Optimizing Combinational Circuits for FPGAs using Genetic Programming

Adrian Willenbücher

Master's Thesis

Supervisors:

Prof. Dr. Klaus Schneider

Dr. Jens Brandt

November 2010

Embedded Systems Group
Department of Computer Science
Technische Universität Kaiserslautern

Abstract

Although the synthesis of various hardware description languages for FPGA platforms is an intensely researched topic, the techniques for automatically inferring and instantiating components from a library of built-in, complex, flexible *hard blocks* in a given circuit are only poorly developed.

In this thesis, I present a heuristic, non-deterministic algorithm to solve this problem using *genetic programming*, which is a subtype of evolutionary algorithms. Individual circuits are mutated using semantics-preserving *transformation rules*, which employ logical and arithmetic equivalences.

In order to demonstrate the viability and effectiveness of this approach, I implemented and evaluated it using various inputs and configurations. The results as well as implementation details and design decisions are documented in this thesis.

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und die aus anderen Quellen entnommenen Stellen als solche gekennzeichnet habe.

Kaiserslautern, den 14. November 2010

Adrian Willenbücher

Contents

List of Figures

Chapter 1

Introduction

A *field-programmable gate array* (FPGA) is a type of integrated circuit which can be reprogrammed after fabrication. Although they provide functionality similar to *application specific integrated circuits* (ASIC), which are not programmable, FPGAs consist of a regular structure: they are usually built up from a two-dimensional array of *logic blocks*, each of which in principle consists of a look-up table (LUT) and a flip-flop (FF) (Figure 1.1). In modern FPGAs, several such LUT/FF pairs are combined and share some control signals; additional logic is available for common tasks, e.g., carry-bit networks (Figure 1.2). The individual logic blocks are connected by a routing network (Figure 1.3).

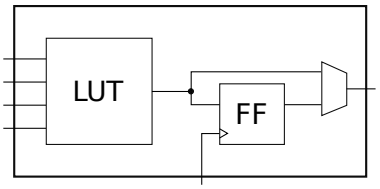


Figure 1.1: Basic structure of a logic block

In most FPGAs, the contents of the LUT as well as the settings for the router switches are stored in SRAM cells (*static random-access memory*), making them reprogrammable arbitrarily often.

In comparison to ASICs, the disadvantages of such fixed but programmable structures are larger (and thus more expensive) chips, higher power require-

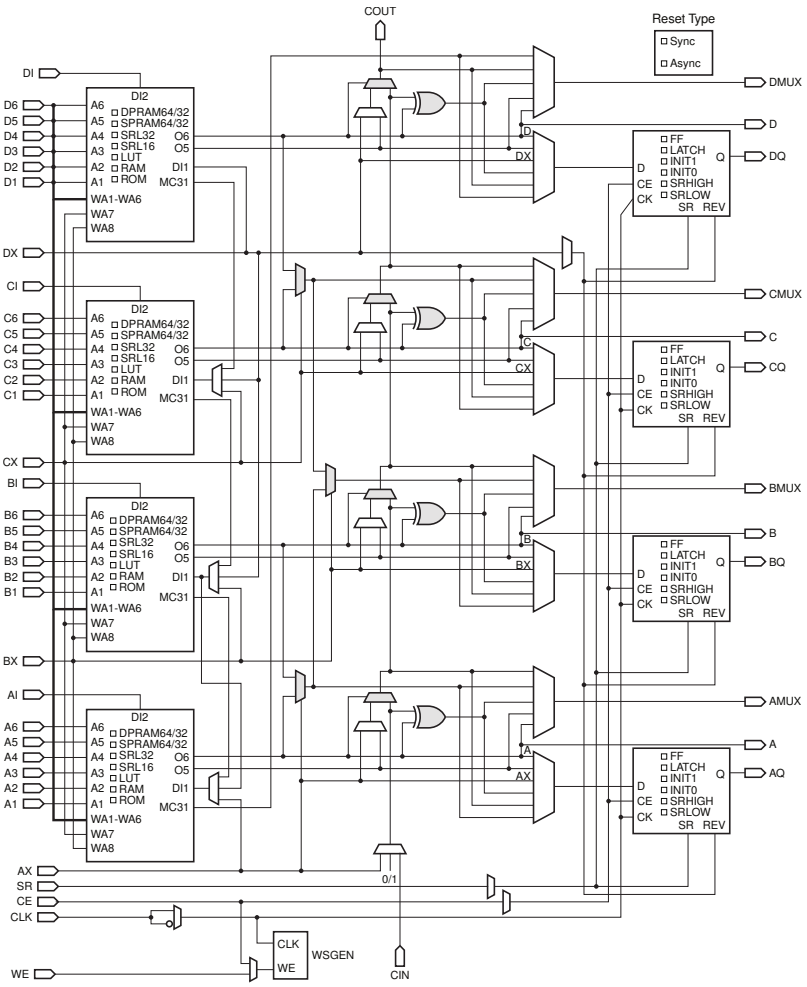


Figure 1.2: Structure of a real FPGA logic block [22]

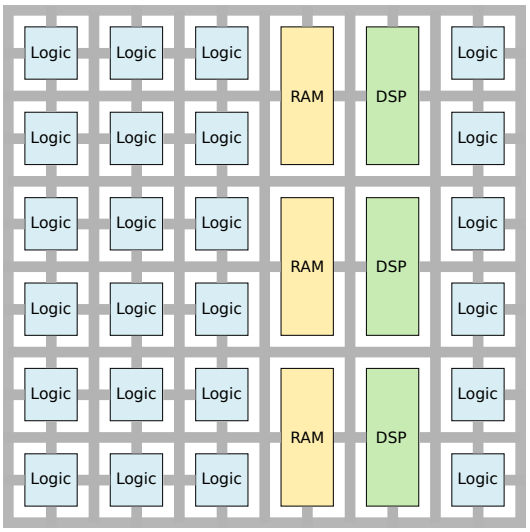


Figure 1.3: Typical structure of a modern FPGA [16]

ments, and lower clock frequencies. A study by Kuon and Rose [15] has shown that, on average, FPGAs need 35 times more chip area, that they are 3.4 times slower, and that they draw 14 times more power than ASICs. For these reasons, modern FPGAs contain *hard blocks*, which implement frequently needed functionality in dedicated silicon circuits. Examples for such hard blocks include *block RAM* (a large, monolithic block of memory, as opposed to memory implemented using the FFs in the logic blocks), digital signal processing (DSP) blocks (containing adders/subtracters and multipliers), complete microprocessors, and circuits which could not (or only partially) be implemented using the digital logic of the FPGA LUTs and FFs (e.g., high-speed serial transceivers, PCI Express endpoints and Ethernet MACs). When such hard circuits are used, the drawbacks mentioned before shrink to 18 times as much chip area, 3 times as much delay, and 7.1 times as much power, compared to ASICs [15].

In order to maximize the applicability of hard blocks and to provide support for a broad range of applications, they are usually designed with a great deal of flexibility. For example, Figure 1.4 shows a diagram of the DSP48E hard block contained in Xilinx Virtex-5 FPGAs. Most of the multiplexers are dynamically controllable; the others (including the bypasses of the registers) can be set before the design is synthesized. Figure 1.5 shows the DSP hard block of Altera Stratix IV FPGAs.

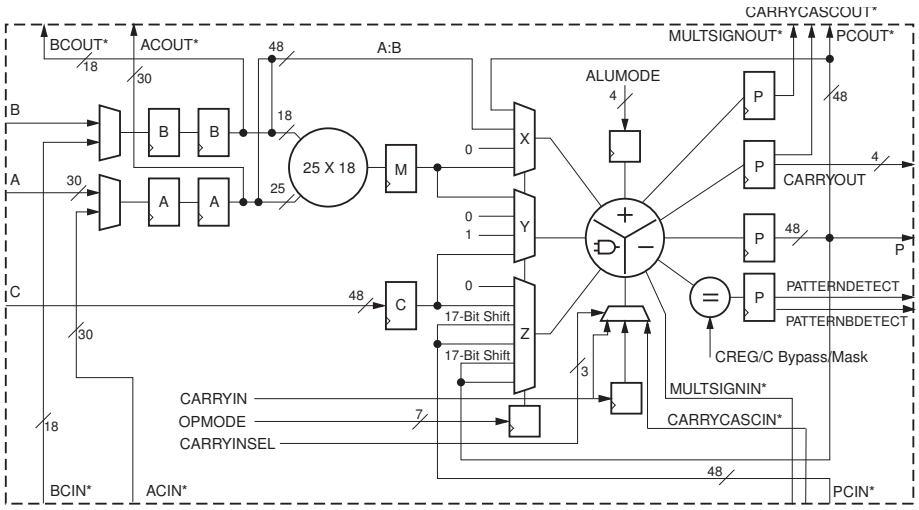


Figure 1.4: The DSP48E hard block from the Virtex-5 family of FPGAs [23]. The signals marked with a star are only accessible to the DSP48E blocks which are directly adjacent. Multiplexers without a select signal are statically selected.

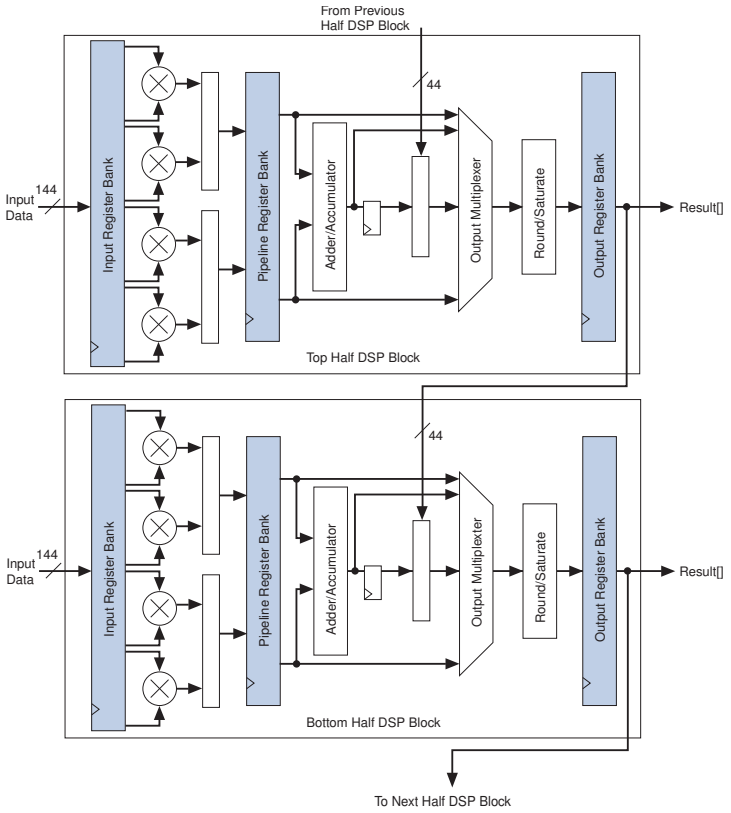


Figure 1.5: The DSP hard block from the Stratix IV family of FPGAs [2]

There are basically three ways of using hard blocks in a circuit design:

- *Manual instantiation*, where the programmer defines the configuration of the hard block and its surrounding logic in order to implement the desired functionality. This approach requires knowledge of the exact behavior of the hard block; it demands the most time and expertise, but it is also the most flexible way to use hard blocks.
- Most hard blocks can be instantiated using a *wizard*: the user selects a certain operation and specifies parameters for it; the wizard then generates wrapper code for one or more hard blocks which implement the desired functionality. For example, a wizard might create a square root function using DSP hard blocks; possible options are the precision of the input and the result, the latency of the circuit, whether it is pipelined or not, etc. This approach is easier to use than manual instantiation, but it is also less flexible. Furthermore, the set of functions provided by a wizard is often sparse compared to all possible functions implementable using a hard block, and usually the programmer has to write code that interfaces with the generated wrapper, because it does not fit perfectly into the design (as opposed to automatically inferred hard blocks).
- Some synthesis tools allow the automatic *inference* of hard blocks from HDL code. Although this is by far the easiest way of using hard blocks, it is also severely limited: typically, only simple operations can be inferred (for example, multiplications or additions of large numbers), but not more complex or atypical ones (like using the FFs in a DSP hard block for memory, thereby saving FFs in logic blocks).

1.1 Structure

This thesis is structured as follows: Chapter 2 lists the goals of my work and the problems which I intend to solve as well as those which I do not intend to solve. This also includes a discussion of related problems and explanations as to why the solutions to them are not applicable in my case.

In Chapter 3, the concepts which are used to solve the stated problems are

explained. It also contains a short introduction to evolutionary algorithms and genetic programming, and lists a selection of other, related techniques.

Chapter 4 describes the representation of circuits as well as the available operators and data types. It also describes the mechanisms of transformation rule application and hard block instantiation in detail. Details about the implementation, including some code excerpts, are given in Chapter 5.

The results of the experimental evaluation of the implementation are described and discussed in Chapter 6. The conclusions drawn from these results are listed in Chapter 7, which also discusses ideas and possible improvements for future work.

Chapter 2

Contributions

In this thesis, I present an algorithm which optimizes a combinational logic circuit by replacing part of the logic with hard blocks. This algorithm iteratively modifies the circuit in a non-deterministic fashion using rewrite rules that are guaranteed to preserve the semantics of the circuit. This approach has two main goals:

- Exploring the design space by performing transformations that are unintuitive to a programmer, thereby opening up new possibilities for optimization and for hard block inference.
- Replacing logic with hard blocks — where possible — in order to reach an optimization objective.

The algorithm is explicitly not intended to solve the following problems:

- General optimizations of the circuit (for example, logic minimization), although some of these optimizations are performed as a side effect.
- Mapping logic to FPGA resources (i.e., partitioning into LUTs, placing and routing the design, etc.).

The algorithm, as well as the implementation that I wrote, is not specific to any manufacturer or model of FPGAs. However, due to my experience and familiarity with Xilinx Virtex FPGAs, I will concentrate on them in this thesis, as well as in the experiments I conducted.

2.1 Previous and related work

To the best of my knowledge, there has not been any previous academic work on automatic inference of hard blocks in FPGAs, which is in strong contrast to the extensive and active research on mapping logic designs to different FPGA architectures. There are, however, other approaches to related problems, namely *rule-based optimization*, *boolean matching*, and *sketching*.

Rule-based optimization The idea of using transformation rules to optimize a circuit is not new. The SOCRATES system [8, 11] optimizes a gate-level circuit by applying low-level rules to it (for example, converting two nested two-input or-gates to a single three-input or-gate). Rules are applied only if they optimize the cost of the circuit, which makes this technique a greedy algorithm; as a consequence, there is not really any exploration of the design space. Furthermore, the low-level nature of the circuit representation makes it virtually impossible to apply high-level rules like arithmetic equivalences, or to infer hard block instances.

Boolean matching Boolean matching [3] solves the problem of mapping a logic circuit to a net of connected instances of cells from a given library (also called *library binding*). This technique consists of solving two subproblems: determining whether a certain part of the circuit can be implemented by a given cell ("matching"), and choosing a cell which optimizes a given objective ("selection"). This approach is not feasible for the problem which this thesis intends to solve:

- The matching of cells is exact, i.e. cells cannot be matched partially in cases where only a subset of the functionality is needed.
- Since the matching process usually tests for semantic, not syntactic, equality, determining whether a cell matches a part of the circuit is computationally expensive, especially since all permutations of inputs have to be considered. This leads to a limit of only a few inputs (up to about 20 boolean inputs in newer approaches [1, 12]), while hard blocks have several dozens or even hundreds of inputs.

In case the matching is done syntactically, the probability of successfully matching a hard block is low because complex operations (like addition or multiplication) have to be broken down into boolean operators, with the result that if the width of the operator in the circuit does not exactly correspond to the width of the operator in the hard block, it can't be matched.

- Boolean matching does not transform the input circuit, even if this would lead to a circuit for which a more optimal selection of cells can be found.

Sketching "Sketching" refers to the concept of writing a partial implementation of an algorithm and automatically synthesizing the rest of it from a specification (which can be in the form of a more abstract, less optimized implementation). It was introduced by Solar-Lezama et al. [18] for bit-streaming programs, and was subsequently refined to support more general, finite programs [19]. Sketching is intended for cases in which the approximate structure of an efficient algorithm is known, but filling in the "gaps" is difficult and error-prone. Sketching does not, however, solve the problem considered in this thesis:

- The programmer still needs to provide the framework of the efficient implementation, and thus has to have some insight into it.
- Sketching, as currently implemented and researched, does not try to find an optimal solution; specifically, it does not try to change the sketch in order to improve it according to an objective.

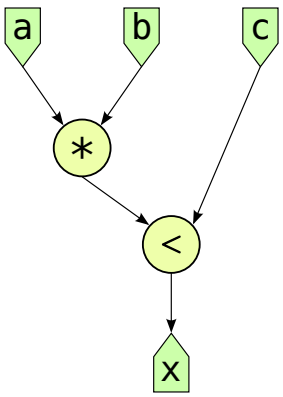
Chapter 3

Conceptual approach

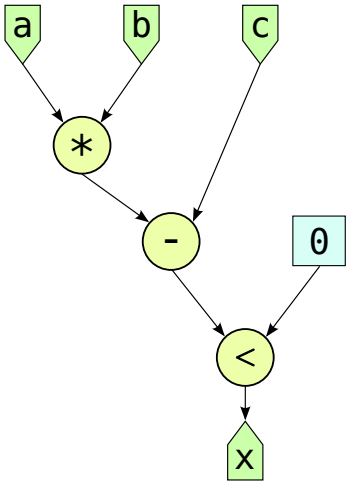
Hard blocks are designed by the manufacturer of the FPGAs and have to satisfy requirements for a broad range of applications and uses. Therefore, a single hard block typically provides complex, programmable functionality. Due to this versatility, it might not always be obvious how the full potential of a hard block can be used. For example, a barrel shifter (a circuit that can shift an input by a variable number of bits) is very costly if implemented in the LUTs of an FPGA. However, a left shift by n bits can also be computed by a multiplication with 2^n, which can be done by a DSP block, and generating the number 2^n is a lot cheaper than a barrel shifter.

Furthermore, transforming the circuit can lead to new possibilities for instantiating hard blocks. For example, consider the circuit in Figure 3.1a: the multiplier in it could be directly replaced by a DSP hard block. However, the circuit can be transformed so that the "<"-comparator is replaced by a test for negativity, which is much more efficient provided that the result of the subtraction is already available (Figure 3.1b). The hard block can replace the multiplier and the newly created subtracter (Figure 3.1c). In fact, some DSP hard blocks even provide a test for negativity, so this might be covered by the hard block as well (Figure 3.1d).

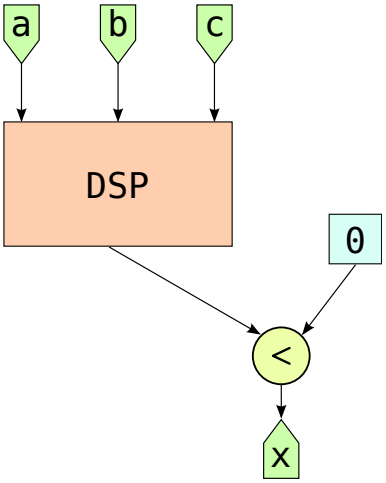
By randomly exploring the design space, such optimizations and counter-intuitive uses for hard blocks can potentially be found automatically. For this thesis, I chose a variant of evolutionary algorithms to transform a circuit so that it can be optimized by instantiating hard blocks in it.



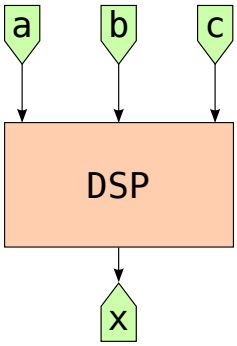
(a) Initial circuit



(b) After transformation using $x < y \equiv x - y < 0$.



(c) Instantiated hard block which replaces the multiplication and the subtraction



(d) Alternative instantiation which additionally replaces the test for negativity

Figure 3.1: Exploring the design space in order to make better use of hard blocks. Control signals for the hard blocks are not shown.

3.1 Evolutionary algorithms

The term *"evolutionary algorithm"* (EA) describes a broad class of non-deterministic, heuristic algorithms used to search for and/or optimize a solution for a given problem. As their name indicates, they are inspired by natural evolution and its fundamental principles [10]: given a population of *individuals* (representing potential solutions) defined by their genes, environmental pressure (representing the problem at hand) leads to natural *selection* among the population and reproduction of individuals, according to their *fitness* (which represents the ability to solve the problem, respectively the quality of the solution). The following influences lead to a series of populations (called "generations"), which should be increasingly well adapted to the environment (i.e., provide better solutions):

Selection determines which individuals are allowed to survive into the next generation, based upon their respective fitness.

Reproduction creates a copy of an individual, usually with some modifications to its genes (see *mutation* and *recombination*). Whether an individual is allowed to reproduce is usually influenced by its fitness.

Mutation introduces random changes into the genes of the offspring when an individual reproduces. This way, genuinely new solutions can be found, exploring the search space.

Recombination combines the genes of two or more individuals in order to create a new one. This allows genes to spread in a population, without that they have to be discovered repeatedly by mutation.

This process is repeated until a solution of sufficient quality has been found (or a given time limit has been reached).

Similar to natural evolution, evolutionary algorithms are strongly influenced by chance: although it is possible to let just the fittest individuals reproduce and survive, it is advantageous to allow survival and reproduction of less fit members of the population (albeit with a lower probability, otherwise no convergence towards better solutions will result). The reason for this is that the fitness function usually contains local maxima, in which a population can be caught if only the fittest individuals are carried over to the next

generation.

Evolutionary algorithms are classified into four categories, depending on the representation of the individuals [10]:

- *Genetic algorithms* (GA): strings over a finite alphabet
- *Evolutionary strategies* (ES): vectors of real numbers
- *Evolutionary programming* (EP): finite state machines
- *Genetic programming* (GP): abstract syntax trees

These names have historical origin; the respective category should be chosen depending on what is an appropriate representation for the problem at hand. For example, if the individuals are fixed programs with variable parameters, then a genetic algorithm is usually a superior choice over genetic programming.

The technique of evolutionary algorithms has been used before in circuit design. For example, genetic algorithms have been used to automatically evolve a circuit on an FPGA that can discriminate between square waves of 1 kHz and 10 kHz frequency [21]. The challenge of the stated goal was that neither a clock nor off-chip components which could measure time in any way were available, so individual circuits had to make use of the combinational and network delays on the FPGA.

The best solutions performed very well, but were also susceptible to changes in environmental temperature and input voltage. Even when the circuit was moved to another area on the chip, its performance dropped, because it was specialized to the variations of the underlying silicon. In fact, some subcircuits were not even connected to the rest of the circuit, but were crucial to the functioning of the design because they influenced it through analog means, e.g., through electro-magnetic inference.

3.2 Genetic programming

Since a directed graph is a natural representation of the individuals considered in this thesis, genetic programming is the obvious choice (note, however,

that not trees, but directed acyclic graphs are used; see Section 4.1). They are easier to manipulate than circuits encoded as strings and their size can change, while GAs usually handle fixed-size strings. Finite state machines (as in EP) are not appropriate, as only combinational circuits are handled.

Due to their varying size, individuals in GP can suffer from *code bloat* [20, 5]: over the course of several generations, they tend to grow if the size itself is not a selection criterion [4]. This problem is implicitly solved in this thesis by considering the area of the circuit as an optimization objective.

3.3 Other optimization techniques

There exist other heuristic algorithms for finding good solutions to optimization problems. Some of these will be listed here, accompanied by an explanation as to why they were not chosen for the problem at hand.

Particle swarm optimization [13, 17] is based on the behavior of animal swarms, like flocks of birds or shoals of fish. Initially, individual solutions, called "particles", are distributed in the search space. Two forces influence each particle: one pulls towards the particle's fittest location encountered so far, the other pulls it towards the fittest location encountered so far by another particle in its neighborhood. The velocity of each particle is determined by these two, randomly weighted, forces. This also means that particles can only move between points which have already been visited by at least one member of the swarm, so the possible search space is determined by the convex hull of the positions of the initial population.

This optimization technique is not suitable for the problem considered in this thesis, because there is no notion of "position" in the search space, and hence not of "direction" and "velocity", so it is not possible to move a circuit towards another circuit. Another minor problem is the creation of the initial distribution: to ensure that the semantics of the input circuit is maintained, the initial distribution would have to be generated by randomly applying transformation rules. It is doubtful that the search space would be thoroughly covered this way.

Simulated annealing [14, 24] is inspired by the technique of annealing in metallurgy. It starts with a given initial solution and a high "temperature", which is successively lowered. In every iteration, the current solution is randomly modified; if the new solution is better, it replaces the old solution; if it is worse, it is accepted with a probability related to the current temperature (higher temperatures allow larger modifications). The algorithm terminates after the temperature reaches zero (or if the solution is of sufficient quality).

Compared to simulated annealing, evolutionary algorithms don't suffer from the problem of recovering from a bad, disadvantageous mutation, since there are other individuals that can provide a good solution. Furthermore, the solution instances that are handled in this thesis are relatively small and mutation is fast, so having only one instance is not an advantage.

Extremal optimization [6, 7] tries to optimize an initial solution by repeatedly replacing the component having the worst fitness value by a randomly selected new component, which in turn affects the fitness values of its surrounding components. This approach, however, is not suitable for the optimization problem considered in this thesis: although the least fit components could be identified (e.g., groups of operators which take up a lot of area, or which introduce long delays), replacing them with another random component is not a good idea, since that would change the semantics of the circuit. Even if this could be avoided, the achievable optimizations would be restricted: after all, a change in another (non-critical) part of the circuit can lead to the removal or improvement of the worst component.

Chapter 4

Detailed approach

A substantial part of this thesis is the development of a circuit representation, which includes the structure as well as available operators and data types. I decided to provide high-level operators like multiplication and comparison in addition to low-level ones like conjunction, disjunction, etc., because they are used relatively often in hardware designs, and DSP hard blocks tend to offer those functions. Although all these high-level operators could be constructed using boolean operators (in fact, a single operator like $\overline{\wedge}$ or $\rightarrow$ would be sufficient), I decided against that solution, since it would make high-level transformations on the arithmetic level virtually impossible.

One the other hand, complex operators like division or exponentiation are not available for reasons of simplicity. Floating-point data types are also not supported, because, for example, $a * b = b * a$ only holds on an abstract level, and a transformation based on that equivalence might actually change the semantics of the circuit.

4.1 Circuit representation

Every circuit is represented using a directed, acyclic graph (*DAG*). The nodes in the DAG represent inputs, outputs, operations (like addition or bitwise conjunction), and instantiated hard blocks. Every node has zero or more inputs and zero or more outputs. This design has been chosen over

a set of trees because hard blocks often have more than one output, which would be difficult to represent using a tree in which every node has at most one parent node. Aside from hard block instances, only the fork node has more than one output, and only the if-then-else operator has more than two inputs. To illustrate this concept, Figure 4.1 shows the DAG for the following circuit:

```
t_1 := a + b;
t_2 := c + d;
x := s[1] & s[0] ? t_2 : t_1;
y := t_1 * t_2;
```

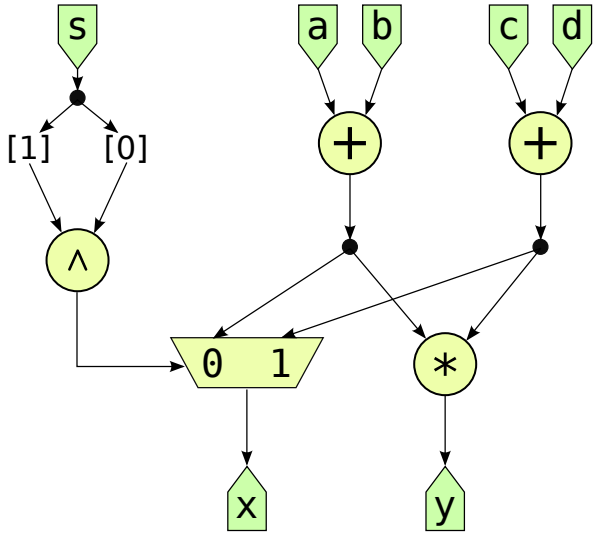


Figure 4.1: Example of circuit represented using a DAG

As an invariant, every input has to be connected to an output, and vice versa; unused outputs of a hard block instance are connected to a "dummy" node. The connection between two nodes is called a "signal".

4.1.1 Data types

Every signal has a certain data type. In most cases, the type of the output signals depends on the types of the input signals (e.g., for arithmetic operators); for some operators (e.g., comparison operators), the output type is fixed. I developed and implemented three concepts of data types, of which

the first two had considerable flaws, necessitating the development of an improved approach.

Concept one

The first concept of data types consists of an uninterpreted bitvector type and two different types for numbers (a signed and an unsigned type). Additionally, every signal has an associated bit-width, which, together with the data type, provides information about the possible range of values of the signal. The typing is strict, so that only numeric input signals are allowed for arithmetic operators, and only bitvector input signals for logic operators (cast operators exist so that signals can be reinterpreted). However, this approach has two disadvantages:

- Inferring the correct output types of arithmetic operators is complicated and error prone, especially since signed and unsigned input types can be mixed.
- The inferred bounds on the possible range of values are not very exact. For example, consider an input a which has a range of $[0, 255]$ and thus requires 8 bits to represent. The negation of a, $-a$, has a range of $[-255, 0]$ and thus requires 9 bits (because it has to be encoded in two's complement). The double negation of a, $-(-a)$ has a range of $[0, 255]$ and actually requires only 8 bits. However, this information is lost under this type system, because $-a$ has a 9 bit data type and thus might hold values between -256 and $+255$ (since a 9 bit two's complement number can contain any integer in $[-256, +255]$), so $-(-a)$ requires 10 bits to represent the values in $[-255, +256]$, because the information that $-(-a)$ can only evaluate to numbers in $[0, 255]$ is lost.

Concept two

The next data type concept unifies the signed and unsigned numeric types, and the possible range of values of a signal is represented using a lower and an upper bound. The bit-width of such a signal is calculated implicitly,

with the assumption that a signal with a minimum value lower than zero is represented as a signed number, and those with a minimum value not lower than zero as an unsigned one. Continuing the example above, $-a$ has the output type `num (-255, 0)` and requires 9 bits; $-(-a)$ has the type `num (0, 255)` and requires 8 bits.

Although this approach solved the two problems listed above, another problem emerged when I began implementing transformation rules which utilized properties of the binary representation of numbers (for example, $a * 2^n \equiv a << n$). Since arithmetic operators allowed only numeric types, this required a lot of cast operators. This problem was even more severe for hard block inference: since they accept only limited bit-widths of inputs, it was often necessary to split the inputs of operators. For example, an addition of two 16-bit numbers can easily be split into two additions for 8-bit numbers (where one addition needs to accept a carry-bit). However, since the addition operator node took only number types as inputs, each input number had to be cast to a bitvector so that the lower and upper halves could be extracted, then they had to be cast back to number types in order to be valid inputs for the new addition operators. In text representation, the result would have looked like this (where `<bv[16] ...>` is a cast to a 16-bit bitvector, and `<uint ...>` is a cast to an unsigned number):

```
// extract lower and upper halves
a_high := <uint (<bv[16] a> [15:8])>;
a_low := <uint (<bv[16] a> [7:0])>;
b_high := <uint (<bv[16] b> [15:8])>;
b_low := <uint (<bv[16] b> [7:0])>;

low_bits := <bv[9] (a_low + b_low)>;  // add lower halves
// add upper halves plus carry from addition of lower halves
high_bits := <bv[9] (a_high + b_high + <uint low_bits[8]> )>;

// concatenate upper and lower result bits
x := <uint (high_bits # low_bits[7:0])>;
```

Although casts are essentially no-ops when the circuit is translated to a hardware description language, writing such a transformation requires more

code, which is an inconvenience and a possible source for programming errors.

Concept three

For this reason, the remaining types are again unified to a single data type, which is now a valid input for every type of operator. Essentially, a signal is defined by a minimum and a maximum value (to accommodate numbers); an uninterpreted bitvector is represented by an unsigned number with the required number of bits. The following examples illustrate this concept:

- A signal which can carry values in the interval $[0, 127]$ is considered unsigned and needs 7 bits when synthesized for hardware.
- The interval $[-32, +25]$ is signed and needs 6 bits.
- The interval $[-27, -19]$ is signed and needs 6 bits as well (5 bits could only represent $[-16, +15]$.
- The interval $[0, 0]$ (a constant 0) is unsigned and needs 0 bits.
- The constant -2 is signed and needs 2 bits.
- A bitvector of length n is represented as unsigned number with the range $[0, 2^n - 1]$.

A single cast operator is still available for rare cases where an unsigned signal has to be reinterpreted as signed, and vice versa.

With this unified data type, the above example looks like this:

```
low_bits := a[7:0] + b[7:0];
high_bits := a[15:8] + b[15:8] + low_bits[8];
x := high_bits # low_bits[7:0];
```

4.1.2 Node types

Nodes can be classified into inputs/outputs/constants, common nodes, arithmetic nodes and bitvector nodes. The operations of arithmetic nodes are

"perfect", meaning that their output is never truncated. The bit-width required to implement an operation is still bounded since the maximum size of the inputs is known. All nodes automatically infer their output data types from their input data types.

Sources and sinks:

- *Inputs* represent the input ports of the circuit; their data type is declared in the description of the input circuit.
- *Outputs* represent the output ports of the circuit; like inputs, they have a fixed data type.
- *Constants* can be any integer or sequence of bits (they are restricted to $[-2^{63}, +2^{63} - 1]$ in this implementation).
- *Dummy sinks* are connected to unused hard block outputs. They exist solely for the purpose of preserving the invariant that there are no unconnected inputs or outputs.

Common nodes:

- *Forks* act as duplication nodes in case that the output of a node is used several times. This is necessary because node outputs can be connected to only one node input.
- *Multiplexers* (also called *if-then-else* operators) take a bitvector of length 1 as condition input, and forward either their first or their second operand, depending on the value of the condition.
- The *equality operator* "$a = b$" compares two inputs for equality. The $a \neq b$ operation is encoded by $\neg(a = b)$.
- A *cast operator* "$<$ `sint` $a >$" respectively "$<$ `uint` $a >$" which reinterprets a signal as signed respectively unsigned.

Arithmetic operators:

- *Comparison* "$a < b$" — $a > b$, $a \leq b$, and $a \geq b$ can be encoded using $<$ and $\neg$.
- *Negation* "$-a$"
- *Addition* "$a + b$"
- *Subtraction* "$a - b$"
- *Multiplication* "$a * b$"

Logical operators:

- *Complement* "$\neg a$"
- *Conjunction* "$a \wedge b$"
- *Disjunction* "$a \vee b$"
- *Exclusive disjunction* "$a \oplus b$"
- *Biconditional* "$a \leftrightarrow b$"
- *Implication* "$a \rightarrow b$"
- *Concatenation* "$a\#b$" appends the second parameter to the end of the first parameter.
- *Extraction* "a[`msb` : `lsb`]" retrieves a sub-bitvector, characterized by the indices of the *most significant* and the *least significant* bits; both index parameters have to be constants.

For bitwise operators (e.g, conjunction and disjunction), only inputs of equal lengths make sense. Therefore, their semantics are defined as follows: the shorter input is either sign- or zero-extended (depending on its data type) to the length of the longer one; the output has the length of the longer input and is always unsigned. Similarly, the output data type of the multiplexer is the smallest data type that contains the value ranges of either operand.

The input of the extraction operator is either sign- or zero-extended in case a sub-bitvector lying partially or completely outside of the input is extracted.

The output data type has the signedness of the input data type if the `msb`-parameter is greater than or equal to the index of the most significant bit of the input, otherwise it is unsigned. For the concatenation operator, the output has the signedness of the left operand. This way, partitioning a signal and re-concatenating the individual subsignals is an identity function. For example, consider the signal a of type $[-128, +127]$. Then $a[7:4] \# a[3:0]$ has the same type and the same value as a itself. Furthermore,

$$a[7:4] * 2^4 + a[3:0] = a$$

holds, which is an important property when splitting inputs of an operator for hard block instantiation.

4.2 Input format

The input circuit is read from a text file which defines the header (in- and outputs), the area- and delay-constraints, and the body of the circuit.

The header defines the name of the inputs and outputs, as well as their respective data types. For example,

```
a: bv[5], b: int(-7 .. 30) => x: bool, y: int(0 .. 255);
```

declares an input, `a`, which is a bitvector of length 5, and `b`, which can take on numbers between 7 and 30. The outputs are `x`, which is a bitvector of length 1, and `y`, which is a number between 0 and 255. The following data types are available:

- `int`(low .. $high$) declares a signal type with range $[low, high]$
- `uint`$[n]$, with $n \geq 0$, is a shortcut for `int`$(0 .. 2^n - 1)$
- `sint`$[n]$, with $n > 0$, is a shortcut for `int`$(-2^{n-1} .. 2^{n-1} - 1)$
- `bv`$[n]$ is an alias for `uint`$[n]$
- `bool` is an alias for `bv`$[1]$

The constraints section defines the bounds on area and input-output delays. Example:

```
Area: 20;
Delay:
    a => c: 1.5;
    a => d: 3;
    b => c: 2.4;
```

These constraints set the maximum area used by the circuit to 20 LUTs, the delay from input `a` to output `c` to 1.5 nanoseconds, the delay from `a` to `d` to 3 nanoseconds, etc. Note that not all input-output pairs have to be constrained. In fact, both area and delay constraints are optional.

The body defines the actual circuit through assignments of expressions to outputs or local signals. Example:

```
temp := 2 * <uint a> + 5;
c := b < temp;
d := temp - (a[4] ? 2 : 3);
```

Here, a local signal `temp` is implicitly declared and used in the expressions for `c` and `d`. It is an error

- to assign an expression to a local signal or an output more than once
- to assign an expression to an input
- to use an output in an expression
- to leave an output unassigned
- to introduce a static cycle into the circuit

Most operators map directly to their corresponding node types. There are some exceptions, however. For example, the fork operator is inserted whenever a local signal or an input is referred to more than once. Furthermore, operators like "$\geq$" or "$\neq$", for which no direct node type exists, are available.

4.3 Selection and mutation

Selection is an important part of evolutionary algorithms: by favoring "fitter" individuals over less fit ones, the population (or at least some of its members) should gradually improve in optimality. The algorithm employs a simple selection scheme: all individuals of the current generation are sorted by their fitness, so that the least fit circuit is at index 1, and the fittest one is at index n, where n is the size of the current generation. The probability that a given individual with index i survives is equal to $\frac{i}{n}$.

An individual circuit is mutated by randomly applying transformation rules to a part of it. Like selection, the probability that a circuit is mutated (i.e., that it reproduces) is equal to $\frac{i}{n}$. Note that it is possible that an individual reproduces without surviving, and that it survives without reproducing (in this case, it is not mutated). It is also possible that an individual survives *and* reproduces: in this case, only its offspring is mutated. The fittest member of a generation is always guaranteed to survive and to reproduce — this ensures that the output is always at least as good as the input.

Given a population of size n, the expected size n' of the next generation is

$$n' = \sum_{1}^{n} \frac{i}{n} + \sum_{1}^{n} \frac{i}{n} \tag{4.1}$$

$$= 2\frac{1}{2n} n(n+1) \tag{4.2}$$

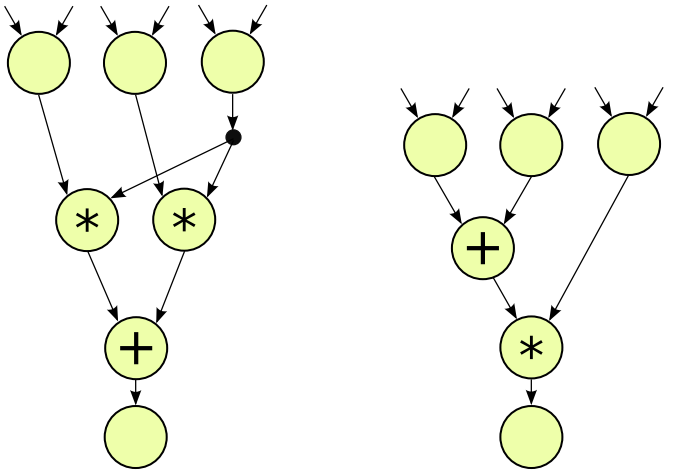
$$= n + 1 \tag{4.3}$$

so the population tends to grow continuously. To counter this effect, the least fit 10 % of a population are removed whenever its size grows to more than 125 % of its desired size.

Recombination is not implemented. The reason for this is that simply replacing one part of a circuit with another one can lead to different semantics of the circuit, which must be avoided.

4.3.1 Transformation rules

Every rule has a target node, which can be interpreted as its "pivot point". When a DAG is mutated, a random node is selected and a random, potential rule is applied to the DAG, using the selected node as pivot. For example, the `DistributivityRule2` has a `MultiplicationOp` as its pivot, so this distributivity rule is a potential rule for every multiplication node in the DAG (each node type has a list of potential rules which can be applied to it). After a node and a rule have been selected, the rule checks whether it really matches that part of the DAG, and, if this is the case, applies itself to the DAG. Figure 4.2 shows such an application of a transformation rule.



(a) Circuit before rule application

(b) Circuit after applying rule $(a + c) * (b + c) \Longrightarrow (a + b) * c$

Figure 4.2: Example of a transformation rule application

Generally, rules have those nodes as pivot which are expected to occur in a circuit the least often. For example, multiplications are rarer than additions, so a rule including an addition operator and a multiplication operator should have the latter as its pivot, so that there are fewer unsuccessful rule applications.

Initially, the two sides of a rule were represented by small circuit DAGs. Rule application was handled by a single method which performed the matching of the rule nodes and connections with a part of the circuit (starting from the pivot node), thereby mapping the inputs of the rule DAG to nodes of

the circuit. This information was then used to remove the nodes covered by the rule's left side and copy the rule's right side nodes into the circuit.

This approach works for simple rules like symmetry, associativity or distributivity. However, some rules require a more complex treatment (e.g., moving a multiplexer across an operator), or require some computation (e.g., transforming a multiplication with a power of two to a shift operation). Furthermore, simple rules don't profit a lot from the automation provided by this approach. For these reason, the concept behind transformation rules was changed to hard-coded rules, i.e., the code which checks whether the rule can be applied and which transforms the DAG is written by hand. This has even one more advantage: the matching of rules is more flexible. For example, a single hard-coded rule could test for $a * 0$ and for $0 * a$ — the original approach would need two rules for that — which in turn increases the frequency of unsuccessfully tests.

Each rule is defined in a class which implements a method

```
TestAndApply(pivot, dag);
```

When a node and a potential rule is selected, this method is called and expected to apply itself to the DAG if it matches. This process takes place in two phases:

1. The method checks whether the rule matches the given part of the DAG, starting from the pivot node. In this phase, it also finds the nodes which the rule connects to. If the rule does not match the DAG, the method returns without modifying the circuit.

2. The nodes covered by the rules are removed from the circuit, new nodes are instantiated, added to the circuit, and connected with each other and with the surrounding nodes. Finally, if necessary, the output types of the connected nodes are updated.

Although transformation rules are hard-coded, they are modular and not firmly integrated with the algorithm itself, so they could be dynamically loaded from a library at runtime with only very minor changes to the implementation. Also, most rules are written in a generic way and can be instantiated in a single line of code for several operator types.

Some equivalences are not completely available as rules. For example, there is a rule for transforming $a*0$ to 0, but the other direction is not implemented yet. Future versions of the algorithm could explore a larger part of the design space by providing such rules. The available rules are listed in the following; the arrows indicate the implemented directions. Together, they represent more than 50 individual rules. Some of these rules are illustrated in Figure 3.1.

$(a \circ b) \bullet c \Longleftrightarrow (a \bullet c) \circ (b \bullet c)$ — for $\circ \in \{+, \wedge, \vee\}$ and $\bullet \in \{*, \vee, \wedge\}$, resp. (`DistributivityRule`)

$(a \circ b) \circ c \Longleftrightarrow a \circ (b \circ c)$ — for $\circ \in \{+, *, \wedge, \vee, \oplus, \leftrightarrow\}$ (`InputAssociativityRule`); a similar rule exists for the output associativity of fork nodes

$(a \circ b) \Longleftrightarrow (b \circ a)$ — for $\circ \in \{+, *, \wedge, \vee, \oplus, \leftrightarrow\}$ (`InputSymmetryRule`); a similar rule exists for the output symmetry of fork nodes

$a \circ \gamma \Longrightarrow a$ — for $\circ \in \{+, *, \wedge, \vee, \oplus, \leftrightarrow\}$ and $\gamma \in \{0, 1, \texttt{true}, \texttt{false}\}$, resp. (`NeutralityRule`)

$a \circ \gamma \Longrightarrow \bullet a$ — for $\circ \in \{*, \oplus, \leftrightarrow\}$, $\gamma \in \{-1, \texttt{true}, \texttt{false}\}$, and $\bullet \in \{-, \neg\}$, resp. (`NegativeNeutralityRule`)

$\circ(\circ a) \Longrightarrow a$ — for $\circ \in \{-, \neg\}$ (`DoubleNegationRule`)

$a \circ a \Longrightarrow \gamma$ — for $\circ \in \{-, \oplus, \leftrightarrow\}$ and $\gamma \in \{0, \texttt{false}, \texttt{true}\}$, resp. (`CancelationRule`)

$a \circ (\bullet a) \Longrightarrow \gamma$ — for $\circ \in \{+, \wedge, \vee\}$, $\bullet \in \{-, \neg\}$, and $\gamma \in \{0, \texttt{false}, \texttt{true}\}$, resp. (`NegativeCancelationRule`)

$a \circ \gamma \Longrightarrow \gamma$ — for $\circ \in \{*, \vee, \wedge\}$ and $\gamma \in \{0, \texttt{true}, \texttt{false}\}$, resp. (`AbsorptionRule`)

$\circ(x_1, \ldots, x_n), \circ(x_1, \ldots, x_n) \Longrightarrow \circ(x_1, \ldots, x_n)$ — for any operator $\circ$; this rule finds two equal operators with equal arguments, removes one instance of them, and forks the outputs of the remaining node (`EqualSubtermEliminationRule`)

$a \circ a \Longrightarrow a$ — for $\circ \in \{\vee, \wedge\}$ (`IdempotenceRule`)

$a \to b \Longleftrightarrow \neg a \vee b$ — (`ImplicationOrRule`)

$a \circ b \Longleftrightarrow \neg(\neg a \bullet \neg b)$ — for $\circ \in \{\vee, \wedge\}$ and $\bullet \in \{\wedge, \vee\}$, resp. (`DeMorganRule`)

$a * \gamma \Longrightarrow a << n + a * \gamma'$ — this rule takes the multiplication of an argument a and a constant γ and replaces it with the addition of a left-shifted by n bits, and a multiplied with a new constant γ', where n is the zero-based index of the highest bit set in γ, and γ' is computed from this constant by setting the highest bit to 0; example: $a * 10 \equiv a << 3 + a * 2$ (`MultToShiftRule`)

$s \ ? \ a : b \Longrightarrow \neg s \ ? \ b : a$ — swaps the arguments of a multiplexer and adjusts its condition (`ITESwapArgumentsRule`)

$s \ ? \ (t \ ? \ a : b) : c \Longrightarrow s \wedge t \ ? \ a : (s \ ? \ b : c)$ — this rule changes the order of two multiplexers; this is useful for terms like $s \ ? \ (t \ ? \ a : 0) : 0$, because, together with other rules, it can optimize away the second multiplexer (`ITEAssociativityRule`)

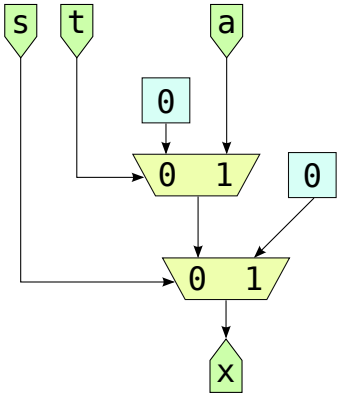
$\gamma \ ? \ a : b \Longrightarrow a/b$ — for $\gamma \in \{0, 1\}$ removes the multiplexer if the condition argument is a constant (`ITEConstantConditionRule`)

$s \ ? \ a : a \Longrightarrow a$ — removes the multiplexer if its inputs are identical (`ITEEqualArgumentsRule`)

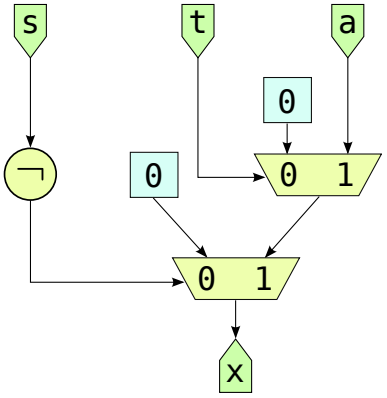
$s \ ? \ (a \circ b) : (c \circ d) \Longrightarrow (s \ ? \ a : c) \circ (s \ ? \ b : d)$ — for any operator $\circ$, including hard blocks; this rule moves a given multiplexer over its input nodes, provided that they are of the same kind; this is especially useful for hard block multiplexing (`MoveITEUpRule`)

4.4 Hard block instantiation

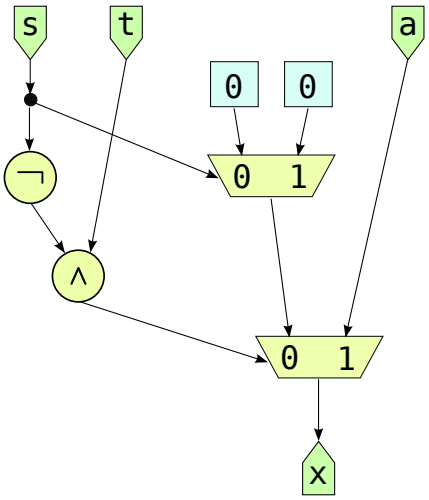
My first approach to automatic hard block instantiation by the algorithm was to have the program read in a library of hard block descriptions, each in the same textual form as the input circuit, together with input-output delay information. The algorithm would then try to match a random hard block at a random place in the circuit (similar to transformation rule application) and, if it fits, it would replace the corresponding nodes with the hard block instance.



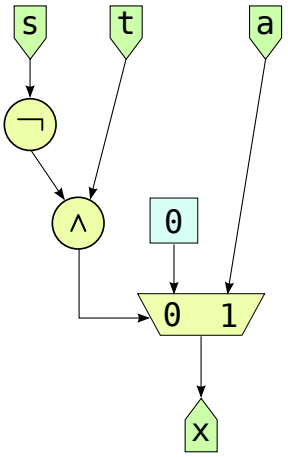
(a) Initial circuit



(b) After application of `ITESwapArgumentsRule`



(c) After application of `ITEAssociativityRule`



(d) After application of `ITEEqualArgumentsRule`

Figure 4.3: Successive application of three different rules

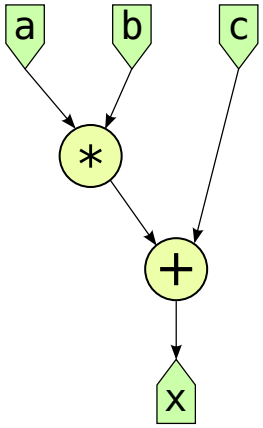
As explained above, hard blocks virtually never completely match a part of a circuit, due to their complexity. Therefore, the circuit would have to be modified to accommodate every node and connection of the hard block. Figure 4.4 illustrates this approach.

Although this procedure bears some resemblance to unification, there are major differences:

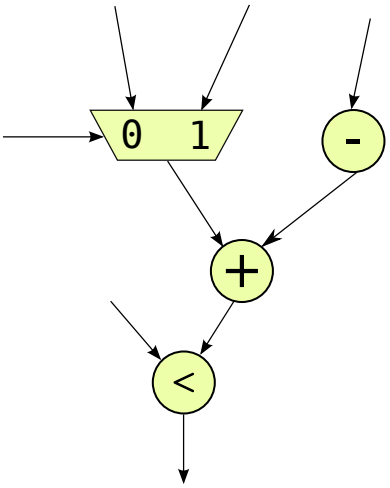
- Circuits are represented as directed, acyclic graphs, while unification (typically) operates on trees.
- Unification does not modify the terms to be unified, it merely replaces variables (which correspond to inputs of the circuit representing the hard block). As pointed out above, it is necessary to modify the circuit for hard block instantiation.
- Unification unifies "top-level" terms, while hard blocks can be instantiated anywhere in a circuit, not just at its outputs.

Unfortunately, devising a way to transform the circuit so that the hard block matches it proved to be more difficult than expected. Upon further inspection, three more arguments against this approach turned up:

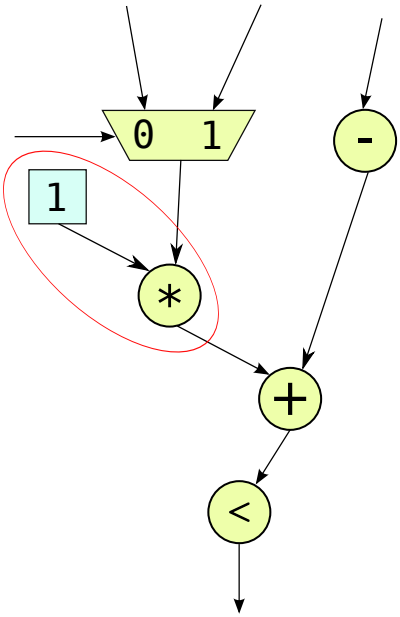
- Hard blocks tend to have constraints regarding their functionality (for example, if a certain submodule is used, a certain other one cannot be used). Modeling these constraints in a DAG is either hard or outright impossible.
- When nodes of the circuit are replaced by a hard block instance, this instance might require configuration data which is evaluated from the nodes it replaces. For example, if block ROM is used as a large look-up table in order to replace logic, the contents of the ROM have to be computed.
- An FPGA provides only a very limited set of different hard block components which are able to replace digital logic. For example, the Xilinx Virtex family of FPGAs only contain a single type of hard block with combinational functionality (the DSP blocks). Hence, supporting the flexibility of this approach is probably not worth its complexity and effort.



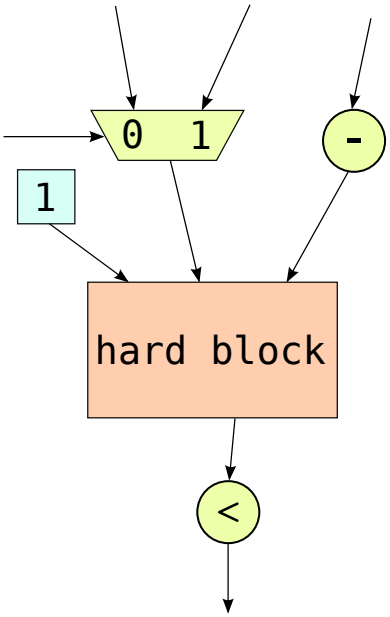
(a) A very simple hard block (multiply-add)



(b) The algorithm tries to match the hard block onto the circuit, with the addition operator as pivot.



(c) The circuit has to be transformed (red oval), so that the hard block structure matches.



(d) The hard block can be instantiated now.

Figure 4.4: Original approach to inferring and instantiating hard blocks.

For those reasons, I decided to take another approach: similar to transformation rules, the code for matching a hard block has to be written by hand for each individual type. This turned out to work surprisingly well, and to require less effort than expected. For example, to match a multiplier for the DSP48E hard block, only about 25 lines of straightforward code are required:

1. Constrain the inputs of the multiplication operator to 25 respectively 18 bits.
2. Instantiate the hard block and remove the multiplication operator.
3. Create constants that select the correct functions in the hard block.
4. Connect all new nodes.

In effect, matching a hard block works very much like rule application, so both share the same interface. This also means that hard blocks can be dynamically loaded at runtime.

There is one difference to transformation rules, though: the widths of the operators are bounded, so if the inputs of an operator to be matched are wider than the operator in the hard block, then the operator in the circuit has to be "split". Fortunately, writing a method which splits an operator takes only about one hour (including devising a formula, testing and debugging), and it is independent of any specific hard block, so it can be reused.

Figure 4.5 shows the circuit which results from splitting $x := a * b$, where a is a 32-bit signed input and b is an 18-bit signed input; the hard block multiplier to be matched takes one 25-bit and one 18-bit signed input. The split utilizes the following equivalences:

$$\begin{aligned} a * b &= \langle a_{n-1}, \ldots, a_0 \rangle_{2c} * b \\ &= (\langle a_{n-1}, \ldots, a_{k-1} \rangle_{2c} * 2^{k-1} + \langle 0, a_{k-2}, \ldots, a_0 \rangle_{2c}) * b \\ &= \langle a_{n-1}, \ldots, a_{k-1} \rangle_{2c} * 2^{k-1} * b + \langle 0, a_{k-2}, \ldots, a_0 \rangle_{2c} * b \end{aligned}$$

where n is the bit-width of a, k is the bit-width of the first input of the hard block multiplier, and $\langle \ldots \rangle_{2c}$ denotes the two's-complement interpretation of a bitvector. Since the left operand of the addition is shifted right by k bits,

a further optimization is possible: let $t := \langle 0, a_{k-2}, \ldots, a_0 \rangle_{2c} * b$, then

$$\begin{aligned} a * b &= \ldots \\ &= (\langle a_{n-1}, \ldots, a_{k-1} \rangle_{2c} * b + \langle t_{k-1+m}, \ldots, t_{k-1} \rangle_{2c}) \ \# \ \langle t_{k-2}, \ldots, t_0 \rangle \end{aligned}$$

where m is the bit-width of b and $\#$ is the concatenation operator.

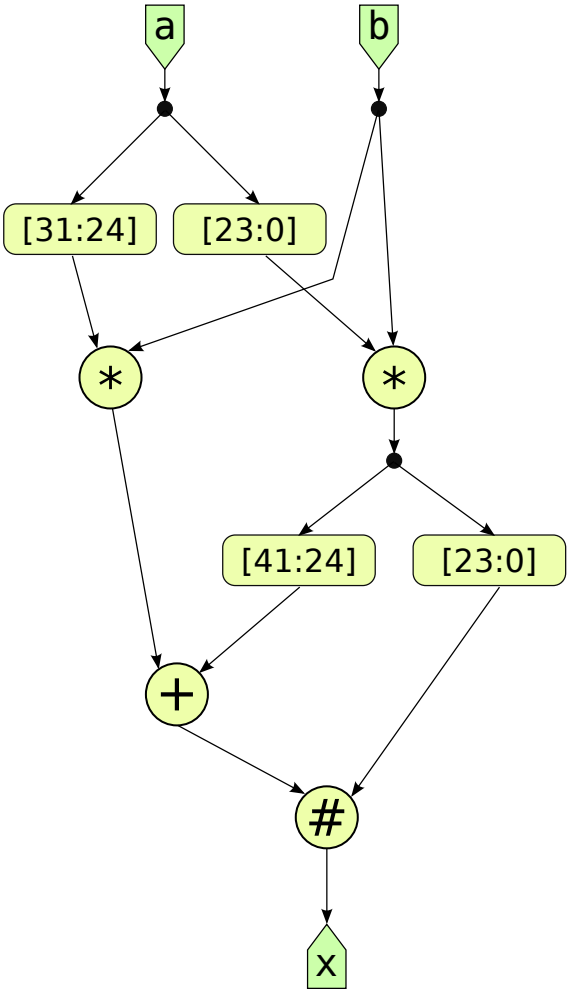
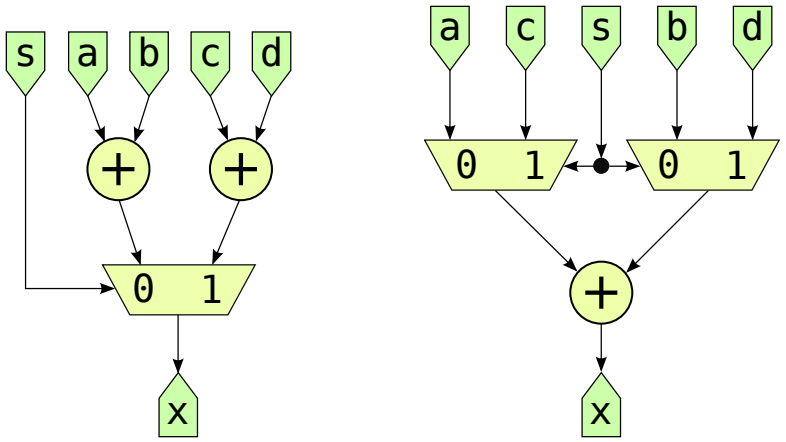


Figure 4.5: Splitting the first input of a signed 32x18 multiplication for a 25x18 signed multiplier.

4.5 Hard block multiplexing

Besides inferring hard blocks, the algorithm can also multiplex a hard block for two operations, provided that it can determine that the two results are never used at the same time. For example, assume that an FPGA provides a hard block which either adds or multiplies its data inputs, depending on a control input, and that the circuit contains a subcircuit of the form `s ? a + b : c * d` at some place. Then the algorithm is able to move both arithmetic operators into one hard block and multiplex its inputs.

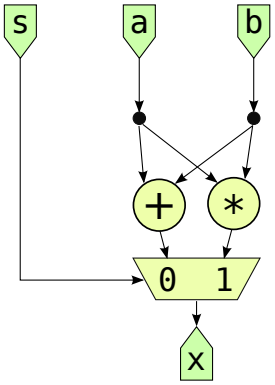
To the best of my knowledge, there exists no synthesis software that is able to do this automatically, even if the source contains the above line. The crucial part is that this ability is not built into the code that instantiates a hard block; instead, it is a side effect of a transformation rule (Figure 4.6) and therefore reusable for other hard blocks as well. Figure 4.7 illustrates the steps taken by the algorithm.



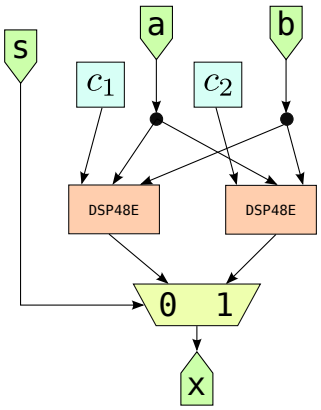
(a) Before...

(b) ...and after moving the multiplexer across the operator.

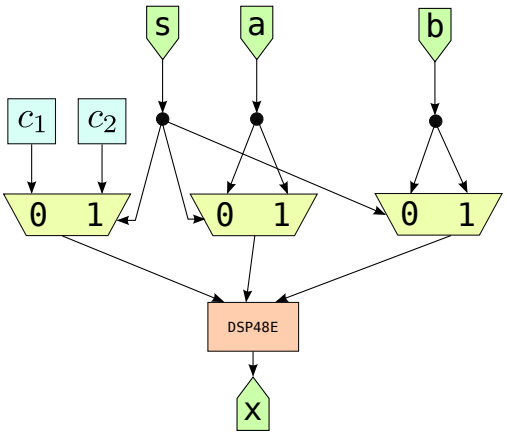
Figure 4.6: Moving a multiplexer across an operator



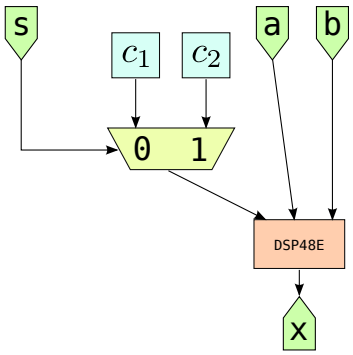
(a) Original circuit.



(b) Instantiated hard blocks for addition and multiplication. The constants c_1 and c_2 represent the operation modes for the DSP elements.



(c) Moved multiplexers across hard blocks.



(d) Final result after further optimizations.

Figure 4.7: Using hard blocks to implement a simple ALU

Chapter 5

Implementation

The algorithm has been implemented in C# for Mono 2.6 and consists of about 6500 lines of code. The implementation is divided into two parts:

- The *library*, which contains all the logic of the algorithm, including DAG representation, transformation rules, fitness evaluation, and evolutionary processing.
- The *interface*, which contains the parser and printer, as well as code to control the library.

The library is independent of the interface and can be accessed by other programs without the detour over parsing and printing the circuit.

5.1 Nodes

All node types are derived from `Nodes.Node`. This class provides the fundamental functionality of a node: the input- and output-ports. Every port is represented by a `Node` $\times$ `uint` pair: the first component specifies the connected node; the second component specifies the index of the port. Whether the (`node`, `port`) pair of a port refers to an input or an output port depends on whether the port itself is an input or an output port (as explained above, input ports are always connected to output ports, and vice versa). The number of ports depends on the node type; they are stored in two separate

arrays, one for inputs and one for outputs.

A node also stores the data types of each output port; the data type of an input port is not explicitly stored, since it is the same as the data type of the output port to which the input port is connected.

5.2 Rules

Every rule is implemented in a class derived from `Rules.Rule` and provides a method

```
bool TestAndApply (Nodes.Node target, Common.DAG dag);
```

where `target` is the node to be matched with the pivot of the rule, and `dag` is the circuit to be modified. The method first tests whether the rule can be applied and returns `false` if that is not the case. Otherwise it modifies the circuit by applying the transformation and returns `true`. In this section, some of the implemented rules are shown.

InputSymmetryRule This is one of the most basic rules: it uses the commutativity property of some operators. Since rules are always called with a suitable pivot node, this rule never fails (for example, this rule is never called with a subtraction operator as pivot).

```
public override bool TestAndApply (Nodes.Node target,
                                   Common.DAG dag)
{
    var x = target.Inputs[0];
    var y = target.Inputs[1];
    x.ConnectTo (target, 1);
    y.ConnectTo (target, 0);
    return true;
}
```

The first two lines get the source ports of the two operands of the node. These ports are then connected to the respective other input port of the target operator (port indices are zero-based).

InputAssociativityRule1 This rule can be used for operators with the associativity property, it transforms

$$(a \circ b) \underline{\circ}\ c$$

into

$$a \circ (b \circ c)$$

The pivot is the outermost operator (marked with an underscore). Unlike the `InputSymmetryRule`, this rule has to check whether the other operator has the same type. Therefore, it has a generic parameter `T`, which is set to the respective node type when the rule is instantiated.

```
public override bool TestAndApply (Nodes.Node target,
                                   Common.DAG dag)
{
    var op2_node = target;
    var op1_node = op2_node.Inputs[0].Node;

    // Test whether the other node has the correct type
    if (!(op1_node is T))
        return false;

    var in_b = op1_node.Inputs[1];
    var out_x = op2_node.Outputs[0];

    // Reuse both operators, but reconnect them
    in_b.ConnectTo (op2_node, 0);
    op2_node.ConnectTo (0, op1_node, 1);
    op1_node.ConnectTo (0, out_x.Node, out_x.Port);

    // Recursively ensure that all nodes have
    // the right output types
    op2_node.UpdateOutputTypes (true);

    return true;
}
```

This rule also has to make sure that the two operator nodes, as well as all nodes in their output paths, have the correct output types. This is done by a call to `UpdateOutputTypes(bool recursive)`; the parameter specifies whether the update should be restricted to the node (`false`) or whether it should recursively update all output types on the path (`true`).

DoubleNegationRule This rule transforms

$$\circ\ (\underline{\circ}\ a)$$

into

$$a$$

Like the associativity rule discussed above, this rule has a generic parameter `T` that is set to the respective node type. Both operator nodes are removed from the circuit.

```
public override bool TestAndApply (Nodes.Node target,
                                   Common.DAG dag)
{
    var other_op = target.Outputs[0].Node;
    if (!(other_op is T))
        return false;

    var a = target.Inputs[0];
    var x = other_op.Outputs[0];

    // Remove both operator nodes from the DAG
    dag.Nodes.Remove (target);
    dag.Nodes.Remove (other_op);

    a.ConnectTo (x);

    return true;
}
```

Updating the output types is not required, since this rule is only used for the arithmetic negation (no update is required due to the type system) and bitwise complement operators (no update is required because the bitwise complement has the same output types as its input).

5.3 Hard blocks

For each hard block, several classes have to be provided: one which represents an instance of the hard block (derived from `Nodes.HardBlock`), and one class (derived from `Rules.Rule`) for each instantiation rule. For this thesis, I implemented the DSP48E hard block mentioned above. The instantiation rule which has the multiplier as pivot is shown below.

```
public override bool TestAndApply (Node target,
                                   Common.DAG dag)
{
    var mult = (MultiplicationOp) target;

    // Split the multiplication operator to 25 bits
    // in the first operand, resp. 18 bits in the
    // second operand (if necessary)
    mult = ConstrainMultInput (mult, 0, 25, dag);
    mult = ConstrainMultInput (mult, 1, 18, dag);

    // Create a new instance of this hard block
    var instance = new Instance ();
    dag.Nodes.Add (instance);

    // Replace the multiplier node in the circuit
    // with the hard block
    mult.Inputs[0].ConnectTo (instance, 0);
    mult.Inputs[1].ConnectTo (instance, 1);
    instance.ConnectTo (0, mult.Outputs[0]);

    // Input C is connected to a constant 0
    var c = new Constant (0);
```

```
    dag.Nodes.Add (c);
    c.ConnectTo (0, instance, 2);

    // Connect control inputs of the hard block
    var opmode = new Constant (5);
    dag.Nodes.Add (opmode);
    opmode.ConnectTo (0, instance, 3);

    var alumode = new Constant (0);
    dag.Nodes.Add (alumode);
    alumode.ConnectTo (0, instance, 4);

    // The CARRYOUT output is unused
    var dummy = new DummySink ();
    dag.Nodes.Add (dummy);
    instance.ConnectTo (1, dummy, 0);

    // The hard block gets the same output types as
    // the multiplication operator which it replaces
    instance.OutputTypes[0] = mult.OutputTypes[0];
    instance.OutputTypes[1] =
        new Optimizer.Common.SignalType (0);

    dag.Nodes.Remove (mult);
    return true;
}
```

Note that this rule only uses the multiplier of the hard block.

Chapter 6

Experimental results

The following circuit is used to test the algorithm's ability to multiplex hard blocks. It represents a simplified ALU, with four operations which are selected by a control signal.

```
a: sint[16], b: sint[16], s: bv[2] => x: sint[32];

x := s[0] ? (s[1] ? a + b : a * b) : (s[1] ? a & b : a | b);
```

It has three inputs: the data inputs a and b, and a two-bit control input s. Depending on the value of s, the data inputs are either added, multiplied, or combined by conjunction or disjunction. The output x is 32 bit wide so that the complete results of all operations fit into it. For reasons of simplicity, no delay constraints were given. The intuitively optimized version of the circuit would look like this:

```
argA := s[0] & !s[1] ? b : 0;
argC := s[0] & !s[1] ? 0 : b;
op := s[0] ? (s[1] ? 15 : 5) : (s[1] ? 51 : 59);
alu := (s[0] ? 0 : 12);

// CARRYOUT is unused because all results are < 48 bits
DSP48E (.A (argA), .B (a), .C (argC), .OPMODE (op),
        .ALUMODE (alu) => .P (result), .CARRYOUT (unused));
```

```
x := result;
```

This benchmark poses a difficult challenge for the algorithm:

- Before all operators can be implemented by one hard block, they have to be replaced individually, so that there exist several hard block instances at the same time, which has a negative effect on the fitness value of the circuit.
- Since the addition and bitwise operators (in this case, | and &) are cheap, the difference in area between using them and using a single hard block with multiplexed inputs is small.
- The chosen DSP block has three main inputs: inputs A and B are used for operands of the multiplier, while $A\#B$ (the concatenation of A and B) and C are used for operands of the adder respectively of the logic module. Hence, the algorithm has to utilize the symmetry property of multiplication, so that operand a can be mapped to input B for all hard block instances, thereby removing the need for a multiplexer for input B.

These complications are problematic: if the fitness weight of a hard block is set too high, replacing the addition and bitwise operators will cause the circuit to be removed from the population with a high probability. If, however, the weight is set too low, the multiple hard block instances will not be combined to one, because the cost of the input multiplexers is higher than the cost of having several hard blocks. If a middle ground for the weight is chosen, only the multiplication and the addition are replaced and combined.

The solution to this dilemma is to have two phases with different weights for hard blocks. In the first phase, with a low weight, hard blocks are instantiated. The second phase assigns a high weight to them, which causes multiple hard blocks to be combined to a single one.

Additionally, it is necessary to run a couple of random mutations on the circuits before hard blocks are instantiated, so that a larger part of the design space is explored. This leads to better results since there is no symmetry transformation rule available for hard block inputs, and, as explained above, exploiting the symmetric property of the operators is a precondition for

saving input multiplexers in this case.

Through experimentation it turned out that it is advantageous to perform multiple mutation steps on an individual circuit before its fitness is determined and it is potentially removed from the population. I tested several combinations of numbers of mutation steps for each phase to find out which values give the best results (Figure 6.1). It can be seen that the smallest circuits are achieved for $n_1 \in \{1, 3, 6, 10\}$, and that the values for n_2 have comparatively little influence, especially for $n_1 = 10$.

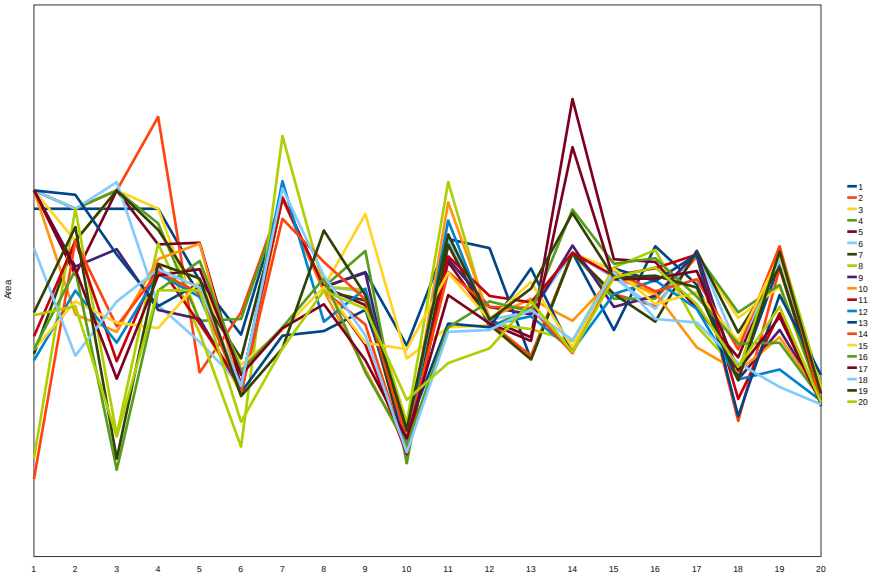


Figure 6.1: Comparison of numbers of mutation steps. The x-axis shows n_1 (the number of mutation steps for phase 1: hard block instantiation); the individual lines denote n_2 (the numbers for phase 2: hard block multiplexing). The y-axis shows the resulting area of the best circuit.

The best solution found by the algorithm looks as follows:

```
x := local_7;
local_0 := (s[1:1]);
local_1 := (s[0:0]);
local_2 := !local_0;
local_3 := (local_1 ? (local_2 ? b : 0) : 0);
local_4 := ((local_1 & local_2) ? 0 : b);
local_5 := (local_0 ? (local_1 ? 15 : 51)
```

```
                    : (local_1 ? 5 : 59));
local_6 := (local_1 ? 0 : 12);
DSP48E (.A (local_3), .B (a), .C (local_4),
        .OPMODE (local_5), .ALUMODE (local_6)
        => .P (local_7), .CARRYOUT (local_8));
```

One optimization has been missed: the expression for `local_3` (i.e., the A input) can be optimized to

```
local_3 := (local_1 & local_2) ? b : 0;
```

(thereby saving a 16-bit multiplexer), so it is inferior to the handwritten version.

Unfortunately, even slight changes to the circuit (e.g., `s[1] ? b*a : a+b`) result in different optimal values for n_1 and n_2 (Figure 6.2). The simplest, albeit inelegant, solution is to optimize an input circuit with all 400 combinations of n_1 and n_2, and pick the best result. The total runtime using this method is about 8 minutes.

The algorithm achieves better results if the multiplication in the input circuit is replaced by another operator. Consider, for example:

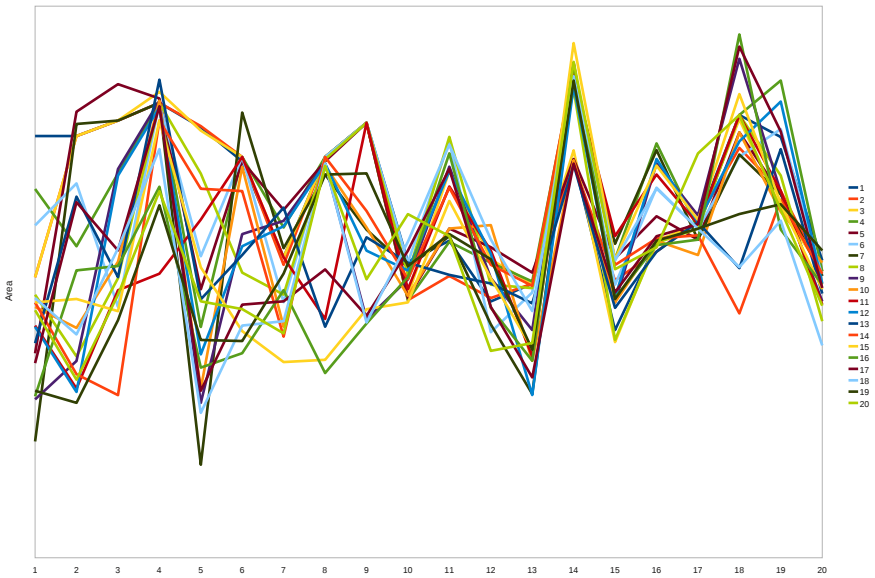
```
a: sint[32], b: sint[32], s: bv[2] => x: sint[33];

x := s[0] ? (s[1] ? a + b : a ^ b)
          : (s[1] ? a & b : a | b);
```

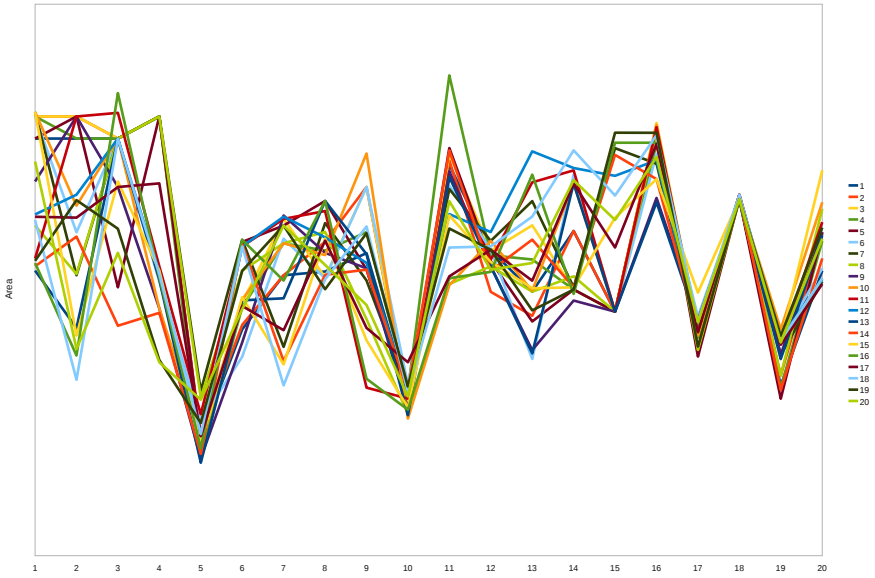
In this case, b can be mapped to C and a can be mapped to $A : B$ for all operators, which makes it easier for the algorithm to find an efficient instantiation. The handwritten optimized version could look like this:

```
op := s[0] ? (s[1] ? 15 : 51)
           : (s[1] ? 51 : 59);
alu := s[0] ? (s[1] ? 0 : 4) : 12;

DSP48E (.A (a[31:18]), .B (a[17:0]), .C (b),
        .OPMODE (op),.ALUMODE (alu)
        => .P (result), .CARRYOUT (unused));
```



(a) Modified circuit 1



(b) Modified circuit 2

Figure 6.2: Comparison of numbers of mutation steps for slightly modified versions of the circuit.

```
x := result;
```

In fact, the algorithm finds a circuit which is even 25 % smaller:

```
x := local_6;
local_0 := (s[1:1]);
local_1 := (s[0:0]);
local_2 := (a[31:18]);
local_3 := (a[17:0]);
local_4 := ((local_1 ? local_0: !local_0) ?
               (local_1 ? 15 : 59) : 51);
local_5 := (local_1 ? (local_0 ? 0 : 4) : 12);
DSP48E (.A (local_2), .B (local_3), .C (b),
        .OPMODE (local_4), .ALUMODE (local_5)
        => .P (local_6), .CARRYOUT (local_7));
```

Chapter 7

Conclusion

In this thesis, I presented a new approach to automatic hard block inference and instantiation for FPGAs using genetic programming. The algorithm was implemented in order to demonstrate its effectiveness, and to test and evaluate it using a benchmark.

The results of the experimental evaluation were mixed: depending on the input circuit, the algorithm can find a circuit which is smaller than the handwritten one. In other cases, it fails to deliver a circuit which is even close to the quality of a manually optimized solution, even though it could theoretically find it if the needed transformations were applied in the right order. For that reason, it is important to develop new heuristics which determine what rule to apply next, and where to apply it.

For example, after a transformation has been applied, it might be beneficial to constrain the following rule applications to that area of the circuit; otherwise an individual might be removed from the population before a mutation with short-term disadvantages but long-term benefits can be exploited.

Another problem is the multiplexing of hard blocks. Although the algorithm can instantiate and multiplex hard blocks, this process is not yet fully automated, since it requires guidance in the form of weights for the fitness evaluation. A new kind of rule which combines the instantiation- and the multiplexing-phase might solve this problem. It would try moving as many operators as possible into one hard block instance, thereby avoiding the in-

termediate situation where many hard blocks are instantiated but not yet multiplexed. When and where this new rule is applied would still be decided in a random way.

7.1 Future work

The approach presented in this thesis, as well as the implementation of the algorithm, still offers a lot of potential for improvement.

7.1.1 Support for sequential circuits

Although only combinational circuits have been treated in this thesis, the approach I presented can be extended to sequential ones. This extension would pose new challenges, but it would also enable powerful optimizations that are only possible once sequential elements (registers, block RAM, and distributed RAM implemented in LUTs) are considered. Such optimizations include pipelining a computation or moving calculations into block RAM (hard block memory elements are clocked, so they can't be used in purely combinational circuits).

This functionality could even simplify the programming of digital logic. For example, instead of manually designing a sequential circuit for a square root operation, the programmer could describe it in an arithmetic (i.e., combinational) way, set the constraints (e.g., "a maximum of 10 cycles latency at a clock period of 5 ns; optimize for area usage"), and the algorithm would try to find an optimal implementation by sequentializing the circuit and inferring hard blocks.

7.1.2 More accurate fitness evaluations

The currently implemented fitness function, which estimates the area and delays of a circuit, is probably very imprecise for two reasons:

1. There is not direct relationship between the nodes in the DAG and the LUTs on the FPGA.

2. A synthesis tool can perform optimizations which are not available to the algorithm (e.g., logic minimization).

Tests will have to be conducted to determine how accurate the performance approximations are. In case they deviate too much from the real values, either another, more exact approach to fitness evaluation will have to be found, or the circuits will have to be synthesized using an external tool, which then returns information about delays and area usage.

Unfortunately, such tools suffer from a long runtime, which is in the order of seconds even for very small circuits, so it is not reasonable to perform synthesis too often. On the other hand,

- the process of completely implementing an FPGA, which includes mapping, placing and routing a circuit, can take hours or longer for large designs, so long tool runtimes are common in the hardware design domain;
- evaluating the fitness of several individuals can be trivially and efficiently parallelized;
- the automation of hard block inference done by the algorithm saves a lot of expensive programmer time, while the increased CPU time is cheap.

So in the end, a long runtime would not be a big disadvantage, and implementing the design to get accurate performance measures could be done in practice, as long as the number of fitness evaluations can be kept to a minimum.

7.1.3 Other improvements

Further possible improvements include:

- *Multicriteria optimization* [9] allows optimization for several objectives (currently the fitness values for delay and area are weighted and added). The algorithm would return several optimal solutions, and the user could select one of them after comparing their respective trade-offs.

- Support for *don't care*-values, which potentially enable more optimizations.
- Transformation rules for equivalences which are not implemented yet. Furthermore, helper functions which make the manipulation of circuits easier and more convenient, thereby allowing faster and more error-free writing of rules.
- *Recombination* would likely be beneficial to the efficiency and effectiveness of the algorithm since good sequences of transformations don't have to be discovered repeatedly in different individuals. However, this is not easy to implement, because one part of a circuit must not simply be copied to another part, as that might change the circuit's semantics. A possible solution would be the replacement of the complete input tree of an output with the input tree of the same output of another circuit.
- Better heuristics which decide where the next rule should be applied, as well as heuristics which control hard block instantiation.

Bibliography

[1] A. Abdollahi and M. Pedram. A new canonical form for fast boolean matching in logic synthesis and verification. In *DAC '05: Proceedings of the 42nd annual Design Automation Conference*, pages 379–384, New York, NY, USA, 2005. ACM.

[2] Altera Corporation. *DSP Blocks in Stratix IV Devices*, November 2009.

[3] L. Benini and G. De Micheli. A survey of boolean matching techniques for library binding. *ACM Trans. Des. Autom. Electron. Syst.*, 2(3):193–226, 1997.

[4] S. Bleuler, M. Brack, L. Thiele, and E. Zitzler. Multiobjective genetic programming: reducing bloat using SPEA2. volume 1, pages 536 –543 vol. 1, 2001.

[5] T. Blickle and L. Thiele. Genetic programming and redundancy. 1994.

[6] S. Boettcher and A. G. Percus. Extremal optimization for graph partitioning. *Phys. Rev. E*, 64(2):026114, Jul 2001.

[7] S. Boettcher and A.G. Percus. Optimization with extremal dynamics. *Physical Review Letters*, 86(23):5211–5214, 2001.

[8] A. De Geus and W. Cohen. A rule-based system for optimizing combinational logic. *IEEE Des. Test*, 2(4):22–32, 1985.

[9] M. Ehrgott. *Multicriteria Optimization.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[10] A. E. Eiben and J. E. Smith. *Introduction to Evolutionary Computing.* SpringerVerlag, 2003.

[11] D. Gregory, K. Bartlett, A. deGeus, and G. Hachtel. SOCRATES: A system for automatically synthesizing and optimizing combinational logic. In *25 years of DAC: Papers on Twenty-five years of electronic design automation*, pages 580–586, New York, NY, USA, 1988. ACM.

[12] Yu Hu, V. Shih, R. Majumdar, and Lei He. Exploiting symmetries to speed up SAT-based boolean matching for logic synthesis of FPGAs. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 27(10):1751 –1760, 2008.

[13] J. Kennedy and R. Eberhart. Particle swarm optimization. In *Neural Networks, 1995. Proceedings., IEEE International Conference on*, volume 4, pages 1942 –1948 vol.4, 1995.

[14] S. Kirkpatrick. Optimization by simulated annealing: Quantitative studies. *Journal of Statistical Physics*, 34:975–986, 1984.

[15] I. Kuon and J. Rose. Measuring the gap between FPGAs and ASICs. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 26(2):203 –215, feb. 2007.

[16] I. Kuon, R. Tessier, and J. Rose. FPGA architecture: Survey and challenges. *Found. Trends Electron. Des. Autom.*, 2:135–253, February 2008.

[17] R. Poli. Analysis of the publications on the applications of particle swarm optimisation. *J. Artif. Evol. App.*, 2008:1–10, 2008.

[18] A. Solar-Lezama, R. Rabbah, R. Bodík, and K. Ebcioglu. Programming by sketching for bit-streaming programs. *ACM SIGPLAN Notices*, 40(6):281–294, 2005.

[19] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat. Combinatorial sketching for finite programs. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 404–415. ACM, 2006.

[20] T. Soule, J. A. Foster, and J. Dickinson. Code growth in genetic programming. In *GECCO '96: Proceedings of the First Annual Conference*

on Genetic Programming, pages 215–223, Cambridge, MA, USA, 1996. MIT Press.

[21] A. Thompson. An evolved circuit, intrinsic in silicon, entwined with physics. In Tetsuya Higuchi, Masaya Iwata, and Weixin Liu, editors, *Evolvable Systems: From Biology to Hardware*, volume 1259 of *Lecture Notes in Computer Science*, pages 390–405. Springer Berlin/Heidelberg, 1997.

[22] Xilinx Inc. *Virtex-5 FPGA User Guide*, May 2010.

[23] Xilinx Inc. *Virtex-5 FPGA XtremeDSP Design Considerations User Guide*, June 2010.

[24] V. Černý. Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. *Journal of Optimization Theory and Applications*, 45:41–51, 1985.