Automatic Translation from Binary Object Files to Hardware Description Languages

Studienarbeit im Fach Informatik

vorgelegt von

Alexander Würstlein

geb. am 15.1.1982

Angefertigt am

Lehrstuhl für Informatik 4 (Verteilte Systeme und Betriebssysteme)

Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuer:	Prof. Dr.-Ing. W. Schröder-Preikschat Dipl.-Inf. M. Gernoth
Beginn der Arbeit:	2.3.2011
Abgabe der Arbeit:	4.7.2011

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, den

Zusammenfassung

Diese Arbeit beschreibt das Design und die Implementierung eines Werkzeugs zur automatischen Übersetzung von Datenstrukturbeschreibungen aus in Objektdateien enthaltenen Debug-Informationen in Hardwarebeschreibungssprachen. Vorhandene, bewährte Technologien wie *DWARF* und *VHDL* auf neuartige Weise verbunden um Hilfe bei Problemstellungen des Einsatzes von FPGAs zur Verbesserung nicht-funktionaler Eigenschaften von Betriebssystemen zu gewähren. Verschiedene Gesichtspunkte der Einsatzumgebung, der Arbeitsweise und insbesondere weiterer Anwendungsgebiete des Werkzeugs werden eingehend beleuchtet. Auch Erweiterungen und Erweiterungsmöglichkeiten auf andere Ausgabesprachen und -formate werden ebenso diskutiert wie mögliche Probleme in der Anwendung.

Abstract

This work describes the design and implementation of a tool to automatically translate data structure descriptions, extracted from debug information embedded into object files, into hardware description languages. Existing proven technologies like *DWARF* and *VHDL* are used in novel ways, thereby tackling certain challenges in the overall context of improving non-functional properties of operating systems through the use of FPGAs. Various aspects of the usage environment, the inner workings and of further possible uses are discussed in detail. Also described are current and possible extensions to other output formats and languages and potential problems during use.

Contents

1 Introduction

1.1 Motivation and Objectives

First I will provide a broader overview of the context in which *st* is intended to be used and describe what *st* will be used for in this context.

Within the last years, relatively cheap FPGA hardware has become available, for example as PCI cards, as components in certain embedded systems and even as co-processors on current desktop processor chips[1]. In the hopes of increasing raw computing power, making designs more flexible and reducing power consumption this trend will most probably continue. FPGA applications like signal processing or parallel computing are predominant, but of course the list of possible applications is far longer than that.

This leads to the question of how operating systems could benefit from the usage of FPGA hardware. System functions that require a large amount of processing power like disk and network encryption are the most common answer to this question. While these functions can as easily be executed on the CPU as on the FPGA, thus giving the system no new functional properties, speed of execution and energy consumption are improved. Speed of execution, energy consumption, security, response time and adherence to response time limits are examples of non-functional properties of a system. Thus moving tasks to FPGA hardware can predominantly be used to improve the non-functional properties of an operating system.

I will give three examples of how one could imagine using FPGA-augmented functions in operating systems. Those ideas are still actively being developed and should thus be expected to change or perhaps to be found impractical:

- The operating system scheduler could be replaced by a scheduler implemented in hardware. A scheduling decision would then be written to main memory by that hardware through DMA and the process switch would be initiated by use of an interrupt. The intention would be to improve the real-time capabilities of the system.

- Routing decisions which involve searching large routing tables could be moved to FPGA hardware. The hardware could then, through the use of an inherently high degree of parallelism, speed up routing table lookups, thereby improving routing performance.

- A DMA-capable hardware may perform anomaly detection on the behaviour of the operating system and its processes. By those means unauthorized software such as viruses, root kits or breaches of a pre-defined security policy such as processes

running with undesired privileges may be detected, improving the security of the system.

1.1.1 Adaptation to Changing Operating System Kernels

One of the primary challenges when moving operating system functions to an FPGA is the interface between the operating system kernel and the hardware. For all tasks that are moved onto an FPGA, a communication between the operating system components running on the FPGA and on the CPU needs to be established.

A communication method for this scenario needs to overcome a number of obstacles. First and foremost, the communication should not negate the benefits provided by the FPGA. Thus a communication method which incurs a high cost in e.g. processing power, energy consumption or latency is out of the question. The communication needs to be achievable with current off-the-shelf hardware. The changes required to the operating system to be extended should be kept to a minimum, or, if possible, the CPU part of the operating system should not have to be changed at all.

As in communication with other peripheral devices, the most promising technique to fulfil those requirements seems to be DMA. By DMA, a device can directly access the main memory where the CPU places its programs and data.

The question is then about the data exchange format between the CPU and the FPGA. Various approaches could be used here: A communication protocol could be established where each item is properly encoded into some commonly defined format which the FPGA and CPU data formats have to be translated to. Or the CPU could adapt to the data in such a way that it fits the internal representation used in the FPGA core. Lastly, the FPGA core could be enabled to interpret the data structures used in the CPU part of the operating system.

Deciding which data exchange format is most suitable seems easy. A common communication format would provide the most generic interface, but would hardly fulfil the requirements specified above: data would need to converted twice, both on the FPGA and in the CPU, and the code running on the CPU would need to be extended significantly to perform this conversion. Similarly, adapting the code running on the CPU to fit the data formats used by the FPGA would also require unacceptably large changes to the operating system code. Thus the only workable way of communication is for the FPGA to be able to understand operating system data structures as current implementations of e.g. a vanilla *Linux* kernel use them.

This creates a problem: Since the FPGA core would need to read and write internal data structures from main memory written by the currently running operating system, the core must be adapted to each operating system it should cooperate with. This means that for each operating system version and configuration that changes a data structure accessed by the FPGA, the FPGA core needs to be adapted. If in a *Linux* system, the kernel is recompiled with a different compiler or different configuration options, structure members will be moved around, some will become available while others vanish.

It is clear that this situation is not manageable without proper tools to automatically generate the interface code in question. A tool is needed that is able to extract a

description of the data structures of the operating system and translate that description into a part of the FPGA core that implements the interface to the CPU part of the operating system. The design and implementation of this tool —named *st* for "structure tool"— is the intent of this work.

1.1.2 Scope of this Work

The purpose of this work is to build a tool to support the aforementioned automated adaptation of hardware interfaces.

Since *VHDL* is the preferred language for FPGA core implementations in the larger project for which *st* is being developed, *VHDL* is the natural choice for the output language *st* should generate.

There are several considerations as to which input format *st* should work on. The data structures are described first and foremost by the operating system source code. Usually, `struct` data types are used to form compound data types which describe the state of various kernel objects. The definitions of those `struct` data types could be extracted from the operating system source code by *st*.

But this approach would suffer from two problems: sometimes the operating system source code is not readily available. While for *Linux* as an open source system this would not occur, there is another complication that also occurs with *Linux*: the memory layout of a `struct` data type is not only defined by the source code, but also by the compiler in use.

The compiler is allowed to change the in-memory layout of data structures within certain bounds. This means that *st* has to either emulate the behaviour of every compiler version with regard to the memory layout, or that *st* has to work off the compiled object file that the compiler outputs. In this work, the latter approach is chosen as shown in figure 1.1.

This is accomplished by reading the type information contained within the debugging symbols embedded in object files. This type information is then searched for structure types. The information gathered about the memory layout of those structure types is used to automatically generate the *VHDL* description of an address generator capable of calculating the member addresses of each of the structure's members. This address generator can then be used as a building block in an FPGA core.

1.2 Technologies

1.2.1 The VHDL Hardware Description Language

In this work, *VHDL*[2, 3] is used as a target language for *st* to output. The *VHDL* components output by *st* are intended to be included into hardware designs that use those components as interfaces to help interpret structures in memory.

VHDL is a hardware description language. It was developed through the initiative of the U.S. Department of Defense's *VHSIC*[4] program as a description language for ASIC designs.

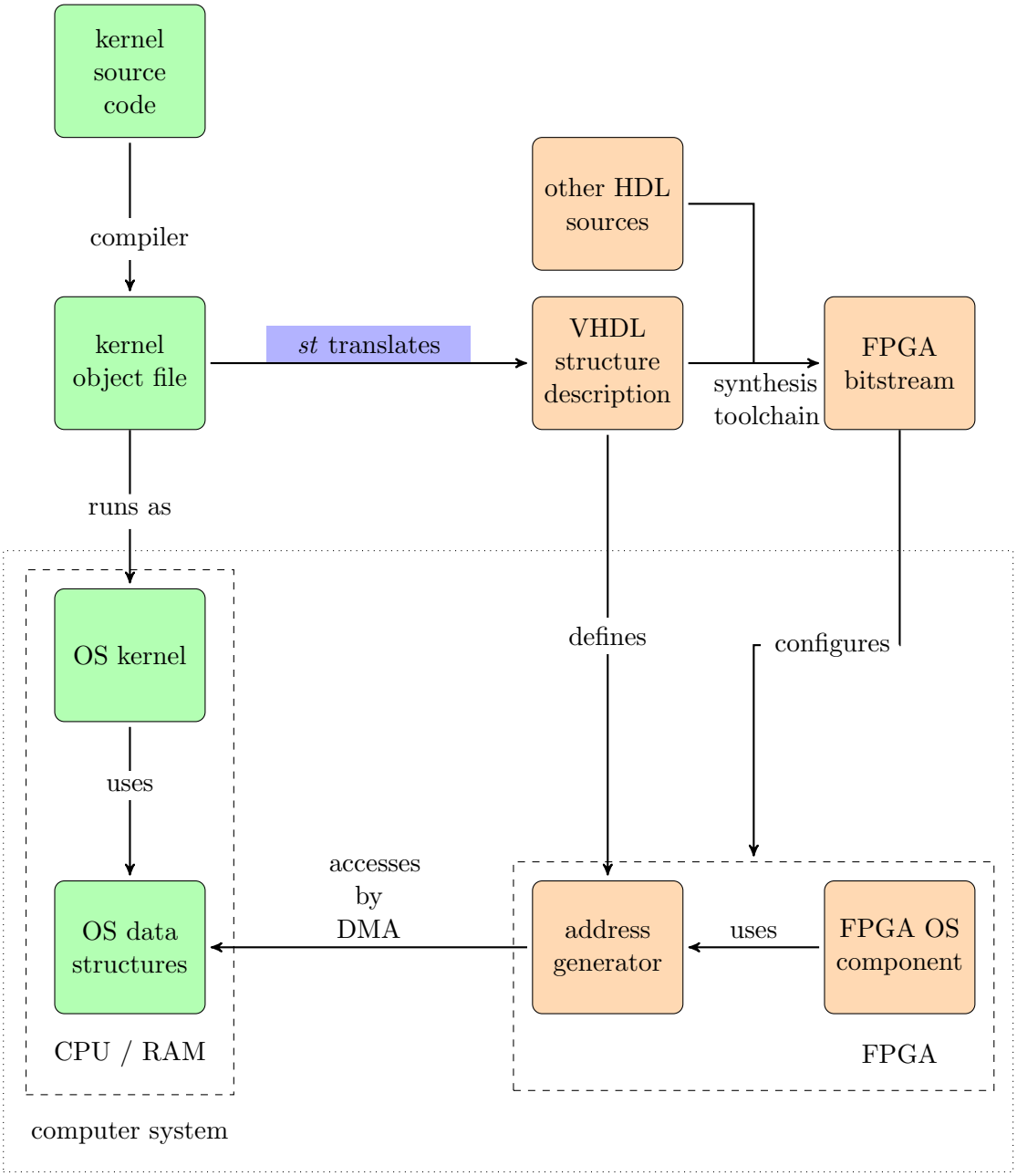


Figure 1.1: Usage scenario for the *st* tool. The green components to the left represent a traditional operating system with a kernel running on a CPU with its internal data structures in RAM. The role of *st*, marked in blue is to translate the structure descriptions found in the kernel object files into *VHDL* which is then used to program the FPGA component of the computer system. There the resulting address generator allows access to the operating system data structures for operating system components implemented in the FPGA, marked in orange to the right.

VHDL describes the data flow between the components of a hardware design. On the lowest level, such a component may be a logic gate like `AND` or `XOR`, but constructing higher level components like a PCI[5]-controller or a processor from simpler parts is of course possible and common. Since *VHDL* describes the wiring of a circuit, all processes described are happening in parallel. Serialization is accomplished implicitly through the serial nature of the data flow and explicitly through the use of clock signals which create artificial synchronization points in time. "Wiring up" a *VHDL* circuit is accomplished via "signals" and "variables" which are analogous to physical wires connecting various elements of the design as shown in figure1.2.

VHDL designs are organized into reusable modules called "entities"[1]. An entity has an interface described by its "port". The port specification of an entity is analogous to the pinout of a physical chip. An entity is used by connecting its interface ports in a port "map" to the appropriate signals and variables of the surrounding module. The behaviour of an entity is specified by one or more "architecture" specifications. This is usually accomplished by specifying the internal structure by nesting and wiring smaller and smaller entities down to the logic gate level. For purposes of simulation or verification it is also possible to specify architectures in other programming styles, e.g. by specifying the timing behaviour in a procedural manner in terms of state changes interspersed with delay instructions.

To create hardware from a *VHDL* design, the design must contain a complete set of architectures that are synthesizable. This means that a synthesis tool must be able to infer a circuit layout from the given design. Therefore for specifying real hardware, only a subset of *VHDL* defined by the capabilities of the synthesis tool is usable. Usually that subset is limited to certain data types representing bits, arrays of bits and the compositions of such bits through logical operations and simple conditions.

1.2.2 Programmable Hardware

Aside from hardware designs for fixed integrated circuits that are physically produced in silicon and that are immutable after production, *VHDL* is also ideally suited for programming "flexible hardware".

Such "flexible" or "programmable" hardware comes in the forms of "CPLDs" or "FPGAs". CPLDs are grids of logic gates, that are connected by programmable fuses, programmable meaning that those fuses can be set to a connected or disconnected state through a programming input. The programming of those fuses can of course be reset and reprogrammed later on. Given a large enough grid of logic gates and a flexible enough interconnecting grid of fuses, any logic circuit can be programmed into such a device. FPGAs also contain a interconnecting grid with programmable fuses, but the primary logic functions are implemented by lookup tables Lookup tables are programmable memory elements with some number n of inputs and m outputs. The lookup table is programmed by specifying the value of all m output bits for each combination of the

[1] The *VHDL* keyword for an entity is also `entity`. Similarly the VHDL keywords for `port`, `port map` and `architecture` are also equal to their terms in common parlance.

```
entity C is port(
  ci1, ci2, ci3: in std_logic;
  co1, co2, co3: out std_logic;
);
end entity C;

architecture example of C is

signal tmp: std_logic;

component Atype
  port( ai1, ai2: in std_logic;
    ao1, ao2: out std_logic );
end component;

component Btype
  port( bi1, bi2: in std_logic;
    bo1, bo2: out std_logic );
end component;

begin

A: Atype port map(ci1 => ai1, ci2 => ai2,
  ao1 => co1, ao2 => tmp );
B: Btype port map(tmp => bi1, ci3 => bi2,
  bo1 => co2, bo2 => bo3 );

end architecture example;
```

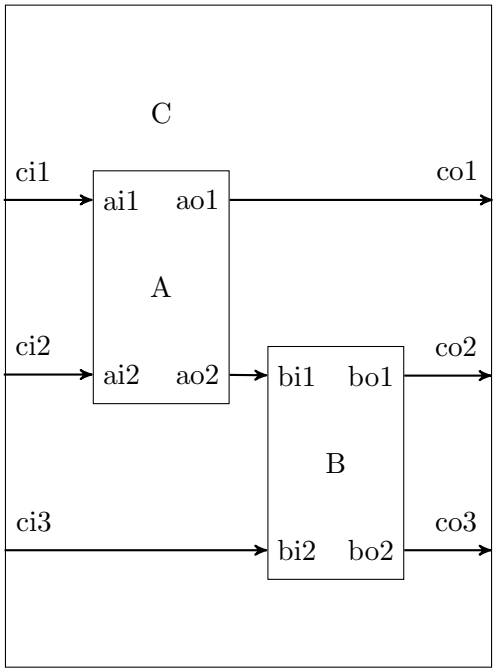


Figure 1.2: Examples of a VHDL source code on the left and the resulting component connections on the right. The source code shows the entity declaration for a hypothetical component named `C` which is built from two components of type `Atype` and `Btype` with instances named `A` and `B` respectively.

n input bits, giving the truth table for the boolean function it represents. The LUT therefore needs a memory capacity of $2^n \cdot m$ bits.

1.2.3 The DWARF Debugging Format

DWARF[2][7, 8] is a debugging file format originally designed for the *System V Release 4 sdb* debugger. Three improved versions of the original *DWARF* format were standardized. All versions of the *DWARF* standard are still in use in different domains and niches.

For this work, the relevant revision is *DWARF 3*. *DWARF 4* has been specified in 2010[9] but is of course only supported in most recent compilers and tools. Therefore this work still uses the older version 3. Since *DWARF 4* is specified to be compatible to *DWARF 3*, the tools produced within the scope of this work should continue to work with *DWARF 4* input data.

The purpose of a debugging file format is to standardize the embedding of debugging symbols into object files. Debugging symbols may consist of variable and function name

[2]The name *DWARF* is a play on the acronym *ELF*[6] of the "Executable and Linkable Format" (formerly "Extensible Linking Format") object file format.

and type annotations, line number and filename annotations as well as descriptions of data types. To enable the use of a wide range of debugging tools, possibly from vendors other than the compiler vendor, a number of standards exist to encode debug symbols. Examples are the *stabs* format, common in *Unix* environments, *COFF*[3] which is also used by *Microsoft Windows* systems and of course *DWARF* which is not only the current standard in the *Unix* world but also widely used e.g. for embedded systems development.

The *DWARF* standard specifies only the data format for the debugging symbols embedded in the resulting object file. The *libdwarf*[8] library is used to access *DWARF* symbols in the software implemented for this work. *libdwarf* provides a comprehensive, low-level view of the relevant data while abstracting the irrelevant details of encoding and embedding.

Since the scope of this work only encompasses using *DWARF* symbols through *libdwarf*, I will refrain from discussing internal details like the binary format used by *DWARF*. Instead I will limit the discussion to the concepts and interfaces necessary for understanding the code produced.

DWARF data is organized in units called *DIE*. *DIE*s are organized in a tree structure, with one compilation unit[4] represented by one such tree. Each *DIE* contains and is described by one "tag" and one or more "attributes". A tag denotes the specific type of a *DIE*, for example whether the *DIE* represents a compilation unit, function, variable or data type. Attributes add specific details to the information the tag provides us with. For example a *DIE* that represents a compilation unit will be tagged `DW_TAG_compile_unit` and contain attributes like `DW_AT_name`, `DW_AT_language` or `DW_AT_producer` specifying the file name, the source language and the compiler used respectively. Attributes can be of a more general kind, applicable to many different types of *DIE*s, like for example `DW_AT_name`, or they can be specialized, applicable to only a subset of all *DIE* types. The data an attribute carries can be of one of several classes, a class being e.g. "string", "address" or "constant".

*DIE*s may contain or reference other *DIE*s. As an example, a *DIE* representing a compilation unit will contain all *DIE*s describing that specific compilation unit. This "contains"-relation forms the tree of *DIE*s mentioned above. In addition, references can be used to point to other *DIE*s independently of the sibling/child-relation formed by the tree, thereby forming a directed graph. As an example, each declaration of a variable points to the appropriate *DIE* describing the type of that variable.

Correspondingly, *libdwarf* contains functions to traverse the *DIE* tree, look for certain attributes and extract their respective data values.

[3] *COFF* is a complete object file format including debug symbols. *DWARF* on the other hand is independent of the choice of an object file format, although the most common combination is to use *DWARF* together with *ELF* object files.

[4] Usually a compilation unit is a single object file generated from one source file. After the linking stage, one file will probably contain more than one compilation unit.

1.2.4 The C Programming Language

The *C* programming language[10, 11, 12] was designed and currently remains in use as a language for the implementation of operating systems as well as systems programming and general low-level tasks. Owing to its simplicity combined with great flexibility, it remains one of the most popular programming languages in current use.

Reasons for choosing C

Within the scope of this work, the use and importance of *C* is two-fold: First, *C* is used to implement the software *st*. Second, the data model and memory layout of *C* is used as a model, testing environment and primary target for the development and testing of *st*. In this chapter I will elaborate on the reasoning for choosing *C* in both roles.

For the implementation of *st*, a number of implementation languages would have been viable. Aside from bindings to `libdwarf`, demands on the features provided by the language are low. While advanced text processing and text templating capabilities such as those of *Perl* would have been useful they are not imperative. *Haskell* was also considered, especially since its pattern matching[13] seemed to be a very useful tool for the purpose of implementing *st*.

Yet, the familiarity of future and current users with the language was pivotal in the decision for *C*. Most programmers involved in low-level software projects such as operating systems and hardware designs are at least familiar with or even prefer *C* as a programming language. Also future users within the project *st* was developed for voiced strong preferences towards *C*.

In the role as a testing and model environment *C* is without practical alternative. The primary target is the *Linux* operating system which is implemented in *C*. But also other target systems are often implemented in *C* or sometimes *C++*. Since the data model of *C++* is a superset of that of *C*, implementing support for the *C* data model is also a first step in implementing *C++* support.

Memory Layout of C Data Types

C is often used in the implementation of operating systems, device drivers or network protocols. This is thanks to the fact that the memory layout of data structures can be easily predicted and controlled in *C*. That way, data transmitted over a network or written into a buffer by some hardware device can be easily interpreted: a pointer to the memory location of the buffer is cast into a pointer of a data type suited to the data contained within the buffer. E.g. when receiving some network packet, the `void *` pointing to the beginning of the receiving buffer is cast to a `struct ip_packet *` through which fields within the packet like sending and receiving address, flags, checksum or data are readily addressable and accessible. That way, serialisation and deserialisation of data is practically free, since the cast operation incurs no computational overhead.

The data types *C* implements can be categorized into various groups. Basic types or base types are data types which are defined by most processor architectures through the

registers and instructions —the instruction set architecture— provided[5]. Examples of such base types are `int`, `long int`, `float` or pointer data types like `float *`. Compound types or complex types are compositions of one or more elements which are by themselves either of a base type or a nested compound type. The leaves of the tree defined by the nested types will always be base types. *C* defines three kinds of compound types: arrays, structures and unions.

Arrays An array is a compound type of N (with $N \in \mathbb{N}$) elements of the same type t addressed by an index n (with $0 \leq n < N$.) The memory layout of an array consists of the array elements stringed together starting from a base address a_0. The layout is tightly packed such that the ith element of the array can be found at an address $a_i = a_0 + i \cdot s(t)$ where $s(t)$ is the size of a single element of the array. The size of the whole array is therefore given by $S = N \cdot s$. An array element is accessed via the array index operator in *C*, e.g. the ith element of an array `a` is given by `a[i]`.

Structures A structure or "record"[6] data type is an ordered set of structure members $(\{m_i\} : m_i < m_{i+1})$. Each structure member m_i is a pair $m_i = (t_i, n_i)$ of a data type t_i and a name n_i where the name n_i has to be unique within one structure type: $i \neq j \Leftrightarrow n_i \neq n_j$. The memory location a_i of a structure member is calculated from the base address of the structure variable a_0 and the offset o_i of that member in the structure type by $a_i = a_0 + o_i$. A general rule to calculate offsets o_i however is impossible to specify for reasons that will now be discussed.

The memory layout of structure types is more unpredictable than for array types. The primary reason for this is that on most modern processor architectures, alignment of data is necessary for fast access to memory. "Aligned to an b-byte boundary" means that an address a fulfills $a \mod b = 0$, that is, an aligned address is always an integer multiple of b.

Usually, memory can only be read in specific chunks of e.g. 4 bytes since the physical layout of the RAM chips only allows for reading those 4 bytes in parallel. This means all 4-byte or smaller chunks that are aligned to a 4-byte boundary can be read with one "memory read" instruction, whereas reading "across a boundary" would require two "memory read" instructions and would make subsequent assembly of the value into a register necessary. While those instructions can be automatically generated either by the compiler or the processors instruction decoder, there is always a significant cost in processor cycles associated with unaligned accesses. Additionally, modern processor architectures often employ a multi-level system of caches which are meant to accelerate accesses to the slower main memory or lower levels of the cache hierarchy. These caches are organized in so-called "cache lines" which are the smallest unit of cache that can be filled by reading from main memory. Therefore, on architectures where the performance

[5] Strictly speaking, most ISAs also provide limited support for compound data types through addressing modes which provide the necessary pointer arithmetics. But this does not define the exact representation of such compound types like it does e.g. for floating point formats. Therefore we will ignore that support for the purpose of this discussion.

[6] The term "record" is e.g. used in *Pascal* or *Haskell*.

of memory accesses is dependent on fast cache accesses —which is true for most modern architectures— alignment to cache line boundaries can improve the performance of code with a high dependence on fast memory accesses at the cost of a possibly larger memory footprint.

Alignment can be achieved by two principal means: reordering and padding. Reordering means that the compiler changes the ordering of structure members such that a higher fraction of structure members is aligned. It is easy to see that it is impossible to find a "perfect" reordering for every possible structure and the *C* language standard prohibits the reordering of structure members. Yet for other languages or by manual intervention of the programmer, reordering may of course be successfully applied.

The other, rather obvious and more common option is the addition of "padding" between structure members. Padding is an unused area of memory of a specific size that is inserted after a member m_i, such that the following member m_{i+1} is aligned. So the padding size p_i for the padding that follows the ith member with size $s(t_i)$ and offset o_i is chosen to fulfil:

$$\underbrace{(o_i + s(t_i) + p_i)}_{=o_{i+1}} \mod b = 0$$

Of course this assumes that the base address a_0 of a structure also needs to be aligned because the alignment of member addresses a_i is the required property, the alignment of offsets o_i only helps to achieve it via $o_i \mod b = a_0 \mod b = 0 \Rightarrow (a_0 + o_i) \mod b = a_i \mod b = 0$.

This means that there is no simple way to calculate the offset of a structure member in the common case. Only a lower limit for the address of a member can be given as $o_i = \sum_{j<i} (s(t_j) + p_j) \geq \sum_{j<i} s(t_j)$. In the special case of padding being turned off, e.g. by declaring a structure with `__attribute__((packed))` in *C*, this simplifies[7] to $o_i = \sum_{j<i} s(t_j)$. The overall size of a structure type S is similarly given by $S = \sum_i (s(t_i) + p_i)$.

A structure member named `foo` in a structure variable `bar` is usually accessed via `bar.foo` in *C*.

The `struct` type was also extended to allow for "bit-fields", that is structure members where the size of the member is specified in an amount of bits. That is significantly different from the earlier situation where types could only be specified in the whole-byte sizes of given base types or arrays of base types.

Bitfields allow portions of an `int` or `unsigned int` to be used as structure members. The *C* standard leaves most of the details like the exact arrangement of bit-fields within the containing `int` or `unsigned int` as "implementation-defined". It is only stipulated that adjacent bit-fields should be packed into the same containing `int` if possible and may not exceed the size of an `int`. It is important to note that bit-fields have a two-dimensional address in memory: a byte-offset o_i which is the offset of the containing `int` within the `struct`, and a bit-offset bo_i which specifies the offset in bits from the start of the containing `int`. Similarly, there are two sizes given for a bit-field: The size of the

[7] If the structure contains bit-fields, this simplification might not apply depending on how the bit-fields are "rounded up" to whole bytes.

containing `int` $s(t_i) = s(\texttt{int})$ in bytes as well as the actual size of the bit-field bs_i, given in bits.

Unions Unions are in some ways similar to structures in that union members are pairs $m_i = (t_i, n_i)$ of a type t_i and a name n_i. Access to members is accomplished via the same syntax (e.g. `bar.foo`) as with structures. But other than in structures, all member offsets o_i in a union are 0. This means that all union members share the same memory space. The overall size of a union is given by the maximum of all sizes of the member types: $S = \max_i (s(t_i))$.

Additional data types from C99 The most recent revision of the *C* language standard most commonly called *C99*[12] defines several additional data types and extends the definitions of the previously described data types.

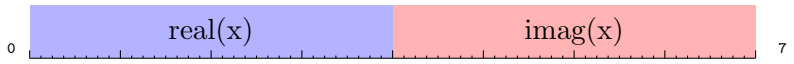


Figure 1.3: Memory layout of a `float _Complex` type variable named `x`.

Where in earlier versions it was common practice to use the `int` data type in boolean expressions, it is now possible to use the newly defined `_Bool` base type instead. *C99* first introduces a standardized data type for complex numbers into the *C* language. Complex numbers consist of two floating point numbers, one representing the real, the other the imaginary part of the complex number. Accordingly, the memory layout of a complex data type defined as `float _Complex` or `double _Complex` consists simply of two `float`s resp. `double`s one after another. An example of this is shown in figure 1.3. In fact, the *C99* standard[12] states that "[each] complex type has the same representation and alignment requirements as an array type containing exactly two elements of the corresponding real type; the first element is equal to the real part, and the second element to the imaginary part, of the complex number."

2 Implementation

2.1 Overall Design

The tools designed within the scope of this work share a common basic design. To be able to easily replace and modify input and output formats as well as to be able to change and filter the intermediate data stream, processing is done in three layers. An input layer handles the input and processing of an object file containing *DWARF* symbols into a common data format. A filter layer operates on that common data format and applies transformations to it. That common data format is then processed by the output layer to generate output in various languages.

All layers are modular. Input and output modules can be swapped out to accommodate different needs in input and output formats. The filters in between the input and output layers are represented by a configurable chain of modules, each of which processes the output of the preceding filter as input. Filtering is optional, so zero or more filters can be used.

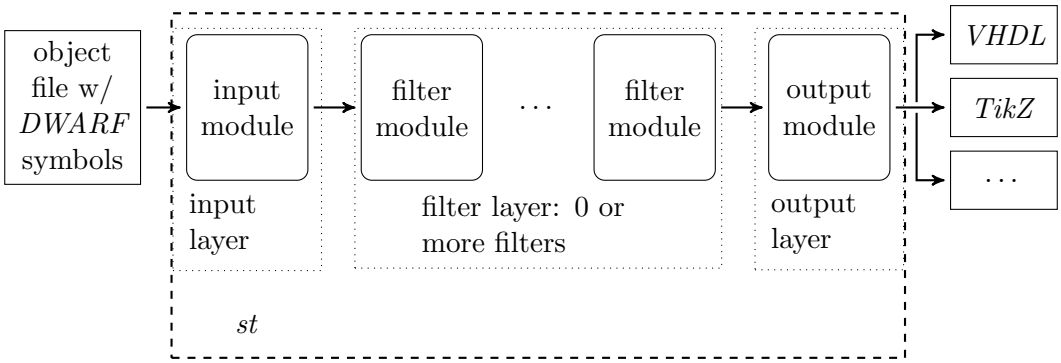


Figure 2.1: Modular architecture of *st*

2.1.1 Input layer

Currently, only one input module is implemented in `read_dwarf.c`. This module reads *DWARF 3* debug symbols from object files in *ELF* format.

After opening the object file, the module searches for all structure declarations denoted by *DIE*s with the tag `DW_TAG_structure_type`. For each structure type that is found, the member types are parsed recursively. A list of trees of all those structure types is then output, the format of which will be described in section 2.1.2.

When reading the types of structure members, there are various possible outcomes:

- The member type can be a *C* base type. A base type is not composed of other types and does not reference other types. Therefore encountering a base type is the simplest case. Examples of such base types are `float` or `unsigned int`.
- A type alias, or as it is more commonly known, a "typedef"[1] can occur. A typedef is just a different name for another type, so its encoding is also simply a pointer to its aliased type and of course the alias name. Please note, that a typedef does not consume any additional memory within the structure. It would be possible to entirely replace the typedef by its referenced type without changing the semantics of the data structure. Yet for the purpose of readability and identifiability of input and output code the information a typedef provides is carried along.
- Pointer types are also commonly used in the *C* programming language. A pointer consists of a memory location where the pointer is stored and a data type which is to be found at the location the pointer references.
- Structures may also contain other structures within them. The contained structure is recursively parsed and the result is referenced together with the name that structure member has been given.
- Unions are parsed in a manner very similar to structures. The only difference is that reading of member offsets is skipped since those are always assumed to be 0 in unions.
- Arrays specify an element type as well as an index range thereby giving the number of array members and the overall size of the array.

In *C* code, definitions of structure types can occur in various contexts. A definition can be global inside a module, it can be valid across various modules e.g. by use of a common header file. But local definitions are also possible, most notably inside a function or another type definition. In several situations it is possible to define structure types which bear the same name or even no name at all[2].No special attempt is made to create unique names or identifiers for those structures. Duplicate names used in different scopes are retained and anonymous structure types are given the special non-unique name "`ANONYMOUS`". A special mechanism for creating unique identifiers would of course be possible. Yet several considerations cast doubt on the usefulness of such a mechanism: anonymous structure types and duplicates could be numbered or tagged with a hash of

[1] The *C* keyword for a type alias is `typedef`, therefore the term typedef has become common parlance.

[2] A structure type without a name is a so-called "anonymous structure type". Anonymous structure types occur when a structure declaration without a structure name is used e.g. in a function or variable declaration. E.g. `struct {int a; int b;} foo;` defines a structure variable `foo` of an anonymous struct containing two `int` members named `a` and `b`. A similarly named but different feature are "anonymous structures"[14] which are unnamed structure-typed members within a structure. Anonymous structures are not part of the *C* standard, but implemented as an extension by some compiler vendors.

their definitions. In that case every reordering or change of those structures would change their naming, making simple reuse of code that is dependent on what is generated by our toolchain impossible. Another approach would name structures according to their ancestry inside the *DWARF DIE*-tree, e.g. prefixing a structure type defined inside a function with the name of the function it was defined in. While slightly more robust than numbering or hashing, this would yield large and therefore unreadable and unusable names for those structure types. Also, renaming structure types in the input stage would prevent output modules from making more intelligent choices depending on the type of output that is to be generated.

Please note that later, output modules do follow naming conventions for their output that e.g. name structure members by prefixing them with their ancestry. For details see section 2.1.4.

2.1.2 Interface

A special format is used to transfer the structure type information from the input to the output module. Theoretically, *DWARF* would be suitable for that purpose. In practice, the idea of using *DWARF* for internal interfaces was discarded for three important reasons:

First, the *DWARF* data format is very complex and therefore not very easy to use. The code size and complexity of the output modules would be considerably increased if *DWARF* were used as an exchange format. While at first glance it would seem that using *DWARF* as an internal format would simplify the input module considerably, that impression is not correct. The input module would still need to perform much the same tasks as it does now, which is pre-filtering the interesting parts of the *DWARF* data. This also means that the *DWARF* data would need to be "repackaged" to a *DWARF* subset in the input module, incurring mostly the same complexity as "repackaging" to the custom format.

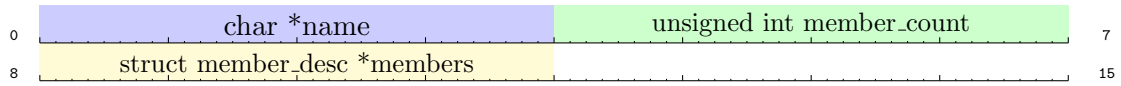
Second, a simplified exchange format creates the opportunity to write different input modules which read formats other than *DWARF* without the considerable complexity of converting that input format to *DWARF*. Also, serializing the simplified exchange format and reading it later would become an option.

Third, the filter layer also hugely benefits from a simplified representation to operate on.

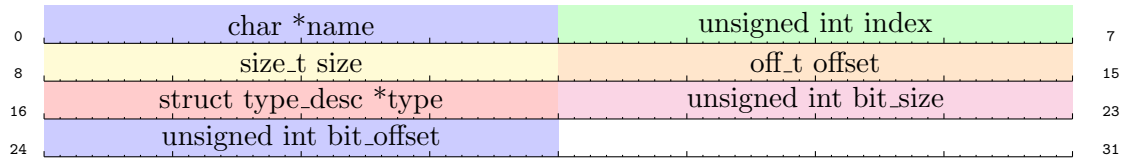
Our tool can be seen as a compiler which compiles from an input format to an output format. In that interpretation, the intermediate format used would be equivalent to the abstract syntax tree a compiler operates on.

The intermediate format is implemented as an array of structures of type `struct struct_desc` as shown in figure 2.2. Each of the array's elements describes one structure read from the input file. The type `struct struct_desc` contains the name of the described structure type and an array of member descriptions. Each member description contained in a `struct member_desc` contains the name of the member, its storage size, offset, bit size and bit offset and a reference to its type, which is described by a `struct type_desc`.

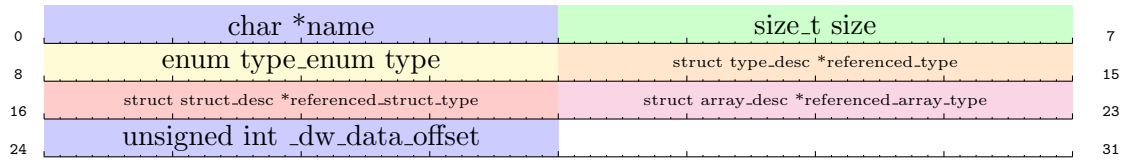
`struct struct_desc`:



`struct member_desc`:



`struct type_desc`:



`struct array_desc`:

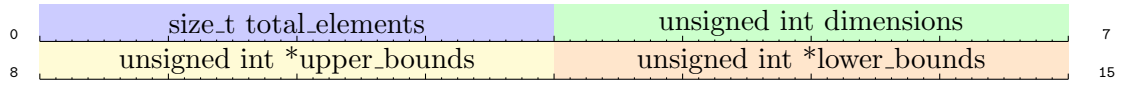


Figure 2.2: Internal data structures of *st* describing the types read from the input file.

This type description contains the name of the type, its storage size and a type category, as well as references to other types. The type category is important for us to be able to interpret what the fields in the type description mean or whether they are important at all. Possible categories include e.g. `MT_BASE_TYPE` or `MT_POINTER`. These categories are analogous to categories *DWARF* applies to various types. In that classification a base type would be a simple, non-compound type such as `int` or `float`. A pointer on the other hand is a more complicated type in that it not only has similar characteristics as an `int` but it also references another type which the pointer "points to". Similarly array and structure-typed members reference their respective array member type or structure type description.

2.1.3 Filter layer

The filter layer applies various transformations to the intermediate representation described in the previous section. Generally filters have complete access to all of the intermediate representation, so it is possible for a filter to delete, generate or arbitrarily alter structure descriptions. The only limitation is given by the expressiveness of the intermediate representation.

Two example filters have been written in the course of this work but there are of course many more possible applications for the filter mechanism. Those two examples will be presented here.

"`struct_name`" filter

One trivial example of a filter is the `struct_name` filter. This filter takes a name as parameter and simply iterates over the array of structure descriptions and drops all structure descriptions where the structure is not named as the parameter specifies. So this filter can be used to "pick" only a certain structure out of all structures read from an input file and apply further processing and output only to that certain structure. It is also possible to invert the meaning of this filter to e.g. filter out all structures named "`ANONYMOUS`".

"`explicit_padding`" filter

Another possible filter is the `explicit_padding` filter. A compiler generating the memory layout of a structure e.g. from *C* code will usually add padding[3] to align structure members to 4-byte boundaries, cache-line boundaries or similarly advantageous addresses. This alignment will leave "holes" in the memory layout of structures. These "holes" are called padding. When accessing such structures in memory, the padding is usually unreachable via the usual structure members. Only by pointer arithmetic, access to padding can be achieved.

Usually access to padding is unnecessary, but sometimes it can be useful. If a structure is placed in previously used memory the padding will often contain the data previously stored in that memory segment. This can be a problem in security critical applications which is why one might want to check for such occurrences of "leftover" data leaks.

Also, when code of certain languages is generated by our tool, a compiler might again try to add padding on top of that code. Since the algorithms by which padding and alignment are determined are very much dependent on the compiler, its configuration and the architecture and characteristics of the target system, the padding will be very hard to predict. To ensure an exact memory layout of the target structure the `explicit_padding` filter can be used as follows: When explicit padding is added to the output, a special compiler option, pragma or attribute (like for example `__attribute__((packed))` in *C*) can be used to prevent the compiler from adding any further padding or alignment at all. So since all padding will be explicitly specified in the structure definition and no further

[3] See also chapter 1.2.4 on page 15.

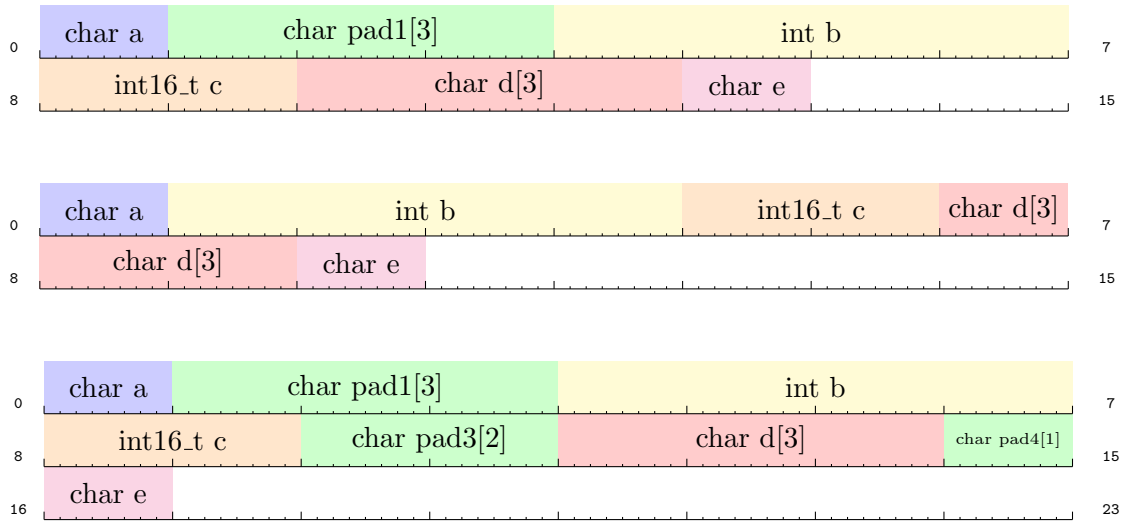


Figure 2.3: The same structure, compiled (from top to bottom) with no additional hints, without padding specified through `__attribute__((packed))` and with padding for alignment on 4-byte boundaries through `__attribute__((aligned(4)))`. Padding members as added by the `explicit_padding` filter are marked in green colour. The figure was created by *st* with the `explicit_padding` filter and the `tikz` output.

padding will be added by the compiler, the layout of the output will be guaranteed to match the layout of the input. The only drawback with this approach is the slight loss of readability the explicitly added padding members incur.

The implementation of `explicit_padding` works by linearly iterating over adjacent pairs of structure members i and $i+1$. When the offset of a member o_{i+1} is larger than expected from offset and size of its predecessor, that is, if $o_{i+1} > o_i + s_i$, then a padding member is added in between. The size of the padding member is determined by the difference given by this inequality $s_{p_{i+1}} = o_{i+1} - (o_i + s_i)$.

The data type of the added padding members will be array of s_{p_i} `char`s. An array of `char` is very useful for this purpose since its size in bytes can be easily adjusted and only one padding member is necessary for each pair. Also, when using the generated code to read out data contained in the padding space, an array is far easier to use than other possible data types[4].

```
$ ./st -o pseudo -f struct_name --name with_attr_packed ./test/obj/test-packed.o

struct with_attr_packed: 5 members {
        a: index 0, size 1, offset 0, bit: 8 @ 0, type char (s 1b, t MT_BASE_TYPE, 0, 0);
        b: index 1, size 4, offset 1, bit: 32 @ 0, type int (s 4b, t MT_BASE_TYPE, 0, 0);
        c: index 2, size 2, offset 5, bit: 16 @ 0, type int16_t (s 2b, t MT_TYPEDEF int16_t :=¶
              short int (s 2b, t MT_BASE_TYPE, 0, 0), 8100d78, 0);
        d: index 3, size 3, offset 7, bit: 24 @ 0, type ANONYMOUS (s 0b, t MT_ARRAY char[0 to ¶
             2], 8100e50, 0);
        e: index 4, size 1, offset 10, bit: 8 @ 0, type char (s 1b, t MT_BASE_TYPE, 0, 0);
}
```

Figure 2.4: Example invocation and output of the `pseudo` output module. The structure is also shown in figure 2.3. Line breaks that were included due to the limited page width and are not part of the original output are marked by the "¶" sign.

2.1.4 Output layer

Simple "pseudo" output

For easier debugging and development of the input layer a simple output module was necessary. This output module is called "pseudo" and simply emits a complete but short representation of the interface format described in section 2.1.2 which in the first versions bore a superficial resemblance to the *C* programming language, hence "'pseudo'-C".

This output module is also very handy for a getting a quick overview over the structures contained in an object file without having to read through the more verbose output of the other output modules.

VHDL

The *VHDL* output module creates a *VHDL* entity and corresponding architecture for each structure in its input. The functionality and usage has been adapted to the peculiarities of an implementation in hardware as well as the project it is to be used in. Therefore certain assumptions and choices were made which might not apply to different projects and situations.

The primary function of each of the *VHDL* entities is to calculate for a given base address and structure member the address at which that member can be accessed. For the purpose of easier discussion such an entity will be called "address generator". The address generator has a base address input and an address output as well as a one-bit input for each structure member as a selector. The signal name of each selector input is based on the member name of the respective structure member. For example for a structure member named `foo_bar` the corresponding selector input would be named `foo_bar_i`[5].

[4] One could e.g. imagine filling the space with several `int32_t`s until only a part smaller than 4 bytes remains which is then filled with an `int16_t` or `char`.

[5] It is assumed that the user will handle resulting conflicts and ambiguities manually. Though it would be possible to devise an automatism, the user would have to read the code and inform himself about the disambiguated names in any case, such that the automatism would not really simplify matters.

Type information about the data located at the calculated address is not used or included in the output. So if the user accesses a structure member that is a pointer, it is the responsibility of the user to write his code to handle the data correctly as a pointer, e.g. to read the data it points to. No "type flag" or similar hint is output by the address generator.

Yet some types are handled in a special way. Nested structures and unions are "flattened" into the surrounding structure by using the structure name of the surrounding structure as a prefix for all selector names of the nested structure or union member. A member that would be accessed as `nested.foo.bar` in *C* code would be selected through the `nested_foo_bar_i` signal.

An array as a structure member can be accessed via a special mechanism that allows access to its elements via their index. For this, an index input `array_index_i` and an output flag `bounds_violation_o` are specified. The `array_index_i` input takes the unsigned integer array index which is then multiplied by the element size of the array and added to the address of the zeroth element. Since embedded arrays usually have a fixed size, array bounds checking can be done. If a violation occurs, that is, if an array index beyond the last element is requested, `bounds_violation_o` is set. If no array is accessed, the value of `bounds_violation_o` is unspecified, if an array is accessed within its bounds, `bounds_violation_o` is unset.

Multidimensional arrays are handled by flattening the array into a one-dimensional array with the number of elements equal to the number of elements of the multi-dimensional array. That is, if the n-dimensional array has an index range of $[0, k_i]$ for each dimension $i = 1...n$ then the index range of the corresponding one-dimensional array will be $[0, K]$ where $K = (\prod_{i=1}^{n} k_i + 1) - 1$.

This leaves the responsibility of ensuring the proper index ordering and range checks for the array stored in memory on the shoulders of the programmer using the address generator module. This is unfortunate but unavoidable since implementing proper support for multidimensional arrays would have required either multiple index inputs thereby drastically increasing the signal count and thereby the resource usage of the generated address generator module. Or multiple indices could have been input in a serialized manner, which would have required additional registers for storage and an increase in delay from one clock cycle to n clock cycles. Due to the scarceness of multidimensional arrays and the high cost of implementation the simplifying flattening approach was deemed acceptable.

Similarly, the difficulties in handling arrays of structures and arrays of other compound data types were found too complex to implement and too rare to warrant the effort. If necessary, those can of course be manually supported: If e.g. the element `foo.a[2]` is a structure, its base address can be output by one appropriate address generator module for the structure type of `foo` and used as base address input for another address generator module for the structure type of the elements of `foo.a`.

Structure members that are bit-fields also need special handling. Since the addresses output by an address generator module are specified in bytes and structure members only have integer multiples of bytes as sizes, the usual addressing scheme by bytes is sufficient. Yet for bit-fields, a bit-wise addressing and size handling is needed. For

this, two different mechanisms are provided: A bit-mask to extract the appropriate bit-field member from the containing bytes is provided via `bit_mask_o`. Also, the size and offset in bits are output via `bit_size_o` and `bit_offset_o`. These can be used in combination or separately to extract the bit-field member according to the requirements of the application or the preferences of the programmer: Either the bits are extracted by generating the bit-wise `AND` with the `bit_mask_o` and left-shifted by `bit_offset_o`. Or `bit_size_o` and `bit_offset_o` can be used to get the data bit-by-bit from a shift-register in serialized form. In this case, both approaches were considered useful and easy to implement.

In all those considerations, it was assumed that synthesis tools are sufficiently capable in removing unused parts of the generated design like unused inputs or outputs or branches of conditional statements that are never taken. Therefore all accessible members are included in the output by default as well as inputs and outputs for nested array, struct and bit-field handling, even if the structure the output is generated for does not contain such members. If necessary, a suppression mechanism e.g. by specifying command line parameters to `st` to suppress this possibly unnecessary code may easily be added to future versions of `st`.

C

Originally, the "pseudo"-output described above was intended to generate working *C* code. But since the information contained in the *DWARF* data structures is more detailed than *C* code can express, "pseudo" rather focused on providing a short but complete representation of all details.

But of course the usefulness of outputting *C* code is still obvious. The *C* output therefore generates `struct` declarations containing an approximation of the original members. Currently the implementation is very simple and not as complete as it could be. Special members like bit-fields are unsupported, but could be easily added should the need arise.

TikZ

TikZ[15] is a popular package for the LaTeX typesetting language to generate drawings from a description language. Since TikZ is easy to use as well as feature-rich and capable of producing visually pleasant results it was also my favourite tool for drawings in scientific documents. Therefore it seemed logical to also use TikZ to generate graphical output from *st*.

The TikZ output consists of two main parts. An output module in *st* called `tikz` generates an intermediate representation of the structure in a special LaTeX environment called `struct`. This environment acts as a container for one or more `member` elements which each specify the characteristics of a single structure member.

A LaTeX package called `struct_type.sty` then uses the data given in the `struct` environment and `member` elements to typeset a visual representation of the structure. The reasons for the separation between the generation of the description and the type-

setting functionality are obvious: First this separation makes it possible to manually edit the description and for example shorten long member names or delete unimportant parts which would distract. That way it is even possible to manually generate the whole description, for example for purposes of documentation or design. Second and more importantly, typesetting and the creation and layout of the visual representation is far easier to accomplish in LaTeX.

An example of a description and the graphical representation of that description can be seen in figure 2.5. Each member is described by its position, length, colour and name which are rendered into a multi-lined arrangement. Typically each line represents 4 bytes, each subdivided into 8 bits, although this is configurable. Positions and lengths are given in bytes, bit offsets and non-integer lengths are specified by adding the appropriate decimals. A member, for example in figure 2.5 the member "d" marked in green, may also span multiple lines. "Line-breaking" in that case is automatic and the name label is repeated on each line. A complete description of all functionality of the `struct_type.sty` package will not be given, the interested reader may peruse the source code of the package. As further illustration, figure 2.6 shows an Internet Protocol version 4 network packet header.

```
\usepackage{tikz}
\usepackage{struct_type}
...
\begin{struct}[yscale=0.5, xscale=1.2, bytes ¶
    per line=3, bits per byte=8]
        \member{1}{1}{blue!30}{b};
        \member{0}{1}{yellow}{a}
        \member{6}{2}{red!30}{cr};
        \member{8.25}{1.375}{blue!30}{bit};
        \member{10.875}{6.250}{green!30}{d};
\end{struct}
```

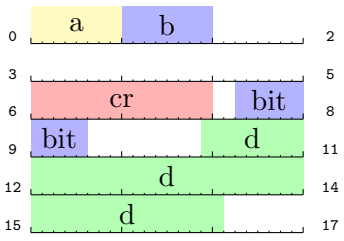


Figure 2.5: Structure description (left) of an arbitrary structure and the corresponding graphical representation (right) generated by the `struct_type.sty` package. Line breaks in the structure description which are not present in the original source code are indicated by the "¶" character.

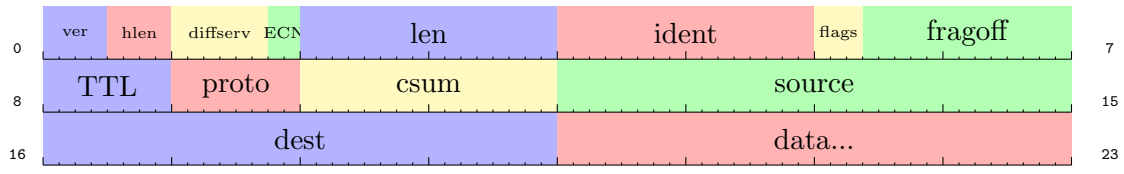


Figure 2.6: Graphical representation of an IPv4 header. The last member "data..." is of variable length indicated by the ellipsis.

2.2 Important Details

2.2.1 Endianness

When transferring data from a host system's memory to e.g. an FPGA over PCI DMA accesses or when using networks to transfer data between different systems, different endianness[6] can become an issue. The endianness of a system specifies whether the low-order byte of a two-byte or longer integer is found at the lowest or highest address within the integer. There are two basic arrangements called "little-endian" and "big-endian". In a little-endian system, the byte with the lowest (the lowest address is usually also considered the "first") address in a multi-byte integer is the least significant byte. Therefore little-endian systems are also called "LSB first" systems. Correspondingly a big-endian system is called "MSB first" and places the most significant byte at the lowest address in memory.

Usually, when a single system is programmed, most of the issues of endianness are hidden from the programmer. Yet it is possible to write code that behaves differently on a big-endian and a little-endian system, e.g. by converting from an `int*` to a `char[4]` and then accessing the elements[7] of the array. With the same integer placed at the location the `int*` points to on both systems, the contents of the elements of the `char` array will be reversed. When e.g. a data structure is serialized and transmitted over a network, such a conversion to a `char` array is quite the same process as the network hardware and software will perform. Therefore endianness conversions and endianness conventions are very important in this area.

In our primary use-case of reading data structures over PCI DMA this endianness problem also occurs. The FPGA initiating the DMA transfer might have a different byte-ordering than the system it reads its data from. Since the *VHDL* output of *st* can not know which byte-ordering the resulting *VHDL* code will run on and since only addresses are calculated, no automatic data conversion will take place. Therefore it is necessary for the user of *st* to ensure the correct endianness-conversion between the host and the FPGA systems.

The idea might arise that this conversion between the host and FPGA system's byte ordering should be the responsibility of the PCI bridge core used within the FPGA. But this is an impossible task for any given transfer of data. The PCI bridge can not know which sequences of bytes belong to larger base types and thus can not decide which bytes to swap and which ones to leave in their original ordering. So if e.g. a transfer of 4 bytes occurs, the PCI bridge would not distinguish between an `int32_t` and a `char[4]`. Yet for the `int32_t` an endianness conversion might be necessary, but for a `char[4]` it must never be done.

[6] Endianness is used synonymous with "byte ordering" within this document. In other publications "bit ordering" is sometimes also subsumed under the term "endianness".

[7] For this example we assume a 32-bit system where an `int` is 4 bytes long. On systems with a different word length only the number of elements in the `char` array changes.

2.2.2 Bit-order and bit-field layout

Similarly the ordering of bits within a byte might be different when transferring data, either from the host system's RAM to the FPGA or within the FPGA system. The first case of a bit reversion occurring between RAM and FPGA we will consider this the responsibility of the PCI bridge since such a conversion will either always or never be necessary. Therefore it is possible —other than with the endianness conversion mentioned before— for this bit-order conversion to take place within the PCI bridge.

The second case where bit ordering is important is when using and transferring data within the different parts of the FPGA. When using the bit-mask generated by *st* for the support of bit fields within structures, *st* inserts a `std_logic_vector` literal to define this bit-mask. If the endianness or bit-order of this bit-mask are different from the endianness or bit-order of the data it is applied to, wrong bits will be extracted from the data in question.

This has not been tested and various designs might produce unforeseeable situations in which this behaviour will of course be problematic. It may be necessary for the user of *st* to take steps to alleviate this issue, e.g. by reversing the bit-order of the bit-mask in question.

2.2.3 C++ support and support for other languages

While the data model of *C* was used as a reference target to implement *st*, other languages can of course also be supported, if their compilers are able to generate *ELF* object files with *DWARF* debugging symbols. Since its similarity and common feature set, *C++* is an obvious example of such a language.

Preliminary tests and some simple changes have been made to support the subset of *C++* that is equivalent to *C* as well as inheritance and extension. Since *C++* classes, declared with the `class` keyword, are within the scope of *C++* just structures[8], *st* has been extended to just handle the *DWARF* tag `DW_TAG_class_type` in exactly the same way as `DW_TAG_structure_type`. This allows *st* to read *C++* classes without inheritance, that is, without parent classes.

To support inheritance, a *C* language extension mentioned earlier comes into play. Anonymous structures have already been mentioned on page 18 in chapter 2. Anonymous structures allow the embedding of structures without a member name within another structure. This is exactly how *C++* embeds the parent class into a child, by adding an anonymous structure as the first member of the child class. An example of this is shown in figure 2.7.

While "normal" data members are handled as expected, static data members are not handled correctly by our code. Since static data members are shared across all instances of a class, they are not stored along with the other normal data members. Also, their storage location can not be calculated as an offset from the instance pointer. *st* does not take this special handling of static members into account and always outputs an offset

[8]The only difference between the *C++* `struct` and `class` keywords are different visibility rules for members and inheritance.

```
class parent {
public:  int b;
private: char c;
         int d;
};

class child : parent {
public:  int extend_1;
private: double extend_2;
};
```

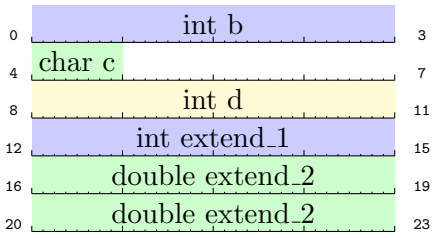


Figure 2.7: Example of *C++* parent and child classes (left) and the representation of the `child` class in memory (right). Note how the members of the `parent` class precede the members of the `child` class. The graphical representation to the right was generated using *st* and the *TikZ* output module.

of 0 from the instance pointer for every static member. While this is no problem for the *VHDL* output module, this is a problem for e.g. *TikZ* output.

For similar reasons, class members that are methods are currently handled incorrectly. No tests have been done regarding more complicated features like e.g. templates or class variables since *C++*-support lies outside of the scope of this work. Of course support for those features and fixes for the problems described above may be the goal of future work on *st*.

Other languages may also be already supported or supported with only small changes or enhancements to *st*. As an example, a trivial *Pascal* record[16] was compiled and *st* was tested on it. The *FreePascal* compiler used emits some previously unimplemented *DWARF* attribute and tag types. With about 20 lines of code and in half an hour of time those were implemented. The example code shown in 2.8 shows the simple *FreePascal* support resulting from this. Since no further tests beyond that very simple case were done, no full support for *Pascal* is claimed. But the relative ease of adaption of *st* to another source language shows the high potential of the *st* code base for the support of a large number of languages.

```
program ptest1;

Type test1 = Record
        a : real;
        b : array [0..10] of integer;
        c : integer;
end;

begin
end.
```

```
struct TEST1: 3 members {
        A: index 0, size 8, offset 0, bit: 64 ¶
             @ 0, type REAL (s 8b, t ¶
             MT_TYPEDEF REAL := REAL (s 8b, t ¶
             MT_BASE_TYPE, 0, 0), 8101730, 0);
        B: index 1, size 22, offset 8, bit: ¶
             176 @ 0, type ANONYMOUS (s 22b, t¶
              MT_ARRAY SMALLINT[0 to 10], ¶
             81017f0, 0);
        C: index 2, size 2, offset 30, bit: 16¶
              @ 0, type SMALLINT (s 2b, t ¶
             MT_TYPEDEF SMALLINT := SMALLINT (¶
             s 2b, t MT_BASE_TYPE, 0, 0), ¶
             8101910, 0);
}
```

Figure 2.8: Simple *Pascal* program defining a record type (left) and the corresponding output of *st* using the "pseudo" output module (right). Line breaks that were not part of the original output are marked by the "¶" symbol.

3 Evaluation

3.1 Tests

To manually test the implementation of *st* during development, a number of synthetic test cases were created. The test cases are implemented as *C* files that are compiled into object files with embedded debug information. Each test case contains various structures that test certain features to be implemented or aspects that are expected to be interesting or problematic:

- different combinations of alignment and packing pragmas[1]
- differently sized members and members with odd sizes
- arrays of base and structure types and multidimensional arrays
- bit-fields
- *C99* types like `_Bool` or `_Complex` as members
- structures and unions as members
- type aliases or typedefs
- various combinations of the above

An object file from a *Linux* kernel was also included in the test suite to make sure no details necessary to support this intended field of use were missing.

Manual tests were continuously performed during development by using the pre-existing tool *dwarfdump*, *dwarwdump*[2] and the tool *st* under development. The output of each of those tools applied to the test objects was manually interpreted and compared with respect to the expectations and specifications described in this document.

It would of course be possible to implement automated tests using the test cases described above. This has not yet been done since during the development of *st* the expected output was very much in flux. Therefore implementing tests against this moving target would have been too involved.

[1] In *C* this is usually –as in our case– accomplished by the use of `__attribute__((aligned(...)))` and `__attribute__((packed))`.

[2] *dwarwdump* is my reimplementation of parts of *dwarfdump* to familiarize myself with the usage of *libdwarf* and to explore the *DWARF* format. Both tools output a textual representation of the *DWARF* data contained in an object file.

3.2 Real Applications

The *st* tool has been successfully used by Michael Gernoth in his Ph.D. thesis to create a working interface from a *Linux* kernel.

Currently work has been completed on a prototype implementation by Michael Gernoth using *st* in a build process. In this process, a hardware implementation is automatically generated which, for a given running Linux kernel, iterates over the kernel's process list in `struct task_struct` and detects and terminates prohibited processes in the running system.

Preliminary reports from Michael Gernoth indicate, that *st* is suitable for the task it is intended to perform, easy to use and sufficiently flexible to also be used for applications beyond its intended uses. Especially the ability to generate graphical representations via the *TikZ* module has been received well by prospective and current users.

Of course this work also contains several examples of illustrations and code generated using the *st* tool.

3.3 Possible further Usage Scenarios

Besides the motivation given in sections 1.1 and 1.1.1, *st* has number of possible uses. Time will tell which of these possible uses are applicable in reality.

Generating graphical representations of the memory layout of data structures is useful in many situations. Technical documentation is easier to understand when textual descriptions are supported by meaningful graphics. Similarly teaching and presentations may benefit. *st* can be used to generate such graphics automatically via the *TikZ* output module. While automatic generation makes one-time use easier, it is also possible to have *st* automatically regenerate illustrations to make documentation fit the code that is in use e.g. by using *st* in a "Makefile".

Especially in section 1.1.1, the need for adaptation of a hardware interface to changing operating system interfaces was discussed. But even when only using fixed operating system interfaces and fixed, non-programmable hardware, *st* may still be useful. By freeing the developer from the need to write parts of the interface code that can be automatically generated, development time and cost may be saved. Also, automatic generation of interfaces could avoid certain instances of human error in writing those interfaces. An example where this could certainly be useful is the development of hardware devices and their respective drivers.

But also interfaces between different pieces of software can be partially generated using *st*. Using the *C* output of *st*, with use of the `explicit_padding` filter, it is possible to exactly reproduce the memory layout of a given structure. Since *st* works from the compiled form of the program —at least as long as debug symbols are embedded—, the source code of the program to interface to is not even necessary. This is a situation frequently found when interfacing to proprietary software or software where the source code has been lost or is unavailable for some other reason. Security research and debugging could similarly benefit from such interfaces. As an example the possibility to access

padding inside structures has been mentioned.

4 Conclusion

4.1 Related Work

Regarding the automated generation of HDL source from object files or debugging information as described in this work, investigation seems to indicate that our approach is quite unique. While the problem of interfacing a microprocessor program with a hardware core certainly has been recognized, all published related work seems to concentrate on translating *C* source code to HDL or to generate both *C* and HDL from a common language. Also, when generating an interface mostly timing concerns have been explored while the transfer of complex data structures seems to have been ignored.

[17] describe a tool *C2H* to generate HDL from *C* sources as a means to generate accelerator cores for the function implemented by the *C* source. While they describe their data interface between the HDL core and the binary executed on the main processor as using DMA memory accesses, they do not elaborate on the problem of matching the data layout[1]

Similarly [18, 19] describe their *ROCCC* tool which also generates HDL from *C* sources. Here the primary focus seems to be on the automated extraction of appropriate timing when communicating from the compiled *C* program to the generated core.

Significant work has been done on the co-generation of hardware and software from a common source code. Examples of this approach are *SystemADA*[20], *Lava*[21] and *SystemC*[22] among many others. While those approaches are without question very useful for designing new systems of hardware and software, they are quite a bit less useful when faced with existing source code or even object file blobs.

Regarding the automated generation of graphical representations from debugging information or source code, there are of course pre-existing tools like *DDD*[23] with very similar capabilities to *st*. The further uses of *st* like helping in the automated extraction of padding data for security analyses like [24, 25] *st* could prove invaluable, especially since it facilitates the automated generation of *C* as well as *VHDL* code. In [25] an information leak via the padding of *Ethernet* frames was described. Such information leaks could be automatically detected and analysed through hardware and software tools generated with the help of *st*. Since this was not the primary focus of *st*, no further work on this subject has been done by us. However, a deeper investigation of this field of use is certainly merited.

[1] In the related field of dereferencing pointers [17] point out that of course different addressing modi —various virtual and physical address spaces— do pose a significant problem. This is of course an orthogonal problem to interpreting the data structures those pointers point at or are contained in.

4.2 Summary

Interfacing hardware and software components in various fields of use is an important problem. This work described the development and implementation of the tool *st* While primarily written to read *Linux* kernel data structures through DMA transfers from a *VHDL* core, *st* was shown to be very flexible and thus useful far beyond its original purpose.

The *st* tool can automatically extract the memory layout of structure types from object files containing *DWARF* debugging information. By this approach *st* avoids the problems of tools operating on *C* source code like the differences in the code generation of compilers or the possibility of supporting languages besides *C*. Flexible input, output and filter layers allow for a variety of transformations to the structures as well as various output formats like *S* source code, *TikZ* graphics and of course *VHDL*.

The unique capability of *st* of generating data access interfaces in *VHDL* from object files has been proven for its intended purpose of interfacing hardware to *Linux* kernel structures. Yet many ideas of further uses for this special capability as well as for other functions of *st* have been in this work. Reverse engineering, security research or graphics generation for teaching are only part of the imaginable further possibilities that may form the basis for further work on *st*.

Bibliography

[1] Intel releases x86-FPGA hybrid - app store to follow?, The Daily Circuit, available via `http://www.dailycircuitry.com/2010/11/intel-releases-x86-fpga-hybrid-app.html`, 2010.

[2] F. Kessel and R. Bartholomä, *Entwurf von digitalen Schaltungen und Systemen mit HDLs und FPGAs*, 2 ed. (Oldenburg Verlag München, 2009).

[3] VHDL, Wikipedia — The Free Encyclopedia, available via `http://en.wikipedia.org/wiki/VHDL`, 2011.

[4] VHSIC, Wikipedia — The Free Encyclopedia, available via `http://en.wikipedia.org/wiki/VHSIC`, 2010.

[5] PCI-SIG home, Website `http://www.pcisig.com/home`.

[6] Executable and Linkable Format, Wikipedia — The Free Encyclopedia, available via `http://en.wikipedia.org/wiki/Executable_and_Linkable_Format`, 2011.

[7] M. Eager, Introduction to the DWARF Debugging Format.

[8] D. Anderson, *A Consumer Library Interface to DWARF*, 2007, rev 1.66.

[9] DWARF Debugging Information Format, Version 4, 2010.

[10] B. Kernighan and D. Ritchie, *The C Programming Language*, 2nd ed. (Prentice Hall International, 1988).

[11] D. M. Ritchie, History of programming languages—ii, chap. The development of the C programming language, pp. 671–698, ACM, New York, NY, USA, 1996.

[12] ISO/IEC 9899:1999 Programming Languages — C, 1999.

[13] B. O'Sullivan, J. Goerzen, and D. Stewart, *Real World Haskell*, 1 ed. (O'Reilly Media, 2008).

[14] Anonymous Structures — MSDN library, available via `http://msdn.microsoft.com/en-us/library/z2cx9y4f.aspx`.

[15] T. Tantau, *TikZ & PGF Manual for Version 2.10*, Institut für Theoretische Informatik, Universität zu Lübeck, 2010.

[16] Freepascal wiki — record, Website `http://wiki.freepascal.org/Record`.

[17] D. Lau, O. Pritchard, and P. Molson, Automated generation of hardware accelerators with direct memory access from ansi/iso standard c functions, in *Field-Programmable Custom Computing Machines, 2006. FCCM '06. 14th Annual IEEE Symposium on*, pp. 45 – 56, 2006.

[18] Z. Guo, B. Buyukkurt, W. Najjar, and K. Vissers, Optimized generation of datapath from C codes for FPGAs, in *Design, Automation and Test in Europe, 2005. Proceedings*, pp. 112 – 117 Vol. 1, 2005.

[19] Z. Guo, A. Mitra, and W. Najjar, Automation of IP core interface generation for reconfigurable computing, in *Field Programmable Logic and Applications, 2006. FPL '06. International Conference on*, pp. 1 – 6, 2006.

[20] N. Mahani, P. Mokri, M. Sedghi, and Z. Navabi, Ada Lett. **29**, 15 (2009).

[21] S. Singh, The Lava Hardware Description Language, Website `http://www.raintown.org/lava/`.

[22] Open SystemC Initiative, Website `http://www.systemc.org`.

[23] DDD — Data Display Debugger, Website `http://www.gnu.org/software/ddd/`.

[24] J. Heusser and P. Malacaria, Quantifying information leak vulnerabilities, available via `http://arxiv.org/abs/1007.0918`, 2010, 1007.0918.

[25] O. Arkin and J. Anderson, EtherLeak: Ethernet frame padding information leakage, available via `http://leetupload.com/database/Misc/Papers/atstake_etherleak_report.pdf`, 2003.

Nomenclature

ASIC	Application Specific Integrated Circuit
DIE	Debug Information Entry
DMA	Direct Memory Access. DMA enables devices to access the main memory of a computer system directly without the need for processor and operating system interaction.
HDL	Hardware Description Language
ISA	Instruction Set Architecture
LSB	Least Significant Byte
LUT	LookUp table
Makefile	"Makefile" is the colloquial term and also the common filename for an input file to the "*make*" tool. *make* is a common build system for software projects. This document and the accompanying source code are also built using the *make* tool.
MSB	Most Significant Byte
PCI	Peripheral Component Interconnect. PCI is a common bus system in desktop and other systems.
TikZ	TikZ ist kein Zeichenprogram, german for "Tikz is not a drawing program"
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuits

License

GFDL licensing statement

CC-BY-SA 3.0 licensing statement

Licensing of code examples

[2] `http://creativecommons.org/licenses/by-sa/3.0/`
[3] `http://www.gnu.org/licenses/fdl.html`