$\mathfrak{NoGap}^{CL}$: A Flexible Common Language for Processor Hardware Description

Wenbiao Zhou
Department of EE
Linköping university
Linköping
zhou@isy.liu.se

Per Karlström
Department of EE
Linköping university
Linköping
perk@isy.liu.se

Dake Liu
Department of EE
Linköping university
Linköping
dake@isy.liu.se

***Abstract*—Flexible Application Specific Instruction set Processors (ASIP) are starting to replace monolithic ASICs in a wide variety of fields. However the construction of an ASIP is today associated with a substantial design effort. $\mathfrak{NoGap}$ (Novel Generator of Micro Architecture and Processor) is a tool for ASIP designs, utilizing hardware multiplexed data paths. One of the main advantages of $\mathfrak{NoGap}$ compared to other EDA tools for processor design, is that $\mathfrak{NoGap}$ impose few limits on the architecture and thus design freedom. $\mathfrak{NoGap}$ does not assume a fixed processor template and is not a data flow synthesizer. To reach this flexibility $\mathfrak{NoGap}$ makes heavy use of the compositional design principle. This paper describe $\mathfrak{NoGap}^{CL}$, a flexible common language for processor hardware description. A RISC processor using $\mathfrak{NoGap}^{CL}$ has been constructed with $\mathfrak{NoGap}$ in less than a working day and synthesized to an FPGA. With no FPGA specific optimizations this processor met timing closure at 178MHz in a Virtex-4 LX80 speedgrade 12.**

***Index Terms*—ASIP, ADL, CAD**

I. Introduction

The design and implementation of a new processor is usually the result of a substantial design effort. There are a number of different software tools (some of them mentioned in Section II) that relax the design effort in one way or another. However all these tools force the designer into a predefined architecture template. This limitation in design flexibility often makes designers of novel ASIP processors and programmable accelerators revert back to an HDL language, e.g. Verilog or VHDL. HDL languages offer full design flexibility at the register transfer level, but the flexibility comes at the cost of increased design complexity. All details, e.g. register forwarding and/or pipeline control, have to be handled manually.

The term "EDA tool" is often used for design automation tools, however this term includes a vast range of tools in various disciplines. This paper focuses on the class of tools used for higher level modeling of hardware architectures, thus, the description used in these tools are often called an Architecture Description Language (ADL), therefore these kinds of design tools will be referred to as "ADL tools" in this paper.

Since an ADL tool presents an abstract view of the design, some details have to be hidden and there must be some priori assumptions about the device being designed. This fragments the ADL tools in different directions with different areas of use. Currently most ADL tools either describe systems at a higher level of abstraction, modeling transactions and interaction of modules at the system wide scope, or their aim has been to ease processor design with supporting compilers and simulators. In both of these ADL tool classes there are a number of mature and well performing products which can be used with great success if the system being designed fits into the tools a priori assumptions. However, the risk of using an ADL tool is that the final design is just a product of what the ADL tool supported, than the innovative product the designer had in mind.

$\mathfrak{NoGap}$ does not aim to make processor construction easy for the masses.$\mathfrak{NoGap}$ aims to give support to experienced designers, i.e., the designer knows what he does and uses $\mathfrak{NoGap}$ to support him/her with tedious and erroneous tasks.

Many key concepts used in $\mathfrak{NoGap}$ is based on the work presented in [1], [2].

II. Related Work

A number of tools such as LISA [3], EXPRESSION [4], nML [5], MIMOLA [6], ArchC [7], and ASIP Meister [8], are tools that support processor design. All of these tools however force a designer into a predefined template architecture. On the other end of the spectrum of design tools are HDL languages such as Verilog, VHDL or SystemC [9], These tools however requires manual handling of all miniscule details of an RTL design. $\mathfrak{NoGap}$ offers an unique trade off between these two extremes. No template design is assumed but support is given for managing details regarding pipelined instruction controlled architectures. The advantage and disadvantage of these tools can be referenced in [1].

III. $\mathfrak{NoGap}$ Overview

$\mathfrak{NoGap}$ consists of a number of components. The central component in $\mathfrak{NoGap}$ is the $\mathfrak{NoGap}$ Common Description ($\mathfrak{NoGap}^{CD}$). $\mathfrak{NoGap}^{CD}$ in turn consists of three parts; the Micro Architecture Structure Expression ($\mathfrak{Mase}$), the Micro Architecture Generation Essentials ($\mathfrak{Mage}$), and the Control Architecture STructure LanguagE ($\mathfrak{Castle}$).

The Mase description is an annotated graph representing connections between functional units. The Mage description is an Abstract Syntax Tree (AST) representation of leaf modules containing the actual functionality. And the Castle description contains directives for how instruction decoders should be generated. Even though the NoGap^{CD} can be constructed directly with an C++ API, this is not the intended way to construct a new micro architecture. The intended flow is to use a front end tool called a facet and then let the facet generate the NoGap^{CD}. NoGap has a default facet for this, implemented as a language called the NoGap Common Language (NoGap^{CL}). The final stage in the NoGap flow is to produce something useful, this is done by implementing a tool reading the NoGap^{CD} and from that generate a useful output. These output tools are called spawners in the NoGap terminology. A spawner can, for example, be an RTL code generator or a cycle accurate simulator generator. The NoGap framework principle is depicted in Figure 1, where a number of possible facets generate a NoGap^{CD} and from that a number of spawners generate the various tools needed. This approach will allow for a modularized approach with the possibility to reuse code over different projects. For example, a cycle accurate simulator spawner could have been implemented for an earlier project but a new facet would ease the design effort for a new project. In this case only a new facet has to be implemented and time/money can be saved by reusing the old spawner. Note that for example, NoGap does not

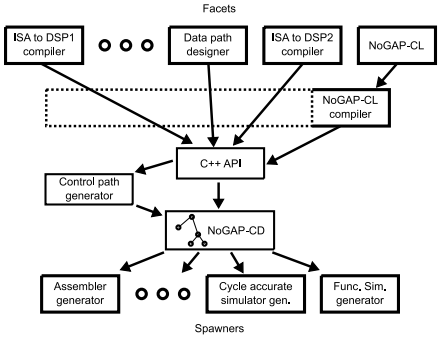

Fig. 1: NoGap principle

treat storage elements in any special way. A register will be treated as any other Functional Unit (FU). To inform NoGap about the possible source and destination operands, the Castle description is used to mark certain nodes as data sources and other nodes as data destinations. This is all according to the powerful yet flexible design by composition principle prevalent in NoGap.

IV. NoGap^{CL} Organization

NoGap^{CL} is the default NoGap facet used to construct the Mase, Mage and Castle descriptions. It is also a flexible common language for processor hardware description. The main features of NoGap^{CL} are as follows:

1) Less micromanagement needed for control path construction.
2) No processor template restriction, providing more freedom for the designer.
3) Support of dynamic port sizes.
4) Automatic decoder generation.
5) Pipeline stages can be adjusted easily, different pipelines can be defined for different operations.

A. Sample Architecture of Processor

A RISC like processor with Harvard architecture will serve as an example. The processor called NoGapOne handles normal operation flow instructions such as jump, call, and return. It has a 32×32 bit register file, a data memory with load store from register a 32 bit ALU and a 16×16 bit multiplier. Its general architecture can be seen in Figure 2, note the two pipeline configurations after the `OF` stage, the normal pipeline has white background, the longer pipeline has gray background. NoGapOne was designed in less than a working day. The instruction set of NoGapOne is a bit limited but supports the following instructions: move, load, store, common alu operations, multiply, multiply and accumulate (add/sub), jump, call, and return. The multiply accumulate instruction using the long pipeline (gray background) utilizes the `alu` in pipeline stage EX2 to perform the post multiplier addition or subtraction.

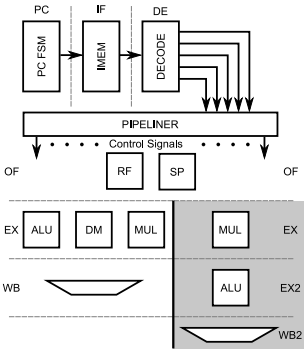

Fig. 2: NoGapOne

B. Mage FU Specification

The Mage FU specification is similar to HDL languages such as Verilog and VHDL. The main differences from Verilog and VHDL are that no combinatorial loops are allowed, ports can have dynamic sizing, and implicit clock and reset inputs. Combinatorial loops can be checked for with the techniques for SSP generation as discussed in [1]. An Mage FU does not need a clock or reset input, these are implicitly assumed for all Mage FUs using the `cycle` construct. All logic takes place within either a `cycle` or a `comb` block. The `comb` block describes combinatorial logic and the `cycle` block describes clocked logic. If a Mage FU should be controllable with operations, clauses in the Mage FU description needs to be named. NoGap can dynamically select the clasuse name if needed.

C. Mase FU Specification

A Mase FU specification is written in much the same way as a Mage FU, but with many other language constructs. A Mase FU specification can include: phase statements, pipeline constructs, and/or operation constructs. Using phases, the operation's pipeline timing can be computed by NoGap. Assignments written after a phase statement take place in the timing of that phase. The operation construct defines the operations of the processor, and also defines the connection relationship between different Mage FUs.

D. Sequencer Specification

The sequencer in NoGap is used to generate the instruction streams, the data path itself should not know about how a sequencer operates apart from being able to stall the instruction stream if necessary. The sequencer description contains information about how source and destination operands are generated, if and how jumps are handled. Interrupt handling is also specified in the sequencer specification.

E. Dynamical Port Sizes

NoGap can dynamically determine the size of wires and ports, even if the input port sizes to a Mase FU is not know at compile time. This extends the functionality already existing in Verilog and VHDL with parameters and generics respectively. While the parameterized modules can have their port sizes set from the outside, they can not determine their own port sizes depending on where in the data path they end up. The dynamic port sizing in NoGap ensures that adding an instruction will not break already existing functionality even if the same FU is used by other instructions with hardware multiplexing.

A Mage description using dynamic input port sizing will usually require dynamic output port sizing as well. The out port sizes will be set during the construction of the Mase graph, The size of an output port can be written as a function of one or several input port sizes. Apart from the normal four operator (add, subtract, divide, and multiply) two additional operators can be used; binary maximum ($) and binary minimum (@).

F. Clause Name

All clauses, i.e. all code blocks starting with '{' and ending with '}' can be named. This is important for operation constructions since an operation on a leaf FU is constructed by declaring which of the clause(s) shall be reached. NoGap can compute how the input signals have to be set to reach this clause. A number of clause uses is illegal, NoGap detect those and inform the designer if an illegal construction is used.

G. Pipeline Description

The pipeline construct describes how a number of phases is related to each other. Each phase is linked to the other phase via a stage. A stage construct is a template description of how its outputs are related to its inputs . Phases are linked via stages, a stage might have 0 or longer delay . And for different operations, different pipelines can be constructed. For example, the multiply accumulate instruction in processor(see in Figure 2) uses the long pipeline, other instructions use the normal pipeline.

H. Inline Fu Generation

Functionality can be entered directly into an operation . The resulting size of the operation is computed using the same method as for dynamic port sizing.

I. Forwarding Path Description

Pipelined processors often have read-after-write (RAW) data hazards. To avoid RAW hazards, forwarding and bypassing path can be used. Forwarding paths are typically used in a processor where the operands are read and the results are written in seperate pipeline stages. The designer can explicitly specify in NoGap^{CL} how to solve data hazard using forwarding path. It can be constructed by specifying the path's source and destination, and the pipeline phase of source and destination port is also specified. Based on their description in NoGap^{CL}, the Verilog generation tool can automatically implement the necessary hardware for forwarding path.

J. Stall Operation Description

A real world processor must have the ability to be stalled depending on conditions outside of the processor's control. This is needed for example if the processor is going to use a cache or simply communicate over a shared resource with random delays. NoGap handles this by giving the designer the ability to insert stallable stages. The stallable stage has a single input controlling if stalling should be performed or not. The stall signal is assumed to come from an FU in the data path. If there is a stallable stage in the processor NoGap will insert a load enable signal for all registers in the design. As long as stall is asserted, no registers in the processor will be written, essentially stopping time for the processor. There must however be some registers which are still writable, e.g. if the stall logic is controlled by a FSM. For this reason, some registers in NoGap can be marked as unstallable. Note that this stalling does not describe the stalling needed to resolve resource conflicts in the data path.

K. Port Timing Offsets

Port in Mage FUs can be given timing offsets. Mage FUs might also have internal pipelining, NoGap can be notified about this by setting timing offsets for the ports affected. Both positive and negative offsets can be used. A port that has an offset will be placed a number of cycles earlier/later (as specified by the offset count) in the data path pipeline from its normal placement.

L. Static Connections and External Interfaces

It is sometimes desirable to link external modules, e.g. cores that have already been written for other projects. This can be done by using interface modules where some ports are connected to the data path controlled by NoGap and some of the ports are statically connected to top level input/output ports. A direct connection is a connection in the data path that

is independent of the pipeline. This is often useful for external memory interface, allowing processor cores to be used with different memory technologies.

M. Design Template

In a processor design, not all FUs can be independent from all other FUs. One example is a PC-FSM that should handle delay slots and flushing that is needed for program flow control. Not only the pipeline depth of the data path, but also the distance in cycles from the PC register to the decoder input, will determine how many cycles of flushing has to be done. For this reason some design elements can be expressed as template FUs. NoGap will then instantiate real FUs during the compilation process when NoGap have full knowledge of the system. By utilizing template designs it is possible to design, with NoGap^{CL}, general library FUs that can be reused in many different designs. Much like the standard template library in C++.

V. Results

As one example, a processor like the NoGapOne in Figure 2 is constructed using the NoGap^{CL}. The Verilog code generated by NoGap was synthesized to the Virtex-4 LX80 speedgrade 12 using the tool chain supplied by Xilinx ISE 10.0. The generated Verilog code contains no FPGA specific optimizations. For this reason both the instruction memory and data memory could not be synthesized into block RAMs and their sizes had thus to be kept reasonable.

Table I displays the utilization by part in the synthesized result. Two numbers are given for each column, A/B, A is the number of elements that belong to that specific hierarchical module, B is the total number of elements from that hierarchical module and any lower level hierarchical modules below.

TABLE I: FPGA utilization by part

Part	Slices	LUTs	DSP48
NoGapOne	358/1675	264/2544	0/1
+alu	177/177	258/258	0/0
+data. mem.	325/325	649/649	0/0
+control_path[a]	136/639	24/1024	0/0
++decoder	53/53	102/102	0/0
++instr. mem.	425/425	849/849	0/0
++pc	25/25	49/49	0/0
+mul	0/0	0/0	1/1
+rf	163/163	324/324	0/0
+sp	13/13	25/25	0/0

[a] All resource utilization on this level is from the pipeliner

The generated processor met timing closure at 178MHz as compared to xi2[10] at 334MHz and Microblaze at 200MHz. When comparing these values it is important to remember that both xi2 and Microblaze are heavily optimized for FPGA usage, whereas NoGapOne is completely generated by NoGap. Future work on NoGap will integrate possibilities to instantiate custom code depending on the targeted architecture and further research into target specific spawners would alleviate these problems. This comparison is not entirely fair since all processors are different. But it gives a ball park figure about how NoGap generated processors can compare to heavily optimized implementations.

VI. Conclusions

ADL tools are valuable tools that can be used to speed up design times for processor designs. Current ADL tools for processor design assumes a lot about the underlying architecture of the processor and thus limit the design freedom for novel processor architectures. Although NoGap also assumes something about the system being designed it assumes a lot less than any other ADL tools, giving the designer more freedom to explore novel processor architectures. The paper presents NoGap^{CL}, a flexible common language for processor hardware description. A RISC like processor has been implemented using NoGap^{CL}. The processor was designed with NoGap in under a workday and it was also synthesized to an FPGA. Future work will focus on compiler generation. The NoGap^{CD} is however too flexible to accurately generate compilers. Compiler generation would require using a subset of NoGap^{CD} with extra information extracted from a processor generation facet.

References

[1] P. Karlström and D. Liu, "Nogap: A micro architecture construction framework," in *Embedded Computer Systems: Architectures, Modeling, and Simulation*, K. Bertels, N. Dimopoulos, C. Silvano, and S. Wong, Eds., vol. 5657. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 171–180. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-03138-0_18

[2] P. Karlström, W. Zhou, and D. Liu, "Operation classification for control path synthetization with nogap," in *Information Technology: New Generations, Seventh International Conference on*, 2010.

[3] V. Zivojnovic, S. Pees, and H. Meyr, "Lisa - machine description language and generic machine model for hw/sw co-design," in *Proceedings of the IEEE Workshop on VLSI Signal Processing*, 1996, pp. 127–136. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.49.7123

[4] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau, "Expression: a language for architecture exploration through compiler/simulator retargetability," in *Design, Automation and Test in Europe Conference and Exhibition 1999. Proceedings*, August 2002, pp. 485–490. [Online]. Available: http://dx.doi.org/10.1109/DATE.1999.761170

[5] A. Fauth, J. Van Praet, and M. Freericks, "Describing instruction set processors using nml," in *European Design and Test Conference, 1995. ED&TC 1995, Proceedings.*, 1995, pp. 503–507. [Online]. Available: http://dx.doi.org/10.1109/EDTC.1995.470354

[6] R. Leupers and P. Marwedel, "Retargetable code generation based on structural processor descriptions," 1998. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.57.4520

[7] S. Rigo, G. Araujo, M. Bartholomeu, and R. Azevedo, "Archc: a systemc-based architecture description language," in *Computer Architecture and High Performance Computing, 2004. SBAC-PAD 2004. 16th Symposium on*, 2004, pp. 66–73. [Online]. Available: http://dx.doi.org/10.1109/SBAC-PAD.2004.8

[8] M. Itoh, S. Higaki, J. Sato, A. Shiomi, Y. Takeuchi, A. Kitajima, and M. Imai, "Peas-iii: an asip design environment," in *Computer Design, 2000. Proceedings. 2000 International Conference on*, 2000, pp. 430–436. [Online]. Available: http://dx.doi.org/10.1109/ICCD.2000.878319

[9] Osci, *System C*, Internet, www.systemc.org, January 2008.

[10] A. Ehliar, P. Karlstrom, and D. Liu, "A high performance microprocessor with dsp extensions optimized for the virtex-4 fpga," in *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*, September 2008, pp. 599–602. [Online]. Available: http://dx.doi.org/10.1109/FPL.2008.4630018