Arbitrary Hardware Software Trade-Offs

Peter F.A. Middelhoek
University of Twente
P.O. Box 217, 7500 AE Enschede, The Netherlands
e-mail: pfam@cs.utwente.nl

Abstract

This paper discusses a novel transformation based design methodology and its use in the design of complex programmable VLSI systems. During the life cycle of a complex system the optimal trade-off between, partially, implementing in hardware or software is changing. This is due to varying system requirements (short time-to-market, low-cost, low-power etc.) and improving device technology. The proposed methodology allows such redesigns to be made, using different hardware-software trade-offs, in a guaranteed correct way.

1. Introduction

During the life cycle of a system the optimal trade-off between software versus hardware implementation changes. Initially time-to-market and rapid prototyping are the primary objectives, which require the use of flexible general-purpose reusable and programmable hardware. Later on in its life cycle other considerations become important. High-end features move down towards the domain of mass production where implementation cost becomes the dominant driving force. Other requirements, such as low power consumption, are also likely to appear. In addition device technology improves during the life cycle allowing for more sharing and reuse of hardware while offering the same throughput. These requirements and possibilities, while maintaining the same functional behavior, call for completely different target (hybrid) hardware/software architectures. Many examples of such retargeted designs can be found in both the computer and consumer electronics market.

Current tools for high-level synthesis do not offer such flexibility. To the contrary, they have often been designed with very narrow application domains and target architectures in mind, as a result they only allow the exploration of a very small part of the total design space. Examples are the Cathedral compilers [1] which partition the application domain based on throughput requirements and regularity of the algorithms. As target are used for instance parameterized multiple Von Neumann architectures or regular array style processors. This partitioning was needed to allow the construction of efficient synthesis tools. Clearly such an approach is a severe limitation when targeting the complete life cycle of systems. Therefore we propose a novel design methodology, implemented in our system TRADES [2, 3, 4], which is independent of a specific target architecture and supports the complete life cycle of a product. It does so by allowing arbitrary trade-offs between hardware or software implementation to be made in a guaranteed correct way. At the same time we are targeting the problem of interfacing different languages used for specification, such as VHDL, Silage and C, and synthesis tools. This is especially important when covering both the hardware and software domain and supporting the partial transition between the two.

This paper will show how, starting from a single high-level specification completely different guaranteed correct implementations can be derived. The implementations range from full-custom hardware to software implementations. We will show how the target architecture can be derived *from* the specification instead of being input to the design process. Section 2 gives a short introduction and demonstrates the methodology by means of a small example IIR design. A more detailed look at our methodology is given in section 3. In section 4 the results of the design of both a prototype implementation and a very efficient custom implementation of an interlaced-to-progressive scan conversion processor are summarized.

2. Design methodology, representation and transformations

The purpose of high-level synthesis is to *optimize* and *refine* behavioral descriptions and *assign* the resulting operations a location in time and space, such that an efficient realization in a combination of hardware and software can be obtained in a straightforward way. Optimizations deal with changing the structure of the algorithm while refinement is responsible for reducing the level of abstraction. These steps must be performed in a guaranteed correct way. Minimizing the combined cost associated with optimization, refinement, assignment and guaranteeing design correctness is the objective of formal methods for high-level synthesis [5]. In this section our design methodology, design representation and the use of *pre-proven behavior-preserving transformations*, which guarantee the design of correct implementations, are introduced. Focus is on the assignment of operations in time and space. Section 3 will elaborate on our approach towards high-level synthesis.

As design representation and backbone for the synthesis path we have selected a control data flow graph (CDFG) representation which integrates both control and data flow into a *single* graph notation. Specifications written in for instance VHDL, Silage or C are translated into this CDFG. The use of CDFGs and the similar signal flow graphs (SFGs) has proven to be very useful in high-level synthesis [6]. Within TRADES we are using SIL [7, 8, 9] which is a CDFG language

1074-6005/95 $04.00

```
// first order IIR-Filter //
for ever do
    y = x + y@1 * c
od
```

Figure 1. Pseudocode representation of a first order IIR filter

cooperatively developed by Philips Research, IMEC and the University of Twente as part of the ESPRIT/SPRITE project. SIL has formal semantics [10], needed for transformation verification, and includes support for hierarchy (procedural abstraction), structured data types (data abstraction), recursion, multi-rate, ordering and control structures. In this paper a somewhat simplified version of SIL is used.

Figure 1 shows the specification in pseudocode of a first order IIR filter. The first step in our design flow consists of the translation of this specification into a CDFG like representation as shown in figure 2. This representation closely matches the implementation suggestion contained in the specification, i.e. the output of the filter is calculated by adding the current input to the delayed output multiplied by the filter coefficient. This process is repeated infinitely often, as is usually the case in digital signal processing applications. The suggested hardware implementation consists of an interconnected adder, multiplier and register. The CDFG notation is actually a short hand notation for the infinite data dependency graph (DDG) as shown in figure 3. The latter suggests an implementation consisting of an infinite number of adders and multipliers operating in parallel. Although the behavior of both representations is equal the suggested implementation is very different. By applying transformations to the DDG that change the *folding* (structure in time) and the structure in space of the computations, different implementations can be suggested. This process is similar to the folding of DDGs (operation spaces) onto an array of processing elements as used in regular array synthesis [11], only now we are also targeting non-regular structures and offer transformations for changing the structure of the algorithm and the level of abstraction .

Many different foldings for a particular DDG are possible. While figure 1 showed the default folding figure 4 shows a partially folded multi-rate graph. This graph suggests two parallel operating interconnected first order IIR filter sections. The down sample (DS) and up sample (US) nodes with corresponding period and phase indicate that the input and output of the filter are operating at twice the frequency of the body. The original filter shown in figure 2 can not be retimed due to its recursive structure. The filter shown in figure 4 can be further transformed by means of common subexpression insertion, algebraic transformations and constant propagation into a form which can be effectively retimed. This design path was demonstrated in [12, 3] and resulted in an implementation which requires only half the power of the original folded filter as shown in figure 2, while

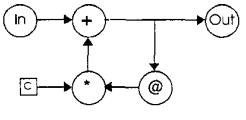

Figure 2. CDFG representation: folded operation space.

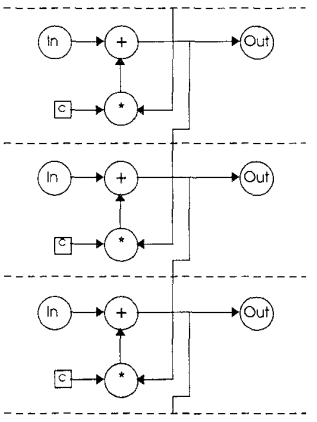

Figure 3. DDG representation: unfolded operation space

maintaining throughput. In [3] possible problems with the numerical accuracy of the filter that can occur when the computational structure of a filter is changed are discussed. Due to the use of behavior-preserving transformations these effects can be controlled precisely .

In figures 5-10 we will show how, by means of transformations, both a custom programmable processor architecture and the corresponding program required for implementing the IIR filter can be *constructed* from the specification. In a general processor architecture operations are performed in ALUs and interconnections are created by means of register and memory structures, which form a complex switch in time and space. Such an architecture is capable of exhibiting a class of behaviors. The specific behavior of the ALUs and registers is prescribed by the controller or program. Unfolding the folded CDFG, representing a processor, results in a DDG which consists of all possible interconnections and operations that can be used. Which interconnections and operations are actually used for the execution of an algorithm is determined by the controller or program.

The derivation of a custom processor architecture including its program, from the specification of an algorithm can be summarized as follows. Generalized and regular structures for both the functional aspect and the interconnections in the algorithm are created. The irregular or algorithm specific aspects are separated and used for program construction. The generalized operations and interconnections are assigned a location in both time and space (scheduling, allocation by means of folding). Figures 5-7 show the creation

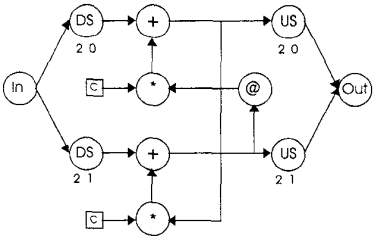

Figure 4. CDFG representation: partially unfolded operation space

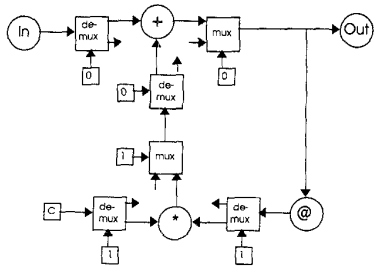

Figure 5. Insertion of (de)multiplexers with 'dead' input (output)

of these generalized operations by means of primitive behavior-preserving transformations. The irregularities in the functional behavior of the operators have been moved to the constants, which can later be transformed into one of the many alternatives available for generating these control signals.

Figure 8 shows a scheduling step which puts constraints on the folding in time. In this case the scheduling step could have been skipped since the causality of the data dependencies restricts the possible foldings sufficiently.

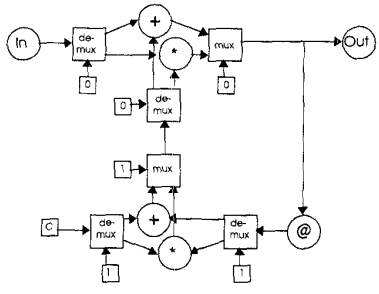

Figure 6. Insertion of 'dead' operators

After scheduling the graph can be folded. In figure 9 folding of the operations is shown. The constants 0 and 1 ensure that the ALU toggles between a multiply and an add operation. Figure 10 shows the result once the interconnections are also generalized and folded in time. The implementation now consists of one ALU, a register and some (de)multiplexers. The constants 0 and 1 in combination with the control of the US, DS nodes form the simple program and can be further transformed into different implementations, such as a trivial one bit counter or read from a memory.

The processor in figure 10 is very simple and can only be used for the implementation of very few algorithms. It is also not very efficient because the adder and the multiplier in the ALU are only operating half of the time. Efficiency could, however, be improved by decomposing the multiplier by

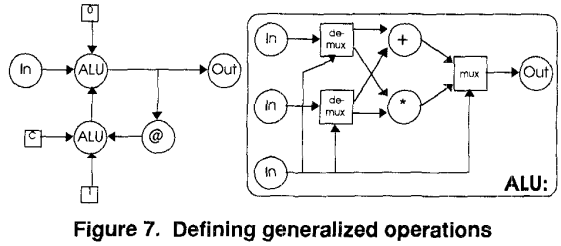

Figure 7. Defining generalized operations

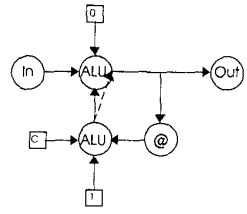

Figure 8. Scheduling of the operations

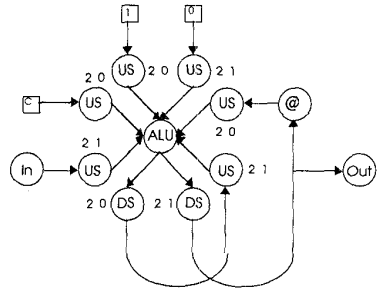

Figure 9. Folding the operations

means of a strength reduction transformation in a combination of adders and shifters and merging the adders.

In figure 11 a more general processor architecture containing one ALU, for multiply and add operations is shown. The operands can be read from one of the three registers, can be a constant or an input sample. The output of the ALU can be stored in a register or be the output of the system. The instruction-format of the processor consist of a concatenation of the control signals marked with *<signalname>*. This architecture and the corresponding program can be derived in a similar way as shown above by introducing more redundancy, for instance dead code, into the graph and folding it.

While the example is simple the same approach can also be applied to complex designs which use multiple, complex or pipelined ALUs, background memory or LIFO, FIFO memory structures or even more complex hybrid architectures. The powerful abstraction mechanisms in SIL ensure that such designs remain manageable. In section 4 the mapping of an edge direction detection algorithm, used in interlaced-to-progressive scan conversion algorithms, onto an existing multi video signal processor (VSP) based prototyping system is discussed.

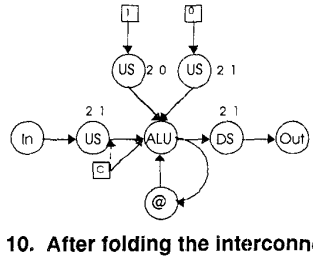

Figure 10. After folding the interconnections

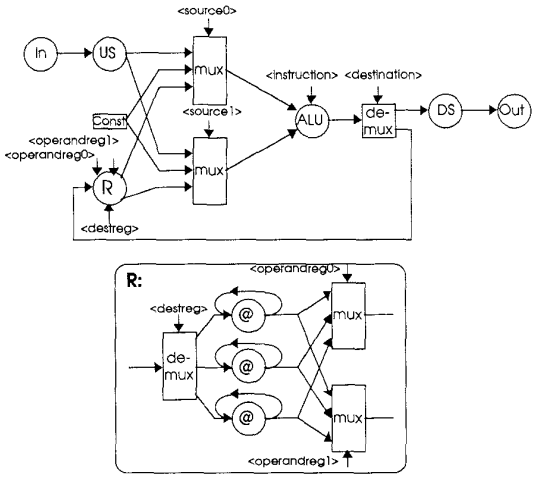

Figure 11. Generalized processor architecture

3. Transformational Design

The objectives of formal methods for high-level synthesis are minimizing the costs associated with the realization, design and achieving correctness of a system.

It is our belief that very efficient realizations can only be obtained by exploiting the flexibility and creativity of the designer. Due to the use of design automation the transistor density normalized towards the minimum process feature size has decreased significantly as reported in [13]. This indicates that silicon is not used as efficiently as it used to be. Most design systems address this problem by adding more user interaction to the design flow. Instead we have chosen an opposite approach by developing a user-centered design methodology, with automation as an add on. This focus on the user should also ease the acceptance of the method by designers. Within TRADES this implies that the selection of design steps (i.e. transformations) is done manually by the designer aided by a rule-based system which contains generalized optimization guidelines [4]. Since conventional cost functions strongly depend on a specific target architecture they can only be used with great care. Obviously some well understood parts of design flows, such as scheduling, can and will be automated. Some other transformational design systems do exactly this. They limit the problem domain to for instance data path optimizations in a limited throughput domain, for which efficient approaches are relatively well known, and use a small set (e.g. 30 [14]) of transformations which can be applied automatically. Unfortunately these approaches do not scale very well to the full-scale transformational design of complex systems, because they rely too much on hard coding of (sequences of) transformations used for limited look ahead which again is needed for design space exploration. The number of possible sequences explodes when the number of transformations increases. We know of at least 100 generalized transformations which are needed for full-scale transformational design.

In theory silicon compiler systems offer fast design times by offering push the button design flows by utilizing complex optimization algorithms. This allows designers to quickly evaluate different design alternatives. As mentioned before such an approach will often not result in very efficient realizations and is inflexible with respect to the supported target architectures. To address the first problem user interaction is added to the design flow. Silicon compilers are however complex black-boxes making them difficult to manipulate by a designer.

Instead, within TRADES we concentrate on and automate the correctness aspect of the design and allow the designer to concentrate on efficiency. Achieving correctness can easily account for over 30% of the total design effort as is well known from the design of complex software systems [15]. Therefore, when using a transformational design methodology the total design time consisting of design and achieving correctness can be comparable to that of modern silicon compilation systems. At the same time our methodology offers more efficient realizations and supports a larger design space and the complete life cycle of a design.

Three approaches towards design correctness are possible: simulation, verification and correctness-by-construction [16]. Currently the former is the only practical way of checking the behavior described by a specification against the user requirements, while only the latter two can be qualified as formal methods. We have selected the third, transformation based approach where the design flow from specification to implementation consists of the consecutive application of pre-proven behavior-preserving transformations. Transformations are correct if the set of behaviors allowed by the implementation is a subset of the behaviors permitted by the specification. Correctness is guaranteed because all generalized design steps (i.e. transformations) are verified [17] to be correct in advance: i.e. the behavior of the design before and after the application of a transformation are the same. The main difference with verification is that correctness in transformational design is integrated *in* the top-down design flow while verification is a step performed *after* synthesis steps. In its purest form transformational design uses pre-verified generalized transformations. The correctness of the instantiated transformation is implied by the proof for the generalized transformation. Advantage is that designers can not make incorrect implementations and can concentrate on creating efficient designs. Many systems however, use a mixed approach in which transformations which are 'often' correct are defined and after the application of each or all transformation(s) the correctness of the design is established through verification. An example of the latter is SynGuide, which uses a geometric, polyhedral based model for the verification of loop transformations [18].

3.1 Implementation Suggestion

The design of the IIR filter has shown that the implementation suggestion plays an important role. While the alternative designs all exhibit the same behavior, they differ in the implementation suggestion. A specification contains an implementation suggestion because the only practical way of validating a specification against the user requirements is to execute the specification. Execution requires an implementation. Several systems try hard to make their synthesis results independent from the way in which the specification is written. They consider the implementation suggestion an unwanted side-effect of the specification

process. Examples include [14] which uses a (pseudo) normalization step to dispose of it. In [19] a representation format, targeted at the translation of VHDL, is developed which should be capable of representing all equivalent behaviors in one unique way. Besides the fact that neither systems achieves such a unique representation we believe it is not preferable either. The implementation suggestion of a specification can contain valuable information provided by the designer which can be used to find an efficient implementation. Therefore we try to preserve the implementation suggestion during the translation from for instance VHDL to SIL as much as possible. On the other hand the purpose of synthesis is to change this implementation suggestion in such a way that realizations with lower cost are obtained. Techniques presented in [14 and 19] can be used for changing the implementation suggestion.

3.2 Related work

Several design systems for VLSI design based on the concept of transformational design exist or are under construction as part of larger projects (e.g. HYPER [20, 12, 21, 22], CAMAD [23, 24], SynGuide [25, 18], GATE [14], Yorktown [26], ESPRIT/FORMAT[27]) however most of them are restricted to optimizations at a single abstraction level (algebraic, loop, common subexpression elimination, retiming and scheduling transformations), some, such as GATE and HYPER, include limited refinement by means of strength reduction transformations. Although such approaches have proven to be useful in many cases (examples include optimizing resource utilization, critical path minimization, power reduction, increased testability, all within the context of HYPER, memory reduction [18], and scheduling [23, 24]), we found that they are too limited for full scale transformational design. Therefore the set of transformations supported in our system TRADES [28, 29] is much larger and includes transformations for procedural abstraction (i.e. hierarchy), data abstraction (i.e. type transformations) but also transformations for the manipulation of control structures and, as demonstrated in section 2, time-space mappings. Although others (such as [24]) are investigating hardware/software codesign we believe that our system and representation format has several advantages because SIL has been designed as intermediate format, which preserves the original implementation suggestion, between different specification languages and high-level synthesis tools. Therefore it supports a large application domain and many target architectures. Which among others allows us to support the complete life cycle, starting from a single specification in a guaranteed correct way.

4. Design of an Edge Direction Detector

As a test case for our transformational design methodology we have selected the design of the edge direction detection section of interlaced-to-progressive scan conversion (IPSC) algorithms. IPSC algorithms are used to double the screen retrace frequency by interpolating intermediate scan lines of a field of an interlaced frame. If an edge is present in a field, interpolation takes place in the direction of the edge. Detection of edges is based on the gradient in the luminance. Three gradients are calculated based on the difference in luminance of three pairs of opposing pixels in the line before

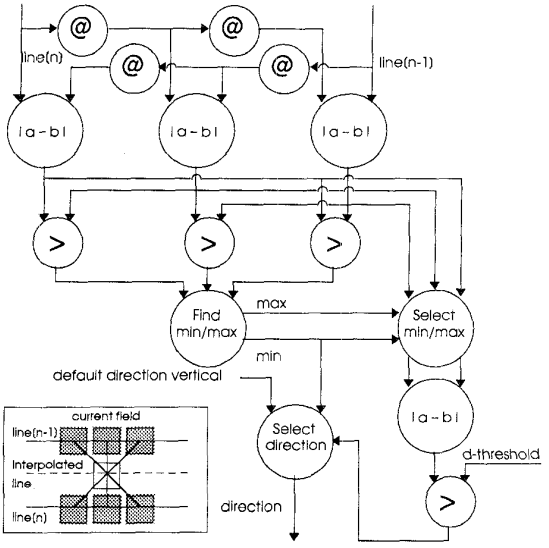

Figure 12. Architecture of a direction detector for IPSC algorithms.

the estimated line and in the next. The smallest gradient indicates the direction of the edge. This algorithm in combination with linear interpolation of the luminance signal is known as the *edge-based line average* (ELA) algorithm. A diagram of the direction detector is shown in figure 12. A more detailed description can be found in [4, 30]. The direction detector was used in combination with an extended form of the ELA algorithm which first feeds the input signal through a low-pass filter and uses median interpolation instead of linear interpolation. This last modification requires the luminance of the pixel from the previous field with the same spatial location as the pixel to be estimated. Extensions to this filter have recently been proposed in [31], which also contains a more complete overview of the algorithm and its history.

We have selected the direction detector algorithm because it is relevant in industry and currently used in television systems. Furthermore, the example has been studied extensively both at Philips Research [32] and IMEC [33] as part of the ESPRIT/SPRITE project and both prototype and custom implementations have been developed. This allows us to compare our results with respect to efficiency and flexibility with those obtained previously using different methodologies and tools.

Using our methodology we designed two implementations for the direction detection algorithm, starting from a single high-level VHDL specification. A prototype software implementation was designed for realization on a multi video signal processor based prototyping system developed by Philips [34] and also an efficient full custom hardware implementation. Both design flows are illustrated in figure 13. More details on the design experiment can be found in [30]. We used the commercial View Synthesis tools to derive an additional full custom implementation which allows us to compare our design results with those generated using existing commercial synthesis tools.

Our design flows start by writing a high level VHDL specification for the direction detector as shown in figure 10.

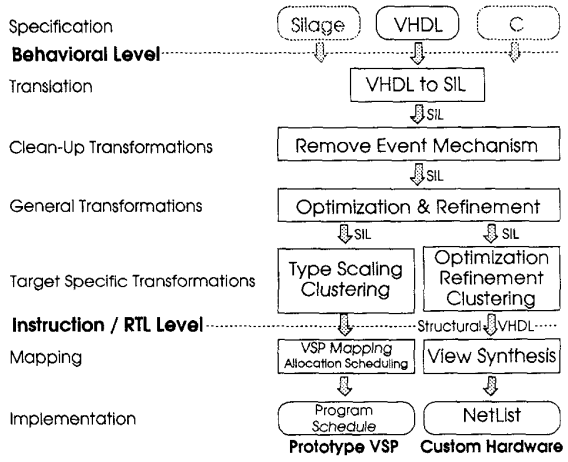

Figure 13. Design flows for prototyping and custom hardware design

We used our syntax based VHDL to SIL compiler [35] which preserves the initial implementation suggestion including the VHDL event mechanism. The first transformation step consists of the reduction of the event mechanism. The result is similar to figure 12. This is followed by a first round of target independent optimizations, which concentrated on reducing the redundancy present in the combination of the comparators, select min/max and la-bl blocks. For the prototype implementation we had to transform the graph in such a way that the behavior of groups of nodes corresponds to the instructions of the target VSP processors; this is called a clustering step which is done by means of hierarchy introduction. Notice that before clustering, the data types in the original graph have to be scaled to the word length of the target architecture. This step can also be done by means of behavior-preserving transformations. The result is an instruction stream which is not yet scheduled and allocated. We used existing VSP programming tools for folding (scheduling and allocation) the graph. This is similar to section 2; only now the target architecture is already known. Mapping to a predefined target architecture requires modeling this architecture or in our case the instruction set in SIL. A unification transformation is used to 'bind' the clustered algorithm to the target architecture. The multi VSP prototyping system can now be used to quickly evaluate in real time complex video processing algorithms. Using our compiled code based SIL simulator running on a fast HP workstation required about 30 seconds per frame.

The custom implementation allowed us to perform several more transformations because its target domain is much more flexible than the instruction set of a processor. Further refinement and the exploration of special cases at the expense of a somewhat more complex control structure resulted in significant savings. To achieve the required high throughput, which requires parallelism, the design had to be pipelined. Alternatively, based on the approach described in section 2, we could also have designed a processor/program like architecture, somewhat similar to the VSPs used in the prototype implementation, with multiple in parallel operating ALUs that perform the absolute difference function and ALUs for the comparator and find min/max sections. This does however not seem that efficient because pipelining is a lower-cost mechanism for achieving the desired parallelism. View Synthesis was used as a low-level synthesis back end to the design flow. The direction detector is used as a processing element within the Phideo design flow developed at Philips Research [32] which takes care of scheduling and generating the interconnection and memory structure. Figure 13 illustrates both design flows. In [4, 30] more details on the custom implementation have been reported.

4.1 Results

Since the step from specification to implementation is guaranteed correct the step from prototype to final implementation is also guaranteed correct. Because our design system TRADES is still in a prototype state we checked the correctness of our designs by means of extensive simulation and compared the results with those obtained from simulations of previous implementations and the specification. Both our designs turned out first-time-right and we were able to find an error in the original prototype implementation of the direction detector.

Our results for the custom implementation, discussed in detail in [4, 30], show a reduction in area in terms of gate count of roughly 50% if compared to the optimized designs described in [32, 33]. When compared to the synthesis results obtained with View Synthesis the improvement is 75%, from 2613 down to 685 gates. In addition our implementation is about twice as fast and would require less than half the power as compared to the optimized designs [22, 23]. For the prototype implementation the improvement in the number of instructions over the previous manually designed implementation was about 10%. This relatively small gain was caused by the difficulty of efficiently implementing the more complex control structure which becomes a dominant factor in the small algorithm.

Since TRADES is still a prototype system, estimating the design times is rather difficult. The actual design time of about two to three weeks, was however much less than the effort required for gathering all information necessary for making the designs. Developing the second implementation also required much less effort because both design paths are largely merged. Also we are still at the beginning of the learning curve with our methodology. Hence, significant improvements in design time are very likely.

Conclusions

Our approach towards transformational design has proven very useful in supporting the life cycle of a design by offering a much larger design space than current tools for high level synthesis do. This was achieved by generating the architecture out of the algorithm specification. We have demonstrated how by changing the amount of time-folding, which translates into a trade-off between hardware and software, designs with completely different costs in terms of area, speed, and power can be obtained. In addition we have shown the method's usefulness in generating efficient designs and achieving design correctness in high-level synthesis. Using the proposed design methodology we were able to obtain very efficient first-time-right designs and in addition find an error in a previous prototype implementation of an IPSC algorithm.

Acknowledgments

The author wishes to thank his colleagues at the University of Twente, Philips Research and IMEC for their contributions. This research was funded by Philips Research.

References

[1] H. De Man, J. Rabaey, P. Six, L. Claesen, *Cathedral-II: A Silicon Compiler for Digital Signal Processing*, IEEE Design & Test, December 1986.

[2] P.F.A. Middelhoek, *Transformational Design of Digital Circuits*, Proc. of the Seventh Computersystems Workshop, pp. 57-69, Eindhoven, November 1993.

[3] P.F.A. Middelhoek, *Transformational Design of Digital Signal Processing Applications*, Proc. of the ProRISC/IEEE Workshop on CSSP, pp. 176-180, Arnhem, March 1994.*

[4] P.F.A. Middelhoek, *Transformational Design of a Direction Detector for the Progressive Scan Conversion Processor*, CS Memorandum 94-64, University of Twente, October 1994.*

[5] F.K. Hanna M. Longley, N. Daeche, *Formal Synthesis of Digital Systems*, Formal VLSI Specification and Synthesis, VLSI Design Methods, I, Editor L.J.M. Claesen, Elsevier, 1990.

[6] R. Camposano, R.M. Tabet, *Design representation for the synthesis of behavioral VHDL models*, Proc. 9th Int. Conf. on CHDL, pp. 49-58, eds. J.A. Darringer and F.J. Ramming, Elsevier Science Publishers B.V., North Holland, June 1989.

[7] W.E.H. Kloosterhuis, M.M.R. Eyckmans, J. Hofstede, C. Huijs, Th. Krol, O.P. McArdle, W.J.M. Smits, L.G.L. Svensson, *SIL-1 Language Report*, SPRITE deliverable LS.a.a/Philips/Y3-M12/2, 1992.*

[8] W.E.H. Kloosterhuis, M.M.R. Eyckmans, Th. Krol, O.P. McArdle, W.J.M. Smits, *SIL-2 Language Report*, SPRITE deliverable LS.a.2/Philips/Y3-M12/1, 1993.*

[9] Th. Krol, J. van Meerbergen, C. Niessen, W. Smits, J. Huisken, *The Sprite Input Language: An Intermediate Format for High-Level Synthesis*, Proc. of EDAC 92, pp. 186-192, Brussels, March 1992.

[10] C. Huijs, Th. Krol, *A Formal Semantic Model to Fit SIL for Transformational Design*, Proc. of 20th Euromicro Conference, pp. 100-107, Liverpool, September 1994.

[11] S.Y. Kung, *VLSI Array Processors*, Prentice Hall, 1988.

[12] A. Chandrakasan, M. Potkonjak, J. Rabaey, R. Broderson, *An Approach For Power Minimization Using Transformations*, Proc. of the IEEE VLSI Signal Processing Workshop, pp. 41-50, 1992.

[13] C. Piguet, Design Methodologies and CAD Tools, Proc. of the CICC 89, 1989.

[14] M. Janssen, F. Catthoor, H. de Man, *A Specification Invariant Technique for Operation Cost Minimisation in Flow-Graphs*, Proc. of the Seventh International Symposium on High-Level Synthesis, pp. 146-151, Niagara-on-the-Lake, Ontario, Canada, 1994.

[15] R.S. Pressman, *Software Engineering: a Practitioner's Approach*, 3th ed., p. 107, McGraw-Hill, Inc., New York, ISBN 0-07-112779-8, 1992.

[16] H. Eveking, *Verification, Synthesis and Correctness-Preserving Transformations - Cooperative Approaches to Correct Hardware Design*, HDL Descriptions to Quaranteed Correct Circuit Designs, Editor S. Borrione, Elsevier Science Publishers B.V., North Holland, 1987.

[17] P.S. Rajan, *Transformations in High-Level Synthesis: Formal Specification and Efficient Mechanical Verification*, technical report, SRI-CSL-94-10, SRI International, Menlo Park, October 1994.*

[18] H. Samsom, F. Franssen, F. Catthoor, H. de Man, *Verification of Loop Transformations for Real Time Signal Processing Applications*, Proc. of VLSI Signal Processing VII, IEEE, 1994.

[19] V. Chaiyakul, D.D. Gajski, L. Ramanchandran, *High-Level Transformations for Minimizing Syntactic Variances*, Proc. of DAC 93, pp. 413-418, 1993.

[20] R.W. Brodersen, et al, *Anatomy of a Silicon Compiler*, Kluwer, ISBN 0-79239249-3, 1992.

[21] Z. Iqbal, M. Potkonjak, S. Dey, A. Parker, *Critical Path Minimization Using Retiming and Algebraic Speed-Up*, Proc. of DAC 93, pp. 573-577, 1993.

[22] M. Potkonjak, J. Rabaey, *Optimizing Resource Utilization Using Transformations*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 13, no. 3, March 1994.

[23] Z. Peng, K. Kuchcinski, B. Lyles, *CAMAD: A Unified Data Path/Control Synthesis Environment, Design Methodologies for VLSI and Computer Architecture*, pp. 53-67, Editor D.A. Edwards, 1989.

[24] Z. Peng, K. Kuchcinski, *Automated Transformation of Algorithms into Register-Transfer Level Implementations*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 13, No. 2, pp. 150-166, 1994.

[25] H. Samsom, L. Claesen, H. De Man, *SynGuide: An Environment for Doing Interactive Correctness Preserving Transformations*, Proc. of VLSI Signal Processing VI, pp. 269-277, Eds. L.D.J. Eggermont, P. Dewilde, E. Deprettere and J. van Meerbergen, IEEE, 1993.

[26] R.K. Brayton, R. Camposano, G. DeMicheli, R.H.J.M. Otten, J.T.J. van Eijndhoven, *The Yorktown Silicon Compiler System*, eds. D. Gajski Addison-Wesley, 1988.

[27] W.D. Tiedeman, S. Lenk, C. Grobe and W. Grass, *Introducing Structure into Behavioral Descriptions obtained from Timing Diagram Specifications*, Proc. of EUROMICRO 93, barcelona September 1993.

[28] W. Engelen, P.F.A. Middelhoek, C. Huijs, J. Hofstede, Th. Krol, *Applying Software Transformations to SIL*, SPRITE deliverable LS.a.5.2/UT/Y5/M6/1A, June 1993.*

[29] W. Engelen, *Applying Software Transformations with Typing to SIL*, SPRITE deliverable LS.a.5.4/UT/Y5/M12/1a, December 1993.*

[30] P.F.A. Middelhoek, G.E. Mekenkamp, E. Molenkamp, Th. Krol, *A Transformational Approach to VHDL and CDFG based High-Level Synthesis: a Case Study*, accepted for CICC 95, 1995.*

[31] M.H. Lee, J.H. Kim, J.S. Lee, K.K. Ryu, D.I. Song, *A New Algorithm for Interlaced to Progressive Scan Conversion Based on Directional Correlations and Its IC Design*, IEEE Trans. Consumer Electronics, vol. 40, No. 2, pp. 119-125, May 1994.

[32] P.E.R. Lippens, J.L. van Meerbergen, A. van der Werf, W.F.J. Verhaegh, B.T. McSweeney, J.O. Huisken, O.P. McArdle, *PHIDEO: A Silicon Compiler for High Speed Algorithms*, Proc. of EDAC 91, pp. 436-441, Amsterdam, February 1991.

[33] Z. Sahraoui, L. Rijnders, *Report on the DIRDET Experiments*, internal report IMEC, 1992.

[34] A.H.M. van Roermund, P.J. Snijder, H. Dijkstra, C.G. Hemeryck, C.M. Huizer, J.M.P. Schmitz, R.J. Sluijter, *A general-purpose programmable video signal processor*, IEEE Trans. Consumer Electronics, vol. 35, No. 3, pp. 249-258, August 1989.

[35] E. Molenkamp, G.E. Mekenkamp, J. Hofstede, Th. Krol, *SIL: an intermediate for syntax based VHDL synthesis*, accepted for VIUF Spring 95, San Diego, April 1995.*

* available at http://wwwspa.cs.utwente.nl/aid/trades/trades.html