Retargetable Assembly Code Generation by Bootstrapping

Rainer Leupers, Wolfgang Schenk, Peter Marwedel

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

Abstract–In a hardware/software codesign environment compilers are needed that map software components of a partitioned system behavioral 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 in order to avoid manual compiler adaption for each 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 permitting to translate high-level language (HLL) programs into real machine-level code using a retargetable microcode compiler. Retargetability is preserved, permitting to compare different architectural alternatives in a codesign framework within relatively short time.

1 Introduction

The "hardware/software codesign" approach increasingly gains importance in digital system synthesis from behavioral descriptions. Codesign implies partitioning an abstract behavioral description into hardware and software components forming a system with the specified behavior and meeting given timing restrictions. Especially, it aims at designing digital controllers performing real-time computations. The target architecture might be a simple system containing a programmable processor (core), a main memory, and several ASICs, as proposed in [1] (fig. 1). Since communication overhead implied by a certain system partitioning is hardly predictable, codesigning a digital system requires several iteration steps in general. 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 partitioning 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 a 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 a 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.

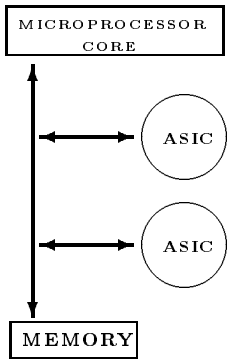

Figure 1: System Architecture

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 drawback of those compilers (when using it in a codesign environment) is that only code for the lowest programming level is generated, i.e. microinstructions. Generating machine code is not provided by the above compilers. If the target processor comprises a complex controller with a two-level interpretation scheme, modelling of the controller becomes quite difficult. Either it does not fit into a RT-level hardware model, or the internal controller structure is not publicly available at all. However, machine instructions have to be produced for the assembly level instead of the microcode level. This requires restricting the compiler to generate only available assembly instructions, whereas the processor datapath might expose more parallelism than visible at this level. To fill this gap in case of an unknown internal controller structure we apply a "bootstrapping" technique using MSSC in a two-phase mode.

We describe the bootstrapping approach using the TMS320C25 DSP as an example. The MSSC compiler has been described elsewhere [5, 6], 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). The paper ends with examples for generated TMS machine-level programs and a conclusion.

2 Microcode generation in the MDS

The retargetable microcode generator MSSC is part of the MIMOLA Design System (MDS), which supplies hardware synthesis, generation of self-test programs, simulation and schematics generation [7]. Each MDS tool is based on the MIMOLA language that allows both hardware and software descriptions [8]. 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 by enumeration of source and sink ports.

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 besides the bitstring. Other scalar types may be declared by the user. Definition of complex data types (ARRAY, RECORD) is supported as well. Hardware and software description together form the input to MSSC, that translates the given program into microinstructions for the given programmable hardware structure. MSSC has been described in detail in [5, 6], we only give a rough summary of the four main steps here.

1. **Program transformation:** The software description is transformed into a RT-level program. All user variables are mapped onto physical memory locations, and loop structures are replaced by conditional jumps. Either default or user-defined replacement rules are used. The result is a RT-level program that only may contain IF-statements as high-level elements. IF-statements can be mapped onto hardware directly using multiplexers and comparators.

2. **Preallocation:** The hardware structure is represented by the *Connection Operation Graph*. It contains vertices for every operation performed by the modules and edges for their interconnections. During preallocation suitable assignments to instruction word fields (*versions*) are calculated for each possible hardware operation. Module control codes can be allocated directly at the instruction word and hardwired constants or indirectly through decoders. The latter feature 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. Each 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 sequentialized. If a statement cannot be allocated even when using temporaries, MSSC generates an error message indicating the failure reason and location. The result of successful code generation is a list of allocated microoperations.

4. **Scheduling:** Finally the microoperations have to be packed into complete microinstructions (control store words). Data dependencies and compatibility of microoperations have to be obeyed. Microoperations executable in parallel are heuristically packed into one control step. Additionally unused registers and tristate bus drivers must be disabled.

The final result is a microprogram executing the given PASCAL program on the target structure.

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 pro-

cessor 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 a 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. 2.

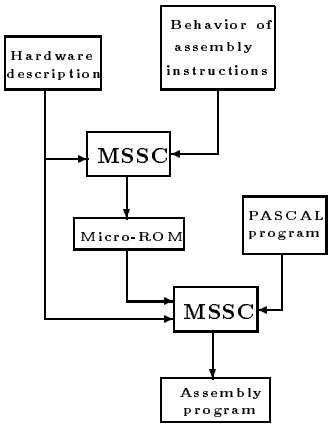

Figure 2: Basic Idea of Bootstrapping

Phase 1: (Micro-ROM generation)

1. **Hardware structure modelling:** The target processor's RT-structure is described in MIMOLA, containing the data path, storage modules as well as a simple microcontroller that uses separate control fields for each 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 user only needs knowledge about the data path 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 severe restriction, however. MSSC generates a microprogram in which each microinstruction corresponds to a realization of a certain assembly instruction. The microprogram is stored into the declared microinstruction memory.

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 start 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 2, MSSC is able to allocate constants 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 restricted to generate encoded machine instructions when applied to the structure and a 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 by table lookup. The result is an assembly-level machine program for the target processor realizing the given HLL program.

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.

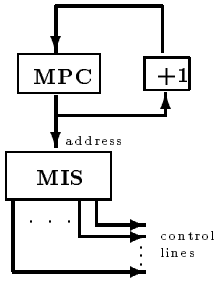

Figure 3: Simple Controller in Phase 1

4 Micro-ROM generation

In phase 1 a micro-ROM is to be generated, in which every control word realizes exactly one machine instruction. At first the target structure (here: TMS320C25) has to be modelled. Most of the data path structure can be found in [9] and can be written as a MIMOLA hardware description, consisting of 2000 text lines in this case. Information about the internal controller structure is not available in [9], but this causes no problems since we only need a simple microcontroller structure for phase 1.

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. 3. The microinstruction storage (MIS) controls all but the residually controlled modules directly. Its wordlength is 150 bits in our model. It is addressed by a microprogram counter (MPC) which is incremented after each cycle. The software input for MSSC is a "program" which simply lists all assembly instructions and their RT-level behavior (in MIMOLA). This information can also be taken from [9]. 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"

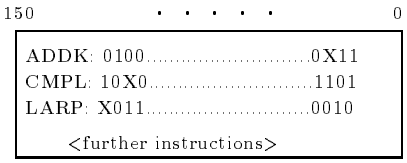

Figure 4: Contents of micro-ROM MIS

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. 3) 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. 4). 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 initialized MIS is used as one MSSC input in the second phase. It contains the information about available machine instructions and their implementation by microinstructions.

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. 5 is used. This step requires only minor changes in the RT-model. 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. Each 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

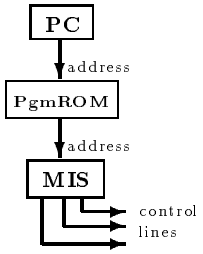

Figure 5: Controller in Phase 2

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 severe restriction, since the above 40 bit control word format is finally transformed to real machine code format by a very simple postprocessing step. Most of the TMS two-cycle instructions may be modelled this way. 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.

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;
```

proceeds 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 recognizes that a temporary is needed to execute the statement. Using the TMS accumulator as a temporary, the assignment is sequentialized:

```
(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 address 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.

6 Examples

In this section we show some examples 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 for that task. Regarding phase 2 we first 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. (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. 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.

Another example is the translation of the *elliptical wave filter*, a typical DSP application mainly consisting of arithmetical operations [10], into TMS code. Due to the limited space, we only mention the results here: MSSC generates 184 machine instructions for 38 PASCAL statements within 239 CPU sec.

7 Conclusions

We introduced a bootstrapping technique allowing retargetable machine code generation using a retargetable microcode compiler. This approach extends the range of target architectures which can be handled by retargetable compilers based on true structural hardware descriptions. In order to generate only valid assembly instructions, although no information about the internal controller structure is available, requires a large decoder to be integrated into the hardware model. The purpose of the bootstrapping technique is to generate the required decoder automatically. This is achieved by a two-phase use of the microcode generator MSSC. The feasibilty of this approach has been shown for a real-life example. Further DSP models are currently investigated. We plan to employ the MIMOLA high-level synthesis tool for obtaining the hardware model instead of using a manual specification.

Due to its retargetability the outlined approach cannot compete with target-specific compilers regarding compilation speed. Regarding the code quality, superfluous instructions might arise during code generation, but this is only due to a technical limitation of MSSC. Work is in progress to overcome that limitation. We predict that retargetable compilers will become an essential tool in hardware/software codesign. In this context, the disadvantage of lower compilation speed is more than compensated by retargetability.

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

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

[7] P. Marwedel, W. Schenk: Cooperation of Synthesis, Retargetable Code Generation and Test Generation in the MIMOLA Software System, Proc. EDAC/EUROASIC 1993, pp. 63-69

[8] R. Jöhnk, P. Marwedel: MIMOLA Reference Manual V 3.45, Technical Report No. 470, available from: University of Dortmund, LS Informatik 12, 44221 Dortmund, Germany

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

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