A Macro-Block Based Methodology for ASIP Core Design

Yann Bajot, Habib Mehrez
Laboratoire LIP6/ASIM UPMC Paris VI, 4 Place Jussieu, 75252 Paris Cedex 05

ABSTRACT

This paper presents a methodology for ASIP Core design, based on macro-blocks generators and retargetable software tools. This methodology includes design space exploration, performance profiling and portable synthesis of the core. An example of a flexible DSP core is presented.

1. INTRODUCTION

In the last decade, the architecture of DSP processors has taken many new directions. The huge success of emerging markets like multimedia and wireless communication have brought DSP architectures to be used in high-volume consumer products, making them very popular. The ever-increasing complexity of multimedia embedded applications imposes several constraints such as high computation power, low-power conception and reduced silicon size. This has led designers to find new architectural solutions. One of these is Application-Specific Instruction-set Processor (ASIP) [1]. These processors include a datapath and instruction-set optimized for the targeted applications, achieving the same or upper performances than a general DSP at a lower cost (silicon, power). ASIPs are also a good alternative to hard-wired blocks, since they provide more flexibility for the algorithm implementation. Within the context of System-On-Chip, ASIP cores provide efficient, flexible and low-cost computation engines. We present here a methodology for the design of these cores, based on macro-block generators and retargetable software tools. This methodology aims to provide shorter design cycle and technological portability.

2. DESIGN SPACE EXPLORATION

Our goal is to determine an optimized datapath with its associated instruction-set in order to minimize power consumption, silicon area, and cycle time. It means choosing among a very large scope of architectural solutions, which we describe in the following sections.

2.1 Functional Units

We have to define the types and number of the datapath functional units. They can be general (ALU, Shifter, Multiplier) or specialized (Bit Manipulation Unit, Viterbi decoder). The later can be considered as hardware accelerators performing complex computations, providing greater performance but at a higher hardware cost. Increasing number of units enables greater instruction-level parallelism (ILP) but results in a wider instruction word.

2.2 Register Set

This can be homogenous (multi-ports register file), heterogeneous (local registers) , or hybrid (both). The homogeneous register set constitues a better compiler target but is more expensive regarding silicon area, power dissipation and cycle time. The heterogeneous one enables heavy instruction encoding, reducing program memory size.

2.3 Data Width

In commercial fixed-point DSPs, data width is 16, 24 or 32 bits. Because we focus on a few particular applications, the optimized width value corresponding to maximum dynamic of the signal can be obtained by simulation. The width of the datapath can be tailored to the exact data dynamic thus reducing hardware size.

2.4 Connectivity

The connectivity represents all the possible links between functional units, registers, and memory ports. A strong connectivity is a good key for compilers, but leads to higher cost, longer cycle time and wide instruction word.

2.5 Memory structure

Most of the existing DSPs use a Harvard architecture, with separated program and data busses. The number of data busses defines our memory bandwidth. Multiple data busses improve parallelism but are more expensive. Each memory port associates an address generator, featuring several addressing modes: general (immediate, direct, indirect) and specialized (circular, bit-reverse).

2.6 Instruction Encoding

Instruction encoding strategy can range from full parallel (VLIW-style) to deeply encoded. VLIW is more compiler-friendly and increases ILP, thus enabling better use of intrinsic application parallelism, but is more expensive because of its longer instruction width.

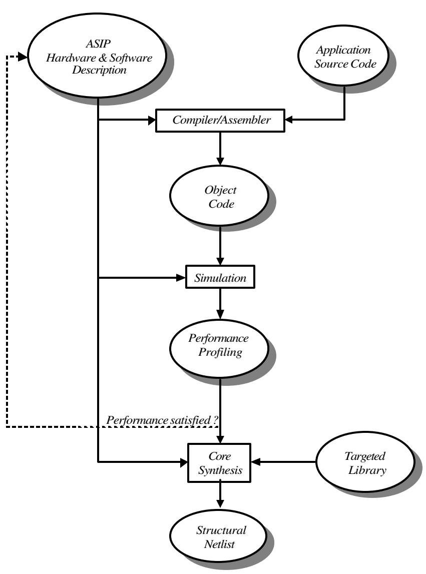

Figure 1. ASIP Design Flow

3 DESIGN METHODOLOGY

Our methodology aims to provide a convenient way for the designer to define the optimized ASIP architecture for the targeted application. The core design flow is illustrated on figure 1.

This method is based on hardware/software co-simulation. The ASIP hardware & software description defines the hardware structure of the core (datapath, memory, connectivity…), its associated instruction-set and encoding strategy.

The application is coded in a high-level language and compiled. The Assembler/Compiler relies on the ASIP hardware and software description, particularly the instruction-set and encoding strategy to create the object code. We assume that this compiler is able to exploit parallelism of the source code and hardware resources to produce a good quality code. Next, this code is simulated on the core simulator to evaluate performances. Interesting information for the designer are: execution time (number of cycles), average utilization of hardware resources (memory ports, functional units, registers, connections) and instructions , program and data memory size. Considering this, the designer can choose either to :

ÿ synthesize the core with current parameters described in the ASIP description to obtain a structural netlist.

ÿ or modify the ASIP description by changing some parameters (more functional units, different encoding, more instructions…). The designer then resimulates the new description to determine the impact of these changes on performance. The optimized design should be able to respect execution time constraints while minimizing hardware cost.

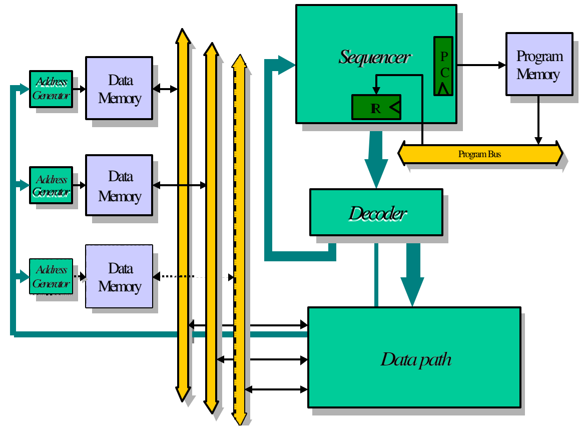

Figure 2. ASIP Core Model

This methodology relies on macro-block generators. The general ASIP architecture illustrated on figure 2 consists of a set of parameterizable macro-blocks. This architecture includes :

- an instruction sequencer with associated program memory and bus,
- a variable number of data memories with associated address generators and busses to provide flexible memory bandwidth,
- an instruction decoder,
- and finally the datapath which can be considered as a set of lower-level macro-blocks such as functional units, register files, etc.

For this architecture, exploration of the design space means defining macro-blocks parameters. Thus finding the optimal architecture for the core is equivalent to finding the best set of parameters for each macro-block.

4 CORE SYNTHESIS

The core synthesis relies on the macro-block generator concept implemented in GenOptim [2, 3], a VLSI CAD environment allowing a very fast implementation of architecture ideas on different technologies and different commercial sets of CAD tools. It furnishes the designer a set of high level C functions allowing to write quickly the architecture, the test patterns and the placement. During the mapping on the targeted technology placement and timing optimization operations are carried out by GenOptim, see figure 3.

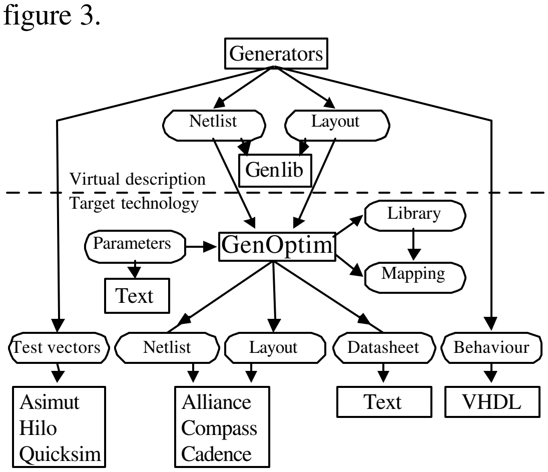

Figure 3. GenOptim description.

Generators created with this tool can be mapped on any standard cell library as well as FPGAs. This shows to be of great interest in regard of the new trends in design with IP blocks.

We present here the different macro-blocks involved in our ASIP core model and their associated parameters. These blocks are implemented in the CAD design environment GenOptim.

4.1 Address Generator

Several addressing modes are furnished: immediate, direct, indirect, indirect with post-increment, bit-reverse, and circular buffers using modulo-addressing. Its structural parameters are address width, max number of circular buffers, and max length of buffers.

4.2 Instruction Sequencer

The sequencer computes the next program memory address used to fetch the next instruction. It provides general control flow modes such as sequential access, conditional/unconditional JUMP and CALL, and finally Branch to Interrupt. It also features some more sophisticated DSP-oriented modes such as hardware loops (enabling zero delay branching in software loops) and conditional instructions. These modes allow best dynamic performances, thus reducing execution time of the application. Program address width, number of allowed nested interrupts, loops and branches are configurable.

4.3 Instruction decoder

It has to decode the instruction fetched by the sequencer and generate control signal towards other blocks of the core. Instruction-set encoding strategy determines the combinatory equations to be implemented.

The generator takes as a parameter a description of this encoding, found in the ASIP hardware and software description mentioned in section 2.

4.4 Datapath

The datapath can be seen as a high-level macro-block built with several lower-level blocks. Figure 4 presents the datapath model implemented in our ASIP core.

- The Register Set includes all registers and register files of the datapath. Its parameters are data width, number of ports In/Out, and register granularity (from one big register file to multiple stand-alone registers).

- Data-switches represent connectivity between the Register Set and the Functional units. This connectivity can vary from a full crossbar model (any in-port can be routed to any out-port) to single wires connecting one in-port to one out-port. Its parameters are data width, number of ports In/Out and a description of the connectivity. The generator then generates the net of switchers needed to implement the desired connectivity.
- Finally, the last low-level blocks are the functional units. We find all general units used in DSPs: ALU, Shifter, Multiplier, Saturation and Rounding operators, and eventually specialized units such as Bit-Manipulation units and Viterbi Decoders.

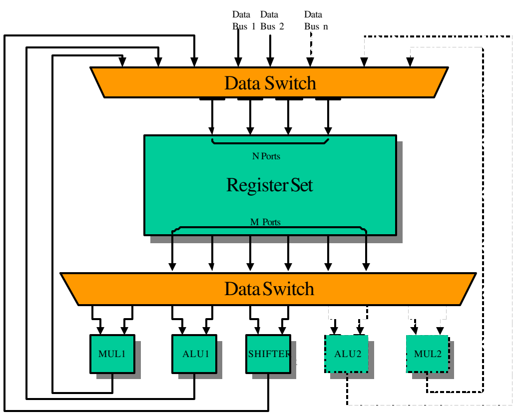

Figure 4. Datapath Model

5. SOFTWARE REQUIREMENTS

Implementation of our methodology requires several particular software tools: a compiler, an assembler and an instruction-set simulator (ISS).

The common aspect for all these tools is retargetability. In fact, they have to adapt to any architecture given in the ASIP hardware and software description. To do this, we use several software tools, as described on figure 5.

Concerning retargetable compilers, several research projects exist, like SPAM or CHESS projects [4]. They aim to provide optimizing compilers able to exploit hardware and instruction-set characteristics of targeted processors, to produce high quality code. Interfacing with such compilers assumes the ASIP hardware and software description is translated in a processor description language known by these compilers (e.g nML for CHESS compiler [5]).

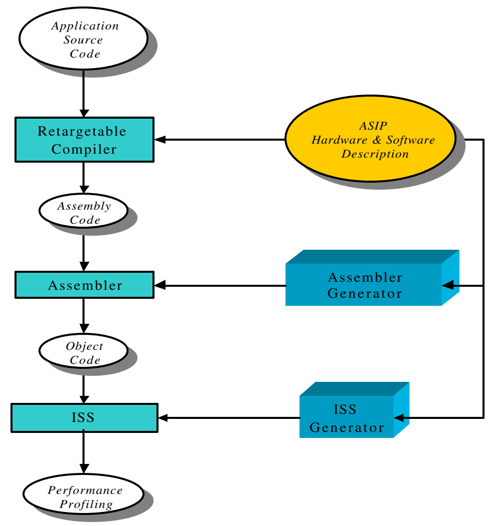

Figure 5. Software Tools

The assembler and simulator are produced by software generators. These generators take into account the current ASIP description to generate the suitable tools. The assembler generator needs a complete description of the instruction-set and some hardware information: different instruction word formats, field encoding values, syntax (name of registers and functional units), and resources constraints (number of functional units, registers, register file and memory ports, connectivity). The ISS generator needs information about the hardware structure of the core and decoding strategy. Its structure is the same as ASIP core model presented in figures 2 and 3. Each macro-block (sequencer, decoder, datapath,) has a C-model taking the same parameters as the netlist generator model. The generator uses these parameters to create the suitable simulator.

The generated simulator allows to run object code on the customized core and gives information about core performances, hardware and instruction-set utilization. It also permits to test correct behavior of the code compared to the desired implemented application. To do this, all C-models are bit-true and cycle-precise. Bit-true aspect permits numerical validation of the code, checking the good adequation between chosen data width and data dynamics. Cycle-precise aspect enables an accurate estimation on dynamic behavior of the code and final execution time. The whole information required by compiler, assembler and simulator generators is provided by the ASIP hardware and software description, presented in section 3.

6. FIRST IMPLEMENTATION: A CONFIGURABLE DSP CORE

Our current works focus on the realization of a configurable DSP core. Figure 6 shows its datapath architecture: it is based on a classic two multiplier-accumulators structure which ensures a high computing power.

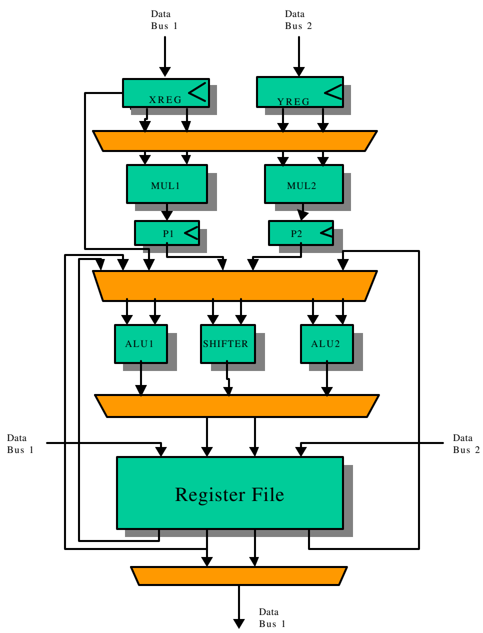

Figure 6. DSP Core Datapath

Data are provided via two independent data busses and can be stored in two register files Xreg and Yreg. Functional units include two general ALUs, two multipliers and a shifter. A larger register file is used for data temporary storing and write accesses to memory. Data traffic between functional units, registers and data busses is managed by three data-switches. The core provides two Address Generators, one for each memory port, and an instruction sequencer based on a 3 stage pipeline (Fetch, Decode, Execute). Data width, address width, register file size, maximum number of nested loops and interrupts are configurable. This core can not be considered as an ASIP, because of its classic architecture, but is used to validate our macro-block generators and software tools.

7. CONCLUSION

In this paper, we have presented a methodology for ASIP core design. It is based on an ASIP core model partitioned in macro-blocks. Each block is configurable and provides a C-model for simulation and a generator allowing mapping on any technological library. It also relies on retargetable software tools taking into account the ASIP hardware structure and instruction-set.

Current works deal with concurrent development of high-level macro-blocks generators and retargetable assembler/simulator. For the time being, applications are hand-coded to obtain efficient assembler code. An interface with a retargetable compiler would avoid this tedious task and would accelerate the path from application source code to architecture performance estimation given by the simulator. This improves the design flow. However, the methodology efficiency is conditioned by the compiler capability to produce a high quality code, maximizing utilization of ASIP hardware resources and instruction-set.

References

[1] Goossens G, & al., Embedded Software in Real-Time Signal Processing Systems: Application and Architecture Trends, Proceedings of the IEEE, Vol. 85, no.3, March 1997

[2] Houelle, A., GenOptim: un environnement d'aide à la conception de générateurs de circuits portables optimisés en performance et en surface, Thèse de doctorat de l'université Paris VI, 1997.

[3] Vaucher, N., Méthodologie de conception d'architectures VLSI génériques appliquée au traitement numérique, Thèse de doctorat de l'université Paris VI, 1997.

[4] D. Lanneer, & al, CHESS: Retargetable Code Generation For Embedded DSP Processors. In P. Marwedel and G. Goossens, editors, Code Generation for Embedded Processors, chapter 5, pages 85-102. Kluwer Academics Publishers, Boston, Massachussets, 1995.

[5] A. Fauth & al, Describing Instruction Sets Using nML. Technical report, Technische Universität Berlin and IMEC, Berlin/Leuven, 1995.