional program design, well-chosen data types can lead us to good implementation. We define representations of cells and grids motivated by two design criteria: efficient mapping to hardware structures, and effortless support of the operations needed for Sudoku solving.

Suggested by Figure 2, we should try avoiding confusion in the geometry, and represent our (n, m)-Sudoku grid as a matrix of boxes, themselves matrices. The Clash standard library provides sized vectors, which we can use as the building block of our matrix type storing a vector of rows.

```
newtype Mtx n m a = FromRows (Vec n (Vec m a))
  deriving (Functor, Applicative, Foldable)
    via Compose (Vec n) (Vec m)
  deriving (Semigroup, Monoid)
    via Ap (Mtx n m) a
```

We use the *Deriving Via* mechanism [4] to generate a large number of typeclass instances with little effort; these instances will be made useful by our monoid-based interface to pruning.

We similarly define `Grid` as a matrix of matrices, with the same instances:

```
newtype Grid n m a = Grid (Mtx n m (Mtx m n a))
  deriving ...
```

To implement pruning, each individual `Cell` tracks the remaining candidates for its value. Since we have $N = nm$ distinct symbols, we can represent each cell of an (n, m)-Sudoku board as an N-bit vector, with the i^{th} bit set to `1` if i is still a possible value for the given cell.

```
newtype Cell n m = Cell{cellBits :: BitVector (n * m)}

canBe :: Cell n m → Index (n * m) → Bool
Cell c `canBe` i = c ! i == 1
```

The all-ones cell represents a wildcard value which could still be anything, and the all-zeroes cell is the result of a conflict in the remaining constraints:

```
wild, conflicted :: Cell n m
wild       = Cell oneBits
conflicted = Cell zeroBits
```

We also need a way to convert given values into our cell format. `Data.Bits.bit` constructs a `BitVector` where the only set bit is the one with the specified index:

```
given :: Index (n * m) → Cell n m
given = Cell ∘ bit ∘ fromIntegral
```

With the `Foldable` instance of `Grid`, we can fill in the first missing local definition of our `sudoku` function. Using the terminology from [2], the board is `void` if any of the cells have no more possible candidate values:

```
sudoku grid = ...
  where
    blocked = void ∨ not safe
    void    = any (== conflicted) grid
```

3.2 Testing consistency

A given collection of cells is `consistent` if the already known values (i.e. cells which only have a single candidate value remaining) don't overlap. The whole board is `safe` if all groups (rows, columns and boxes) are `consistent`:

```
sudoku grid = …
  where
    safe = allGroups consistent grid

consistent ::
  (Foldable f, Functor f) ⇒ f (Cell n m) → Bool
consistent = not ∘ bitsOverlap ∘
    fmap λc → if single c then cellBits c else 0
```

To check if a given `Cell` value is fully known, we can implement `single` by counting the number of set bits in its bit-mask using `Data.Bits.popCount`:

```
single :: Cell n m → Bool
single cell = popCount (cellBits cell) == 1
```

Since each cell is stored as a bit-mask, an overlap can be detected by accumulating the bits in a given group, checking every bit-mask against the previous ones:

```
bitsOverlap ::
  (Foldable f) ⇒ f (BitVector n) → Bool
bitsOverlap = (/= 0) ∘ snd ∘ foldr step (0, 0)
  where
    step x (acc, overlaps) =
      (acc .|. x, overlaps .|. (acc .&. x))
```

As for `safe` itself, unlike `void`, it is not a cell-by-cell proprety; instead, it is tied to the constraint structure of the Sudoku board. We need predicates that can view a whole group at a time:

```
allGroups :: (Group n m a → Bool) → Grid n m a → Bool
```

Each `Group` contains N cells, and a complete set of groups (all rows, all columns, or all boxes) consists of N of them:

```
type Group  n m a = Vec (n * m) a
type Groups n m a = Vec (n * m) (Group n m a)
```

These make up the fundamental constraint structure of a Sudoku puzzle, so we need a convenient way of accessing them. As a first stab, we can try using normal functions to represent possible groupings. We will return to this choice of representation in subsection 3.4, but it will do for now:

```
type Grouping n m = ∀ a. Grid n m a → Groups n m a

cols, rows, boxs :: Grouping n m   -- Defined later
```

This gives us a very simple implementation of `allGroups` as a conjunction over all groupings:

```
allGroups :: (Group n m a → Bool) → Grid n m a → Bool
allGroups p grid =
    allBy rows ∧ allBy cols ∧ allBy boxs
  where
    allBy grouping = all p (grouping grid)
```

3.3 Pruning

Pruning is the only heuristic we are implementing, and it is the fundamental operation of our solver. If we know that a given cell's value can only be e.g. 4, it follows that no other cell in the same row, column, or box can be 4, and so we can remove 4 as a possible value from all these neighbouring cells.

To implement this, the plan is to convert known cell values (i.e. those that have only a `single` possible value left) into *masks* on possible values, then combine the masks in a given group, and disallow the combined values from not-yet-known cell values:

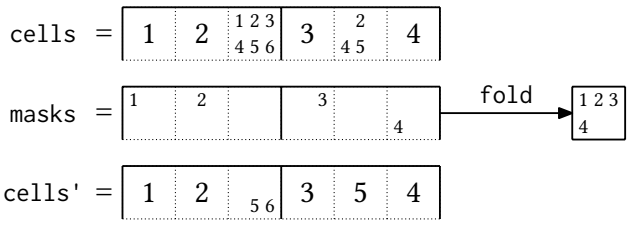
```
sudoku = …
  where
    pruned          = apply ⟨$⟩ groupMasks ⟨*⟩ grid
    changed         = pruned /= grid
    groupMasks      = foldGroups (maskOf ⟨$⟩ grid)
    maskOf     cell = cellMask (single cell) cell
    apply mask cell = act mask  (single cell) cell
```

This means that masks start life with the bit-mask corresponding to single values of cells (`cellMask`), are then combined via bitwise inclusive-or exposed as a monoidal operation, and then `act` on cells via bitwise-and-not, both of which are very efficient to do in hardware.

```
newtype Mask n m = Mask{maskBits :: BitVector (n * m)}

instance Semigroup (BitMask n m) where
  Mask m1 ◇ Mask m2 = Mask (m1 .|. m2)

instance Monoid (BitMask n m) where
  mempty = Mask zeroBits

cellMask :: Bool → Cell n m → Mask n m
cellMask isSingle (Cell c) =
  if isSingle then Mask c else mempty

act :: Mask n m → Bool → Cell n m → Cell n m
act (Mask m) isSingle (Cell c)
  | isSingle  = Cell c
  | otherwise = Cell (c .&. complement m)
```

We pass `isSingle` as a parameter to `cellMask` and `act` instead of computing it on our own from the `Cell` argument because we will later want to avoid computing it repeatedly for the same cell.

To illustrate how all this machinery works, suppose we're solving a $(2, 3)$-Sudoku and come across a row where four of the cells are already known and one cell only has three possible values:



Thus, `cells'` contains the result of propagating the constraints, restricting possible values on the third cell, and solving the fifth one completely. A further propagation step is needed to remove 5 from the third cell's candidates, thus solving it to 6.

Note that `act` is a monoidal action: it commutes with ◊, i.e. `act m1 ∘ act m2 ≡ act (m1 ◊ m2)`. Consequently, we can combine masks from all neighbours of a given cell using bitwise disjunction and apply it just once.

3.4 Mapping and folding by rows, columns, and boxes

To combine the masks of each group per group, we need to implement `foldGroups` which replaces every cell with the monoidal sum of the three groups it belongs to. For a grid of `Masks`, it corresponds exactly to the propagation we need in our `sudoku` function.

```
foldGroups :: (Monoid a) ⇒ Grid n m a → Grid n m a
```

However, we need to revisit our definition of `Grouping`. If a `Grouping` is just a function from grid to groups, this would only allow us to *extract* the groups instead of *mapping* them in-place. Let's say we extract the rows, columns, and boxes of a grid, and calculate some rowwise, columnwise and boxwise results. We then have no way of lining up these three results to do some further processing at each cell as an intersection of its three groups.

Instead, we will represent each `Grouping` as a *bidirectional* mapping of the `Grid` to its `Groups`:

```
type Grouping n m = ∀ a. Grid n m a ↔ Groups n m a
```

These bidirectional mappings are pairs of functions in both directions forming an isomorphism:

```
data a ↔ b = Iso {embed :: a → b, project :: b → a}

instance Category (↔) where ...
```

Isomorphisms lift through functors:

```
imap :: (Functor f) ⇒ a ↔ b → f a ↔ f b
imap iso = Iso (fmap (embed iso)) (fmap (project iso))
```

We use this infrastructure to define the following trivial building blocks:

```
iconcat       :: Vec n (Vec m a) ↔ Vec (n * m) a
itranspose    :: Vec n (Vec m a) ↔ Vec m (Vec n a)
matrix        :: Mtx n m a       ↔ Vec n (Vec m a)
transposeGrid :: Grid n m a      ↔ Grid m n a
grid          :: Grid n m a      ↔ Mtx n m (Mtx m n a)
```

We now have everything to define our groupings as bidirectional mappings, composed with the `Category` typeclass's ∘ operator. The simplest is `boxs` because `Grid`'s inner matrices already directly correspond to the boxes, we just need to enumerate them in row-major order:

```
rowMajorOrder :: Mtx n m a ↔ Vec (n * m) a
rowMajorOrder = iconcat ∘ matrix

boxs :: Grouping n m
boxs = rowMajorOrder ∘ imap rowMajorOrder ∘ grid
```

`rows` is a bit trickier, because we have to reshape across the outer matrix's structure. Thankfully, since the dimensions are tracked in the types, we can use e.g. GHC's typed holes feature while writing to arrive at the correct definition:

```
rows :: Grouping n m
rows =
  imap iconcat ∘ iconcat ∘ imap itranspose ∘
  matrix ∘ imap matrix ∘ grid
```

And finally for `cols` we can take a shortcut via `rows` using transposition:

```
cols :: Grouping n m
cols = rows ∘ transposeGrid
```

The change to `allGroups` to use this new representation is minimal: we just need to use `embed` in `allBy` to retrieve the forward-mapping component of the given grouping.

```
allGroups :: (Group n m a → Bool) → Grid n m a → Bool
allGroups p grid =
    allBy rows ∧ allBy cols ∧ allBy boxs
  where
    allBy grouping = all p (embed grouping grid)
```

We can also implement a row-major order `Traversable` instance [3] for `Grid` using `rows`:

```
instance Traversable (Grid n m) where
  traverse f =
    fmap (project rows) ∘ traverse (traverse f) ∘
    embed rows
```

The implementation of `foldGroups` is similar to `allGroups`. The crucial difference is that the groupwise results are mapped back by `foldBy` into the full `Grid` before combining them cell by cell into the final result:

```
foldGroups :: (Monoid a) ⇒ Grid n m a → Grid n m a
foldGroups = foldBy rows ◊ foldBy cols ◊ foldBy boxs
  where
    foldBy grouping =
      project grouping   ∘
      fmap (repeat ∘ fold) ∘
      embed grouping
```

Figure 4 shows an example of how `foldGroups` on a $(2, 3)$ grid implements constraint propagation. All-zero masks are shown as empty cells.

We take a moment here to make note of the fact that all the constituent parts of our groupings, and thus the groupings themselves, are merely shuffling around the elements between various vector-represented containers. Thus, in a hardware implementation, a function like `embed rows` applied on a `Mtx n m (Signal dom a)` is "free of charge" in the same sense as `bundle` and `unbundle`, i.e. it is purely a relabeling of signals, without any further combinational circuitry, and thus without any use of logic blocks.

3.5 Generating guesses

The job of the `expand` function is to generate guesses for cells whose value we don't yet know for sure. There are multiple heuristics we could employ here, but instead we're going to simply pick the last cell that offers multiple remaining choices.

We can achieve this by traversing our grid using `mapAccumR`, where the state records whether we have already encountered a cell that can be split into different choices. We can then use `sequenceA` to expand our grid-of-list-of-possible-cells into a list-of-possible-grids:

```
expand :: Sudoku n m → [Sudoku n m]
expand grid = sequenceA ∘ snd ∘ mapAccumR guess False
  where
    guess done cell
      | not done, cs@(_ : _) ← choices cell = (True, cs)
      | otherwise = (done, [cell])
```

To generate all possible `choices` for a single cell, we can check all possible `Index (n * m)` values to see if they are still possible assignments for the given cell, and then generate unique `given` choices for them:

```
choices cell =
  [given i | i ← [minBound .. maxBound], cell `canBe` i]
```

With this definition, we have finished our implementation of the software solver.

3.6 Why the rest of this paper?

Recall the type of `sudoku`:

```
sudoku :: Sudoku n m → Maybe (Sudoku n m)
```

If Clash compiles Haskell code to hardware circuit descriptions, can't we just take this function, slap some IO around it, and be on our way? The remaining word count of this paper spoils that the answer is no, but let's examine in detail why. There are two kinds of problems with our current software solver from a hardware perspective:

Unboundedness. A hardware circuit is made of physical components wired together, and so it has a fixed, finite size. The software implementation we have presented has two parts that are unbounded: the list returned by `expand`, and the implicit stack of the recursion in `sudoku`.

Luckily, both of these can be made finite using upper bounds. Since `expand` only guesses one cell at a time, and each cell can only have at most N candidate values, the list returned by `choices`, and consequently `expand`, can be at most N long. Similarly, since each `expand` result contains one more single cell, the stack depth never needs to grow beyond the number of cells, i.e. N^2.

Efficiency. It shouldn't surprise us that we can, at least in principle, design a finite-sized circuit to solve fixed-sized instantiations of a problem in NP. However, the size of the circuit we get from the naïve application of the above bounds is far beyond what is feasible on real hardware.

Remembering that each cell is an N wide bit-mask, and each board configuration has N^2 cells, the N-ary, N^2 deep tree of boards contains $N \times N^2 \times N^{N^2}$ bits. Before even implementing any of the logic that creates this tree, even for the common $(3, 3)$-Sudoku this is more than 10^{80} signal lines. This also would make pruning worthless, since its only effect is dynamically choosing not to use some parts of this enormous circuit.

Our hardware solution solves these problems by decreasing the branching factor from N to just 2 (a first guess and a continuation), and then using temporal multiplexing ("trading time for space") to apply the same small circuit to explore different points of the state space, limiting breadth (number of parts) and depth (longest signal path) by breaking down our function into a sequence of steps over multiple clock cycles:

1. Prune by propagating all constraints coming from single-valued cells. If this results in all cells having a single value, we have found a solution and we're done.
2. If any of the cells have no more possible values left, or the single-valued cells of a group overlap, we have reached a contradiction. Pop from the stack, replacing the current grid, and start again.
   If the stack underflows, it means we have run out of possible guesses without finding a solution, and so the original problem has no solutions and we're done.
3. Otherwise, make a guess by splitting one cell into a single value and all other of its currently allowed values. This results in two grids, both of which only differ from the original grid (and each other) in that one cell. Replace the current grid with the first one, and push the second one onto an external stack. Since we know N^2 is an upper bound on the required stack depth, this

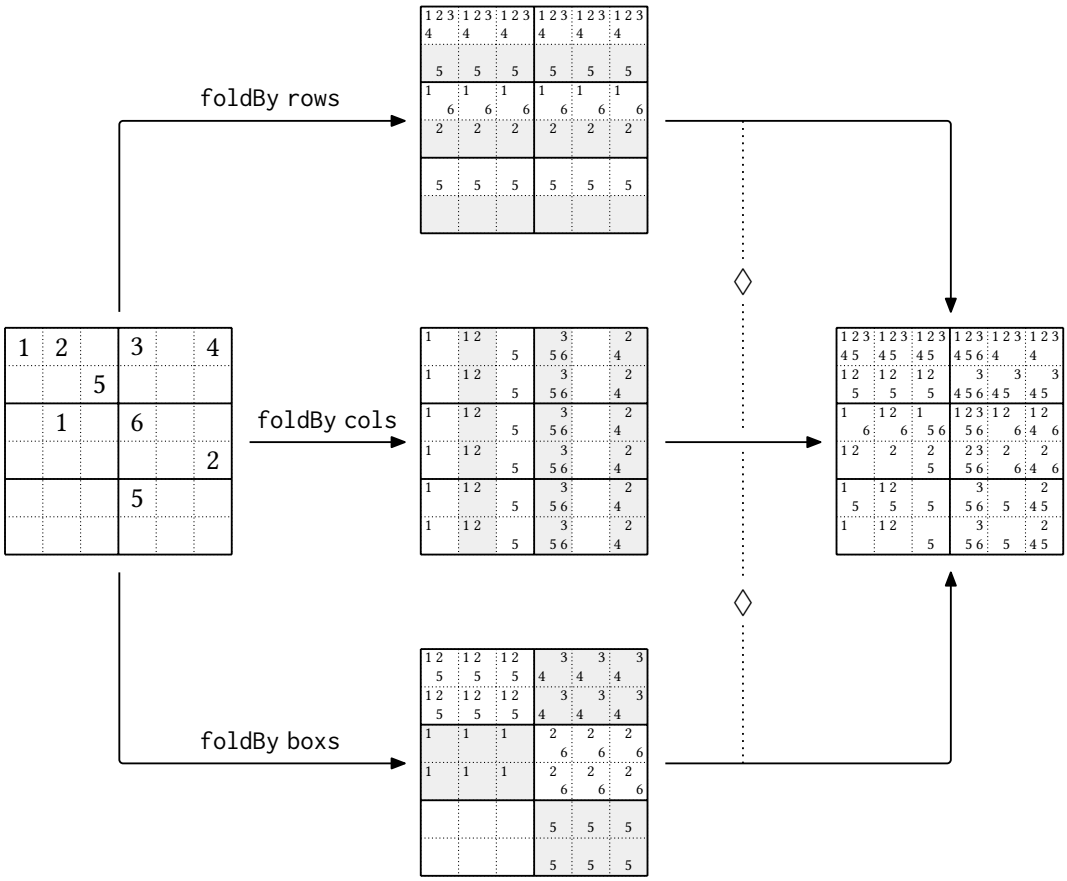


Figure 4. Example of `foldGroups` on the `Masks` of a $(2, 3)$-Sudoku

operation can never overflow if we use a stack at least N^2 deep.

4 Two-way branching

We first tackle the seemingly unbounded result size of `choices`. We can split a cell of possible values into two guesses: a cell with a single bit set, and a cell containing all other remaining possibilities. It doesn't matter which single set bit we return as the first guess, but we have a very hardware-efficient way of computing the least significant set bit [13, 16], thanks to the two's-complement `Num` semantics of `BitVector`:

```
leastSetBit :: BitVector n → BitVector n
leastSetBit x = x .&. negate x
```

Now we can also compute the `rest` of the available cell choices by masking out the `least` set bit:

```
splitCell :: Cell n m → (Cell n m, Cell n m)
splitCell (Cell c) = (Cell least, Cell rest)
  where
    least = leastSetBit c
    rest  = c .&. complement least
```

We can use `splitCell` to write a version of `expand` that returns a pair of grids instead of a whole list of them, but it can also be used as an implementation of `single`, since removing the only possibility results in a `conflicted` cell:

```
single :: Cell n m → Bool
single = (== conflicted) ∘ snd ∘ splitCell
```

This can be implemented efficiently in hardware because underneath all the newtype wrappers, it's just comparing an N-bit value with all zeroes. However, it is perhaps less obvious why going via `splitCell` is not wasteful, compared to a more direct implementation.

In software, we are used to the idea of efficiency arising from avoiding unncesessary computations, so we might think that we only want to compute `splitCell` for the one cell that `expand` selects as the first expandable one. But in combinational circuits, branching corresponds to a multiplexer selecting the output of one of the possible sub-circuits (see Figure 5), so in terms of footprint, number of parts, circuit complexity, or any similar metric, we don't win anything by pushing computations under branches.

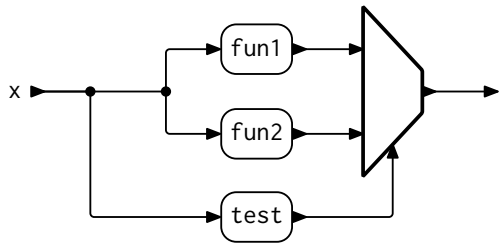


Figure 5. Combinational circuit of λx → **if** test x **then** fun1 x **else** fun2 x

Concretely, if we implement the new version of guess like the following, in hardware there are no wasted parts just from computing nextGuess even for non-single cells.

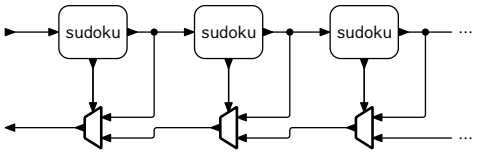
```
expand ::
  Sudoku n m → (Grid n m Bool, Sudoku n m, Sudoku n m)
expand = funzip3 ∘ snd ∘ mapAccumR guess False
  where
    funzip3 xyzs =
      ((λ(x, _, _) → x) ⟨$⟩ xyzs
      ,(λ(_, y, _) → y) ⟨$⟩ xyzs
      ,(λ(_, _, z) → z) ⟨$⟩ xyzs)
    guess done cell
        | not done ∧ not single =
            (True, (single, firstGuess, nextGuess))
        | otherwise =
            (done, (single, cell, cell))
      where
        (firstGuess, nextGuess) = splitCell cell
        single = nextGuess == conflicted
```

In fact, since calculating and applying masks during pruning also heavily depends on single, we can save parts elsewhere in sudoku by reusing the single information returned by expand:

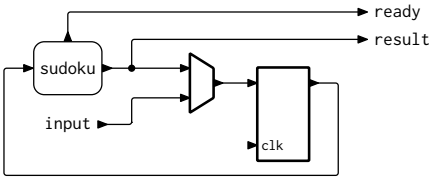
```
sudoku grid | ...
            | otherwise = sudoku guess ⟠ sudoku cont
  where
    (singles, guess, cont) = expand grid
    pruned = act ⟨$⟩ groupMasks ⟨*⟩ singles ⟨*⟩ grid
    groupMasks =
      foldGroups (cellMask ⟨$⟩ singles ⟨*⟩ grid)
```

5 Opening up the recursion

We have seen that a purely combinational Sudoku solver circuit would be enormous. The root of the problem is that if we have a single static circuit, it will necessarily contain as many copies of the circuit implementing the main iteration as there are points in our state space. What we want instead is to use one circuit on the initial Sudoku board's representation in one cycle, then reuse that same circuit in the next cycle on the result of the previous cycle (Figure 6).



(a) Many copies of the sudoku circuit



(b) One copy of sudoku with feedback

Figure 6. Using one copy of sudoku over multiple clock cycles

To achieve that, we will have to go beyond pure functions and start truly using the RTL model. As a first step down that road, let's open up sudoku's recursion by factoring out a solve function that does one round of solving:

```
data Step n m = Blocked
              | Complete
              | Progress (Sudoku n m)
              | Stuck    (Sudoku n m) (Sudoku n m)

solve :: Sudoku n m → Step n m
solve grid | blocked  = Blocked
           | complete = Complete
           | changed  = Progress pruned
           | otherwise = Stuck guess cont
  where
    -- All other local definitions the same as before
```

By circumscribing solve this way, we are making the decision on how much our eventual circuit will do per clock cycle. In particular, since solve includes a definition of pruned, it means our circuit will propagate all immediate constraints in a single clock cycle.

To recover our pure solver, we can pair up solve with a simple driver that turns Stuck into a branch in Maybe:

```
sudoku :: Sudoku n m → Maybe (Sudoku n m)
sudoku grid = case solve grid of
  Blocked          → empty
  Complete         → pure grid
  Progress pruned  → sudoku pruned
  Stuck guess cont → sudoku guess ⟠ sudoku cont
```

However, we get something much more enlightening if we make the use of a *stack* explicit. Note that we have recursive occurrences of sudoku both in a tail call position (in the Progress branch) and in a non-tail call position (in the Stuck branch). Tail calls correspond to a step that *replaces* the current grid before the next step. We can see this more clearly if we switch to continuation-passing style for the failure continuation [7]:

```
type Cont n m = Maybe (Sudoku n m)

sudoku :: Sudoku n m → Maybe (Sudoku n m)
sudoku grid = go grid empty
  where
    go :: Sudoku n m → Cont n m → Maybe (Sudoku n m)
    go grid k = case solve grid of
      Blocked          → k
      Complete         → pure grid
      Progress pruned  → go pruned k
      Stuck guess cont → go guess $ go cont k
```

We can see that k accumulates a growing list of cont grids in its thunk, which are then consumed in the Blocked branch. We defunctionalize this by using an explicit stack [18]. Our state thus becomes a pair of the current grid and the stack. For this software model, we can use a simple list for the stack representation:

```
type Stack n m = [Sudoku n m]
type Cont n m = Stack n m

sudoku :: Sudoku n m → Maybe (Sudoku n m)
sudoku grid = go grid [ ]
  where
    go :: Sudoku n m → Cont n m → Maybe (Sudoku n m)
    go grid st = case solve grid of
      Blocked
        | (grid' : st') ← st → go grid' st'
        | otherwise          → empty
      Complete               → pure grid
      Progress pruned        → go pruned st
      Stuck guess cont       → go guess (cont : st)
```

As the final transformation, we make the state transitions explicit, by splitting off the function that calculates a single step:

```
data Transition k r = Continue k | Done r
data Result = Solved | Unsolvable

sudoku :: Sudoku n m → Maybe (Sudoku n m)
sudoku grid = go grid [ ]
  where
    go :: Sudoku n m → Cont n m → Maybe (Sudoku n m)
    go grid st = case step grid st of
      Done Solved          → pure grid
      Done Unsolvable      → empty
      Continue (grid', st') → go grid' st'

step ::
  Sudoku n m → Stack n m →
  Transition (Sudoku n m, Stack n m) Result
step grid st = case solve grid of
  Blocked
    | (grid' : st') ← st → Continue (grid', st')
    | otherwise          → Done Unsolvable
  Complete         → Done Solved
  Progress pruned  → Continue (pruned, st)
  Stuck guess cont → Continue (guess, cont : st)
```

Let's spell out explicitly what we have done so far. Starting with a pure, recursive function, we have split it into two parts: the first one, solve, is non-recursive, implements the bulk of the desired functionality, and is reasonably sized for synthesis. The second one, sudoku, drives the first one by way of an explicit stack. We are not going to compile this second function as-is into hardware; instead, its role is to keep our specification fully executable, and to guide us when developing the actual controller circuit.

We will keep solve as it is for the rest of this paper. All the nice high-level constructs in it, such as using foldGroups to propagate our monoidal action-represented constraints across the isomorphisms of rows, columns and boxes, are directly turned into hardware by Clash.

6 Backtracking with a hardware stack

For the hardware version, we implement the stack as a piece of RAM combined with a register holding the stack pointer. FPGAs contain blocks of on-chip RAM, which can be used with synchronous single-cycle access. This means that in every clock cycle, the read value output corresponds to the address input that was supplied in the previous clock cycle.

6.1 RAM

First of all, we need a convenient way of accessing RAM. Clash provides an interface to block RAM in the form of the blockRamU function (the U stands for "uninitialized at power-up"):

```
blockRamU :: (Enum addr) ⇒
  SNat n →                          -- Number of elements
  Signal dom a →                    -- Read address
  Signal dom (Maybe (a, d)) →       -- Write addr. and data
  Signal dom a
```

We adapt this interface to our needs in two ways.

The first thing to notice is that blockRamU consumes an address signal and returns a read data signal that both always contain some value[2]. For our stack implementation, we want to think of reading from memory as an "action", i.e.

[2]The read value is ⊥ when reading from a not-yet written-to address

something that happens only in select circumstances: when we want to pop from the stack. This is better modelled if both the address input and the read data output are `Maybe` values.

Moreover, for our stack, in any given single cycle, we are either reading or writing, so at most one of the two addresses will be set. We represent this restriction with a custom data type for memory commands:

```
data MemCmd n d = Write (Index n) d | Read (Index n)

ram ::
  Signal dom (Maybe (MemCmd n d)) → Signal dom (Maybe d)
ram cmd = enableDelayed (isJust ⟨$⟩ rdAddr) rd
  where
    (rdAddr, wr) = unbundle (memLines ⟨$⟩ cmd)
    rd = blockRamU (SNat @n) (fromMaybe ⊥ ⟨$⟩ rdAddr) wr
```

In the implementation of `ram`, we can convert from a memory command into the input lines of `blockRamU`:

```
memLines  :: Maybe (MemCmd n d) →
             (Maybe (Index n), Maybe (Index n, d))
memLines Nothing                = (Nothing, Nothing)
memLines (Just (Read addr))     = (Just addr, Nothing)
memLines (Just (Write addr x)) = (Nothing, Just (addr, x))
```

The `Maybe` gating of the read output is implemented with `enable`:

```
enable :: (Applicative f) ⇒ f Bool → f a → f (Maybe a)
enable = liftA2 λen x → if en then Just x else Nothing
```

Because of the one-cycle read delay of synchronous RAM, a `Just` value in the read output corresponds to a `Just` value in the address line *in the previous cycle*. We use an extra `register` to delay the enable signal by one cycle:

```
enableDelayed ::
  Signal dom Bool → Signal dom a → Signal dom (Maybe a)
enableDelayed en = enable (register False en)
```

Note that since `blockRamU` is a primitive corresponding to some actual stateful circuitry, this is our first true `Signal` function. As we round out our hardware solver implementation, we will use `Signal`'s `Applicative` instance to compose `ram` and similar signal functions with lifted pure functions and each other.

For our Sudoku solver specifically, we know that we need a stack of depth N^2 to support any possible backtracking. For memory word size, Clash allows us to just use `Sudoku n m` as the type of the memory contents, and automatically calculate that it needs a width of N^4 bit.

```
type StackDepth n m = n * m * m * n
type StackPtr n m = Index  (StackDepth n m)
type StackCmd n m = MemCmd (StackDepth n m) (Sudoku n m)
```

6.2 `step` with external stack access

We write a function that "turns the crank" of `solve`, i.e. that implements a single iteration of our previous pure `step` function by externalizing its stack access. Given a current grid and stack pointer, it produces not only the new grid and stack pointer (if we are not done yet), but also a memory command to be sent to the stack. Conversely, when the stack outputs a value, this popped grid is picked up to replace the current grid.

```
step   ::
  Maybe (Sudoku n m) → Sudoku n m → StackPtr n m →
  Transition
    (Sudoku n m, StackPtr n m, Maybe (StackCmd n m))
    Result
step stackRd grid sp = case solve grid of
    _ | Just popped ← stackRd →
      Continue (popped, sp, Nothing)
    Complete → Done Solved
    Blocked
      | sp == 0 → Done Unsolvable
      | otherwise →
        Continue (grid, sp - 1, Just (Read (sp - 1)))
    Progress pruned → Continue (pruned, sp, Nothing)
    Stuck guess cont →
      Continue (guess, sp + 1, Just (Write sp cont))
```

Compared to the software model `step` in section 5, the big difference here is that when we are `Blocked`, we don't have immediate access to the top of the stack. Instead, we emit a `Read` command to RAM and consume the memory output in the next cycle.

6.3 Cell-by-cell loading and retrieval

We want to wire up `step` to read from and write to the stack, but there is one problem: we could store the current grid and the current stack pointer in registers and update them from one cycle to the next on `Continue` transitions, but how do we load in the initial problem description?

The `Traversable` instance of `Grid` is defined via `rows`, i.e. in row-major order, to provide an easy way of implementing cell-by-cell writing and reading of `Grid`s in the natural textual (left-to-right, top-to-bottom) ordering. Without it, we would need to collect all N^2 elements in some temporary storage before we can turn them wholesale into a complete new `Grid n m`. By implementing a shift-in operation from the bottom right, we can start with any `Grid n m` (e.g. `pure wild`), and gradually load N^2 elements into it to replace its contents incrementally:

```
shiftIn :: a → Grid n m a → Grid n m a
shiftIn new =
  snd ∘ mapAccumR (λnew old → (old, new)) new
```

We implement a simple streaming interface using `shiftIn` to fill the `Sudoku n m` grid cell by cell. During this loading process cells that are in different groups in the final grid can temporary become neighbours; we avoid corrupting the grid before it even finishes loading by adding an extra parameter that enables the normal pruning `step`. This parameter is set to `True` by the controller only after all N^2 cells have been loaded in.

```
loadStep ::
  Maybe (Cell n m) → Bool →
  Maybe (Sudoku n m) → Sudoku n m → StackPtr n m →
  Transition
    (Sudoku n m, StackPtr n m, Maybe (StackCmd n m))
    Result
loadStep cellIn enable stackRd grid sp
    | Just newCell ← cellIn =
        Continue (shiftIn newCell grid, 0, Nothing)
    | not enable = Continue (grid, sp, Nothing)
    | otherwise = step stackRd grid sp
```

By resetting the stack pointer when shifting in cells we ensure that we always start solving with an empty stack, without needing an extra explicit reset state between problems.

For output, we can tap into the topmost-leftmost cell and repeatedly shift in some dummy element when this output is consumed to get out the elements in the natural textual ordering. The type of the `head` function on sized vectors ensures that this is valid only on non-zero-sized grids.

```
type Nonempty n m = 1 ≤ n * m * m * n

headGrid :: (Nonempty n m) ⇒ Grid n m a → a
headGrid = head ∘ embed (iconcat ∘ rows)
```

6.4 Putting it together

The complete `solver` takes in an input stream of cells and produces a signal containing the result once it becomes available. It also exposes the grid's first cell, which can be used to implement streaming output after the result becomes `Just Solved`.

```
solver ::
  (NonEmpty n m) ⇒
  Signal dom (Maybe (Cell n m)) → Signal dom Bool →
  (Signal dom (Cell n m), Signal dom (Maybe Result))
```

Internally, we create two registers for the current grid and the stack pointer. These are updated from the transition returned by `loadStep`. The stack is also driven from this transition. The `update` function returns a large tuple, but we can use a pattern matching bind with `unbundle` to take it apart into separate named signals.

```
solver cellIn enable = (headGrid ⟨$⟩ grid, result)
  where
    grid    = register (pure wild) grid'
    sp      = register 0 sp'
    stackRd = ram stackCmd
    (grid', sp', stackCmd, result) = unbundle $
      update ⟨$⟩ cellIn ⟨*⟩ enable
             ⟨*⟩ stackRd ⟨*⟩ grid ⟨*⟩ sp
    update cellIn enable stackRd grid sp =
      case loadStep cellIn enable stackRd grid sp of
        Continue (grid', sp', stackCmd) →
          (grid', sp', stackCmd, Nothing)
        Done result →
          (grid, sp, Nothing, Just result)
```

7 The boring bits

As functional programmers, if we were writing a Sudoku solver *program*, we would implicitly understand that the end result should be something we can run on some operating system on top of some general-purpose computer. The operating system would provide some ways of communicating with the user, and we could, for example, decide to make use of pipe-based IO to read in a problem description from standard input and write out a representation of a solution to standard output. We could implement this design by composing our pure `sudoku` function with a parser and a pretty-printer, and then wrapping it with `interact :: (String → String) → IO ()`.

For our circuit, we have to make analogous design decisions in the hardware space. We use a *universal asynchronous receiver-transmitter* (UART) to deserialize an input stream of ASCII characters describing a Sudoku problem, and then return the result in a similar serialized ASCII stream. With just two wires, some careful timing, and using the right physical signal levels, this allows us to interface with mid-20^{th} century teletype terminals as well as contemporary desktop computers.

By simply ignoring invalid characters, we can support many different input formats with various spacing elements or even ASCII art between boxes. The left-hand side input in Figure 7 shows one example of such a format.

We implement a simple state machine (using the `State` monad) to load Sudoku problems from the user one `given` or `wild` cell at a time, as they become available; after N^2 cells are loaded in, the solver is enabled; finally the solution is sent out (or a single `conflicted` cell if the problem is unsolvable). The high-level structure of the complete circuit is shown in Figure 8.

```
. 2 . | 9 . 8 | . . .          4 2 1  9 5 8  6 3 7
8 7 . | . . 1 | . 5 4          8 7 3  6 2 1  9 5 4
5 . 6 | 4 . . | . 1 .          5 9 6  4 7 3  2 1 8
------+-------+------
. . 2 | . . . | . 9 5          3 1 2  8 4 6  7 9 5
. . . | . . . | . . .          7 6 8  5 1 9  3 4 2
9 4 . | . . . | 8 . .          9 4 5  7 3 2  8 6 1
------+-------+------
. 8 . | . . 4 | 5 . 3          2 8 9  1 6 4  5 7 3
1 3 . | 2 . . | . 8 6          1 3 7  2 9 5  4 8 6
. . . | 3 . 7 | . 2 .          6 5 4  3 8 7  1 2 9
```

Figure 7. Example input and output for the $(3, 3)$ solver

8 Testing, hardware synthesis, and performance

One huge benefit of implementing most of our functionality as pure functions is we can directly apply all the usual software testing tools of Haskell like property-based unit testing [5, 12]. For the stateful parts, Clash provides a simulator that allows observing the given circuit's behaviour cycle by cycle, potentially driving its inputs monadically.

For example, we can write integration tests that feed a given grid's cells to the `controller` and then consume its output until the N^2-th cell is shifted out. We can also use the same mechanism for complete end-to-end testing of our board that includes the serial receiver-transmitter.

In the introduction, we promised performance: after all, why bother with a hardware solver if it ends up taking longer to solve Sudoku problems? We can use the simulator to measure our circuit's performance: every step of the simulation corresponds to one clock cycle. Excluding the time of the serial communication, our testing shows most $(3, 3)$ Sudoku boards get solved in less than 100 cycles. Some $(3, 3)$ problems generally regarded as hard instances [8] can take several thousand cycles. As an outlier, Norvig presents an "impossibly hard" $(3, 3)$ problem [11], which for our solver takes a whopping 867,856 cycles to find out is unsolvable.

For real hardware synthesis, a Xilinx XC7A50T chip was used as a target, feeding the Clash output to the vendor's Vivado toolchain. For simplicity's sake, the on-chip clock's default 100 MHz rate was used. For the $(3, 3)$ solver, synthesis with Vivado takes about 10 minutes and reports 9,298 logic cells used out of 32,600. At 100 MHz, the hardest realistic $(3, 3)$ instances take less than 10 microseconds to solve, and of course even the "impossible" problem falls in 8.7 milliseconds. For the $(3, 4)$ solver, Vivado ends up using 25,978 logic cells.

9 Can we do even more?

Our Sudoku solver makes good use of FPGA resources to implement pruning of all N^2 cells in a single clock cycle. There are two directions we can go from here.

The first is to try being even faster. As a small win, we could get rid of the extra cycle in `step` when popping from the stack with simple form of pipelining: ensuring that the stack RAM is always connected to the right address. This would work because the only cycles in which we want to use a different address value is when we push; and if we discover immediately after pushing that we need to pop, the address we just pushed to is already exactly the right address to pop from.

Since our backtracking strategy implements no heuristics on choosing a guess candidate, we can end up unlucky. On one example problem presented by Norvig [11], we find a solution in 105 clock cycles when `expand` is defined using `mapAccumR`. But if we change `expand` to choose the first ambiguous cell instead of the last one (by replacing `mapAccumR` with `mapAccumL`), solving time is improved to 93 cycles. In contrast, on another example problem from the same source, solving time shoots up from 74 cycles to 15,340 cycles! This suggests a possible improvement of instantiating `solver` multiple times, with different `expand` policies, and connecting them to a single `controller` that runs each of them until one of them finds a solution (i.e. a hardware version of the `amb` combinator [10]). On the XC7A50T, we have enough block RAM that the limitation comes from logic blocks; it should be possible to replicate the $(3, 3)$ solver three times.

The other direction is aiming for larger problem sizes. Unfortunately, our approach of immediate constraint propagation quickly becomes unfeasible, since we have each of the N^2 cells connected to $3(N^2 - 1)$ neighbours with N wires. Direct pruning is the defining characteristic of our design, but we could improve scaling by a factor of N by applying temporal multiplexing to pruning, i.e. by propagating one bit-mask position at a time.

10 Closing Remarks on Clash

The codebase described in this paper is compiled with Clash's development version as of December 2024, Git hash `3b755b90`. The two benefits of using the development version instead of Clash 1.8, the latest stable release, is twofold: GHC 9.10 compatibility for new language features, and the ability to track new bug fixes on the Clash side.

A recurring problem during development has been seemingly equivalent changes in the Clash code suddenly causing large changes in the size of the synthesized circuit.

One flavor of this is changing sum type fields to separate signals. For example, instead of two `Sudoku n m` fields in `Step`'s `Stuck` constructor we can move the second one into a separate return value of `solve` and connect it directly to the stack's write input. We have not applied this transformation in this paper since it makes the code less clear: its effect is that we are always producing a grid to "write", even in clock cycles when we are not pushing (i.e. in cycles where the `MemCmd` signal's value is not `Just Write`). However, its

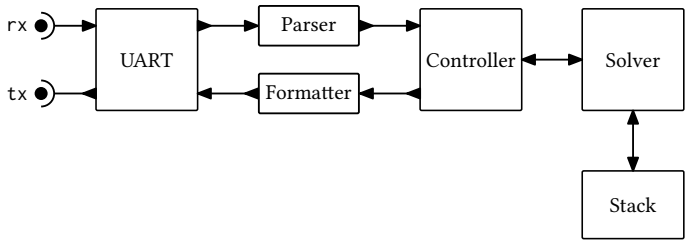


Figure 8. Block diagram of the complete Sudoku solver circuit

positive effect on circuit size is massive: for the (3, 3) solver, it decreases logic block count from 9,298 to 7,781, an improvement of more than 16%. For (3, 4) the effect is even more pronounced, going from 25,978 to 20,057 (-22%).

To illustrate a different situation with supposedly equivalent code yielding different circuit size, recall the definition of `guess` in `expand`:

```
guess done cell | not done ∧ not single = ...
                | otherwise             = ...
```

A reasonable alternative definition is changing the first branch from a conjunction to two guards:

```
guess done cell | not done, not single = ...
                | otherwise            = ...
```

But experimentally, this change leads to generated HDL with the right-hand side duplicated, and since `guess` is instantiated 81 times in the (3, 3) case, we end up with an increase in total logic block usage by more than 12 percent.

In other cases, Clash downright fails to produce HDL. Defining `expand` via `mapAccumL` instead of `mapAccumR` is just as valid and works just as well in the Clash simulator (and we have seen in section 9 that it can lead to interestingly different outcomes for specific Sudoku problems), but it causes the compiler to loop.

Nevertheless, as hopefully the presented program demonstrates, the overall experience of using Clash is very pleasant for the experienced functional programmer. We took an idiomatic Haskell implementation of a Sudoku solver and were able to keep most of its structure, including its usage of high-level abstractions like isomorphisms and monoidal folds, while adopting it into a high-performance FPGA design.

Acknowledgments

Thanks to Felix Klein, Rowan Goemans, and Leon Schoorl for valuable discussions regarding finding and fixing Clash problems, extending the Clash libraries where needed, and micro-optimizing the various low-level bit twiddling operations for efficient and small hardware implementations.

This paper has turned out much better than it would have had without the thoughtful and thorough feedback from Lennart Augustsson, Christiaan Baaij, Atze Dijkstra, Felix Klein, Alex Mason, Jurriën Stutterheim, and Wouter Swierstra.

References

[1] Christiaan Baaij. 2009. *Cλash: From Haskell to hardware.* Master's thesis. University of Twente.

[2] Richard Bird. 2006. Functional pearl: A program to solve Sudoku. *Journal of functional programming* 16, 6 (2006), 671–679. doi:10.1017/S0956796806006058

[3] Richard Bird, Jeremy Gibbons, Stefan Mehner, Janis Voigtländer, and Tom Schrijvers. 2013. Understanding idiomatic traversals backwards and forwards. In *Proceedings of the 2013 ACM SIGPLAN symposium on Haskell*. 25–36. doi:10.1145/2503778.2503781

[4] Baldur Blöndal, Andres Löh, and Ryan Scott. 2018. Deriving Via: or, how to turn hand-written instances into an anti-pattern. In *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell*. 55–67. doi:10.1145/3242744.3242746

[5] Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*. 268–279. doi:10.1145/1988042.1988046

[6] Gergő Érdi. 2021. *Retrocomputing with Clash: Haskell for FPGA hardware design.* https://gergo.erdi.hu/retroclash/.

[7] Jeremy Gibbons. 2022. Continuation-Passing Style, Defunctionalization, Accumulations, and Associativity. *The Art, Science, and Engineering of Programming* 6, 2 (2022). doi:10.22152/programming-journal.org/2022/6/7

[8] Martin Henz and Hoang-Minh Truong. 2009. SudokuSat – A tool for analyzing difficult Sudoku puzzles. In *Tools and Applications with Artificial Intelligence*. doi:10.1007/978-3-540-88069-1_3

[9] Conor McBride and Ross Paterson. 2008. Applicative programming with effects. *Journal of functional programming* 18, 1 (2008), 1–13. doi:10.1017/S0956796807006326

[10] John McCarthy. 1959. A basis for a mathematical theory of computation. In *Studies in Logic and the Foundations of Mathematics*. Vol. 26. Elsevier, 33–70.

[11] Peter Norvig. 2009. Solving every Sudoku puzzle. *Preprint* (2009).

[12] Colin Runciman, Matthew Naylor, and Fredrik Lindblad. 2008. Smallcheck and lazy smallcheck: automatic exhaustive testing for small values. *ACM SIGPLAN notices* 44, 2 (2008), 37–48. doi:10.1145/1411286.1411292

[13] David Seal. 1994. Re: The next ARM-coders Challange. Usenet post. https://groups.google.com/g/comp.sys.acorn.tech/c/blRy-AiIQ-0/m/ySyrwf6EUj4J Posted to `comp.sys.acorn.tech`.

[14] The GHC Team. 1992. *The Glasgow Haskell Compiler.* https://www.haskell.org/ghc/

[15] Takayuki Yato and Takahiro Seta. 2003. Complexity and completeness of finding another solution and its application to puzzles. *IEICE transactions on fundamentals of electronics, communications and computer sciences* 86, 5 (2003), 1052–1060. doi:10.7551/mitpress/2003.003.0032

[16] Brent Yorgey. 2025. You could have invented Fenwick trees. *Journal of Functional Programming* 35 (2025), e3. doi:10.1017/S0956796824000169

[17] Brent A Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. 2012. Giving Haskell a promotion. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*. 53–66. doi:10.1145/2103786.2103795

[18] Kuangya Zhai, Richard Townsend, Lianne Lairmore, Martha A Kim, and Stephen A Edwards. 2015. Hardware synthesis from a recursive functional language. In *2015 International Conference on Hardware/-Software Codesign and System Synthesis (CODES+ ISSS)*. IEEE, 83–93. doi:10.1109/codesisss.2015.7331371

Received 2025-05-24; accepted 2025-07-17