Synchronous Transfer Architecture (STA)

Gordon Cichon, P. Robelly, H. Seidel, E. Matúš, M. Bronzel, and Gerhard Fettweis

Mobile Communcations Chair, TU-Dresden
D-01062 Dresden, Germany
cichon@ifn.et.tu-dresden.de

Abstract. This paper presents a novel micro-architecture for high-performance and low-power DSPs. The underlying Synchronous Transfer Architecture (STA) fills the gap between SIMD-DSPs and coarse-grain reconfigurable hardware. STA processors are modeled using a common machine description suitable for both compiler and core generator. The core generator is able to generate models in Lisa, System-C, and VHDL. A special emphasis is placed on the good synthesis of the generated VHDL model.

1 Introduction

This paper presents a novel architecture for digital signal processing that allows effective compiler support. In this architecture, many different algorithms share the available computational resources. In addition, the resulting hardware is suitable for high-performance, low-power applications, as they occur in battery-powered devices. Furthermore, the hardware implementation is streamlined: As much as possible of the hardware models is generated by our tool-chain. Both the compiler and the core generator use the same machine description in order to create an integrated design flow.

2 Related Work

[5, 6] performed evaluations about instruction set effectiveness and utilization. The observation that compilers could hardly utilize the full flexibility offered by the increasingly complex instruction sets, and that code size and performance do not directly correspond, led to the development of RISC architectures [8]. RISC architectures simplify instruction processing, and thus allow much higher clock rates at the cost of a large instruction footprint. Instruction level parallelism adds substantial overhead to instruction processing, e.g. for larger caches, register renaming, branch prediction, dynamic instruction scheduling, register files with many ports, etc. (see [5]).

Digital signal processors (DSP) have to meet real-time constraints in the worst case scenario. Saving die area and power is more important than compiler-friendliness. Therefore, classical DSPs avoid the overhead of dynamic instruction scheduling by supplying CISC instructions for specific high-throughput tasks. These instructions don't consume a lot of memory while exploiting the full degree of parallelism available. A typical example is a MAC (multiply-accumulate) instruction that performs a multiplication, an addition, two address calculations with circular buffering and bit-reverse addressing, and two loads at once, coded with 16 bits of instruction memory.

Many features of general purpose processors have been adopted in the area of DSPs: Texas Instruments [11] based their high-performance DSP platform on a VLIW architecture. Intel [9] presented a superscalar implementation of the ARM RISC core with SIMD extensions. These processors show high performance and can react flexibly to dynamic situations like cache misses and branch misprediction.

With an increasing degree of instruction level parallelism, wiring dominates computational capacity. [4] presents the Transport Triggered Architecture (TTA) to alleviate this problem: It reduces the amount of interconnection resources and number of register file ports while enabling efficient code generation by compilers.

Still, these extensions are not sufficient to satisfy the demand of even more performance-hungry DSP applications. With the rise of FPGAs, high-performance signal processing applications shift towards reconfigurable computing. In order to maintain flexibility, application specific processors (like [12]) propose to identify application specific tasks for implementation in custom functional units.

For further analysis, we refer to the definition of the efficiency metrics Remanence and Scalability in [2]. According to this metric, a VLIW processor can be regarded as a very quickly reconfigurable (low remanence) piece of coarse-grain reconfigurable hardware. We will use this notion to define a fusion of processor and reconfigurable hardware.

3 The Architectural Template of STA

The hardware resources in a processor or reconfigurable hardware architecture can be classified according to their usage: A processor contains a portion dedicated to instruction processing (e.g. instruction memory, instruction decoder, branch prediction) and a portion dedicated to data processing (e.g. data path, ALU, FPU). Reconfigurable hardware dedicates resources to reconfiguration (e.g. storage of configuration, programmable switching matrix) and resources for computation (e.g. adders, multipliers).

Particularly for battery powered applications, it is desirable to spend as little resources as possible to programmability or reconfigurability. A possible way to achieve this goal is SIMD architecture, in which several data processing resources share a single instruction processing resource.

To reinforce the reduction of resources dedicated to reconfigurability, it is desirable to simplify the instruction processing portion as much as possible without affecting the performance of the data processing portion. The predictable execution nature of digital signal processing applications do not require the flexibility of a superscalar architecture. A VLIW or a TTA [4] architecture would be a good choice. This paper proposes a simplification of the TTA architecture:

While VLIW architectures require expensive crossbar switches for the construction of bypasses between the functional units, TTA architectures require local queues for collecting operands and a controller that determines when exactly an operation is to be started. In the predictable execution environment of DSPs, we propose to trigger the execution of instructions explicitly by supplying control signals from the instruction word. We call this architecture Synchronous Transfer Architecture (STA).

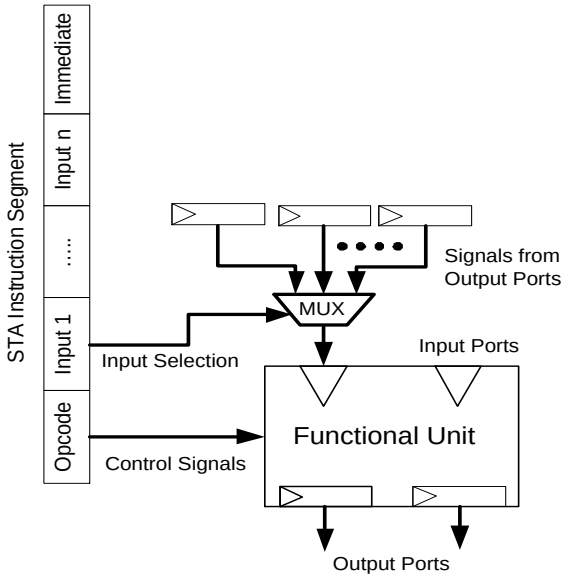

Fig. 1. STA Control Path

Figure 1 shows the architectural template of STA: the design is split up into an arbitrary number of functional units, each with arbitrary input and output ports. To facilitate synthesis and timing analysis, it is required that all output ports are buffered. Each input port is connected to an arbitrary number of output ports. Alternatively, they may originate from immediate fields in the instruction word as well. For each computational resource, a segment of the STA instruction word contains the control signals (opcode) for the functional unit and the multiplexer controls to control the sources of all input ports and associated immediate fields.

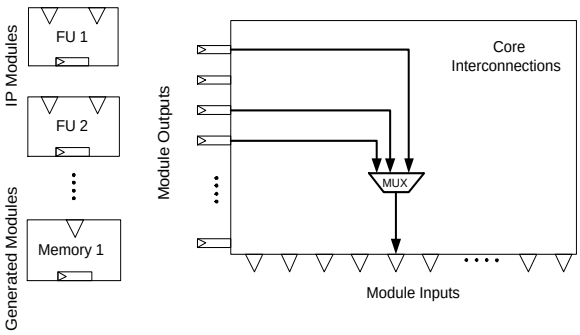

Fig. 2. STA Data Path

Figure 2 shows all input multiplexers together forming a switching matrix between the output and input ports. In total, the system constitutes a synchronous data flow network. The switching matrix may implement arbitrary connections depending on the application, performance, and power-saving requirements. Such an interconnection network can also be constructed in a hierarchical fashion.

The STA architecture poses a new challenge: Storing uncompressed STA instruction segments in memory creates a new bottleneck: It requires a lot of memory and huge bandwidth for instruction fetch. For this reason, the STA instructions need to be compressed. [13] compared different compression strategies. The most efficient ones seem to be 2D run-length encoding schemes. It should be noted that instruction processing of a superscalar processor can also be regarded as a specific decompression technique.

3.1 Compiler Support

In order to apply standard compiler techniques [1, 7] to STA processors, it is required to separate computational resources into the following complementary types:

- **behavior**, i.e. a computational resource that performs some function based on its control signals and its input values. Functional units are not allowed to contain any state excepting pipeline registers.
- **state**, i.e. a resource that stores data. It is not allowed to perform any computation and uses an address input to designate a specific location. If the address comes from an immediate field, it acts like a register file of a RISC processor. Otherwise, it acts like a memory.

Assuming that it is possible to route data between any resources, the Tomasulo algorithm for dynamic scheduling in superscalar processors described in [5] is applicable. With STA processors however, Tomasulo's algorithm can be performed statically by the compiler. The STA interconnection matrix acts like the bypass connections found in classical processors.

It should be noted that the data processing part of any processor can be reinterpreted as an STA processor by assuming an STA-conforming module partition: Each register, register file, or memory turns into an STA-state and each combinatorial circuitry turns into an STA-behavior. E.g., a multiplier containing an accumulator (MAC) can be modeled as a multiplier functional unit connected to a register file containing a single register, the accumulator.

The comparison with the data processing part of a superscalar processor yields a sufficient criterion for the routability of operands, as mentioned above: If there is a designated register file for each data type, and if there is a connection from each port of that type to that register file. This configuration resembles a VLIW processor without a bypass.

3.2 Synthesis-Friendly Implementation of the STA Interconnection Matrix

At the implementation of the STA interconnection matrix, particular care has to be taken of the realization of multiplexers. It turned out to be most advantageous to place these multiplexers directly at the core toplevel without wrapping them into a module:

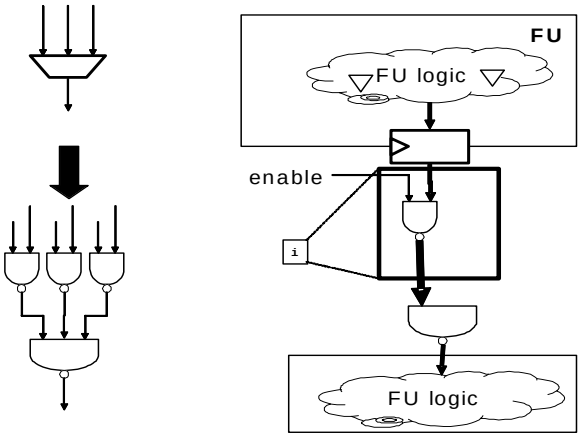

Fig. 3. Implementation of STA Interconnection Matrix

Depending on the library, synthesis can split the multiplexers into NAND gates, as depicted in Figure 3. During the later design phases, the first NAND gate can be placed near the output of the source resource, while placing the second NAND gate near the input port of the target resource. This placement has the advantage of making the the resulting circuitry particularly fast and power-saving:

The first NAND gate can be dimensioned as the driver for the following long wire to the target. It also acts as a power gate for the switched signal. Thus, toggling or recharging of the long wire is prevented in cases when the data are not used at the target.

4 The Integrated Design Flow

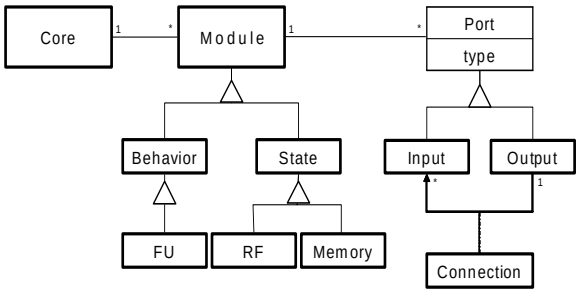

Fig. 4. Machine Description: UML Diagram

Our group created an integrated design flow for the efficient development of STA processors. A formal machine description is used by both compiler and core generator. An UML-diagram overview over the format is shown in Figure 4.

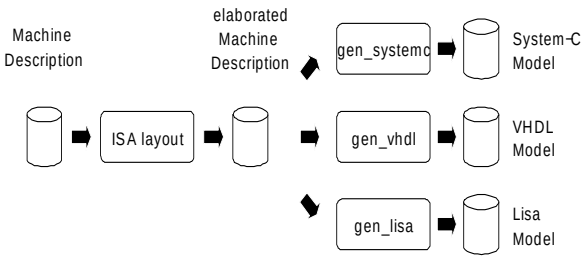

Fig. 5. Core Generator: Design Flow

The design flow of our core generator is depicted in Figure 5. It can be freely used for evaluation purposes at our web site [3]. The web site also provides more detailed documentation and the sample processor model *minimal*.

4.1 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. Finally, a report about the instruction encoding is generated.

4.2 Simulation Models

For simulation and software debugging purposes, a machine model for the Lisa system [14] is generated. The core generator handles the features necessary to support STA and SIMD constructs in Lisa effectively. The generated model contains generated behaviors for all instructions, as well as all code related to the switching matrix. The core generator also generates an assembly preprocessor to support a human-readable entry of parallel programs. Lisa, in turn, generates assembler, linker, and a machine simulator with a debugger.

Alternatively, the core generator can generate a System-C model of the processor.

4.3 Synthesis Models

The core generator also 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 proprietary debug interface for in-circuit debugging 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.

5 Application to SIMD Processors

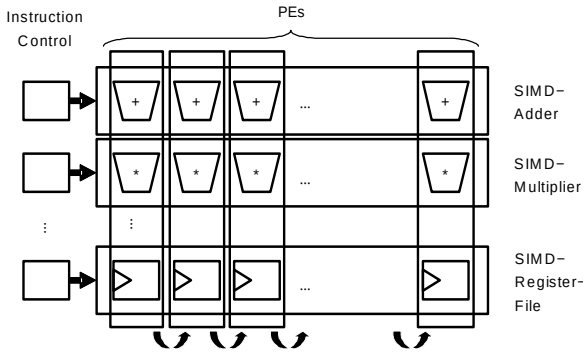

Fig. 6. SIMD Processors

The conceptual view of a SIMD processor from a compiler perspective is different than the perspective from a model more suitable for hardware implementation: A compiler needs to keep track of the resources of a SIMD-architecture that are controlled by a common instruction. Conceptually, such a SIMD-unit performs a vector-computation subject to instruction scheduling. These units are shown as horizontal rectangles in Figure 6, e.g. a SIMD-adder, a SIMD-multiplier, and a SIMD register file. Please note that the term slice refers only to the conceptual representation in Figure 6 and does not imply any restriction on its geometric implementation.

Nevertheless, an efficient hardware implementation can only be derived if modules are prouped into slices or tiles shown as verticals rectangles. Thus, the modules are internally tightly wired, while there is little communication between them. A slice can once be physically placed and routed with a minimal degree of unnecessary freedom, and then be replicated to produce the desired parallelism.

5.1 Slice Recognition

The first step to automatically generate the processor core is to recognize the units that can actually be split into slices. Units that do not perform vector-processing are not subject to the transformation process described in the later steps.

The sliceability of a functional unit can be determined from the machine description by taking into account three indications:

- If the unit inputs or outputs vector data at its ports
- If the behavior of a unit performs vector processing
- Explicit specification by the processor designer

Special care has to be taken of sliceable units that contain scalar ports: Integer types can trivially be reinterpreted as a vectors of single bits. With this technique, the result flags of SIMD-units can be processed by integer scalar units. Another trivial mapping is to generate a broadcast operation for scalar input ports. In other cases, a user-defined mapping has to be specified by the processor designer.

5.2 SIMD-Unit Fission

The second step is to break up the sliceable SIMD-units. This operation is performed by replacing each sliceable unit with a corresponding sliced one. The sliced unit has the same number of input and output ports, while the type of each port is replaced by its corresponding scalarized type. E.g., if the port type used to be *vector of my_type*, it is replaced by *my_type*. The behavior of the sliced unit has to be specified accordingly.

If a unit requires inter-slice communication, additional communication signals have to be inserted into the sliced unit's interface.

5.3 Slice-Module Fusion

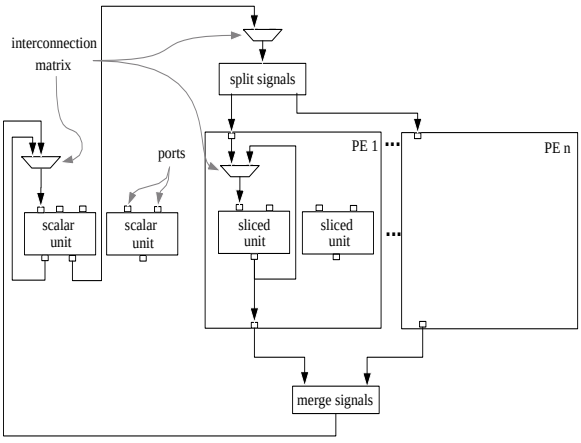

Fig. 7. Interconnection Network with Sliced and Unsliced Functional Units

The third step of the procedure is to group all sliced units into a new module that represents the physical layout entity of a slice. This module has an interface containing

input and output ports for all signals that connect sliced and unsliced units with each other. Additionally it contains ports for slice identification and inter-slice communication.

Within each slice, a synthesis-friendly multiplexer matrix is generated as described above for all locally interconnected signals (see Fig. 7, the multiplexer inside the slice).

5.4 Processor Integration

The fourth and final step builds a processor model containing the replicated slice module, the non-sliceable functional units, and a synthesis friendly interconnection network (see Fig. 7).

Special care needs to be taken of signals that cross the slice boundary:

- **connection from outside to inside**
  For connections to input ports at SIMD-units, only inputs that originate from unsliced units are handled by a second multiplexer outside the slice. As depicted on the upper side of Figure 1, the resulting vector signal is split into its components and connected to the scalarized inputs of the corresponding slices. Inside the slice, the signal is connected to the second multiplexer generated by the fusion step.
- **connection from inside to outside**
  For output signals originating from output ports inside slices which needs to be connected to input ports of unsliced units, all individual scalarized signals originating from inside the slices have to be merged to vector signals (see lower side of Fig. 7).

6 Discussion and Further Work

We presented a novel micro-architecture tailored to high-performance, low-power DSPs. For this new architecture, we are able to generate hardware implementations in an efficient integrated design flow. We used the SIMD paradigm to decrease the amount of resources required to provide programmability.

The performance of an STA processor is determined by the parallelism of its computational resources. It allows to build high-performance processors from libraries of primitive functions. This allows for better resource sharing than other approaches that require to identify specific application specific operations.

In traditional design flows, either two different models of the processor have to be developed and maintained, or a compiler-type model has to be used for synthesis. In contrast to that, our core generator automatically converts a compiler-type machine model into a synthesis-type machine model.

The novel architecture has been tested on a wide variety of systems: A DVB-T receiver, a 802.11/UMTS combo chip, and an OFDM transceiver system. Synthesis and power simulation results show that the proposed architecture is actually suitable for high-performance and low-power DSPs. The results show that programmable solutions are competitive with hardwired solutions. A complete implementation example is presented in [10].

7 Acknowledgments

Thanks to the CATS team and to my sister Caroline Cichon.

This work has been sponsored in part by the German Science Foundation (Deutsche Forschungsgemeinschaft, DFG) within SFB358-A6.

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers. Principles, Techniques, and Tools*. Addison-Wesley, Redding, MA, 1985.
2. P. Benoit, Sassatelli, Torres, Demigny, Robert, and Cambon. Metrics for digital signal processing architectures characterization: Remancence and scalability. In *Proc. of Third International Workshop on Systems, Architectures, Modeling, and Simulation (SAMOS'03)*, pages 102–107, Samos, Greece, July 2003.
3. G. Cichon. MOUSE online core generator. `http://www.radionetworkprocessor.com/gencore.php`.
4. H. Corporaal. *Microprocessor Architecture from VLIW to TTA*. John Wiley & Sons, 1997.
5. J. Hennessy and D. Patterson. *Computer Architecture, a Quantitative Approach*. Morgan Kaufmann Publishers, Inc., San Francisco, CA, 1996.
6. A. Lunde. Empirical evaluation of some features of instruction set processor architectures. *Commun. ACM*, 20(3):143–153, 1977.
7. S. Muchnik. *Advanced compiler design and Implementation*. Morgan Kaufmann Publishers, 1997.
8. D. A. Patterson. Reduced instruction set computers. *Commun. ACM*, 28(1):8–21, 1985.
9. S. Santhanam. StrongArm 110: A 160MHz 32b 0.5W CMOS ARM processor. In *Proceedings for HotChips VIII*, August 1996.
10. H. Seidel, E. Matúš, G. Cichon, P. Robelly, M. Bronzel, and G. Fettweis. An automatically generated core-based implementation of an OFDM communication system. In *Proc. of Fourth International Workshop on Systems, Architectures, Modeling, and Simulation (SAMOS'04)*, Samos, Greece, July 2004.
11. L. Truong. The VelociTI^{TM} architecture of the TMS320C6xxx. In *HotChips IX Symposium*, August 1997.
12. A. Wang, E. Killian, D. Maydan, and C. Rowen. Hardware/software instruction set configurability for system-on-chip processors. In *Proc. DAC 2001*, 2001.
13. M. Weiß and G. P. Fettweis. Dynamic codewidth reduction for VLIW instruction set architectures in digital signal processors. In *3rd. Int. Workshop in Signal and Image Processing (IWSIP '96)*, pages 517–520, Jan. 1996.
14. V. Zivojnovic. LISA - machine description language and generic machine model for HW/SW co-design. In *IEEE Workshop on VLSI Signal Processing*, 1996.