Synthesis of Application Specific Instruction Sets

Ing-Jer Huang, *Member, IEEE* and Alvin M. Despain, *Member, IEEE*

Abstract— **An instruction set serves as the interface between hardware and software in a computer system. In an application specific environment, the system performance can be improved by designing an instruction set that matches the characteristics of hardware and the application. We present a systematic approach to generate application-specific instruction sets so that software applications can be efficiently mapped to a given pipelined micro-architecture. The approach synthesizes instruction sets from application benchmarks, given a machine model, an objective function, and a set of design constraints. In addition, assembly code is generated to show how the benchmarks can be compiled with the synthesized instruction set. The problem of designing instruction sets is formulated as a modified scheduling problem. A binary tuple is proposed to model the semantics of instructions and integrate the instruction formation process into the scheduling process. A simulated annealing scheme is used to solve for the schedules. Experiments have shown that the approach is capable of synthesizing powerful instructions for modern pipelined microprocessors, and running with reasonable time and a modest amount of memory for large applications.**

I. Introduction

MICROPROCESSORS (instruction set processors) offer a flexible and low-cost solution for embedded systems with complex algorithms or control intensive applications. The performance of a microprocessor-based system depends on how efficiently the application is mapped to the hardware. One key issue determining the success of the mapping is the design of the instruction set, which serves as the interface between the hardware and application (software). How to design an instruction set that closely matches the characteristics of the hardware and the application is an important design problem.

The design of instruction sets was once viewed as a design process independent to the design of the hardware (micro-architecture). Instruction sets designed under this principle, such as those of many mainframe computers, suffered from the fact that their supporting hardware was difficult to speed up or hardware was wasted due to the low utilization rate of the related instructions in real applications. The necessity of closely matching the design of instruction sets with the design of micro-architectures was recognized and adopted in the design of many modern RISC-style pipelined processors, in order to achieve better performance and cost trade-off. However, in most design projects, the designs were carried out manually, which limited the exploration of the design space and the understanding of the interaction between hardware and software. CAD tools are necessary to explore and manage such complex design space. While there has been much progress in automating the instruction set processor design, most of the work synthesizes micro-architectures at the RTL level from given instruction sets (e.g., [11], [13], and [14]). How to systematically design instruction sets which closely match the characteristics of hardware and software is still an open problem. The goal of our research is thus to investigate the instruction set design problem in a systematic way. The research intends to provide further understanding of the design and interaction of the hardware and software interface.

In this paper we present the problem formulation and the algorithm of a systematic approach [7] which synthesizes application-specific instruction sets for parameterized, pipelined micro-architectures, from a given application benchmark. The problem is formulated as a modified scheduling problem, with the micro-operations (MOP's) representing the application benchmark as the nodes to be scheduled, subject to several design constraints. Instructions are formed by an instruction formation process which is integrated into the scheduling process. The compiled code of the application is generated, using the synthesized instruction set. A simulated annealing scheme is used to solve for the schedule and the instruction set. The design issues addressed in this approach include: instruction utilization, instruction operand encoding, delay load/store and delay branches.

The rest of the paper is organized as follows. Section II reviews related work. Section III presents the models for the micro-architectures, instruction sets and application benchmarks. Sections IV and V describe the problem formulation and algorithm, respectively. Section VI demonstrates our techniques with some experiments. Section VII discusses the current status, limitations, and future directions.

II. Related Work

Most of the early work in automatic instruction set design views the design problem as a design process independent to the hardware implementation. Instructions were not restricted to single cycle instructions since multiple cycle instructions can be supported through micro-programming (firmware). Without knowing the decode/control complexity, the focus was mainly in directly supporting high-level languages or increasing the code density. The results were CISC-like instructions. These studies include Haney's [1], Bose's [2], and Bennett's [3] work. These techniques are not suitable for designing instruction sets for modern pipelined processors.

Sato *et al.* [6] propose an integrated design framework for application specific instruction set processors. This framework

Manuscript received March 11, 1994; revised January 19, 1995. This work was supported by the ARPA under Contract Rutgers 4-26385. This paper was recommended by Associate Editor M. McFarland.

I.-J. Huang is with the Institute of Computer and Information Engineering, National Sun Yat-Sen University, Kaohsiung, Taiwan 804 R.O.C.

A. M. Despain is with the Department of Electrical Engineering–Systems, University of Southern California, Los Angeles, CA 90007 USA.

IEEE Log Number 9410376.

TABLE I
BIT WIDTH SPECIFICATION FOR SOME INSTRUCTION FIELD TYPES

Instruction Field Type	Number of bits
instruction word	32
opcode	6
register (R)	5
tag (T)	5
displacement (D)	16
immediate (I)	16
relation operator (OP)	2

generates profiling information from a given set of application benchmarks and their expected data. Based on the profiles, the design system customizes an instruction set from a super set, decides the hardware architecture (derived from the GCC's abstract machine model), and the related software development tools. This framework is similar to our work in terms of the inputs and outputs of the design system; however, it is different from ours in terms of the machine model and the design method. They assume a sequential (nonpipelined) machine model, whereas we assume a pipelined machine with data-stationary control model. On the other hand, they generate instruction sets by selecting subsets from a super set, whereas we synthesize the instruction sets directly in order to find new and useful instructions for the given application domain.

Different from previous approaches, Holmer [4], [5] focuses on generating instruction sets which closely couple to the underlying micro-architecture. As pipelined micro-architecture proved its superiority in 1980's, Holmer adopts the modern pipeline control model (data stationary control) and simple, parameterized data path as the underlying micro-architecture model. The parameters for a data path include the number of read/write register ports, memory ports, number of functional units and the cycle counts for memory operation. The user specifies the parameters, and then invokes the system to find the set of instructions which best utilizes the hardware resources such that minimal cycle counts for benchmarks are achieved. Our work builds on the results of Holmer and improves the problem formulation and synthesis algorithms, in order to generate application-specific instruction sets and compiled codes for microprocessor-based embedded systems.

Another design problem that is close to the instruction set design problem is microcode compaction [15]–[17]. However, it differs in terms of the design space and design goals. The micro-instructions do not have "opcodes" (and hence the semantics) and the goal of microcode compaction is to reduce the number of cycles to execute a microprogram. On the other hand, in the instruction set design, the size of the instruction set is determined by both syntax and semantics. The goal of the instruction set design is to optimize and trade off the instruction set size, the program size, and the number of cycles to execute a program.

III. DESIGN MODELS

In this section we present the models of instruction sets, micro-architectures and application benchmark programs, and describe how they are represented in our design system.

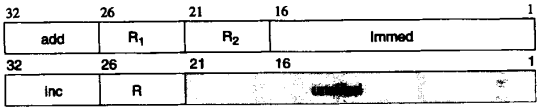

Fig. 1. Examples of instruction formats.

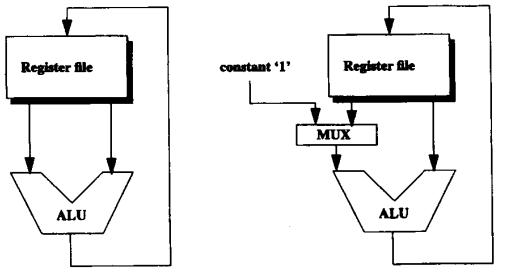

Fig. 2. Variation in data path for different instruction sets.

A. Instruction Sets

The instruction set is assumed to be of fixed word length, typically 32 b, which is specified by the designer. An instruction consists of fields. The fields are a combination of some field types. For example, the instruction `add(`R_1`, `R_2`, Immed)` consists of an opcode field `add`, two register index fields R_1 and R_2, and one immediate data field `Immed`. The bit width of each field type is provided by the designer. Table I lists the specification of some instruction field types and their bit widths, taken from the BAM instruction set [19]. Each instruction has one opcode field, but the use of other fields is constrained only by the total number of bits needed by the operations in the instruction.

Fig. 1 lists the instruction formats for the instructions `add(`R_1`,`R_2`, Immed)` '$R_1 \leftarrow R_2 +$ Immed' and `inc(R)` '$R \leftarrow R + 1$', based on the bit width specification in Table I. Note that there are 21 b unused in the format of `inc`.

The operands of instructions can be encoded in the opcodes. There are two ways to encode operands. First, a specific value can be permanently assigned to an operand and becomes *implicit* to the opcode. Second, the register specifiers can be *unified*. For example, the instruction `inc` is obtained from the general instruction `add`. The facts of $R_1 = R_2$ (unifying register specifiers; i.e., both register accesses refer to the same physical register) and `Immed` $= 1$ (fixing an operand to a specific value which becomes implicit) are encoded into the opcode `inc`. Encoding operands saves instruction fields, at the cost of possibly larger instruction set size, additional connections and hardwired constants in the data path. For example, adding the instruction `inc` to the instruction set increases the instruction set size by one, and adds a hardwired constant '1' and an additional multiplexer in the data path, as shown in Fig. 2.

Furthermore, encoding allows more MOP's to be packed into a single instruction. For example, if we find it happens very often that the values of two independent registers are increased by one at the same time, we may then devise a

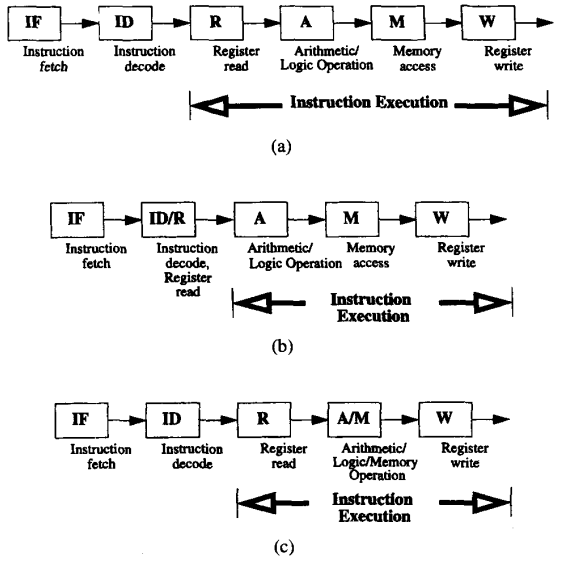

Fig. 3. Basic pipeline and its variations.

TABLE II
MOP SPECIFICATION

Type ID	MOP*	Instruction Format Cost†	Hardware Cost‡
rr	$R_1 \leftarrow R_2$	R_1, R_2	1 R, 1 W
rra	$R_1 \leftarrow R_1 + R_2$	R_1, R_2	2 R, 1 W, 1 F
rrai	$R_1 \leftarrow$ Immed + R_2	R_1, R_2, I	1 R, 1 W, 1 F
rrait	$R_1 \leftarrow$ Tag^(Immed + R_2)	R_1, R_2, T, I	1 R, 1 W, 1 F
ri	$R_1 \leftarrow$ Immed	R_1, I	1 W
rit	$R_1 \leftarrow$ Tag^ Immed	R_1, T, I	1 W
rm	$R_1 \leftarrow$ mem(R_2)	R_1, R_2	1 R, 1 W, 1 M
rmd	$R_1 \leftarrow$ mem(R_2 + Immed)	R_1, R_2, I	1 R, 1 W, 1 M, 1 F
mr	mem(R_1) $\leftarrow R_2$	R_1, R_2	2 R, 1 M
mi	mem(R_1) $\leftarrow$ Immed	R_1, I	1 R, 1 M
mrd	mem(R_1 + Disp) $\leftarrow R_2$	R_1, R_2, D	2 R, 1 M, 1 F
mrad	mem(R_1 + Disp) $\leftarrow R_2$ + Immed	R_1, R_2, D, I	2 R, 1 M, 2 F
jd	pc $\leftarrow$ pc + Immed	I	1F

*. The operator '^' appends a tag to a value before the value is sent to a destination.
†. Refer to the notation in Table I
‡. Notation: 'R'=read port of register-file, 'W'=write port of register-file, 'M'=memory port, 'F'=functional unit, and the value is the number of a particular hardware resource. For example, '2R' means two read ports for register-file.

new instruction `incd(R1, R2)` which performs the MOP's '$R_1 \leftarrow R_1 + 1$; $R_2 \leftarrow R_2 + 1$' (';' represents concurrency). This instruction uses only 16 b, as opposed to 58 b used by its generalized form '$R_1 \leftarrow R_2 + \text{Immed}_1$; $R_3 \leftarrow R_4 + \text{Immed}_2$' which does not meet the instruction word width constraint for 32-b instructions.

B. Micro-Architectures

The styles of micro-architectures considered in this work are pipelined micro-architectures. For example, Fig. 3(a) shows a basic pipeline, which can be functionally partitioned into 6 pipeline stages: instruction fetch (IF), instruction decode (ID), register read (R), arithmetic/logic operation (A), memory access (M), and register write (W). Each functional stage may take more than one cycle, and can be further pipelined. The first two stages are identical to all instructions. The last four stages, the *instruction execution stages*, are dependent on the semantics of the instructions. The combination of pipeline stages can be varied. For example, the pipeline 'IF-ID/R-A-M-W' of Fig. 3(b) can be derived by merging the register-read stage with the instruction-decode stage, at the cost of restricting the instructions to use a single format for register specification such that registers can always be prefetched at the instruction-decode stage. On the other hand, the pipeline 'IF-ID-R-A/M-W' of Fig. 3(c) is derived by merging the arithmetic stage with the memory stage, at the cost of eliminating the displacement addressing mode. The displacements have to be computed by other instructions proceeding the memory-related instructions.

The pipeline is controlled in a data stationary fashion [9]. In the data stationary control, the opcode flows through the pipeline in synchronization with the data being processed in the data path. Fig. 4 shows the relationship between the control path with data stationary model and the data path. The register files at the top and bottom are the same register file. They are duplicated for the ease of readability. Opcodes are forwarded to next stages synchronously. At each stage, the opcode, together with possible status bits from the data path, is decoded to generate the control signals necessary to drive the data path.

This pipeline configuration supports single-cycle instructions[1] which are typical of modern RISC-style processors. Multiple-cycle instructions can be accommodated with some modification to the linear pipeline such as the insertion of internal opcodes [10]. To manage the complexity of this research, general multiple-cycle instructions are not considered at this moment. However, multiple-cycle arithmetic/logic operations, memory access, and change of control flow (branch/jump/call) are supported by specifying the delay cycles as design parameters.

The Specification for the Target Micro-Architecture: The target micro-architecture can be fully described by specifying the supported MOP's and a set of parameters. The supported MOP's describe the functionality supported by the micro-architecture, and the connectivity among modules in the data path. For example, the first two columns of Table II list some of the MOP's supported in the VLSI-BAM microprocessor [20] and their corresponding MOP type ID's. The basic pipeline structure of the microprocessor is the same as Fig. 3(b).

The tabulated specification supports the variations of the micro-architectures easily. For example, the pipeline configuration 'IF-ID-R-A/M-W' in Fig. 3(c) can be derived by eliminating the MOP's `rmd`, `mrd` and `mrad` from Table II.

The set of parameters describes resource allocation and timing. The parameters include the number of register-file read/write ports, number of memory ports, number of functional units, the sizes of the register file and memory, latencies of operations, and the delay cycles between operations of memory access, functional units and control flow change.

[1] A single-cycle instruction has instruction latency of one cycle.

TABLE III
EXAMPLE OF PARAMETERS FOR THE TARGET MICROARCHITECTURES: RESOURCE SIZE AND LATENCY

Resource Type	#	Operation	Latency
Read port; register file (R)	3	Register read	1
Write port: register file (W)	1	Register write	1
Memory read/write port (M)	2	Memory access	1
Functional unit (F)	1	Arithmetic/Logic	1
Register file size	32		
Memory size	2^{32}		

TABLE IV
EXAMPLE OF PARAMETERS FOR THE TARGET MICROARCHITECTURES: DELAY CYCLES

Operation pair	Delay cycles	Operation pair	Delay cycles
arithmetic-arithmetic (A-A)	0	memory-control (M-C)	1
arithmetic-memory (A-M)	0	control-arithmetic (C-A)	1
arithmetic-control (A-C)	0	control-memory (C-M)	1
memory-arithmetic (M-A)	1	control-control (C-C)	1
memory-memory (M-M)	1		

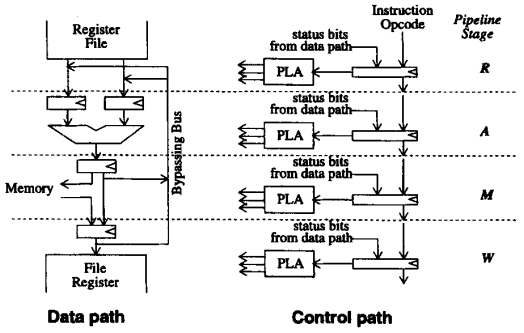

Fig. 4. Data stationary control model.

Table III is an example of the resource parameters for the VLSI-BAM microprocessor. The resource parameters specified in this table include the numbers and sizes of resources, and their operation latencies. Table IV lists the delay parameters for various pairs of operations. For example, the M-A pair in the table specifies that there should be one cycle delay between a memory operation and a succeeding (dependent) arithmetic operation.

Note that the existence of bypassing buses in the data path can be modeled by the delay parameters. For example, if we remove the bypassing bus in the 'A' stage in Fig. 4, then the delay cycles for the A-A, A-M, and A-C pairs all become one, instead of zero.

Each MOP supported by the data path is assigned costs for the instruction format and hardware resources. The costs of the instruction format are the instruction fields required to operate the MOP's, including register index, function selectors, and immediate data. The hardware costs are the hardware resources required to support the MOP. The hardware resources include read/write ports of the register file, memory ports, and functional units. The third and fourth columns in Table II lists the costs for the corresponding MOP's.

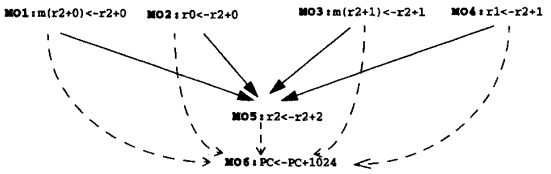

Fig. 5. The control/data flow graph (CDFG) of MOP's of a simple basic block.

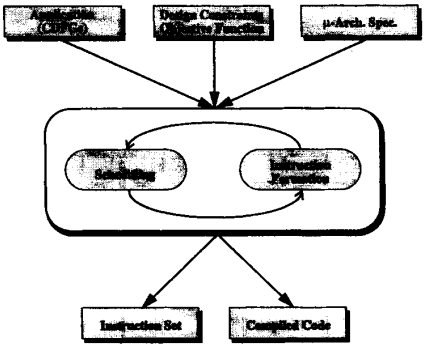

Fig. 6. The integrated scheduling/instruction-formation process.

C. Application Benchmarks

Each application benchmark is represented as a group of weighted basic blocks. The weight is defined by the designers, and is usually used to indicate how many times the basic block is executed in the benchmark. The basic blocks are mapped to control/data flow graphs (CDFG's) of MOP's, based on the given MOP specification. Different micro-architectures result in different MOP specifications, which may map the basic blocks to different CDFG's. Fig. 5 shows an example of a basic block, which consists of six MOP's, based on the MOP specification in Table II. The bold labels before the MOP's are their ID's. The dashed arrows are control dependencies; the MOP's `MO6` changes the control flow at the end of the basic block, and hence logically follows MOP's `MO1` $\sim$ `6`. The solid arrows are data-related dependencies. The data related dependencies can be characterized into three categories: *read-after-write* (RAW), *write-after-read* (WAR), and *write-after-write* (WAW). They all specify a *before* relation: the preceding MOP has to be scheduled before the succeeding MOP, except in micro-architectures where master-slaved latches are used to implement registers. In this case, the WAR dependency indicates a *no-later-than* relation: the preceding MOP has to be scheduled no later than the succeeding MOP. The data dependencies in the figure are all WAR's.

IV. INSTRUCTION SET DESIGN AS A MODIFIED SCHEDULING PROBLEM

The instruction set design problem can be formulated as a modified scheduling problem (Fig. 6). The inputs of the problem are: an application represented in CDFG's, constraints of the instruction word and field widths and hardware resources, the objective function, and the micro-architecture specification. The MOP's in CDFG's are scheduled into time steps, subject

TABLE V
SCHEDULE I FOR THE MOP'S IN FIG. 5 AND THE RESULTED INSTRUCTIONS. *. REFER TO THE FOOTNOTE OF TABLE II FOR THE MEANING OF THE NOTATION

Schedule		Instruction Semantics				Instruction Fields		Costs	
Time step	MOP IDs	RTLs	MOP type IDs	Encoded fields	Inst. name	Format	Field values	Hardware cost*	Inst. word width
1	mo1	$m(R_1+D_1)<-R_2+D_2$	mrad		inst1	R_1, R_2, D_1, D_2	r2, r2, 0, 0	2R, 1M, 2F	48
2	mo2	$R_1<-R_2+1$	rrai	I=0	inst2	R_1, R_2	r0, r2	1R, 1W, 1F	16
3	mo3	$m(R_1+D_1)<-R_2+D_2$	mrad	$D_1=D_2$, $R_1=R_2$	inst3	R_1, D_1	r2, 1	1R, 1M, 1F	27
4	mo4	$R_1<-R_2+1$	rrai		inst4	R_1, R_2, I	r1, r2, 1	1R, 1W, 1F	32
5	mo5	$R_1<-R_2+1$	rrai		inst4	R_1, R_2, I	r2, r2, 2	1R, 1W, 1F	32
6	mo6	pc<-pc+D	jd		inst5	D	1024		22
7	nop		nop		inst6				6
Cycle count 7			Instruction set size 6			Hardware cost 2R, 1W, 1M, 2F		Max. instruction width = 68	

TABLE VI
SCHEDULE II FOR THE MOP'S IN FIG. 5 AND THE RESULTED INSTRUCTIONS

Schedule		Instruction Semantics				Instruction Fields		Costs	
Time step	MOP IDs	RTLs*	MOP type IDs	Encoded fields	Inst. name	Format	Field values	Hardware cost	Inst. word width
1	mo1, mo2	$m(R_1+D_1)<-R_2+D_2$; $R_3<-R_4+I$	mrad, rrai	$R_1=R_2=R_4$, $D_1=D_2=I$	inst7	R_1, R_3, D_1	r2, r0, 0	1R, 1W, 1M, 1F	32
2	mo3, mo4	$m(R_1+D_1)<-R_2+D_2$; $R_3<-R_4+1$	mrad, rrai	$R_1=R_2=R_4$, $D_1=D_2=I$	inst7	R_1, R_3, D_1	r2, r1, 1	1R, 1W, 1M, 1F	32
3	mo6	pc<-pc+D	jd		inst5	D	1024		22
4	mo5	$R_1<-R_2+I$	rrai		inst4	R_1, R_2, I	r2, r2, 2	1R, 1W, 1F	32
Cycle count 4			Instruction set size 3			Hardware cost 1R, 1W, 1M, 1F		Max. instruction width = 32	

*. ';' denotes concurrency. RTLs separated by ';' execute concurrently.

to various constraints to be discussed later. While scheduling MOP's into time steps, instructions are formed at the same time. Finally, the outputs of this problem formulation is a synthesized instruction set and compiled code.

Two schedules of the MOP's in Fig. 5 are shown in Tables V and VI, respectively. In the first column of the table are time steps, and in the second column are the ID's of the MOP's scheduled into the corresponding time step. In this example we assume a one-cycle delay for the jump MOP (MO6) and zero-cycle delay for memory MOP's (MO0 and MO3). The schedule in Table V is a serialized one, with seven cycles. There is one MOP in each time step. Note that there is a nop at the seventh cycle since MO6 is scheduled as the last MOP. The schedule in Table VI is a more compact one, with four cycles. Note that the delay slot of MO6 is filled with MO5 such that there is no need for a nop.

A. Instruction Formation: The Binary Tuple and Its Relation with Scheduling Process

The semantics of an instruction can be represented by a binary tuple ⟨*MOPTypeIDs,-IMPFields*⟩, where *MOPTypeIDs* is a list of type ID's (as shown in the first column of Table II) of MOP's contained in the instruction, and *IMPFields* is a list of fields that are encoded into the opcode.

For example, the binary tuple for the instruction add(R_1, R_2, Immed) is ⟨[rrai], []⟩. The instruction contains one MOP '$R_1 \leftarrow R_2$+ Immed' with the type ID rrai, which is represented by the list in the first argument of the tuple. Since no fields are encoded, the second argument of the tuple is an empty list. On the other hand, the binary tuple for the instruction inc(R), an encoded version of the instruction add(R_1, R_2, Immed) as discussed in Section III-A, is ⟨ [rrai], [R_1 = R_2, Immed = 1] ⟩. The list in the second argument of the tuple specifies how the fields are encoded: The element R_1 = R_2 unifies the register specifiers R_1 and R_2 to the same register, and the element Immed = 1 fixes the immediate value permanently to the constant of one.

Instructions are generated from time steps in the schedule. Each time step corresponds to one instruction. The type ID's of the MOP's scheduled to the same time step are assigned to the first argument of the binary tuple for the instruction at the time step. The operand encoding specification, which is generated by an encoding process integrated into the scheduling process (described in Section V), is assigned to the second argument of the binary tuple.

In Tables V and VI, the columns under the header 'Instruction Semantics' and 'Instruction Fields' describe the semantics and field information of the instructions formed for the two schedules, respectively. The columns 'MOP type IDs' and 'Encoded fields' specify the binary tuples for the instructions. The RTL's for the corresponding MOP types are listed under the 'RTLs' column. Note that ';' denotes concurrency. The 'Inst Name' column assigns names to the generated instructions. The column 'Format' describes the instruction format, i.e., the required instruction fields. The column 'Field values' lists the instantiated field values for the corresponding time step. Note that, in order to demonstrate the variation in the instruction formation, the instruction set in Table V is chosen from a nonoptimal one.

For example, in Table V, the MOP's scheduled into time steps 4 and 5 have the same binary tuple, and thus are mapped to the same instruction inst4(R_1, R_2, I), with their field values instantiated to (r1, r2,1) and (r2, r2,2), respectively. Note that we use capitalized letters, e.g. R_1, to denote the instruction fields, and noncapitalized letters, e.g. r2, to denote the instantiated values of the fields. On the other hand, the MOP in time step 2, is mapped to a different instruction inst2(R_1, R_2), although it contains the same type of MOP rrai as in time steps 4 and 5. The reason is that its field for the immediate data I is permanently assigned to the constant 'zero' and made implicit in the opcode, which is indicated by the specification I = 0 in the 'Encoded field' column. This implicit field makes the generated instruction behave as a 'move' instruction, instead of 'add.'

The compiled code can be obtained easily from the instruction names and instantiated field values. For example, the compiled code for the scheduled basic block in Table VI is represented as the sequence

inst7(r2, r0, 0), inst7(r2, r1, 1), inst5(1024), inst4(r2, r2, 2).

The instruction set is formed by unioning instructions generated from all time steps. For example, the instruction set derived from the schedule in Table V contains six instructions (inst1~inst6), and the instruction set for the schedule in Table VI contains three instructions (inst4, inst5, inst7).

B. Performance (Cycle Count) and Costs (Instruction Bits and Hardware Resources)

The weighted sum of the lengths (number of time steps) of the scheduled basic blocks is the execution cycles of the benchmarks. The length of the basic block includes `nop` slots which are inserted by the design process to preserve the constraints due to multicycle operations. The design process will try to eliminate the nop slots by reordering other independent operations into the `nop` slots.

Each instruction has two costs associated with it. One is the total number of bits required to represent the instruction. The number is a summation of field widths of opcode and all explicit fields required to operate the MOP's contained in the instruction. The implicit fields do not consume instruction bits. For example, in Table V, the instruction `inst4` requires 32 b, using the bit width specification in Table I; whereas `inst2` requires 16 b only because its immediate data field is made implicit, saving 16 b. The maximal bit widths of the instruction sets in Tables V and VI are 48 and 32 b, respectively.

Another cost is hardware. It is the collection of the resources required by all MOP's contained in the instruction, minus the shared resources. The sharing of the resources can be related to field encoding. When two or more register reads of different MOP's are unified, i.e., reading from the same register, one read port of the register file is sufficient, instead of two or more. On the other hand, if more than one destination register receive results of the same arithmetic/logic expression, one functional unit is enough since the computation result can be shared. For example, `inst7` needs only one read port instead of three since R_1, R_2, and R_4 are unified. It also needs only one functional unit, instead of three, since the three destinations (memory data register, memory address register, register file) all receive the same value: $R_1 + D_1$.

The global hardware resources are obtained by choosing the maximal number for each resource type from all instructions. For example, the global hardware resources used for the schedule I and II in Tables V and VI are $\langle 2R, 1W, 1M, 2F\rangle$ and $\langle 1R, 1W, 1M, 1F\rangle$,[2] respectively.

The example in Table VI shows that compact and powerful instructions can be synthesized by packing more MOP's into a single instruction, and making fields implicit and register ports unified to satisfy the cost constraints. This is particularly useful in an application specific environment where instruction sets can be customized to produce compact and efficient codes for the intended applications.

C. Constraints

The MOP's are scheduled into time steps, subject to several constraints. First, the data/control dependencies and the timing constraints (for multicycle MOP's) have to be satisfied. Data-dependent MOP's have to be scheduled into different time steps, subject to the precedent relationship and timing constraints, except single-cycle MOP's with WAR dependencies, which can be scheduled into the same time step if the registers can be read and written simultaneously. A control dependency with a timing constraint, e.g., a delayed jump, has to be dealt with differently. The MOP's that are data-independent to the jump/branch MOP's can be scheduled into the time steps before the jump/branch MOP's or the delay slots after the jump/branch MOP's. The length of the delay slots is determined by the timing constraint. For example, in Table VI, the independent MOP `MO5` is scheduled into time step 4, which is the delay slot of the jump `MO6`.

Second, the instruction word width and the hardware resources consumed by the instructions have to be no larger than what are specified by the designer. Third, the size of the instruction set has to be no more than what the opcode field can afford.

[2] Refer to the footnote ‡. of Table II for the meaning of the notation.

D. Objective Function

General speaking, a richer instruction set may result in more compact and efficient compiled code. On the other hand, the larger the instruction set size, the more complex the decoding circuitry, and the more time the hardware designers spend in design and verification. The same trends hold true in the compiler side as well. Therefore, an objective function is necessary to control the performance/cost trade-off.

The goal of our design system is to minimize the objective function. The objective function is a function of the cycle count C and instruction set size S, where C represents the performance metrics, how many cycles the benchmarks execute on the target machine, and S represents the cost metrics. An interesting objective function suitable for our purpose is the following equation.

$$\text{Objective} = (100/P) \cdot \ln(C) + S. \tag{1}$$

This is an integral form, derived by Holmer in [4], of the statement "a new instruction will be accepted if it provides a $P\%$ performance improvement," which tries to balance the instruction set size with the performance gain. Other types of objective functions can be used with the design system as well.

Note that in our formulation, the design constraints are checked separately, and are not captured in the objective function.

V. Simulated Annealing Algorithm and the Design Flow

Although we have formulated the instruction set design problem as a scheduling problem, it is indeed more difficult than a regular scheduling problem, because we have to control the number of unique patterns (instruction set) in the time steps during the scheduling, in addition to the dependency and performance/cost constraints. Also, the problem size is usually much larger than regular scheduling problems since the application benchmarks may easily contain thousands of MOP's to be scheduled.

We propose an efficient solution to the problem based on a simulated annealing scheme. An initial design state consisting of an initial schedule and its derived instruction set (generated by a preprocessor) is given to the design system, and then a simulated annealing process is invoked to modify the design state in order to optimize the objective function, until the design state achieves an equilibrium state.

```
/* Basic simulated annealing process */
1: GIVEN: design state S, current temperature T, max. movement M;

2: while (not achieving equilibrium state)
3: { C=0;
4:   while (C < M)
5:   { if (violate constraints)Resolve_Constraint_Violation  (S, S_next);
6:     else Generate_Next_State  (S, S_next, T);

7:     if (Accept_Next_State  (cost(S), cost(S_next), T)) then S= S_next;
8:     else S= S;

9:     C = C + 1;
10:  };
11:  T = Update(T);
12: }
```

Fig. 7. The basic simulated annealing algorithm.

Fig. 7 lists the basic structure of our simulated annealing algorithm. In the outer `while` loop are the operations performed at each temperature point T. The temperature T is updated at the end of the operations. At each temperature, several movements (changes of the design state) are generated by the inner `while` loop. The number of movements (M) generated is specified by the designer.

In the following subsection, we present the move operators (Section V-A) and heuristics (Section V-B) for the procedures **Resolve_Constraint_Violation** and **Generate_Next_State**, the cooling schedule (Section V-C) for **Update**, and the move acceptance rules (Section V-D) in **Accept_Next_State**. Finally, we present the global design flow in Section V-E.

A. Move Operators

The move operators change the design state. They provide methods of manipulating the MOP's and time steps. The move operators can be characterized into three groups.

Manipulation of the Instruction Semantics and Format: The first group manipulates the instruction semantics and format of a selected time step. There are five move operators in this group.

Unification: Unify two register accesses in the MOP's; i.e., they always access the same register. For example, the specification of $R_1 = R_2$ in our previous example of the increment instruction `inc(R)` is a result of the 'unification' operator. The effects of this operator are the decreases in the instruction word width and register read/write ports.

Split: Cancel the effect of the 'unification' operator. Two register accesses that are previously unified to the same register are made independent. The effects of this operator are the increases in the instruction word width and register read/write ports.

Implicit value: Bind a register specifier to a specific register, or an immediate data field to a specific value. The specific values are the instantiated values in the MOP's of the selected time step. For example, the specification of `Immed = 1` in the instruction `inc(R)` is a result of this operator. The effect of this operator is the decrease in the instruction word width.

Explicit value: Cancel the effect of the 'implicit value' operator. Instruction fields that are previously bound to specific values are made explicit; i.e., their values are assigned by the compiler and are specified in the regular instruction fields. The effect of this operator is the increase in the instruction word width.

Generalization: If the current instruction format of the selected time step contains encoded operands, make these operands general and become explicit in the instruction fields. The effects of this operator are increased instruction word width and hardware resources.

Manipulation of MOP's Locations: The second group of move operators involves the movement of the MOP's. There are four move operators in this group, which are all subject to the data/control dependencies and delay constraints when moving MOP's. The target MOP's and time steps can be selected randomly or with the guidance of heuristics.

Interchange: Interchange the locations of two MOP's from different time steps. This operator changes the semantics and formats of the two instructions in the corresponding time steps.

Displacement: Displace a MOP to another time step. This operator simplifies the semantics and format of the instruction in the original time step, and enriches the semantics and format of the other instruction in the destination time step.

Insertion: Insert an empty time step after or before the selected time step and move one MOP to the new time slot. This operator simplifies the semantics and formats of instructions in the selected and new time steps, and increases the cycle count.

Deletion: Delete the selected time step if it is an empty one. This operator decreases the cycle count.

In our current implementation, if the selected MOP's contain unified or implicit fields, these fields are restored to the original forms (generalized, explicit) before the move operators in this group are applied to the MOP's. In addition to the aforementioned effects, these move operators may changes the resource usage in the selected time steps as well.

Micro-Architecture-Dependent Operators: The third group of move operators includes methods that explore the special properties of the target micro-architecture. These move operators are provided by the designer as part of the micro-architecture specification.

For example, if the target micro-architecture provides both register file $\rightarrow$ functional unit $\rightarrow$ register file, and register file $\rightarrow$ register file data paths, then the designer can specify that the following MOP's (`rrai` and `rr`) are functionally equivalent and can be transformed from one to another

```
rrai:  R1 ← R2+Immed(Immed = 0)
rr:    R1 ← R2.
```

These MOP's have different costs in hardware and instruction format. While `rrai` uses a functional unit and consumes an additional instruction field for the immediate data, `rr` uses a direct bus between the read and write ports of the register file. When discovering an `rrai` MOP with its immediate data being zero, the design system can map this MOP to the equivalent `rr` MOP, or vice verse.

An Example: Changing the Design State with Move Operators: We demonstrate how the move operators are used to change design states. Here we show a sequence of move

TABLE VII
THE DESIGN STATE AFTER THE APPLICATION OF THE FIRST MOVE OPERATOR

Schedule		Instruction Semantics				Instruction Fields		Costs	
Time step	MOP IDs	RTLs	MOP type IDs	Encoded fields	Inst. name	Format	Field values	Hardware cost	Inst. word width
1	mo1, mo2	$m(R_1+D_1)<-R_2+D_2$; $R_3<-R_4+I$	mrad, rrai		inst11	R_1, R_2, R_3, R_4, D_1, D_2, I	r2, r2, r0, r2, 0, 0, 0	4R, 1W, 1M, 3F	68
2									
3	mo3	$m(R_1+D_1)<-R_2+D_2$	mrad	$D_1=D_2$, $R_1=R_2$	inst3	R_1, D_1	r2, 1	1R, 1M, 1F	27
4	mo4	$R_1<-R_2+I$	rrai		inst4	R_1, R_2, I	r1, r2, 1	1R, 1W, 1F	32
5	mo5	$R_1<-R_2+1$	rrai		inst4	R_1, R_2, I	r2, r2, 2	1R, 1W, 1F	32
6	mo6	pc<-pc+D	jd		inst5	D	1024		22
7	nop		nop		inst6				6
Cycle count 7		Instruction set size 5				Hardware cost **4R, 1W, 1M, 3F**		Max. instruction width = **68**	

TABLE VIII
THE DESIGN STATE AFTER THE APPLICATION OF THE FIFTH MOVE OPERATOR

Schedule		Instruction Semantics				Instruction Fields		Costs	
Time step	MOP IDs	RTLs	MOP type IDs	Encoded fields	Inst. name	Format	Field values	Hardware cost	Inst. word width
1	mo1, mo2	$m(R_1+D_1)<-R_2+D_2$; $R_3<-R_4+1$	mrad, rrai	$D_1=D_2=I$, $R_1=R_2=R_4$	inst7	R_1, R_3, D_1	r2, r0, 0	1R, 1W, 1M, 1F	32
2									
3	mo3	$m(R_1+D_1)<-R_2+D_2$	mrad	$D_1=D_2$, $R_1=R_2$	inst3	R_1, D_1	r2, 1	1R, 1M, 1F	27
4	mo4	$R_1<-R_2+I$	rrai		inst4	R_1, R_2, I	r1, r2, 1	1R, 1W, 1F	32
5	mo5	$R_1<-R_2+1$	rrai		inst4	R_1, R_2, I	r2, r2, 2	1R, 1W, 1F	32
6	mo6	pc<-pc+D	jd		inst5	D	1024		22
7	nop		nop		inst6				6
Cycle count 7		Instruction set size 5				Hardware cost **1R, 1W, 1M, 1F**		Max. instruction width = **32**	

TABLE IX
THE DESIGN STATE AFTER THE APPLICATION OF THE SEVENTH MOVE OPERATOR

Schedule		Instruction Semantics				Instruction Fields		Costs	
Time step	MOP IDs	RTLs	MOP type IDs	Encoded fields	Inst. name	Format	Field values	Hardware cost	Inst. word width
1	mo1, mo2	$m(R_1+D_1)<-R_2+D_2$; $R_3<-R_4+I$	mrad, rrai	$D_1=D_2=I$, $R_1=R_2=R_4$	inst7	R_1, R_3, D_1	r2, r0, 0	1R, 1W, 1M, 1F	32
3	mo3, mo4	$m(R_1+D_1)<-R_2+D_2$; $R_3<-R_4+I$	mrad, rrai	$D_1=D_2$, $R_1=R_2$	inst31	R_1, R_3, R_4, D_1, I	r2, r1, r2, 1, 1	2R, 1W, 1M, 2F	48
4									
5	mo5	$R_1<-R_2+I$	rrai		inst4	R_1, R_2, I	r2, r2, 2	1R, 1W, 1F	32
6	mo6	pc<-pc+D	jd		inst5	D	1024		22
7	nop		nop		inst6				6
Cycle count 6		Instruction set size 5				Hardware cost **2R, 1W, 1M, 2F**		Max. instruction width = **48**	

TABLE X
THE DESIGN STATE AFTER THE APPLICATION OF THE ELEVENTH MOVE OPERATOR

Schedule		Instruction Semantics				Instruction Fields		Costs	
Time step	MOP IDs	RTLs	MOP type IDs	Encoded fields	Inst. name	Format	Field values	Hardware cost	Inst. word width
1	mo1, mo2	$m(R_1+D_1)<-R_2+D_2$; $R_3<-R_4+I$	mrad, rrai	$D_1=D_2=I$, $R_1=R_2=R_4$	inst7	R_1, R_3, D_1	r2, r0, 0	1R, 1W, 1M, 1F	32
3	mo3, mo4	$m(R_1+D_1)<-R_2+D_2$; $R_3<-R_4+I$	mrad, rrai	$D_1=D_2=I$, $R_1=R_2=R_4$	inst7	R_1, R_3, D_1	r2, r1, 1	1R, 1W, 1M, 1F	32
5									
6	mo6	pc<-pc+D	jd		inst5	D	1024		22
7	mo5	$R_1<-R_2+1$	rrai		inst4	R_1, R_2, I	r2, r2, 2	1R, 1W, 1F	32
Cycle count 5		Instruction set size 3				Hardware cost **1R, 1W, 1M, 1F**		Max. instruction width = **32**	

operators which transforms the schedule and instruction set (one design state) in Table V to the ones (a better design state) in Table VI. The sequence is

1) DISPLACEMENT: displace the MO2 from time step 2 to 1 (as shown in Table VII).
2) UNIFICATION: unify fields D_1 and D_2 in the time step 1.
3) UNIFICATION: unify fields D_1 and I in the time step 1.
4) UNIFICATION: unify fields R_1 and R_2 in the time step 1.
5) UNIFICATION: unify fields R_1 and R_4 in the time step 1 (as shown in Table VIII).
6) DELETION: delete the empty time step 2.
7) DISPLACEMENT: displace the MO2 from time step 4 to 3 (as shown in Table IX).
8) DELETION: delete the empty time step 4.
9) UNIFICATION: unify fields D_1 and I in the time step 1.
10) UNIFICATION: unify fields R_1 and R_4 in the time step 1.
11) DISPLACEMENT: displace the MO5 from time step 5 to 7 (as shown in Table X)
12) DELETION: delete the empty time step 5.

Tables VII–X show the resulted schedule and instruction set for the design state after the first, fifth, seventh, and eleventh move operators are applied, respectively. After the twelfth move operator is applied, the design state in Table VI can be obtained. In the last row of the tables we show the cycle count, instruction set size, hardware cost, and instruction word width for the corresponding design states. The deleted time steps are shown as shaded rows. The time steps in which the move operators are applied are emphasized with heavy rectangles around the time step indices. The elements in the design state that are modified by the move operators are listed with bold face. Note that, for ease of illustration, we use the original time step indices in Table V in the above sequence when referring to selected time steps. In the implementation, the indices of time steps have to be adjusted when time steps are inserted or deleted such that the delay constraints between MOP's can be correctly maintained.

Note that there are more than one sequence which accomplish the same design state transition. How such sequences are formed depends on the design algorithm. In our simulated annealing scheme, the move operators are selected with a mix of random and heuristics strategies as described in Section V-B.

B. Heuristics for Target Selection

During each iteration, the design space is examined whether it violates design constraints. If yes, a time step is randomly selected from a pool of time steps that violate constraints. If more than one constraint is violated, the resource violation gets higher priority than the instruction word width violation since a movement that resolves the former may resolve the latter as well.

Depending on the type of the constraints, one of the following rules is applied.

1) If the instruction word width constraint is violated, apply randomly one of the move operators: 'unification,' 'implicit value,' 'interchange,' 'displacement,' or 'insertion';
2) If the resource constraint is violated, apply randomly one of the move operators: 'unification' (only when the register port constraint is violated), 'implicit value,' 'displacement,' or 'insertion.'

When the current design space does not violate any constraint, all move operators are eligible for changing the design state. In this case, a basic block is selected with the probability $Selection_i$, which is the selection weight of a basic block i and is defined by the following equation, where F_i is the execution frequency of the basic block i in the benchmark, N_i is the number of MOP's in the basic block i, and the summation in the denominator is the total number of MOP's executed in the benchmark. Therefore, the selection weight is intended to denote the degree of importance of a basic block in the benchmark. A time step is then randomly chosen from the selected basic block, and one move operator is randomly selected and applied to the time step.

$$\text{Selection}_i = \frac{F_i \cdot N_i}{\sum_i F_i \cdot N_i}. \tag{2}$$

C. Cooling Schedule

The cooling schedule is controlled by five parameters.

1) The initial temperature (T_0) should be high enough so that there is no rejection for high-cost states at the initial temperature. A simple heuristic to set the initial temperature is to start the simulated annealing algorithm with a given initial temperature. If some states are rejected at the initial temperature, then the value of the initial temperature is doubled. The trial run is repeated until the ideal initial temperature is obtained.
2) The number (M) of movements tried at each temperature is proportional to the total number (O_{ps}) of MOP's in the benchmarks, typically five times, which is given by the designer.
3) The next temperature is 90% of the current temperature.
4) A low temperature point is defined such that a special handling routine can be applied to stabilize the design state. The special handling routine stabilizes the design state by adopting move acceptance rules that are different from the ones in high temperatures. The move acceptance rules are described in Section V-D.
5) The annealing process terminates when the design state stays unchanged for a certain (e.g., four) consecutive temperature points. The number of the consecutive stable temperature points is given by the designer.

The complexity of the algorithm is mainly determined by the cooling schedule and the data structures used to represent the design state. As discussed previously, the number of movements tried at each temperature is proportional to the total number (O_{ps}) of MOP's in the benchmarks; the complexity of accessing the data structures, in our current implementation, is proportional to O_{ps} as well. Therefore, the complexity of the algorithm at each temperature is of the order of O_{ps}.[2] This complexity can by lowered by using more efficient data structures in our future implementation.

To derive the global complexity formally, we need to determine the total number of temperature points, which is difficult to analyze since it is affected by both the problem size and the nature of the benchmarks. However, our empirical study shows that the global complexity of the algorithm is roughly about the order of O_{ps}.[3]

D. Move Acceptance

At high temperatures, a movement that satisfies one of the following conditions is definitely accepted.

1) The movement reduces the value of the objective function;
2) The movement is a result of constraint resolution; i.e., it is a necessary movement in order to resolve some constraint violations.

Otherwise, a movement is accepted with the probability of $\exp^{-(\Delta/T)}$ where Δ is the increased value of the objective function and T is the current temperature.

At low temperatures, a different strategy is adopted to stabilize the design state. A movement is accepted when either one of the following conditions is true.

1) The movement generates a new state which does *not* violate any design constraint and has lower objective value;
2) The movement is a result of constraint resolution. This condition is same as the one at high temperatures.

Otherwise, only those movements that generates new states which do *not* violate any design constraint are accepted with the probability of $\exp^{-(\Delta/T)}$.

In addition, the current best design state is kept when the algorithm decides to accept inferior design states. At the end of each temperature point, if the reached design state is inferior to the current best state, the design state falls back to the current best state with the probability $1 - T/T_i$ where T_i is the initial temperature.

E. Design Flow Based on the Simulated Annealing Algorithm

The instruction set design process consists of three major steps.

1) The given application is translated to dependency graphs of MOP's which are supported by the given architecture template. This translation is performed in two steps. First, the application, written in a high-level language, is translated into an intermediate representation by the compiler of the high-level language (in our current environment, the Aquarius Prolog Compiler [21]). Second, a retargetable MOP mapper, consulting the given architectural template specified with the language described in Section III-B, transforms the intermediate representation into the dependency graphs of MOP's.

TABLE XI
THE MOP'S AND THEIR DEPENDENCIES OF A LIST-CREATING APPLICATION

MOP ID	Type ID	RTLs*	MOP ID	Type ID	RTLs
1	rrait	r0 <- lst^(r1 + 0)	10	rrai	r1 <- r1 + 1
2	rit	r2 <- atm^36	11	rit	r2 <- atm^(-1)
3	mr	m(r1) <- r2	12	mr	m(r1) <- r2
4	rrai	r1 <- r1 + 1	13	rrai	r1 <- r1 + 1
5	rrait	r2 <- lst^(r1 + 1)	14	rrait	r3 <- var^(r1 + 0)
6	mr	m(r1) <- r2	15	mr	m(r1) <- r3
7	rrai	r1 <- r1 + 1	16	rrai	r1 <- r1 + 1
8	rit	r2 <- atm^37	17	rrai	r1 <- r1 + 1
9	mr	m(r1) <- r2	18	jd	pc <- pc +1024
Dependencies bf: before ctl: control	bf(1,4). bf(2,3). bf(2,5). bf(3,5). bf(4,5). bf(4,6). bf(4,7).	bf(5,6). bf(5,7). bf(5,8). bf(6,8). bf(7,10). bf(7,9). bf(8,11).	bf(8,9). bf(9,11). bf(10,12). bf(10,13). bf(11,12). bf(13,14). bf(13,15).		bf(13,16). bf(14,15). bf(14,16). bf(16,17). ctl(18).

*. bit width: tag=2, Immed=14

TABLE XII
32-B INSTRUCTION SET

Instruction name	Instruction fields	RTLs	MOP type ID*	Encoded fields*
inst11	R_1, D	pc <- pc + D; R_1 <- R_2 + I	jd, rrai	I=1, R_1=R_2
inst12	R_1, R_2, I	m(R_1) <- R_2; R_3 <- R_4 + I	mr, rrai	R_1=R_3=R_4
inst13	R, T, I	R <- T ^ I	rit	
inst14	R_1, R_2, T, I	R_1 <- T ^ (R_2 + I)	rrait	

*. The right two columns specify the binary tuples for the corresponding instructions.

TABLE XIII
COMPILED CODE WITH THE 32-B INSTRUCTION SET

Time Step	Compiled Code	Time Step	Compiled Code	Time Step	Compiled Code
1	inst13(r2, atm, 36)	5	inst12(r1, r2, 1)	9	inst12(r1, r2, 1)
2	inst14(r0, r1, lst, 0)	6	inst13(r2, atm, 37)	10	inst14(r3, r2, var, 0)
3	inst12(r1, r2, 1)	7	inst12(r1, r2, 1)	11	inst11(r1, 1024)
4	inst14(r2, r1, lst, 1)	8	inst13(r2, atm, -1)	12 (delay slot)	inst12(r1, r3, 1)

TABLE XIV
64-B INSTRUCTION SET

Instruction name	Instruction fields	RTLs	MOP type ID	Encoded fields
inst15	R_1, R_2, R_3, R_4, D, I	pc <- pc + D; m(R_2) <- R_3; R_3 <- R_4 + I	jd, mr, rrai	
inst16	R_1, R_2, R_3, R_4, I	m(R_2) <- R_3; R_3 <- R_4 + I	mr, rrai	
inst17	R_1, R_2, R_3, T, I_1, I_2	R_1 <- T ^ I_1; R_2 <- R_3 + I_2	rit, rrai	
inst18	R_1, R_2, R_3, T_1, T_2, I_1, I_2	R_1 <- T_1 ^ I_1; R_2 <- T_2 ^ (R_3 + I_2)	rit, rrait	
inst14	R_1, R_2, T, I	R_1 <- T ^ (R_2 + I)	rrait	

TABLE XV
COMPILED CODE WITH THE 64-B INSTRUCTION SET

Time Step	Compiled Code	Time Step	Compiled Code
1	inst18(r2, r0, r1, atm,lst, 36,0)	6	inst16(r1, r2, r1, r1, 1)
2	inst16(r1, r2, r1, r1, 1)	7	inst18(r2, r3, r1, atm, var, -1,0)
3	inst14(r2, r1, lst, 1)	8	inst15(r1, r2, r1, r1, 1024, 1)
4	inst16(r1, r2, r1, r1, 1)	9 (delay slot)	inst16(r1, r3, r1, r1, 1)
5	inst17(r2, r1, r1, atm, 37, 1)		

2) A preprocessor generates a simple schedule for the MOP's. The schedule is obtained by serializing the dependency graphs. An initial instruction set is then derived from the schedule. This is done by directly mapping time steps in the schedule into instructions without encoding any operand. The obtained schedule and instruction set constitute the initial design state.
3) The simulated annealing algorithm is invoked to optimize the design state. Several trial runs of the algorithm may be necessary to adjust the cooling schedule.

The best instruction set, micro-architecture, and assembly code which minimize the objective function can be obtained after the design state reaches the equilibrium state.

We have implemented the algorithm and its supporting tools into our design system ASIA (Automatic Synthesis of Instruction-set Architectures). It consists of about 8000 lines of Prolog code.

VI. EXPERIMENTS

We first demonstrate our technique with a small, illustrative example, and then with Prolog application benchmarks.

A. A Small Example

In this example, we assumed the target architecture in Table II, the instruction field specification in Table I with smaller bit widths for tag (2 b) and immediate (14 b), and the delay specification in Table IV. The example used in this subsection is a small application which sets up a list of two elements in Prolog. It consists of 18 MOP's. Table XI lists the MOP's and their dependencies. The `bf` clauses in the last row specify the *before* dependencies between MOP's. For example, `bf(1,4)` constrains that MOP 1 has to be scheduled in a time step earlier than MOP 4's. The `ctl(18)` clause specifies that the MOP 18 changes the control flow. Note that the control flow change has one cycle delay. We synthesized the 32-b and 64-b instruction sets, with the resource constraints $\langle 3R, 1W, 2M, 1F\rangle$ and $\langle 6R, 4W, 4M, 4F\rangle$,[3] respectively. The objective function used is EQ 1 with $P = 1$.

The synthesized 32-b instruction set is listed in Table XII, consisting of four instructions. Note that two instructions `inst11` and `inst12` contain encoded fields, in order to satisfy the required 32-b word constraint. This instruction set compiles the application into 12 cycles, as shown in Table XIII. Note that time step 12 is the delay slot of `inst11` which changes the control flow. An independent instruction `inst12` is scheduled into time step 12 to make use of the delay slot.

Table XIV lists the 64-b instruction sets, consisting of five instructions. Most of the instructions have concurrent MOP's. Since 64 b are wide enough to accommodate all instruction fields, there is no encoded field required in this instruction set. The compiled code (Table XV) consists of 9 cycles, which is 3 cycles less than the 32-b one. Also note that the instruction `inst16` is scheduled to the delay slot of instruction `inst15` which changes the control flow.

[3]Refer to the footnote ‡. of Table II for the meaning of the notation.

TABLE XVI
RESULTS (OBJECTIVE FUNCTION = $100 - \ln(C) + S$)

Benchmark	# of MOPs, data dep., control dep.*	Instruction word width†	Design results			Performance of the algorithm	
			Cycle (C)	Instruction set size (S)	Instruction set space	Time (minutes)	Memory (megabytes)
con1	183, 136, 24	32	135	29	1275	56	2.1
		48	93	38	3733	59	2.7
		64	89	35	3277	48	2.7
nreverse	245, 395, 11	32	169	17	540	69	2.1
		48	157	23	772	57	2.0
		64	154	22	688	48	2.0
query	391, 185, 68	32	305	24	478	95	2.0
		48	215	32	1742	103	2.3
		64	204	39	1445	89	2.3
circuit	1725, 1077, 274	32	1406	40	1710	1358	3.4
		48	1361	25	1389	1722	4.6
		64	1360	24	1362	4726	5.9

*. The number of control dependencies is counted as the total number of branch/jump MOPs.
†. The hardware constraints are 3R, 1W, 2M, 1F for 32-bit instructions; 6R, 3W, 2M, 3F for 48-bit instructions; 8R, 4W, 32M, 4F for 64-bit instructions

B. Prolog Application Benchmarks

In this subsection, experiments are presented to show the versatility and practicality of our tools by synthesizing instruction sets for some application benchmarks, with various design constraints and objective functions. Four benchmarks were selected from the Prolog Benchmark suite [18]. The benchmarks `con1` and `nreverse` are programs for list manipulation. The benchmark query is a program for database query. The benchmark `circuit` maps boolean equations into logic gates. The second column in Table XVI lists the characteristics of the benchmarks, including the numbers of MOP's, data-related dependencies, and control dependencies in the benchmarks. The number of MOP's represents the size of the benchmark; the number of data-related dependencies is related to the degree of parallelism available within the benchmark; the number of control dependencies indicates the degree of the impact of the branch/jump delays on the benchmark.

We assumed that every basic block executes once. We assumed the target architecture in Table II and the instruction field specification in Table I. The delay constraints for control and memory operations are one and zero, respectively. The experiment was conducted on a HP750 workstation with 256 MBytes of memory.

For each benchmark, we synthesized its 32-b, 48-b, and 64-b instruction sets, respectively. We were interested in how the instruction sets vary with bit widths. Table XVI lists the results, synthesized under the objective function with $P = 1$ in (1). For all three benchmarks, as we had expected, the cycle decreases when the instruction word width increases. However, we observed a smaller gain in `nreverse` and `circuit`. This can be explained by their larger ratios of the number of data dependencies to the number of MOP's. Most of the MOP's depend on each other such that there is less parallelism available when packing MOP's into instructions.

In general, the size of the instruction set also increases when the instruction word width increases. This is due to the fact that wider words can accommodate more MOP's, resulting in richer and more powerful instructions. However, the 48-b instruction sets are 'embarrassing' designs for `con1` and `nreverse`. Their instruction set sizes are larger, and their performance is worse than their 64-b alternatives in compiling the benchmarks. The 48 b are not wide enough for these benchmarks to accommodate the most frequent MOP patterns, for which 64 b are sufficient. Therefore, the design process has to specialize the general forms of some powerful instructions into several distinct instructions by making fields implicit or unifying register ports, in order to satisfy the bit width constraint.

TABLE XVII
PERFORMANCE COMPARISON WITH A MANUALLY DESIGNED INSTRUCTION SET

Benchmark	Instruction set	Hardware resources*	Cycle (C)	Instruction set size (S)	Objective value (smaller is better)
con1	BAM†	3R, 1W, 2M, 1F	150	22	523
	ASIA‡	3R, 1W, 2M, 1F	135	29	520
nreverse	BAM	3R, 1W, 2M, 1F	178	16	534
	ASIA	2R, 1W, 1M, 1F	169	17	530
query	BAM	3R, 1W, 2M, 1F	368	22	613
	ASIA	3R, 1W, 2M, 1F	305	24	596
circuit	BAM	3R, 1W, 2M, 1F	1453	24	752
	ASIA	3R, 1W, 2M, 1F	1406	40	764

*. The resource constraints given to ASIA is the same as in the VLSI-BAM processor: 3R, 1W, 2M, 1F.
†. BAM refers to the instruction set that was manually designed for the VLSI-BAM processor.
‡. ASIA refers to the instruction set synthesized by the tools (ASIA) reported in this paper.

In the 'Instruction set space' column we examined the number of instruction candidates explored by the design process. The numbers, much larger than the final instruction sets, show that the design process was able to explore a rich design space for the best candidates while keeping the size of the design space manageable.

In the two right most columns we also list the run time and memory usage of our algorithm, which show that our tools were able to synthesize instructions for application benchmarks within reasonable time and consume a modest amount of memory.

In Table XVII we compared the synthesized 32-b instruction sets for these benchmarks with the BAM instruction set, which was designed for the VLSI-BAM microprocessor by the Aquarius Project at the University of California at Berkeley. The VLSI-BAM microprocessor has RISC-style instructions plus some powerful instructions to support efficient logic computation such as Prolog. The benchmarks were compiled with the BAM instruction set, and we measured the number of distinct instructions used (in the 'Instruction set size' column), and the number of cycles to execute the compiled code (in the 'Cycle' column). The programs were compiled by the Aquarius Prolog Compiler, with the post-phase optimization phase turned off.[4] The experiments show that the synthesized instruction sets produced more compact codes for all four benchmarks, with 10%, 5%, 17%, and 3% reduction in the code size, respectively. This was achieved at the cost of a small number of additional instructions (7, 1, and 2 for `con1`, `nreverse`, and `query`, respectively), except in `circuit` where 16 additional instructions are required. We then used Holmer's objective function '$100 \cdot \ln(C) + S$' to evaluate

[4]The post-phase optimization of the Aquarius Prolog Compiler alters the classic definition of the basic block. Due to the time limit, we were not able to modify our tools to accommodate such change.

TABLE XVIII
SOME SYNTHESIZED INSTRUCTIONS

Instruction word width	RTLs*	Meaning
32	$m(R_1) \leftarrow R_2$; $R_1 \leftarrow R_1+D$	push†
	if (tf=1) { $m(R_1) \leftarrow R_2$; $R_1 \leftarrow R_1+D$}	conditional push†
	if (tag(R_1=T_1) { pc ← pc + D_1); if (tag(R_1=T_2) { pc ← pc + D_2 }	switch on tag†
48	tf ← R_1 OP. R_2; pc ← I	compute condition and jump
	$m(R_2) \leftarrow R_3$; if (tf=1) { $m(R_1) \leftarrow R_2$; $R_1 \leftarrow R_1+D$}	store and conditional push (with a shared register)
	$m(R_1) \leftarrow R_2$; $R_3 \leftarrow R_4+I$	store and add
64	$m(R_3) \leftarrow R_4$; if (tf=1) { $m(R_1) \leftarrow R_2$; $R_1 \leftarrow R_1+D$}	store and conditional push
	$m(R_1) \leftarrow R_2$; $R_3 \leftarrow R_4+I$	store and add
	$R_1 \leftarrow T \wedge I$; $R_2 \leftarrow R_3 + I$	tag data and add

*. Notations: 1). The RTLs in an instruction are executed simultaneously; 2). *tf*: a one bit latch which holds the truth value of a logic computation; 3). The operator '^' appends a tag to a value before the value is sent to a destination.
†. These three instructions can be found in the BAM instruction set.

the global performance/cost trade-offs for both instruction sets and found that in most cases (`con1`, `nreverse`, and `query`) the synthesized ones yield better results, as indicated in the 'Objective value' column (smaller values are better). It is possible to improve the result of `circuit` by adjusting the initial temperature and the cooling schedule in our future experiment. We also compared the hardware resources used by both instruction sets. They both use the same amount of resources, except in the `nreverse` case our synthesized instruction set uses one less register read port and one less memory port than BAM does. This experiment shows that ASIA is capable of competing with manually designed instruction sets within our collection of benchmarks. Further studies will be needed to investigate its competence in more general cases.

Table XVIII shows some interesting instructions synthesized for the benchmark `query`. They are selected from the 32-b, 48-b, and 64-b instruction sets, respectively. For ease of illustration, we do not list the binary tuples for these instructions; instead, we describe the RTL's of these instructions directly. In the RTL's, the register sharing is indicated by using the same register index. Note that the 32-b version of the instructions can be found in the BAM instruction set as well. This fact provides the BAM designers with more confidence about their instruction set, since some of the instructions that they considered 'powerful' retain their existence when the instruction set is designed by other independent designers (in this case, the ASIA design automation system). This observation suggests that ASIA, in addition to its original purpose (an automatic design tool), can be used as a verification tool for designers to verify their manually designed instruction sets as well.

Finally, Table XIX shows how the synthesized instruction sets vary with the objective functions. In this experiment we synthesized 32-b instruction sets for the benchmark `query` with two objective functions: one with $P = 1$, another with $P = 5$. The latter assigns less importance to the cycle count. Therefore, the tools focused on reducing the instruction set size, resulting in 7 instructions less, but 16 cycles more than the former case.

TABLE XIX
INSTRUCTION VARIATION DUE TO DIFFERENT OBJECTIVE FUNCTIONS

Objective function	Cycle (C)	Instruction set size (S)
100*ln(C)+S.....P=1 in EQ 1	135	29
20*ln(C)+S.....P=5 in EQ 1	151	22

VII. CONCLUSION

We have presented a design automation system ASIA (Automatic Synthesis of Instruction-set Architectures) that synthesizes computer instruction sets from application benchmarks. The design problem is formulated as a modified scheduling problem. The benchmarks are represented as data/control flow graphs of MOP's. The MOP's are scheduled into time steps subject to constraints of dependencies, hardware resources, and instruction word width. Instructions are formed during the scheduling phase. A binary tuple is used to describe the semantics and formats of instructions. The binary tuple is the key idea which links the instruction formation to the scheduling process. In addition to the synthesized instruction sets, ASIA also generates the compiled codes for the given benchmarks, showing that how the instruction sets can be actually used to compile programs. An objective function of the cycle count and instruction set size is used to guide the design process, in order to balance the performance/cost trade-off. A simulated annealing algorithm is used to solve for the schedules. We have discussed the move operators suitable for our problem, and other issues such as cooling schedules and heuristics.

We have demonstrated the versatility and practicality of ASIA by conducting experiments on some application benchmarks, with various design constraints and objective functions. The tools used reasonable amount of CPU time and a modest amount of memory. It has been shown that our tools are capable of synthesizing powerful instruction sets. Many of them can be found in today's processors. Compared with manually designed instruction sets, the synthesized instruction sets produce more compact code and may require less hardware. The tools were able to explore a rich design space, and handle important design options such as the instruction word width, and performance/cost trade-off. We were able to explain the variation of the performance of the instruction sets on different benchmarks, based on the characteristics of the benchmarks. The experiments also show that ASIA, in addition to its original purpose in automating the design process, can be used by the designers to verify their manually designed instruction sets as well.

The current limitations include: First, the designers are required to specify the number of hardware resources, which may takes several iterations to find the best hardware allocation. Second, ASIA does not recognize the situation when the constraints are too loose, e.g., the instruction word is too wide or hardware resources are too rich. In this case, it is possible to suggest some partitioning of the constraints. For example, a 128-b instruction word can be realized as a single wide-word instruction or an abutting of several smaller instructions. Third, in our problem formulation, the concept of the basic block is used to partition benchmarks into small pieces.

However, there are other ways of partitioning benchmarks such as traces, and random segments [5]. What is the best way is unknown at this moment. Fourth, even though we have demonstrated that our algorithm is able to synthesize instruction sets from thousands of MOP's within 22 h, real world application benchmarks, such as system, CAD and simulation software, are usually much larger. How to manage problems of such sizes is an important issue. Fifth, the machine model is insufficient to account for the dynamic behavior of some modern architectures such as superscalar machines.

In the future, we will continue our efforts in ASIA and pursue the following issues: 1) improving the aforementioned limitations; 2) code generation for the synthesized instruction sets; 3) synthesis and comparison for application specific uniprocessors and VLIW processors; 4) design and synthesis of low-power instruction set architectures; and 5) analysis of architectural properties for application benchmarks.

ACKNOWLEDGMENT

The authors would like to thank B. Holmer, C.-L. Su, and the anonymous reviewers for their comments and suggestions in improving this work.

REFERENCES

[1] F. M. Haney, "ISDS—A program that designs computer instruction sets," in *Fall Joint Comput. Conf.*, 1969.
[2] P. Bose and E. S. Davidson, "Design of instruction set architectures for support of high-level languages," in *Proc. 11th Annual Int. Symp. Comput. Architecture*, 1984, pp. 198–206.
[3] J. P. Bennett, "A methodology for automated design of computer instruction sets," Ph.D. dissertation, Computer Laboratory, Univ. of Cambridge, England, 1988. Also available as Tech. Rep. 129.
[4] B. Holmer and A. Despain, "Viewing instruction set design as an optimization problem," in *Proc. Micro-24*, 1991.
[5] B. Holmer, "Automatic design of computer instruction sets," Ph.D. dissertation, Comp. Sci. Dept., Univ. of California, Berkeley, 1993.
[6] J. Sato *et al.*, "An integrated design environment for application specific integrated processor," in *Proc. ICCD*, 1991, pp. 414–417.
[7] I.-J. Huang, B. Holmer, and A. Despain, "ASIA: Automatic synthesis of instruction-set architectures," in *Proc. SASIMI Wkshp.*, Nara, Japan, Oct. 1993, pp. 15–22.
[8] I.-J. Huang and A. Despain, "Synthesis of instruction sets for pipelined microprocessors," in *Proc. 31st Design Automation Conf.*, June 1994, pp. 5–11.
[9] P. M. Kogge, *The Architecture of Pipelined Computers*. New York: McGraw-Hill, 1981.
[10] J. Pendleton *et al.*, "A 32-bit microprocessor for smalltalk," *IEEE J. Solid-State Circuits*, vol. SC-21, no. 5, pp. 741–749, Oct. 1986.
[11] M. Breternitz, Jr. and J. P. Shen, "Architecture synthesis of high-performance application-specific processors," in *Proc. Design Automation Conf.*, 1990, pp. 542–548.
[12] S. Devadas and R. Newton, "Algorithms for hardware allocation in data path synthesis," *IEEE Trans. Computer-Aided Design*, vol. 8, no. 7, pp. 768–781, July 1989.
[13] I.-J. Huang and A. Despain, "High level synthesis of pipelined instruction set processors and back end compilers," in *Proc. 29th DAC*, 1992, pp. 135–140.
[14] R. Cloutier and D. Thomas, "Synthesis of pipelined instruction set processors," in *Proc. 30th DAC*, 1993.
[15] G. Goossens *et al.*, "An efficient microcode compiler for application specific DSP processors," *IEEE Trans. Computer-Aided Design*, vol. 9, no. 9, pp. 925–937, Sept. 1990.
[16] S.-Z. Lin, C.-T. Hwang, and Y.-C. Hsu, "Efficient microcode arrangement and controller synthesis for application specific integrated circuits," in *Proc. ICCAD*, 1991.
[17] A. Kumar and S. Kumar, "Automatic synthesis of microprogrammed control units from behavior descriptions," *Proc. 26th DAC*, 1989, pp. 147–154.
[18] R. Haygood, "A prolog benchmark suite for aquarius," Comp. Sci. Dep., Univ. of California, Berkeley, Tech. Rep. UCB/CSD 89/509, 1989.
[19] B. Bush *et al.*, "The Berkeley abstract machine instruction manual," Advanced Computer Architecture Lab., Univ. of Southern California, Los Angeles, Internal Tech. Rep., 1990.
[20] B. Holmer *et al.*, "Fast Prolog with an extended general purpose architecture," *Proc. 27th Int. Symp. Comput. Architecture*, 1990, pp. 282–291.
[21] P. L. Van Roy, "Can logic programming execute as fast as imperative programming," Ph.D. dissertation, Comp. Sci. Dep., Univ. of California, Berkeley, 1990. Also available as Tech. Rep. UCB/CSD 90/600.

Ing-Jer Huang (S'89–M'95) received the B.S. degree in electrical engineering from the National Taiwan University, Taiwan, R.O.C., in 1986, and the M.S. and Ph.D. degrees in computer engineering from the University of Southern California, Los Angeles, in 1989 and 1994, respectively.

He is currently with the Institute of Computer and Information Engineering at National Sun Yat-Sen University, Taiwan, R.O.C. as an Associate Professor. His research interests include hardware/software co-design, high/system level synthesis, computer architecture, and VLSI system design.

Dr. Huang has published ten technical papers in his areas of research. He is a member of ACM.

Alvin M. Despain (S'58–M'65) received the B.S. (1960), M.S. (1962), and Ph.D. (1966) degrees in electrical engineering from the University of Utah, Salt Lake City.

He is currently with the University of Southern California, Los Angeles as the Powell Professor of Computer Engineering and as a Professor in the Computer Science and Electrical Engineering—Systems Departments. He has been an Assistant Research Professor with Utah State University, Logan, a Visiting Associate Professor with Stanford University, Stanford, CA, a Professor with the University of California at Berkeley, and has been with USC since 1989. He is a pioneer in the study of high-performance computer systems for symbolic calculations. His research group builds experimental software and hardware systems including computers, custom VLSI processors, and multiprocessor systems. The goal is to determine principles for the design of high-performance computer systems. His research interests include computer architecture, multiprocessor and multicomputer systems, logic programming, and design automation.

Dr. Despain is a member of ACM and AAAI.