FPGA-on-FPGA emulation

Pim van Leeuwen

Friday 7th February, 2020

Abstract

FPGAs allow reconfiguration of its logic at any point after production. The result is that they are effective at prototyping application-specific integrated circuits, updating the internal logic while in the field and at low-cost low-quantity use cases. To optimise these processes, it is crucial to properly educate engineers in the implementation of FPGA programs and the FPGA compilation process. Traditional FPGA programming pipelines involve a computationally expensive (NP-hard) place & route process that slows down iterations of FPGA programs and hinders the educational process. We propose a virtual environment in which place & route is performed manually in which the student learns about the intricacies of place & route and in which compilation is linear. To this end, we require emulation of a virtual FPGA on a physical, concrete FPGA. In the proposed research, we find such an emulation using an algorithm that solves a variant of subgraph isomorphism. This algorithm aims to find emulations in as many cases as possible and to exploit the hierarchy of FPGAs to speed up computation. The expected result is a software package that computes emulators which each output a program for a concrete FPGA provided with a program for a virtual FPGA.

Contents

1 Introduction

p	q	p XOR q
F	F	F
F	T	T
T	F	T
T	T	F

Figure 1: A truth table that shows the result of an XOR operation

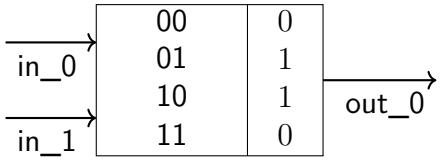


Figure 2: A Lookup Table (LUT) in which an XOR-operation is configured

2 Background

2.1 Field Programmable Gate Arrays

The computing hardware most people are familiar with is CPUs. Manufacturers incorporate them in every desktop pc, laptop, and most mobile devices. CPUs are very flexible and efficient- which is why they are the de facto standard for any computation task. In CPU computation, an integrated circuit (a processor) iteratively reads an instruction from a RAM module (in the form of encoded bits), performs the instruction, and then continues to read the next instruction. The instructions are not embedded in the circuit of the CPU itself.

FPGAs are different from CPUs, as they do not store the programs they execute in RAM- they instead configure them in the (highly parallel) logic of the circuit itself. Configuring an FPGA to execute a specific program entails loading a configuration file onto the hardware and restarting the FPGA such that it reconfigures its logic. The hardware will then perform the configured logic on the input it receives via IO pins.

Because each logic cell performs logic independently, FPGAs can perform computations highly parallel and without delays from loading instructions. This computation process implies that FPGAs are very efficient at executing individual programs, specifically concurrent ones. Reconfiguring an FPGA is, however, a relatively expensive operation. Therefore, FPGAs are unsuitable for changing from program to program, as a CPU does when running an operating system.

FPGAs perform execution using lookup tables, registers, and special-purpose modules. These are physical modules that are on the circuit. In the next sections, we will discuss these modules.

2.1.1 Lookup tables

Any boolean logic formula can be expressed in the form of a truth table, such as in Figure 1. The leftmost column of a truth table specifies all possible combinations of T and F for all inputs; the rightmost column then specifies what output that specific logic function would give. Each

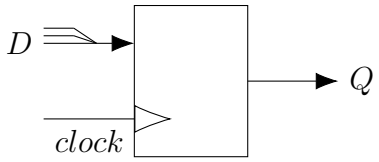


Figure 3: Traditional representation of a register

distinct combination of T and F in the rightmost column corresponds to a different boolean formula. FPGAs execute logic in the form of Lookup Tables (LUTs), which model the evaluation of a truth table, but with ones and zeroes instead of T and F: for every combination of ones and zeroes in the input, the LUT stores what output it should give. Vendor software can reconfigure these outputs. This way, any boolean formula with the appropriate number of variables can be implemented with a lookup table. Figure 2 shows a lookup table that has two input bits and one output bit. This lookup table is configured to perform an XOR-operation. Lookup tables can have input and output consisting of any number of bits, depending on the FPGA design.

2.1.2 Registers

Programs require intermediate data storage to perform any computation that is more complex than finite logic or fixed-point calculations.

FPGAs use registers (or D flip-flops) for this purpose (see Figure 3). Registers can store a small collection of bits, depending on the FPGA design. When a register's input $clock$ changes from 0 to 1, the contents of the register are replaced by the value of the input D, and the output Q takes this value. The output stays constant until the clock changes from 0 to 1 again when D has a different value.

FPGAs usually have a global clock- a wire which signal constantly changes between 0 and 1 that is connected to all registers in the hardware. The frequency of this clock is such that all signals are guaranteed to be stable when the clock becomes 1. This stability is very convenient for programmers, who do not have to calculate the time it takes for a signal to propagate through a wire. The implication is that each circuit combining only lookup tables that end in the D-input of a register takes the same amount of time.

If the FPGA program has longer chains of lookup tables, then it takes longer for the output signal to stabilize. The vendor software accommodates for this by setting the global clock at a lower frequency. Since programs on FPGAs are highly parallel, there are likely some parts of the calculation that do not require a lower clock speed and can cause slowdown because of this. In these scenarios, adding registers to some parts of an FPGA program can improve performance if it allows the global clock speed to be higher.

2.1.3 Logic Cells

A typical logic cell is a combination of one or more lookup tables, a register, and a MUX (a 3-input gate that outputs a copy of the first or second input, depending on the value of the third input). The contents of the lookup tables can be configured to perform any logic operation,

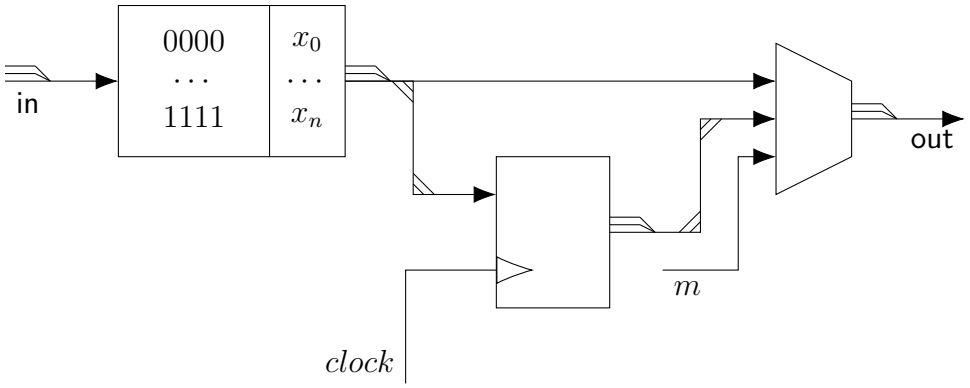


Figure 4: An n-input logic cell. x_k and m must be configured for any $0 \leq k \leq n$

and the value of the third MUX-input can be configured to specify whether the computation is synchronous or asynchronous. A logic cell is shown in Figure 4. Unfortunately for us, vendors have different logic cell designs, meaning we cannot generalize this example.

FPGA manufacturers often group Logic cells in Configurable Logic Blocks (CLB) that make hardware production, placement, and routing (Section 2.1.6) easier.

The main building blocks of FPGAs are CLBs that are connected via routing fabric. These are responsible for most of the computation of FPGA programs. FPGAs can have additional modules such as RAM and Digital Signal Processors that optimize storage and specialized arithmetic, respectively. Each of these modules' functionality could also be performed by a collection of logic cells at the cost of performance. In this research, we will only consider CLBs.

2.1.4 Routing

A single logic cell can only perform simple programs such as logic gates. The FPGA needs to connect different logic cells to perform any kind of logic operation, dependent on what program it needs to execute. The way logic cells and other modules are connected on an a physical FPGA in a specific configuration is called the **routing** of an FPGA. On the most basic level, FPGAs perform routing with configurable switches. These are tiny modules on an FPGA board that is connected with > 2 wires, and can be configured to block signals from specific wires; the other wires can then send- and receive signals.

2.1.5 Pins

To supply the FPGA with input and to allow it to provide output, an FPGA is equipped with a set of metal pins. Each of those pins is connected with a wire on the FPGA board. Each pin can be used for either input or output, depending on the configured program. For example, an FPGA used to control an electric stepper motor will receive input with the requested motor speed and direction and will output power to specific electromagnets that need to be activated.

2.1.6 Compilation

To program an FPGA, an engineer has to specify a hardware design that describes the semantics of the program. They write these hardware designs in a Hardware Description Language (HDL). Commonly used languages for this purpose are VHDL and Verilog. They specify on an abstract level what functionality the program should have, preferably in a modular structure to improve maintainability.

The software then needs to translate this abstract description to a description of logic, data storage, and how they are connected. This process depends on the available hardware components on a physical FPGA and is thus hardware-specific. This process is called **synthesis**. Although synthesis is a polynomial problem, it can still take minutes to perform for some programs[4, 3].

The next step in the compilation process is **placement**. The software maps each LUT, register, and other component to a physical place on the FPGA. The software can place components closer together to optimize the speed or can place components further apart to increase the probability routes can be found. Finding the optimal placement for speed is a proven NP-hard problem.

The last step in the compilation process is **routing**. State-of-the-art FPGA compilation pipelines perform this step sequentially after placement[2]. With the components locked in place, the software attempts to find a configuration of routing switches such that each connection that the synthesis requires is made. Finding the optimal routes for speed is also an NP-hard problem. However, finding any matching routing configuration is an NP problem.

Note that place & route of an FPGA *program* as described here is a very similar problem to placement of an unconfigured virtual FPGA, which is part of this research. A conventional place & route algorithm could (with some minor modification) place a virtual FPGA model with all its components on a concrete FPGA, resulting in a mapping we could use for emulation.
The problem with this approach is that state-of-the art place & route pipelines perform these algorithms sequentially: they perform placement based on speed- and routeability heuristics without guarantee that the placement can indeed be routed[2]. With placement of FPGA programs this is not a problem. Entire FPGAs, however, can have dense collections of components that can severely impact routeability.
Not using well-founded placement algorithms based on heuristics will undoubtedly have an adverse impact on the speed of our placement. However, we can use the hierarchical structure of an FPGA for significant improvements that could not be obtained with placement of FPGA programs.

2.2 Simulation versus Emulation

2.3 Path subgraph isomorphism

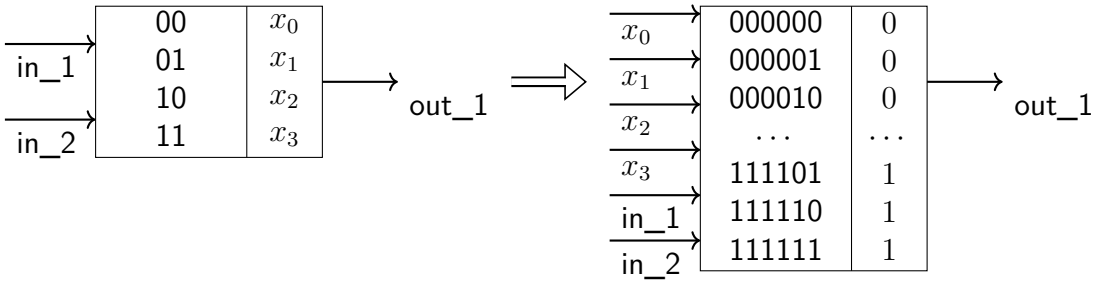


Figure 5: Simulation of an FPGA: the programming of the virtual FPGA is reflected in additional input of the actual FPGA. The value of x in the simulation is identical to the configuration of the program to be simulated.

3 Models

3.1 FPGA

Our algorithm will generate an emulation using path subgraph isomorphism. To this end, we will model FPGA designs as graphs. For this purpose, let us define the type $hierarchygraph$.

Definition 3.1. A $hierarchygraph$ is a structure $\subseteq (V, E, L, H, C)$ where:

- V is a finite set of vertices.
- $E \subseteq (V \times V)$ is a finite set of undirected edges without loops, i.e. $\nexists v \in V.(v, v) \in E$.
- L is a vertex multilabeling function $V \to P(\lambda)$ where $P(\lambda)$ is the power set of a finite set of labels $\lambda = \{\text{“}in\text{”}, \text{“}out\text{”}, \text{“}component\text{”}, \text{“}port\text{”}, \text{“}mux\text{”}, \text{“}lut\text{”}, \text{“}select\text{”}, \text{“}set\text{”}, \text{“}async\text{”}, \text{“}sync\text{”}\}$.
- $H \subseteq (V \times G)$ where G is the set of all $hierarchygraphs$. This describes the hierarchy of an FPGA.
- $C \subseteq (V \times V_H)$ where $V_H = \bigcup_{v_i,(V_i,E_i,L_i,H_i,C_i)\in H} V_i$. This allows us to separate different wires coming out of a hierarchical component of the FPGA by linking them with vertices in the $hierarchygraph$ definition of that component.

  Furthermore, the hierarchy is *finite*. I.e if we define a function $depth \in (G \times \mathbb{N})$:

$$depth((V, E, L, H, C)) = \left\{ \begin{array}{ll} 0 & \text{for } H = \emptyset \\ 1 + \sum\limits_{H_i \in H} depth(H_i) & \text{for } H \neq \emptyset \end{array} \right\}$$

  then $depth$ is defined for every hierarchygraph.

  Definition 3.2. Let $model(B)$ be a $hierarchygraph$ model of (a subset of) an FPGA layout B such that:

- For each pin in B, $model(B)$ has a vertex v such that $L(v) = \{\text{“}pin\text{”}\}$.
- For each routing switch in B, $model(B)$ has a vertex v such that $L(v) = \{\text{“}switch\text{”}\}$.
- For each component in B that is one step lower than B in the FPGA hierarchy (e.g. CLBs), $model(B)$ has a distinct vertex v such that $L(v) = \{\text{“}component\text{”}\}$ and a set of vertices S such that $\forall s \in S.L(s) = \{\text{“}port\text{”}\} \wedge (s, v) \in E$. Furthermore, the contents of this component are recursively defined in $H(v)$ such that ports are linked:

  $\forall s \in S.\exists w \in (H(v).V).(s, w) \in C$.

  Furthermore, this link is unique, i.e.:

  $\forall w_1, w_2 \in (H(w).V).\forall s \in S.((s, w_1) \in C \wedge (s, w_2) \in C) \to w_1 = w_2$

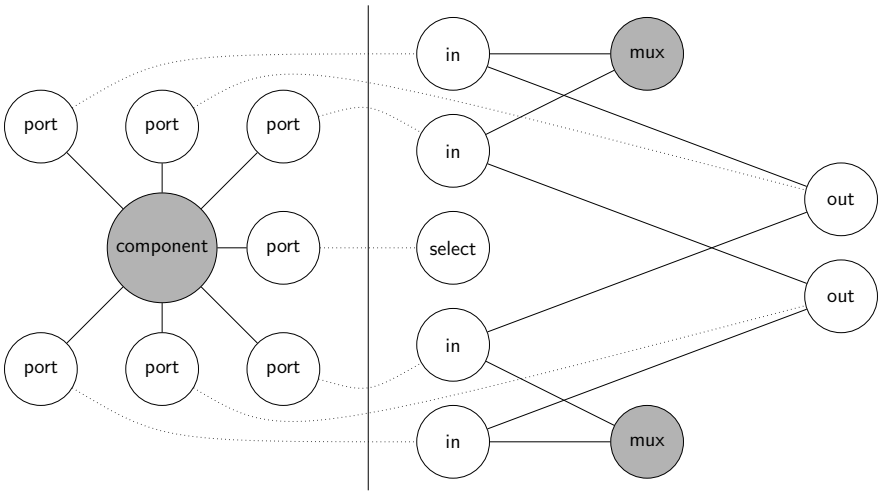


Figure 6: How a 2-wire mux is represented in $model(B)$ (left) in terms of vertex v with $L(v) =$ "$component$", port vertices and edges, and the $hierarchygraph$ $H(v)$ (right) storing the internal details of the mux. Dashed lines indicate the relation C that links the vertices representing outgoing wires. Vertex labels are shown as text. See Figure 9 for an example where the mux is connected to other parts.

- For each mux with n wires for each input and for its output in B, $model(B)$ has a distinct vertex v such that $L(v) = \{\text{"}component\text{"}\}$ and a set of vertices S such that $|S| = 3n+1$.

  $\forall s_i \in S.L(s_i) = \{\text{"}port\text{"}\} \wedge (v, s_i) \in E$

  $H(v) = (V_{mux}, E_{mux}, L_{mux}, \emptyset, \emptyset)$ where:

$$
\begin{aligned}
V_{mux} = \quad & \{v_{left,1} \ldots v_{left,n}\} \cup \{v_{right,1} \ldots v_{right,n}\} \cup \{v_{out,1} \ldots v_{out,n}\} \cup \{v_{left}, v_{right}\} \\
E_{mux} = \quad & \{(v_{left,i}, v_{left}) \quad : v_{left,i} \quad \in \{v_{left,1} \ldots v_{left,n}\}\} \quad \cup \\
& \{(v_{right,i}, v_{right}) \quad : v_{right,i} \quad \in \{v_{right,1} \ldots v_{right,n}\}\} \quad \cup \\
& \{(v_{left,i}, v_{out,i}) \quad : v_{left,i} \quad \in \{v_{left,1} \ldots v_{left,n}\}\} \quad \cup \\
& \{(v_{right,i}, v_{out,i}) \quad : v_{right,i} \quad \in \{v_{right,1} \ldots v_{right,n}\}\} \\
L_{mux} = \quad & \{(v_{left,i} \quad , \{\text{"}in\text{"}\}) \quad : v_{left,i} \quad \in \{v_{left,1} \ldots v_{left,n}\}\} \quad \cup \\
& \{(v_{right,i} \quad , \{\text{"}in\text{"}\}) \quad : v_{right,i} \quad \in \{v_{right,1} \ldots v_{right,n}\}\} \quad \cup \\
& \{(v_{out,i} \quad , \{\text{"}out\text{"}\}) \quad : v_{out,i} \quad \in \{v_{out,1} \ldots v_{out,n}\}\} \quad \cup \\
& \{(v_{left} \quad , \{\text{"}mux\text{"}\}), (v_{right}, \{\text{"}mux\text{"}\})\}
\end{aligned}
$$

  Lastly, the relation C contains a link from nodes in the FPGA $hierarchygraph$ to nodes in the mux $hierarchygraph$, i.e. $\forall s \in S. \exists w \in V_{mux}.(s, w) \in C$. The model of a mux is illustrated in Figure 6.

- For each LUT in B with n inputs and m outputs, $model(B)$ has a distinct vertex v such that $L(v) = \{\text{"}component\text{"}\}$ and a set of vertices S such that $|S| = n + m$.

  $\forall s \in S.L(s) = \{\text{"}port\text{"} : i \in \{1 \ldots n\}\} \wedge (v, s) \in E$

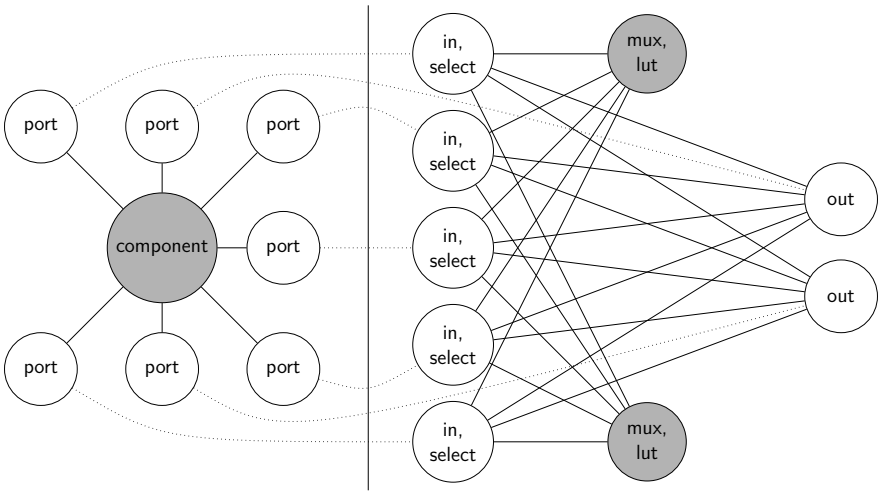


Figure 7: How a 5-input-2-output lut is represented in $model(B)$ (left) in terms of vertex v with $L(v) =$ "$component$", port vertices and edges, and the $hierarchygraph$ $H(v)$ (right) storing the internal details of the lut. This $hierarchygraph$ representation contains subgraphs of muxes that this lut can emulate. Dashed lines indicate the relation C that links the vertices representing outgoing wires. Vertex labels are shown as text.

$H(v) = (V_{lut}, E_{lut}, L_{lut}, \emptyset, \emptyset)$ where:

$$V_{lut} = \{v_{in,1} \dots v_{in,n}\} \cup \{v_{out,1} \dots v_{out,m}\} \cup \{v_{left}, v_{right}\}$$

$$\begin{aligned} E_{lut} = \quad & \{(v_{in,i}, v_{left}) \quad : v_{in,i} \in \{v_{in,1} \dots v_{in,n}\}\} \cup \\ & \{(v_{in,i}, v_{right}) \quad : v_{in,i} \in \{v_{in,1} \dots v_{in,n}\}\} \cup \\ & \{(v_{in,i}, v_{out,j}) \quad : v_{in,i} \in \{v_{in,1} \dots v_{in,n}\}, v_{out,j} \in \{v_{out,1} \dots v_{out,m}\}\} \end{aligned}$$

$$\begin{aligned} L_{lut} = \quad & \{(v_{in,i}, \{\text{“}in\text{”}, \text{“}select\text{”}\}) \quad : v_{in,i} \in \{v_{in,1} \dots v_{in,n}\}\} \quad \cup \\ & \{(v_{out,i}, \{\text{“}out\text{”}\}) \quad : v_{out,i} \in \{v_{out,1} \dots v_{out,m}\}\} \quad \cup \\ & \{(v_{left}, \{\text{“}in\text{”}, \text{“}select\text{”}\})\} \quad \cup \\ & \{(v_{right}, \{\text{“}in\text{”}, \text{“}select\text{”}\})\} \end{aligned}$$

Lastly, the relation C contains a link from nodes in the FPGA $hierarchygraph$ to nodes in the lut $hierarchygraph$, i.e. $\forall s \in S. \exists w \in V_{lut}. (s, w) \in C$. The model of a lut is illustrated in Figure 7.

- For each register in B with n inputs, $model(B)$ has a distinct vertex v such that $L(v) = \{\text{“}component\text{”}\}$ and a set of vertices S such that $|S| = 2n + 2$ if the register has synchronous- or asynchronous reset capabilities, $|S| = 2n+3$ if it has both or $|S| = 2n+1$ if it has neither.

  $\forall s \in S. L(s) = \{\text{“}port\text{”}\} \wedge (v, s) \in E$

  $H(v) = (V_{reg}, E_{reg}, L_{reg}, \emptyset, \emptyset)$ where:

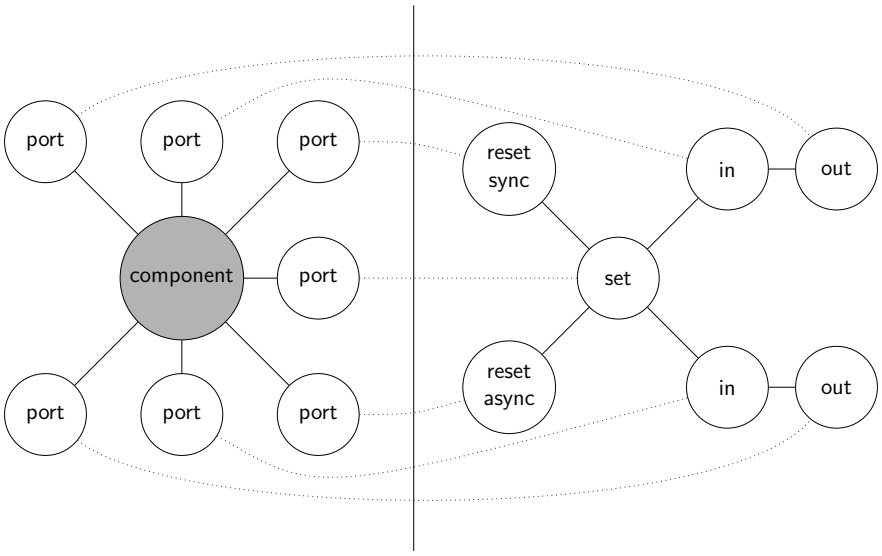


Figure 8: How a 2-input register is represented in $model(B)$ (left) in terms of vertex v with $L(v) = \text{“}component\text{”}$, port vertices and edges, and the $hierarchygraph$ $H(v)$ (right) storing the internal details of the register. Dashed lines indicate the relation C that links the vertices representing outgoing wires. Vertex labels are shown as text.

$$
\begin{aligned}
V_{reg} &= \{v_{in,1} \dots v_{in,n}\} \cup \{v_{out,1} \dots v_{out,n}\} \cup \{v_{set}, {v_{sync}}^1, {v_{async}}^1\} \\
E_{reg} = \; & \{(v_{in,i}, v_{out,i}) && : v_{in,i} \in \{v_{in,1} \dots v_{in,n}\}\} \cup \\
& \{(v_{in,i}, v_{set}) && : v_{in,i} \in \{v_{in,1} \dots v_{in,n}\}\} \cup \\
& \{({v_{sync}}^1, v_{set}), ({v_{async}}^1, v_{set})\} \\
L_{reg} = \; & \{(v_{in,i}, \text{“}in\text{”}) && : v_{in,i} \in \{v_{in,1} \dots v_{in,n}\}\} \cup \\
& \{(v_{out,i}, \text{“}out\text{”}) && : v_{out,i} \in \{v_{out,1} \dots v_{out,n}\}\} \cup \\
& \{(v_{set}, \text{“}set\text{”}), ({v_{sync}}^1, \text{“}sync\text{”}), ({v_{async}}^1, \text{“}async\text{”})\}
\end{aligned}
$$

Lastly, the relation C contains a link from nodes in the FPGA $hierarchygraph$ to nodes in the register $hierarchygraph$, i.e. $\forall s \in S. \exists w \in V_{reg}. (s, w) \in C$. The model of a mux is illustrated in Figure 6.

- Each connection by wire in B corresponds to an edge in $model(B)$. For muxes, luts, registers and hierarchical components this means an edge to the “port”-vertex v for which $C(v)$ represents the appropriate input/output of the component.
- $model(B)$ contains nothing else.

[1] These vertices, their edges, their labels and mapping in C are omitted if the register is incapable of synchronous- or asynchronous reset, respectively.

3.2 FPGA program

Let $model(B)$ be the hierarchygraph of an FPGA layout B. Before we can define what an interpretation of $model(B)$ is, we need to constrain ourselves to hierarchygraphs of FPGAs in Definition 3.3. Next, we define the semantics of the configuration of a lookup table in Definition 3.5 and the semantics of the configuration of routing switches in Definition 3.6. Finally, we define an interpretation of an FPGA hierarchygraph in Definition 3.7. This interpretation is our model of an FPGA program for FPGA layout B.

Definition 3.3 (FPGA hierarchygraph)**.** An **FPGA hierarchygraph** is a hierarchygraph G such that an FPGA B exists for which $G = model(B)$.

Definition 3.4 (Vertex union)**.** Let $V^{\cup}$ be the vertex union of a hierarchygraph, defined as:

$$V^{\cup}((V, E, L, H, C)) = \left\{ \begin{array}{ll} V & \text{for } H = \emptyset \\ V \cup \bigcup\limits_{(v_i, G_i) \in H} V^{\cup}(G_i) & \text{for } H \neq \emptyset \end{array} \right\}$$

Definition 3.5 (LUT mapping)**.** A **LUT mapping** of a hierarchygraph $G = (V, E, L, H, C)$ is a function $L_{map} \subseteq V \times P(V^{\cup}(G)) \times P(V^{\cup}(G))$.

It is a set such that:

$$\begin{array}{l} \forall v.(L(v) = \text{“}component\text{”} \wedge \exists w \in (H(v).V).(H(v).L)(w) = \text{“}lut\text{”}) \rightarrow \\ \quad \forall V_{in} \in P(H(v).V).(\forall v_{in} \in V_{in}.(H(v).L)(v_{in}) = \text{“}in\text{”}) \rightarrow \\ \quad \exists V_{out} \in P(H(v).V).(\forall v_{out} \in V_{out}.(H(v).L)(v_{out}) = \text{“}out\text{”}) \wedge (v, V_{in}, V_{out}) \in L_{map} \end{array}$$

and:

$$\forall v.(L(v) \neq \text{“}component\text{”} \vee \nexists w \in (H(v).V).(H(v).L)(w) = \text{“}lut\text{”}) \rightarrow \quad \nexists V_1, V_2 \in P(V^{\cup}(G)).(v, V_1, V_2) \in L_{map}$$

Furthermore, if L_{map} is a LUT mapping for G' and $\exists v \in V.(v, G') \in H$ then L_{map} is a LUT mapping for G.

Furthermore, we define the semantics of the routing configuration of an FPGA program.

Definition 3.6 (Switch mapping)**.** A **switch mapping** of a hierarchygraph $G = (V, E, L, H, C)$ is a function $S_{map} \subseteq V \times P(V)$ that describes which connections are *not* blocked in an FPGA configuration. It is a set such that:

$(\forall v \in V.L(v) \neq \text{“}switch\text{”} \rightarrow \nexists V'.(v, V') \in S_{map}) \wedge (\forall (v, V') \in S_{map}.V' \subseteq neighbours(v))$

Definition 3.7 (Interpretation)**.** An interpretation I of an FPGA hierarchygraph $G = (V, E, L, H, C)$ is a structure (L_{map}, S_{map}) where L_{map} is a LUT mapping of G and S_{map} is a switch mapping of G.

4 Algorithm

4.1 State space storage

4.2 State space exploration

4.3 Hierarchy usage

4.4 Defining f

5 Performance experiments

6 Software design

6.1 Architecture

6.2 Formats

6.2.1 FPGA layout input

When providing a hierarchygraph $G = (V, E, L, H, C)$ to our software package, you need to provide the graph (V, E, L) in a file in a GraphML[1] format. This file starts with a prefix:

```
<?xml version="1.0" encoding="UTF-8"?>
<graphml xmlns="http://graphml.graphdrawing.org/xmlns"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation=
    "http://graphml.graphdrawing.org/xmlns/1.0/graphml.xsd">
  <key id="label"
    for="node"
    attr.name="label"
    attr.type="string"/>
  <key id="graphref"
    for="node"
    attr.name="is␣hierarchygraph"
    attr.type="string"/>
  <key id="noderef"
    for="node"
    attr.name="linked␣to"
    attr.type="string"/>
```

Next, you have to create a `graph` XML element:

```
<graph id="myfpga" edgedefault="undirected">
```

where `myfpga` should be replaced by any sensible identifier for this hierarchygraph. Next, for each vertex v in V with $L(v) = \{label_1 \dots label_n\}$ you must add a `node` XML-element within the `graph` XML element:

```
<node id="myNode">
  <data key="label">label_1</data>
  ...
  <data key="label">label_n</data>
</node>
```

Where `myNode` should be replaced by any sensible identifier for this node. Next, each "component" node $v \in V, L(v) = \{\text{“}component\text{”}\}$ should have a reference to a file with hierarchygraph $H(v)$:

```
<node id="myComponentNode">
  <data key="label">component</data>
  <data key="graphref">myComponent.graphml</data>
</node>
```

This file should be in the same directory as the file referencing to it. Next, each "port" node $v \in V, L(v) = \{\text{"}port\text{"}\}$ should have a reference to a node identifier from another hierarchygraph:

```
<node id="myPortNode">
  <data key="label">port</data>
  <data key="noderef">referredNode</data>
</node>
```

where `referredNode` is the `id` of a node in the hierarchygraph that the "component" node of this "port" node references.

Next, for each edge $(v_1, v_2) \in E$ you must add an edge XML-element within the `graph` XML-element:

```
<edge id="myEdge" source="id(v1)" target="id(v2)"/>
```

where `myEdge` is replaced by a sensible identifier for this edge and $id(v_1)$ and $id(v_2)$ are the identifiers assigned to the nodes of this edge (i.e. the value of the id-attribute for `node` XML-elements). Lastly, you must close the `graph`- and `graphml` XML-elements to get a valid hierarchygraph document.

```
</graph>
</graphml>
```

A complete example of the input format of a simple FPGA hierarchygraph (shown in Figure 9) is shown in Figures 10 and 11.

6.2.2 FPGA program input

6.2.3 FPGA program output

6.3 Manual

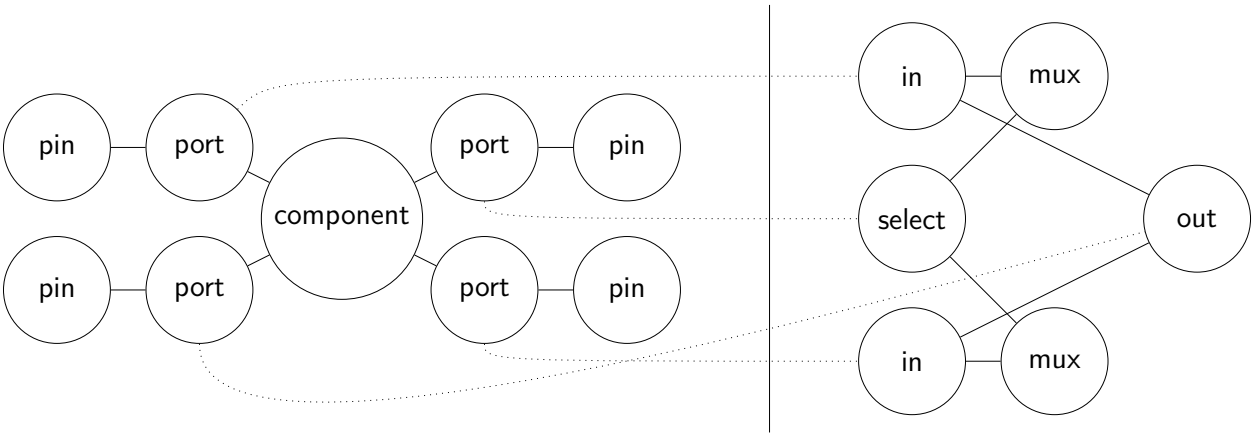


Figure 9: An FPGA hierarchygraph with four pins and a 1-wire mux.

virtualFPGA.graphml

```
<?xml version="1.0" encoding="UTF-8"?>
<graphml xmlns="http://graphml.graphdrawing.org/xmlns"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation=
    "http://graphml.graphdrawing.org/xmlns/1.0/graphml.xsd">
  <key id="label"
    for="node"
    attr.name="label"
    attr.type="string"/>
  <key id="graphref"
    for="node"
    attr.name="is hierarchygraph"
    attr.type="string"/>
  <key id="noderef"
    for="node"
    attr.name="linked to"
    attr.type="string"/>
  <graph id="virtualFPGA" edgedefault="undirected">
    <node id="n0"><data key="label">pin</data></node>
    <node id="n1"><data key="label">pin</data></node>
    <node id="n2"><data key="label">pin</data></node>
    <node id="n3"><data key="label">pin</data></node>
    <node id="n4">
      <data key="label">component</data>
      <data key="graphref">mymux.graphml</data>
    </node>
    <node id="n5">
      <data key="label">port</data>
      <data key="noderef">in1</data>
    </node>
    <node id="n6">
      <data key="label">port</data>
      <data key="noderef">in2</data>
    </node>
    <node id="n7">
      <data key="label">port</data>
      <data key="noderef">select</data>
    </node>
    <node id="n8">
      <data key="label">port</data>
      <data key="noderef">out</data>
    </node>
    <edge id="e0" source="n0" target="n5"/>
    <edge id="e1" source="n1" target="n6"/>
    <edge id="e2" source="n2" target="n7"/>
    <edge id="e3" source="n3" target="n8"/>
    <edge id="e4" source="n4" target="n5"/>
    <edge id="e5" source="n4" target="n6"/>
    <edge id="e6" source="n4" target="n7"/>
    <edge id="e7" source="n4" target="n8"/>
  </graph>
</graphml>
```

Figure 10: Hierarchygraph input of the left hierarchygraph in Figure 9

mymux.graphml

```
<?xml version="1.0" encoding="UTF-8"?>
<graphml xmlns="http://graphml.graphdrawing.org/xmlns"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation=
    "http://graphml.graphdrawing.org/xmlns/1.0/graphml.xsd">
  <key id="label"
    for="node"
    attr.name="label"
    attr.type="string"/>
  <key id="graphref"
    for="node"
    attr.name="is hierarchygraph"
    attr.type="string"/>
  <key id="noderef"
    for="node"
    attr.name="linked to"
    attr.type="string"/>
  <graph id="1-wire mux" edgedefault="undirected">
    <node id="in1"><data key="label">in</data></node>
    <node id="in2"><data key="label">in</data></node>
    <node id="select"><data key="label">select</data></node>
    <node id="mux1"><data key="label">mux</data></node>
    <node id="mux2"><data key="label">mux</data></node>
    <node id="out"><data key="label">out</data></node>

    <edge id="e0" source="in1" target="out"/>
    <edge id="e1" source="in1" target="mux1"/>
    <edge id="e2" source="in2" target="out"/>
    <edge id="e3" source="in2" target="mux2"/>
    <edge id="e4" source="select" target="mux1"/>
    <edge id="e5" source="select" target="mux2"/>
  </graph>
</graphml>
```

Figure 11: Hierarchygraph input of the right hierarchygraph in Figure 9

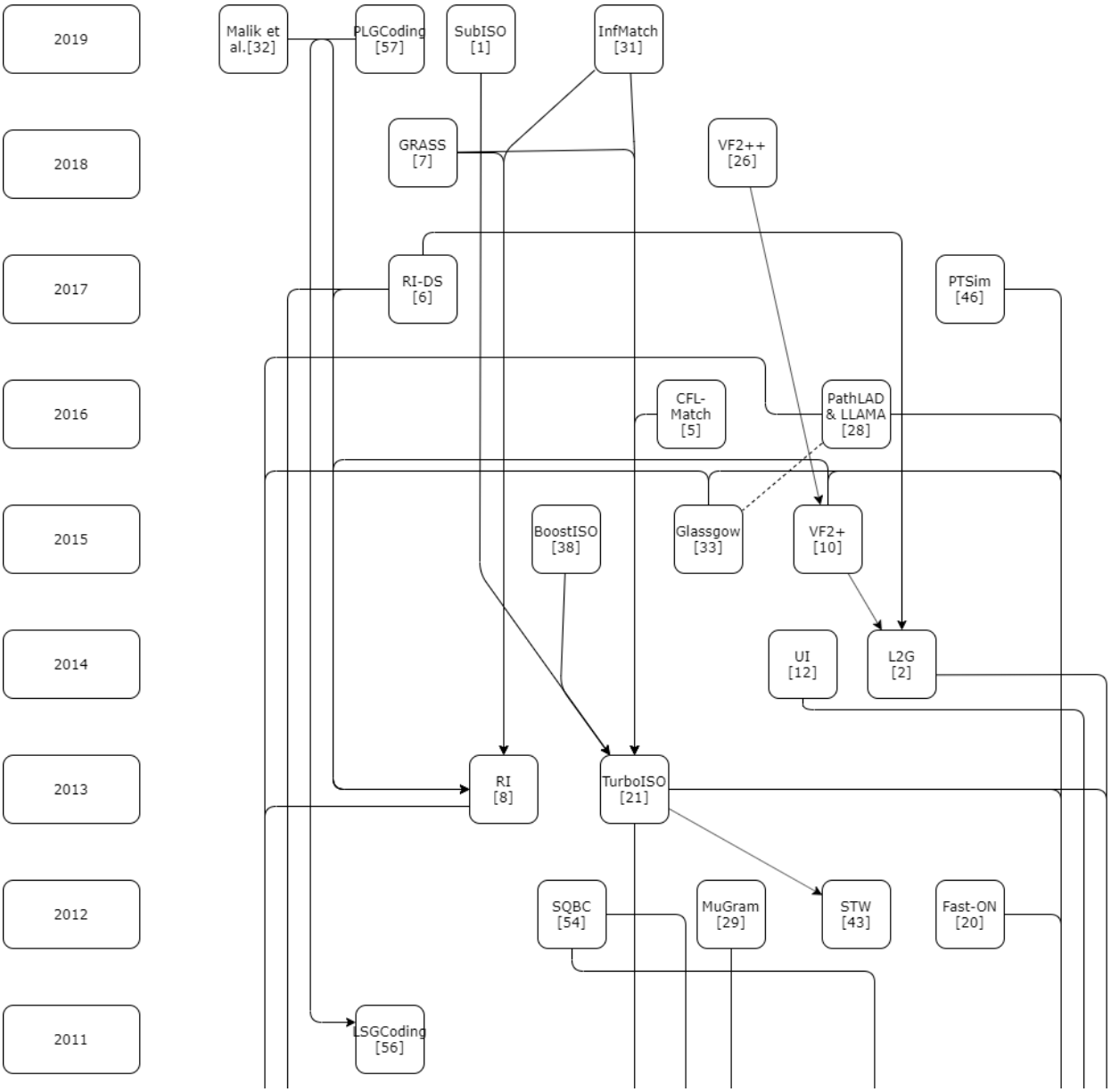
7 Conclusion

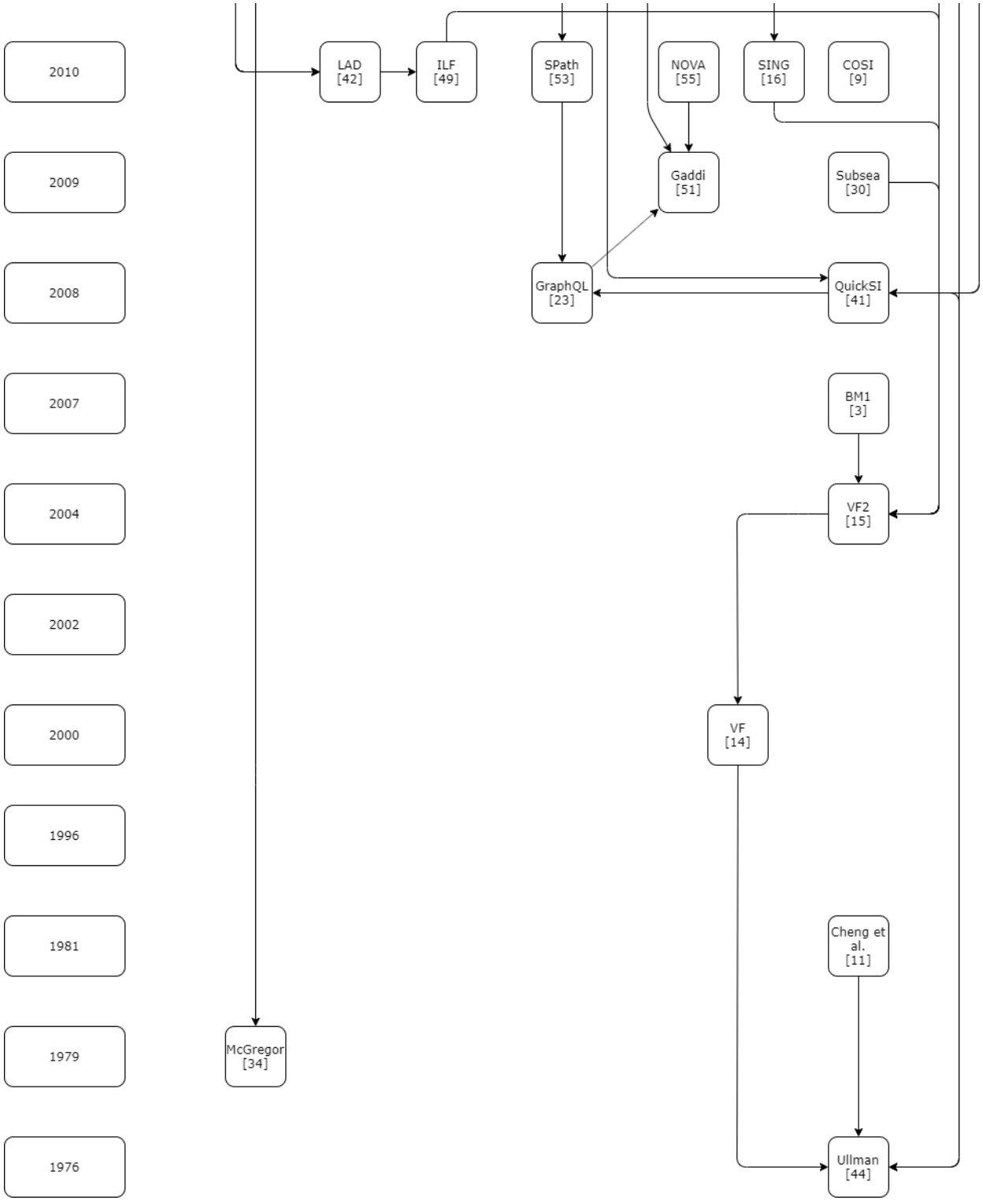
8 Discussion

9 Future Research

Appendices

A History of subgraph isomorphism algorithms




2010
2009
2008
2007
2004
2002
2000
1996
1981
1979
1976
LAD [42]
ILF [49]
SPath [53]
NOVA [55]
SING [16]
COSI [9]
Gaddi [51]
Subsea [30]
GraphQL [23]
QuickSI [41]
BM1 [3]
VF2 [15]
VF [14]
Cheng et al. [11]
McGregor [34]
Ullman [44]

References

1. MISC
2. Al-Hyari, A.: Towards Smart FPGA Placement Using Machine Learning (2019).
3. Pistorius, J., Hutton, M., Mishchenko, A., and Brayton, R.: Benchmarking method and designs targeting logic synthesis for FPGAs. In: International Workshop on Logic & Synthesis, pp. 230–237 (2007)
4. Vemuri, N., Kalla, P., and Tessier, R.: BDD-Based Logic Synthesis for LUT-Based FPGAs. ACM Transactions on Design Automation of Electronic Systems 7(4), 501–525 (2002)