Karlsruher Institut für Technologie

Institut für Programmstrukturen
und Datenorganisation (IPD)

Lehrstuhl Prof. Dr.-Ing. Snelting

Verbesserte Optimierung von Integer-Konvertierungen und VHDL-Codeerzeugung mittels Bitbreitenanalyse

Bachelorarbeit von

Marcel Hollerbach

an der Fakultät für Informatik

Erstgutachter:	Prof. Dr.-Ing. Gregor Snelting
Zweitgutachter:	Prof. Dr. rer. nat. Bernhard Beckert
Betreuende Mitarbeiter:	M.Sc. Andreas Fried
Bearbeitungszeit:	1. Januar 1990 – 31. Dezember 2000

Zusammenfassung

Normal high level programming languages often just present standard data types for 8, 16, 32 and 64 bit numbers. However, the whole bandwidth of those data types is often not used at all.

Looking at bigger C projects also shows that often enough C developers are using data types which are a lot bigger than there maximum number. *integer* types are used as a standard complete number type in those projects. Instead of a smaller better fitting number type. With the analysis presented in this work, the data types can be better adjusted to a good fitting data word length.

In high-level Programiersprachen wie C gibt es vordefinierte Datentypen wie 8 / 16 / 32 / 64 bit Datentypen. Diese werden jedoch often nicht so eingesetzt das die moeglichst beste auslastung der moeglichen Zahlen erreicht wird.

Im hinblick auf C Projekte zeigt sich auch das der datentyp *integer* immer mehr zu einem allgemein gueltigen Zahlendatentyp wird, welcher jedoch so gut wie nie die maximale groesse seiner zahlen erreicht.

In dieser arbeit geht es um die analyse, welche es erlaubt zu erkennen ob ein datentyp voll ausgenutzt wird oder nicht.

Das Titelbild ist von http://www.flickr.com/photos/x-ray_delta_one/4665389330/ und muss durch ein zum Thema passendes Motiv ausgetauscht werden.

Contents

1. Introduction

The problem to solve is called bitwidth analysis. The bitwidth with a data word can be seen as the bits that are actually used in the runtime of the source code. Lets take the following example:

```
int arr[4];

for (int i = 0; i < 4; i++) {
  int res = (i << 4) + i*i;
  arr[i] = res;
}
```

After running the bitwidth analysis, every operation has a attached information structure which indicates how many bits are actually used by the code. The developed algorithm applied to the the code results in something like this: $i \in 0..3$; $res \in 0..90$;

2. Basics

2.1. cparser / libfirm

Modern compilers are now developed for over half a century. Over the time a new structure has evolved. Compilers are split into three parts. The first part is called *front end* and handles everything related to the language specific parts. The second part is called *middle ware* and handles the abstract notation of code execution. The last part is called *back end* and does the translation into something like assembler or java bytecode. As an example, cparser is handling the C specific tasks. The parsed c code is then translated into a control flow graph from *lib*Firm, which is the middle ware. The *lib*Firm middle ware then also acts as back end and translates the control flow graph into assembler / java bytecode.

Control flow graph in libFirm The interaction between the front and middle ware is based on a data structure called control flow graph (CFG). A CFG is a directed graph. Each node is representing a operation. A node can have operands. We say that a node always depends on a other node, when there is a edge from the node to the other node. CFGs are also the base structure for analyzing the control flow of a software. Such a analysis is called control flow analysis. In *lib*Firm the nodes of the CFG can have different types: control flow, arithmetical operations, memory handing, constant expressing. A node is additionally placed in a block. A node is executed if all its operands have been executed. Or, if there are no operands, when its block is executed. Another software that does the same thing is *clang*, where the back end is handled by *llvm*. A more in detail explanation can be found in [1]. A more theoretical and abstraction introduction can be found in [2]

Confirm nodes in libFirm In *lib*Firm there are several node types, one of them is called Confirm nodes. A Confirm node is not having a hardware representation. It is used to express that the operand of the Confirm node will output a value that is within the confirmed node. A Confirm node therefore has 2 operands. They are called value and bound, additionally a Confirm node has a relation. As relations $\neq$, $=$, $<$, $>$, $>=$, $<=$ are possible. Written as prefix notation we can say that

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char const *argv[]) {
  printf("Hello Firm!");
  return 0;
}
```

Figure 2.1.: Sample source code

value	7	6	5	4	3	2	1	0
5	0	0	0	0	0	1	0	1
-2	1	1	1	1	1	1	1	0

Figure 2.2.: Number representation

the Confirm node gives the assertion that $\tau(value, bound)$ is true, where τ is the relation.

True / false Blocks A compare node in libFIRMhas always a relation, 2 operands and 2 blocks. One of the blocks is executed depending on if the criteria was meet or not. The block that is executed when it was meet, is called *true-block* the other block is called *false-block*. The structure is visualized in Figure 3.2

Interaction from front to back end As an example, we want to compile the program code from 2.1. First of all the front end will parse the given source code. The parsed code then will be stuffed into a abstract syntax tree (AST). The representation in the abstract syntax tree will encode the syntactical informations from the software. Informations like curly brackets will not be needed anymore.
After that the AST will be transformed into a CFG.
At this point the front end handed the informations for binary creation to the back end. However, the front end can now tell the back end to perform optimizations on the information. In terms of C this is often controlled with the -O1,2,3 flags.
After the optimizations have taken place, the binary files can be written into a file, and the compiler call is finished.

Number representation So called *modes* are used in libfirm to represent data words on which operations are performed. A mode specifies a length in bits. A data word can have a type, this can be *integer*, *float*, *reference*, *data* and *boolean*. For us, the only interesting type is the integer type, since this is the only type where we can

save bitwidth for now. Those integer-modes can be signed or unsigned. The length of the mode defines the maximum number. If it is signed, then it also defines the minimum number. Otherwise the minimum is just null. The sign is encoded using two's complement. As an example, in Figure 2.2 two numbers are displayed with its hardware representation. There are the following integer type modes in libFirm: *int8*, *uint8*, *int16*, *uint16*, *int32*, *uint32* , *int64*, *uint64*,

The term *mode* is used to describe integer-modes for the rest of this thesis.

2.2. VHDL code generation

Projects like *libva* have made the first step into the direction of hardware accelerated video decoding on a computer system. These projects provide standardized access to decoding video streams like MPEG-2, H.264/AVC, H.265/HEVC, VP8, VP9, VC-1. The project itself handles the passing and messaging from a front end application like a video player, to the driver itself. The driver itself then answers the calls from the API. And the decoded picture goes back to the front end application. However, the driver itself still needs to do the decoding on its side. Usually implemented in hardware, since this is resulting in the best performance. The amount of work needed for implementing such a decoder for MPEG-2 can be observed while reading [3]. It seems like a good idea to think about writing the code in a higher level language and transpile it into vhdl.

2.3. Mathematical theory

There are a few mathematical basics, which are known to be useful for performing compiler analysis. The first basic is lattice. A lattice can be useful for comparing elements, which are not directly comparable. The second basic is the fixed point iteration.

2.3.1. Lattice

A lattice is a algebra structure. For a lattice V, there are the following rules:

- the height of V must be ∞

- $\sqcup : V \times V \to V$ returns the smallest element, that is bigger or equal than the two operands
- $\sqcap : V \times V \to V$ returns the biggest element, that is smaller or equal than the two operands
- $\forall u, v \in V : u \sqcup (u \sqcap v) = v \wedge u \sqcap (u \sqcup v) = v$

The element, which is the least of all elements, is called $\bot$. The greatest of all is called $\top$. A lattice can also be made visible in a Hesse diagram, which you can see in Figure 3.1

A more detailed introduction can be found in [4].

2.3.2. Fixed point iteration

Fixed point iteration is a method for finding a x where we know that $f(x) = x$. First of all we define $f : D \to D$. f is monotone, additionally $|D|$ must be finite. We call $x \in D$ a fixed point if $f(x) = x$. For iterating we start with the minimum of D. We define $y_0 = f(min(D))$, $y_i = f(y_{i-1})$. With the requirements from above we can say: $\exists i \in N : y_i = y_{i-1}$. For further reading, please see [5]

Fixed points in compiler analysis We can transform and use the fixed point iteration to be working on a CFG. For the following we define additionally:

- CFG is called (G, V, E)
- The lattice we use is called L
- $f_{node} : L \to L$ As the function for iterating on a single node. f_{node} is continuous.
- D is a set, where we can say that: $\forall d \in D : d = \{(0, v_0), ..., (n, v_n)\}$
- $f(d) := \{(id, \hat{v}) | (id, v) \in d \wedge f_{node}(v) = \hat{v}\}$

We start the iteration with $d_{bot} := \{(0, \bot), ..., (n, \bot)\}$. At some point, while continuing the iteration, we will find a $\hat{d} = f(\hat{d})$.

2.4. Software basic

2.4.1. Worklist algorithm

A possible solution for implementing fixed point algorithms could be done by iterating the whole CFG everytime a node changes. However, this is quite wasteful, since we know that the state of a node only depends on the state of its operands. Therefore a algorithm called *worklist algorithm* has evolved and can be found in algorithm 1.

```
worklist := CFG.Nodes;
while worklist not empty do
    node := worklist.pop();
    node_changed := recalc(node);
    if node_changed then
        foreach operand ∈ node.operands do
            if !worklist.contains(operand) then
                worklist.appi jend(operand)
            end
        end
    end
end
```

Algorithm 1: Worklist alogithm

The base idea is to maintain a worklist which contains all nodes that are dirty. A node is considered dirty, when its operands have changed, but the node itself is not recalculated. Additional informations can be found at [6]

3. Design & Implementation

3.1. Bitwidth analysis

The analysis is a data flow analysis. The analysis attaches an $(int, boolean)$ tuple to every meaningful node. A node is considered meaningful if the node has an integer-mode. We will reference the first value as *stable bits* and the second bit as *is positive*.
What stable here means can be observed when looking at 2.2. For the first line of the table we have five stable digits. The second line has 7 stable digits.
Since this is a data flow analysis, which is built on an iterative approach. We might want to address the current tuple of a node x, while we have a new one calculated. Therefore we call $\hat{x}$ the currently associated value, and x the newly calculated one.

Bit representation The stable bits indicate how many bits are stable, and therefore not used. The second value of the tuple indicates if the value will ever reach negative numbers or not. Thus it indicates at least one stable bit at the highest position. However, the second value is only meaningful for modes that allow signs.

Range representation There is also a second way of interpreting the two values. The stable bits can define a minimum and maximum range. The maximum number is reached if the stable bits are all zero, and the rest one. If the mode is signed and the node is not positive, then the minimum number is reached by assuming all stable bits are one, and the rest zero. Otherwise the minimum is 0. We can define the following min max definitions for the ranges:

$$max_{bitwidth}(x) = \begin{cases} 2^{stable\_digits-1} - 1 & mode.signed \\ 2^{stable\_digits} - 1 & Otherwise \end{cases}$$

$$min_{bitwidth}(x) = \begin{cases} 2^{stable\_digits-1} & mode.signed \wedge is\_positive \\ 0 & Otherwise \end{cases}$$

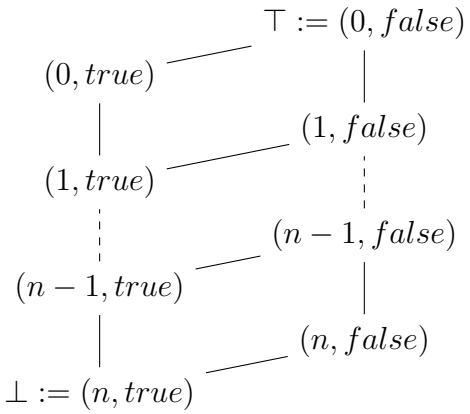

Figure 3.1.: The definition of a upper bound compare node

Analysis The analysis works as a fixed point iteration. For it we use Figure 3.1 as lattice. The values of the lattice represent the tuples from the analysis.

As a first step, we iterate over every single node and initialize the node with $\top$ and mark it as *dirty*. If the node is constant, we calculate its bitwidth. Nodes with the opcodes *Const*, *Size* and *Address* are considered constant.

The second step consists of recalculating every *dirty* node in the graph. if the stable bits of x are smaller than those from $\hat{x}$, then the new value is memorized as $\hat{x}$ of the node. Also every successor of the node is marked as dirty. The used rules for recalculating the nodes are described in section A.1.

3.1.1. Value prediction

In addition to the normal analysis results, the fixed point iteration can insert additional confirm nodes. Those confirm nodes help making the analysis more accurate. First of all we need a few definitions for easier understanding:

Definition: Upper bounds A compare node defines an upper bound if the relation is $<$ and the second operand is constant.
The definition also applies for compare nodes that can be transformed by swapping the two operands and the relation accordingly.

Definition: Predecessor in a certain block In the detection described later, we often need to find a predecessor that is placed in a certain block. Therefore we

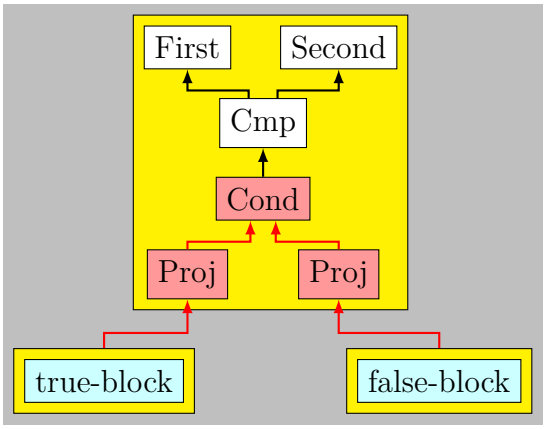

Figure 3.2.: The definition of a upper bound compare node

define:

$$\kappa(a, b) := \{X | X \leftarrow a \wedge X.block = b\}$$

It will return every node that is located in b and is a predecessor of a.

Definition: Constant dependencies While looking at C code we often see that addition and multiplication nodes are used for calculating array addresses or addresses for structure access. Therefore, one operand of the arithmetical operations is often constant. An example for this can be found in Figure 3.3. We define ξ to explore the whole tree of nodes with one constant and one none-constant operands, and return us every node that was not constant.

$$\xi(a) := \begin{cases} a \cup \xi(c) & \text{, If there is only one not constant dependency } c \\ \emptyset & \text{, otherwise.} \end{cases}$$

If a has only one none-constant operand c, then ξ returns the element a and $\xi(c)$. Otherwise it returns an empty set. In Figure 3.3 the highlighted nodes are part of $\xi(Y)$.

Upper bounds for block execution The values that are calculated in a node are (even if the fixed point iteration is not stable yet) within the ranges for the later stable result. The iteration starts at $\top$ and moves into the direction of $\bot$. We now evaluate an upper bound compare node, every time the first operand changes. In the beginning we can say that that with those possible values, each time the true-block

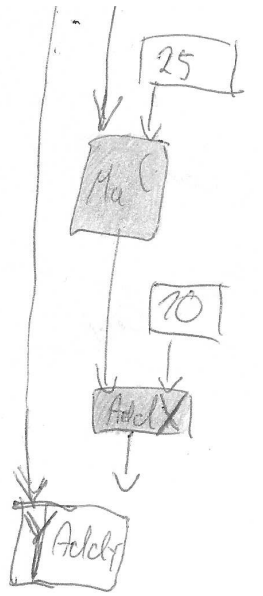

Figure 3.3.: Definition of xi

will be executed. However, if $max_{bitwidth}(n)$ grows enough to get bigger than the second operand or the compare node, then we can say, that we have found a upper bound for the execution of the true-block. Which is $max_{bitwidth}(n)$ in the current state. Thus we can insert a confirm node between every node $e \in \kappa(i, j)$ and i, where i is the first operand and j is the true block.

Extended confirm insertion The confirm nodes that we have inserted in the paragraph before can also be transported backwards. With $\xi(i)$ we can get a set of nodes, where the current state in the analysis is only depending on one node. Thus we can say that the state of every node from $\xi(i)$ will not change as long as the topmost upper element in the tree structure does not change. Which means that we can insert a confirm node between (e, g) for every $e \in \xi(i)$ and $g \in \kappa(e, j)$

3.2. Stable Conversion nodes

In libfirm a conversion node can be used to convert a value from one mode to another. This type of node has one operand. A conversion like this can have one of two effects. The value stays the same, or the value changes, due to the inability of displaying the value in a different mode. We call the first case *stable conversion node*. A example for an unstable conversion node may look like $(unsigned)((int)$-10). A stable conversion node may look like $(unsigned)((int)10)$.

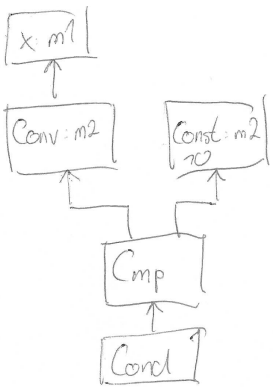

Figure 3.4.: Conversion compare construction

Finding stable conversion nodes Such stable conversion nodes can be found using the bitwidth analysis. We compare the range of the operand with the range of the conversion node itself. If the ranges are the same, then we know that the conversion is stable.

Removing conversion nodes In case we found a stable conversion node, we can say that this node only exists for syntax rules, there is no semantical value in it. Removing those nodes also has the advantage of helping other analyses. The confirm insertion algorithm of *lib*Firm searches for assertions that can be made based on looking at compare nodes. This works quite well. However, the example in Figure 3.4 does not work. The insertion code could only insert a Confirm between the Compare and the conversion. However, the Confirm node would need to be an operand of the conversion node, in order to help the rest of the code, since other nodes will likely depend on the operand of the conversion node and not on the conversion node itself. After removing the conversion node, the analysis can find an assertion based on the compare node. This could help the branch prediction and dead code elimination.

However, for really removing the conversion nodes, we need to find situations where we can eliminate the conversion node. We have already seen an example with a compare node. Additionally we can do the same with an arithmetical operation.

Compare-Conversion optimization The rule in *lib*Firm is that the two operands of a compare node have to have the same mode. This means, when we remove the conversion node, we also need to adjust the mode of the second operand. This is not easily possible for a none-Constant node. Thus we confine for now that the the operand needs to be a Conversion node, and the second one a Constant node. With this we can simple adjust its mode, and remove the conversion.

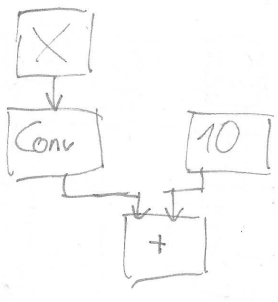

Figure 3.5.: Arithmetical optimization example

Arithmetical-Conversion optimization The situation of arithmetical nodes, with one operation being constant, one operation being a conversion, can also be optimized. In this case we move it through the arithmetical operation, and place it afterwards. In the same time we change the mode of the constant and the arithmetical node to the earlier node. We do this in order to hope that we can remove it with a compare conversion optimization after that. Such a construct might look like Figure 3.5 . After we adjusted the mode of the constant we can move the conversion from the operand of the arithmetical operation to the end.

3.3. VHDL generation

There is a *lib*Firm tool called firm2vhdl. The tool takes the output from the *cparser* compiler, and outputs the VHDL. Every node in the firm graph gets transformed into a VHDL statement. This is done by transforming the operation of the nodes into VHDL code. Each result of a node operation is assigned to a new variable, which then can be used again later by the next operation. Each of those those variables are represented in hardware, and thus have an certain amount of bits. The maximum amount of bits that is needed, can be calculated by using the bitwidth analysis.

bitwidth in firm2vhdl The amount of bits per variable were previously just the amount of bits the mode of a node needs. In VHDL this wastes a lot of space on the FPGA chip later on. Minimizing the amount of bits used per variable here can be important, since most of the variables used in C do not use the complete bitwidth. The bitwidth information gathered from the analysis can help here, as it defines how many bits of a node are used, and how many are not. Thus we can add code to the transformation, for taking the bitwidth, instead of the number of bits in a mode.

3.4. How about section

This section is about decisions that where made, and why they are made.

3.4.1. Value range vs. bitwidth

The analysis explained in 3.1 is outputting bitwidth informations. The fixpoint iteration uses the set of rules described in section A.1. The same analysis could use a set of rules where we don't calculate the bitwidth, but rather the direct range of possible values (Not restricted to 2^x form). This is already implemented in libFirm, its called value range propagation (VRP).

However, there should be some way to compare both models. We know that every node in *lib*Firm has a mode, this mode is defining the length of a data word. The analysis is measuring how many of the bits from this data word are used. So once a node gets marked dirty, there is some sort of delta, like $\delta = max_{bitwidth}(x) - max_{bitwidth}(\hat{x})$. In case of bitwidth, we know that the delta will have the form of 2^x. In case of the value range detection, the only assertion we can have is $\delta = 1$. Now lets take a graph like ???, where X is some sub graph that performs a operation. Under the assertion that the delta of last node is minimal, then the bitwidth analysis is faster. A real life example where this is happening is for example the simple access to a array, like $array[i]$.

3.4.2. Widening & Narrowing

4. Evaluation

The most interesting projects for evaluating the implementation, are decoding softwares that are transforming a binary blob into a image. For this purpose, libjpeg, libpng and libgif are evaluated. Additionally zlib is done, since it is a general purpose library for lossless compression, which is often used in reallife products.

For those projects we are measuring two things. First, how effective our bitwidth saving is and how much we have saved. Second, we are trying to compare the assembler with and without the applied optimizations.

4.1. General bitwidth

A node in *lib*Firm uses a amount of the available bitwidth. We note down the bitwidth of a node as $bitwidth_{irg}(n)$. If we don't perform our analysis, then we need to assume the worst case, the node uses all the bits that are available from its mode. If the analysis is performed, then we can say that the stable bits of the analysis are defining the bitwidth of a node. For comparing the functions directly, we define:

$$bitwidth_{irg}(irg) := \frac{\sum_{n \in graph.nodes} bitwidth_{irn}(n)}{|graph.nodes|}$$

We divide the sum by the number of nodes, to be able to compare smaller graphs with bigger ones.

Const nodes While evaluating the projects, it came up that a normal graph consists of a lot of constant nodes, exact percentages can be found in the third column of Figure 4.1.

The problem with a Constant node is, the bitwidth can be quite low. While the constant does not matter on hardware in most cases, since they are decoded directly in a hardware instruction. Additionally they don't take up any additional space in VHDL.

"lib"	"nodes"	"percentNodes"
"zlib"	32,726	0.24
"libjpeg"	66,887	0.22
"libpng"	31,934	0.29
"libgif"	4,562	0.35
"libtiff"	51,683	0.3

Figure 4.1.: Statistics from the projects

"lib"	"avgBw"	"avgBwNc"	"safedBw"	"safedBwNc"
"zlib"	44.09	40.67	22.33	9.22
"libjpeg"	44.85	40.76	25.34	12.93
"libpng"	37.15	32.73	20.44	9.21
"libgif"	41.05	36.53	23.37	8.76
"libtiff"	41.11	37.14	22.53	9.48

Figure 4.2.: Statistics from the projects

Calculating how much bitwidth a constant takes is anyway only the matter of a log2(...) call, which is not really interesting to evalulate, in order to see how good the analysis works.

Summed up, Constant nodes are sophisticating the bitwidth of a function a lot. Thus the statistics later are divided into results with and without the constant values.

Project results In Figure 4.1 the second column is showing the total number of nodes that got created in order to compile the library. This can be used as some sort of complexity metric. Figure 4.2 shows that all the libraries except *libjpeg* have a similar amount of bitwidth that got saved. However, the amount of saved bitwidth still does not really correlate with the complexity of the projects, since *libgif* is not similar complex to *zlib* or *libpng*.
We can say that we safe up about 9 bits in average per node.

4.2. Optimizations on assembler output

In this section we compare the shared object files of the library. The first run was done with plain cparser, the second run was done with cparser + the patches created for this thesis.

After that the disassembly of the shared object files are compared. However, comparing those two shared object files is quite hard, due to addresses and instruction order changes. Making the comparing easier was possible after removing the address of every line and removing the arguments of the instructions. Loosing the arguments of the instruction makes it hard to compare if something in the calling of the function has changed. However, we can better study the raw changes on the instructions. What follows is the explanation, how the assembler output changes.

libgif There was not a single change, the two binary files are identical.

libjpeg The assembler output from after the optimization is about 30 instructions longer. The functions don't differ a lot. It looks most of the times like simple reordering of instructions. The amount of additional instructions is caused by instruction duplication, which can be caused by loop unrolling.

libtiff The results here are similar to libjpeg. One difference is, the optimization causes add instructions to be translated to lea instructions.

libpng libpng seems to be different here. The assembly file after the optimization is shorter than before, by 10 instructions. Instructions that are removed are mainly mov instructions.

zlib zlib is similar to libpng. The file gets 3 lines shorter, which is not much. The rest of the binary differences are basically instruction order changes.

The complete repository with the releases and data and scripts can be found at (FIXME)

4.3. Optimization vhdl improvements

5. Conclusion

5.1. Assembler generation

The previous chapter showed that the optimizations are not that helpful on decoder code. This thesis was done for evaluating decoder code, so no others projects have been checked. However, other projects like direct UI or CLI implementations also could be evaluated, since the patterns from decoding code are quite different to ui codes. Without evaluating others, we can say, that the optimization for dropping conversion nodes from the graph is not that helpful.

5.2. vhdl generation

5.3. Further improvements

5.4. Additional analyzer usage

Bibliography

[1] G. Lindenmaier, "libFIRM – a library for compiler optimization research implementing FIRM," Tech. Rep. 2002-5, Sept. 2002.

[2] F. E. Allen, "Control flow analysis," *SIGPLAN Not.*, vol. 5, pp. 1–19, July 1970.

[3] K. Rosengren, "Modelling and implementation of an mpeg-2 video decoder using a glas design path," tech. rep.

[4] G. Birkhoff, *Lattice Theory - Third edition*. American Mathematical Society, Colloquium Publications, 1995.

[5] V. Berinde, *Iterative Approximation of Fixed Points*. Springer, Berlin, Heidelberg, 2007.

[6] K. K. Keith D. Cooper, Timothy J. Harvey, "Iterative data-flow analysis, revisited."

Erklärung

Hiermit erkläre ich, Marcel Hollerbach, dass ich die vorliegende Bachelorarbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis beachtet habe.

Ort, Datum	Unterschrift

A. Sonstiges

A.1. Table of node rules