Hardware-Software-Codesign of Application Specific Microcontrollers with the ASM Environment

A.W. Both, B. Biermann, R. Lerch, Y. Manoli, K. Sievert

Fraunhofer-Institute of Microelectronic Circuits and Systems
Finkenstr. 61, D-47057 Duisburg, Germany

A new CAD system for generation of application specific microcontrollers is presented. It is based on a properly defined architecture model and consists of a set of tools for instruction set description, HW-SW-cosimulation of the processor and the application software, compiler generation, datapath and controlpath synthesis, and datapath animation. One of these tools, the MicroDebugger, follows a new approach of HW-SW-cosimulation particularly matching the problems of microcontroller design. The system proved it's effectiveness throughout numerous designs including a small multitasking RISC for embedded applications.

1. Introduction

The Application Specific Microcontroller (ASM) Environment is a CAD system for rapid development and prototyping of instruction set architectures, or ASIPs (Application Specific Integrated Processors) as introduced by Alomary et al. [1]. These circuits consists of a CPU core, memory and peripheral circuits and are mainly targeted to embedded applications for fast growing markets like automotive, telecommunications and mechatronics control.

As it is pointed out in [1], many high-level synthesis systems surveyed in the literature [2][3][4] target general purpose microprocessors or digital signal processors where operation scheduling and resource allocation have the biggest impact on the overall performance. However, for a lot of applications, this is not generally true. Actually, for a high performance embedded controller operation scheduling and resource allocation play a significant role, but an optimal instruction set and a well chosen architecture, which allows short interrupt latency, play at least an equally important role. Even worse, some applications are not speed critical but very sensitive to EMI (Electro-Magnetic Interference). They are better served by lower clock frequencies and architectural robustness for electromagnetic compatibility.

The attempt to automatically generate application specific microcontroller cores (instruction set architectures) will fail if one does not restrict the huge design space. Such a restriction must not affect the implementation method but it must properly define an architecture model for synthesis. The criteria for such a first design space demarcation result from an analysis of the class of targeted applications. The ASM Environment targets control and interrupt dominated applications with medium data throughput. Architectures of this class are part of a lot of electronic equipment, as embedded microcontrollers in the automotive, industrial, telecom and white good markets.

Furthermore our architecture model is based on the assumption that it will be programmed in C rather than in assembler language. In practice this assumption seems to be highly justifiable as more and more embedded applications are programmed in a high level language for increased code transparency and reusability, even though the high level language support of the underlying architectures is often restricted.

The remainder of this paper is organized as follows: Section 2 gives an introduction into the structure of the ASM Environment. In section 3 an example for an instruction set description is discussed. In section 4 we explain in detail our methodology for HW-SW-Cosimulation using the *MicroDebugger*. Datapath and control path synthesis are subjects of section 5 and 6, respectively. A brief conclusion and some considerations about further investigations finishes the paper.

2. Overview of the ASM Environment

Fig. 1 shows an overview of the ASM Environment. A detailed analysis of the application or class of applications yields a machine and instruction set description. This is still a highly manual procedure, supported by automated tools for the quantitative analysis of the application source code. The instruction set description serves as input for the *Model Synthesizer*

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission. © 1994 ACM 0-89791-687-5/94/0009 3.50

which generates in a first step functional models for the datapath in an HDL as well as the microcode in an uncompressed binary form. At this stage the microcode is only used as a source of how and when to switch the different paths in the datapath and activate the different operations inside the ALU. Finding an adequate and area efficient implementation for the control is the task of the *Control Path Synthesizer*.

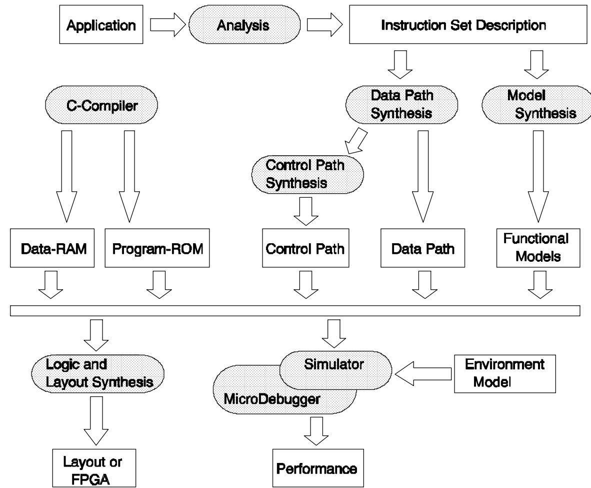

Fig.1 Main Design Flow within the ASM Environment

Starting from the machine description it is possible to generate a C compiler and an assembler which are dedicated exactly to the given instruction set architecture. In contrast to [1] this C-Compiler is not based on GCC, but on a completely new programmed retargetable C Compiler [5]. Hence the compiler does not restrict the architecture to 16 or 32 bit.

The behavioural models of datapath, control and peripherals can be simulated using a commercial circuit simulator. Via interprocess communication (IPC) the simulator is coupled to the *MicroDebugger*, which enables the user to concurrently simulate the hardware and the application software. Thus performance estimation and software development is supported on purely functional, RTL, gate-level as well as mixed level. Up to now the Verilog simulator and the Lsim simulator are supported. Another IPC channel links the *MicroDebugger* to the *Datapath Animator*. This tool visualizes the datapath structure and highlights the currently active connections during a simulation.

The HDL models generated by the ASM Environment are synthesizable by a commercial logic synthesis tool. Thus rapid prototyping on FPGAs is supported. Furthermore, this and the usage of conventional layout synthesis tools provides a proven way from the functional level down to mask level.

3. Instruction Set Description

The development of an instruction set of an application specific microprocessor core in the ASM Environment is a highly iterative procedure starting with a first draft coming either from the statistical analysis of the application program or from a previously generated processor core. The instruction set and the data processing hardware blocks are specified in a C-like language. The description is parsed by the *Instruction Set Specification Interface*, which generates the internal database consisting of coupled data flow and structure graphs [6].

```
/* the registers */
long dedireg        pc;                          /* 16 bit pc */
dedireg             fp, cc, ir;                  /* dedicated 8 bit regs */
register            X,Y,Z;                       /* temp. regs */
nil                 NIL;

/* the busses */
memory              dbus {r,w};                  /* data bus */
long memory         ibus {r};

/* variables */
int                 opcode, pc_l, pc_h ;

/* the functional units */
branchcalc          branch {add,adds};
statuscalc          statcalc {U,U,U,U,C,N,Z,U};
alu                 alu_0 {&,|,^,+, -,cmp,>>,==,<=,~};
mask                mask_0 {0, 1, 8, 16,255};
incrementer         inc_0 {++};
jmpmux              jump_0, jump_1;
cline               alu_crin_ctrl0 {status}, stop {one,zero};
```

Fig. 2 : Declaration of the hardware elements

Roughly divided the instruction set description contains three sections :

1. Declaration of the available hardware elements (registers, busses, ports, functional units)
2. Declaration and assignment of temporary variables
3. Instruction scheduling description

Fig. 2 shows an example for section 1. Registers, busses and functional units like the ALU or a dedicated branchcalculator are declared in this section. All units can be declared as single word or double word units where the word width has to be predefined to 4, 8 or 16 bit.

```
/* the instructions */
instructionset() {

  switch (opcode) {
        case lwg    : seq(par(Y=ibus[pc],X=X,PCINC),
                          par(X=X,Z=dbus[Y]),
                          par(dbus[X]=Z,PREFETCH));
        case swg    : seq(par(Y=ibus[pc],Z=dbus[X],PCINC),
                          par(dbus[Y]=Z,PREFETCH));
        case rtf    : seq(par(fp=X, X=X-1),
                          par(pc_l=dbus[X],X=X-1),
                          pc_h=dbus[X],
                          par(PREFETCH));
  }
}
```

Fig. 3 : Examples of an instruction set description

The ASM Environment was used to describe the instruction set of a small RISC architecture. This architecture was designed to support C-programming and fast context switching. The description of three instructions (load word global (lwg), store word global (swg), return from function (rtf)) of this processor is shown in Fig. 3. The keywords *par* and *seq* mean, that the parenthesized actions are happening concurrently or sequentially, respectively. As the complete information about operation scheduling and resource allocation is part of the instruction set description it could be seen as another form of symbolic microprogramming. However the description on this level usually is compact enough so that additional effort for further automation was not made at this stage.

4. Hardware-Software-Cosimulation

The need for a cosimulation tool arises immediately since an iterative optimization of the instruction set might be necessary. With a standalone conventional circuit simulator such a simulation task would be difficult to handle - the problem is the data abstraction and the lacking noninteractive simulation control as well as the lack of software debugging aids.

In the ASM Environment cosimulation has to be understood as the simulation of a processor running his application software together with additional hardware. The processor and the additional hardware exist as behavioural modules. This is a major difference to the system of Thomas, Adams and Schmit [7], where the software runs as a different UNIX process on a general purpose CPU, i.e. the host system. Their approach requires a complicated synchronization mechanism between the different processes, which can be avoided with our approach. On the other hand a software process running on the host system allows to use conventional source code debuggers. Since this is not possible in our approach, it motivated us to design the *MicroDebugger*. The *MicroDebugger* is dedicated to debug software running on a simulated processor core as well as the simulated hardware itself. Thus the *MicroDebugger* incorporates the following features :

- A homogeneous user interface throughout all abstraction levels from architectural evaluation down to gate and even transistor level.
- Assembler and C-source display with current instruction highlighting.
- Single clock, single step as well as free running operation.
- Full symbolic access to C-variables, hardware registers and processor inputs/outputs.
- Support of program and data break points. Program break points can be specified in the C- as well as in the assembler display.
- Interface to the *Datapath Animator* for highlighting the active connections.

During a cosimulation session the simulator transmits the current value of the program counter to the *MicroDebugger* at a configurable condition, e.g. an active clock edge. This enables the *MicroDebugger* to highlight the corresponding line in the assembler listfile and the C application program. Then the simulator enters a wait state - it remains in this state until it receives a message or request from the *MicroDebugger.* In case of a single step or a free running operation the *MicroDebugger* sends *simulate*-commands until a load instruction register event (single step) or a breakpoint event (e.g. pc value matches the address breakpoint) is detected.

Furthermore the tool provides a comfortable way to simulate large programs including reaction on external events without manual intervention. In this case simulator control commands can be specified within the software code using a dedicated directive. For example, peripheral timers can be set to a predefined value or interrupts can be triggered automatically. This feature is particularly powerful for writing complete suites of test programs which usually serve as a reference in all stages of top-down design.

In order to achieve maximum portability the graphical interface of the *MicroDebugger* was designed

with the XView library running on top of the X-window system. Usually during a software debugging session the simulator will be iconized as this increases the simulation speed by a factor of 2. Whereas the first implementation of the tool communicated over bidirectional named pipes with the simulator, the newer version works as a client/server application using UNIX sockets. Fig. 4 shows a screenshot of a cosimulation session with the *MicroDebugger* and the Lsim simulator.

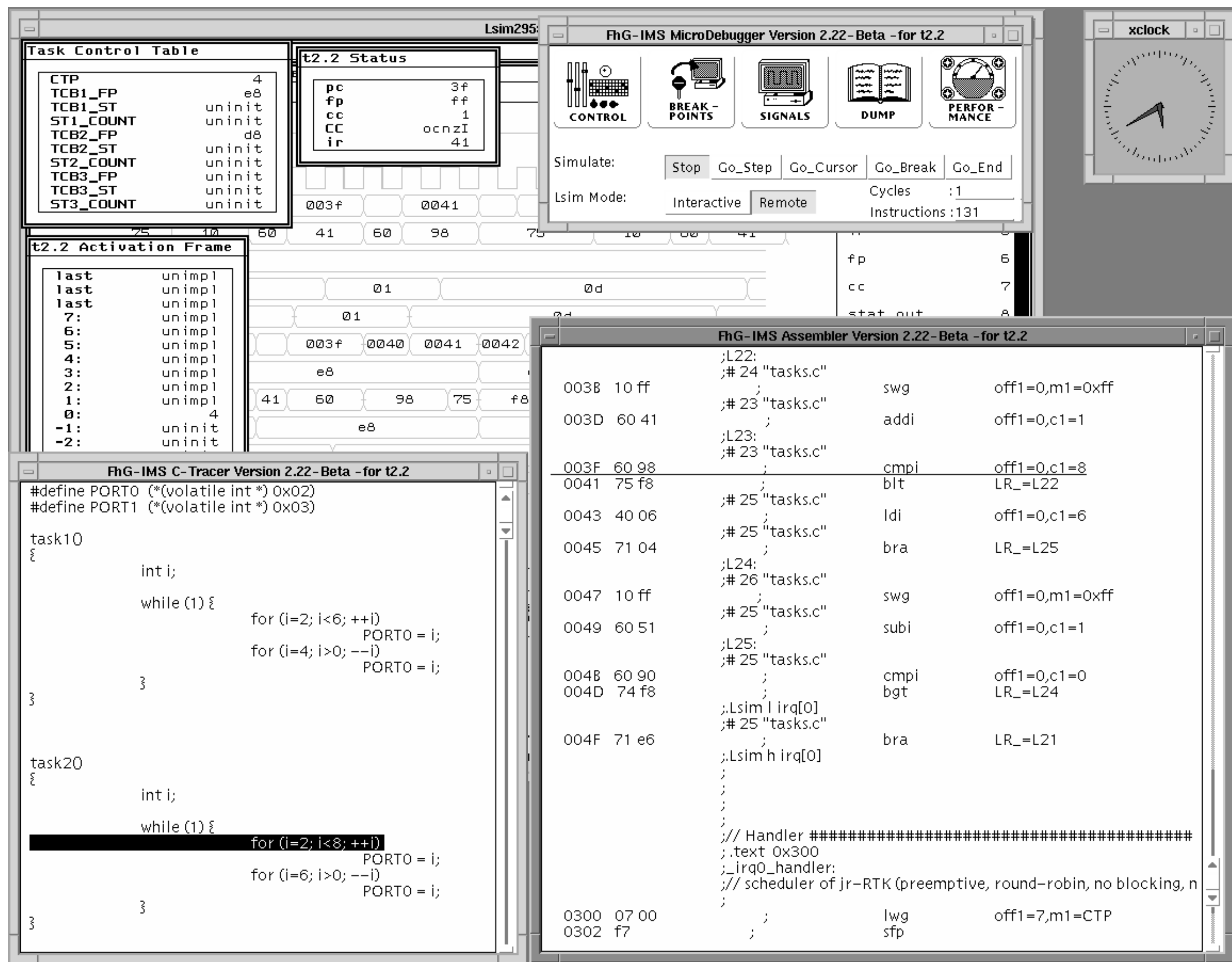

Fig. 4 : Cosimulation session with the *IMS MicroDebugger*

5. Datapath Generation

After evaluation of the instruction set on the architectural level, functional models of the datapath are generated by the *Model Synthesizer*. Similar to the approach in [1], the model generation is based on a module library which comprises ALUs, incrementer/decrementer, branch and status calculator, mask generator, etc.

6. Control Path Synthesis

As a byproduct the *Model Synthesizer* produces the microcode in an uncompacted binary form totally free of implementation issues. In general there is the choice between a microcoded or a hardwired control path. Depending on the chosen implementation style, the *Control Path Synthesizer* has to fulfil different tasks.

For a microcoded machine the main task is to choose an adequate sequencing mechanism and to apply compaction techniques [8,9]. For hardwired control the binary microcode is associated with the output of a finite state machine. The number of states mainly depends on the availability of the opcode during a complete instruction cycle. The main tasks now are state reduction by common tail sharing and state assignment using NOVA [10], MUSTANG [11] or the newer STOIC [12] algorithm.

The *Control Path Synthesizer* is programmed in C and incorporates a C-Scheme interface as a Lisp interpreter. This allows to write meta-programs with different strategies for generating an area-efficient implementation of the control in form of an HDL model.

7. Conclusion and Further Investigations

We presented a new CAD system for the automatic synthesis of application specific microcontroller cores. The ASM Environment consists of a set of tools for instruction set description, HW-SW-cosimulation of the processor together with the application software, compiler generation, datapath and controlpath synthesis, and datapath animation.

The hardware-software cosimulation tool *MicroDebugger* proved it's usefulness throughout numerous design sessions. One of the microcontroller designs included a collection of timers and counters which were in a complex interaction with external events and of significant sequential depth (22 bits). How can an event which occurs only every 2^{22} clock cycles be simulated safely and reproducable in a complex system environment? The problem was perfectly addressed by our cosimulation tool which is able to preset any register or state of any part of the simulated system using automated and sychronized mechanisms.

Further investigations will be performed in the improvement of the Instruction Set Specification Interface and the Control Path Synthesizer. For example we have not yet applied our approach to the synthesis of complex pipelined architectures. Another branch of research will be area estimators for different implementation styles during control path synthesis and the meta-programming. Since more and more applications use an IEEE 1149.1 conformant test interface, the MicroDebugger will be enhanced for semiautomatic simulation of such an interface.

References

[1] Alomary, A., Takeharu, N., Yoshimichi, H., Jun, S., Nobuyuki, H., Masaharu, I.: "PEAS-I: A Hardware/Software Co-design System for ASIPs", *Proc. of Euro-DAC '93*, pp. 2 - 7, 1993

[2] Camposano, R., Walker, R.A., eds.: "*A Survey of High-Level Synthesis Systems*", Kluwer Academic Publishers, 1991

[3] Camposano, R., and Wolf, W. eds.: "*High Level VLSI Synthesis*", Kluwer Academic Publisher, 1991

[4] Catthoor, F., Svensson, L., eds.: "*Application-Driven Architecture Synthesis*", Kluwer Academic Publishers, 1993

[5] Krohm, F.: "*Ein retargierbarer Compiler für anwendungsspezifische Mikrocontroller*", VDI Verlag, R.20, Nr.69, 1992

[6] Haeck, H.-G., Krohm, F., Manoli, Y. : "Data Path Synthesis from a Microcontroller Instruction Set Specification in MicroSyn", *Microprocessing and Microprogramming 32*, North Holland, pp. 193 - 198, 1991

[7] Thomas, D.E., Adams, J.K., Schmit, H. : "A Model and Methodology for Hardware-Software Codesign", *IEEE Design & Test of Computers*, September 1993, pp. 6 - 15

[8] Landskov, D., Davidson, S., Shriver, B., Mallett, P.W.: "Local Microcode Compaction Techniques", *ACM Computing Surveys*, Vol.12, No.3, 1980

[9] Wei, R.-S., Tseng, C.-J. : "Column Compaction and Its Application to the Control Path Synthesis", *IEEE Trans. on CAD,* pp. 320 - 323, 1987

[10] Villa, T., Sangiovanni-Vincentelli, A.: "NOVA: State asignment of finite state machines for optimal two-level logic implementations", *Proc. 26th Design Automation Conference*, pp. 327 - 332, June 1989

[11] Devadas,S., Ma,H.T., Newton,A.R., Sangiovanni-Vincentelli,A., "MUSTANG: State Assignment of Finite State Machines for Optimal Multi-Level Logic Implementations", *Proc. ICCAD 87*, 1987, pp. 16-19

[12] Pomeranz, I., Cheng, K.-T. : "STOIC: State Assignment Based on Output/Input Functions", *IEEE Trans. on CAD of Integrated Circuits and Systems*, Vol. 12, No. 8, pp. 1123 - 1131, August 1993