Accepted to Journal of Design Automation for Embedded Systems, 1999.

A Machine State Transition Approach to Instruction Retargeting for Embedded Microprocessors[1]

Ing-Jer Huang and Wen-Fu Kao
Department of Computer Science and Engineering
National Sun Yat-Sen University
Kaohsiung, Taiwan
Republic of China

Tel: +886-7-525-2000 ext 4315

email: ijhuang@cse.nsysu.edu.tw

keywords: instruction set mapping, retargeting, embedded system, machine state transition, CISC instruction set, RISC instruction set, x86 instruction set

Abstract

This paper presents an interesting approach to retargeting existing software at the assembly (or binary) level from one instruction set to another instruction set. The approach is based on abstracting the instruction set behaviors as symbolic transitions of the machine states. The retargeting process is modeled as a planning process, an AI technique, that finds a plan (a sequence of operations) which brings the target processor from the same initial state to the same final state as the original software does on the source processor. The approach has been successfully applied in a design project of an x86 compatible microprocessor with an embedded internal RISC core for efficient execution. The proposed approach produced optimal x86-to-RISC mapping. In addition, the approach made it easy to keep up with microarchitecture revision during the design exploration phase since the mapping table can be automatically re-generated and re-evaluated promptly, which is difficult to achieve manually.

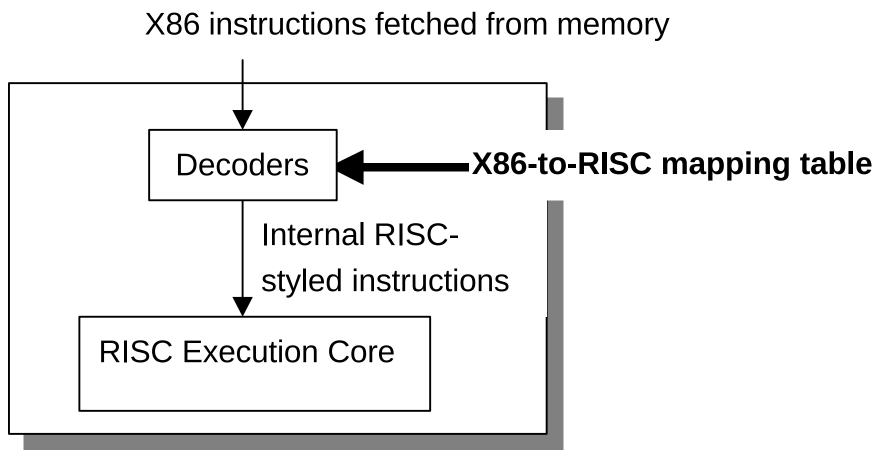
[1] This work is supported by NSC, R. O. C. under contact number 85-2262-E-009-010R.

1. Introduction

Instruction set retargeting becomes an important problem as the introduction rate of new microprocessors/microcontrollers becomes faster and the demand for better performance/cost tradeoffs becomes higher than ever before. Besides its applications in software porting such as [1], [16], [20], [26], *etc*., instruction set retargeting also plays an important role during the design phase of microprocessors/microcontrollers in which various architectures or microarchitectures are examined [1]. Fast adapting the instruction set to the new architecture and microarchitecture is the key to a successful design space exploration.

We participate in a large scale collaborative research project to develop an x86-compatible microprocessor. The microprocessor has an embedded RISC core with a superscalar implementation. To pursue performance, the microprocessor adopts a two-layered structure, as shown in Figure 1, which is similar to Intel's Pentium processor [13] or AMD's K5 processor [12]. The inner layer of the microprocessor is a RISC-based execution core for high speed execution. The outer layer of the microprocessor accepts the x86 instructions and translates them with hardware decoders into RISC-based instructions to be executed by the inner layer. There may be more than one decoder (only one is shown in the figure). Each decoder accepts one x86 instruction at a time. The translation is based upon an x86-to-RISC mapping table, which could be implemented as a ROM, in the decoder.

During the design exploration phase, fine tuning of the RISC instruction set and the execution core is inevitable, which results in frequent revision and re-evaluation of the x86-to-RISC instruction set translation. Due to the CISC nature of the x86 instruction set, the mapping table consists of hundreds of pages which are difficult to construct, debug and maintain manually (tedious, time-consuming and error-prone). Given a hardware modification, identifying which entries in the complicated mapping table and determining how to update the entries are by no means an easy task. Therefore, some automatic approach to the x86-to-RISC translation becomes crucial to the design process.


X86 instructions fetched from memory
Decoders
X86-to-RISC mapping table
Internal RISC-styled instructions
RISC Execution Core

X86 compatible microprocessor

Figure 1. CISC-to-RISC on-chip translation for a high performance X86 compatible microprocessor

In this paper we propose an automatic approach to the retargeting problem based on the notion of *machine state transition* (*pair*). The approach is based on two observations. First, 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 possess the same grammatical expressions (or trees, graphs, *etc*.) or not. Second, since the on-chip x86-to-RISC translation is done at the instruction level, the information available to us is how each instruction modifies the machine state. Therefore, representing the behavior of instructions as the transition of machine states is a more intuitive approach than graph/tree-based approaches in conventional retargeting compilers within our design context (i.e., constructing the mapping table for the decoder). Based on these observations, we have developed an instruction set retargeting tool, *StateMapper*, based on the concept of machine state transition, and applied this tool successfully to solve the x86-to-RISC decoder problem.

The rest of the paper is organized as follows. Section 2 reviews related work. Section 3 shows the problem modeling based on the machine state notation. Section 4 models the retargeting problem as a planning problem and provides the retargeting algorithm. Section 5 presents the application of the state-based retargeting approach to the x86-to-RISC mapping. Section 6 summarizes this paper and suggests future directions.

2. Related Work

Most of retargeting compiler systems are based on the graph or tree matching algorithms. Aho *et al.* [2] propose an optimal tree matching algorithm for retargeting compilers. This dynamic-programming-based algorithm for optimal code selection has been successfully implemented in the code generators for many different machines. Marwedel also presents a tree-based approach for mapping to a predefined hardware structure [17]. Corazao [3] proposes a matching method for DSP processors, based on templates of the CDFG's (control/data flow graphs) of instructions. Liem *et al*. [4][5] 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 [16]. 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*. [20] develop a binary translator that translate 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 [25]. The major difference between these two works and our work is that the binary translation takes place at the software level for the two related works, whereas at the hardware level for our work. The software-level approach is suitable for platform migration: to support the execution of existing software applications based on some legacy instruction sets, such as VAX, on a faster processor with an incompatible instruction set, such as DEC's Alpha. On the other hand, the hardware approach is not for the purpose of platform migration, but for the purpose of efficient execution of the existing instruction set. For example, the instruction set of the Pentium family processors remains the same as their predecessors such as the 486 processors. However, for better execution performance, a RISC core is embedded in the Pentium processor, and the fetched x86 instructions are internally translated by a hardware decoder into corresponding RISC instructions to be executed by the RISC core. The RISC instruction set is not observable or accessible by the outside world.

Cifuentes and Ramsey develop an integrated reverse engineering environment for binary code which is capable of translating binary programs from a given machine to a different machine [26]. 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 [28] and a semantics specification language SSL [27]. 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 more suitable for application specific embedded system. In the case of our work presented in this paper, the embedded RISC core can be considered as an application specific system since its sole task is to efficiently emulate x86 instructions. Second, we don't 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.

Focusing on the "state" of a system, instead of its operations, has been a successful approach to a wide range of software/hardware problems. The state notation is used as part of the semantics of flowchart programming languages, such as C or assembly programs, for the purpose of software program verification [29]. In [30], software program verification is conducted by analyzing the reachable states in a finite-state graph. Finite state models are also useful in formal hardware verification, such as the works reviewed in [31] for machine equivalence checking. Symbolic execution, based on machine state notation, is adopted to solve scheduling and binding problems in high level synthesis [32]. Bashford and Leupers [33] also use state-like model to represent data path operations in order to map data flow graph to DSP processors with irregular data paths. Holmer [7] views the instruction set design as a compaction problem. An interesting technique that he developed is the state pair notation, which is used to represent the benchmarks. We extend his state pair idea for our instruction set retargeting problem.

3. Problem modeling

As opposed to conventional approaches which view the instruction retargeting as manipulation/optimization on the expression trees, control/data flow graphs and the data path topology, we cast the instruction retargeting problem as searching for a sequence of operations that accomplish the same machine state transition as the original code to be retargeted. We first present the machine state model and then apply a well-known AI technique, planning, to solve the instruction set retargeting problem based on the proposed model.

Through the rest of the paper we will assume that the program structure, i.e., program segments, data segments and basic block boundaries, of the source code are known before the retargeting taking place. Under this assumption, binary code and assembly code can be considered equivalent and the two terms will be used interchangeably whichever is more appropriate within the discussion context.

In addition, we assume that before the retargeting process can be started, a pre-processing step called storage I/O mapping has been conducted for instruction set architectures with different organizations in memory, registers and I/O. For example, to translate the MIPS assembly code into Intel x86 assembly code, one needs to assign some of the MIPS's thirty two general-purpose registers to the limited number of general-purpose registers (less than eight) of the x86 microprocessor, and assign the rest of the general-purpose registers to some x86 memory locations. Currently the mapping is performed manually and will be automated in our continuing work.

3.1 Machine abstraction: machine state

A typical first step towards instruction retargeting, as found in some work in Section 2, is to study the behaviors of instructions by observing the microarchitecture and the corresponding operations within the data path modules in the microarchitecture. However, such study involves a great amount of information which are more than necessary to the purpose of instruction retargeting. In addition, such approach may also lead to the construction of retargeting tools with less degree of flexibility since the model of the underlying microarchitecture may unnecessarily constrain the retargeting power.

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 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 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, `content(reg(a), reg(b)+immed(1))` 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 that 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.

3.2 Representation of instructions (operations): state pair

Based on the machine state notation, the operation of an instruction can then be represented with a pair of an initial state and a final state, called the *state pair*, with the notation `pair(InitialState, FinalState)` where `InitialState` is the necessary machine state (*pre-condition*) for the instruction to operate, and `FinalState` is the machine state after the execution of the instruction (*post-condition*). Both `InitialState` and `FinalState` are represented as lists of the content data structure defined in the previous section. For example, the operation of the x86 instruction `PUSH ax` can be represented as the following state pair:

```
pair([], [content(mem(reg(ss):reg(esp)),reg(ax)),
          content(reg(esp),reg(esp)-immed(2))]).
```

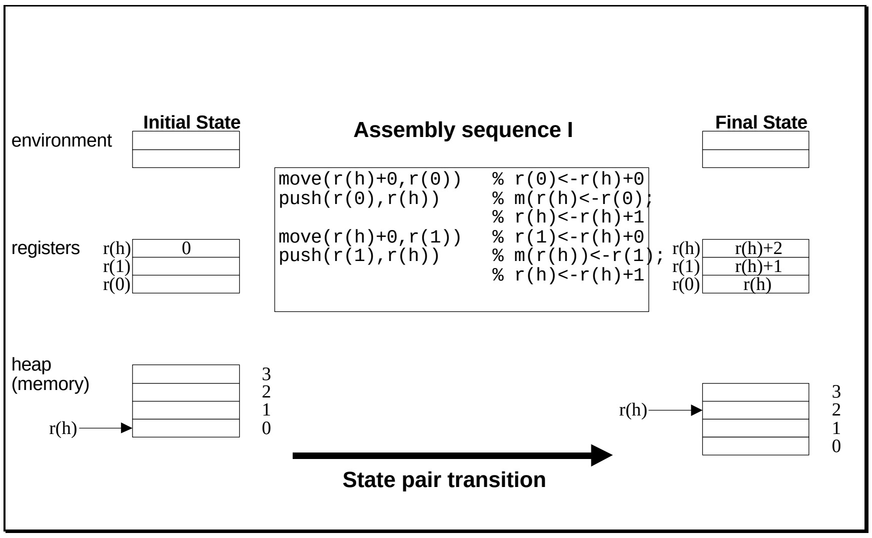
The above representation specifies that the `PUSH` instruction does not require any pre-condition, as specified by the empty list, and the effects of the instruction are that the memory location specified by the registers `reg(ss)` and `reg(esp)` gets the value of the register `reg(ax)`, and the register location `reg(esp)` gets the value of its initial value minus two.

As an another example, the x86 instruction `JZ label` is a conditional branching instruction. It jumps to the location `label` if the flag `ZF` is set. The content of the program counter `eip` is conditionally set. If `ZF` is 1, `eip` is set to `label`, otherwise it is sequentially incremented to the address of the next instruction. Since incrementing the program counter sequentially to the address of the next instruction is considered as a default behavior of a basic microprocessor, the increment operation is implicitly implied and does not appear in our state pair notation. With the above analysis, the representation of the x86 conditional branching instruction is as the following:

```
pair([], content(reg(zf)=immed(1)), [content(reg(eip),immed(label1))], []).
```

Note that in most cases, InitialState is an empty list since most instructions do not require any special initial condition for the instruction to operate.

The state pair notation can be applied to a larger scope such as a basic block in the assembly program. Each basic block of the program can be represented by a pair of machine states as well. The final state is the machine state after executing the basic block on the initial state. Figure 2 gives an example of the state pair notation for a basic block from a Prolog benchmark. The basic block, called *sequence I*, computes data, store them to the registers, and pushes them to the heap in the memory, using `r(h)` as the heap pointer.



```
move(r(h)+0,r(0))   % r(0)<-r(h)+0
push(r(0),r(h))     % m(r(h)<-r(0);
                    % r(h)<-r(h)+1
move(r(h)+0,r(1))   % r(1)<-r(h)+0
push(r(1),r(h))     % m(r(h))<-r(1);
                    % r(h)<-r(h)+1
```

Figure 2. State pair notation for a basic block (Prolog assembly sequence I)

The state pairs of basic blocks can further connected together according to the control flow of the basic blocks. The connected state pairs become a compact representation of an assembly program: a state transition graph. With such insight, a software program can be thought as a sequence of operations that modifies the machine states of an instruction set processor in a well-controlled way. Therefore, *programming can be viewed as finding a sequence of instructions (operations) that brings the machine state from the given initial state to the desired final state*.

By representing the assembly program as state transitions, we are able to eliminate possible inefficiencies embedded in the original operations in the program. For example, the piece of code in Figure 2 is generated by the Aquarius Prolog compiler for a high performance RISC microprocessor [6]. Although the code has been optimal for uni-processor architectures, it is not well-suited for superscalar or superpipelined architectures for the following reasons. First, the compiler creates spurious data dependencies for the operations on the heap pointer `r(h)`: the creation of the heap is serialized by incrementing the heap pointer from zero to two by two steps. Second, the serialization of the heap operations causes extra ALU operations, which dissipates more power than necessary. Fortunately, the inefficiencies are automatically removed in the final state representation, as shown in Figure 2. The inefficiencies are not unavoidable; they are just the consequence of an inefficient solution (implementation) to accomplish the desired computation specified by the final state.

The state transition notation focuses on *what* the programmer really wants to accomplish, instead of *how* the programmer accomplishes the desired computation. Therefore, the programmer will not be biased or limited by the possible inefficiencies embedded in the original program. For example, by abstracting the assembly sequence I into its state notation and then observing the state transition, it is easy to come up with better assembly sequences to achieve the same computation, such as the sequences II and III in Figure 3. These sequences use other instructions that accomplish the same state transition as does the sequence I in Figure 2 but require less number of instructions.

Assembly sequence II

```
add_st(r(h),0,r(0))        %r(0)<-r(h)+0;
                           %m(r(h)+0)<-r(h)+0;
add_st(r(h),1,r(1))        %r(1)<-r(h)+1;
                           %m(r(h)+1)<-r(h)+1;
add(r(h),2,r(h))           %r(h)<-r(h)+2
```

Assembly sequence III

```
mv_st(r(h),r(0))           %r(0)<-r(h);
                           %m(r(h))<-r(h);
add_st(r(h),1,r(1))        %r(1)<-r(h)+1;
                           %m(r(h)+1)<-r(h)+1;
add(r(h),2,r(h))           %r(h)<-r(h)+2
```

Figure 3. Alternative sequences that perform the same computation as the sequence I

3.3 Instruction set retargeting as a planning problem

Figure 4 illustrates the state transition view of assembly programs on different instruction set processors. 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, the latter sequence can be regarded as the result of retargeting the former sequence from machine I to machine II, and vice versa.

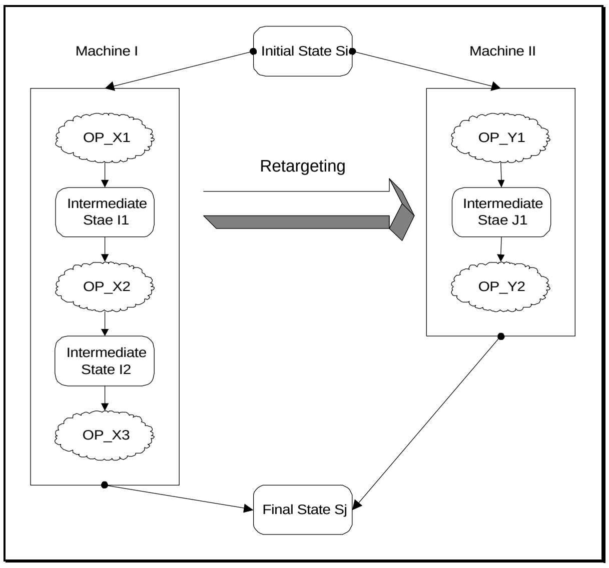


Figure 4. Retargeting process

Figure 5 shows the intermediate states of the assembly sequence I in Figure 2. Each state transition is caused by the execution of one instruction. There are four instructions in assembly sequence I and three intermediate (sub figures b, c, and d) are created during the transition.

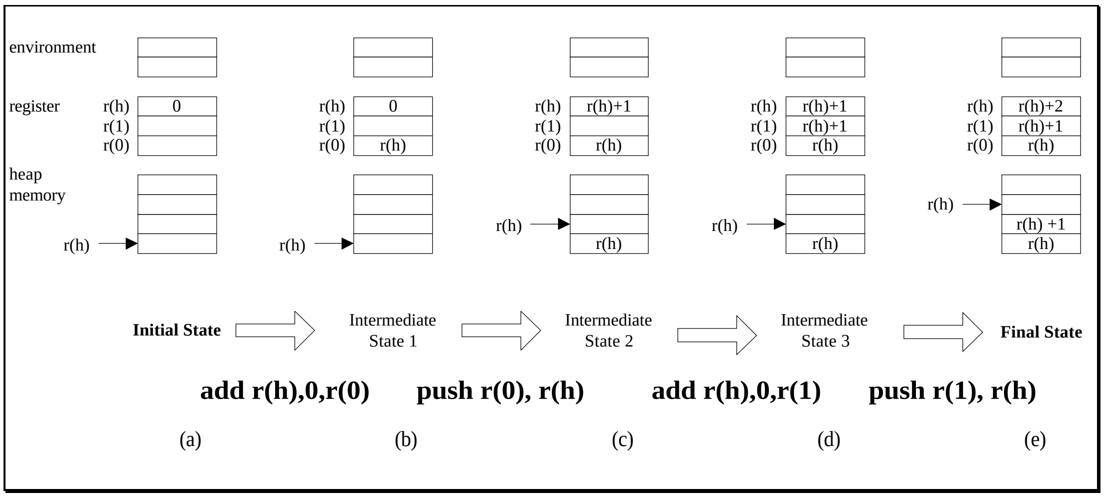


Figure 5. Intermediate states of the assembly sequence I in Figure 2

Based on the model described in the previous section, the instruction set retargeting can be casted as a *planning* problem [8][10], which is an important problem solving technique in artificial intelligence. A planning system composes of a problem world and a set of operators which change the problem world. The problem world, represented as *states*, is an abstraction of the problem to be solved with appropriate level of details. Since the description of the complete state may be tedious, it is a usual practice to explicitly describe only the portion of the complete state which is of interest. The problem is then given as a state pair: the *initial* state and the *final* (goal) state. The problem world is manipulated by *operators*. The operators change the states. A *plan* is sequence of operators that brings the world from the initial state to the specified final state. *Planning* means to search for a plan that accomplishes the desired state transition and also satisfies given constraints.

Casting the instruction set retargeting problem as a planning problem is straightforward: the state pair of a desired computation on the source platform, which could be either an instruction or a basic block, is the problem world, while the instructions of the target platform serve as the candidate operators, among which the planning engine searches for a best plan under given constraints. The constraints are modeled in the cost function that guides the search process.

Table 1 lists the state pair representations for some instructions (operators) for the examples in Figure 2 and Figure 3. The initial states for the operators in the table are "don't case", since there is no pre-condition for these operators. The final states explicitly specify the storage locations of which the values are modified by the operators. The capitalized terms in both the specifications of the operators and the final states denote operand templates that will be bound to specific (grounded) values such as physical or symbolic address locations, register index or constants. The binding is performed by the `match` procedure of the retargeting engine, to be presented in Section 4.2. For example, if the templates `R1` and `R2` for the `move` operator are bound to `ax` and `bx` respectively, then the move operator is instantiated to be a `mov r(ax),r(bx)` operation.

In addition, operators are associated with costs. The table lists two costs: the cycle count and power consumption. The cost information is useful in selecting operators to apply to the world.

Operator	Initial State (location, value)	Final State (location, value)	Cost1 cycle count	Cost2 Power
`move r(R1),r(R2)`	`don’t care`	`(r(R2),r(R1))`	1	1

`push r(R1),r(R2)`	`don’t care`	`(m(r(R2)),r(R1),`
		`(r(R2),r(R2)+1)`
			2	3

`add r(R1),Imm,r(R2)`	`don’t care`	`(r(R2),r(R1)+Imm)`	1	1

`add_st r(R1),Imm,r(R2)`	`don’t care`	`(r(R2),r(R1)+Imm)`
		`(m(r(R1)+Imm),r(R1)+Imm)`
			2	1.5

`mv_st r(R1),r(R2)`	`don’t care`	`(r(R2),r(R1))`
		`(m(r(R1)),r(R1))`
			2	1.5

Table 1. Example of the state pair notation and costs for instruction operators

4. The retargeting (planning) engine

Given the state pair of the desired computation (the problem world) and the state pairs of the instructions of the target processor (operators), the retargeting (planning) engine searches for the best instruction sequence (plan) that satisfies the given constraints (cost function). In this section we describe the search method and the actual algorithm of the retargeting engine.

4.1 The search method: backward chaining

We adopt the backward-chaining algorithm to solve the retargeting problem. A plan 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 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. Figure 6 illustrates the idea of backward-chaining. 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 the final state. State *Sy* is the result of applying operator *op3* backwards, and so on. Once the initial state is reached, the search process is terminated. The operator sequence *op1*, *op2*, *op3* and *op4* is a plan that brings the problem world from the given initial state to the desired final state. As an example, the sub figures in Figure 5, viewed from right to left, reveal the backward chaining process and the resulted plan is the assembly sequence I in Figure 2.

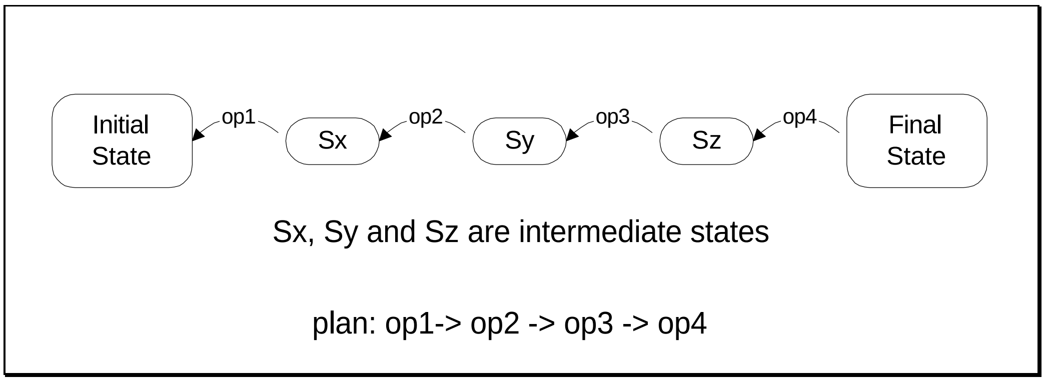

Initial State
op1
Sx
op2
Sy
op3
Sz
op4
Final State
Sx, Sy and Sz are intermediate states
plan: op1-> op2 -> op3 -> op4

Figure 6. Backward Chaining

4.2 The algorithm

We now present the algorithm of the retargeting engine, as listed in Listing 1. The algorithm takes in as inputs the `InitialStateList` and `GoalStateList`, which are the lists of contents (the data structure described in Section 3.1) of the initial and goal (final) machine states, respectively. When finished, the algorithm returns the plan (instruction sequence) in `Solution`. The main actions of the steps in the algorithm are described as the following:

1. The algorithm first checks in line 3 if the goal state list is equal to the initial state list. If yes, terminates planning, reverses the operator order in the solution and returns it; otherwise, continue;

2. Some contents in the goal state may have dependency relationships, which prevent certain contents from being processed before others. Therefore, the algorithm constructs a dependency graph for the contents in line 8;

3. In line 10, select a ready content from the goal state list, according to the dependency relationships. Since the retargeting engine works backwards, the ready content actually is a sink node in the dependency graph (i.e., without any outward arc). The subroutine `match` is then invoked to choose an operator whose state best matches the selected content. The chosen operator is added into the solution.

4. If the chosen operators require a pre-condition that does not exist in the machine state, new contents are created for the pre-condition and appended to the rest of the goal state list, as in line 11. We will explain this issue later in this section.

5. Recursively apply the planning engine to the updated goal state list, as in line 12.

The procedure `match` is listed in lines 15 through 25. It picks up a viable operator and matches the state pair of the operator to the target content. During this step the operand templates of the operator are bound to the specific physical or symbolic values in the target content. If the cost of the operator is favorable by the cost function, the operator is accepted; otherwise, other operators are tried. A tree-based matching mechanism is adopted in our current implementation. The tree-based approach is easy to be implemented and is sufficient to our x86 problem (described in Section 5) since the mapping targets is individual x86 instructions. As the mapping targets become more complicated, such as entire binary programs, more sophisticated approaches such as graph covering would have to be implemented.

Note that new contents may be created, if the pre-conditions of the chosen operator requires them. The pre-conditions include explicit ones and implicit ones. The *explicit pre-conditions* are the conditions that are explicitly stated in the pre-condition field of the content data structure. On the other hand, the *implicit conditions* are generated by the retargeting engine while performing retargeting between architectures with different memory or register organizations, which will be explained shortly with the example in Figure 8.

The optimal solution can be found with two approaches. First, as shown in Listing 1, we can exhaustively search all possible solutions with a depth-first search, and use the current best solution to prune inferior partial solutions during the search process (as in line 20). This approach has been effective enough for our x86 mapping problem. Second, not shown in the listing, we can conduct a successive-

```
 1: planning(InitialStateList, GoalStateList, Solution)
 2: begin
 3: if GoalStateList=InitialStateList
 4: then
 5:     reverse the Solution list and return
 6: else
 7: begin
 8:     construct the dependency graph for the contents in GoalStateList
 9:      select a ready content from the state list
10:      call match to select an operator to solve the content,
         append the selected operator to the solution;
11:     append any new contents produced by match to the rest of GoalStateList
12:     planning(UpdatedGoalStateList,InitialStateList,UpdatedSolution)
13:end
14:end
15:match(content(Loc,Val),NewGoals,Operator)
16:begin
17:     pick an operator from the target operators
18:
19:     match the state pair of the operator with content(Loc,Val)
20:     if current_cost+operator’s cost > cost limit
21:      then
22:         pick next operator
23:     else
24:         produce necessary new contents (demanded by some operators)
25:end
```

Listing 1. The planning algorithm

deepening search in which we first try to find all solutions with *i* (initially *i* starts from one) operators; if no solution can be found, then try to find solutions with *i*+1 operators, and so on.

To further speed up the search process, we rank the operators with the following order when picking a candidate operators in line 17:

1. operators of which both the location and the value can be matched to the given content;
2. operators of which the location can be matched and the value is similar to the value of the given content.
3. operators of which the value is matched and the location is similar;
4. operators of which the location is matched and the value is similar and is an expression;
5. the rest of the candidate operators.

We illustrate the retargeting engine with the example in Figure 5. The engine starts from the sub figure (e), which is the goal state. The candidate operators are defined in Table 1. To simplify the illustration, we don’t consider the cost in this example. The engine finds the `push` operator, with the operand templates `R1` and `R2` being bound to `1` and `h`, respectively. After the operator is matched to the goal state, the intermediate state 3 in the sub figure (d) is obtained, which serves as the new goal state for the next iteration of the algorithm. The intermediate state 3 is closer to the initial state than the original goal state in sub figure (e): the intermediate state 3 has one less memory content defined (i.e., `m[r(h)+1]` is removed) and the value of the register `r(h)` is closer to that of the initial state. Recursively, the engine works on the intermediate state 3 and finds the `add` operator, with the bindings of `R1=h`, `Imm=0`, and `R2=1`, which produces the updated goal state in sub figure (c). The same process repeats until the updated goal state is equivalent to the initial goal state. Finally, by reverse the order of the chosen operators we obtain a plan that is exactly the assembly sequence I in Figure 2. By applying different selection strategies (cost functions), the assembly sequences II and III in Figure 3 can be found as well.

Next we demonstrate a retargeting case with implicit pre-conditions, using the example in Figure 7, which is a state transition of an x86 code. This code is retargeted to RISC instruction set, as shown in Figure 8. The final state of the code indicates that the top of the stack, which is also in the memory, gets the value of the memory location 1234. However, there is no instruction in the RISC instruction set that can perform a direct memory to memory copy. The closest one is the store instruction that copies a register value to memory. However, in order to select this instruction, the retargeting engine has to assume that the value of the memory location 1234 resides in some register `tmp`. With this assumption, the operator `st tmp,m[ss:es]` can be applied, as shown between sub figures (c) and (d). In addition, the engine has to create a new content for such assumption (an implicit pre-condition), shown as the new register `tmp` in the intermediate state 2 in the sub figure (c). The engine continues to work on the intermeidate state 2 and finds the operator `ld tmp,m[1234]` to load the data in the memory location 1234 to the register `tmp`, resulting in the new intermediate state 1. Finally, the engine finds the subtraction operator `subi esp,2` to bring the intermediate state 1 to the initial state and finishes the retargeting. By reversing the order of the chosen operators, the following RISC instruction sequence is obtained.

```
subi esp,2          % reg(esp) <- reg(esp) - 2
ld tmp,m[1234]      % reg(tmp) <- m[1234]
st tmp,m[ss:esp]    % m[reg(ss):reg(esp)] <- reg(tmp)
```

Note that allocation and optimization of the temporary registers, required for the pre-conditions, has been an important research topic on its own. We adopt a simple heuristics to solve this problem in our current implementation: reuse the temporary registers as much as possible. Fortunately, our experiments show that at most two temporary registers are needed for the specific x86-to-RISC mapping problem discussed in this paper. There are plenty of available registers in the register file of the RISC core to serve such purpose. More comprehensive approaches will be adopted in our future work.

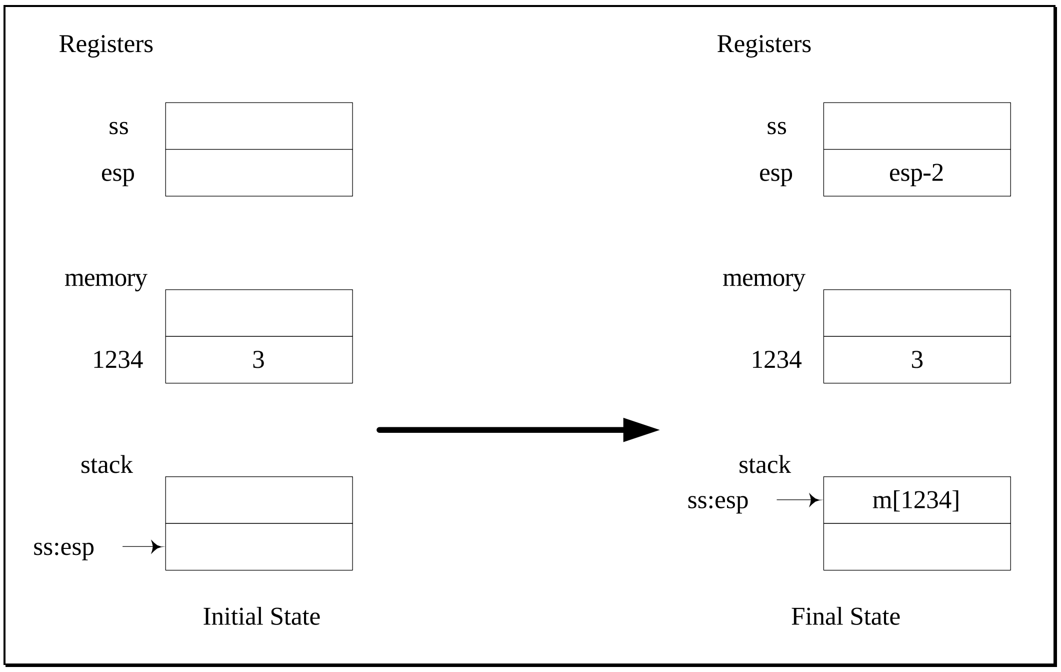


Figure 7. Another example of the retargeting process

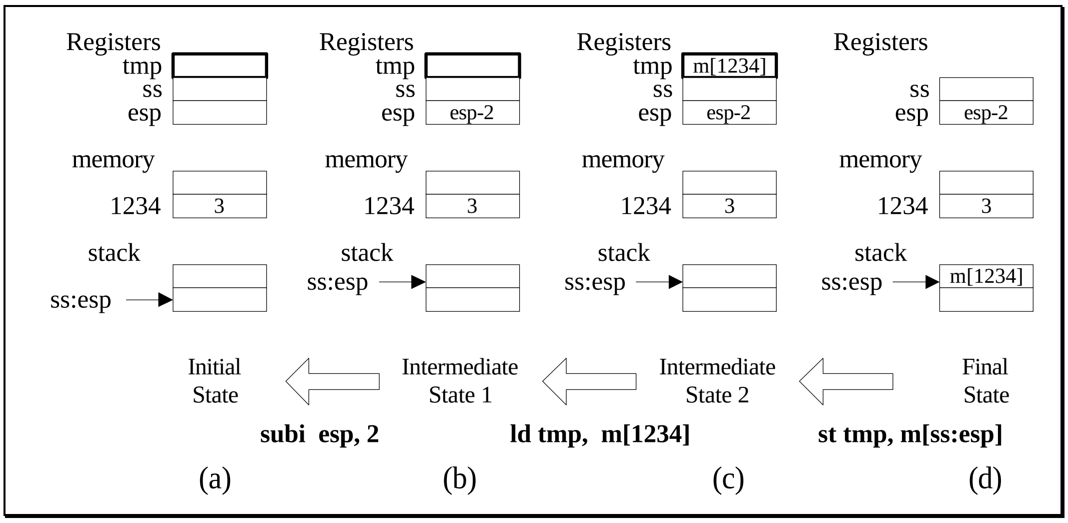


Figure 8. Creation of temporary storage locations (retargeting for Figure 7)

5. Application: x86-to-RISC mapping

With the models and algorithms in previous sections, the retargeting tool StateMapper has been successfully constructed and applied to construct the x86-to-RISC mapping table for the x86 instruction decoders in our x86 compatible microprocessor. Remember that the decoder accepts one x86 instruction at a time, and maps it into internal RISC instruction sequences.

5.1 The mapping of general x86 instructionss

The behavior of each x86 instruction, except some complex instructions to be discussed in Section 5.2, is expressed as a state pair: the initial and the final states. The state pair is given to the algorithm in Section 4.2 as the problem to be solved. The RISC instructions are considered as the candidate operators for the algorithm. A table similar to Table 1 is built for the RISC instruction set.

Here we show an example of translating an x86 instruction `MOV ax,[1234]` into internal instructions. The x86 instruction moves the content of memory location of `1234` to register `ax` Its state representation is as follows:

```
pair([],[content(reg(ax),mem(immed(1234)))]).
```

Figure 9 (a) depicts the state for the x86 instruction; Figure 9 (b) depicts the states of two RISC instructions (`ri` and `rm`) of a RISC core. In the RISC core, we can't find any single instr uction that has the same state pair as the x86 instruction. However, we find an instruction `rm` that moves the value of the register `R1` to the assigned memory location indexed by the register `R2` can match part of the first content element in the goal state list. The state pair of the instruction is

```
pair([],[content(reg(R1),mem(reg(R2)))]).
```

So we bind `R1` to `ax` and `R2` to an temporary register, say `tmp1`. It means if we can find another operation can move the immediate value `1234` to `R2` then we can use this operation to achieve the goal. So we record it into the solution list and insert the following new content into the goal list.

```
content(reg(tmp1),immed(1234)).
```

Taking the updated goal, we find an operation register-immediate move operation `ri`, moving an immediate value `Immed` to a register `R1`, can match perfectly the goal. So `R1` is bound to `tmp1` and `Immed` is bound to `1234`. This instruction does not produce any new content, and therefore the translation is finished.

Figure 9 (c) depicts the backward chaining of the translation process. Finally, by reversing the instruction order, we obtain the following translation for the x86 instruction.

```
ri tmp1, 1234              % reg(tmp1) <- immed(1234)
rm ax, tmp1                % reg(ax) <- m[reg(tmp1)]
```

In a later revision to the RISC instruction set, new load and store instructions are introduced. The instruction `ld` loads a memory location, specified by the immediate value `Immed`, to a register `R1`. The state pair representation of the load instruction is

```
pair([],[content(reg(R1),mem(immed(Immed)))]).
```

By running StateMapper for the same x86 instruction with the new RISC instruction set, we find a perfect match that the final state of `ld` is identical to the goal state. Therefore, we obtain the new translation as the following.

```
ld ax, 1234                 % reg(ax) <- m[immed(1234)]
```

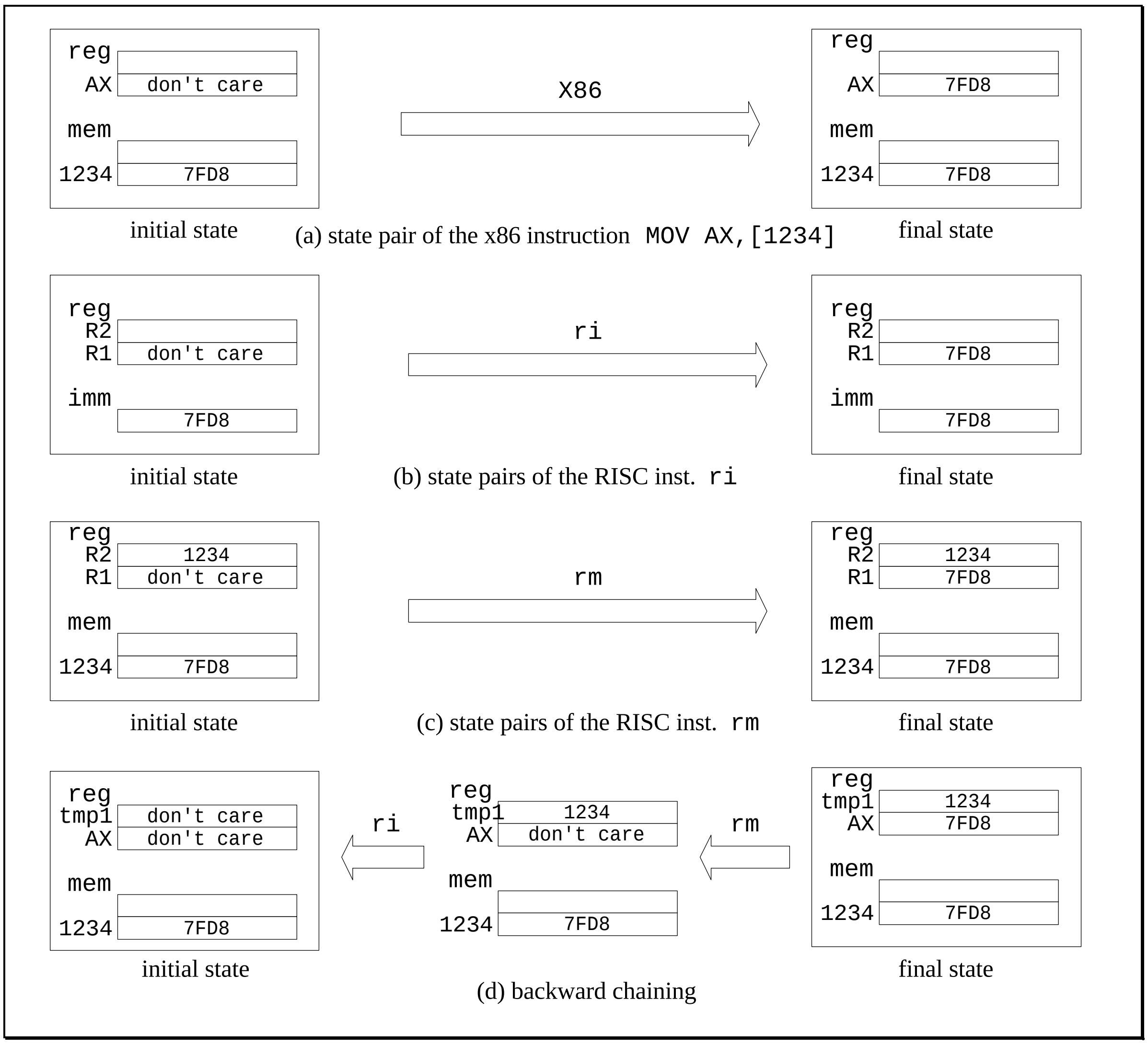


Figure 9. The retargeting example for an x86 instruction

5.2 The mapping of complex x86 instructions

There are several complex instructions in the x86 instruction set that are difficult to be model with simple state pairs, including ENTER, PUSHA, CMPS, REP, *etc*. For example, the instruction ENTER pushes the run-time stack in the external memory and copies multiple items of the calling procedure's stack to the called procedure's stack. It has two forms: if the second operand is greater than zero, the operation includes a while loop.

To model this instruction, we have to divide it into many simpler operations. Table 2 lists the RTL (register transfer level) description of ENTER. The RTL is divided into 7 basic blocks. Following the discussion in Section 3.2, we can derive the state pair notation for each basic block. The state pairs of the first four basic blocks are shown in Figure 10. The left side of this figure is the identifiers of the basic blocks and their corresponding RTL's. The right side of the figure are their sta te pairs. Note that basic blocks b1, b2 and b4 have conditional state pairs.

Basic Block	RTL Description of ENTER
b1	nesting ← max(nesting,31)

b2	PUSH EBP
	temp ← ESP
	if(nesting>0) then

b3	nesting ← nesting - 1
b4	while (nesting>0)

b5	EBP ← EBP - 4
	PUSH SS:[EBP]
	nesting ← nesting - 1

b6	end while PUSH temp

b7	endif
	EBP ← temp
	ESP ← ESP - locals

Table 2. RTL description of ENTER

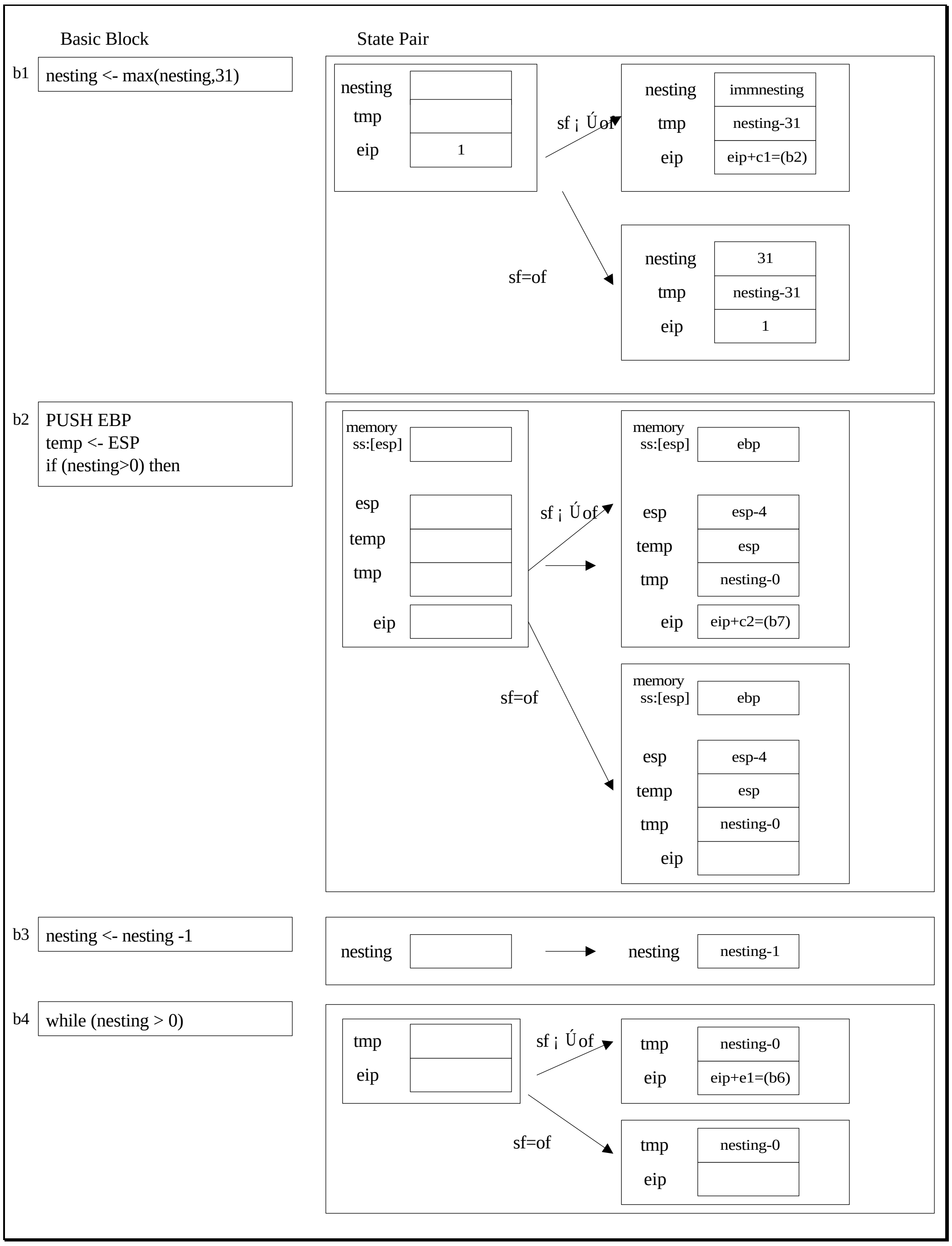


Figure 10. State pairs of the first four basic blocks of the x86 instruction ENTER.

The state pair of each basic block is then given as a problem to be solved by the retargeting algorithm. Figure 11 shows the mapping results for these basic blocks. The left side of the figure are the state pairs of the basic blocks. The right side are the mapped RISC instruction sequences. The figure shows that the four basic blocks are mapped into 4, 5, 1 and 2 RISC instructions, respectively. Table 3 shows the complete RISC translation for the ENTER instruction by merging the mapping for individual basic blocks.

Other complex x86 instructions such as the string manipulation instructions can be processed with the same approach.

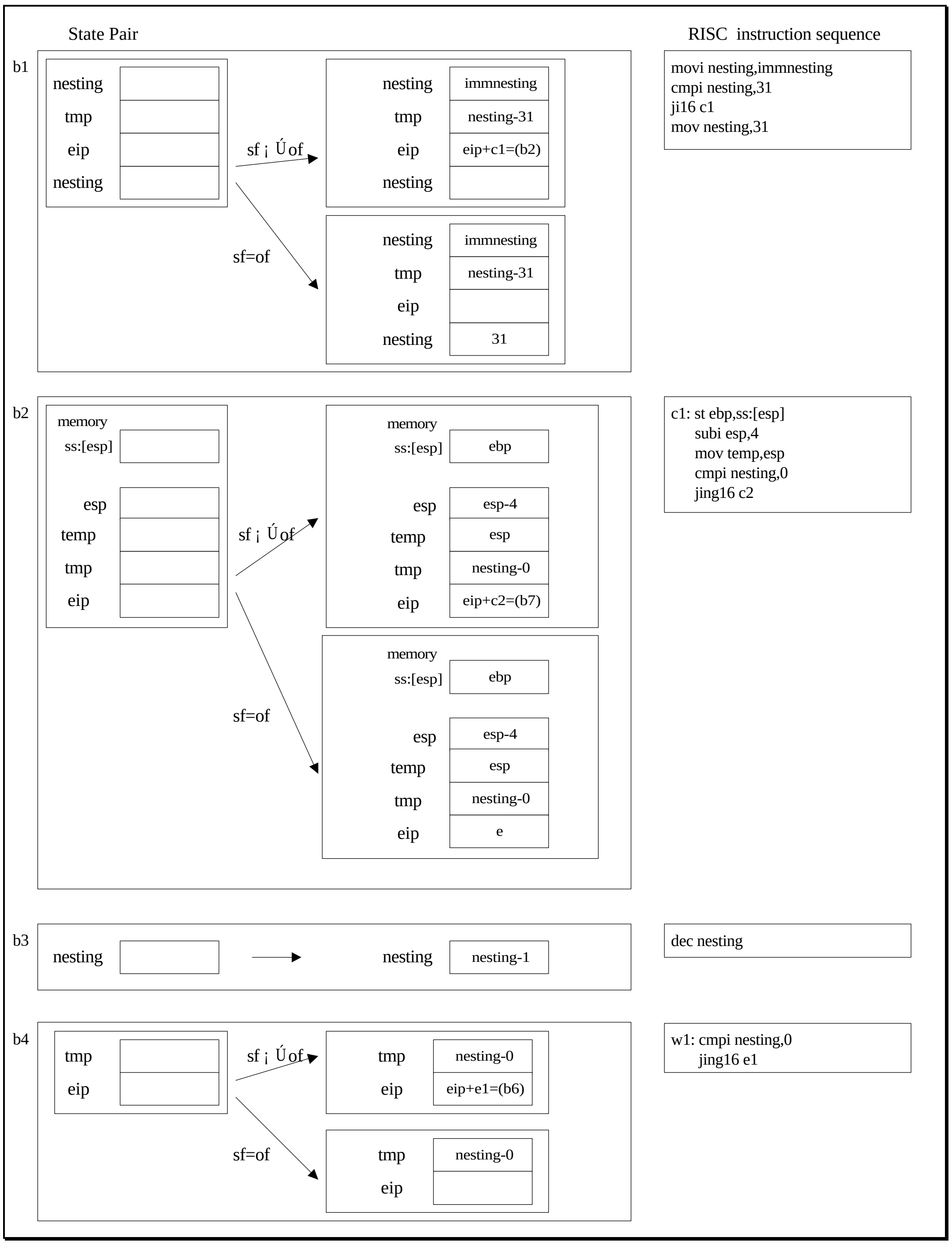


Figure 11. Mapping the state pairs in Figure 10 into RISC instruction sequences

Basic Block	RISC Translation

b1	`movi nesting,immnesting`
	`cmpi nesting,31`
	`ji16 c1`
	`mov nesting,31`

b2	`c1:  st ebp,ss:[esp]`
	`subi esp,4`
	`mov temp,esp`
	`cmpi nesting,0`
	`jing16 c2`

b3	`dec nesting`

b4	`w1:  cmpi nesting,0`
	`jing16 e1`

b5	`subi ebp,4`
	`st ss:[ebp],ss:[esp]`
	`subi esp,4`
	`dec nesting`
	`addi eip,w1`

b6	`e1:  st temp,ss:[esp]`
	`subi esp,4`

b7	`c2:  mov ebp,temp`
	`subi esp,locals`

Table 3. Complete RISC translation for `ENTER`

Note that there are some x86 instructions that do not allow the decomposition of their behaviors, as we did in the previous cases, into basic blocks in order to map them into RISC instruction sequences. Most of them are system related instructions that require atomic operations. In these cases, a single complex RISC instruction is constructed for each of these x86 instructions. Therefore, automatic mapping is not necessary for these instructions. Examples of these instructions include the instruction `HLT` that halts the CPU until reset or interrupt and the instruction `INVD` that invalidates internal cache. A limited number of non-system instructions also have the same property and are dealt with similarily, such as the REP instruction. These instructions, in many RISC-cored x86 compatible microprocessors, are treated as extremely difficult instructions and are not map to RISC-like internal instructions. They are often implemented as hardwired controls or microcode and do not take advantage of the superscalar execution (i.e., the processor enters the sequential mode when executing these instructions).

5.3 Statistics of the mapping results

5.3.1 Instruction classification

We have mapped all x86 instructions, a total of 204 according to their opcode [27]. Since some x86 instructions have up to five addressing modes, these 204 instructions are actually equivalent to more than 340 differentiated instructions (i.e., each differentiated instruction is an opcode associated with a specialized addressing mode). If data bit width is considered, the number would further multiplies. Table 4 classifies the x86 instructions into five categories, lists numbers of instructions for the corresponding categories, and shows the mapping methods for the corresponding categories. The table shows that 85% (174/204) of the x86 instructions can be directly mapped with our StateMapper tool, and another 5% of the instructions, which are complex instructions, can be mapped with our tool after they are broken into smaller pieces (similar to basic blocks). Only about 10% of the x86 instructions are directly implemented as single complex instructions of the RISC core (due to the performance or cost reason) and thus do not require the automatic mapping. These complex instructions are executed by a micro sequencer or some finite state machines. In summary, our method has successfully mapped more than 90% of the x86 instructions.

Category	Number (in terms of different opcodes)	Mapping Method
General Case	174	StateMapper
Complex Instructions	10	Divide into basic blocks and use StateMapper
System Instructions	17	one-to-one mapping (single complex RISC instruction)
Miscellaneous Instructions	3	one-to-one mapping (single complex RISC instruction)
Total	204	

Table 4. Mapping Methods of the complete x86 instruction set

5.3.2 Mapping alternatives and quality

In order to fully explore the mapping space, we configured our algorithm to search for all meaningful solutions for *each* x86 instruction. Table 5 shows the alternative mappings for some x86 instructions. The first column lists the x86 instructions. The second column lists the binding examples of the operands, used for the automatic mapping. The third column shows the alternatives solutions for each x86 instructions. To save space, we only show the shortest and longest mapping for each instruction. The shortest mappings are highlighted with bold faces. The mapping space exploration has revealed many interesting characteristics of the x86 compatible microprocessor, as described in the following.

- The shortest mappings (for all x86 instructions) found by the StateMapper tool are equivalent to the optimal results by manual mapping. This observation ensures that the quality of the automatic mapping is as good as the quality of human experts. Therefore, the automatic mapping approach is a viable solution to our design project.

- On the average, there are 8.68 ways to map an x86 instruction. Some of the ways involves the use of different operations, such as the `INC` case. Some involve the use of temporary registers, such as the `PUSH` case. Some involve the permutation of RISC instruction order.

- On the average, there are 2.09 RISC instructions per x86 instruction. This average is obtained by summing up the total number of RISC instructions of the shortest solutions and then dividing the number with the number of x86 instructions.

- The average ways of mapping and the average RISC instructions per x86 instruction drop significantly from 8.68 to 3.85 and from 2.09 to 1.84, respectively, when considering only the top 25 x86 opcode which cover more than 90% of execution in typical DOS/Windows applications [14]. When the importance of the x86 instructions is weighted according to their frequencies in typical applications, the average RISC instructions per x86 instruction further drop to 1.5. *These observations confirm that powerful x86 instructions are rarely used, and therefore, designers should concentrate on optimizing the implementation of the simpler x86 instructions*.

X86 Instruction	Binding	Automatic Mapping	

PUSH R16	`R16=AX`	Shortest	**`reg(sp)=reg(sp)-immed(2)`**
			`mem(ss:[esp])=reg(ax)`

		Longest	`reg(tmp1)=immed(2)`
			`reg(esp)=reg(esp)-reg(tmp2)`
			`mem(ss:[esp])=reg(ax)`

INC R16	`R16=AX`	Shortest	**`reg(ax)=reg(ax)+immed(1)`**
		Shortest	**`reg(ax)=inc reg(ax)`**

		Longest	`reg(tmp1)=ax`
			`reg(tmp2)=immed(1)`
			`reg(tmp1)=reg(tmp1)+reg(tmp2)`
			`reg(ax)=reg(tmp1)`

CALL I16	`I16=1234`	Shortest	**`reg(esp)=reg(esp)-immed(4)`**
			`mem(ss:[esp])=reg_pc(eip)`
			`reg_pc(eip)=immed(1234)`

		Longest	`reg(tmp1)=reg(esp)`
			`reg(tmp2)=immed(4)`
			`reg(tmp1)=reg(tmp1)-reg(tmp2)`
			`mem(ss:[esp])=reg_pc(eip)`
			`reg_pc(eip)=immed(1234)`

CMP R16 I8	`R16 = AX`
	`I8 = 7`
		Shortest / Longest	**`reg(tmp2)=immed(7)`**
			`reg(tmp1)=reg(ax)`
			`reg(tmp1 )=reg(ax)-reg(tmp1)`

SUB R16 R16	`R16 = AX`
	`R16 = CX`
		Shortest	**`reg(ax)=reg(ax)-reg(cx)`**

		Longest	`reg(tmp1)=reg(ax)`
			`reg(tmp2)=reg(cx)`
			`reg(tmp1)=reg(tmp1)-reg(tmp2)`
			`reg(ax)=reg(tmp1)`

Table 5. Some x86-to-RISC translation

5.3.3 Keeping tracks of the microarchitecture revision

The tool StateMapper not only helps us in exploring the mapping space of the x86 instructions, but also helps us in keeping tracks of the design revision quickly. As shown in Figure 12, by giving the state pair representation of revised versions of the RISC instruction set, we can use StateMapper to automatically re-construct the mapping table accurately and consistently. An example is presented here.

In the original microarchitecture of our microprocessor, there is an address generation unit (AGU) which generates an effective address and store it in the register. The load/store unit then uses the register to access memory. In such implementation an x86 load instruction is mapped into two RISC instructions: `AGU` and `LOAD`. The later version of the microarchitecture merges the AGU into the load/store unit. Therefore, the x86 load instruction is mapped into a single RISC instruction: `LOAD`.

Such simple revision requires the modification of 874 lines in the mapping table, which consists of 3721 lines and is printed on more than 140 pages (in its condensed version). We conducted the modification with both manual and automatic approaches. In the manual approach, it took the designer *more than a week* to scan through the mapping table to manually modify the entries. The time does not include the debugging time nor the mapping exploration. On the other hand, in the automatic approach that uses StateMapper, it only took a few minutes to edit the state pair notations of the `LOAD/STORE` instructions of the RISC core and delete the state pair of the `AGU` instruction of the RISC core, and then it took StateMapper *30 minutes* on a UltraSparcII workstation to regenerate the mapping table. In the 30 minutes, the tool not only re-built the entries but also explored all the mapping space and found the optimal solutions.

With the above example, the contribution of the automatic instruction set retargeting can be immediately appreciated for architecture exploration. Both human resources and information management complexity can be greatly reduced.

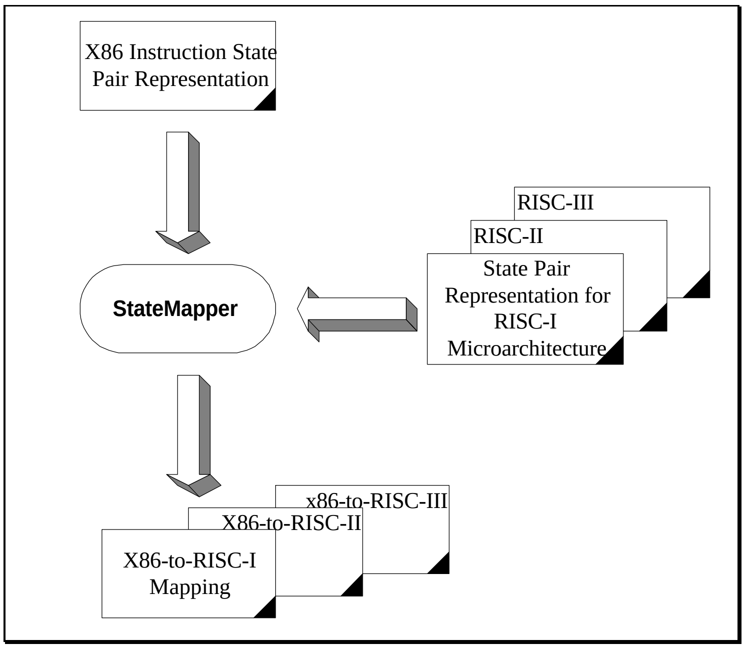

X86 Instruction State Pair Representation
StateMapper
RISC-III
RISC-II
State Pair Representation for RISC-I Microarchitecture
x86-to-RISC-III
X86-to-RISC-II
X86-to-RISC-I Mapping

Figure 12. Keeping tracks of design revisions

6. Conclusions

We have presented a new approach to retarget existing software at the assembly level from one instruction set to other instruction sets. The approach is based on abstracting the instruction set behaviors as the symbolic transitions of the machine states. The retargeting process is then modeled as a planning process that finds a plan with the lowest cost, consisting of a chain of state transitions, which brings the processor from the same initial state to the same final state as the original software. The unique feature of the approach is that the assembly-to-assembly retargeting capability helps renovating the existing software investment while upgrading to processors with newer instruction sets, at the absence of source code or intermediate code, which is commonly encountered in industrial embedded system applications.

We have demonstrated the application of the proposed approach by translating Intel's x86 instruction set to a RISC-based instruction set in a microprocessor which maps x86 instruction set into an internal RISC-based instruction set for efficient execution. The automatic mapped results are as optimal as the manually mapped results. In addition, the proposed approach has made it possible to keep up with the architecture/microarchitecture revision during the design exploration by automatically re-generating and re-evaluating the x86-to-RISC mapping table promptly, which is difficult to achieve manually.

Future work can be extended in several directions. First, in addition to the instruction-by-instruction style reported in this paper, our proposed approach can be applied to translate entire assembly programs for software migration between embedded microcontrollers. In such case a state pair may denote the aggregate effect of a basic block or even a longer trace of instructions, which would make the matching process more challenging. Therefore, graph-based matching algorithms, instead of the tree-based one in this paper, could become necessary to provide better matching quality. Second, a mechanism to measure, match and adjust the timing behaviors between the original and translated assembly programs is also an important task for embedded applications since what matters in such applications is not only the computation behavior but also the temporal behavior. Third, a formal approach to ensure the correctness of the state-based translation is also highly desired.

Reference

[1] Manuel E. Benitez, Jack W. Davidson : *"A Retargetable Code Improver"*, University of Virginia

[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, Oct. 1989

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

[4] 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.

[5] Clifford Liem, Retargetable Compilers for Embedded Core Processors, Methods and Experiences in Industrial Applications, Kluwer Academic Publishers, 1997.

[6] Ashok Sudarsanam, Sharad Malik : "*Simultaneous Reference Allocation in Code Generation for Dual Data-Memory Bank ASIPs*," IEEE International Conference on CAD, 1995.

[7] Bruce K. Holmer, Alvin M. Despain : "*Viewing Instruction Set Design as an Optimization Problem*", Proc. of MICRO-24,1991

[8] Peter L. Van Roy, "*Can Logic Programming Execute as Fast as Imperative Programming*", Ph.D. thesis, Technical Report UCB/CSD 90/600, Univ. of California, Berkeley, 1990

[9] R. E. Fikes, Nils J. Nilsson, "*STRIPS: A new approach to the application of theorem proving to*

problem solving" Artificial Intelligence, 2:189-208, 1971.

[10] David E. Wilkins, "*Practical Planning: Extending the Classical AI Planning Paradigm*", Morgan Kaufmann, San Mateo, CA, 1988

[11] A.V. Aho, R. Sethi and J.D. Ullman, "*Compilers Principles, Techniques, and Tools*", Addison Wesly, 1985

[12] AMD-K5 Technical Reference Manual, *http://www.amd.com/products/cpg/techdocs/appnotes/18524c.pdf*

[13] Pentium Processor Overview, *http://developer.intel.com/design/pentium/*

[14] I. J. Huang and T. C. Peng, "Analysis of x86 Instruction Set Usage for DOS/Windows Applications and Its Implication on Superscalar Design," *Proceedings of International Conference on Computer Design*, 1998.

[15] I. J. Huang and W. F. Gao, "Instruction Retargeting Based on the State Pair Notation," *Proc. of Asia and Pacific Conference on Hardware Description Languages*, pp. 114-120, Aug. 1997.

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

[17] P. Marwedel, "Tree-based Mapping of Algorithm to Predefined Structures", ICCAD 1993

[18] RASM51E MCS-51 Cross Assembler, *http://ftp.unina.it/pub/eletrocnics/ftp.armory.com.8051/RASM51E.TXT*

[19] XASM Cross Assembler, *http://www.nuri.net/simtel.net/msdos/crossasm-pre.html/xasm220.zip*

[20] Richard L. Sites *et al.*, "*Binary Translation*", Communication of the ACM, Feb, 1993

[21] George, W. Ernst and Allen Newell, "*Some issues of representation in a general problem solver*", Sprint Joint Computer Conference 30, 1967

[22] Colin Hunter and John Banning, "*DOS at RISC*" , BYTE, Nov. 1989.

[23] Mike Johnson, "*Superscalar Microprocessor Design*", Prentice Hall,1991.

[24] Ross P. Nelson, "*Microsoft's 80386/8 0486 Programming Guide*", Microsoft Press, 1991.

[25] Anton Ghernoff et at., "FX!32 A Profile-directed Binary Translator," IEEE Micro, pp. 56-64, March/April 1998.

[26] 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.

Accepted to Journal of Design Automation for Embedded Systems, 1999.

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

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

[29] J. Loeckx and K Sieber, Chapter 3, *The Foundations of Program Verification*, 2nd ed., John Wiley & Sons, 1987.

[30] J. Fernandez, *et al*., “On-the-fly Verification of Finite Transition Systems,” *Journal of Formal Methods in System Design*, 1: 251-273, Kluwer Academic Publishers, 1992.

[31] R. Kurshan, Section 3.2, *Computer-Aided Verification*, Kluwer Academic Publishers, 1993.

[32] C. Monahan and F. Brewer, “Symbolic Modeling and Evaluation of Data Paths,” *Proc. ACM/IEEE 32nd Design Automation Conference*, June 1995.

[33] S. Bashford and R. Leupers, “Phase-Coupled Mapping of Data Flow Graphs to Irregular Data Paths,” Journal of Design Automation for Embedded Systems, c