Development and Implementation of a 3.6 GFLOP/s SIMD-DSP using the Synopsys Toolchain

Hendrik Seidel, G. Cichon, E. Matúš, T. Limberg, P. Robelly, Gerhard Fettweis

Vodafone Chair, TU-Dresden
D-01062 Dresden

seidel@ifn.et.tu-dresden.de

ABSTRACT

Mobile communication systems, wireless and handheld terminals require low power, yet high performance DSPs. These DSPs are used in software defined radio applications and for computation of streaming media. To decrease time-to-market, development of these DSPs has to be done in a very short time. A design flow which is based on proprietary tools and powerful components from SYNOPSYS achieves this goal. In this paper a design methodology based on a novel architecture template called synchronous transfer architecture is presented. In a use case study a 3.6 GFPLOP/s SIMD digital signal processor was implemented and taped out. The whole implementation process from system idea to netlist, hence Architecture, Design Verification, Test and Synthesis Strategy are described.

Table of Contents

1.0 Introduction

Mobile communications systems require high-performance low-power DSPs for enabling flexible and thus software based implementations of wireless standards. In [1], the CATS group of the Vodafone Chair presented the M3 DSP. It is implemented on 289 mm^2 of a 0.35µm CMOS technology, and it is able to run a 1024 point FFT in 55 µs while consuming less than 1W of power. However, it required a lot of engineering work to build it and also to program it efficiently in assembly language.

For this reason, a new architecture for digital signal processors is required that allows for effective compiler support, and in which different algorithms share the available computational resources.

The resulting hardware should still be suitable for high-performance, low-power applications, in order to be used for battery-powered mobile terminals.

Furthermore, the design flow for the hardware implementation should be streamlined: As much as possible of the hardware models should be generated automatically from a core generator. Both the compiler and the core generator should use the same machine description in order to establish an integrated design flow (see Figure 1).

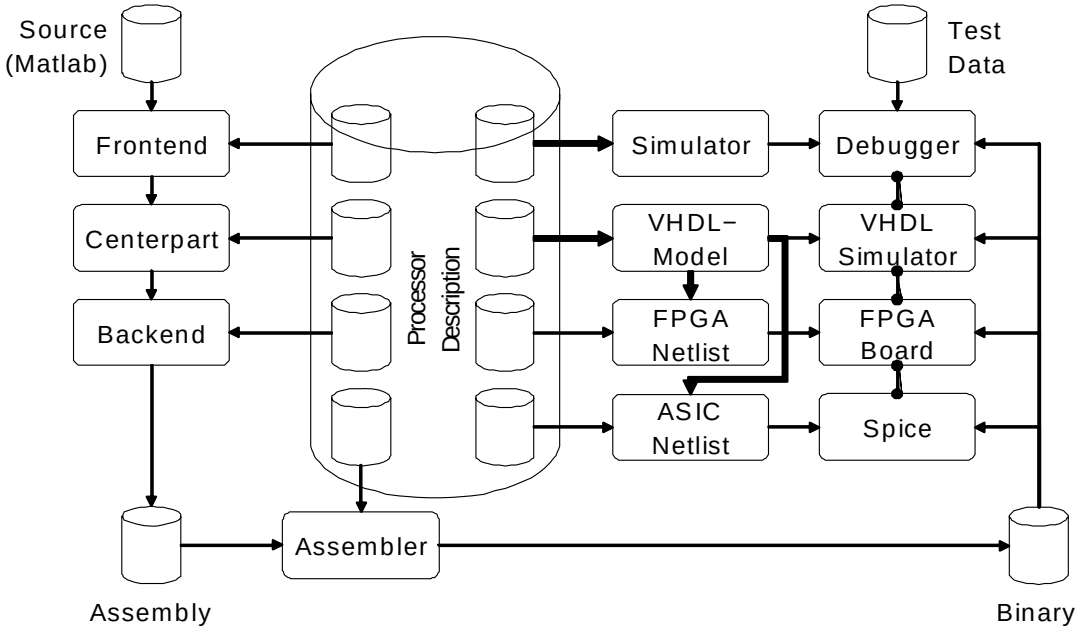

Figure 1: The Integrated Design Flow of MOUSE

1.1 Synchronous Transfer Architecture (STA)

In the Synchronous Transfer Architecture (STA), the functional units of a processor constitute a synchronous data flow network (see Figure 2). The switching matrix may implement arbitrary connections depending on the application, performance, and power-saving requirements.

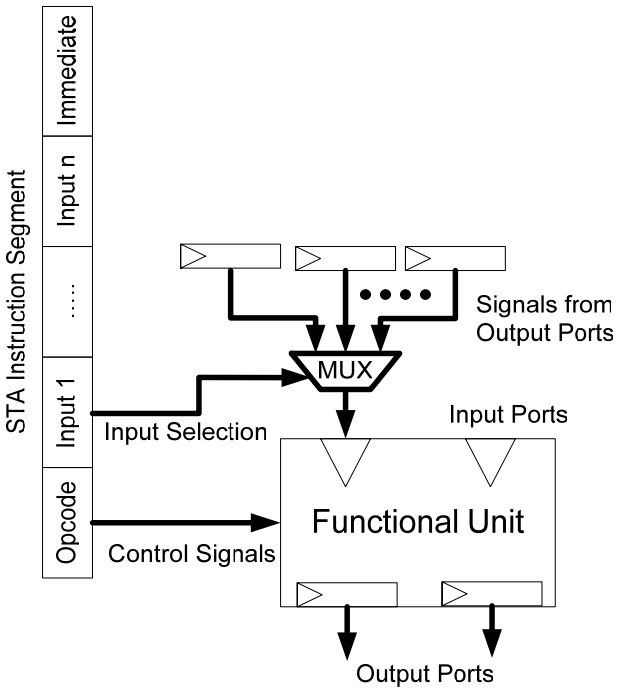

Figure 2: Architectural Template of STA

This architectural templates omits the expensive crossbar switches of VLIW for the construction of bypasses between the functional units, and TTA ‘s local queues for collecting operands and superscalar controllers that determines when exactly an operation is to be started. The entire communication between functional units takes place synchronously.

Another improvement of efficiency can be achieved by adopting SIMD architecture, in which several data processing resources can share a single instruction processing resource. In order to avoid instruction storage and dynamic scheduling overhead, STA proposes storing instructions in a compressed form that does not require many hardware resources to decompress. The most efficient ones seem to be 2D run-length encoding schemes [3].

Due to its regular structure combined with generic computational primitives, STA can effectively support compilers, as shown in [4].

It has successfully been demonstrated that systems which are based on this approach can be realized rapidly using the Mouse core generator developed by the CATS group. [5] presents an OFDM transceiver system which was realized in a very short time frame.

2.0 High-Level Design Flow

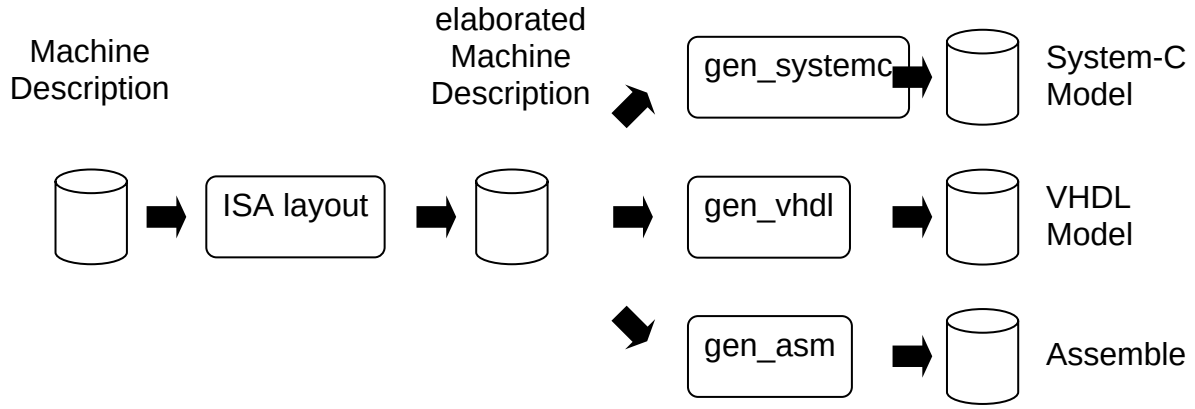

Figure 3: GENCORE: The core generation tool generates an RTL model of the design and all necessary tools for program development.

The design flow of our core generator is depicted in Figure 3.

2.1 ISA Layout

The core generator first computes an optimized instruction set layout for the given architecture. For this purpose, the binary encodings for instructions and multiplexer controls are determined. The binary encodings also determine the bit width of the corresponding fields in the instruction word. Correlated control fields are placed close to each other in the instruction word in order to improve compression efficiency. At last, a report about the instruction encoding is generated.

2.2 Simulation Models

For simulation and software debugging purposes, a cycle accurate machine model in SystemC is generated. The core generator handles the features necessary to support STA and SIMD constructs effectively. The generated model contains generated behaviors for all instructions, as well as all code related to the switching matrix.

2.3 Assembler

The core generator also generates an assembler to support a human-readable entry of parallel programs. This assembler also handles instruction compression.

2.4 Synthesis Models

The core generator generates a synthesis friendly VHDL model. The generated parts consist of a VHDL package with constant definitions, a core toplevel with interconnection wires and multiplexers, different types of decompressing instruction decoders, register files with debug capability, memories using technology dependent hard-macros with debug capability, and stubs for the implementation of functional units. In addition to that, the core generator generates synthesis scripts for ASIC- and FPGA-libraries. A memory bus to access the internal memories is inserted automatically.

The core generator makes no assumption on the implementation of the functional units. It turned out to be most advantageous to use optimized models obtained from third parties, e.g. the technology foundry, SYNOPSYS DesignWare, Altera Megafunctions, Xilinx DSP-Blocksets, etc.

3.0 Synthesis Strategy

The selection of a synthesis strategy for a fully automated design flow is not a trivial problem. On the one hand, synthesis results need to be competitive in order of maximum clock speed, area and power requirements, but on the other hand a unique synthesis process with low user interaction, regardless of the synthesized design is desirable for a rapid prototyping platform, in order, that the designer can concentrate on architectural features without mattering about synthesis. Figure 4 depicts the simplified version of the CATS design flow. As one can see, there are two branches from GENCORE. The SystemC path is important for the verification of the architecture. The HDL path can be used to verify, if design constraints can be met in every design stage. Not until both paths, SystemC as well as HDL, meet all the target constraints, the final design is reached. Using synthesis in early design stages helps finding and fixing architectural deficiencies and shortening design time.

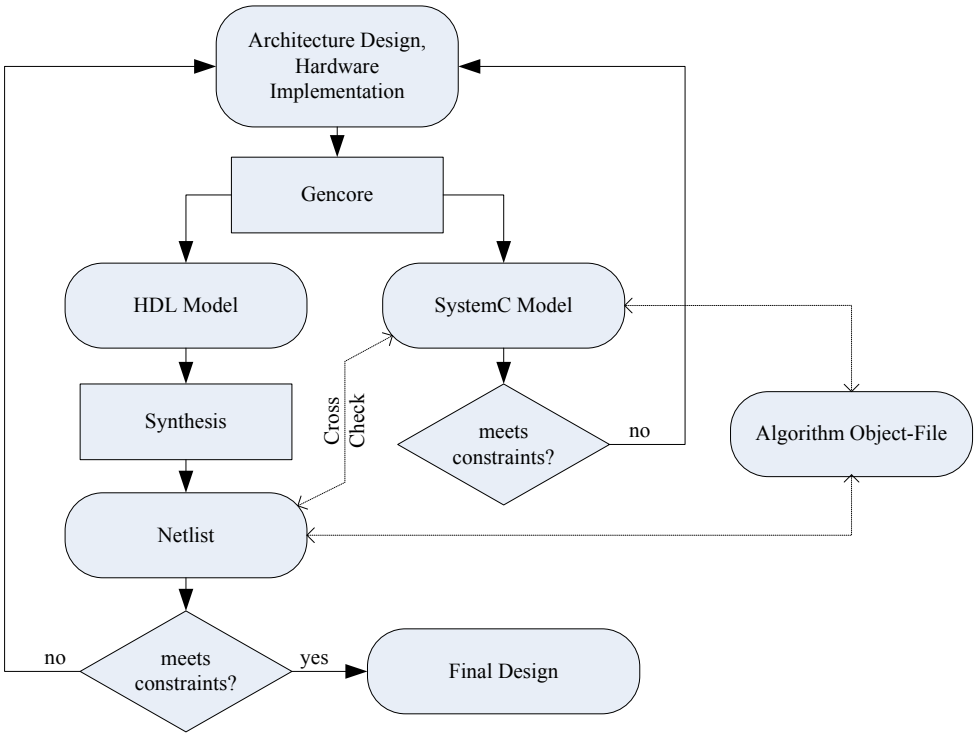

Figure 4: Simpyfied CATS design flow

Traditional design flows require experienced designers to get competitive synthesis results. Usually traditional methods lead to big numbers of manually modified highly heterogeneous compile scripts. Even minor architectural modifications can require changes to the scripts. Therefore, the system architect, working mainly with our machine description, needs to be an synthesis expert, too. This is unsatisfactory. To relieve the system architect from this auxiliary burden, it is desirable to automate the synthesis process as far as possible.
Another important consideration is the execution time of the synthesis process. The functional verification of the whole synthesized design is a very challenging task. Especially designs using clock gating combined with pipelining appear to be very error prone. Most of the errors pop up in conjunction with a stall signal, which may be generated by the instruction decompression unit. Very often errors are not detected, since very special programs states are required to make them appear. Usually not all of these program states are

considered by the test engineer. Therefore some errors may remain undetected before tapeout of the chip. Simulating target applications on a synthesized model, including all features of the final design helps finding hidden errors. Shorten synthesis time results in faster iteration cycles until the design is fixed.

3.1 Automated script generation

How can the requirements formulated above be met by our core generation tool? The key to successful implementation of a script generator for Synopsys synthesis tools is the knowledge of the design environment. This knowledge founds on our design template. All parameters of the interconnection network are well known to the core generator. Based on these information, gencore is able to calculate constraints for an initial synthesis sweep.

What compile strategy should be used? Synopsys supports bottom up, top down and mixed compile strategies. The answer is heavily dependent from design size and designers preferences. In almost every case a mixed compile is the best way. Bottom up compilation leads to longer run-times, while top-down compilation produces better results [6,7]. By default, Gencore selects a top down strategy for the parallel slice instances, while using the bottom up approach for the remaining design modules. The slice is compiled once and then instantiated the appropriate number of times. Thus, it is possible to apply hierarchical place and route in the backend tools.

The fully automated compile script generator is not always able to produce appropriate results. For this reason it is possible top set parameters on the synthesis. Using this feature, one is able to overwrite the parameters generated by default. Further it is possible to add functionality like clock gating or scan chains. In cases where it is useful, the configuration is implemented hierarchically. This means, if we set a parameter like clock uncertainty for the toplevel of our design, Gencore will use excactly the same value for all submodules unless no other value is specified for them.

After running the initial synthesis sweep, there are usually not all constraints met. This is, since Gencore can only estimate constraints from design environment. Now the characterize command is applied to the module instances and the characterization script is written out. If a characterization script for a module is present, this script is used for constraining in place of the core generator constraints. This process is repeated iteratively until all constraints are met.

3.2 Toplevel synthesis

The toplevel architecture is mainly conditioned by the manufacturer or PCB-designer, where the core level is conditioned by the requirements of the signal processing system.

The interface between core and toplevel has usually not to be changed during the design process. To be most flexible on the toplevel, we are doing the toplevel design manually and assign the delays and loads of the input and output pads with the characterize command to the core. After assigning suitable I/O pads we use the SYNOPSYS BSD-COMPILER to add a TAP-Controller and a JTAG Interface. To access the memory and Registers over the JTAG-Bus, additional JTAG-commands were added. If the JTAG-RST signal is raised, the clock tree is fed by the JTAG-clock instead of the PLL-clock. With a special JTAG-Register, a stepwise control of the core is possible. Other registers are used to start and stop the core and to read and write the memories. We automatically instance these control registers with the BSD-COMPILER. If the JTAG mode is active, all scan-chains of the core are connected to a single scan chain for simple access of all internal registers and to perform a core test after chip assembly. A single MBIST module in the toplevel is capable to test all implemented memories.

4.0 Samira DSP Core

With the high-level design flow programming-tools, VHDL-Model, and Instruction Set Simulator are generated for the Samira DSP Core. Our synthesis strategy was used to produce the netlist.

The Samira DSP Core is a DSP with 8 parallel SIMD Data Paths and a scalar data path for address generation and program control. The VLIW instructions are compressed using a mixture between Lempel-Ziv and differential compression. Each compressed instruction has a bitwidth of 128bit. The compressed instructions are decompressed to a 400bit Microcode. The decompression is done online and usually in a single cycle. If the information of the new instruction is too high, the VLIW decoder is able to stall the core until the decompression is completed. In every cycle 54 functional units in 8 SIMD data paths and 13 functional units in the scalar data path and their input multiplexers get new instructions. This allows for an extremely high utilization of the silicon area and the implementation of different signal processing algorithms ranging from mobile communication systems to streaming media applications. Clock gates for the functional units are inserted by the SYNOPSYS POWER COMPILER. The enable-signals for them are directly connected to the decompression decoder. Hence, if core stalls occur or not all functional units are used, the energy consumption is reduced to static power dissemination.

To interconnect the SIMD vector data paths an interconnection unit is used. This interconnection unit is able to broadcast values from the scalar data path to the vector data paths and is doing shift operations on word granularity. Hence, it exchanges the data between the vector data paths. The Core has three different memories: A scalar memory, to store temporal control data, like address pointers, counters, and configuration data. The vector memory is used as main memory for data processing. In the instruction memory programs are stored. This advanced Harvard architecture allows synchronous access of control, program and data memory. In addition each memory consists of multiple blocks allowing for synchronous access from a Host-Processor like ARM or MIPS while a signal processing algorithm is computed. In Figure 5 the overall architecture of the SAMIRA DSP is shown. Figure 6 shows one of the 8 Vector Data Paths.

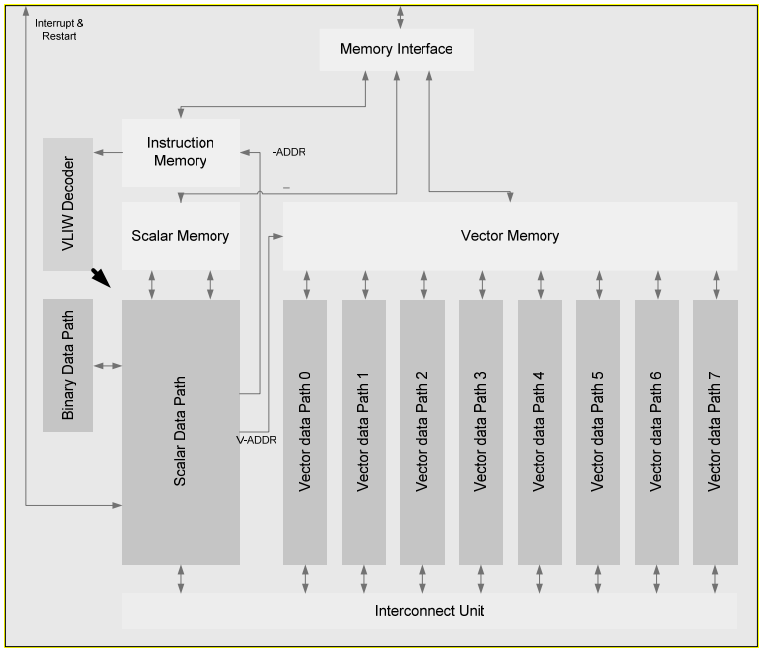

Figure 5: Architecture overview of the SAMIRA DSP

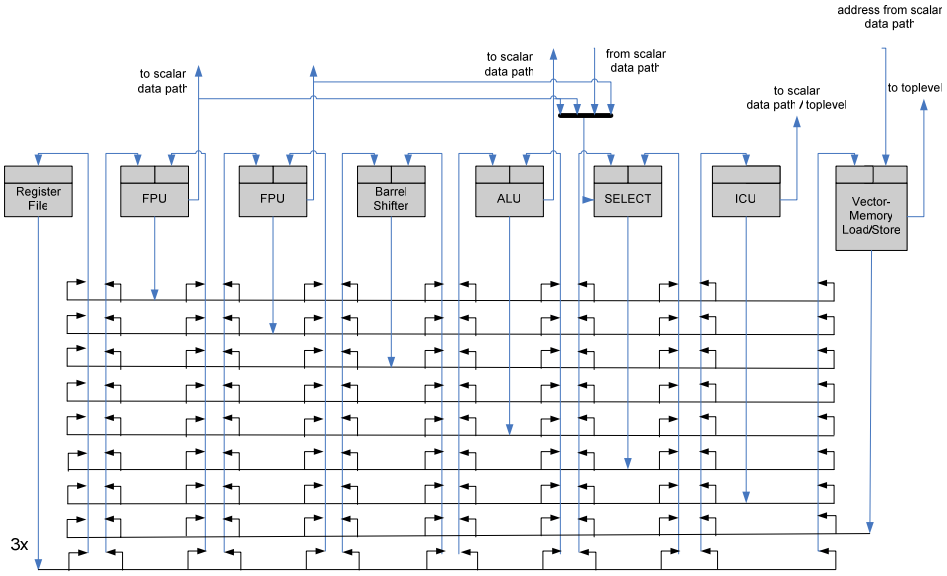

Figure 6: Architecture of a Vector Data Path

The SAMIRA-DSP-CORE is an accelerator for signal processing applications with hard real time constraints. In the first reference design it is connected to a FPGA with a NIOS II softcore as user interface. Having a simple SRAM interface it could be also connected to standard Microcontrollers. In our next silicon run an application tailored version will be placed in a complex system on chip as accelerator module.

5.0 Floating Point Unit (FPU)

Seventeen identical floating point units (FPU) are implemented in SAMIRA DSP: two FPUs in each data path of vector processing unit and one FPU in scalar processing unit. FPUs support operations addition (ADD), multiplication (MUL), float to integer (F2I) and integer to float (I2F) conversions on single-precision normalized 32-bit data types as specified by the IEEE-754 standard. SAMIRA architecture doesn't support operations on denormalized numbers. Restricted capabilities with respect to the IEEE standard are implemented. e.g. only rounding to nearest (even) and subset of exceptions flags like zero and infinity.

5.1 FPU architecture overview

Figure 7 shows the principal connection of FPU to other functional units through interconnection multiplexer network. In order to be conform to the rules of STA concept mentioned before, the result register have to be placed at the output of FPU. Because the implementation of FPU operations is a complex problem and can cause long time delays, further internal pipeline register stages are possible in order to meet the timing constrains. In our design, we placed the second pipeline stage on the inputs of the FPU (dotted line in Figure X). This solution separates FPU's internal logic from interconnection logic and eliminates internal FPU switching activity and power consumption during FPU idle time. The drawback is in the increased FPU latency to 2 clock cycles.

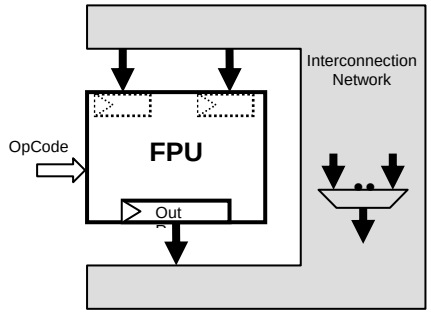

Figure 7: STA based FPU connection to the interconnection network

In Figure 8 the internal block schema of the FPU microarchitecture is shown. It is evident, that in this concept independent parallel arranged modules for realization of arithmetic operations are implemented. As mentioned before, two pipeline stages are implemented on both inputs and outputs of the FPU.

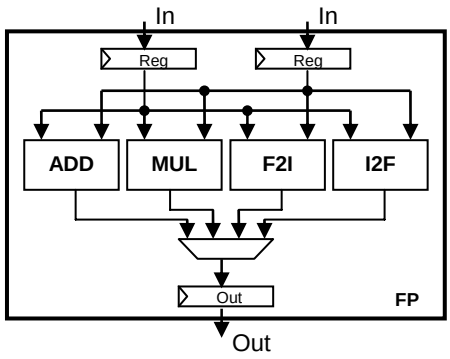

Figure 8: Block schema of FPU

5.2 FPU implementation

The floating point operations are implemented using SYNOPSYS Design Ware (DW) floating point components and SYNOPSYS Module Compiler. Module compiler is a tool for synthesis and optimization of data paths using high level modeling with module compiler language (MCL). Some reasons we decided to use DW FP components and Module compiler are:

- synthesized DW floating point components represent high quality hardware
- DW floating point components are fully parameterizable and support IEEE floating point number format
- accuracy conforms to the IEEE standard
- due to high-level modeling and using IP library drastically reduce design time

6.0 Test and Verification

6.1 Testbench Generation

From the generated SystemC model the modules describing the functional units of the DSP are extracted. A wrapper capable to read a proprietary verification language is used to instantiate the extracted modules. After reading the verification description of the DSP core the executable, generated from wrapper and extracted SystemC module, produces test patterns for the RTL description of the functional unit.

An automatically generated Verilog testbench instantiates the RTL model and reads the test patterns and expected results. The functional unit is stimulated with the test pattern and the results are checked.
The same method is used to test the synthesized functional unit. All checks are invoked automatically on all functional units. This White Box verification methodology is used during the whole RTL implementation phase.

6.2 Automated software-based verification

We used the method of automated software-based verification for in-system verification of the floating point units. Because of generality of the method, we extended this approach also for testing all functional units in STA based DSP architectures.
STA architecture approach is based on the interconnection of functional units (FU) and synchronous transfer of FU values. Structure, functionality and properties of STA architecture are stored in machine description file. An instruction set simulator (ISS) and hardware model of STA architectures are generated from machine description automatically. All information regarding operations and connections to a functional unit are stored in the machine description file. Explaining this concept, we define now the procedure for automated software-based verification:

- suppose - all memories and register files of ISS and HW model are verified and are correct,
- extract the information - from machine description for selected FU concerning operations and connections,
- select - operation and connections of FU to be verified
- generate test patterns - for selected operation and create memory image from these patterns
- generate the assembler program - that reads the test patterns from memory to tested FU directly or through register file, perform selected operation on selected FU and write result to memory
- start testbench -
    - start simulation - load test pattern memory image and assembled program to processor memories of ISS and HW simulator and execute test program
    - read results - from ISS and HW simulator memories after execution
    - compare results - for correctness
- go to the next test

The presented verification method represents extension of the existing verification methodology e.g. black, white or gray box verification. The main advantage of this method is in automation of verification process. Verification environment doesn't depend on architecture modifications. Machine description file and library of test pattern generators represents only input parameters of the verification process.

6.3 Blackbox Verification

Blackbox Verification is used to verify the overall architecture in selective use cases. We are running standard signal processing algorithms on the architecture and compare them with the results of a Matlab™ testbench. After a program is run on the SystemC and RTL Model the memory content of both models is compared. If the memories are equal it is shown that the SystemC and RTL-Model behave equal running this special algorithm. It is not proven that the both architectures are equivalent. Even so, this kind of functional verification of the architecture is capable of testing the architecture for real world applications. This increases the trust to the implementation. Using a VCD-Diff program variations between SystemC

model and RTL-Model are detected and could be visualized using VirSim. Our testcases for the implemented SAMIRA DSP are ranging from FFT, over DCT to a complete implementation of a LMS-algorithm.

6.4 Design for Test

To perform end of line tests, scan chains were inserted into the core. 9 parallel scan paths routed through external I/O pads allow for a fast core tests. TetraMAX was used to generate 11354 test pattern achieving 91.11% test coverage for the full chip. Hence, the additional logic in the toplevel for MBIST and JTAG is not included into the scanchain, the core coverage is approximately around 95%. Hence, most of the die area is consumed by the memories, the MBIST module is most important. The MBIST-module writes the whole memory with test patterns and performs an automatic check after that. This is done three times for each memory module. A single I/O pad indicated, if the MBIST process was successful or not.

7.0 Results

7.1 Core Synthesis

Using our synthesis approach mentioned above, we are able to synthesize our core, including Synopsys DesignWare floating point units, PowerCompiler clock gating and DFTCompiler testability logic within 95 minutes on an Intel Pentium4 machine with 3GHz clock speed and 4GB of RAM. Thereby we can meet all timing constraints.

Technology library	UMC 0.13µm
Maximum clock speed	212 MHz
Logic size	2.4 mm²
Core size (incl. memories)	13.2 mm²
Toplevel size (incl. JTAG, I/O)	14.2 mm²
NAND gate equivalents	460 k

Table 1: Synthesis results

The speed of our design is mainly memory limited. By inserting additional pipelining into some of the functional units, we would be able to double the current clock speed easily if memory speed limitation could be overcome.

7.2 FPU Synthesis

We synthesized the DW floating point components using our target standard cell technology library UMC 0.13um. The synthesis results we achieved for maximum speed criterion are summarized in Table 2. As seen in the table, the most critical part of the FPU from point of view of delay time is, as expected, FP adder. The adder delay of 4.5 ns is lower than the projected maximum delay 5ns (200MHz). In this case, we need not implement any additional internal pipeline stages into FP units. After performing the global synthesis and place&route we achieved the target clock frequency 212MHz, what is over projected frequency.
In the next designs we plan to pipeline floating point units to achieve higher performance and reduce power consumption. Module compiler enables automatic pipelining. For low power architectures and clock gating mechanism it is necessary to control each pipeline stage separately.

FP unit	Area [um x um]	Delay [ns]
Adder	16840	4.5
Multiplier	32713	3.0
Float to integer	9687	1.8
Integer to float	10932	2.3
Total	70172	11.6

Table 2: Synthesis results for floating point units using SYNOPSYS module compiler and Design Ware floating point components. The synthesis criterion was for the maximum speed.

7.3 Architecture Efficiency and Features

In order to show the effectiveness of the architecture proposed in [5], the degree of efficiency of a specific architecture can be classified using the metric defined in [2]. This metric is defined as the portion of a system implementation which performs the actual payload work in comparison to the total implementation effort. This metric may be defined in terms of die area or in terms of power consumption. When defined as area efficiency, it may take the following form:

$$\eta_A = \frac{A_{payload}}{A_{payload} + A_{overhead}}$$

The efficiency of the SAMIRA-DSP is 41.36%. Hence 41.36% of its logic die area are dedicated to adders and multipliers, most of which is consumed by its 17 IEEE single precision floating point units. Figure 9 shows a layout picture of the SAMIRA DSP developed using the presented methodology. We are able to compute a 256-point floating point FFT within 8.5µs at 212Mhz clock rate on this architecture. The performance of other algorithms is going to be determined. The memory size and computational power is dimensioned for DVB-T, hence an OFDM System with 8192-point FFT capability.

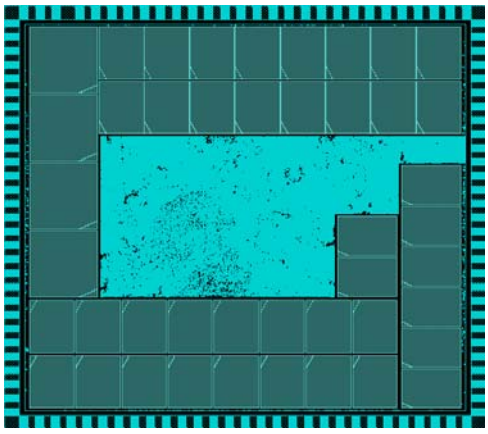

Figure 9: SAMIRA Processor

8.0 Conclusions and Recommendations

STA's low-overhead philosophy, combined with SIMD parallelism and instruction compression allows for the construction of systems maximizing the hardware fraction dedicated to payload. The presented results show that the proposed approach enables the rapid implementation of efficient high-performance, low-power DSPs. Using an automatic

design flow which is utilizing the capabilities of the Synopsys Tool Chain a low power, yet high performance DSP could be developed in a very short time. The SAMIRA-DSP was designed, implemented and synthesized in 6 month by two engineers working full time on the project. Without any consulting support by SYNOPSYS, just using the SOLVENET database and application notes, it was possible to build a 0.13µm Floating-Point DSP.

9.0 Acknowledgements

The Vodafone Chair is member of the distinguished university partner program. The strong research cooperation between Synopsys and the TU-Dresden gives students and researches the chance to work with professional tools within their research activities.

10.0 References

[1] T. Richter, W. Drescher, F. Engel, S. Kobayashi, V. Nikolajevic, M. Weiß, and G. Fettweis. *A platform-based highly parallel digital signal processor.* In Proc. CICC 2001, San Diego, USA, May 2001.

[2] G. Cichon, P. Robelly, H. Seidel, E. Matúš, M. Bronzel, and G. Fettweis. *Synchronous transfer architecture (STA).* In Proc. of Fourth International Workshop on Systems, Architectures, Modeling, and Simulation (SAMOS' 04), pages 126–130, Samos, Greece, July 2004.

[3] M. Weiß and G. Fettweis. *Dynamic code width reduction for VLIW instruction set architectures in digital signal processors.* In 3rd. Int. Workshop in Signal and Image Processing (IWSIP `96), January 1996.

[4] G. Cichon, P. Robelly, H. Seidel, M. Bronzel, and G. Fettweis. *Compiler scheduling for STA-processors.* In Proc. of Fourth International Conference on Parallel Computing in Electrical Engineering (PARELEC'04), Dresden, Germany, September 2004.

[5] H. Seidel, G. Cichon, P. Robelly, E. Matúš, and G. Fettweis. SFB358-A6 demonstrator: HW/SW co-design of a DVB-T receiver. In Proc. of Fourth International Conference on Parallel Computing in Electrical Engineering (PARELEC'04), Dresden, Germany, September 2004.

[6] *Design Budgeting Users Guide*, Synopsys ...

[7] *Design Compiler Reference Manual*, Synopsys ...