Retargetable Assembly Code Generation by Bootstrapping

Rainer Leupers, Wolfgang Schenk

University of Dortmund,
Lehrstuhl Informatik 12
D-44221 Dortmund, Germany

Report No. 488

July 1993

ABSTRACT

In a hardware/software codesign environment compilers are needed that map software components of a partitioned system behavior description onto a programmable processor. Since the processor structure is not static, but can repeatedly change during the design process, the compiler should be retargetable to avoid manual compiler adaption for any alternative architecture. A restriction of existing retargetable compilers is that they only generate microcode for the target architecture instead of machine-level code. In this paper we introduce a bootstrapping technique allowing us to translate high-level language (HLL) programs into real machine-level code using a retargetable microcode compiler. The retargetability is preserved, permitting to compare different architectural alternatives in a codesign framework within relatively little time. As an application of the new code generation technique we consider hardware/software codesign of heterogeneous information processing systems.

Contents

1 Introduction 1

2 Microcode Generation in the MIMOLA Software System 3

3 Bootstrapping Approach 6

4 Micro-ROM Generation 8

5 Machine Code Generation 10

6 Code Example 12

7 Hardware/Software Codesign for Heterogeneous Systems 14

8 Exploring Architectural Alternatives 15

9 Conclusions 18

1 Introduction

The "hardware/software codesign" approach increasingly gains importance in digital system synthesis from behavioral descriptions. Codesign means partitioning an abstract behavioral description into hardware and software components forming a system with the specified behavior and meeting given timing restrictions. Especially it supports design of digital controllers performing real-time computations. The target architecture might be a simple system containing a programmable processor, a main memory, and several ASICs, as proposed in [1] (fig. 1).

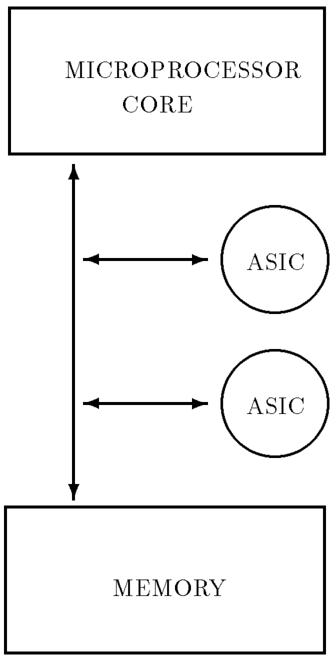

Figure 1: System Architecture

There, software components of the partitioned description are executed on the processor (core), whereas real-time routines are performed by ASICs (the hardware components of the partitioned description). Additionally the processor controls component communication that is done using a central bus. A standard processor as well as a processor core may be used. Especially in the DSP domain a very simple core may suffice, since DSP systems typically perform short, arithmetic-intensive tasks. In case that arithmetical subtasks are done by ASICs, the smallest possible core should be chosen, that can execute the necessary software components still allowing other hardware modules to be put onto the same chip. Complex behavioral descriptions might require extended target architectures, e.g. more than one core and local component communication, but the simple model depicted in fig. 1 suffices for studying general requirements on codesign tools.
Since communication overhead implied by a certain system partition is hardly predictable, codesigning a digital system requires several iteration steps in general (fig. 2).
During the iteration the necessary hardware and software components change, causing different core

requirements in each step. In order to simulate the system behavior for a given hardware/software partition the hardware components have to be synthesized and the software components have to be mapped onto the core. For the latter a compiler is needed that translates an HLL program into the core instruction set. Commercial compilers are available for some standard processors but never for special cores. Therefore we recommend using a retargetable compiler, processing both an HLL program and a processor (core) description, and producing machine code for the described hardware. The compiler retargetability enables the designer to study different core alternatives without manually changing the compiler itself.

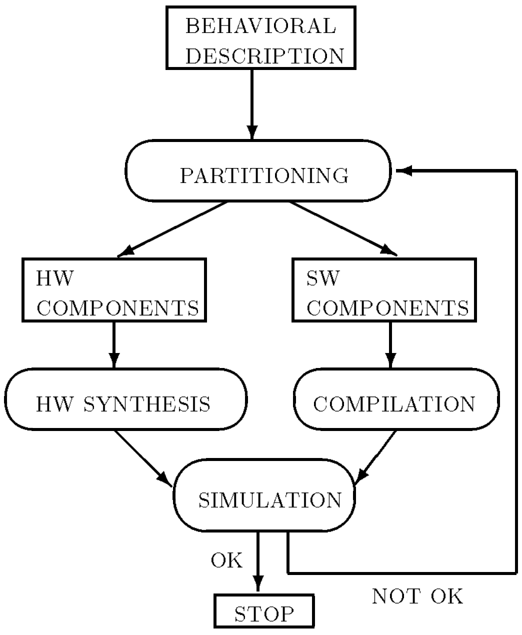

Figure 2: HW/SW Codesign Flow

Several retargetable compilers are mentioned in the literature [2, 3, 4], among them our code generator MSSC. MSSC takes both a target structure description and a PASCAL program emitting binary code automatically, that executes the PASCAL program on the given structure if possible. A disadvantage of those compilers (when using it in a codesign environment) is that only code for the lowest programming level is generated, i.e. microinstructions. if only the processor datapath but not the controller is to be changed, the core still must be programmed in (assembly-level) machine code. Generating machine code is not provided by the above compilers. The GNU C Compiler [5], which is also regarded to be retargetable, generates assembly level code, but retargeting requires a large amount of translation directives and recompiling for each target machine. MSSC generates code from the pure RT-level machine description and therefore needs not to be recompiled. To fill the gap between microcode and assembly level code generation, we propose a bootstrapping technique enabling the designer to translate an HLL program into machine code using an existing retargetable microcode compiler. Using this technique makes the microcode compiler a useful tool

in a codesign framework. Different processor datapaths implying different machine instruction sets can be tried during the design iteration without compiler adaption, still allowing the designer to specify software components in high-level language, and thus accelerating evaluation of a certain system partition. The approach even may be used for standard processors, when no target-specific compiler is available although this is not its main purpose.
We describe the bootstrapping approach using the TMS320C25 as an example, a widely-spread standard DSP. The compiler MSSC itself has been described elsewhere [6, 7], so only a short overview is given in the following section. After that the basic bootstrapping idea is explained, followed by a detailed description of the two main steps (micro-ROM generation and machine code generation). An example for generated machine-level code is given in section 6. Finally it is shown how the new code generation technique can be applied for hardware/software codesign of heterogeneous systems. The paper ends with a conclusion.

2 Microcode Generation in the MIMOLA Software System

The retargetable microcode generator MSSC is part of the MIMOLA Software System (MSS), which supplies hardware synthesis, generation of self-test programs, simulation and schematics generation, too [10, 11]. Every MSS tool is based on the MIMOLA language that allows both hardware and software descriptions. Hardware descriptions contain RT modules, their behavior and their interconnections. For instance, a 32 bit ALU might be specified in MIMOLA as follows:

```
MODULE ALU (IN a, b : (31:0);
            OUT outp: (31:0);
            FCT ctr : (1:0))
BEGIN
 outp <- CASE ctr OF
  0:   a + b;
  1:   a - b;
  2:   a;
  3:   a XOR b;
 ENDCASE
END;
```

The ALU has two 32 bit data inputs, a 32 bit output, and a 2 bit control input selecting the ALU function. A complete hardware description enumerates all modules and all interconnections (wires). For code generation one register has to be marked as program counter and one memory module as instruction storage.
Module interconnections are explicitly given in the MIMOLA description, e.g. by

```
CONNECTIONS ACCU.outp -> ALU.a;
            ALU.outp  -> ACCU.inp;
```

Software descriptions in MIMOLA may consist of PASCAL statements, but RT-level programming is supported, too. All high-level control structures (FOR, WHILE, REPEAT,...) are supplied, but there are no predefined data types. They can be declared by bitstrings, e.g. by

```
TYPE Integer = (15:0);
```

Definition of complex data types (ARRAY, RECORD) is supported, e.g. the user could declare

```
TYPE ArrType = ARRAY [1..10] OF Integer;
```

Hardware and software description together form the input to MSSC, that translates the given program into microinstructions for the given programmable hardware structure (fig. 3). MSSC has been described in detail in [6, 7], we only give a rough summary of the four main steps here.

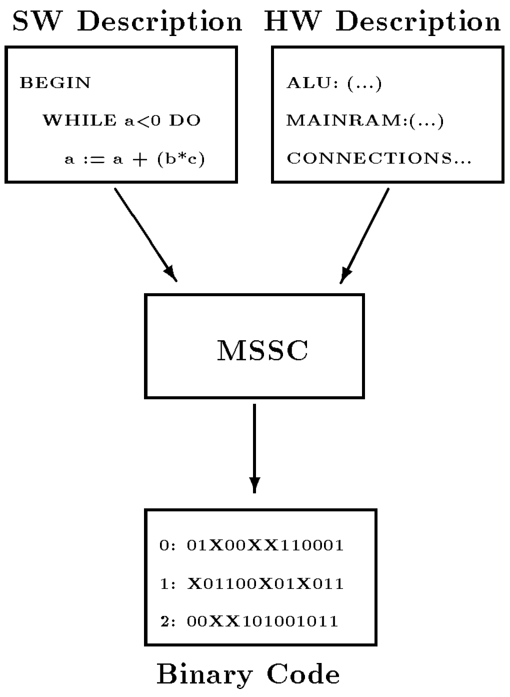

Figure 3: Functionality of MSSC

1. **Program transformation:** The software description is transformed into an RT-level program. All user variables are mapped onto real memory locations, and loop structures are replaced by conditional jumps. Either default or user-defined replacement rules are used. For example the loop

   ```
   <label>: REPEAT <block> UNTIL <cond>
   ```

   could be replaced by

```
<label>: <block>
         PC := IF <cond> THEN "INCR" PC
                         ELSE <label>
```

where PC denotes the program counter. So, execution of <block> is repeated until <cond> becomes true. The result is an RT-level program that only may contain IF-statements as high-level elements. IF-statements can be mapped onto hardware directly using multiplexers or comparators.

2. **Preallocation:** The *Connection Operation Graph* is constructed that represents the hardware structure. It contains vertices for every operation performed by the modules and edges for their interconnections. Within the preallocation step suitable assignments to instruction word fields (*versions*) are calculated for each possible hardware operation. Module control codes can be provided directly by the instruction word as well as via other modules or by hardwired constants. If for instance an ALU addition is selected by ALU control code c which can only be supplied by a decoder, MSSC tries to find an appropriate version forcing the decoder to generate code c. This ability of indirectly supplying control codes via decoders is crucial for our bootstrapping technique. Since a large number of versions might be found for each hardware operation during preallocation, a special data structure is used for efficiently handling version alternatives.

3. **Code generation:** Code generation is done by pattern matching within the Connection Operation Graph. Every assignment can be represented by a data flow tree. If the CO-Graph contains a matching subtree, the assignment can be allocated immediately. Otherwise, the assignment is split and MSSC tries to find an allocation using temporary cells. If a statement cannot be allocated even when using temporaries, MSSC generates an error message indicating the failure reason and location, e.g. a missing path between modules. In that case the hardware structure has to be modified. The result of successful code generation is a list of micro-orders each performing one micro-operation.

4. **Scheduling:** Finally the micro-orders have to be packed into complete microinstructions (control store words). Data dependencies as well as compatibility of micro-orders have to be obeyed. Micro-operations executable in parallel are packed into one control step. Additionally unused registers and tristate bus drivers must be disabled.

The result then is a microprogram executing the given PASCAL program on the target structure. MSSC has proven its capabilities for numerous target structures, e.g. implementation of a virtual machine language interpreter on the VLIW processor SAMP [8, 9].

3 Bootstrapping Approach

This section gives an overview of the bootstrapping technique. A detailed explanation containing examples is given in sections IV and V. The basic idea for generating machine-level instead of microinstructions is a two-phase use of MSSC. In the first phase MSSC produces a binary program that corresponds to the processor instruction set. This program is stored into a micro-ROM (a decoder), that serves as an additional input in the second phase. Extending the hardware description by the micro-ROM enables MSSC to translate an HLL program into machine-level code in phase 2. This means, phase 1 uses the microcode compiler MSSC for "bootstrapping" a real HLL to machine code compiler for a specified processor structure and its machine instruction set. The whole procedure is shown in fig. 4.

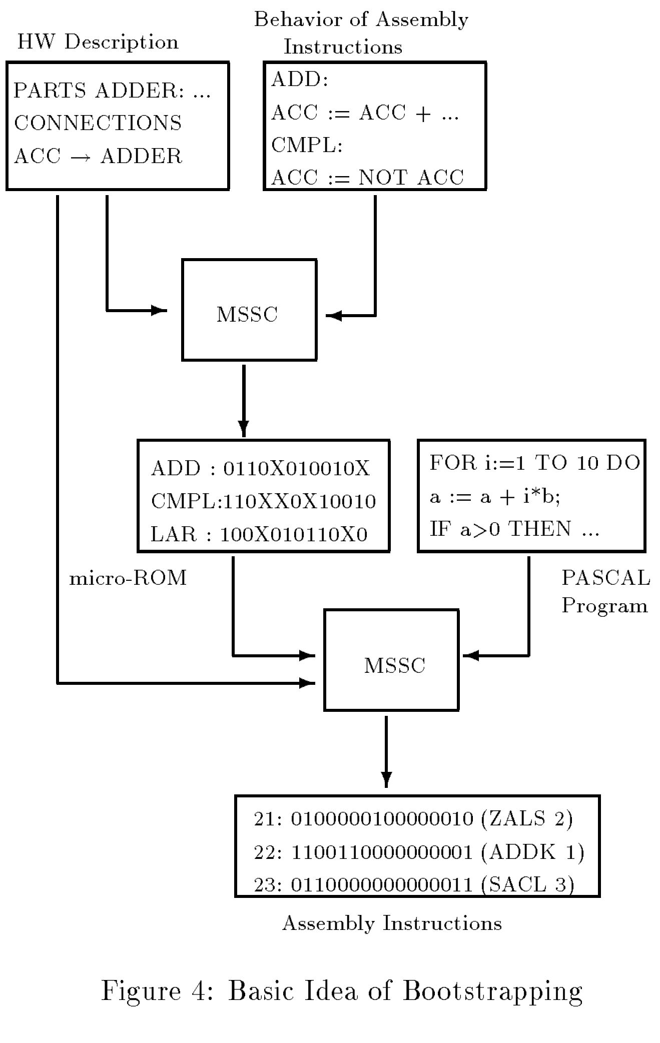

Figure 4: Basic Idea of Bootstrapping

Phase 1 (Micro-ROM generation)

1. **Hardware structure modelling:** The target processor's RT-structure is described in MIMOLA, containing the datapath, storage modules as well as a simple microcontroller that uses separate control lines for every module. This controller is dropped in the second phase and does not need to be structurally identical to the real controller. Therefore, when modelling the target processor the designer only needs knowledge about the datapath and storage/register modules. This information can be taken for example from a processor data book, whereas information about the controller usually is not provided.

2. **Assembly instruction modelling:** The RT-level behavior of available assembly instructions is modelled in MIMOLA. The result is a "program" that simply consists of a listing of all assembly instruction behaviors. This "program" forms the software description for the first MSSC run.

3. **Micro-ROM generation:** The compiler MSSC is applied to the hardware description and to the "program" containing the assembly instruction behaviors. For technical reasons we assume every machine instruction to be executable within a single cycle. As described later, this means no restriction, however. Thus MSSC generates a microprogram in which each microinstruction corresponds to a realisation of a certain assembly instruction. The microprogram is stored into the declared instruction memory. Since this memory is not part of the real target hardware, it is a kind of "virtual" micro-ROM. Note that successful micro-ROM generation implies verifying the hardware description, i.e. the structure is able to execute the specified machine instructions. Therefore, "correctness by construction" is guaranteed. This feature of our approach may also be used to easily obtain a correct processor model for simulation purposes.

Phase 2 (Machine code generation)

1. **Controller replacement:** The micro-ROM generated in phase 1 is now assumed to be part of the target hardware structure. All control lines still radiate from the micro-ROM that simply serves as a decoder here. By addressing a line in the micro-ROM execution of a certain machine instruction can be selected. Addressing the micro-ROM is now done from the "real" machine instruction memory which in its turn is addressed by the "real" machine-level program counter. As mentioned in section II, MSSC is able to produce binary code for controlling modules via decoders. Since every module only can be controlled via the micro-ROM, and the micro-ROM only contains microcode for machine instructions, MSSC is forced to generate encoded machine instructions when applied to the structure and an HLL program.

2. **HLL program translation:** Now the same hardware structure as in phase 1 serves as an input to MSSC, extended by the micro-ROM. The software description in principle could be any HLL (PASCAL in our case) program. MSSC produces binary code in which every

instruction contains an address for the micro-ROM (and thus an encoded machine instruction) as well as necessary operands. This binary code can be easily transformed to real machine code using a table. The result is an assembly-level machine program for the target processor realising the given HLL program.

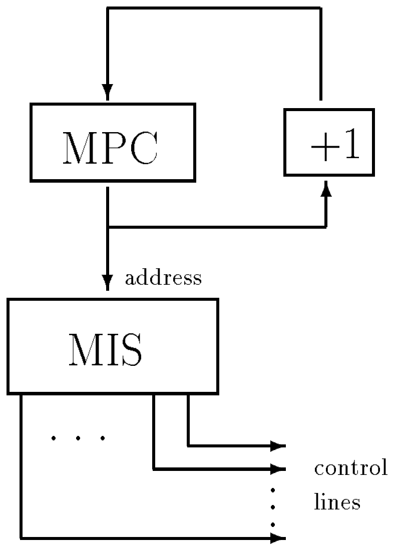

Figure 5: Simple Controller in Phase 1

For a given target processor, phase 1 has to be performed only once. After that any PASCAL program can be translated into machine code by a single call of MSSC. Both phases are described in detail in the two following sections. For better understanding of the bootstrapping technique, we consider the digital signal processor TMS320C25 as an example.

4 Micro-ROM Generation

In phase 1 a micro-ROM is to be generated, in which every control word realises exactly one machine instruction. At first the target structure (here: TMS320C25) has to be modelled. Most of the datapath structure can be found in [12] and can be written as a MIMOLA hardware description. Information about the internal controller structure is not available in [12], but this causes no problems since we only need a simple microcontroller structure for phase 1.
The MIMOLA model of the TMS contains about 2000 text lines. The model needs not to be exactly equal to the real hardware, the crucial point is that modelled hardware is able to perform the desired machine instructions. This simplifies the modelling phase, especially if only a rough datapath description is available to the user.
The TMS contains a 16 bit program counter and a 4k program ROM. These modules are modelled, too, but not according to their real functionality in the first phase. Instead we use the simple controller shown in fig. 5. The microinstruction storage (MIS) controls all but the residual control

modules directly. Its wordlength is 150 bits in our model. It is addressed by a microprogram counter (MPC) which is incremented after each cycle. MPC and incrementer are only used for micro-ROM generation, since MSSC always needs a program counter and at least an incrementer for compilation. Both modules are unimportant for the further steps.
The software input for MSSC is a "program" which simply enumerates all assembly instructions and their RT-level behavior (in MIMOLA). This information can also be taken from [12]. The "program" looks as follows:

```
PROGRAM InstructionSet IS
LABEL ADDK, CMPL, ...
BEGIN
 ADDK: (* add to accu short immediate *)
  PARBEGIN
   ACC := ACC + ZeroExtend24(PgmROM[PC].(7:0));
   PC := "INCR" PC;
  PAREND;
 CMPL: (* complement accumulator *)
  PARBEGIN
   ACC := "NOT" ACC;
   PC := "INCR" PC;
  PAREND;

 <further instructions>

END;
```

This extract shows how the behavior of two simple assembly instructions might be modelled in MIMOLA. For every instruction a label of the same name is declared. The ADDK instruction adds an 8 bit constant from the instruction word in the program ROM (addressed by the real program counter PC) extended by 24 zero bits to the accumulator and stores the result into the accumulator again. The PC is incremented in parallel. The CMPL instruction inverts the accumulator and can be modelled similarly.
This "program" is mapped onto the target structure by MSSC. It is never executed, only the resulting binary code is important. Since no branches occur, the incrementer (fig. 5) is sufficient for modifying the MPC. The generated microcode is stored into MIS, containing a sequence of 150 bit microinstructions then. Each microinstruction corresponds to a machine instruction (fig. 6). MIS contains as many lines as machine instructions have been specified, because a suitable hardware model guarantees that only single-cycle instructions are generated. The initialised MIS is used as one MSSC input in the second phase. It contains the information about available machine instructions and their implementation by microinstructions.

```
149            . . . . .            0
0100...........................0X11 (ADDK)
10X0...........................1101 (CMPL)
X011...........................0010 (LARP)
        <further instructions>
```

Figure 6: Contents of micro-ROM MIS

5 Machine Code Generation

In phase 2 a PASCAL program is to be translated into a TMS machine program. The micro-ROM is now assumed to be part of the target structure and the controller illustrated in fig. 7 is used. This step requires only minor changes in the RT-model.

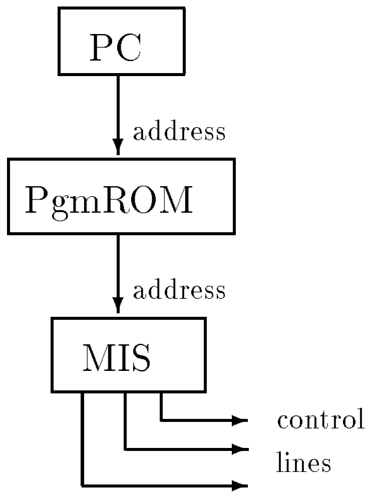

Figure 7: Controller in Phase 2

In the second phase the TMS 4k program ROM serves as instruction memory, addressed by the program counter PC. The microprogram counter MPC of phase 1 is dropped. The modules are controlled by the program ROM indirectly via the micro-ROM. Thus the micro-ROM now works as an instruction decoder. Every line in the program ROM has the following format:

16 bit constant	8 bit address	16 bit operands

The 16 bit constant field is only used in case of two-word instructions, e.g. jumps. The jump address is then stored in the constant field instead of the next program ROM line, just to avoid the necessity of generating two-cycle instructions. This means no restriction, since the above 40 bit control word

format is finally transformed to real machine code format by a very simple postprocessing step. An 8 bit address field is used for controlling the MIS. So a certain address corresponds to a certain machine instruction. The 16 bit operand fields carry immediate operands for the current instruction if necessary.
Since the program ROM controls the modules indirectly through MIS, and MIS in its turn only contains microcode for machine instructions, the code generator is forced to generate nothing but encoded machine instructions, whereas the hardware structure might be able to perform several other instructions, too.
The TMS RT-structure together with the MIS and an arbitrary PASCAL program now forms the input for MSSC. In addition, memory locations for user variables and temporaries can be declared. Code generation for a single PASCAL statement of the form

```
 a := b + 4;
```

could be done as follows. We assume the user variables a and b to be located at addresses 0 resp. 1 of the data RAM. At first MSSC recognises that a temporary is needed to execute the statement. Using the TMS accumulator as a temporary, the assignment is split into:

```
(1)  ACC := DataRAM[1];
(2)  ACC := ACC + 4;
(3)  DataRAM[0] := ACC;
```

Each statement can be allocated directly now, since there are corresponding microinstructions in MIS:

```
(1) ZALS (zero high accu, load low accu)
(2) ADDK (add to accu short immediate)
(3) SACL (store low accu with shift)
```

Assuming these instructions are located at addresses 1, 2 resp. 3 of MIS, MSSC will generate the intermediate code:

No.	16 bit const	8 bit addr	16 bit operands
(1)	xx...xx	00000001	xxxxxxxx00000001
(2)	xx...xx	00000010	xxxxxxxx00000100
(3)	xx...xx	00000011	xxxxx00000000000

The 16 bit constant fields are don't cares, since each instruction occupies only one TMS word. The 8 bit adress fields select the particular instructions in MIS (1, 2 and 3), and the 16 bit operand fields provide the instruction-specific operands: memory address 1 of variable b for the ZALS instruction, the 8 bit constant 4 for ADDK, and for SACL the shift value (here: 0) and the address

0 of variable a. This intermediate code can be transformed to real machine code or mnemonics very easily. Only a table is needed, containing the information about correspondence between addresses and instructions in MIS, and about operand field interpretation for each instruction. For the above example one obtains:

No.	Assembly Code	Machine Code
(1)	ZALS 1	0100000100000001
(2)	ADDK 4	1100110000000100
(3)	SACL 0	0110000000000000

Thus, we get a translation of a PASCAL program into real machine code, immediately executable on the TMS. As mentioned above, this compilation is retargetable, too, i.e. if the target structure is changed and a new micro-ROM is generated, machine-level output for other processsors or cores is produced. Therefore, several structural alternatives for software components in a codesign framework can be tried without adapting the compiler itself. Only the bootstrapping procedure has to be repeated.

Another important feature of our code generation technique is the exploitation of potential parallelism. Besides the central ALU the TMS320C25 contains an 8×16 bit auxiliary register file (AR) having its own arithmetic unit (ARAU) which is able to perform addition and subtraction on auxiliary registers. Many TMS instructions allow AR modification in parallel to the "main operation". Consider the statements

```
a := a + b;
i := i + 1;
```

After allocating these statements, the MSSC scheduler finds that both additions can be performed within a single control step. If variable `a` is temporarily located in the accumulator and `i` is kept in an auxiliary register, a single ADD ("add to accumulator") instruction is sufficient, that additionally supplies an increment control code to the ARAU. Thus, the scheduler is capable of code optimisation. Even a manually designed target-specific compiler might have difficulties to recognise this kind of potential parallelism.

6 Code Example

In this section we show an example for generated TMS assembly code. Phase 1 of the bootstrapping procedure has to be performed only once, in our model MSSC needs 135 CPU sec on a SPARCstation for that task. Regarding phase 2 we consider a small program for Euclidian greatest common divisor computation:

```
PROGRAM gcd IS
VAR u, v, t: Integer;
BEGIN
 REPEAT
  IF u < v THEN BEGIN
   t:=u; u:=v; v:=t
  END;
  u := u-v
 UNTIL u = 0;
END;
```

After termination of the REPEAT loop, variable `v` contains gcd(`u,v`). MSSC generates the following code for this example within 48 CPU sec on a SPARCstation (AR denotes TMS internal auxiliary register, `help` is a temporary located at `DataRAM[101]`):

```
  1:  ZALS    0         // ACC := u
  2:  SUBS    1         // ACC := ACC - v
  3:  SACL    101       // help := ACC
  4:  ZALS    101       // ACC := help
  5:  BGEZ    12        // IF ACC >= 0 GOTO 12
  6:  LAR     AR1,0     // AR1 := u
  7:  SAR     AR1,2     // t := u
  8:  LAR     AR1,1     // AR1 := v
  9:  SAR     AR1,0     // u := v
 10:  LAR     AR1,2     // AR1 := t
 11:  SAR     AR1,1     // v := t
 12:  ZALS    0         // ACC := u
 13:  SUBS    1         // ACC := ACC - v
 14:  SACL    101       // help := ACC
 15:  LAR     AR1,101   // AR1 := help
 16:  SAR     AR1,0     // u := AR1
 17:  ZALS    0         // ACC := u
 18:  BNZ     1         // IF ACC <> 0 GOTO 1
```

This code is not optimal, for example the lines 3 and 4 may be dropped. Those superfluous instructions arise from the fact, that MSSC does not yet include book-keeping of temporary locations beyond single statements. Also the compilation speed cannot compete with a commercial target-specific compiler, but that is the price for retargetability. However, the code suboptimality is not crucial within a codesign framework, since critical routines are assumed to be executed by ASICs. Important here is the ability to map software components onto a certain target structure without compiler redesign. Future versions of MSSC will include global book-keeping of temporary locations.

The next sections deal with applications of retargetable assembly code generation for hardware-software codesign of heterogeneous systems.

7 Hardware/Software Codesign for Heterogeneous Systems

Among typical DSP applications are image and speech processing, telecommunication, consumer electronics, medical equipment, as well as military domains like radar processing and missile guidance. Regarding DSP system components two classes of functions can be distinguished:

- *Real-time data processing functions:* The inputs are data streams that have to be processed with a certain rate (sample frequency or throughput). In most cases the sample frequency is part of the system specification. Therefore, any implementation has to meet given throughput constraints, whereas other system parameters (power dissipation, chip area) are subject to minimisation.

- *Control processing functions:* The inputs are variables which have to be processed at irregular points of time. Normally there are only few timing constraints, the main goal is to guarantee correct communication between the system and its environment.

A system comprising both kinds of functions is called a *heterogeneous system*. Its control processing functions form a shell, where its data processing functions are embedded in. This shell acts as an interface between the real-time components and the nondeterministic system environment. Examples for heterogeneous systems are the GSM Speech Coder [14] and the epsilon processor [15]. By means of increasing chip integration scales it has become possible to implement complete heterogeneous systems on a single chip. This is desirable especially for portable systems, where small physical volume and weight are required. Furthermore, single-chip implementations result in lower production costs. According to the two function classes mentioned above, these systems show a *heterogeneous architectural style*: a programmable DSP core with additional specific datapath elements for realising data processing functions, a controller, and a small amount of on-chip memory for storing programs and internal constants and variables.
Using a programmable core in combination with a controller allows late specification changes and implementing custom functions simply by reprogramming. Instead of allocating a separate controller it is more reasonable to execute both control and data processing functions by means of the DSP core, supported by accelerator datapaths. The heterogeneous architectural style resulting from that is depicted in fig. 8.
Due to this architectural style the designer has to make decisions regarding the right accelerator datapaths as well as the core itself and its instruction set. Finding suitable datapaths for meeting given timing restrictions and minimising other system parameters apparently is a problem of hardware/software codesign.
Codesigning heterogeneous systems poses some problems if we are looking for single-chip implementations as depicted in fig. 8. The DSP core together with the accelerator datapaths in a heterogeneous system can be regarded as one application specific processor whose instruction set

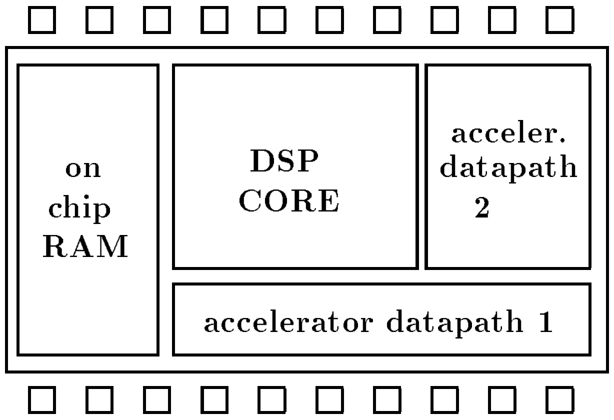

Figure 8: Heterogeneous Architectural Style

varies during the design cycles due to changing datapaths. For example, the designer could decide to use a hardware divider instead of performing division by sequential subtractions and thereby "moving" division from software to hardware. This measurement would result in establishing a new machine instruction DIV. Making use of this new instruction requires either manual compiler or manual machine code adaption. So we come to a slightly different view of the hardware/software codesign flow: Starting with a behavioral system description, the partitioning step decides which accelerator datapaths are to be implemented. Since we assumed the DSP core to perform (besides data processing functions) global system control, different datapaths implement different instruction sets. Hence, implementing the system means mapping its behavioral description onto the current instruction set within each iteration cycle. So the emphasis lies on the "software branch" of the codesign flow. Using a standard DSP core instead of a custom one within a heterogeneous system dramatically reduces design costs, but on the other hand requires code generation on assembly level instead of microcode level. Thus, the code generation technique explained above can be applied for studying performance effects of different accelerator datapaths. The following section gives examples of exploring architectural alternatives for the TMS processor.

8 Exploring Architectural Alternatives

We consider two examples for datapath changes: a hardware divider and an additional adder/accumulator unit. Although these datapath changes result in changing TMS instruction sets, the HLL system behavioral description can remain unchanged and can be mapped onto the new structure by our retargetable compiler. In order to recompile programs onto the new structure the designer simply repeats phase 1 of the bootstrapping technique, i.e. adding the new datapaths elements to the structural description, specifying the behavior of the resulting new machine instructions, and using MSSC for generating the corresponding new MIS.

Let us consider a system behavioral description that contains a division statement

```
 x := y DIV 27;
```

within a nested loop. Division is not directly supported by the TMS instruction set, but has to be implemented by repeated subtraction which requires a relatively large number of clock cycles. This might turn out to be a bottleneck in the implementation, that violates given timing restrictions. An apparent solution is placing a sufficiently fast hardware divider into the system. If there is not enough chip area, the designer can decide to drop some unnecessary processor modules and thereby some machine instructions, but this of course is also supported by retargetable code generation. Adding the hardware divider allows establishing a new machine instruction, let us say DIVK (divide accumulator by short immediate), whose RT-level behavior can be modelled by (see section IV):

```
DIVK:
 PARBEGIN
  ACC := ACC "DIV" ZeroExtend24(ROM[PC].(7:0));
  PC := "INCR" PC;
 PAREND;
```

This means, the accumulator is divided by an 8-bit immediate constant stored in the program ROM, addressed by `PC`, extended by 24 zero bits. In parallel, `PC` is incremented. Applying MSSC to the new structure, the above statement is translated into the following TMS instruction sequence (for sake of simplicity, we consider mnemonics instead of binary code)

```
 ZALS y    // ACC := y
 DIVK 27   // ACC := ACC DIV 27
 SACL x    // x := ACC
```

making use of the new DIVK instruction instead of repeated subtraction. So the HLL system behavioral description can be recompiled for the new structure and the performance gain can be evaluated immediately without manual compiler or code adaption. Of course, datapath changes require partial controller re-synthesis, but this has to be performed only once after performance evaluation has led to the final system implementation.

As a second example we consider the effects of placing a second adder/accumulator unit into the system. Fig. 9 shows the architectural extensions.
The new unit works in parallel to the original TMS ALU/accumulator unit, thereby allowing parallel additions and data RAM manipulations. By supplying separate control signals to the new modules and assuming the adder to have a transparent mode, five new machine instructions can be established, able to be executed in parallel to the normal TMS instructions:

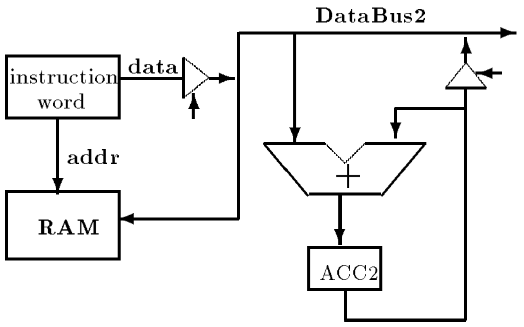

Figure 9: New adder/accumulator unit

LDA <memadr>: (*load ACC2*)
  ACC2 := DataRAM[<memadr>]
LDI <cnst>: (*load ACC2 immediate*)
  ACC2 := <cnst>
ADA <memadr>: (*add to ACC2*)
  ACC2 := ACC2 + DataRAM[<memadr>]
ADI <cnst>: (*add to ACC2 immediate*)
  ACC2 := ACC2 + <cnst>
STA <memadr>: (*store ACC2*)
  DataRAM[<memadr>] := ACC2

For instance, both additions within the two PASCAL statements

```
(1) a := b + c;
(2) d := d + 7;
```

can be performed within a single machine instruction combining the original TMS ADD instruction with the new ADI instruction. Of course, also other kinds of parallelisation are possible, e.g. storing `ACC2` with `STA` and in parallel jumping to a new program address. This parallelisation results in more compact code. We applied the original and the extended TMS structure for translating two benchmarks, well-known from High Level Synthesis: the *differential equation solver* and the *elliptical wave filter* [13]. Both are typical DSP applications, mainly consisting of additions and multiplications. We only mention the results here.

prog	PASCAL statements	original TMS	extended TMS	CPU sec
ellip	38	184	105	54
diffeq	35	100	73	64

Columns 3 and 4 show the number of machine instructions that have been generated for the original TMS instruction set and the extended instruction set allowing parallel additions. In this case the

additions had to be bound to the adder modules, but this also can be done automatically by a preprocessor. Work is in progress to overcome that limitation.
The above data indicate, that using the new adder/accumulator unit accelerates execution of *ellip* by 43 % and execution of *diffeq* by 27 %. When using MSSC combined with the two-phase bootstrapping technique, a designer can obtain this information rather quickly by simply adapting the hardware model and recompiling the programs.

9 Conclusions

A hardware/software codesign strategy for heterogeneous information processing systems built around a DSP core was presented. In order to meet given system timing restrictions, several iteration cycles for evaluating performance gains of different accelerator datapaths are required in general. Therefore, the system behavioral description has to be re-mapped onto different target structures. We presented a bootstrapping technique for retargetable machine-level code generation, based on a two-phase use of the microcode compiler MSSC. This overcomes a restriction of exixsting retargetable compilers and provides a valuable support for hardware/software codesign of heterogeneous systems, since retargetability allows fast evaluation of architectural alternatives concerning the target machine. No manual compiler or code adaption is required during design iterations, only the hardware model has to be changed. Code generation on the assembly level instead of the microcode level permits using standard DSP cores as the central component within a heterogeneous system, thus reducing design costs by a considerable amount. Code generation examples for the widely-spread DSP TMS320C25 with different accelerator datapaths indicate practical applicability of the new approach.

The authors would like to thank Gert Goossens from IMEC (Belgium) for providing valueable ideas on heterogeneous system modelling.

References

[1] R.K. Gupta, G. De Micheli: System-level Synthesis using Re-programmable Components, Proc. EDAC 1992, pp. 2-8

[2] R.A. Mueller, J. Varghese, V.H. Allan: Global Methods in the Flow Graph Approach to Retargetable microcode Generation, Proc. 17th Annual Microprogramming Workshop (MICRO-17), 1984, pp. 275-284

[3] S.R. Vegdahl: Local Code Generation and Compaction in Optimizing Microcode Compilers, PhD Thesis and Report CMUCS-82-153, Carnegie-Mellon-University, Pittsburgh, 1982

[4] T. Baba, H. Hagiwara: The MPG System: A Machine-Independent Efficient Microprogram Generator, IEEE Trans. Comp., Vol C-30, 6(1981), pp. 373-395

[5] Using and Porting GNU CC (V 2.4), Free Software Foundation, Cambridge, Massachusetts, 1993

[6] L. Nowak: Graph Based Retargetable Microcode Compilation in the MIMOLA Design System, Proc. 20th Annual Microprogramming Workshop (MICRO-20), 1987, pp. 126-132

[7] L. Nowak, P. Marwedel: Verification of Hardware Descriptions by Retargetable Code Generation, Proc. 26th Design Automation Conference, 1989, pp.441-447

[8] W. Schenk: A High Speed Prolog Implementation on a VLIW Processor, Microprocessing and Microprogramming Vol. 27, Nos. 1-5 (1989), pp. 601-606

[9] L. Nowak: SAMP: A General Purpose Processor Based on a Self-Timed VLIW Structure, ACM Comp. Arch. News, Vol. 15, No. 4, Sept. 1987, pp. 32-39

[10] P. Marwedel: The MIMOLA Design System: Tools for the Design of Digital Processors, Proc. 21st Design Automation Conference, 1984, pp. 587-593

[11] P. Marwedel: A new Synthesis Algorithm for the MIMOLA Software System, Proc. 23rd Design Automation Conference, 1986, pp. 271-277

[12] TMS320C2x User's Guide, Rev. B, Texas Instruments, 1990

[13] S.Y. Kung, H.J. Whitehouse, T. Khailath: VLSI and Modern Signal Processing, Prentice Hall, 1985

[14] V.Öwall, P.Andreani et al.: Custom DSP Implementation of a GSM Speech Coder, Proc. User Forum and EURO ASIC Prizes, EDAC/EUROASIC 1993, pp. 162–165

[15] C.Chu, M.Potkonjak et al.: HYPER: An Interactive Synthesis Environment for High Performance Real Time Applications, Proc. ICCD 1989, pp. 432–435