A Parametrizable Hybrid Stack-Register Processor as Soft Intellectual Property Module

Peter Lüthi, Thomas Röwer, Manfred Stadler, Daniel Forrer, Stefan Moscibroda, Norbert Felber, Hubert Kaeslin, Wolfgang Fichtner

Integrated Systems Laboratory, Swiss Federal Institute of Technology, Zürich, Switzerland

Abstract— **Hardware/Software Co-Design usually encounters serious problems to guarantee strong real-time constraints while serving many interrupt routines. We present an enhanced register-based RISC processor, which is capable of launching every interrupt routine within two clock cycles. This processor is implemented as soft IP-Module and features a customizable instruction set, extensive parameterization, and a synthesis model with separate core and interfaces. An automatic derivation of adequate test vectors from the current parameter setting verifies the correct functionality.**

I. Introduction

The design of integrated circuits is currently subject to extensive changes. Until now, project-specific code has been written for every new design. This results in highly optimized code for the target application, but also leads to inacceptable development time, especially for large designs. Since time to market and product lifetime are shrinking while, at the same time, circuit complexity is growing, the traditional way of designing integrated circuits has to be altered to achieve higher efficiency than before. As the complexity of circuits increases, there is urgent need for new design methodologies, that allow fast development of demanding applications up to complete system-on-a-chip integrations [1][2].

One possibility to cope with this efficiency problem is the use of Intellectual Property (IP) Modules or Virtual Components (VC). Quick and easy adaptations on the reusable blocks speeds up system design and provides more time for thorough testing, an important issue in cost-intensive chip design.

Another new trend in system design is the inclusion of programmable parts into the ASIC. This is commonly termed Hardware/Software Co-Design, which means that a system functionality is partitioned into hardware and software running on an embedded processor. This solution offers great flexibility by allowing fast alterations in functionality if project or system specifications are to change. Although software on embedded systems is very convenient, it poses numerous problems in conjunction with hard real-time requirements. Demanding telecommunication applications ask among others for an interrupt latency which is hard to obtain from traditional microprocessors.

This paper presents an embedded processor with a new approach in processor architecture to guarantee such demanding real-time constraints. Section II elaborates on the processor-specific IP requirements, introduces the architecture of the processor and its parameterization. Section III explains the functional verification flow for the IP-Module. In Section IV, we give an impression of the area occupation for different parameter sets and the details of our test integration. In the final section, we summarize the results of our work and discuss possible future enhancements.

The authors want to thank KTI (Swiss Commission for Technology and Innovations) for funding this project.

II. Project "SILVERBIRD"

A. IP requirements

Designing an embedded processor as a soft IP-Module engenders different design problems not encountered during the design of an application-specific integrated circuit (ASIC).

First of all, the user needs the ability to decide whether the IP-Module meets his requirements *at the beginning* of his project. Therefore all functionality and all limits concerning the processor IP must be stated clearly.

Moreover, the designer of a processor IP-Module has to be aware of the following issues:

1. A microprocessor IP has to be highly adaptable to satisfy multiple application requirements. Qualitative customization of the instruction set as well as quantitative customization of hardware parameters are important prerequisites.
2. An IP-Module can be plugged into different system environments. To make this possible, either various communication protocols must be supported by the IP itself or the possibility for the user to implement application-specific interfaces has to be clearly defined.
3. The processor IP-Module has to provide a convenient environment to allow for fast functional verification. It is highly recommended that this feature is already available during implementation time to verify the hardware/software concept and to reveal possible conceptual errors.
4. The ability to support a high-level programming language to allow quick adaptations on the software-based functionality. This feature greatly simplifies future software enhancements.

B. The "SILVERBIRD" IP-Module

During the design of our processor IP, close attention has been paid to all particularities of IP-Module design. We have realized a processor IP-Module featuring the following items:

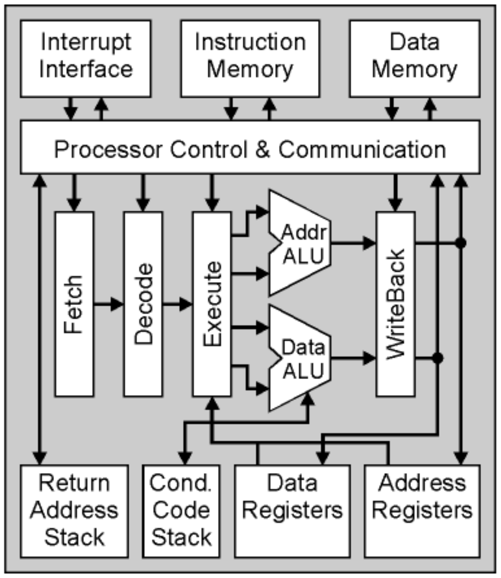


Fig. 1. Architecture of the "SILVERBIRD" RISC processor

- Qualitative adaptability: The functionality of the processor IP can be modified by customizing the instruction set.
- Quantitative parametrizability: Only the basic architecture of the processor core is fixed: Two separate ALUs for data and address computations, register-based architecture, and Harvard memory organization. On the other hand, all key parameters of this architecture are adaptable to the current application's needs. Fig. 2 shows the customizable parameters in Greek letters.
- Quick interface adaptations to comply with the target specifications. To achieve a clearly structured IP, the processor core and the system interfaces have been separated.
- The register-based architecture simplifies the implementation of a high-level programming language compiler.
- An assembler is part of the IP. It is self-parameterizing based on the parameters configured for the RTL model.

C. The processor architecture

To meet the demanding requirements of managing high interrupt loads and being parametrizable, we have decided to combine the advantages of a stack architecture with the ones of a register-based approach. Therefore, the general purpose registers of our processor are implemented as top-of-stack registers. In case of an interrupt, precious processing time can be saved by just pushing the current register contents on the stack. The maximum interrupt latency achieved by our architecture is two clock cycles. To obtain maximum processor performance, neither the pipeline is ever flushed nor any no-operation cycles are performed.

A striking argument against a pure stack processor was the need for compiler-compatibility: A compiler for a stack architecture is difficult to implement because it always needs to trace the exact position of each register [3][4]. As a consequence, the entire stack has to be controlled by "push" and "pop" instructions. Our solution provides a fixed amount of general purpose registers for every interrupt level. The whole stack control is done by the processor itself and requires no software-based "push" and "pop" operations. This organization is easy to support by a high-level compiler since the compiler does not have to control the stack at all.

One slight disadvantage of our architecture is the large chip area taken by the stacks, a consequence of the traditional trade-off between speed and area. But this can be avoided by implementing an interface from the top-of-stack registers to an on-chip RAM and spilling the major part of the stack contents to the RAM. It will result in more control logic and maybe in lower performance, unless the user builds a complex control logic to cope with the slow RAM. This way to save chip area is only preferable on large parameter values. On the other hand, decreasing costs for chip area and even increasing integration densities seem to justify this compromise.

Key features of the hybrid stack-register architecture of our RISC processor "SILVERBIRD" are:

- Separate data and instruction memory (Harvard architecture).
- Read-after-write sequences are allowed. Being able to access the same register in consecutive order yields much more efficient code.
- Data and address register banks have been implemented as top-of-stack, which allows for fast interrupt launch.
- A classic four-stage pipeline. In the first stage instructions are read from the program memory (Instruction Fetch). After decoding the instruction in stage two (Instruction Decode), it is executed in the third stage (Execute). In stage four, registers are updated and data memory access takes place (Write Back).
- An additional address ALU with reduced functionality for efficient block access operations on the data memory.
- Return addresses and condition code storage on separate stacks.
- Parametrizable instruction set of up to 40 instructions.

D. Qualitative adaptation of the processor core

Application-specific optimization of the core's functionality can be performed by selecting the *number of supported instructions* ν needed for the current application from a *total set* μ of 40 instructions. The hardware associated with the unwanted instructions will be implicitly discarded during logic synthesis. As an example, if all arithmetic data memory address instructions are disabled and only immediate memory access is retained, the address ALU will be completely removed. Otherwise, with full parameterization, the address ALU will be inferred (shaded areas I or II in Fig. 2).

E. Quantitative parameterization of the processor core

From the perspective of an IP user, the most significant adaptations influencing the final chip area have to be done by choosing the right number of general purpose registers, as well as the required stack depths. A thorough

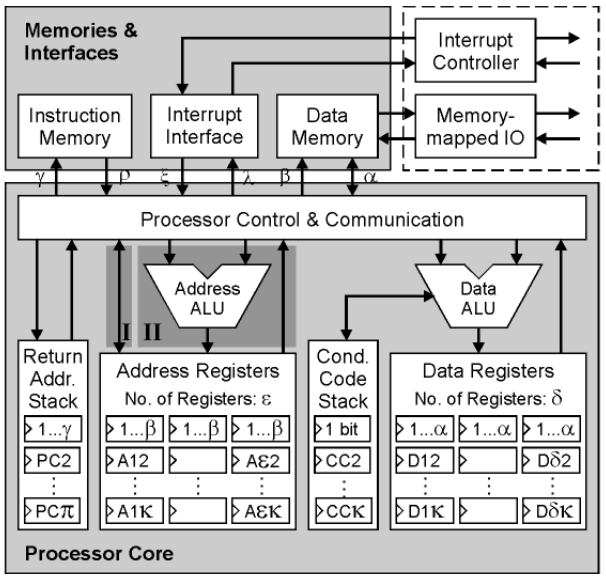


Fig. 2. "SILVERBIRD" IP-Module with customizable parameters shown as Greek letters

evaluation of the parameterization going to be used is strongly recommended since the stacks consume most of the processor area.

The following parameters can be varied:

1. *Data Width* α of the processor data path and of all data registers.

2. *Address Width* β of the data memory defining the maximum addressable memory size 2^{β}. The value for the address width β can range up to twice the data width ($\beta \leq 2\alpha$).

3. *Address Width* γ of the instruction memory defining the maximum accessible instruction memory size 2^{γ}. The instruction width is expressed by $[5 + 2{*}\text{ceil}(\log_2 \delta) + \alpha]$.

4. *Number of Data Registers* δ: Randomly accessible top of data stack registers. There is no upper boundary for this parameter, but excessive size will result in high area occupation.

5. *Number of Address Registers* ε: Randomly accessible top of address stack registers. It is either possible to increment the address register contents directly with the address ALU or an immediate offset can be specified within the memory instruction.

6. *Data and Address Stack Depth* κ: This stack depth defines the limit of simultaneously launched interrupt service routines. The expression $[\kappa * (\delta * \alpha + \varepsilon * \beta)]$ specifies the total number of required registers in the data and address stacks.

7. *Return Address Stack Depth* π: The return address stack depth π is dependent on the number of supported interrupt priorities and subroutine calls and has therefore to be adapted to the application by hand. The total amount of return address stack registers is calculated as $\gamma * \pi$.

An interrupt launch is always accompanied by the start address of the corresponding interrupt service routine. The width of this address is equal to the instruction memory address width γ. Therefore, the interrupt routines can be spread over the whole program memory. The carry flag is saved on the condition code stack during an interrupt. The condition code stack's depth equals the number of interrupt priority levels κ, its width is 1 bit.

In addition, the processor supports trap instructions with different priorities λ to allow for communication between processor and interrupt controller. When the processor executes a trap instruction, the corresponding priority will be passed to the external interrupt controller. This instruction can also be viewed as software interrupt from the processor to the interrupt controller. Finally, there are four different run-time exceptions: Data and Address Stack Under- & Overflow and Return Address Stack Under- & Overflow.

F. System interfaces

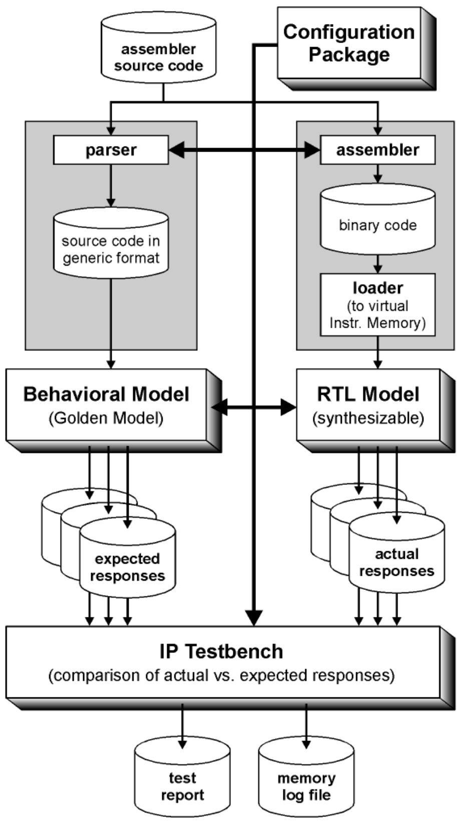
The processor IP-Module provides separate core and system interfaces to permit for easy adaptation to the target environment. A parametrizable data memory interface with a FIFO buffer supports Asynchronous Static RAMs. It allows memory burst writes from the processor core to the data memory without any processor stalls. FIFO depth, memory read latency, and write latency are individually parametrizable. For the instruction memory, an interface based on another Asynchronous Static RAM is available. Interaction and data exchange between the embedded processor and its system environment is done by an external interrupt controller and memory-mapped IO.

III. Functional verification

Functional verification of highly parameterized IP-Modules poses several problems. As described in section II, the IP-user can choose a parameter set that exactly matches the application-specific requirements. After this customization, the functional correctness of the IP-Module has to be verified. Because the designer of a parametrizable IP-Module can not provide test vectors for all possible parameter settings, a behavioral model has to do so: The expected responses for the synthesizable RTL model are generated based on the current parameterization by this behavioral model.

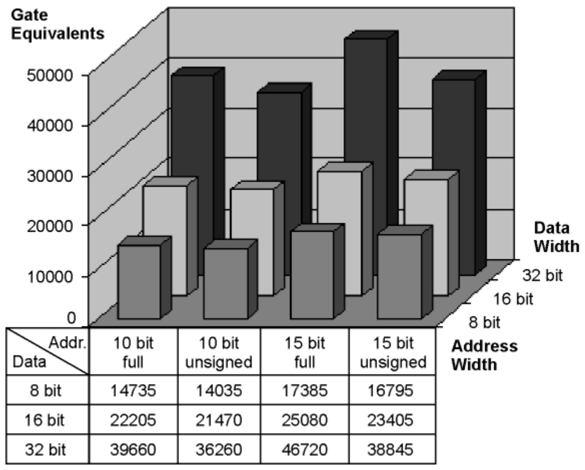
A suitable functional verification flow has already been published and comprehensively discussed in [3] and [5]. Here we only want to sketch this functional verification method. As Fig. 3 shows, the whole configuration flow is based on *one* configuration package. This guarantees consistency of the IP for synthesis and verification. An assembler source code file is used as common starting point for the flow consisting of the following three steps:

1. Translation of the assembler code into binary format by the parametrizable assembler (right side of Fig. 3): Thereby the assembler refers to the configuration package to check the range and validity of all parameters associated with the corresponding instructions in the assembler source code. In case of a mismatch, an error message is reported by the assembler. Since the assembler is entirely integrated



assembler source code
Configuration Package
parser
assembler
source code in generic format
binary code
loader (to virtual Instr. Memory)
Behavioral Model (Golden Model)
RTL Model (synthesizable)
expected responses
actual responses
IP Testbench (comparison of actual vs. expected responses)
test report
memory log file

Fig. 3. Verification flow of the "SILVERBIRD" IP-Module



Addr. / Data	10 bit full	10 bit unsigned	15 bit full	15 bit unsigned
8 bit	14735	14035	17385	16795
16 bit	22205	21470	25080	23405
32 bit	39660	36260	46720	38845

Fig. 4. Various synthesis runs with different parameterizations

into the verification flow, it's correctness is implicitly assured as well.

2. Generation of the expected responses from the assembler source (left side of Fig. 3). A behavioral model of the processor serves as a generator of the expected responses. They are first converted into generic format. The behavioral model processes the code and generates separate expected response files for every interface, which are used later on for the verification of the RTL model.

3. Functional verification of the customized RTL model (right side of Fig. 3): The binary code generated by the assembler is passed to the RTL model, which processes the code using the custom-specific settings. In a final step the generated output is compared against the expected responses and test report and memory log file are generated.

IV. Comparison of various parameterization examples and test integration

A. Configurations for the test synthesis runs

To get an impression of the area required by the processor IP-Module, we present an evaluation of 12 different parameter settings. For the demonstration of the effects of both quantitative and qualitative parameterizations, synthesis runs with different numeric parameters as well as with full and reduced instruction set have been carried out. Thereby the data width α was set to either 8, 16 or 32 bit and the address width β to 10 or 15 bit. The qualitative changes have been addressed by taking the entire instruction set μ, or a set reduced to unsigned instructions and immediate memory access only. As stack depths, we assigned $\kappa = 3$ to the data and address stacks and $\pi = 8$ to the return address stack. There are $\delta = 8$ top of stack data registers and $\varepsilon = 1$ address register. The number of supported traps is $\lambda = 8$.

B. Results of the various synthesis runs

While the processor version having a complete instruction set inferred both address and data ALU, the reduced version only made use of the data ALU. This is due to the restriction to immediate memory address operations and the elimination of all instructions needing an address ALU for calculations on address registers. Although the tiny 8 bit processor version does not reveal big differences between full and reduced instruction set, a significant influence is obvious with the 32 bit implementation. Increased area differences show up when larger data widths are chosen. This is to be interpreted as a consequence of the timing constraints, which are gaining more and more impact on large parameter values. The extra area overhead of the full instruction set version is likely to come from an increased inference of parallel acting functional blocks for meeting the timing requirements due to the higher complexity of the data ALU. From the perspective of area efficiency, the IP-Module allows no major optimizations concerning the chip area. The main part of the area required by the processor is occupied by data, address and return address stack registers. The area taken by these registers is inherent with the process used. The only combinatorial part, which can be influenced in size with different synthesis constraints is the execute stage. But the percentage of combinatorial chip area is of minor scale and therefore negligible.

C. Test integration

For final verification of the functionality and qualification of speed and power consumption of our IP-Module, we let the circuit fabricate with on-chip data memory (see Fig. 5 and Table I). For the test implementation a 0.6 μm 3 layer

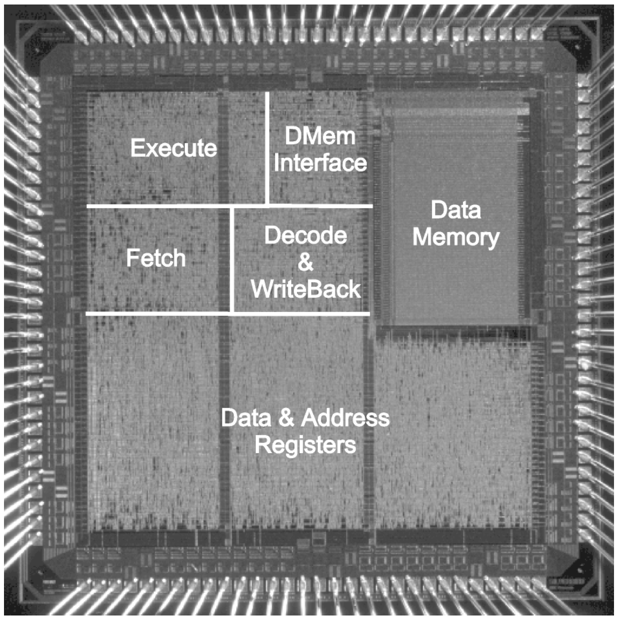


Fig. 5. Picture of the test integration with on-chip data memory

metal CMOS process has been chosen. The settings of the parameters for the integration are given in Table II.

Process	0.6 μm 3LM CMOS
Supply Voltage	5 Volt
Chip size incl. pads	4.6×4.6 mm
Chip area incl. pads	21.16 mm^2
Core area incl. data memory	13.69 mm^2
Max. operating frequency	121.5 MHz
Max. throughput	121.5 MIPS
Max. interrupt latency (without memory R/W stalls)	2 Tclk $\widehat{=}$ 16.46 ns

TABLE I
Key values of the test integration

The program and data memory interfaces are implemented for asynchronous RAMs. While the data memory has been chosen on-chip, the instruction memory is placed off-chip. Finally all stacks (return address stack, condition code stack, data and address stacks) are implemented using registers.

V. Results and Outlook

We have developed an embedded processor IP-Module that is highly adaptable in both functionality and configuration. This was achieved by separating the processor core and the system interfaces. The hybrid stack-register processor is excellently suited for applications with high interrupt loads. There is a convenient functional verification flow covering automatically the configuration of the processor. It uses assembler code as common starting point for both synthesizable RTL model and behavioral model.

Furthermore, implementing a high-level language compiler could so be done easily because of the register-based architecture, the read-after-write operation support, and the fully parametrizable assembler. As the assembler is already integrated into the verification flow, no further adaptations to the flow are necessary to check the complete processor-assembler package. To compile high-level code for the current configuration of the IP-Module, a compiler needs to know about the instruction set, the data width, the number of data and address registers and the data memory address range.

As conclusion, we estimate the design effort for the IP to be about twice as much as for a one-time implementation. But this extra effort is easily recovered in future system designs because the extensive parameterization, the adaptable instruction set, the configuration-independent verification flow and the self-parameterizing assembler make reuse of our IP-Module very simple. Additionally, the ability to check hardware/software concepts already during implementation time allows for straightforward design and saves precious development time.

Param.	Description	Value
α	Data Width	16 bit
β	Data Memory Address Width	10 bit
γ	Instruction Memory Address Width & Interrupt Address Range	11 bit
δ	Number of Data Registers	12
ε	Number of Address Registers	2
κ	Data / Address Stack Depth	4
π	Return Address Stack Depth	20
λ	Number of available traps	8
$\nu\widehat{=}\mu$	Number of supported instructions	40
ϱ	Instruction Width $(5 + 2{*}\text{ceil}(\log_2 \delta) + \alpha)$	29 bit

TABLE II
Parameter settings for the test integration

References

[1] Stefan Pees, Martin Vaupel, Vojin Živojnović, and Heinrich Meyr, "On core and more: A design perspective for systems-on-a-chip," in *Proc. International Conference on Application-Specific Systems, Architectures and Processors*. IEEE Press, July 1997, pp. 448–457.

[2] Rajesh K. Gupta and Yervant Zorian, "Introducing core-based system design," *IEEE Design & Test of Computers*, vol. 14, no. 4, pp. 15–25, October-December 1997.

[3] Thomas Röwer, Manfred Stadler, Markus Thalmann, Norbert Felber, Hubert Kaeslin, and Wolfgang Fichtner, "Intellectual property module of a highly parametrizable embedded stack processor," in *Proc. Twelfth International IEEE ASIC/SOC Conference*. IEEE, September 1999, pp. 399–403.

[4] John L. Hennessy and David A. Patterson, *Computer Architecture a Quantitative Approach*, Morgan Kaufmann Publishers, second edition, 1996.

[5] Manfred Stadler, Thomas Röwer, Markus Thalmann, Norbert Felber, Hubert Kaeslin, and Wolfgang Fichtner, "Functional verification of intellectual properties (ip): a simulation-based solution for an application-specific instruction-set processor," in *Proc. International Test Conference*. IEEE, September 1999, pp. 415–420.