Annotated Data Type Declarations for Bus Interface Synthesis

Gordon Cichon
Institute for Integrated Circuits, TU-München, D-80290 München
gordon@cichon.de

Abstract

This paper investigates the applications of data type declarations describing memory mapped interfaces of hardware components. The rise of abstract bus interfaces like in the VCI standard for System on Chip design enables the logical separation of a protocol adaptor and a functional adaptor part in the bus interface of hardware components. In this approach, the functional part of a component's bus interface together with a symbolical testbench can be synthesized at multiple abstraction levels from an annotated declaration of its memory map: a register transfer model, a behavioral model, and documentation.

A companion paper will treat implementation aspects within application specific hardware functionality.

1 Introduction

Cost reduction requirements force consumer electronic manufacturers to integrate as much functionality as possible into a single piece of silicon in order to save expenses for interconnection wires and printed circuit boards (PCBs). The integration densities allow System on Chip (SoC) design to implement complete electronic systems on single ICs.

Since product cycles are fast, and custom IC development is expensive, SoC development cannot be started from scratch every time. For this reason, repeating parts of these systems are encapsulated into modules which can be reused in other designs. For traditional system development on PCBs, there is a market for modules implemented as premanufactured components to be used for building complex systems. SoC design uses a similar reuse mechanism with the difference that components are not premanufactured physically. Instead, "virtual" components are described at higher abstraction levels. These virtual components are reused or sold as intellectual property (IP) respectively.

To enable teams to cooperate in developing a system, the interfaces between different modules need to be agreed on. In traditional PCB designs, the interfaces between different components were either completely fixed for premanufactored components or completely custom within a development team for a specific system. With SoC design, the situation is more differentiated. Traditional component interface standards like PCI [23] pin down all details of the physical implementation while the upcoming standard on Virtual Component Interfaces (VCI) [1] allows a broad range of implementation tradeoffs.

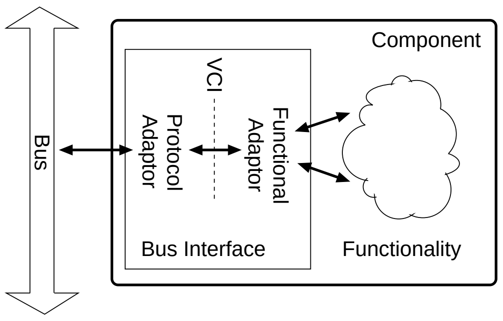

Figure 1: Virtual Component Architecture

Figure 1 shows a component with one external interface. There is no restriction on the number of interfaces a component may have. The hardware dedicated to connecting the custom functionality of a component to the interface serves two different purposes: The functional adaptation maps the functional aspects of the component to transactions on the interface like read and write transactions. The protocol adaptation maps the interface transactions to their physical implementation like assertion of signals in correct timing.

This paper focuses on random access bus interfaces. Random access means the presence of a notion of an address assigned to each data item, and the transactions on the interface are mainly read and write transactions. This is typically the case with system buses of processors, like PCI [23], Rambus [10], and Amba [3]. On the other hand, data stream oriented protocols like RS-232, Ethernet, or ATM do not fall into this category.

Additionally, a “bus” is defined at other places as a set of electrical lines capable of transferring a “word” of data in parallel, like in “address bus” and “data bus” of some processors. Throughout this paper, “bus” is defined to be a random access interface (like the “system bus” of a processor) without implication on the physical implementation. All information may be transferred serially over one wire as in I2C, or in a stream of data packets like in the Rambus [10] protocol.

2 Related Work

Related work can be arranged according to three categories: Strictly protocol synthesis, interface synthesis for data stream oriented interfaces, and synthesis of complete communication channels.

The Synopsys Protocol Compiler (formerly known as Dali [27]) belongs to the first category. It synthesizes a finite state machine from a grammar based description consisting of a set of regular expressions over the course of signals and corresponding actions. [19] presents a protocol translator between different bus protocols based on the translation of read and write transactions. [5] uses a declarative description language for specifying the transaction protocol on a bus. [22] uses regular expressions as seen in Protocol Compiler to define a protocol for bus transactions. These approaches enable encapsulation of bus protocols into another as demonstrated with CAN and I2C in [20] and [21].

In the second category, there is the FORM system [12]. It facilitates functional adaptation for data stream oriented protocols like ATM.

The third category is the generation of custom communication topologies including their interface implementation as seen in [24], [11], [28], and [17]. The Chinook system [6] handles functional adaptation by synthesizing address decoders for custom components [7]. The VCC system by Cadence [25] is designed for making design tradeoffs. Memory map locations and other resources are allocated for specific functions of either hardware or software within a graphical user interface. This information can also be used to synthesize VCI compliant functional adaptors for hardware components. In these approaches, the interfaces are not synthesized for the purpose of inter-operating with other independently developed components. This restricts the scenario in which such an component can be used.

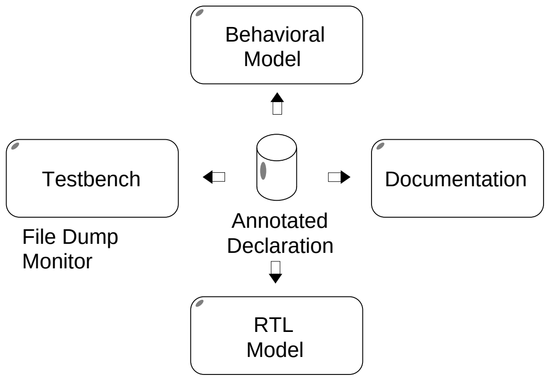

Figure 2: Interface Compilation

3 Functional Adaptation

The Virtual Chip Interface (VCI) [2] lifts the interface between components from the physical level. Part of the VCI is the On Chip Bus (OCB) [1] standard which defines an abstract bus interface with read and write transactions. At this interface, the protocol part and the functional part of the bus interface can be separated (see Fig. 1).

Although the external interfaces seem trivial when looking at high level block charts, they tend to make up a significant part of the whole design complexity. Even a mid-size design like Infineon’s Cerberus_FPI [16] uses less then 20% of its code for modeling the component’s functionality. More then 40% is used by the testbench, and about another 40% is used up for software driver code. This shows that automation of the design tasks which are not related to modeling payload functionality bears a significant saving potential in the hardware development process.

The paper focuses further on the functional adaptor part of the bus interface. While the functional adaptor is dependent on the actual functionality of the component, and has to be synthesized for each component individually, the protocol adaptor depends mainly on the physical implementation of the bus. So the protocol adaptor can be developed once for each individual physical bus interface (like PCI [23], Amba [3], etc.), or it can be synthesized by the tools mentioned in Section 2 (e.g., Protocol Compiler [27])

Adopting the VCI standard is not the only possible way to separate these two parts of the bus interface. A tool synthesizing the functional adaptor could also be based on another suitable abstract bus specification, or it could cover the protocol aspects, and synthesize a monolithic bus interface.

The center of the approach to synthesize the functional adaptor part of a bus interface is an annotated declaration of its memory map. Details about this declaration language will be given in the next section. From this declaration, the functional adaptor can be synthesized consistently at different abstraction levels (see Fig. 2). The design model propagates changes in the component's memory map immediately to the documentation, the behavioral model (e. g., a C model), and the RT model (e. g., a VHDL or Verilog model). This lightens the burden of keeping the models and the documentation consistent with each other throughout the development process. On the one hand, it becomes very easy to change the memory map, and the process is much less error prone. On the other hand, the design cycle gets shortened. Therefore, making design tradeoffs becomes practical.

4 Memory Map Declaration

The name "memory map" originates from the fact that the component appears to an external processor like a piece of memory. The functionality exposed at this memory window can be structured. The description of this structure can be modeled on declarations of structured data types in programming languages like Pascal [14]. This section relates data types to annotated map declarations.

The abstract bus interface defines read and write transactions. In advanced bus protocols, there may be additional transaction types like configuration and arbitration cycles which are not of interest for functional adaptation.

The transactions operate on a constant quantity of bits which will be called "words" throughout the paper. The component exposes a certain number of locations through its interface containing a word each, and which can be individually addressed by bus transactions. The size of the word may or may not correspond to the internal processing width or to the definition of words in other parts of the system. Traditionally, the word size corresponds to the bit width of the data part of the bus, e. g., a typical word size is 32 bits. However, recent designs tend to operate on entities of cache lines, e. g., 4 times 32 bits in AGP [9], or 16 times 16 bits in Rambus [10]. Within a word, specific sub-quantities of bits (typically bytes) may be enabled or disabled, i. e., taken into account for the transaction or be ignored.

The treatment of words for functional adaptation does not imply the physical transfer of an individual address for each word. The address may

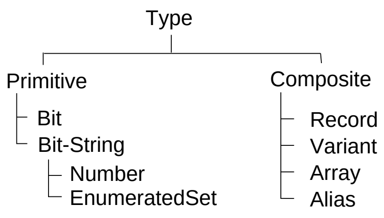

Figure 3: Type System

also be assigned implicitly during a contiguous "burst" transfer of words, or recovered by other means which are under the responsibility of the protocol adaptor (e. g., side-band addressing [9]).

This paper presents the memory map declaration in textual form to point out similarities to data types in structured programming languages. Map declarations may also be entered graphically using a schematic entry with graphical user interface ([25]).

The concept of an annotated map declaration consists of two parts: First, here is a skeleton of a data type declaration, as described in Section 4.1. In a second step, this skeleton is furnished with annotations which take the special properties of the active nature of memory mapped components into consideration, as described in Section 4.2. However, some properties of data type skeleton, which are usually defined statically, will be defined using annotations here. Therefore, the notion of annotations is already used before it is exactly defined. An example how this works in practice can be found in Figure 5.

4.1 Type System

Figure 3 shows an overview of the type system of the declaration language. It is very similar to what can be found in Pascal [14], C [13], or VHDL [4].

There are two kinds of item types. The first are composite in that their declarations are composed recursively from items other types. The others are primitive in that they can be described independently. Primitive constructs is where the nested type declaration bottoms out.

4.1.1 Primitive Types

Primitive types can either be single bits or bit strings. Bit strings can either be interpreted as numbers or as enumerated sets. The bit string representing a bus transaction unit has special significance and is referred to by the name "word". The declaration language allows different number representations of integer, fixed, and floating point formats with customizable parameters. Bit strings do not necessarily need to be arranged contiguously in the memory map, they can be concatenated from chunks which are spread to different locations. This is an especially useful feature as the chunks can be selected differently based on variant discriminators.

4.1.2 Composite Types

Record The major building block for bus interfaces is the record type. It provides the way of grouping together subordinate items and assigning a symbolical name to each contained item.

Records work very similar to records in VHDL or structs in C. The record itself as well as the contained items can be used for constructs either spanning individual bits within a word or spanning a group of several words.

Variant While a Record groups a set of items it contains, a variant enables to switch between different interpretations of a map area.

Variants can be annotated to determine their current content type by an action (for details about annotations, see Section 4.2). This action is called the variant discriminator.

The discriminator may use information from the current state of the interface which may be recovered conveniently from a read back memory (see Sect. 6.1). It is also possible to use information that is available only at run-time, or that is not visible from the interface.

Variants are particularly useful for bridging the gap between pure data stream oriented and pure map oriented interface declarations. With Variants, data packets can be embedded into the memory map by accessing them at a specific address range, and distinguishing between different data packet types by the discriminator.

Array Like a record, an array contains a set of other items. Unlike a record, all elements of an array have to be of the same type, and the elements are accessed by a number instead of a symbolic name.

Another characteristic of the array is the number of content elements. In order to facilitate the bridging between data stream and map oriented interfaces as introduced in the section about Variants, the number of elements may be determined by an annotation action. (details about annotated actions will be given in Sect. 4.2)

A nontrivial action for determining array size and variant discrimination seriously affects the complexity of the address decoder. If items following a variable sized array of variant do not start at a fixed address, the address decoder is required to perform arithmetic operations like additions and subtractions.

Alias Alias types are named references to types declared elsewhere. They correspond to the “typedef” construct in C [13] and can be used to avoid repetitions in interface declarations.

4.2 Map Annotations

The declaration of a memory map of a component differs from a data type on the following points:

First, the declaration of a component pins down exactly the way into which the items are arranged in the map. While a software compiler is granted full flexibility to take its own choices about memory layout in order to maximize execution performance, the user needs to be able to take full control over the map arrangement in component declarations. Therefore, the map declaration can contain annotations to influence map layout.

Second, the items in a data types are memories. A memory can be written to, and it can be read from. At memory mapped interfaces, the situation is more complex. Writing to a location may trigger an action inside the component, and reading from a location may return a computed or otherwise determined value. For this reason, the map declaration is annotated with actions that get executed in order to trigger actions inside the component when an annotated item is written to, or to compute the returned data when an annotated item is read.

Though the condition under which they are triggered is different, these actions are similar to that given in a Protocol Compiler specification. Currently, the actions have to be specified both for the behavioral and the RTL models independently. It is subject for further research to figure out how these two specifications can be unified.

Whereas the action trigger approach of Protocol Compiler corresponds to that of the YACC compiler generation system [15], the approach presented here corresponds to the software design pattern “access method” [8] which can be found in object oriented programming languages.

5 Example: Encryption Engine

To show how the interface declaration works, a typical example for network processing is presented: A 56-bit DES stream encryption engine.

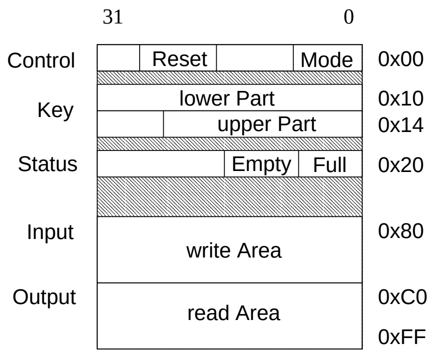

Figure 4: Encryption Engine: Interface

(see Fig. 4) For an introduction to DES encryption, see [26]. The interface is very simple so it can be handled within the scope of this paper.

Suppose, the encryption engine is used inside a cell phone for example. It is connected to a general purpose processor by a 32-bit system bus. The engine contains a reset bit which can be written to 1 by the processor in order to reset it. Let this bit be placed at position 16 at memory address 0 for some external constraint. A mode bit can switch the engine from encryption to decryption mode. Furthermore, there is an 56 bit encryption key which does not fit into a single memory word. This location can be written and read back. At a status register, an empty and a full bit provide feedback about the processing status of the engine. These bits are read only.

To use this engine, the processor has to initialize the key, and write the data to be encrypted to the input data location. The encrypted data can then be read back at the output location. The addresses of the read and write addresses to the input and output locations are not utilized by the engine, they are provided for convenient access for the processor only.

Figure 5 shows a simplified version of the interface declaration for the encryption engine. The data type is annotated with details regarding actual interface implementation.

6 Interface Compilation

6.1 RTL Model

From the interface declaration, several system components can be synthesized consistently by a set of tools: The first is an address decoder for the memory map of the bus interface (see Fig. 6). If it is required, the tool can also generate a read back memory to enable reading back specific parts of

```
INTERFACE des_engine IS length=0x100
BEGIN
  Control : RECORD access=w BEGIN
    mode : ENUMSET (encrypt, decrypt);
    reset : BIT pos=16
                action={reset <= 1};
  END.
  Key : RECORD access=rw
               addr=0x10 length=56;
  Status : RECORD access=r BEGIN
    full : BIT action={engine.full};
    empty : BIT action={engine.empty};
  END.
  Input : ARRAY ( 1 TO 0x40 ) OF WORD
    addr=0x80 length=0x40 access=w
    action=Write_Data;
  Output : ARRAY ( 1 TO 0x40 ) OF WORD
    addr=0xC0 length=0x40 access=r
    action=Read_Result;
END.
```

Figure 5: Encryption Engine: Functional Interface Declaration (simplified)

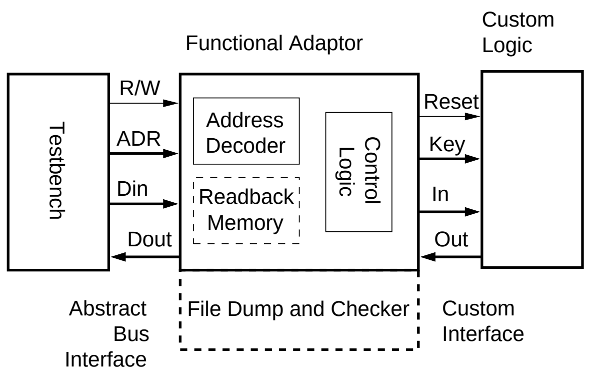

Figure 6: Bus Interface: RTL Model

```
> reset := 1;
> Key := 0x08150815;
> Input[0] := 'GGGG';
> Input[1] := 'cccc';
> Output[0]
0x12345678
> Output[1]
0x56789abc
```

Figure 7: Encryption Engine: Sample Debugging Session

```
RW Addr Data
 1  00  00010000
 1  10  08150815
 1  80  47474747
 1  81  11111111
 0  C0  12345678
 0  C1  56789ABC
```

Figure 8: Encryption Engine: Generated Stimuli

```
RW Addr Data
 1  00  00010000 # Write Control:
                   mode=encrypt, reset=1
 1  10  08150815 # Write Key: 25495573
 1  80  47474747 # Write Input[0]: ‘GGGG’
 1  81  63636363 # Write Input[1]: ‘cccc’
 0  C0  12345678 # Read Output[0]: 305419896
 0  C1  56789ABC # Read Output[0]: 1450744508
```

Figure 9: Encryption Engine: Output of File Dump Module

the memory map. This is useful for state saving at power down for example.

Each item of the declaration can be annotated as readable and/or writable, and a suitable VHDL or Verilog action is be executed if an access occurs. The language allows for customizing the signal port part of the synthesized entity in order to accommodate the requirements to interface the custom functionality of the component.

6.2 Behavioral Model

For use in system simulation, the tools can also generate a VSIA [18] compatible C model of the address decoder consistently. For the synthesis of an address decoder for the C model, suitable actions have to be provided for the C programming language using annotations.

6.3 Documentation

The tool is also able to generate a human readable form of the memory map by emitting TEX-code or a Frame-Maker document. This way, the documentation is inherently consistent with the hardware.

6.4 Testbench

The interface declaration allows for the generation of a testbench for either the RTL or the behavioral model. In Figure 6, two protocol adaptors may be inserted between the testbench and the functional adaptor in order to model the physical transmission channel correctly. This makes simulation more realistic and slower.

With this testbench, reading and writing to the device can be performed using symbolical names (see Fig. 7). This makes the test process less error prone, and the test stimuli become more portable if the map layout or symbolic value change.

Figure 8 shows the stimuli necessary to test the component. The meaning of the binary numbers is not obvious when looking at the stimuli. With the symbolical testbench, the developer can directly access the locations of her interest without having to look up its binary coding first. Debugging becomes much easier because potential problem areas can be narrowed down significantly faster when working on symbolical data. Additionally, the address layout of the interface can be changed without the need to recode all test stimuli manually.

When synthesizing C or VHDL code the tool is able to generate file dump code that protocol the actions at the bus interface with symbolical values (see Fig. 9). Since Verilog lacks basic string processing capabilities to produce symbolical file dumps, and auxiliary perl program can be synthesized by the tool which communicates with the Verilog model using pipes to achieve this behavior. The problem specific view makes debugging the functional part of a component more straightforward.

7 Results

A novel approach for interface synthesis has been presented with a focus on functional adaptation instead of protocol adaptation. The proposed tools allow rapid development of new interfaces. The created interfaces are easy to modify and to extend, and a set of models and documentation can be generated consistently. The tools enable to make tradeoffs in the implementation with a small effort. This makes the tool useful when developing a large number of independent and rapidly changing IP components for different bus systems.

A similar set of tools has been successfully used in the development of another chip set which was more complex than the Infineon's Cerberus_FPI [16] project. The relative modeling effort for testbench, bus interface, and application specific hardware were comparable though. Although this project used an earlier methodology in its tools missing several key concepts, the development time savings were significant. The development time spent on testbench and bus interface could be reduced by almost an order of magnitude.

Thanks to Thomas Wild, Walther Stechele, Stephan Herrmann, Axel Hof, and Ingolf Ruge

References

[1] VSI Alliance, editor. *Virtual Component Interface Standard*, chapter On-Chip Bus Development Working Group. VSI Alliance, 2000.

[2] VSI Alliance, editor. *Virtual Component Interface Standard*. VSI Alliance, 2000.

[3] ARM Ltd. *AMBA Specification 2.0*.

[4] Peter J. Ashenden. *The Designer's Guide to VHDL*. Morgan Kaufman, 1996.

[5] Pai Chou, Ross Ortega, and Gaetano Boriello. Synthesis of the hardware/software interface in microcontroller-based systems. In *ICCAD '92*, 1992.

[6] Pai Chou, Ross B. Ortega, and Gaetano Boriello. The chinook hardware/software co-synthesis system. In *ISSS '95*, 1995.

[7] Pai Chou, Ross B. Ortega, and Gaetano Borriello. Interface co-synthesis techniques for embedded systems. In *ICCAD '95*, 1995.

[8] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[9] intel. *Accelerated Graphics Port Interface Specification Revision 2.0*.

[10] intel corp. *Rambus technical manual (disappeared from the web)*.

[11] Bjarne Hald Jan Madsen. An approach to interface synthesis. In *ISSS '95*, 1995.

[12] Kazushiro Shirakawa Kazushige Higuchi. Innovative system-level design environment based on form for transport processing system. In *DATE '98*, 1998.

[13] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 1988.

[14] Kathleen Lensen and Nicolaus Wirth andA. Mickel. *Pascal User Manual and Report: Iso Pascal Standard*. Springer Verlag, 1991.

[15] John R. Levine, Tony Mason, and Doug Brown. *lex & yacc*. O'Reilly & Associates, 1990.

[16] Albrecht Mayer. Systemc applications. Slides at SystemC User Group Meeting, Munich, Jan. 31st, `http://www-ti.informatik.uni-tuebingen.de/~systemc/Documents/Presentati%on-Mayer.pdf`, 2000.

[17] Manfred Glesner Michael Gasteiner. Bus-based communication synthesis on system-level. In *ISSS '96*, 1996.

[18] Carsten Mielenz. Common api/vsia. Infineon Proprietary.

[19] Sanjiv Narayan and Daniel D. Gajski. Interfacing incompatible protocols using interface process generation. In *DAC '95*, 1995.

[20] Ross B. Ortega and Gaetano Boriello. Communication synthesis for embedded systems with global considerations. In *CACHE '97*, 1997.

[21] Ross B. Ortega and Gaetano Borriello. Communication synthesis for distributed embedded systems. In *ICCAD '98*, 1998.

[22] Roberto Passerone and James A. Rowson. Automatic synthesis of interfaces between incompatible protocols. In *DAC '98*, 1998.

[23] PCI SIG. *PCI Local Bus Specification Rev. 2.2*, 2000.

[24] Thomas Brenner Rolf Ernst, Jörg Henkel. Hardware-software cosynthesis for micro-controllers. *IEEE Design & Test of Computers*, 1993.

[25] Frank Schirrmeister and Stan Krolikoski. Modeling techniques for evaluation and configuration of system level intellectual property. Cadence Whitepaper, `http://www.cadence.com/whitepapers/vcc.html`.

[26] Bruce Schneider. *Applied Cryptography*. John Wiley & Sons, 1995.

[27] Andrew Seawright, Ulrich Holtmann, Wolfgang Meyer, Barry Pangrle, Rob Verbrugghe, and Joseph Buck. A system for compiling and debugging structured data processing collectors. In *Euro-DAC '96*, 1996.

[28] Mani B. Srivastava, Brian E. Richards, and Robert W. Broderson. System level hardware module generation. *IEEE Transactions on VLSI Systems*, 1995.