DESIGN OF TRANSPORT TRIGGERED ARCHITECTURE PROCESSOR FOR DISCRETE COSINE TRANSFORM

Jari Heikkinen, Jaakko Sertamo, Tino Rautiainen, and Jarmo Takala

Tampere University of Technology, P.O.B. 553, FIN-33101 Tampere, Finland

ABSTRACT

The trend in programmable architectures for digital signal processing is to move towards high-level language programming and customizable architectures. Several design methodologies have been proposed for designing application-specific instruction-set processors (ASIP) where the hardware resources are tailored according to the requirements of the application. This paper describes the design of an ASIP for a 32-point discrete cosine transform using the tools from the MOVE framework, which is a semi-automatic design methodology for designing processors that utilize the paradigm of transport triggered architecture. Estimations of the designed processor are obtained on program execution, code size, timing and area.

I. INTRODUCTION

The current trend in programmable architectures in the field of digital signal processing (DSP) is to move towards high-level language (HLL) programming and customizable architectures [1]. The reason behind this is the increasing gap between the productivity of designers and increased complexity of DSP applications. By using customizable architectures, the hardware resources of the processor can be tailored according to the requirements of the application. However, it is difficult to find a satisfactory solution from the large design space; several different architecture alternatives must be designed and evaluated. Efficient evaluation requires a set of software tools, such as HLL compiler, assembler, linker, and instruction set simulator, which must be developed for each architecture configuration. However, some design methodologies for customizable processors have been proposed. In [2], several digital signal processor specific features are added to a RISC core to improve the performance in DSP applications. A new compiler for the refined architecture is obtained by using a dedicated code-converter that modifies the assembly code generated by the RISC compiler.

In [3], a development system for designing application-specific instruction-set processors (ASIP) for DSP applications is proposed. Hardware resources and instruction set are described using a specification language. A satisfactory architecture solution is obtained by modifying manually the processor specification. A performance analyzer is used to obtain information on performance and cost.HDL description of the processor design is obtained automatically. Applications described in HLL are compiled using a retargetable compiler. Another similar design methodology proposed in [4] is based on LISA machine description. Software tools, such as assembler, linker, simulator, and debugger front-end, can be generated automatically from a LISA specification describing the hardware resources and operations of the architecture. A standard profiling tool can be used to obtain information of the critical parts of the application, thus the designer can create the LISA model of the architecture. LISA tools generate the HDL of the data path and control part but execution units of the data path must be coded manually. The tools described in [4] do not contain a HLL compiler and thus applications need to be described in assembly language.

Very long instruction word (VLIW) architectures have been widely used in digital signal processing. VLIW architectures are modular; the number of function units (FU) can be increased. There are even VLIW architectures, which support customized, application-specific function units. In VLIW, this may, however, restrict the flexibility, e.g., in Trimedia [5], support for multi-operand instructions, i.e., multi-operand FU, reserves several instruction fields from the VLIW instruction. VLIW architectures have also been criticized for their requirements for read/write ports in the register file [6].

An alternative architecture where the drawbacks of VLIW architecture can be avoided is transport triggered architecture (TTA) [7] where a program describes only the operand transfers between the computational resources. Such a mirrored programming paradigm allows new scheduling and allocation techniques to be used in HLL compilers. TTA concept supports heterogeneous, even multi-operand FUs without restrictions. MOVE framework proposed in [8] is a design environment based on TTA approach. MOVE framework provides a tool assisted architecture exploration; the designer needs only to choose an architecture configuration fulfilling the requirements. A retargetable HLL compiler is used to compile the application onto chosen architecture.

In this paper, the tools of the MOVE framework are used to design an application-specific instruction-set processor for a 32-point discrete cosine transform (DCT). The processor design is taken down to the floorplan level.

II. TRANSPORT TRIGGERED ARCHITECTURE

The bypass complexity of VLIW can be reduced by making the bypass registers visible at architectural level. This way the spilling of bypass values into register file (RF) is made under the program control. The bypass complexity can also be reduced by reducing the number of read and write connections and, therefore, the number of bypass buses. This implies that, besides the operations, also the operand transfers (transports) need to be scheduled at compile-time. Thus, the bypass transports become visible at the architectural level and operations can be hidden. In this model, the data transports trigger the operations implicitly. In principle, the traditional operation triggered programming paradigm is mirrored, hence the name transport triggered architecture [7]. In the TTA programming model, program specifies only the data transports to be performed by the interconnection network. Therefore, only one type of operation is supported: move operation, which performs a data transport from a source to a destination. The number of move operations per instruction is equal to the number of simultaneous transports supported by the interconnection network.

A TTA processor consists of a set of functional units and register files containing general-purpose registers. These units are connected by an interconnection network consisting of buses as illustrated in Fig. 1. Connections to buses are established through input and output sockets; an input socket contains multiplexers feeding operands from the buses into the FUs and an output socket contains demultiplexers placing the FU results into the correct bus. The TTA concept provides flexibility in form of modularity; functional units with standard interface are used as basic building blocks. Therefore, the architecture can be tailored with special function units without a need to change the transport capacity.

III. MOVE FRAMEWORK

MOVE framework is a design environment containing a set of software tools for designing ASIPs [8]. It provides a semi-automatic design process shortening the design-time. MOVE framework exploits the scalability, flexibility, and simplicity of TTA. The design flow consists of three principal components as illustrated in Fig. 2.

The design space explorer searches for a processor configuration, which yields the best cost/performance ratio for a given application. Hardware resources of the processor, such as the number and type of buses, FUs and RFs and their connectivity, are described in an architecture description file. The design space explorer optimizes first the hardware resources. An architecture configuration fulfilling the cost and performance requirements is then chosen by the designer for connectivity optimization where unnecessary connections are removed.

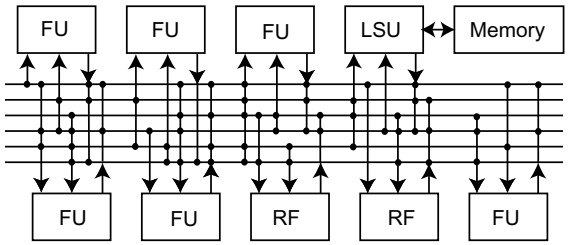

Fig. 1. Principal block diagram of TTA. FU: Function unit. RF: Register file. LSU: Load-store unit. Dots represent socket connections.

The hardware subsystem is responsible for generating the processor floorplan. A processor generator is first used to generate a structural hardware description (VHDL) of the optimized processor configuration obtained from the design space explorer. Commercial tools can then be used to perform logic synthesis, placement, and routing to obtain the layout of the processor.

The software subsystem generates instruction-level parallel code for the chosen processor configuration. It consist of front-end, back-end, and simulator. The front-end, based on GNU gcc, is used to compile the HLL code into sequential code. The back-end schedules the data transports of the sequential code to the available resources described in the architecture configuration file and generates the parallel code. The simulator provides also statistics, e.g., cycle count, instruction count, and hardware resource utilization.

IV. ASIP FOR 32-POINT DCT

The MOVE framework was used to generate an application-specific processor for a 32-point DCT used in an audio coding application. First, the fast algorithm for DCT proposed in [9] was described in C language. The created C code contains five functions, one for each processing column of the signal flow graph of the algorithm. Each function is written totally unrolled, i.e., no iterations are used. This alleviates detection and exploitation of the inherent parallelism of the algorithm. On the other hand, such a code results in larger program code.

In DSP realizations, fractional number representation is often used where the number range is normalized into range $[-1,1)$, i.e., the fixed-point representation contains a single digit for sign followed by the binary point and digits for fractions. In our case, 16-bit data words were used; one bit for sign and 15 bits for magnitude. Due to the fact that ANSI C does not contain predefined data type for such a representation, the normalization needed in multiplications was included into the C description as a shift of 15 bits to the left as follows

```
int a,b,c;
c = (a*b) >> 15;
```

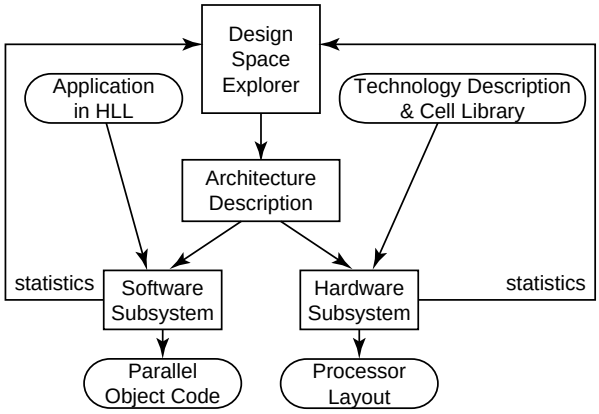

Fig. 2. Principal design flow in MOVE framework.

A. Design Space Exploration

The design space explorer was used to find the optimal architecture configuration for the DCT application. The resource optimization was performed to obtain a set of configurations that are local optimal solutions for a certain cost or performance. A configuration with the best compromize between the cost and performance was selected. The chosen architecture configuration had three load-store units since the used DCT algorithm uses memory frequently to store and load intermediate results during the execution. The current MOVE tools assume that a single memory space is used, thus each load-store unit requires a port to the memory; in our case, a three-ported memory would be required. However, the libraries of the target ASIC technology provided only dual-ported memories and, therefore, the third load-store unit was omitted. This resulted in 26% increase in the number of clock cycles. After the hardware resources were fixed, the connectivity optimization was performed to remove unnecessary connections to reduce area and bus capacitance. An architecture configuration, which had the minimal number of connections and best performance was selected as the configuration to be implemented.

The estimations on area and execution time provided by the design space explorer are based on cost models developed for a 0.7 μm ASIC technology. Since our target technology was a 0.13 μm technology, the architecture configuration obtained from the design space explorer may not be the best one. Due to the lack of cost models for the target technology, we had to compare the performance of different architecture configurations suggested by the design space explorer in terms of clock cycles rather than the execution time, although the optimization is based on execution time. Because of this, the selected architecture configuration does not have the most optimal resources. In order to obtain the most optimal architecture configuration, the cost models would have to be updated to correspond to the 0.13 μm technology. However, the architecture configuration obtained from the design space exlorer is feasible for our case study.

The selected architecture configuration depicted in Fig. 3 has nine 32-bit buses, two load-store units, three ALUs, one multiplier, one shifter, and 32 registers. An I/O unit shown in Fig. 3 was added manually in later design phases. The ALUs perform only two arithmetic operations: addition or subtraction. The shifter is needed to scale the signal levels during the computations for avoiding overflow and to normalize the result of multiplication. The register file configuration had to be chosen manually since the design space explorer cannot modify the internals of the register file. Few experiments were made to find a reasonable configuration for the register file and finally 32 registers were used. In general, a TTA processor also contains compare units but, in this case, the code was unrolled, thus no compare operations were needed. In this sense, the code contains only data flow.

B. Processor Generation

After the processor configuration was found, the structure of the processor can be defined. For this purpose MOVE framework contains a processor generator, which produces a VHDL description of the processor. However, the MOVE processor generator creates architectures with unnecessary features, which are not needed in our design case, e.g., cache and exception support. In addition, currently the MOVE compiler and processor generator contain some incompatibilities, which require manual modifications to the obtained VHDL code.

In order to avoid manual design phases, we developed a simple processor generator, which supports only the basic features and creates a processor, which is compatible with the machine code generated by the MOVE compiler. The developed generator creates VHDL description based on the information obtained from the design space explorer. The generator simply combines manually written leaf cells, such as shifters or multipliers, produces the necessary control logic and wires everything together at the top level. The generator creates also the scripts for logic synthesis, a testbench, and test vectors for RTL verification. The test vectors realize the entire application.

The new processor generator was run and the RTL-verification was performed using the automatically created testbench and test vectors. The processor was synthesized with the automatically generated synthesis script onto a 0.13 μm CMOS standard cell technology with 5 levels of metal (copper wires). Starting point for the back-end flow was hierarchical Verilog netlist from the synthesis tool with about 18,000 standard cells and 7 macro cells (memories), one for the data memory and the rest six for the instruction memory. Total gate count of the standard cell area was about 56,000 and of the macro cells 34,000.

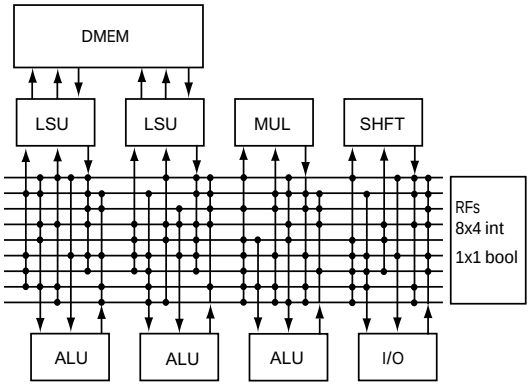

Fig. 3. Optimized architecture configuration for 32-point DCT.

The obtained netlist, cell library, and hard macros were imported to a back-end tool and floorplanning tasks were performed. The area of the processor core was estimated based on the number of standard cells in the design and utilization limit of the technology. Core aspect ratio was modified together with macro placement in order to optimize the macro placement and the shape of standard cell area. After these, I/O placement and automatized power routing estimation were executed.

Flat approach was used for placement and routing. Connectivity based placement was used without timing constraints and without any additional physical constraints such as critical net weighting, cell grouping, placement blockages, or routing blockages. The memory blocks have been implemented with only four metal layers, thus one metal layer could be used for routing over the macros. Some placement and routing iterations, including core size modifications, were executed in order to find an optimal routable area for standard cells. Floorplan of the design is illustrated in Fig. 4. Macro cells forming the instruction memory are placed on the left and right edges of the core. Data memory is placed at the bottom and I/O on top of the core. The size of the data memory is 512 bytes, and the size of the instruction memory is 24576 bytes.

TABLE I
STATISTICS OF 32-POINT DCT PROCESSOR DESIGN.

Core size (width x height) [μm]	850.00 x 928.20
Chip size (width x height) [μm]	920.55 x 998.75
Clock cycles	538
Data transports	2722
Instruction width [bits]	180
Instructions	559
Code size [bytes]	12578
Code efficiency [%]	54.1

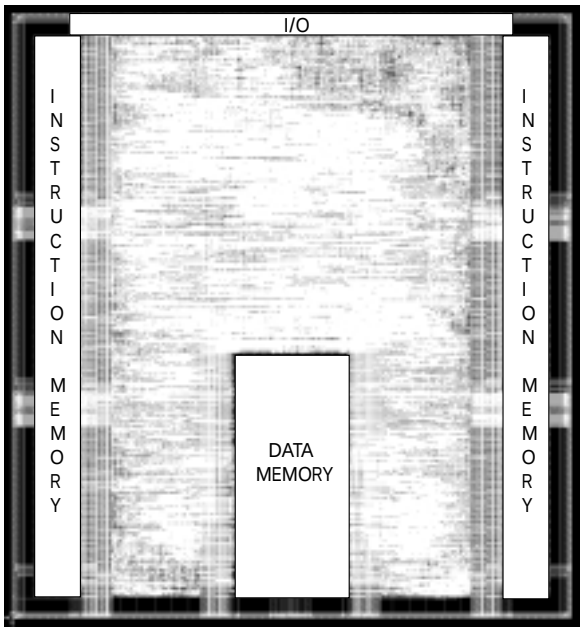

Fig. 4. Floorplan of ASIP designed for 32-point DCT application.

Acceptable level of routing congestion was achieved with the core and chip dimensions described in Table I and with standard cell utilization of 78%. Clock tree synthesis was done in order to distribute the clock signal to flip-flops with minimum skew. The design has only one clock domain with total number of 140 clock distribution buffers and the number of loads is 3283. Estimated maximum skew value is 0.047 ns after the clock tree synthesis and 0.032 ns after pre-layout gate sizing. The maximum clock frequency limited by switching for this clock domain is about 700 MHz. These numbers are pre-layout estimations without exact statistics from static timing analysis or gate level simulations.

In this design case, only the first phase of the routing tasks was performed; neither physical violation checks nor final routing optimization were performed. However, the first routing phase results in quite accurate estimations; in general, for simple floorplans without special physical constraints, the difference is about 5% compared to final post layout results. After the back-end flow, the design was verified at gate-level with clock frequency of 250 MHz using back-annotated timing information from the back-end tool. Clock frequency of 250MHz was chosen in the logic synthesis phase. The attainable clock frequency is limited by the memory access time; 350 MHz could have been achieved with the used technology.

C. Simulation

The MOVE software subsystem was used to compile the DCT application into an executable binary code for

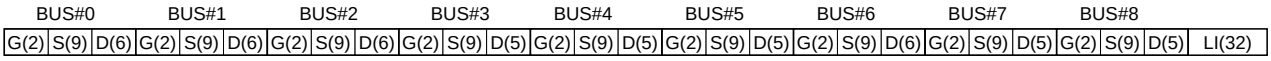

Fig. 5. Structure of instruction word. G: Guard field. S: Source Field. D: Destination field. LI: Long immediate field. (x): x-bit field.

the generated processor architecture. The MOVE instruction set simulator was then run to obtain statistics of the program execution and code size. The obtained results are shown in Table I. The code size is fairly large due to the long instruction word having dedicated slots to define data transports on each bus as depicted in Fig. 5. Each move slot contains three fields. The guard field specifies the guard value that defines if the data transport on the bus is executed or squashed. The source field specifies the address of a socket that writes data on the bus. The destination field specifies the address of a socket that reads the data from the bus. In addition to move slots, instruction word contains also a dedicated field used to specify a long immediate value. In addition to long instruction word, the code efficiency shows that approximately 46% of the data transport are empty transports (NOPs) increasing the number of instructions and thus the code size. A method to improve code density of processors designed with MOVE framework by applying entropy encoding to compress instructions has been discussed in [10]. Evaluations show code compression ratios of about 0.6.

The simulator provided also information about the hardware resource utilization. The utilizations of the the two load-store units are 82% and 41% indicating that the three load-store units, as suggested by the design space explorer, would have been a better configuration than having only two of them. The utilizations of the ALUs are 48%, 37%, and 12% indicating that three ALUs are sufficient.

The utilization of the shifter is quite high (27%) implying that the shift capability should be incorporated into the multiplier in this kind of a DSP application. However, the shifter would still be needed to perform the scaling of intermediate signal levels for avoiding overflows. The utilization of multiplier is fairly low (15%) due to the small number of multiplications in the used DCT algorithm.

V. CONCLUSIONS

In this paper, design of an application-specific instruction-set processor for a 32-point discrete cosine transform using the tools from the MOVE framework was described. The processor design was taken down to the floorplan level and estimations on area and timing were obtained. Due to compatibility problems of the MOVE tools, the generated design is not the most optimal solution for the DCT application. A guard unit controlling the conditional execution was included in the design although it is not used since the application was written totally unrolled, i.e., no conditional structures were used. Removal of the guard unit would have required changes in the hardware subsystem. Since the guards in this design case are always true, the guard fields could have been removed from the MOVE instruction. However, this would have caused modifications to the MOVE compiler. The multiplier of the designed processor should include shifter required to normalize the results of multiplication. This would have made the data buses available for other data transports. Furthermore, since the 32-bit results from the multiplier would no longer be transported to the shifter, the bus width could have been reduced.

As discussed above, to obtain a more optimal processor design for the application, the tools of the MOVE framework would have had to be modified manually. Since our focus was to evaluate the current status of the MOVE tools to design an application-specific processor such modifications were not done.

REFERENCES

[1] J. M. Rabaey, W. Gass, R. Brodersen, T. Nishitani, and T. Chen, "VLSI design and implementation fuels the signal-processing revolution," *IEEE Signal Processing Mag.*, vol. 15, no. 1, pp. 22–37, Jan. 1998.

[2] J. Kang, J. Lee, and W. Sung, "A compiler-friendly RISC-based digital signal processor synthesis and performance evaluation," *Journal of VLSI Signal Processing*, vol. 27, no. 3, pp. 297–312, 2001.

[3] J-H. Yang, B-W. Kim, S-J. Nam, Y-S. Kwon, D-H. Lee, J-Y. Lee, C-S. Hwang, Y-H. Lee, S-H. Hwang, I-C. Park, and C-M Kyung, "MetaCore: an application-specific programmable DSP development system," *IEEE Trans. VLSI Syst.*, vol. 8, no. 2, pp. 173–183, 2000.

[4] A. Hoffmann, T. Kogel, A. Nohl, G. Braun, O. Schliebusch, O. Wahlen, A. Wieferink, and H. Meyr, "A novel methodology for the design of application-specific instruction-set processors (ASIPs) using a machine description language," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 11, pp. 1338–1354, 2001.

[5] J. T. J. van Eijndhoven, F. W. Sijstermans, K. A. Vissers, E. J. D. Pol, M. I. A. Tromp, P. Struik, R. H. J. Bloks, P. van der Wolf, A. D. Pimentel, and H. P. E. Vranken, "TriMedia CPU64 architecture," in *Proc. IEEE Int. Conf. Computer Design*, Austin, TX, U.S.A., Oct. 10–13 1999, pp. 586–592.

[6] R. P. Colwell, R. P. Nix, J. J. O'Connel, D. B. Papworth, and P. K. Rodman, "A VLIW architecture for a trace scheduling compiler," *IEEE Trans. Comput.*, vol. 37, no. 8, pp. 967–679, Aug. 1988.

[7] H. Corporaal, *Microprocessor Architectures: From VLIW to TTA*, John Wiley & Sons, Chichester, UK, 1997.

[8] H. Corporaal and M. Arnold, "Using transport triggered architectures for embedded processor design," *Integrated Computer-Aided Engineering*, vol. 5, no. 1, pp. 19–38, 1998.

[9] J. Takala, D. Akopian, J. Astola, and J. Saarinen, "Constant geometry algorithm for discrete cosine transform," *IEEE Trans. Signal Processing*, vol. 48, no. 6, pp. 1840–1843, June 2000.

[10] J. Heikkinen, J. Takala, and J. Sertamo, "Code compression on transport triggered architectures," accepted to *Int. Workshop on System-on-Chip for Real-Time Applications*, Banff, Canada, July 6–7 2002.