A New Approach to Assembly Software Retargeting for Microcontrollers

Ing-Jer Huang and Dao-Zhen Chen

Institute of Computer and Information Engineering

National Sun Yat-sen University

Kaohsiung, Taiwan

Republic of China

ijhuang@cie.nsysu.edu.tw

Abstract

A new approach is proposed to translate existing software programs from one instruction set to other instruction sets at the assembly level. The behaviors of instructions are abstractly represented as manipulation of the machine state. The behavior of each basic block of the software program is then represented as a pair of state transition. Instruction set retargeting is then modeled as the problem of finding sequences of instructions accomplishing the same machine state transitions at the target machine as does the software program at the source machine. The proposed approach has been successfully demonstrated on the software translation between several industrial microcontrollers.

1. Introduction

The fast progress of IC design technologies has offered many new microcontrollers/microprocessors with better code density, more functionality and lower power consumption to the markets of portable and consumer electronics products, where the issue of cost/functionality is more concerned than the issue of software compatibility. To replace the old microcontrollers with the new ones in the products often faces one challenge: in order to preserve the original software investment and to shorten the time-to-market, the existing software that has been developed under the old microcontrollers has to be retargeted (ported) to the new microcontrollers (if they have different instruction sets, which is usually true if better code density or more functionality is desired). However, unlike in the environment of desktop computers, a significant portion of software is usually developed directly at the assembly or binary level to optimize for the cost or performance of these products. In such case traditional retargetable compilers based on high level languages are of little help.

In this paper we present a new approach to the instruction set retargeting problem at the assembly level based on the *machine state transition* notation. This is motivated by the observation that the goal of retargeting is actually to look for an instruction sequence of the new instruction set that produces (emulates) the *same machine state* as the instruction sequence of the original instruction set. As long as the same machine state is reached, it does not matter whether the two instruction sequences come from the same grammatical expressions (or trees, graphs, *etc*.) or not. This observation is especially applicable to microcontroller-based embedded systems since the major purpose of the software running on them is to monitor and control the machine state (status). What the programmers care most is how the machine state transits, instead of the operations of the instruction set itself.

The machine state based retargeting approach has been successfully applied to solve a special version of the retargeting problem: constructing the x86-to-RISC instruction mapping table for an embedded RISC processor core [1]. The embedded core can be considered as an application specific instruction set processor (ASIP) with its sole application being to efficiently emulate x86 instructions. An x86 instruction is fetched into the core and then is decoded on the fly into simpler RISC instructions according to the mapping table for efficient execution. In this case, the retargeting is performed instruction by instruction.

The work presented in this paper is an extension of the above work by expanding the translation scope from individual instructions to entire software programs for microcontrollers. This extension makes it possible to perform instruction retargeting, code optimization and verification with the same machine state notation.

The rest of the paper is organized as follows. Section 2 reviews related work. Section 3 discusses the problem modeling. Section 4 presents the retargeting framework. Section 5 demonstrates the proposed technique with translation between three industrial 8-bit microcontrollers: PIC (RISC-like), MC8052 (CISC-like) and HT48100 (RISC-like).

2. Related Work

Most of retargeting compiler systems are based on the graph or tree matching algorithms. Aho *et al*. [2] propose an optimal dynamic-programming-based tree matching algorithm for retargeting compilers. Marwedel also presents a tree-based approach for mapping to a predefined hardware structure [4]. Corazao [6] proposes a matching method for DSP processors, based on templates of the CDFG's (control/data flow graphs) of instructions. Liem *et al*. [7] use rule-driven compilation. It has a shorter compilation time than those of pattern matching. Extensive reviews on the retargetable code generation theories and practices can be found in the book by Marwedel and Goossens [8].

These matching algorithms usually need high level source code in order to generate the necessary trees or graphs for matching. Therefore, they are not well suited for binary or assembly level retargeting because source code is not available.

Sites *et al*. [5] develop a binary translator that translates the VAX VMS and MIPS Ultrix binary code into DEC Alpha AXP and its execution environment. They build a translator called VEST and a run time environment called TIE. The translator maps the VAX code to AXP code according to a mapping table. When the translator encounters the portion of the VAX code that the translator is unable to distinguish whether it is the program or the data, the portion is embedded in the new AXP code. The VAX code embedded in the AXP code is executed by a run-time interpreter. Another similar work is Digital's FX!32 [3]. We are interested in how the mapping tables are constructed but

unfortunately these are not discussed in the papers. We guess that they are constructed manually since their target platforms are fixed and therefore a one-time, ad-hoc effort is sufficient.

C. Cifuentes cooperates with N. Ramsey to develop an integrated reverse engineering environment for binary code which is capable of translating binary programs from a given machine to a different machine [9]. The binary translation is achieved by three phases: a front-end to translate the binary representation to an intermediate representation, a middle-end to perform analysis and optimization and a back-end to map the intermediate representation to the binary representation of the target machine. The retargeting tool is built upon a syntax specification language SLED [10] and a semantics specification language SSL [11]. Our work is different from theirs in two aspects. First, their techniques are targeted at general purpose microprocessors, such as SPARC, x86 and PowerPC, while ours is targeted at application specific embedded system. Second, we do not rely on the typical intermediate representation, found in many retargeting research work, as the interface between two instruction sets. Instead, we abstract the source instructions into machine state transitions and then try to accomplish the same machine state transitions with the target machine instructions.

In our previous work [1], we have already proposed state notation to guide the instruction-to-instruction retarget, but have not considered dependency schedule issues. And the phase that performs state abstraction of basic block is not yet accomplished. In this paper, we consider the retarget of basic block, and extend the idea to application program retargeting.

3. Problem Modeling

3.1 Machine states

3.1.1 State Abstraction

A more effective way to study the behaviors of instructions is to observe their effects on the storage elements such as registers, flags, latches and memory, which together define the characteristics or the state of the entire machine. The right side of Figure 1 shows the machine state of the microarchitecture, shown at the left side of Figure 1. The ultimate objective of instructions is to manipulate the machine state; the operations (of the instructions) in the microarchitecture are just the means to accomplish such manipulation. Therefore, the machine state can serve as an abstraction for the machine. This state-based abstraction is more suitable to the instruction retargeting problem

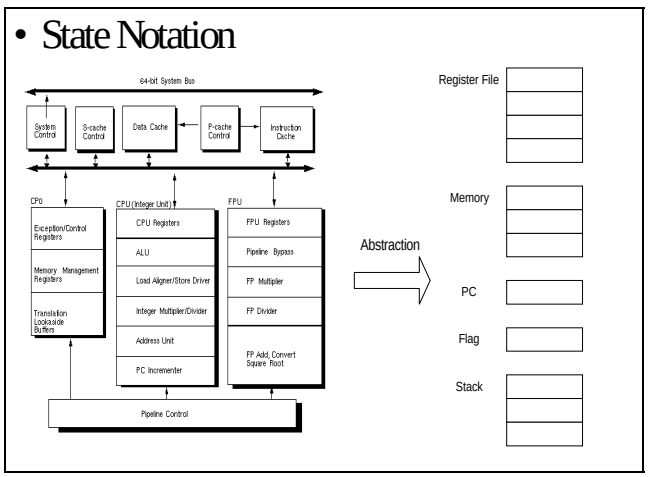

Figure 1. State Abstraction Concept

since observing how the machine state is modified by instructions requires less amount of information and efforts than observing the data path structure and detailed operations in the microarchitecture. In addition, the view point of machine state makes it easy to accommodate variations in the microarchitecture and the instruction set: two pieces of software code, although different in their contents or even in the instruction sets that they are based upon, can be regarded as compatible as long as they carry out the same state transition (from the same initial state to the same final state).

Based upon the above concept, we construct the machine *state* with a list of symbolic values of storage locations, called *contents*. The content of each storage location can be expressed as a binary tuple: `content(Location, Value)` where `Value` is the symbolic value of the storage element in `Location`. The storage location can be a special or general register, a memory word, a latch, an IO port, *etc*. The symbolic value can be a constant, a value from another location, or an expression comprising constants and values from some storage locations. For example, the instruction `add a, b, 1` (a = b + 1) is represented as `content(reg(a), reg(b)+immed(1))` which shows that the register location `reg(a)` gets the value of the register `reg(b)` plus the immediate value of one. Notice that register index may be physical or symbolic. In the latter case, register allocation is necessary to couple the retargeting process. Since a machine state may consist of numerous storage locations (e.g., a few giga-words of memory), only the locations those are modified or of a particular interest need to be explicitly specified.

The above binary tuple needs to be extended to support conditional states. In such case, the machine state is represented by the triple `content(Condition, TrueStateList, FalseStateList)` where `Condition` is a Boolean expression, and `TrueStateList` and `FalseStateList` are the machine state (i.e., a list of contents as defined previously) under the true and false conditions, respectively.

3.1.2 State Representation of Basic Blocks

The machine state representation of a basic block is derived by consolidating the machine states of individual instructions in the basic block. To consolidate the machine states, we take union of the machine states of individual instructions, and perform constant/expression propagation if some instructions reuse some parts of the machine state. For example, suppose a basic block has a simple instruction sequence `[mov a,b; mov c,a]`. After processing the first instruction, the machine state becomes `[content(reg(a),reg(b))]`. After processing the second instruction, the machine state becomes `[content(reg(a),reg(b)), content(reg(c),reg(b))]`. Note that original semantics of the second instruction copies the value of register `a` to `c`. However, since the register `a` is stored with the initial value of register `b` by the first instruction, so the final value of register `c` is represented as `reg(b)`, instead of `reg(a)`.

One of the advantages of the machine state representation is that while abstracting the basic block behavior into states, some code optimizations can be automatically (implicitly) achieved. Some examples are given below.

Eliminate architecture bias

Figure 2 shows the state representation of a simple basic block containing two instructions. The instructions are part of the instruction set of a two-operand architecture. The derived state specifies that the effect of the basic block is that register `a` gets the summation of registers `b` and `c`. Note that in the state

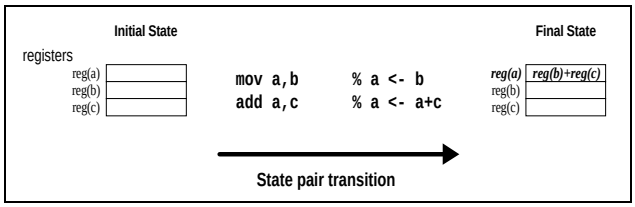

Figure 2. Implicit optimization with state abstraction (1)

representation the effect of the two-operand architecture disappears. When the basic block is translated into a three-operand architecture according to the state representation, it will be directly translated into one instruction such as the `add a,b,c` instruction. In this example, the architecture dependent feature (bias) is automatically eliminated without extra effort.

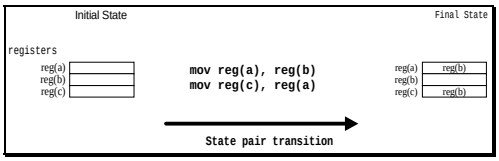

Figure 3. Implicit optimization with state abstraction (2)

Produce less dependent code

Consider the basic block in Figure 3 which copies the value of register `b` to register `a` first and then copies the value of register `a` to `c`. There is a data dependency on register `a`. However, when we abstract the basic block, it shows that the net effect is that the value of register `b` is duplicated to both register `a` and `c`. The data dependency is implicitly eliminated by the state abstraction process.

Produce Less redundant code

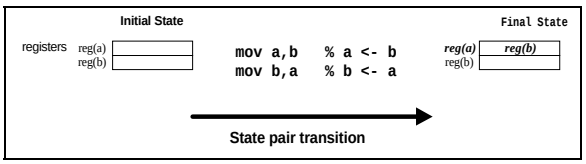

Figure 4. Implicit optimization with State Abstraction (3)

Redundant code can also be automatically removed by the state abstraction process. For example, the second instruction of assembly code in Figure 4 is a redundant instruction that does not create any new machine state. The corresponding state representation is simply `[content(a,b)]`, as shown in the figure. When this basic block is mapped to other instruction sets based on the derived state representation, only one instruction will be needed, instead of two instructions as in the original inefficient code.

3.2 Modeling the retargeting process

3.2.1 Retargeting as state transition

Figure 5 illustrates the state transition view of assembly programs on different instruction sets. Machine I executes three instructions `Op_X1`, `Op_X2`, and `Op_X3` to bring the initial state S_i to the final state S_j with S_{i1} and S_{i2} being the intermediate states. On the other hand, machine II executes two instructions `Op_Y1` and `Op_Y2` to bring the machine from the same initial state S_i to the same final state S_j with S_{j1} being the intermediate state. Although with different intermediate states, the instruction sequences {`Op_X1`, `Op_X2`, `Op_X3`} and {`Op_Y1`, `Op_Y2`} bring machine I and machine II, respectively, to the same final state, as long as they start from the same initial state. Therefore,

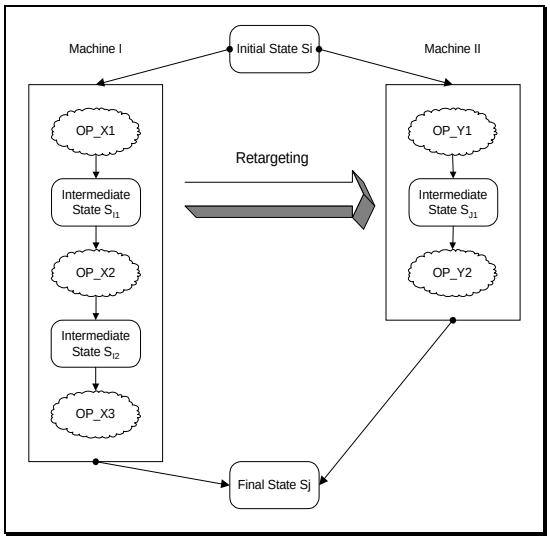

Figure 5. Retargeting process

the latter sequence can be regarded as the result of retargeting the former sequence from machine I to machine II, and vice versa.

3.2.2 Applying operators and creating the Intermediate States

Instructions are considered as operators which, when applied, change the state of the machine. Therefore, *the retargeting process is a process of selecting appropriate operators to bring the machine from an initial state to a final state*. Since both the operators (instructions of the target machine) and the basic blocks (of the original assembly program) are abstracted as states, the selection of the operators to be applied can be regarded as a matching process. In the current implementation, we select the appropriate operator with the following order:

1. Find the perfect match first. Both location and value are matched.
2. Find an operator that the location is matched and value is similar to the value part of instruction's.
3. Find an operator that the value matched and the location is similar.
4. Find an operator that location is matched and the value part is similar and is an expression

The operators selected by the rule 1 reduces the size of the state to be achieved (i.e., the problem state). The retargeting process is completed when the size of the problem state is empty. The operators selected by the rules 2 to 4 do not reduce the size of the problem state, but create some intermediate state to which other operators can be applicable. If there is no applicable operator, then the retargeting process halts.

3.2.3 Backward Chaining

We adopt the backward-chaining algorithm to solve the retargeting problem. A solution can be constructed backwards in the following way: first, select an operator whose post-condition best match the given final state; second, an intermediate state (a

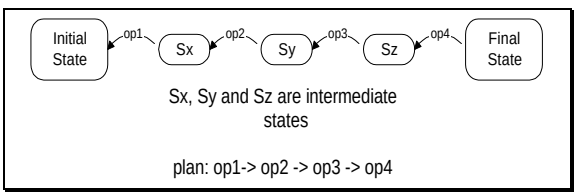

Figure 6. Backward Chaining

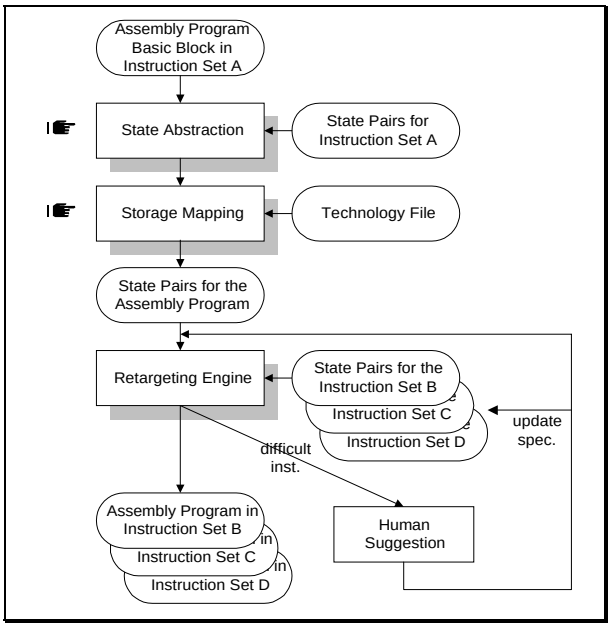

Figure 7. The retargeting framework

state closer to the initial state than the original final state) can be constructed by deleting the post-condition from and adding the pre-condition to the original final state; third, if the intermediate state is not equal to the initial state, then it serves as the new final (goal) state, and the plan construction is repeated. An example is depicted in Figure 6. State *Sz* is the result of applying operator *op4* backwards. In other words, applying *op4* to the state *Sz* can make a state transition to final state when we are in state *Sz*. State *Sy* is the result of applying operator *op3* backwards, and so on. After the searching, we obtain the solution in the sequence of *op4*, *op3*, *op2* and *op1*. The final solution is reversed and we get *op1*, *op2*, *op3* and *op4*.

4. The Retargeting framework

Figure 7 shows the flow graph of our retargeting system framework. The main retargeting phase includes state abstraction, storage/IO mapping, retargeting engine, and human suggestions. The main input files are application programs which want to be retargeted to other instruction set platform, instruction set specification of the source and target architecture, storage and IO mapping file. In the figure, it is assumed that the original assembly program is given in by the instruction set A. The assembly program is to be translated to instruction set B, C, D, *etc*. We will call instruction set A as the *source* instruction set, and the instruction sets B, C, D, *etc*., as the *target* instruction sets.

4.1 The translation flow

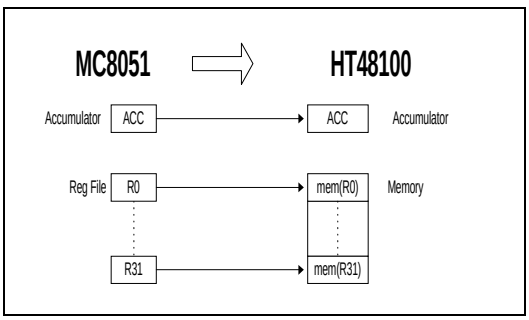

Figure 8. Storage mapping of MC8051 to HT48100

The retargeting granularity is based on the basic block boundary. It is assumed that the basic blocks in the assembly program have been clearly marked.

The first step, state abstraction, in the translation is to derive the state representation for each basic block. This step requires an instruction set specification file for the instruction set A. The specification is also based on the state representation.

Different processor architectures may have different organization of registers file, memory, or IO architecture. Therefore, a second step, storage mapping, in the translation is necessary to match the storage elements and their access methods between these architectures. For example, the registers in the MC8051 microcontroller has to be mapped into a certain memory locations in the HT48100 microcontroller, as shown in Figure 8.

After the previous pre-processing steps, the basic blocks are now ready for translation. The retargeting engine, implementing the algorithm in Section 3, translates each basic block into a sequence of instructions, based on the instruction set specification of the target instruction set, which is also represented with the state notation.

4.2 The feedback loop in the translation flow

We acknowledge that assembly program translation is not an easy thing to be automated. Therefore, we provide a manual feedback loop to the translation flow to take care of the difficult cases.

One difficult case is that there may be some instructions in the source instruction set for which the translation algorithm fails to find a solution with the target instruction set. Many of these instructions are control related. Efficient solutions to these kinds of instructions may involve some combination of more than one control flow instructions which our current algorithm fails to deal with.

For example, in MC8051, the `JNZ` is a conditional jump on the condition when zero flag is not true. On the other hand, in HT48100, there is no conditional jump instructions. But HT48100 provides a conditional skip instruction `SZ` which skips the following instruction when the zero flag is true. By combining the `SZ` and the `JUMP` (unconditional jump) instructions of HT48100, we can achieve the same control flow mechanism as the `JNZ` of MC8051, as shown in Figure 9. However, our current algorithm is not able to derive this combination since the basic block structure of the `[SZ;JUMP]` sequence is totally different from that of the `JNZ` instruction.

Once the solution (an instruction pattern) is manually found, it can be viewed as a powerful instruction and added into the instruction set specification for the target instruction set. Next time when the state of the source instruction is encountered, the algorithm automatically selects the known solution.

Understanding that assembly translation is by no means an easy problem, we do not expect that the entire program can be automatically translated. Instead, we aim to take care as much as possible of the portion of the program which can be

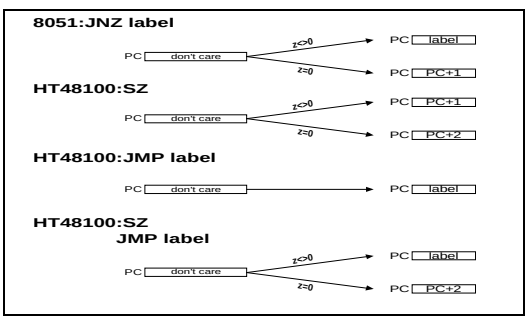

Figure 9. Instruction Pair for conditional jump

automatically translated, usually more than 85% of the entire program in our experience, so the designer can focus on manually translating the most challenging portion trough the feedback pass of the translation flow. Moreover, the manual derived solutions can be added into the target instruction set specification, so next time when the same situation is encountered, the algorithm will know how to deal with it. As more solutions are accumulated into the specification, less manual intervention will be required.

5. Experimental result

The proposed instruction set retargeting technique is demonstrated with three experiments. In Section 5.1, we show the inherent optimization benefit of the state abstraction of basic blocks (discussed in Section 3.1.2). In Section 5.2, we shows the experiment of mapping MC8051 (with a CISC-like instruction set) assembly program to HT48100 (with a RISC-like instruction set). And finally, in Section 5.3, we show that the retarget system can be easily adapted to perform mapping between different instruction sets, PIC (RISC-like), MC8051 and HT48100. All microcontrollers used in the experiments have 8-bit data path.

5.1 Example Mapping: HT48100 to HT48100 with Optimization

To illustrate the inherent optimization benefit of our state abstraction, we abstracted three basic blocks in HT48100 instruction set into machine states and then mapped the states back to HT48100 for comparison, as shown in Table 1. The first column lists the case number. The second column presents the original code. The third column shows the translated optimized code.

The first case shows that the redundancy of move data around is automatically removed during the retargeting process. The second case shows that the `ACC` is written twice and only the effect of second instruction remains, so the first instruction's effect is automatically eliminated. The third case shows that the true dependence relationship on `ACC` is removed and the code is

Case	Original code	Optimized code
1	Mov ACC, mem(m1) **Mov mem(m1), ACC**	Mov ACC, mem(m1)
2	**Mov ACC, mem(m1)** Mov ACC, mem(m2)	Mov ACC, mem(m2)
3	Mov **ACC**, immed(10) Mov mem(m1), **ACC**	Mov ACC, immed(10) Mov mem(m1), immed(10)

Table 1. Inherent optimization due to state abstraction removed and the code is improved after the retarget process.

5.2 Example: MC8051 (CISC-like) to HT48100 (RISC-like)

Table 2 shows an example of mapping an MC8051 keyboard-scan program to HT48100. The second column is the MC8051 program, partitioned into basic blocks. The third column is the mapping result of each basic block on HT48100 platform. The basic blocks 4, 6, 8, 11, 13, 14 and 16 contain MC8051 instructions with richer semantics and are thus mapped into larger number of HT48100 instructions.

There is a complex MC8051 instruction `CJNE` in the basic block 16. This instruction performs multi-way branching, depending on the result of comparison (greater, less, or equal). Multiple conditional states are necessary to model this instruction. Each state is mapped to several HT48100 instructions. We see that two labels `exit1` and `exit2` are automatically inserted to the target assembly program by our retargeting engine. This case demonstrates one of the challenges in assembly translation: the basic block structure might not be preserved during translation. The basic block 16 of the MC8051 assembly, containing seven instructions, is mapped into twenty-two HT48100 instructions, which correspond to ten basic blocks.

Basic Block ID	Processor : MC8051 (Original Code)	Processor:HT48100 (Translated Code)
1	***START:*** MOV SP,#30H	 mov mem_sp, 030h
2	***MAIN:*** CLR PSW.5	 Clr mem_psw.5
3	***S1:*** ACALL KSCAN	 Call kscan
4	**JNB PSW.5,S1**	**Snz mem_psw.5** **Jmp s3**
5	***S2:*** ACALL KSCAN	 Call kscan
6	**JB PSW.5,S2**	**Sz mem_psw.5** **Jmp s2**
7	MOV P1,A LJMP MAIN	mov mem_p1, a jmp main
8	***KSCAN:*** **JB PSW.5,S3**	 **Sz mem_psw.5** **Jmp s3**

9	MOV R1,#0FEH
	MOV R4,#04H
	MOV A,R1
	MOV R2,#00H
		Mov mem_r1, 0feh
		Mov mem_r4, 04h
		Mov a, mem_r1
		Mov mem_r2, 00h

10	***COLLUM:***
	MOV PP2,A
	MOV R3,#04H
	MOV A,PP2
	ANL A,#0F0H
	MOV 20H,A
		
		mov mem_pp2, a
		mov mem_r3, 04h
		mov a, mem_pp2
		and a, 0f0h
		mov mem_20h, a

11	***JUG:*** **JB ACC.4,NT1**	 **Sz [05H].4** **Jmp nt1**
12	MOV A,R2 SETB PSW.5 RET	Mov a, mem_r2 Set mem_psw.5 Ret
13	***NT1:*** INC R2 RR A **DJNZ R3,JUG**	 Inc mem_r2 rr [05H] dec mem_r3 **sz mem_r3** **jmp jug**

14	MOV A,R1
	RL A
	MOV R1,A
	DJNZ R4,COLLUM
		Mov a, mem_r1
		rl [05H]
		mov mem_r1, a
		dec mem_r4
		sz mem_r4
		jmp collum

15	RET	ret

16	***S3:***
	MOV A,#00H
	MOV P1,A
	MOV A,R1
	MOV PP2,A
	MOV A,PP2
	ANL A,#0F0H
	CJNE A,20H,S4
		
		mov a, 00h
		mov mem_p1, a
		mov a, mem_r1
		mov mem_pp2, a
		mov a, mem_pp2
		and a, 0f0h
		mov mem_tempA, a
		sub a, mem_20h
		mov a, mem_tempA
		sz reg(c)
		jmp exit1
		sz reg(z)
		jmp exit1
		jmp s4
		exit1:
		mov,mem_tempB, a
		sub a, mem_20h
		mov a, mem_tempB
		snz reg(c)
		jmp exit2
		sz reg(z)
		jmp exit2
		jmp s4
		exit2:

17	JMP S3	Jmp s3
18	***S4:*** CLR PSW.5 MOV A,R2 RET	 Clr mem_psw.5 Mov a, mem_r2 Ret

Table 2. Mapping from MC8051 assembly code to HT48100

code

5.3 Mapping between multiple instruction sets

We further conducted an experiment by mapping several assembly programs from PIC and MC8051 to HT48100. Since both PIC and HT48100 have the similar RISC architecture, the translated code has the same size as the original code. On the other hand, the code size gets expanded when a MC8051 (CISC-like) code is translated into HT48100 code.

Programs	Source Architecture	Source Assembly Lines	State Pairs	Target Architecture	Target Assembly Lines
1.LED display	PIC	25	27	HT48100	25
2.Random Number	PIC	33	35	HT48100	33
3.LED Light	PIC	37	43	HT48100	37
4.Keyboard-scan	MC8051	42	48	HT48100	65
5.LED display	MC8051	13	25	HT48100	25
6.TrafficControl Signal Light	MC8051	45	72	HT48100	63
7.Timer	MC8051	21	38	HT48100	45

Table 3. Mapping between PIC, MC8051 and HT48100

Since both PIC and HT48100 have a similar architecture, there is no need for human help; i.e., no feedback loop in the translation flow in Figure 7 is taken. On the other hand, some human intervention is necessary to help the translation of MC8051 to HT48100. After eight instruction patterns (as discussed in Section 4.2) are added into the target instruction set specification,

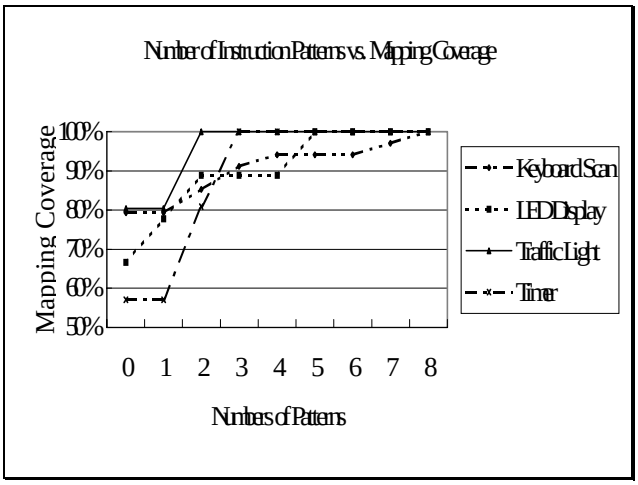

Figure 10. Number of Instruction Patterns vs. Mapping Coverage (MC8051 to HT48100)

all assembly programs can be completely (100%) automatically translated, as shown in Figure 10.

6. Conclusions

We have proposed a new approach to translate assembly programs between different microcontrollers. In the approach, the specifications of both source and target instruction sets are represented as machine state transitions. The assembly program to be translated is first divided into basic blocks. Each basic block is represented as machine state transition as well. The retargeting process is then modeled as selecting appropriate operators (instructions in the target instruction set) to bring target microcontroller from the same initial state to the same final state as the original program in the source microcontroller. The state notation serves as a useful canonic representation for both the instruction sets and the assembly programs. In addition, many code optimizations are implicitly achieved during the state abstraction process.

The proposed technique has been successfully demonstrated by translating various assembly programs between three industrial 8-bit microcontrollers, including both RISC and CISC styles.

In the future, we will investigate the automatic verification of the translation results based on the same state notation. In addition, transformation of the basic block structures is necessary to check if two groups of basic blocks produce the same machine state transition.

Reference

[1] W. F. Kao and I. J. Huang, "Instruction Retargeting Based on the State Pair Notation", *Asia Pacific Conference on Hardware Description Languages*, 1997, page 114-120.

[2] A.V. Aho, M. Ganapathi, S.W.K. Tjiang : "Code Generation Using Tree Matching and Dynamic Programming", *ACM Trans. On Programming Languages and Systems*, Vol11, No. 4, pp.491-516, Oct. 1989

[3] Anton Chernoff et al. : "FX!32 A Profile-Directed Binary Translator", *IEEE Micro*, pp.56-64, 1998.

[4] Peter Marwedel, "Tree-based Mapping of Algorithm to Predefined Structures", *International Conference on Computer-Aided Design*, pp.586-593, 1993

[5] Richard L. Sites et al., "Binary Translation", *Communication of the ACM*, Feb, pp. 69-81, 1993

[6] M. Corazao, M. Khalaf, L. Guerra, M. Potkonjak, J. Rabaey : "*Instruction Set Mapping for Performance Optimization*", Proc. of ICCAD, Nov. 1993

[7] Clifford Liem, Pierre Paulin, Marco Cornero, Ahmed Jerraya : "*Industrial Experience Using Rule-driven Retargetable Code Generation for Multimedia Applications*", TIMA Laboratory and Central R&D

[8] P. Marwedel and G. Goossens, "*Code Generation for Embedded Processors*", Kluwer Academic Publisher, 1995.

[9] C. Cifuentes, "Partial Automation of Integrated Reverse Engineering Environment of Binary Code," *Proceedings Third Working Conference on Reverse Engineering*, pp. 50-56, IEEE-CS Press, Nov. 1996.

[10] C. Cifuentes and S. Sendall, "Specifying the Semantics of Machine Instructions," Proceedings of the International Workshop on Program Comprehension, pp. 126-133, June 1998.

[11] N. Ramsey and M. Fernández, "Specifying Representations of Machine Instructions," *ACM Transactions on Programming Languages and Systems*, 19(3):492-524, May 1997.

[12] C.Monahan and F.Brewer: "Symbolic Modeling and Evaluation of Data Paths," Pcoc. ACM/IEEE 32nd DAC, June 1995.