Institut für Programmstrukturen
und Datenorganisation (IPD)

Lehrstuhl Prof. Dr.-Ing. Snelting

Improved integer conversion optimisation and VHDL code generation based on bit-width analysis

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 dass diese möglichst beste Auslastung der möglichen Zahlen erreicht wird.

Im hinblick auf große C Projekte zeigt sich auch das der Datentyp *integer* immer mehr zu einem allgemein gültigen Zahlendatentyp wird, welcher jedoch so gut wie nie das Maximum seiner Zahlen erreicht.

In dieser Ausarbeitung 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 **7**

2. Basics **9**
2.1. cparser / libfirm . . . 9
2.2. Software theory . . . 11
2.2.1. Lattice . . . 11
2.2.2. Fixed point iteration . . . 12
2.2.3. Worklist algorithm . . . 13

3. Design & Implementation **15**
3.1. Bitwidth analysis . . . 15
3.1.1. Value prediction . . . 16
3.2. Stable Conversion nodes . . . 18
3.3. VHDL generation . . . 20
3.4. Value range vs. bitwidth . . . 20

4. Evaluation **23**
4.1. General bitwidth . . . 23
4.2. Optimizations on assembler output . . . 24
4.3. Optimization VHDL improvements . . . 25

5. Conclusion **29**
5.1. Assembler generation . . . 29
5.2. VHDL generation . . . 29
5.3. Further improvements . . . 29
5.3.1. Widening & Narrowing . . . 29
5.3.2. More conversion nodes . . . 30

A. Appendix **35**
A.1. Table of node rules . . . 35
A.2. Evaluation C source code . . . 35

1. Introduction

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 [1]. Thus it would be nice if software that gets written in C, can also be adopted to work directly on hardware. As an example, a C library which does decoding of MPEG-2 could just share its decoding code with a Hardware Description Language. There are techniques for doing this. However, they suffer from a lag of performance. This thesis tries to improve the speed of such code by reducing the used bits of the software. The solution has two pieces, an analysis and a optimization. The initial idea of the analysis is to annotate every operation with the bitwidth it requires. The following C snippet is formated a bit unlikely in order to make it easier to annotate.

```c
int arr[4];

for (int i = 0; i < 4; i++) {
  //i requires 3 bits
  int x = i*i; //x requires 6 bits
  int y = (i << 4); //y requires 7 bits
  int res = x + y; //res requires 8 bits
  arr[i] = res;
}
```

Every operation has a bitwidth. The bitwidth is the number of bits that is used by the operation, as maximum the modes amount of bits is used. The required bits per operation are annotated as comment after the operation. How exactly the bitwidth of a operation is calculated will be covered in the later chapters. The information gathered from the analysis is then used for optimizing the compiler output. The optimization here can happen at two layers. The compiler we use here can be used to output a hardware description language called VHDL. This output will be optimized to have a more compact memory layout. The other compiler output is assembler output, which can also be optimized. If those optimizations are successful and really safe up resources can be found in the later sections.

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 similar to a control flow graph, it is called FIRM graph. A FIRM graph is a directed graph. Each node is a operation. Nodes can have operands. We say that a node always depends on another node, when there is a edge from the node to the other node. The FIRM graphs are also the base structure for analyzing the control flow of a software. Such a analysis is called control flow analysis. The nodes of the FIRM graph can have be categorized in 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 [2]. A more theoretical and abstracted introduction to control flow graphs can be found in [3]

Confirm nodes in libFirm In *lib*FIRM there are several node types, one of them is called "Confirm" node. A Confirm node does not have a hardware representation. A confirm node 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 the Confirm node gives the assertion

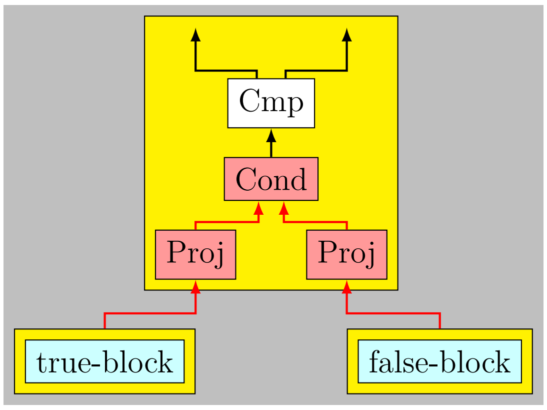


Figure 2.1.: The construct that is called a upper bound compare node

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

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

Figure 2.2.: Sample source code

that $\tau(value, bound)$ is true, where τ is the relation. This is useful for control flow analysis, as the Confirm nodes can indicate possible shortcuts with knowledge that is gained from other analysis.

True / false Blocks A compare node in *lib*FIRM has always a relation, 2 operands and 2 blocks. The compare node executes the true block, when the relation and operands are evaluating to true. The false block is executed otherwise. The structure is visualized in Figure 2.1

Interaction from front to back end As an example, we want to compile the program code from Figure 2.2. First of all, the front end parses the given source code. The parsed code then is transformed into a abstract syntax tree (AST). The representation in the abstract syntax tree encodes the syntactical informations from the software. File contents like curly are encoded in the structure, and are not explicitly represented in the AST.
After that the AST is transformed into a FIRM graph.
At this point the front end handed the informations for creating a binary programm

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 2.3.: Number representation

to the back end. However, the front end can now tell the back end to perform optimizations on the FIRM graph. In terms of C, this is often controlled with the -O1,2,3 flags.
After the optimizations have taken place, the binary files are written into a file, and the compiler call is finished.

Number representation So-called *modes* are used in *lib*FIRM to categorize data words. A mode specifies a length in bits. A data word can have a sort, 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 compute 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 zero. The sign is encoded using two's complement. There are the following integer modes : *int8*, *uint8*, *int16*, *uint16*, *int32*, *uint32*, *int64*, *uint64*,

As an example, in Figure 2.3 two numbers are displayed with its hardware representation. The term *mode* is used to describe integer-modes for the rest of this thesis.

2.2. Software theory

There are a few mathematical basics which are known to be useful for performing compiler analysis. The first basic is the lattice. The second basic is the fixed point iteration.

2.2.1. Lattice

A lattice is an algebraic structure. For a lattice $L = (V, \sqcup, \sqcap)$, there are the following rules:

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

Additionally we add te requirement of $|V| \neq \infty$. This is not a requirement for lattices in general. However, for fixed point iterations this requirement is precise enough.

The element, which is the least of all elements is called $\perp$. The greatest of all is called $\top$. A lattice can also be written down as a Hesse diagram, which can be seen in Figure 3.1

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

2.2.2. Fixed point iteration

Fixed point iteration is a method for finding a $\tilde{x}$ where we know that $f(\tilde{x}) = \tilde{x}$. Given a none-empty set D, where $|D|$ is finite. And $f : D \rightarrow D$, where f is monotone. And a $x_0 \in D$ which we call start point.

We define $x_i = f(x_{i-1})$ for every $i > 0$. Now we can say: $\exists i \in N : x_i = x_{i-1}$. For further reading, please see [5].

Data flow analysis Data flow analysis are the connection of lattices, FIRM graphs, and a fixed point iteration. As seen before, a FIRM graph consists of nodes. After our data flow analysis finished we want to have a value from the lattice for each node. So for the data flow analysis we need the following:

- A lattice called L
- $f_{node} : L \rightarrow L$, which is the f from a fixed point iteration. Each function is unique for each node, and also depends on the operands of the node.

In the beginning each node of the FIRM graph is associated with the $\perp$ value from the lattice. After that we call f_{node} of each function, the result of f_{node} is then associated again with the particular node. We repeat this last step as long as the results of some f_{node} are changing. At the point where no result changed, we can know that the values that are associated with the nodes are useful and correct. Additional readings for this can be found in [6].

2.2.3. Worklist algorithm

A implementation of a data flow analysis is called worklist algorithm. The base idea of the algorithm is to only calculate those nodes, where a operand changed. This idea is based on the fact, that the state of a node can only change when the state of its operands are changing. A pseudocode implementation 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 $\in$ node.operands do
            if !worklist.contains(operand) then
                worklist.append(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 yet recalculated. The node is not considered dirty after it was recalculated. Additional information can be found at [7].

3. Design & Implementation

3.1. Bitwidth analysis

The analysis is implemented with the worklist algorithm. 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 value as *is positive.*
What stable here means can be observed when looking at 2.3. For the first line of the table we have five stable digits. The second line has 7 stable digits.
With the assumption that both numbers are defined as a signed int. The tuples would be $(4, true)$ and $(6, false)$ for the two numbers from the table.
The whole algorithm is iterative, which means, we might want to address the current tuple x of a node, 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 not used. The second value of the tuple indicates if the value will ever reach negative numbers or not, it is only meaningful for modes that allow signs. We note down the stable bits of a node as $stable\_nodes(n)$, same for $is_positive(n)$. Sometimes we don't want to have the used bits instead of the stable bits, in this case we note the result as:$used\_bits(n) := mode\_bits(n) - stable\_bits(n)$ Where $mode\_bits$ expresses the amount of bits per node.

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:

$$max_{bitwidth}(n) = \begin{cases} 2^{used\_bits(n)-1} - 1 & mode.signed \\ 2^{used\_bits(n)} - 1 & Otherwise \end{cases}$$

$$\begin{tikzcd}
 & \top := (0, false) \arrow[dl, dash] \arrow[dd, dash] \\
(0, true) \arrow[dd, dash] & \\
 & (1, false) \arrow[dl, dash] \arrow[dd, dash, dashed] \\
(1, true) \arrow[dd, dash, dashed] & \\
 & (n-1, false) \arrow[dl, dash] \arrow[dd, dash] \\
(n-1, true) \arrow[dd, dash] & \\
 & (n, false) \arrow[dl, dash] \\
\bot := (n, true) &
\end{tikzcd}$$

Figure 3.1.: Hesse diagram of the lattice which is used in this analysis

$$min_{bitwidth}(n) = \begin{cases} -2^{used\_bits(n)-1} & mode.signed \wedge \neg is\_positive \\ 0 & Otherwise \end{cases}$$

Analysis The analysis works as a fixed point iteration, implemented with the work list approach. You can find the used lattive in Figure 3.1. 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 $\bot$ 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.

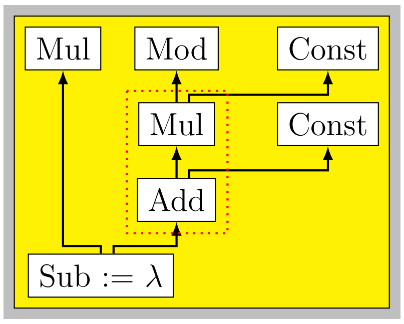


Figure 3.2.: A subgraph of a FIRM graph. The result of $\xi(\lambda)$ is highlighted in red

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 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 3.2. We define ξ to explore the whole tree of nodes with one constant and one none-constant operand, and returns 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 3.2 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 $\perp$ and moves into the direction of $\top$. 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 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.bot 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 $g \in \kappa(e, j)$. Those additional confirm nodes then must get there bound adjusted, so the confirm nodes do make sense with the arithmetical operation between them.

3.2. Stable Conversion nodes

In *lib*FIRM 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 representing 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 $(unsigned)((int)10)$.

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.
One example for this is the confirm insertion algorithm of *lib*FIRM, it searches for assertions that can be made based on looking at compare nodes. This works quite well. However, the example in Figure 3.3 does not work. The insertion code can only insert a Confirm between the Compare and the conversion. However, this is not useful for other analysis, therefore the Confirm node should be before the Conv node. After removing the conversion node, the analysis can find an assertion based on the compare node. Other examples where this can improve things is in the dead

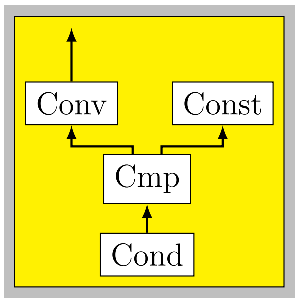


Figure 3.3.: A subgraph that falls into the definition of a conversion compare construction

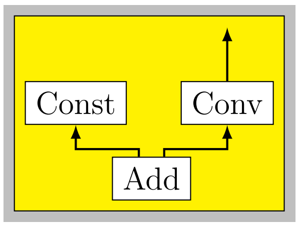


Figure 3.4.: A subgraph where the arithmetical optimization can be performed.

code elimination, and the branch prediction. Next two paragraphs are explaining two concrete optimizations for removing conversion nodes.

Compare-Conversion optimization The rule in *lib*FIRM is that 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 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.

Arithmetical-Conversion optimization The situation of arithmetical nodes, with one operation being constant, one being a conversion, can also be optimized. In this case we move it through the arithmetical operation, and place it afterwards. At the same time we change the mode of the constant and the arithmetical node to the not converted mode. 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 3.4 . After we adjusted the mode of the constant we can move the conversion from the operand of the arithmetical operation to the end.

```
int op(char i, char y) {
        return (int)i + y;
}
```

a Sample c code

node	unoptimized	optimized
node230	8	3
node231	32	6
node228	8	9
node229	32	9
node232	32	9

b VHDL defined variable sizes

Figure 3.5.: firm2vhdl example

3.3. VHDL generation

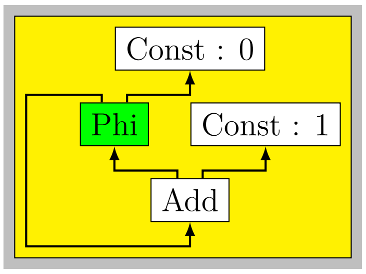
There is a *lib*FIRM tool called firm2vhdl. The tool takes the output from the *cparser* compiler, and outputs the VHDL. Every node in the firm graph gets transformed into a VHDL statement. This is done by transforming the operation of the nodes into VHDL code. Each result of a node operation is assigned 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.

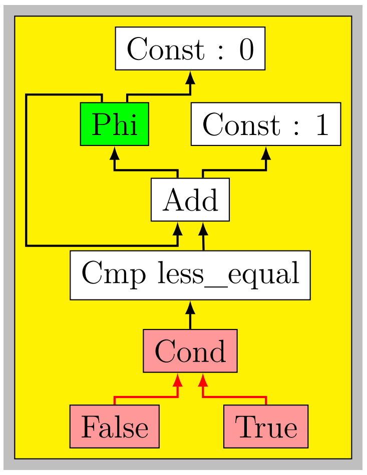
As an example, the C code from 3.5a can be compiled with the tool. Without the optimization the 5 created nodes will have the bit numbers from the first column in 3.5b. However, after we applied out bitwidth analysis, we have the bit usages from the last column in 3.5b.

3.4. Value range vs. bitwidth

The analysis explained in 3.1 is using rules which are based on the bitwidth. The same algorithm could use rules that are working on the range, and not the bitwidth. The later is called value range propagation. The result of both analyses are similar.



a Loop configuration where no assertion is possible



b Loop configuration where a assertion is possible

Figure 3.6.: Subgraphs that are highlighting two different kind of loops in *lib*FIRM

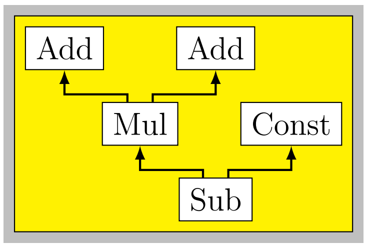


Figure 3.7.: A arithmetical chain, where narrowing would be possible

However, a master thesis from MIT ([8]) claims that the value range propagation is more accurate than the bitwidth analysis, and not only because of the rounding up.

However, the thesis misses one essential fact. In case of a loop like 3.6a, where no assertions can be made, both analyses share the same result: The nodes will use their whole bitwidth / range. However, the difference between the two analysis here is, that the VRP will need 2^{msb} iterations, while the bitwidth analysis only takes msb iterations, where msb expresses the number of the maximum bit. Thus the bitwidth analysis produces the same result, while it's a lot faster.

Another interesting situation is when a Confirm node can be inserted in a loop (See 3.6b). If this is the case, then the bitwidth analysis will return $2^{\lceil log2(bound) \rceil}$ while the VRP would return $log2(bound)$ as the biggest number required by the code. What this shows is, that the VRP here is only more accurate, because the data structure of the bitwidth info operates on bits, while the VRP operates on normal numbers. The runtime here is the same.

The only configuration where the VRP returns constantly a more accurate result, is when arithmetical operations are not looping, but are rather used as some sort of tree structure, like it is illustrated in 3.7. Here the VRP is more accurate, the runtime is the same. However, this drawback can be fixed, and will be covered in a later section.

4. Evaluation

For this thesis, libjpeg, libpng and libgif and zlib are evaluated. zlib is often use as a general purpose lossless compression. The other libraries are used for decoding the most used image formats.

For those projects we are measure two things. First we check how much bitwidth is really used, compared to the full modes. Additionally, we check what impact this has on our assembler generation. Second, we are checking if the saved bitwidth does impact the VHDL outputting of *firm2vhdl*, this is done by compiling the generated VHDL files with two different FPGA-IDEs.

4.1. General bitwidth

A node in *lib*FIRM uses an amount of bits, the maximum is defined by the mode. We note down the bitwidth of a node as $bitwidth_{irn}(n)$. 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 defines the bitwidth of a node. For comparing the functions directly, we define:

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

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 ratio can be found in the third column of Figure 4.1.

The problem with a Constant node is that the bitwidth can be quite low, while this does not improve the assembler generation, nor impact the overall usage of registers.

Library	Nodes	Percent of Nodes
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.: Node statistics of the project.

	with Const		without Const	
Library	mode usage(0)	bitwidth usage(0)	mode usage(1)	bitwidth usage(1)
zlib	44.0928	21.7587	40.6729	31.4571
libjpeg	44.8540	19.5139	40.7650	27.8380
libpng	37.1493	16.7050	32.7320	23.5180
libgif	41.0484	17.6774	36.5323	27.7742
libtiff	41.1068	18.5768	37.1373	27.6623

Figure 4.2.: Bitwidth saving statistics from the project

Calculating how much bitwidth a constant takes is anyway only the matter of a log2(...) call, which is not really interesting to evaluate, 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 at 4.2 are divided into results with and without the constant values.

Project results In Figure 4.1 the second column shows the total number of nodes that were 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 save 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 disassemblies of the shared object files are compared. However, comparing those two shared object files is quite hard, due to changes in addresses and instruction order. 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 probably 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 [1].

4.3. Optimization VHDL improvements

We use the code at Figure A.1 for evaluating the VHDL generation improvements. The saved bitwidth from the analysis can be observed in Figure 4.3. The unoptimized VHDL code has a summed bitwidth usage of 928, while we drop this usage down to

[1]`https://github.com/marcelhollerbach/bachelor-data`

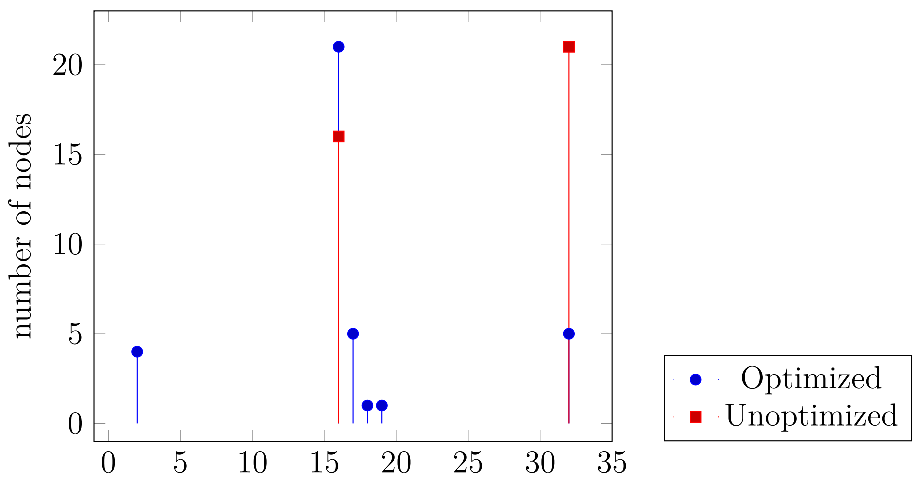


Figure 4.3.: Comparison of bit usage before and after the optimization, for the code in A.1

-	No optimization	Optimization	Hand optimization
LUTs	108	112	109
Registers	20	21	21

Figure 4.4.: Statistics from the vivado projects

609 when optimized. This means on a average base unoptimized 25.1 and optimized 16.91.

After the C snipped was compiled into VHDL, the Xilinx Vivado 2018.2 IDE for FPGA generation was used to compile the VHDL code into a bitstream. A second try was done to compile the VHDL with Quartus. Quartus is a FPGA SDK from Intel Altera.

Vivado In Vivado two projects of the same kind have been created, one with the VHDL file without the improved *ir2vhdl* tool, one with them. Both projects have been using the Kintex UltraScale+ xcku5p device. After the syntesis was running on both projects we took the statistic table from the report. We compared the number of LUTs and registers. In Figure 4.4 you can observe in the first two columns that the unoptimized code has a much lower number of LUTs and registers. A quick look at the optimized VHDL code shows that there are a lot of redundant *resize* calls. After that observation the decision was made that the redundant calls can be removed by hand right now, as from the beginning on we thought the VHDL compiler would take care of removing those. The results of this third try can be observed in the last column. All in all this was not improving the results at all. A deeper look showed that redundant resize calls are not handled by this VHDL compiler. Even after removing them, the amount of LUTs was still bigger than in

-	No optimization	Optimization
ALMs for LUT	54	54
ALMs for Registers	4	4

Figure 4.5.: Statistics from the quartus projects

the beginning.

Quartus As in Vivado two projects have been created. Both projects are using the Board Intel Altera Cyclone V as their target device. After the "Compiling and Fitting" build step, the two "Resource Usage Reports" have been compared. What can be observed in Figure 4.5 is that the optimization does not impact the numbers of ALMs used. The hand optimized code was resulting in the equal results as the optimized VHDL output, and is not explicitly listed therefore. Quartus provides an additional setting where the "Compiling and Fitting" steps are more focused on performance or area usage. Tweaking those setting did not impact the differences between the two generated VHDL files.

Overall we can see that it's quite hard to measure the improvements the VHDL optimization gives, due to the missing insight into the processes that are performed in order to compile a VHDL file into a bitstream.

5. Conclusion

5.1. Assembler generation

The previous chapter showed that the optimizations are not that helpful on decoder code. However, other projects like UIs 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 improving the overall assembler generation.

5.2. VHDL generation

Its similar to the assembler generation. There is no obvious improvement over the not optimized VHDL code. However, its very hard to see real differences due to the inability to understand the real differences in the bitstream. A open bitstream standard would help here, since the projects then could be compared on the lowest possible hardware layer.

5.3. Further improvements

5.3.1. Widening & Narrowing

Widening and Narrowing is a technique that tries to achieve the same or slightly worse results while maintaining a better runtime. Here are a few things that could be done in this analysis to achive this. However, the time was short and thus the following is only thought about, but not yet implemented. Statistics from the projects

Widening not terminating loops There is the theoretical question of when we are able to predict that a loop will terminate before the whole bitwidth is used or not. The question for this analysis is quite simple, if there is a Compare node in the loop, and it is defining an upper bound, then we might find an upper bound when we insert the confirm node there as described in section 3.1.1. If there is no Confirm or Compare node, then there is no chance for the loop to terminate. And thus we might want to set our stable bits to 0.

Narrowing for arithmetical chains As explained in 3.4, the VRP returns a more accurate result for arithmetical chains. An arithmetical chain can be defined as a set of arithmetical nodes, where each node has a successor which is a arithmetical node.

The problem with arithmetical operations within the bitwidth analysis is, is that we need to calculate the worst case in terms of bit usage. As an example, two operands with the ranges [0..2] and [0..4] will result in the range [0..6]. However, we are in a bitwidth analysis here, which means the ranges here are always rounded up to the next power of two. This means for our example, that the two operands will have the same ranges, but the result of the addition will be [0..8]. In higher bit ranges, the loss is even bigger.

The problem can be fixed by calculating the result for a arithmetical node as the bitwidth information, as well as the exact range. The successor of the node can then use the exact range instead of the bitwidth info for calculating its results. Thus we end up in the same result as the VRP. However, we still would only do this for arithmetical chains, thus we still would be faster when there is a loop, since the bitwidth informations would be rounded up after the end of the chain.

5.3.2. More conversion nodes

In this thesis we looked at removing the conversion nodes where we can. However, in some situations it might makes sense to insert more conversion nodes where most of a mode is not used, and thus the reduction to a smaller mode is possible. A example is a shift operation on 32bit system. Implementing a 64 bit shift on a 32 bit system adds a overhead, which might not be required.

Bibliography

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

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

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

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

[5] O. Veblen, "Continuous increasing functions of finite and transfinite ordinals," *Transactions of the American Mathematical Society*, vol. 9, no. 3, pp. 280–292, 1908.

[6] U. Khedker, "Theoretical abstractions in data flow analysise." Online handout, 2027.

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

[8] M. W. Stephens, "Bitwise:optimizing bitwidths using data-range propagat," tech. rep., 2000.

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. Appendix

A.1. Table of node rules

A operation has operands, all of them are referred to as *operands*, the first is called a, the second is called b. In the first column the formula for the new stable digits is displayed. The second column is the formula for the is_positive flag. n_{stable} is used in the second column to refer to the value calculated in the first column.

op	used bits	is_positive
add	$max(min(a_{stable}, b_{stable}) - 1, 0)$	$a_{positive} \wedge b_{positive} \wedge n_{stable} > 0$
minus	$max(a_{stable} - 1, 0)$	$false$
sub	$max(min(a_{stable}, b_{stable}) - 1, 0)$	$false$
mul	$max(b - mode + a_{stable}, 0)$	$a_{positive} \wedge b_{positive} \wedge n_{stable} > 0$
div	$\begin{cases} a_{stable} & , mode \text{ is signed} \\ max(a_{stable} - 1, 0) & , \text{otherwise.} \end{cases}$	$a_{positive} \wedge b_{positive} \wedge n_{stable} > 0$
mod	$\lfloor log2(max(a)) \rfloor$	$a_{positive}$
shl	$a_{stable} - b_{stable}$	$\begin{cases} a_{positive} & , n_{stable} > 0 \\ \text{false} & , \text{otherwise.} \end{cases}$
shr	$a_{stable} + b_{stable}$	$true$
shrs	$a_{stable} + b_{stable}$	$a_{positive}$
conv	$max(a_{stable} + (mode - old\_mode), 0)$	$a_{positive} \wedge n_{stable} > 0$
max phi and eor or	$min(operands)$	$\bigwedge(operands_{positive})$

A.2. Evaluation C source code

```
uint32_t test_atom(uint32_t input0, uint32_t input1) {
int16_t inp0, inp1, inp2, inp3,
abs0, abs1, abs2, abs3;
#define ABS(x) ( x<0 ? -x : x )

inp0 = input0;
inp1 = input0 >> 16;
inp2 = input1;
inp3 = input1 >> 16;

abs0 = ABS(inp0);
abs1 = ABS(inp1);
abs2 = ABS(inp2);
abs3 = ABS(inp3);
return abs0 + abs1 + abs2 + abs3;
}
```

Figure A.1.: Test code for measuring VHDL generation improvements