AN APPROACH TO APPLYING THE CONTOUR MODEL TO THE DESIGN OF A HYPOTHETICAL MULTIPLE-REGISTER-WINDOW ARCHITECTURE FOR THE BLOCK-STRUCTURED PROCESS

By

HSU-KU BRUCE YING

Bachelor of Science

Chung-Yuan Christian University

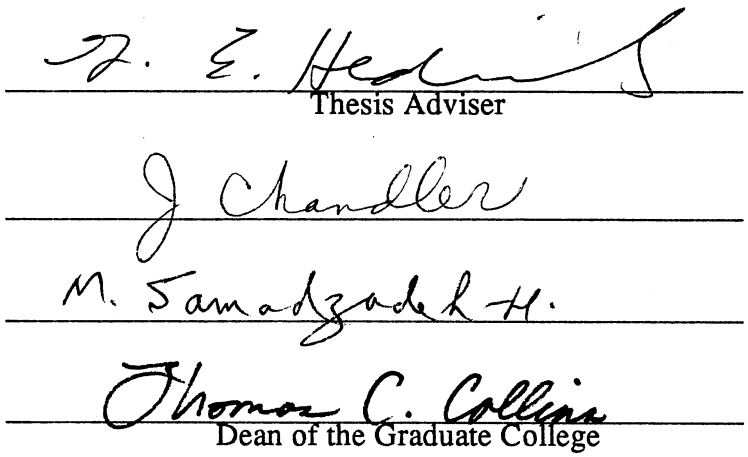
Chungli, Taiwan

1983

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
May, 1992

AN APPROACH TO APPLYING THE CONTOUR MODEL
TO THE DESIGN OF A HYPOTHETICAL
MULTIPLE-REGISTER-WINDOW
ARCHITECTURE FOR THE
BLOCK-STRUCTURED
PROCESS

Thesis Approved:



Thesis Adviser

Dean of the Graduate College

PREFACE

The concepts described in this thesis are towards the implementation of the basic functions of a pipelined, load/store, multiple-register-window and scalar-oriented uniprocessor architecture. During the formation phase of these concepts, I am glad to have the opportunity to investigating the interrelation of computer architectures, data structures and systems programming, which are the fundamentals underlying virtually every software design. I also took pleasure in learning AWK and C++ programming languages (only the elementary things of the latter, however) for the simulation conducted in this thesis and the UNIX® document formatting/typesetting tools for the preparation of the text and figures presented in this thesis on the UNIX®-based Perkin-Elmer 3230 computer system of the Computer Science Department.

I wish to express sincere gratitude to Dr. George E. Hedrick, my thesis adviser, and to Dr. Mansur Samadzadeh and Dr. John P. Chandler for serving on my graduate committee. The useful informations offered by Dr. Samadzadeh vitally benefited this thesis as well as my graduate study.

Many thanks are due to Mr. Mark Vasoll and Mr. Roland Stolfa, who are always helpful in the matter of accessing the departmental computing facilities. I am also grateful to Dr. John B. Johnston of New Mexico State University for his kindness. He sent me a stack of materials describing his work on the contour model (the contour model architecture and the contour model assembly language) at New Mexico State University. Special thanks are due to my friend and classmate, Mr. Yuh-Ching Su, who kindly offered me extensive access to his microcomputer and Borland C++® software for debugging the simulator program. And hearty appreciation goes to Van Morrison, whose songs were my companion through the dismal days.

My deepest gratitude is to my parents, who unconditionally support my education in both spiritual and financial terms and extend their support throughout my graduate study, even though they have been in a very difficult financial situation for many years. I am deeply sorry for aggravating their suffering because of this overdue thesis.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

NOMENCLATURE

CIP	the current instruction pointer
CIR	the current instruction register
CWP	the current window pointer
FIP	the fetch instruction pointer
HMA	a Hypothetical Multiple-Register-Window Architecture
SWP	the saved window pointer
WAV	the window activation vector
XIP	the execute instruction pointer

CHAPTER I

INTRODUCTION

Motivation

This thesis work expands the results of two papers: the contour model, which was proposed by Johnston [Johns71]; and the Berkeley RISC architecture, which was proposed by Patterson and Sequin [PaSe82]. The study of the implementation of the Berkeley RISC [Kate85] motivated the author to develop a scheme which takes advantage of both models and apply it to a microprocessor-based computer design which makes extensive use of its on-chip storage and expects high performance with the synergy of hardware and software.

Review of Literature

The contour model is an attempt to close the gap between the semantic mechanisms of programming languages and the computer architectures to be implemented in hardware. It contains a powerful exposition of the data structures of block-structured processes. However, the contour model's methodology is independent of the memory techniques for achieving high speed data access, which is an important aspect in computer architectures. Fortunately, the concept of contour cells may be converted into a hardware implementation using high-speed on-chip storage.

In the development of a high-performance computer architecture, the memory subsystem has been a bottleneck [HaLi91, Hsu87, SoFr91]. This is because off-chip memory access is slow, and more importantly, the pin bandwidth is limited. However, this problem would be alleviated by incorporating a large on-chip storage which the processor can directly access.

To use on-chip storage effectively, two kinds of memory access: instruction access and data access, must be examined separately. Instruction access typically has higher locality than data access and hence an instruction cache is satisfactory for the purpose of efficient instruction access due to its high hit-ratio. The implementation of an instruction cache is relatively simple since it can be read-only. Hennessy suggested the use of a small on-chip instruction cache to lower the required off-chip instruction bandwidth [Henn84].

The problem in bandwidth consumption for data access is more serious because data access is less predictable than instruction access [Henn84]. Having the chip use a data cache is not the most efficient method: the unit of every read/write for the cache is a block, but for scalar data, only one word may be needed after a whole block is transferred into the cache after a miss. The other data in that block may be unused before that block is replaced by another one.

Registers as on-chip storage are usually a scarce resource, but in respect of processing scalar data, registers have the following advantages over the cache: (1) a register set can yield double the performance of a data cache in both speed and cost [DiMc82]; (2) registers can be specified explicitly in the program; cache memory does not have this flexibility. Compilers optimize for register allocation, while the management of cache is handled by hardware, hence is transparent to assembly-language programmers; (3) register allocation potentially can achieve lower bus traffic [GoHs86] and involves less overhead than using a cache; (4) it is possible to put a reasonably large register set on the chip because the area per stored bit in registers is smaller than in a cache [Henn84].

A large register set has been considered for fast data access since as early as 1978 [Russ78]. The VLSI technology has made it possible to implement a large register set on a processor chip. Ditzel and McLellan surveyed numerous designs that take advantage of a large register set [DiMc82]. Those designs include Sites' advocating using either a renaming mechanism or banks of registers for efficiently using 100 to 1000 registers [Site79], Dannenberg's proposal for using many registers for holding local variables in block structured languages but avoiding the problem of aliasing or compiler complexi-

ty [Dann79], BBN C/70's design which divides 1024 registers into multiple register sets and treats them as a circular buffer [Kral80], and finally, the multiple register windows on the Berkeley RISC which has a total of 138 registers [PaSe82]. An even earlier case is CRAY-1, which has 656 registers [Russ78]. Similar approaches are used in the designs of the C Machine [DiMc82] and the Adept [WaFl87] architectures.

Although there are different opinions that advocate using register allocation to keep the operands in a smaller register set [Henn82, Hsu87, Radi83, Wall88], a multiple-register-window approach which requires more techniques on the hardware still is attractive because of the ease of implementation of compilers and less register saving/restoring overhead associated with procedure calls [GoHs86, PaSe82, Patt85]. Moreover, if the hardware is able to handle overlapped register windows, the cost of passing parameters of procedure calls can be reduced. It can be expected that, with the multiple-register-window approach, two essential advantages can be achieved: (1) operands can be accessed at high speed; and (2) memory traffic can be minimized.

Both Tanenbaum [Tane78] and Patterson, et al. [PaSe82] point out that the dynamic percentage of the use of constants and scalar data in the average programs written in procedural languages is approximately 75 percent. The constants can be encoded directly in the instructions; the scalar data can be accommodated by registers; and the remaining 25 percent of data requires memory references. Hence, a load/store architecture (or register-oriented architecture) with a reduced instruction set is chosen for the target machine. This approach might increase the number of instructions, but in the tradeoff between instruction bandwidth and data bandwidth, a reduced data bandwidth is desirable.

Problem Statement

The objectives of this thesis are to extend the existing multiple-register-window architecture and to explore a new approach to supporting block-structured languages such as Pascal, Ada, and Modula-2. The development of the target machine architecture, which is subsequently referred to as HMA (Hypothetical Multiple-Register-Window Ar-

chitecture) in this thesis, involves the following subject matters:

(1) A major interest of this research is on the storage hierarchy that includes the on-chip register set, caches, and the main memory. A large multiwindowing register set is placed on the top of the storage hierarchy. However, there are several problems accompanying the register set, such as the window overflow/underflow related to the dynamic call/return, referencing registers with pointers that need memory addresses, the allocation of overabundant local scalar variables and non-scalar data that cannot be held by the register set, and the access to non-local variables. The Berkeley RISC's approach to resolving the problems of window overflow/underflow and pointers to registers is adopting a conceptual window stack which is a one-to-one mapping from registers to memory and vice versa; and it resolves the rest of the problems with a conventional stack for activation records. Both stacks imply a strict last-in-first-out (LIFO) nature of the architecture. In HMA's strategy the stacks are replaced by contour cells that are similar to the ones developed in the contour model. But the data structure of the contour cell is extended, and a method which orchestrates the register windows and contour cells is developed in Chapter 3.

(2) As the workhorse of HMA, the microarchitecture of a pipelined load/store processor is to be developed. Its hardware organization and instruction set are described in Chapter 4.

(3) The entire design of HMA presented in Chapter 3 and Chapter 4 contains a simulation with two test programs, The Towers of Hanoi and The Shortest Path, both written in assembly code then translated into HMA's machine code by an assembler written in the AWK programming language. The simulator is written in the C++ programming language. It generates traces and profile of the simulation as discussed in Chapter 5.

CHAPTER II

A REVIEW OF RELATED DESIGNS

This chapter reviews the previous designs of: (1) multiple-register-window architectures, (2) load/store architectures, and (3) the contour model, that constitute the foundation of this thesis.

Multiple-Register-Window Architectures

Patterson and Sequin point out that the procedure call is the most time-consuming operation in the programs written in procedural languages [PaSe82]. On the Berkeley RISC, multiple overlapped register windows are used to reduce the saving/restoring of registers upon each procedure call/return, and passing arguments/results to/from procedures is through the overlapped registers instead of the memory. Those operations are considered the major factors causing procedure calls to be slow.

The fundamental mechanisms of the overlapped register windows of the Berkeley RISC are: (1) allocate a new window of registers upon each procedure call. A Berkeley RISC processor has a large register set (138 general-purpose registers), divided into eight windows. A pointer called CWP (current window pointer) points to the youngest register window, which contains the parameters and local scalar variables of the most recently activated procedure. A procedure call advances the CWP forward and a new register window is prepared for it; and the CWP ''backs out'' to restore the old register window upon a procedure return; (2) the registers containing the outgoing arguments of the caller and the registers containing the incoming arguments of the called procedure overlap. Thus, the parent procedure copies the actual parameters to the child procedure's input-argument registers before the control is transferred to the child procedure.

The set of registers available to every procedure are shown in Figure 1† [PaSe82]. Six registers are available on each of the overlapped sections (HIGH and LOW); ten registers on the LOCAL section are available for local scalar variables; and ten registers on the GLOBAL section are available for the global scalar variables that are common to all procedures; i.e., every procedure shares the same set of global registers. The new set of registers that are allocated to each procedure activation are numbered from R10 to R31. High registers R26 to R31 contain incoming arguments passed from the parent procedure. Local registers R16 to R25 contain a procedure's local scalar variables. Low registers R10 to R15 contain temporaries and outgoing arguments passed to the child procedure. Registers R10 to R15 of the parent procedure become registers R26 to R31 of the child procedure. Thus, parameters are transferred by registers, without memory references. Figure 2 illustrates an instance of nested calls, where procedure A calls procedure B, which calls procedure C [PaSe82].

The multi-windowing register set of the Berkeley RISC is arranged as a circular buffer to facilitate the allocation of register windows, so that when the register windows are used up in the first cycle, another cycle is ready to start and reuse the register windows.

The Berkeley RISC maintains a window stack in the memory; it is referred to as a conceptual window stack (CWS) [Kate85]. When the nesting depth of procedure calls is so large as to use up all physical register windows, a window overflow occurs. The oldest register window(s) is (are) saved in the CWS upon a window overflow. Conversely, a register window (or a series of register windows) is (are) restored from the CWS upon a window underflow. The register saving/restoring caused by window overflow/underflow automatically are handled by the hardware. In addition to the handling of overflow/underflow problems, the conceptual window stack of the Berkeley RISC also provides the registers with addresses to solve for the problems of up-level ad-

† All figures are presented in Appendix D.

dressing and pointers to registers.

Hitchcock et al. [HiSp85] and Eickemeyer [Eick88] emphasize that Berkeley RISC's multiple register windows have substantial contribution to its high performance. They ran several trace-driven simulations on VAX 11/780, Motorola 68000, and RISC I, and found that the performance gain achieved by the multiple-register-window scheme is very significant even though the architectures vary.

The C Machine stack cache proposed by Ditzel and McLellan [DiMc82] also takes a multiple-window approach, but the windows are implemented on a stack cache instead of a register stack, and the window size for each procedure activation is variable. Nevertheless, there is strong similarity between the structure of the C Machine stack cache and the structure of the Berkeley RISC's conceptual window stack. Wakefield and Flynn implemented the Adept architecture at Stanford University as contour storage on multiple register sets [WaFl87]. In [Eick88], the register sets are organized as parallel stacks. Other variations of the fixed-size multiple-register-window architecture are a reduced, multi-size-register-windows, RISC architecture which was proposed by Huguet and Lang [HuLa85] and a two-size, overlapping-register-windows, RISC architecture which was proposed by Furht [Furht88].

Load/Store Architectures

The load/store architecture is also known as the register-oriented architecture. The philosophy of the instruction set design of the load/store architecture class emphasizes register-to-register operations with only load and store instructions accessing memory. Both load and store instructions need multiple CPU cycles to execute, while most of the other instructions operating on registers only need a single CPU cycle. Another significant advantage of the load/store architecture is its effectiveness in lowering the data bandwidth [Henn84]. CDC 6600 designed by Seymour Cray is conceived of as the earliest load/store architecture; this computer architect also originates the Cray supercomputers that belong to the same architecture class [Milu89].

Reduced-Instruction-Set Computers

The RISC architecture is a variant of the load/store architecture antecedent. The first RISC machine is the IBM 801, which was built in 1979 [Radi83]. David Patterson, Carlos Sequin, and their graduate students at University of California, Berkeley designed and implemented RISC I and RISC II VLSI microprocessors which formally used the acronym "RISC" [PaSe82]. About the same time, John Hennessy and his research group at Stanford University embarked on their project of MIPS, another streamlined VLSI microprocessor [Henn83]. And there have been many more RISC designs developed by both academic and commercial organizations. Generally speaking, the features of RISCs are: (1) They are load/store architectures and they take advantage of a large set of general-purpose registers [Milu89, Patt85]; (2) A reduced instruction set is used. Patterson pointed out that for the VAX-11, 20 percent of its instructions are responsible for 60 percent of the microcode and are only 0.2 percent of all instructions executed [Patt85]. Those complex instructions that lead to heavy microcode are primarily designed for emulating high-level-language statements, but as in the case of the VAX-11, they are not used frequently. Further, they lengthen the clock period, and thereby slow down the microprogram [Patt85, Radi83]. Hence, he proposed using a reduced instruction set which contains only those primitive instructions that are as simple as microinstructions and compiling programs down to microinstruction level [Patt85]; (3) Regular instruction format is used. The size of all RISC instructions is one word long. Opcode and register operands should always be in the same place for all instructions. This feature simplifies the instruction-decoding logic [Patt85]; (4) Simple addressing modes are used. With few addressing modes, it is easier to map instructions onto a pipeline, since the pipeline can be designed to avoid a number of computation related conflicts [Milu89]; (5) Instruction pipelining is used for all RISCs to simultaneously execute multiple instructions [Milu89, Patt85].

Both the IBM 801 [Radi83] and the Intel 80960 [MyBu88] microprocessor require that all operands be aligned on boundaries consistent with their size, and the Stanford

MIPS provides only word addressing [Henn84]. There are differences among various RISCs in their approaches to handling pipeline hazards. Both the IBM 801 and the Berkeley RISC use a hardware internal forwarding technique to avoid pipeline interlocks [Patt85, Radi83]. The Stanford MIPS uses a reorganizer, which is a software interface for its compiler, to prevent pipeline interlocks from occurring [Henn82].

Support to delayed branches is also important to RISCs. A delayed branch means the instruction following a branch instruction in the source code is always fetched and executed no matter whether the branch will be taken. The compiler can support delayed branches by either rearranging the instructions or inserting no-operation instructions following the branch instructions. The IBM 801, the Berkeley RISC, and the Stanford MIPS use this mechanism in their architectures and compilers. Hennessy reported that 21 percent of CPU cycles could be saved by using delayed branches [Henn84].

In addition to the advantages of reduced memory traffic, execution speed-up, and a highly regular hardware design, the RISC design also results in reducing control-unit area due to the reduced instruction set [PaSe82].

The Contour Model

The contour model, proposed by Johnston, is a vehicle to interpret the block-structured process [Johns71]. It views the morphological structure in the procedural language as nested contours; it presents this nesting property in a topographic map of contours and defines the cellular storage organization of various objects.

In the contour model, algorithm and record of execution are disjoint but related components of a process. Johnston made the following definitions: "A process is a sequence of snapshots, each containing the invariant algorithm and a stage of the record of execution. Both the algorithm and the record are basically nested sets of contours . . . The contour structure of the algorithm functions as a template for the formation of the contour structure of the record."

An algorithm contour contains the code and compile-time information of the symbols of the program block it represents. Algorithm contours spawn their record-of-execution

counterparts, which are referred to as record contours in Johnston's paper, during run-time. In his paper, Johnston demonstrates the execution of SAM, an Algol-60 program, with a series of snapshots of the algorithm and the record of execution. The allocation/deallocation of record contours follows the change of the locus of control (or the site of activity). The locus of control is realized by the virtual processor with two pointers in its cell — an environment pointer (ep) and an instruction pointer (ip). The access environment of a virtual processor comprises the record contour pointed to by *ep* and all the record contours enclosing that record contour. They are extension of the activation records linked by access links in the stack model [Aho86].

Figure 3 illustrates the generic format of the contour cell described by Johnston [Johns71]. The organization part of a contour cell consists of a contour valid bit (cvb) and three special subcells: static link, antecedent link, and height; the residence part contains the declaration array subcells. The static link of contour A whose height is i+1 must point to contour B which both has height i and immediately encloses contour A. The antecedent link of a record contour must point to an algorithm contour of the same height. The antecedent links of algorithm contours are left unspecified. The height of a contour indicates how deep it is nested. A contour which is not enclosed by any contours has a 0 height and has a null static link. The array subcells contain the parameters, variables, labels, and the return parameter that a procedure/block has access to. Every label and the return parameter consists of two pointers: an environment pointer (ep) and an instruction pointer (ip), together they can direct the virtual processor to a specific site of activity and to branch to a specific instruction.

Following the 1971 paper, Johnston developed the Contour Model Architecture (CMA) and the Contour Model Assembly Language (CMAL) [Johns80]. In his words, CMA is a relatively conventional, stack-oriented architecture whose tagged record structure and assembly language are intended to be implemented in microcode. In CMA, the cell structures are defined formally as combinations of mono-records and/or poly-records. The assembly language CMAL contains 180 instructions. The execution of most of the instructions includes updating an operand stack, from/onto which instruc-

tions retrieve/store informations in the form of mono-records or poly-records. The Burroughs B5700/B6700 computers use data structures that strongly resemble those of the contour model's. Organick's monograph [Orga73] contains documentation of the B5700/B6700 series.

Two more features provided by the contour model are proliferation of processors and cell retention. The proliferation of processors is for realization of the multiple-activity processes, which for example may be operating system processes, tasking, or coroutines. The principle of cell retention is: a storage cell C in the record of execution of a process must be retained; i.e., not be deallocated, if either C is an awake virtual processor or if one or more pointers still points to C which thereby remains accessible.

Instead of fully exploiting the versatility of the contour model, this thesis follows its concepts only to the extent of using contour cells.

CHAPTER III

THE RUN-TIME STORAGE OF HMA

The storage hierarchy associated with the scope of this thesis includes the on-chip multi-windowing register set, caches, and the virtual memory. This chapter discusses the management of HMA's run-time storage, especially the multi-windowing register set and the virtual memory.

The Multi-Windowing Register Set

Like the Berkeley RISC predecessor, multiple windows of registers are incorporated in the HMA design, and they are arranged as a circular buffer to let the programmer have an illusion that the number of register windows logically is unbounded.

Figure 4 illustrates the circular buffer organization which is a modification of the Berkeley RISC's [Kate85]. Two pointers, the saved-window pointer (SWP) and the current-window pointer (CWP), are used to keep track of the allocation of register windows. SWP points to the window which is most recently saved in the memory due to a window overflow (the cause and handling of window overflow are discussed later on this section). CWP points to the window of the most recently activated procedure. As shown in snapshot (a), eight register windows (w0 to w7) are physically available, and six of them (w0 to w5, that are marked in shade) are occupied. A register window, as delimited by solid lines, contains the incoming arguments (denoted as a procedure's name followed by .in) and the local scalar variables (denoted as a procedure's name followed by .loc) of a procedure. For the convenience in notation, the overlapped registers that contain the outgoing arguments of the parent procedure's are depicted as belonging to the window of the child procedure's in which they constitute the incoming arguments.

The windows grow clockwise in the circular buffer as the nesting depth of procedure calls increases. When the circular buffer is fully loaded; i.e., in case of window overflow, the oldest windows must be spilled into memory.

The following hypothetical case demonstrates this scenario. If procedure F in snapshot (a) calls procedure G, then it writes the parameters into its outgoing-argument registers (the incoming-argument registers of G) and executes a call instruction. The call instruction moves CWP forward by one window in the circular buffer. The snapshot of the circular buffer right after performing ''F calls G'' is illustrated in snapshot (b). If G calls another procedure, H, then an overflow trap is invoked immediately after the call instruction is executed; otherwise H may destroy the contents of A.in, the incoming-argument registers of A, in case that H further calls another procedure. The overflow trap transfers control to the window-overflow handler, an interrupt service routine, which then saves A.in and A.loc of window 0 in the memory and moves SWP forward by one window to the beginning point of B.in. The snapshot of the circular buffer after performing ''G calls H'' is illustrated in snapshot (c). From the observation on this example, an overflow of register windows occurs when a call instruction attempts to modify CWP and make it equal to SWP.

The underflow of register windows is handled in an analogous way. A return instruction moves CWP back by one window (counter-clockwise in the circular buffer in Figure 4). When a return causes CWP to coincide with SWP, an underflow happens and control is transferred to the window-underflow handler which restores the current register window from memory and moves SWP backward by one window.

Although the cost of handling window overflow/underflow is expensive, previous research has found that typically the fluctuation of nesting depth for programs written in C and Pascal are fairly small. In other words, programs seldom execute a long sequence of nested calls and followed by a long sequence of returns [DiMc82, Patt85]. Thus, it is not a large problem for the multiple-register-window scheme to deal with the rare occurrence of window overflows and underflows.

Organizing the Objects in Virtual Memory

Contour Cells

The on-chip storage has limitations. First, register windows must be spilled into memory in case of window overflow. Second, registers are not suitable for storing non-scalar data. Third, sometimes the register set is not large enough to hold all of the scalar variables in a program. And fourth, non-local data must be stored or retrieved into/from main memory rather than registers. The Berkeley RISC uses a conceptual window stack and a conventional activation-record stack in handling these problems. However, Johnston points out that there are two unfortunate connotations — the strict LIFO nature and the limitation for multitasking applications — that are associated with the stack model [Johns71]. Hence he proposed the contour model, in which a process' run-time environment consists of two components: the algorithm and the record of execution; both are data structures consisting of contour cells. In the following text and figures, the contour cells that dynamically are allocated to the record of execution are referred to as execution contours, and the contour cells of the lifetime-invariant algorithm are referred to as algorithm contours. Opposed to the stack model where the activation records are organized as contiguous frames on a last-in-first-out basis, rather, the contour cells can flexibly be managed with a series of threads.

Figure 5 illustrates the organization of an execution contour, in which both the control record and the data record are combined together. The control record further is divided into two subrecords. The register-spilling-information subrecord contains the base-relative word offsets of those receptacles which is tied to their register counterparts (r16 to r31 in a register window). Each register's offset is an 8-bit item in which the most significant bit is a valid offset bit (vob) where a one indicates that the following word offset is valid and otherwise a zero indicates the invalidity of that offset. The thread subrecord contains four pointers: a-link (the antecedent link) points to an algorithm contour which is the execution-contour's template in the algorithm; s-link (the static link) points to the execution contour which immediately encloses the execution contour to

which the static link belongs; z.ip functions as a saved program counter; z.ep points to the execution contour which is both innermost to the instruction which z.ip points to and about to be the locus of control (or site of activity) when the block exit happens. The data record contains the input arguments, the local scalar variables, the base addresses of local arrays that are allocated in the heap, and other local data structures.

The Window Activation Vector and the Register Spilling Algorithm

When a procedure is activated, a register window as well as an execution contour are prepared for its use. The allocation of register windows was discussed on last section. The allocation of an execution contour is explicitly defined by a memory-allocation instruction which is interpreted into a system routine at run-time. After the allocation of an execution contour, a base register specified in the memory-allocation instruction contains the base address of that execution contour, which is also automatically kept in the window activation vector (WAV). As illustrated in Figure 6, the WAV is an array of registers maintaining the threads of existing execution contours that are represented by C_0, C_1, and C_j in the figure.

Unlike the circular-buffer organization of the register windows, the WAV registers are straightened and the number of elements in this vector is multiple times the number of register windows that are physically available. In this case, 32 WAV registers are available and they are capable of handling 32 nested procedure calls. Two indexes to the WAV registers, SWP_{psw} and CWP_{psw}, are the 5-bit saved-window pointer and the 5-bit current-window pointer in the processor status word. Actually, the SWP and CWP used by the register windows are the three least significant bits of SWP_{psw} and CWP_{psw}, respectively. SWP_{psw} and CWP_{psw} are advanced (i.e., moved right-hand-bound in Figure 6) or backed up (i.e., moved left-hand-bound in Figure 6) when they need to be relocated upon the allocation/deallocation of register windows discussed on last section†. SWP_{psw} delim-

† Note that SWP_{psw} and CWP_{psw} contain the carry-overs as the SWP and CWP discussed on last section are incremented.

its the stream of the elements (WAV_0 to WAV_j in Figure 6) that stand for the windows spilled into memory. The elements following WAV_i delimited by SWP_{psw} until WAV_j delimited by CWP_{psw} stand for the windows that are still using the on-chip storage. The window activation vector is an important aid to the saving/restoring of register windows upon window overflow/underflow. The algorithms of saving and restoring register windows are written in C-like pseudo-code and are shown as follows.

```
/*****************************************************************
 * Saving a register window in the main memory in case of
 * window overflow
 *****************************************************************/

 if  (CWP_psw − SWP_psw ≡ 8)  /*overflow* /
 {
   /*  Copy the pointer to the base of the target execution
    *  contour where the register window is spilled.
    */
      BasePtr = WAV [++SWP_psw ];

   /*  Consult the register spilling information and store
    *  the object GPR (general-purpose register) into the
    *  appropriate receptacle in the target execution contour
    *  if and only if the valid offset bit (vob) is set.
    */
        for (reg_no = FirstLocalReg; reg_no <= LastLocalReg; reg_no++)
           if (BasePtr->RegSpillInfo[reg_no].vob == 1)
           /* save the register in memory */
              BasePtr->DataCell[BasePtr->RegSpillInfo[reg_no].offset] \
              = GPR[current_window][reg_no];
           else  /* doing nothing */;
 }


/*****************************************************************
 * Restoring a register window from the main memory in case of
 * window underflow.
 *****************************************************************/

 if  (CWP_psw ≡ SWP_psw )  /*underflow* /
 {
   /*  Copy the pointer to the base of the object execution
    *  contour from which the register window is restored.
    */
      BasePtr = WAV [SWP_psw —];
```

```
  /* Consult the register spilling information and load
   * the target GPR (general-purpose register) with the
   * appropriate receptacle in the object execution contour
   * if and only if the valid offset bit (vob) is set.
   */
       for (reg_no = FirstLocalReg; reg_no <= LastLocalReg; reg_no++)
          if (BasePtr->RegSpillInfo[reg_no].vob == 1)
          /* restore the register from memory */
             GPR[current_window][reg_no] = \
             BasePtr-> DataCell[BasePtr->RegSpillInfo[reg_no].offset];
          else  /* doing nothing */;
}
```

Pointers to Registers and Non-Local Variables

The window activation vector also facilitates handling pointers to registers. The customized address for a register contains a [tag, window#, register#] triple. A tag of "11" on the two most significant bits of the virtual address denotes a register address; if $SWP_{psw} \leq$ window# $\leq CWP_{psw}$ then the reference automatically is directed to a register window; register# is the offset of the target register within the register window. If window# $< SWP_{psw}$ then the reference goes to an execution contour.

To reference a non-local variable, a [levels-back, id#] pair must be known at compile-time. *levels-back* is the difference in nesting depth between the procedure/block which references the non-local variable and the procedure/block which declares it as a local variable. *id#* is the offset of the non-local within its execution contour. A display of access environment according to the locus of control is updated during run-time by tracing the static links. Thus, the receptacle of a non-local variable can be pinpointed by using *levels-back* to fetch a thread of execution contour from the display vector and using *id#* to find out the non-local's offset within that execution contour.

An Instance of Block-Structured Program

A block-structured program consists of nested algorithm contours that spawn execution contours at run-time. An algorithm contour contains the code and the definition of the variables of a procedure/block. Algorithm contours remain invariant during execu-

tion, while the contents of execution contours may vary during execution. Also unlike the execution contours, there is only one algorithm contour for each procedure or block. A static height is associated with every algorithm contour according to the nesting depth of the procedure/block in the program. For example, Figure 7 shows a Pascal program in which the height of the algorithm contour of MAIN is 0; both the height of the algorithm contour of BB and the height of the algorithm contour of DD are 1; the height of the algorithm contour of CC is 2. The execution contours spawned by a specific algorithm contour have the same static height.

Both Figure 8 and Figure 9 illustrate a process snapshot which may occur during the execution of the program in Figure 7. In this example, the actions that have taken place so far are the following: MAIN calls procedure BB; BB makes a recursive call; and procedure CC is called during the execution of this recursive call. Figure 8 shows the topographic map of the nested execution contours in this snapshot. Figure 9 is a more accurate portrait of the cellular structures of the run-time objects in the same snapshot. However, for those execution contours, only the control-organization part is shown in this figure, and the configuration of the algorithm is omitted since it is identical to the one shown in Figure 7. To differentiate execution contours from algorithm contours, an apostrophe (or a couple of apostrophes) is put on the upper-right corner of every execution contour's name. The antecedent link of each execution contour respectively points to its template, an algorithm contour. The static link, z.ip, and z.ep of MAIN' are null pointers. The static link of BB', the execution contour which was formed right after the first call on procedure BB, is a pointer to MAIN'; z.ip of BB' points to the next-to-execute instruction in the algorithm of MAIN after the exit of BB'; z.ep of BB' points to MAIN' because the locus of control will be in MAIN' after the exit of BB'. The static link of BB", the execution contour which was formed right after the second call (a recursive call) on procedure BB, is a pointer to MAIN'; z.ip of BB" points to the next-to-execute instruction in the algorithm of BB after the exit of BB"; z.ep of BB' points to BB' because the locus of control will be in BB' after the exit of BB". The static link of CC" points to BB" because the locus of control was in BB" when procedure CC was

called; z.ip of CC" points to the next-to-execute instruction in the algorithm of BB after the exit of CC"; z.ep of CC" points to BB" because the locus of control will be in BB" after the exit of CC". The window activation vector — WAV_0, WAV_1, WAV_2 and WAV_3 — indicates that the order of the activations of procedures is: MAIN', BB', BB", CC". However, since the present locus of control is in CC" (as shown in Figure 8 with a "@" in it), therefore the elements in the display — D_0, D_1 and D_2 — point to MAIN', BB", and CC", respectively.

CHAPTER IV

THE PROCESSOR ARCHITECTURE OF HMA

A 32-bit microprocessor is intended to be the engine of the present HMA design. This chapter discusses the microarchitecture and the instruction set of the target processor design.

The Microarchitecture

Figure 10 illustrates the layout of the hardware units and the data path on the HMA processor. The hardware organization of the HMA processor is described as follows.

Instruction-Fetch Unit

The instruction-fetch unit prefetches instructions from the instruction cache and dispatches an instruction to the instruction decode unit during every clock period. It has three instruction pointers: the Fetch Instruction Pointer (FIP), the Current Instruction Pointer (CIP), and the Execute Instruction Pointer (XIP). FIP specifies the address of the instruction which is latched for the Current Instruction Register (CIR). CIP specifies the address of the instruction which is in CIR and is being decoded by the instruction decode unit. XIP specifies the address of the instruction which is dispatched to the integer execution unit. This arrangement, like the Intel 80960's [MyBu88], takes precaution of saving the state of the processor and allows the machine to recover from exception handling.

Instruction-Decode Unit

The instruction-decode unit is responsible for decoding the instruction delivered by the instruction fetch unit, looking up the jump table for microinstruction sequencing, and sending the identifications of the referenced registers to the integer-execution unit, which

thereafter fetches the register operands. If the instruction decode unit detects a memory-referencing instruction, then it prepares the integer-execution unit to compute the effective address of the memory operand.

Control Unit

The control unit contains a microprogram sequencer and a microprogrammed control-memory. It generates the control signals that activate or deactivate the data paths of the processor to execute the instruction decoded by the instruction-decode unit.

Integer-Execution Unit

The integer-execution unit contains the most important resources of this microengine. The ALU-Shifter-Merger (ASM) sub-unit carries out the micro-operations of the computation instructions. A register file comprising 138 general-purpose registers (that includes ten global registers and eight register windows with sixteen registers belonging to each window) has four ports — two read ports and two write ports. The ASM sub-unit fetches source operands (src1, src2 and/or an immediate operand) through the read ports (for src1 and src2), and the result of a computation is sent to the result bus and written into the destination register through a write port. Another write port is used to load a register with the data on the external data-bus. The external data bus transfers data from/to the data cache for load/store instructions, that take more than one clock cycle.

Load/Store Unit

The load/store unit has two buffers — an input FIFO (first-in-first-out) buffer and an output FIFO buffer — to resolve the bus conflict with consecutive loads and stores. If a load or a store which is using a bus cycle is followed by a load instruction, then the latter is suspended and the effective address is held in the input FIFO until a new bus cycle is available; on the other hand, if a load or a store which is using a bus cycle is followed by a store instruction, then the latter is suspended and both the effective address and data are held in the output FIFO until a new bus cycle is available.

Instruction Pipelining

The instruction pipeline has four pipestages: instruction fetch (IF), instruction decode (ID), instruction execution (EX), and operand write-back (WB). For an instruction which performs only register-to-register operations, it is latched for the input of CIR during pipestage IF; it is decoded by the instruction decode unit and the source operands are fetched during pipestage ID; it triggers an ALU operation during pipestage EX; and the result operand is written into the destination register during pipestage WB. Normally, each pipestage takes one clock cycle to finish. As shown in Figure 11, a maximum of four instructions can concurrently be in the pipeline. However, the execution pipestage of a load/store instruction takes at least two clock cycles — one for computing the effective address and one for transferring data from/to cache. In this case, a scenario without stalling the pipeline is used: the instructions following a load or store continue being executed while the load/store instruction is accessing memory until the register which is being loaded is needed for a source operand. A register-scoreboarding hardware must be incorporated in this approach. A register is marked as invalid in the scoreboard during the execution of a load, and when the load is completed, the invalid mark is removed. If an instruction references a register which is marked as invalid then the pipeline is blocked until the register is available; otherwise the pipeline continues as usual. Sometimes the pipeline is blocked because of data dependencies. Figure 12 illustrates an example of read-after-write data dependency. The "add" instruction modifies r3 which is a source register of the "and" instruction. r3 is scoreboarded during the instruction-decode pipestage of "add"; hence "and" cannot fetch r3 until "add" writes the result into r3 and removes r3 from the register scoreboard. The consequence is the "bubble" pipestages; it is also referred to as the pipeline interlock. Usually, an optimizing compiler would detect and avoid data dependencies during code generation.

Another kind of pipeline interlock is caused by branches. Since the target of a branch is not known until the execution pipestage of the branch instruction, the fetching of the instructions following the branch must suspend. In this case, an optimizing compiler

moves an useful instruction to the place immediately after the branch instruction, and the instruction pipeline fetches, decodes, and executes this instruction as usual while it is processing the branch instruction. An example is shown in Figure 13. In code sequence (b), the subtract instruction is executed before the control is branched to the instruction at L1. The execution of the rearranged code sequence must yield the same result as the original code sequence. This approach is called a delayed branch. The delayed-branch approach effectively utilizes one instruction slot on each instruction branch; otherwise the CPU will sit idle.

The Instruction Set

The assembly-instruction set of HMA is listed in Appendix A. It has forty instructions of four instruction types: computation instructions, data-transfer instructions, control-flow instructions, and extended instructions. Each instruction type may have several groups of instructions so that all the instructions of each group have a uniform instruction format. Essentially, these instructions are selected from the instruction set of the Hewlett-Packard Precision Architecture [HP86], but a few instructions; i.e., the unconditional-branch instructions, are redefined. Further, four new instructions are extrapolated for HMA and given an identical instruction format under the category of extended instructions.

Only data-transfer instructions — i.e., loads and stores — can access memory. The addressing modes used by this group include: (1) base-plus-displacement mode, in which an operand's virtual address is the sum of the following items: the value in a base register, a 14-bit signed displacement, and the value in a space register which identifies a module of the virtual memory, and (2) indexed mode, in which an operand's virtual address is the sum of the following items: the value in a base register, the value in an index register, and the value in a space register which identifies a module of the virtual memory. Load Byte and Store Byte instructions are used for memory-mapped I/O and they reference absolute addresses. Load Immediate Left and Load Offset actually are not memory reference instructions. They are included for loading 32-bit constants.

Branches can be conditional or unconditional ones. The target address of a branch can be specified by using the following modes: (1) PC-relative with static displacement, in which the target address is the location of the current instruction plus a 17-bit signed word displacement which is encoded in the instruction, (2) PC-relative with dynamic displacement, in which the target address is the location of the current instruction plus a shifted value of an index register, (3) base-relative with static displacement, in which the target address is the value in a base register plus a 17-bit signed word displacement which is encoded in the instruction, and (4) base-relative with dynamic displacement, in which the target address is the value in a base register plus a shifted value of an index register. Unconditional-branch instructions unanimously are dedicated to the procedure call/return. Branch and Link (''bl'') is used for intrasegment procedure-calls. The branch address is the result of adding a 17-bit word-displacement, which is formed by concatenating x-, y-, and z-field in the instruction, to the current instruction address. Branch and Link External (''ble'') is used for intersegment procedure-calls. The branch address is the result of adding a 12-bit word-displacement, which is formed by concatenating y- and z-field in the instruction, to the value in code-segment register x (x also is encoded in the instruction), which identifies the code segment that contains the procedure callee. For both ''bl'' and ''ble'' instructions, the sequential fetch-instruction pointer is saved in register t before the address of the target of the branch is formed. Branch Vectored (''bv'') is used for both intrasegment and intersegment procedure-returns. The branch address is the sum of the values in register b and register x.

The Extended Instructions category contains four instructions developed for HMA. Customize Register Address (''cra'') assigns an address to any general-purpose register which may belong in a procedure's register window. The new address is placed in register t, which thereafter is used to pinpoint the target register with a triple [''11'', window#, reg#] in its content. The binaries ''11'' are the MSB's of the new address which identifies that the referenced item is a register; window# is equal to CWPpsw; and *reg#* is encoded in im5-field in the instruction. Allocate Memory (''alloc'') allocates a contiguous block in the memory module of execution contours (if s = 0) or in the heap (if s

= 1). *len* (whose complement is encoded in clen-field) is the length (in bytes) of this block. The base address of this block is placed in register b. No Operation (''nop'') is used for delayed branches. And the ''ret'' instruction terminates the user's program.

CHAPTER V

THE SIMULATION OF HMA

The simulation was developed on the Perkin-Elmer 3230 computer system. The flowchart of the simulation process is shown in Figure 14. In the first phase, a test program prepared in HMA's assembly language is converted into HMA's machine code by an assembler; in the second phase, a software simulator accepts the object file and the data requested by the test program as input and generates an intermixed trace/statistics file in addition to the normal output of the test program; in the third phase, a simple profiler sorts out the trace/statistics file and breaks it down into separate files; i.e., an instruction-address trace-file, a data-address trace-file, a data-size statistics-file, a window-depth statistics-file, and an execution profile.

The Test Programs

Two test programs, The Towers of Hanoi and The Shortest Path, are listed in Appendix B. The assembly-language format is rather intuitive. In the beginning of each code segment (or algorithm contour), the register-spilling information and control threads of the to-be-allocated execution contour are properly set up for the housekeeping work. The assembly programmer does not need to worry about which register window it is using, but the consistency of passing arguments to/from each caller/callee must be handled carefully. The code is organized in such a way that the elimination of pipeline interlocks caused by data dependencies and the utilization of delayed-branch slots are taken care of, but however, one-hundred-percent optimization is not achieved.

The Assembler

A simple two-pass assembler was written in the AWK programming language [Aho88]. The first pass of the assembler eliminates comments, labels, leading spaces, and blank lines from an assembly program and assigns memory locations to the segment identifiers and labels; thus filters the source assembly program into an intermediate file which contains only segment identifiers and assembly instructions. The second pass directly maps the assembly instructions into machine code and annotates the segment addresses and segment lengths in the object file. The filename extensions used for the assembly-program source file, the intermediate file, and the object file are ".asm", ".tmp", and ".obj", respectively. The static sizes of the two test programs' object code are listed on the table shown below.

TABLE I

THE MACHINE-CODE SIZE
OF THE TEST PROGRAMS

program	machine-code size
Towers of Hanoi	792 bytes
Shortest Path	1372 bytes

The Simulator

The software simulator of HMA was developed in the C++ programming language [Stro87]. The major components of this program are described as follows.

The Abstract Data Types of the Virtual Memory

The objects in the virtual memory consist of three abstract data types. They are implemented with classes in C++:

```
/*
   Class program is an abstract data type that
   handles the operations on the code module.
*/
class program {
 instruction cell[CodeModuleSize];
public:
 void read(unsigned addr, instruction& buffer);
 void write(unsigned addr, instruction inst);
 void dump();
};

/*
   Class record is an abstract data type that
   handles the run-time activation record.
*/
class record {
 CELL* head;     // pointer to the header cell
 unsigned base;  // base address
 unsigned length; // record length
public:
 record(unsigned size); // constructor
 ~record();   // destructor
 void read(unsigned offset, void* bufaddr);
 void write(unsigned offset, CELL item);
};

/*
   Class heapobj is an abstract data type that
   handles the vectors in the heap.
*/
class heapobj {
 VecElem* head;   // pointer to the header vector element
 unsigned base;     // base address
 unsigned length;   // vector length
public:
 heapobj(unsigned size); // constructor
 ~heapobj();   // destructor
 void read(unsigned offset, VecElem& buffer);
 void write(unsigned offset, VecElem item);
};
```

Methods program::write() and program::read() are accessed only by the loader and the instruction-fetch module, respectively. And program::dump() is included for debugging. The organizations of classes record and heapobj are akin to each other. In both's initialization stages, linked lists are constructed as the data structures for each class' read() and write() methods to access. Both classes' functionalities include interpreting load and store instructions (with methods read() and write(), respectively), albeit there is a subtle point of method record::read(), whose argument *bufaddr* use a void-type pointer, because in addition to ordinary integers that are accepted by the CELL element, it also processes the register-spilling information upon window overflow/underflow. The register-spilling informations of four registers are grouped into a word which has the following format:

```
struct RSI_WORD {
  unsigned  vob0: 1;    // valid-offset bit of register 4*k
  unsigned   ofs0: 7;   // offset of register 4*k
  unsigned  vob1: 1;    // valid-offset bit of register 4*k+1
  unsigned   ofs1: 7;   // offset of register 4*k+1
  unsigned  vob2: 1;    // valid-offset bit of register 4*k+2
  unsigned   ofs2: 7;   // offset of register 4*k+2
  unsigned  vob3: 1;    // valid-offset bit of register 4*k+3
  unsigned   ofs3: 7;   // offset of register 4*k+3
};
```

Miscellaneous Objects in the Simulation

There are a variety of run-time objects defined as global variables in the simulator program. They include a global-register array, a local-register array, an array of flags on the register scoreboard, the special-purpose registers like the processor-status word and code-segment registers, a code module which is an instance of the *program* class, execution contours of the *record* class, heap vectors of the *heapobj* class, and an array representing the window-activation vector. The virtual processor, a major object of the simulation, is left for an individual discussion later.

Loader

The loader() function installs a test program's machine code in the code module in ac-

cordance with the segment addresses and segment lengths annotated in the object file. Code-segment registers are properly set up during the loading procedure.

Virtual Processor

The virtual processor is an abstract data type which defines the operations on the instruction pipeline and packages the information related to those operations. The definition of a virtual processor is as follows.

```
class pipeline {
 unsigned  fip,  // fetch-instruction pointer
           cip,  // current-instruction pointer
           xip,  // execute-instruction pointer
           gip;  // graduate-instruction pointer

 INST   cir,  // instruction buffer between fetch and decode stages.
        id_latch,  // instruction buffer between decode and execute stages.
        exc_latch; // instruction buffer between execute and write-back stages.

 short  dd,          // data-dependency semaphore
        blocked,     // pipeline blocked
        close_pipe;  // pipeline terminates

 unsigned t_resume;  // the time for resuming the pipeline.
public:
 pipeline();  // constructor
 void fch_inst(program&);  // fetch instruction
 void dec_inst();  // decode instruction
 void exc_inst();  // execute instruction
 void writeback();  // write back result
 void xtrace();  // instruction trace
 int  chk_status();  // check pipeline status
};
pipeline vp; // virtual processor
```

Figure 15 illustrates the flowchart of the instruction pipelining. The flowcharts of the four pipestage-methods — namely, pipeline::fch_inst(), pipeline::dec_inst(), pipeline::exc_inst(), and pipeline::writeback() — are illustrated on Figure 16 through Figure 19. Several architectural parameters — e.g., the number of register windows, the number of global registers, the number of local registers in a register window, and the maximum nesting depth of procedure calls, etc. — are defined in the header files and can

be changed for different platforms.

The Window-Overflow Handler and the Window-Underflow Handler

Two functions, window_overflow() and window_underflow(), respectively emulate the window-overflow handler and window-underflow handler that implements the register-spilling algorithm discussed in Chapter 3. Saving and restoring register windows are carried out by calling the execution contour's member functions write() and read(), respectively.

The Execution Monitor

During the simulated execution of a test program, various performance-related metric values are monitored to reflect the run-time statistics, or the execution profile. They include the execution time in clock cycles, dynamic instruction count, dynamic branch count, saved CPU cycles due to the delayed-branch approach, the dynamic count of no-operation instruction, the utilization of the buses, the dynamic count of loads and stores, the number of window overflow/underflow, and the number of register saved/restored. Also presented on the profile are the register-usage table, which reports the accumulated reference counts of the general-purpose registers, and the instruction histogram, which reports the times and percentage each instruction is executed among the instruction set.

Simulation Results

The results of a simulation-run are generated through the following sequential processes. First, run the assembler by issuing:

awk -f asm *source-assembly-program intermediate-file*

Second, run the simulator by issuing:

sim *-option object-file trace/stats-file*

At last, run the profiler by issuing:

awk -f pfl *trace/stats-file* [> *filename*]

The options used by the simulator are:

i — enables the instruction trace;

d — enables the data trace;

id — enables both instruction trace and data trace;

others — neither instruction trace nor data trace is enabled.

If enabled, the instruction address and/or the data address are/is collected every clock cycle in the trace files. The simulator samples the data size allocated to the test program and the nested depth of procedure calls; i.e., the number of active windows, every 50 clock cycles; they are collected in different files by the profiler and can be produced as line charts by the commercial spreadsheet program. The standard output of the profiler, the execution profile, can be redirected to a file. Appendix C shows two sample results — one is generated by running The Shortest Path with eight nodes, and another is generated by running The Towers of Hanoi with 15 disks. Figure 20 through Figure 23 show the fluctuations of the depth of the window stack and the data size of the two simulation-runs.

CHAPTER VI

SUMMARY, CONCLUSIONS, AND FUTURE WORK

This thesis presents the design and simulation of HMA, which contains a pipelined, load/store and multiple-register-window processor architecture and a method of managing its run-time storage. The contour model, rather than the conventional stack model, was adapted to handle the dynamic data structures at the run-time. The target processes for this computer architecture are the programs written in block-structured languages such as Pascal, Ada, or Modula-2.

A simulation was conducted to investigate this computer architecture. Two test programs, The Shortest Path and The Towers of Hanoi, were used in this simulation. They were translated into assembly programs from their Pascal-language counterparts by hand-coding in the assembly-instruction set developed in this thesis. The assembler was written in the AWK programming language to translate the assembly programs into HMA's machine code. The simulator was developed in the C++ programming language, which generates both instruction and data traces as well as the statistics of the execution of the test programs. From the two tentative simulation-runs, the HMA on the average executes an instruction for 1.35 clock cycles; the average speed-up of delayed branches is 19.78 percent; the fraction of memory-referencing instructions is 9.15 percent for the Towers of Hanoi and 13.48 percent for the Shortest Path — the major factor of the memory traffic is the housekeeping work (namely, storing register-spilling information into execution contours and saving/restoring registers for occasional window overflow/underflow) for the former and the frequent access of array elements for the latter. There are many places in the test programs that can be optimized further.

In the preparation of test programs, considerable time was consumed in hand-coding,

optimization, and debugging. For the future work, a compiler must be built first so that a substantial set of benchmarks may be selected for simulation. Further, the traces generated by the simulator are useful data for the ultimate design of the caches.

BIBLIOGRAPHY

[Aho86] Aho, A. V., Sethi, R., and Ullman, J. D. Compilers: Principles, Techniques, and Tools, Second Edition, Addison-Wesley, 1986.

[Aho88] Aho, A. V., Kernighan, B. K., and Weinberger, P. J. The AWK Programming Language, Addison-Wesley, 1988.

[Dann79] Dannenberg, R. B. "An Architecture with Many Operand Registers to Efficienyly Execute Block-Structured Languages," Proceedings of the 6th Annual Symposium on Computer Architecture, April 1979, pp. 50-57.

[DiMc82] Ditzel, D. R. and McLellan, H. R. "Register Allocation for Free: The C Machine Stack Cache," Proceedings of the First Symposium on Architectural Support for Programming Languages and Operating Systems, March 1982, pp. 48-54.

[Eick88] Eickemeyer, R. J. "Performance Evaluation of Multiple Register Set Architectures and Cache Memories," Ph. D. dissertation, U. of Illinois at Urbana-Champaign, 1988.

[Furht88] Furht, B. "A RISC Architecture with Two-Size, Overlapping Register Windows," IEEE Micro, Vol. 8, No. 2, April 1988, pp.67-80.

[GoHs86] Goodman, J. R. and Hsu, W. "On the Use of Registers vs. Cache to Minimize Memory Traffic," Computer Architecture News, Vol. 14, No. 2, June 1986, pp. 375-383.

[HaLi91] Harper III, D. T. and Linebarger, D. A. "Conflict-Free Vector Access Using a Dynamic Storage Scheme," IEEE Transactions on Computers, Vol. 40, No. 3, March 1991, pp. 276-283.

[Henn82] Hennessy, J., Jouppi, N., Baskett, F., Gross, T., and Gill, J. "Hardware/Software Tradeoffs for Increased Performance," Proceedings of the First Symposium on Architectural Support for Programming Languages and Operating Systems, March 1982, pp. 2-11.

[Henn83] Hennessy, J. L., Jouppi, N. P., Przybylski, S., Rowen, C., and Gross, T. "Design of a High Performance VLSI Processor," Proceedings of the 3rd Caltech Conference on VLSI, March 1983, pp. 33-54.

[Henn84] Hennessy, J. L. "VLSI Processor Architecture," IEEE Transaction on Computers, Vol. C-33, No. 12, Dec. 1984, pp. 1221-1246.

[HiSp85] Hitchcock III, C. Y. and Sprunt, H. M. Brinkley. "Analyzing Multiple Register Sets," Proceedings of the 12th International Symposium on Computer Architecture, Boston, MA, June 1985, pp. 55-63.

[HP86] HP 3000/930 and HP 9000/840 Computers, Precision Architecture and Instruction Reference Manual, Hewlett-Packard Company, 1986.

[Hsu87] Hsu, Wei-Chung. "Register Allocation and Code Scheduling for Load/Store Architectures," Ph. D. dissertation, U. of Wisconsin at Madison, 1987.

[HuLa85] Huguet, M. and Lang, T. "A Reduced Register File for RISC Architectures," Computer Architecture News, Vol. 13, No. 4, Sept. 1985, pp. 22-31.

[HwBr84] Hwang, K. and Briggs, F. A. Computer Architecture and Parallel Processing, McGraw-Hill, 1984.

[Johns71] Johnston, J. B. "The Contour Model of Block Structured Processes," SIGPLAN Notices, Vol. 6, No.2, Feb. 1971, pp. 55-82.

[Johns80] Johnston, J. B. "The Contour Model Architecture and Assembly Language," unpublished monograph, revised in August 1980.

[Kate85] Katevenis, M. G. H. Reduced Instruction Set Computer Architectures for VLSI, MIT Press, Cambridge, MA, 1985.

[Kral80] Kraley, M., Rettberg, R., Herman, P., Bressler, R., and Lake, A. "Design of a User-Microprogrammable Building Block," Proceedings of the 13th Annual Microprogramming Workshop, Dec. 1980, pp. 106-114.

[Milu89] Milutinovic, V. M. and Gimarc, C. E. "RISC Principles, Architecture, and Design," High-Level Language Computer Architecture, edited by V. M. Milutinovic, Computer Science Press, 1989, pp. 1-64.

[Orga73] Organick, E. I. Computer System Organization: The B5700/B6700 Series, Academic Press, 1973.

[PaSe82] Patterson, D. A. and Sequin, C. H. "A VLSI RISC," IEEE Computer, Vol. 15, No. 9, Sept. 1982, pp. 8-21.

[Patt85] Patterson, D. A. "Reduced Instruction Set Computers," Communications of the ACM, Vol. 28, No. 1, Jan. 1985, pp.8-21.

[Radi83] Radin, G. "The 801 Minicomputer," IBM Journal of Research and Development, Vol. 27, No. 3, May 1983, pp. 237-246.

[Russ78] Russell, R. M. "The CRAY-1 Computer Systems," Communications of the ACM, Vol. 21, No. 1, pp. 63-72, Jan. 1978.

[Site79] Sites, R. "How to Use 1000 Registers," Proceedings of 1st Caltech Conference

on VLSI, Calif. Inst. Tech., Pasadena, California, Jan. 1979, pp. 527-532.

[SoFr91] Sohi, G. and Franklin, M. "High-Bandwidth Data Memory Systems for Superscalar Processors," Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, April 8-11, 1991, pp. 53-62.

[Stro87] Stroustrup, B. The C++ Programming Language, Addison-Wesley, 1987.

[Tane78] Tanenbaum, A. S. "Implications of Structured Programming for Machine Architecture," Communications of the ACM, Vol. 21, No. 3, March 1978, pp. 237-246.

[WaFl87] Wakefield, S. P. and Flynn, M. J. "Reducing Execution Parameters Through Correspondence in Computer Architecture," IBM Journal of Research and Development, Vol. 31, No. 4, July 1987, pp.420-434.

[Wall88] Wall, D. W. "Register Windows vs. Register Allocation," SIGPLAN Notices, Vol. 23, No. 7, July 1988, pp. 67-78.

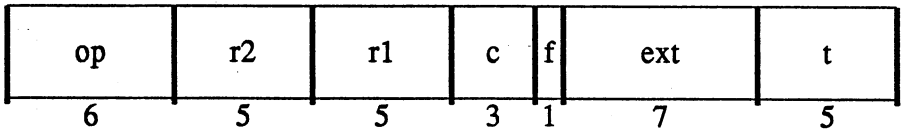
APPENDIXES

APPENDIX A

THE ASSEMBLY-INSTRUCTION SET AND MACHINE-CODE FORMATS OF HMA

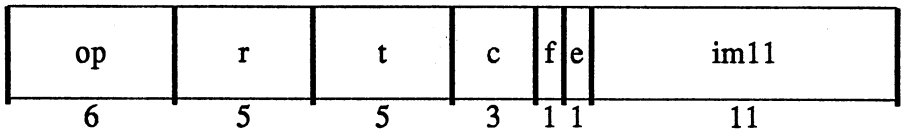
I. Computation Instructions

1. Arithmetic/Logical:



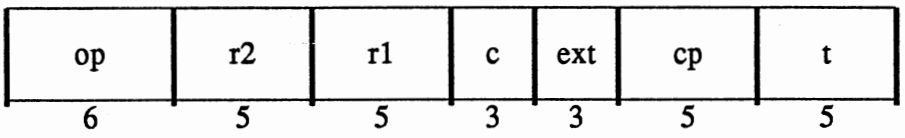
assembly instruction	annotation
add,cond r1, r2, t	Add, op=02 hex, ext=30 hex
addc,cond r1, r2, t	Add with Carry, op=02 hex, ext=38 hex
and,cond r1, r2, t	And, op=02 hex, ext=10 hex
or,cond r1, r2, t	Inclusive Or, op=02 hex, ext=12 hex
sub,cond r1, r2, t	Subtract, op=02 hex, ext=20 hex
xor,cond r1, r2, t	Exclusive Or, op=02 hex, ext=14 hex
sh2add,cond r1, r2, t	Shift Two and Add, op=02 hex, ext=34 hex
sh3add,cond r1, r2, t	Shift Three and Add, op=02 hex, ext=36 hex

2. Arithmetic Immediate:



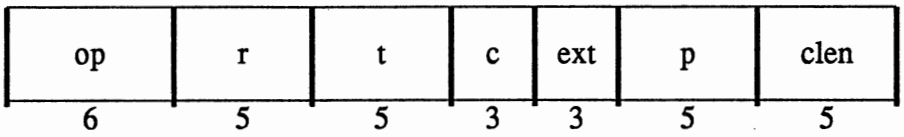
assembly instruction	annotation
addi,cond i, r, t	Add to Immediate, op=2D hex, e=0

3. Shift:



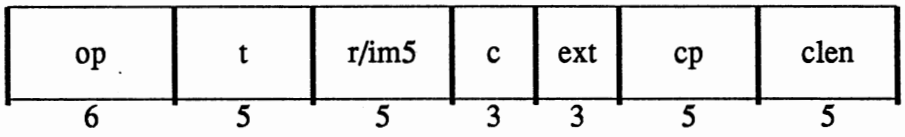
assembly instruction	annotation
shd,cond r1, r2, p, t	Shift Double, op=34 hex, ext=2

4. Extract:



assembly instruction	annotation
extru,cond r, p, len, t	Extract Unsigned, op=34 hex, ext=6
extrs,cond r, p, len, t	Extract Signed, op=34 hex, ext=7

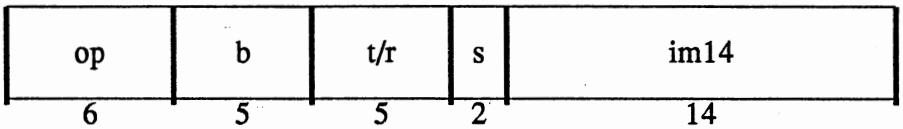
5. Deposit:



assembly instruction	annotation
dep,cond r, p, len, t	Deposit, op=35 hex, ext=3
depi,cond i, p, len, t	Deposit Immediate, op=35 hex, ext=7

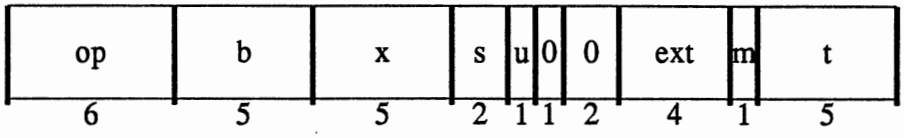
II. Data-Transfer Instructions

1. Base-plus-Displacement Mode:



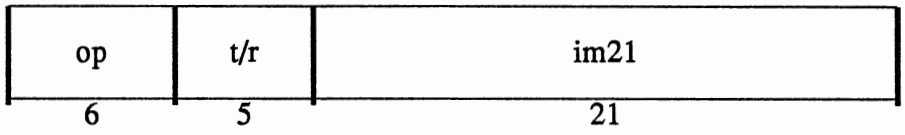
assembly instruction	annotation
ldw d(s,b), t	Load Word, op=12 hex
stw r, d(s,b)	Store Word, op=1A hex
ldo d(b), t	Load Offset, op=0D hex
ldb d(b), t	Load Byte, op=10 hex, s=0
stb r, d(b)	Store Byte, op=18 hex, s=0

2. Indexed Mode:



assembly instruction	annotation
ldwx,cmplt x(s,b),t	Load Word Indexed, op=03 hex, ext=2

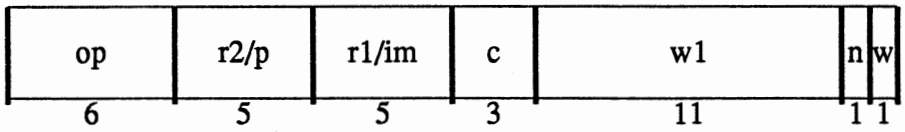
3. Long Immediate:



assembly instruction	annotation
ldil i, t	Load Immediate Left, op=08 hex

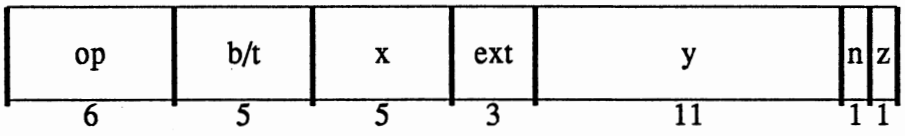
III. Control-Flow Instructions

1. Conditional Branches:



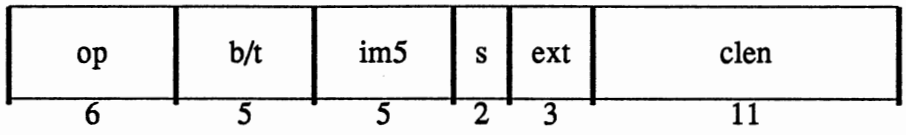
assembly instruction	annotation
movb,cond,n r1,r2,target	Move and Branch, op=32 hex
movib,cond,n i,r2,target	Move Immediate and Branch, op=33 hex
combt,cond,n r1,r2,target	Compare and Branch if True, op=20 hex
combf,cond,n r1,r2,target	Compare and Branch if False, op=22 hex
comibt,cond,n i,r2,target	Compare Immediate and Branch if true, op=21 hex
comibf,cond,n i,r2,target	Compare Immediate and Branch if false, op=23 hex
addbt,cond,n r1,r2,target	Add and Branch if True, op=28 hex
addbf,cond,n r1,r2,target	Add and Branch if False, op=2A hex
addibt,cond,n i,r2,target	Add Immediate and Branch if true, op=29 hex
addibf,cond,n i,r2,target	Add Immediate and Branch if false, op=2B hex
bb,cond,n r1, p, target	Branch on Bit, op=31 hex

2. Unconditional Branches:



assembly instruction	annotation
bl,n target, t	Branch and Link, op=3A hex, ext=0
ble,n target, t	Branch and Link External, op=3A hex, ext=3
bv,n x(b)	Branch Vectored, op=3A hex, ext=6, y=z=0

IV. Extended Instructions



assembly instruction	annotation
cra reg #, t	Customize Register Address, op=0C hex, ext=1, s=0, clen=0
alloc,s len, b	Allocate Memory, op=0C hex, ext=2, im5=0
nop	No Operation, op=0C hex, ext=3, other fields are 0's
ret	Terminate Program, op=0C hex, ext=4, other fields are 0's

APPENDIX B

LISTINGS OF THE TEST PROGRAMS

```
;*********************************************
; hanoi.asm -- assembly code of The Towers of Hanoi   *
;*********************************************
;
MAIN  algcon
;
;--- data ---
;r1: integer N
;
;--- code ---
  alloc,0  7FA, r16     ;allocate MAIN's exec. contour
;store register-spilling information
  stw  r0, 0000(2,r16)
  stw  r0, 0004(2,r16)
  stw  r0, 0008(2,r16)
  addi 085, r0, r17
  dep  r0, 1F, 08, r18
  dep  r17, 07, 18, r18
  stw  r18, 000C(2,r16)
;
;store antecedent link in MAIN'
  ldil  l%MAIN, r17
  ldo  r%MAIN(r17), r18
  stw  r18, 0010(2,r16)
;
;input prompt
  addi  069, r0, r17      ;'i'
  stb   r17, 0001(r0)
  addi  06E, r0, r17      ;'n'
  stb   r17, 0001(r0)
  addi  070, r0, r17      ;'p'
  stb   r17, 0001(r0)
  addi  075, r0, r17      ;'u'
  stb   r17, 0001(r0)
  addi  074, r0, r17      ;'t'
  stb   r17, 0001(r0)
  addi  020, r0, r17      ;' '
  stb   r17, 0001(r0)
  addi  04E, r0, r17      ;'N'
  stb   r17, 0001(r0)
  addi  03A, r0, r17      ;':'
  stb   r17, 0001(r0)
;
  addi  000, r0, r1       ;initialize N for accumulation
;
INPUTC:                   ;read N
  ldb  0000(r0), r17      ;input a character to r17
  addi  020, r0, r18      ;r18 = ASCII # of space
  addi  041, r0, r19      ;r19 = ASCII # of 'A'
```

```
  comibt,= 0A, r17, CRSP  ;line feed?
  nop
  comibt,= 0D, r17, CRSP  ;carriage return?
  nop
  combt,= r18, r17, CRSP  ;space?
  nop
  combt,<= r19, r17, ALPHA  ;the character belongs to 'A' to 'F'
  nop
  comibt,= 00, r0, ACCUM   ;delayed branch to ACCUM
  addi 7D0, r17, r17            ;convert '0' to '9' into hex value
ALPHA:
  addi 7C9, r17, r17            ;convert 'A' to 'F' into hex value
ACCUM:
  shd r1, r0, 03, r1           ;r1 <- r1 * 16
  comibt,= 00, r0, INPUTC  ;delayed branch back to INPUTC
  add r17, r1, r1              ;r1 <- r17 + r1
CRSP:
  addi 04E, r0, r17       ;r17 = 'N'
  stb r17, 0001(r0)       ;output 'N' onto screen
  addi 03D, r0, r17       ;r17 = '='
  stb r17, 0001(r0)       ;output '=' onto screen
  addi 020, r0, r17       ;r17 = ' '
  stb r17, 0001(r0)       ;output ' ' onto screen
;output the rightmost two hex digits of N
N_DIGIT1:
  extru r1, 04, 1C, r18     ;the 1st hex digit of N
  comibt,<= 0A, r18, N_NONDIGIT1
  nop
  addi 030, r18, r18            ;convert '0' to '9' into ASCII code
  comibt,= 00, r0, N_DIGIT2  ;delayed branch to next digit
  stb r18, 0001(r0)             ;output the hex digit
N_NONDIGIT1:
  addi 037, r18, r18       ;convert 'A' to 'F' into ASCII code
  stb r18, 0001(r0)        ;output the hex digit
N_DIGIT2:
  extru r1, 00, 1C, r18     ;the 2nd hex digit of N
  comibt,<= 0A, r18, N_NONDIGIT2
  nop
  addi 030, r18, r18           ;convert '0' to '9' into ASCII code
  comibt,= 00, r0, N_EXIT    ;delayed branch to exit
  stb r18, 0001(r0)            ;output the hex digit
N_NONDIGIT2:
  addi 037, r18, r18       ;convert 'A' to 'F' into ASCII code
  stb r18, 0001(r0)        ;output the hex digit
N_EXIT:
  addi 00D, r0, r18        ;r18 = CR
  stb r18, 0001(r0)        ;output CR onto screen
  addi 00A, r0, r18        ;r18 = LF
  stb r18, 0001(r0)        ;output LF onto screen
```

```
;
;pass parameters and call TOWERS
  add  r1, r0, r10          ;pass N to dummy parameter N
  addi 041, r0, r17         ;r17 = ASCII # of 'A'
  addi 042, r0, r18         ;r18 = ASCII # of 'B'
  dep  r17, 0F, 18, r11     ;pass 'A' to FROMA
  dep  r18, 17, 18, r11     ;pass 'B' to TOB
  addi 043, r0, r18         ;r18 = ASCII # of 'C'
  add  r16, r0, r12         ;pass static link
  dep  r18, 1F, 18, r11     ;pass 'C' to BYC
  add  r16, r0, r13          ;pass z.ep
  ble  TOWERS, r14      ;call TOWERS and save z.ip in r14
  nop
  ret       ;terminate program
  nop
;
MAIN  END


TOWERS  algcon
;--- data ---
;r26: integer N as input argument
;the input argument in r27 contains the following data:
;FROMA: character
;TOB: character
;BYC: character
;
;--- code ---
  alloc,0  7F5, r16      ;allocate TOWERS's exec. contour
;store register-spilling information
  addi 085, r0, r17
  addi 086, r0, r19
  dep  r17, 07, 18, r18
  dep  r19, 0F, 18, r18
  addi 087, r0, r17
  dep  r0, 1F, 18, r18
  dep  r17, 17, 18, r18
  stw  r18, 0000(2,r16)
  dep  r0, 0F, 10, r18
  addi 088, r0, r17
  addi 089, r0, r19
  dep  r17, 17, 18, r18
  dep  r19, 1F, 18, r18
  stw  r18, 0004(2,r16)
  stw  r0, 0008(2,r16)
  addi 08A, r0, r17
  dep  r0, 1F, 08, r18
  dep  r17, 07, 18, r18
  stw  r18, 000C(2,r16)
```

```
;
;store antecedent link
  ldil  l%TOWERS, r17
  ldo  r%TOWERS(r17), r18
  stw  r18, 0010(2,r16)
;static link in r28
;z.ep in r29
;z.ip in r30
;
;if N <= 0 then go to T_EXIT
  combt,<=  r26, r0, T_EXIT
  nop
;pass parameters and call recursively
  addi  7FF, r26, r10       ;pass N - 1 to N
  extru  r27, 10, 18, r17   ;extract FROMA
  extru  r27, 00, 18, r18   ;extract BYC
  dep  r17, 0F, 18, r11     ;pass FROMA to FROMA
  dep  r18, 17, 18, r11     ;pass BYC to TOB
  extru  r27, 08, 18, r17   ;extract TOB
  add  r28, r0, r12         ;pass static link
  dep  r17, 1F, 18, r11     ;pass TOB to BYC
  add  r16, r0, r13         ;pass z.ep
  bl  TOWERS, r14           ;call recursively and save z.ip in r14
  nop
;
  addi  04D, r0, r17        ;r17 = 'M'
  stb  r17, 0001(r0)        ;output 'M' onto screen
  addi  04F, r0, r17        ;r17 = 'O'
  stb  r17, 0001(r0)        ;output 'O' onto screen
  addi  056, r0, r17        ;r17 = 'V'
  stb  r17, 0001(r0)        ;output 'V' onto screen
  addi  045, r0, r17        ;r17 = 'E'
  stb  r17, 0001(r0)        ;output 'E' onto screen
  addi  020, r0, r17        ;r17 = ' '
  stb  r17, 0001(r0)        ;output ' ' onto screen
  addi  044, r0, r17        ;r17 = 'D'
  stb  r17, 0001(r0)        ;output 'D' onto screen
  addi  049, r0, r17        ;r17 = 'I'
  stb  r17, 0001(r0)        ;output 'I' onto screen
  addi  053, r0, r17        ;r17 = 'S'
  stb  r17, 0001(r0)        ;output 'S' onto screen
  addi  04B, r0, r17        ;r17 = 'K'
  stb  r17, 0001(r0)        ;output 'K' onto screen
  addi  020, r0, r17        ;r17 = ' '
  stb  r17, 0001(r0)        ;output ' ' onto screen
;
;output the rightmost two digits of N
N_DGT1:
  extru  r26, 04, 1C, r17       ;the 1st hex digit of N
```

```
  comibt,<= 0A, r17, N_NONDGT1
  nop
  addi 030, r17, r17          ;convert '0' to '9' into ASCII code
  comibt,= 00, r0, N_DGT2     ;delayed branch to next digit
  stb r17, 0001(r0)           ;output the hex digit
N_NONDGT1:
  addi 037, r17, r17          ;convert 'A' to 'F' into ASCII code
  stb r17, 0001(r0)           ;output the hex digit
N_DGT2:
  extru r26, 00, 1C, r17      ;the 2nd hex digit of N
  comibt,<= 0A, r17, N_NONDGT2
  nop
  addi 030, r17, r17          ;convert '0' to '9' into ASCII code
  comibt,= 00, r0, N_NEXT     ; and go to N_NEXT
  stb r17, 0001(r0)           ;output the hex digit
N_NONDGT2:
  addi 037, r17, r17          ;convert 'A' to 'F' into ASCII code
  stb r17, 0001(r0)           ;output the hex digit
N_NEXT:
  addi 020, r0, r17           ;r17 = ' '
  stb r17, 0001(r0)           ;output ' ' onto screen
  addi 046, r0, r17           ;r17 = 'F'
  stb r17, 0001(r0)           ;output 'F' onto screen
  addi 052, r0, r17           ;r17 = 'R'
  stb r17, 0001(r0)           ;output 'R' onto screen
  addi 04F, r0, r17           ;r17 = 'O'
  stb r17, 0001(r0)           ;output 'O' onto screen
  addi 04D, r0, r17           ;r17 = 'M'
  stb r17, 0001(r0)           ;output 'M' onto screen
  addi 020, r0, r17           ;r17 = ' '
  stb r17, 0001(r0)           ;output ' ' onto screen
;output FROMA
  extru r27, 10, 18, r17      ;extract FROMA
  stb r17, 0001(r0)           ;output FROMA onto screen
;
  addi 020, r0, r17           ;r17 = ' '
  stb r17, 0001(r0)           ;output ' ' onto screen
  addi 054, r0, r17           ;r17 = 'T'
  stb r17, 0001(r0)           ;output 'T' onto screen
  addi 04F, r0, r17           ;r17 = 'O'
  stb r17, 0001(r0)           ;output 'O' onto screen
  addi 020, r0, r17           ;r17 = ' '
  stb r17, 0001(r0)           ;output ' ' onto screen
;output TOB
  extru r27, 08, 18, r17      ;extract TOB
  stb r17, 0001(r0)           ;output TOB onto screen
;
  addi 00D, r0, r17           ;r17 = CR
  stb r17, 0001(r0)           ;output CR onto screen
```

```
  addi  00A, r0, r17          ;r17 = LF
  stb  r17, 0001(r0)          ;output LF onto screen
;
;pass parameters and call recursively
  addi  7FF, r26, r10         ;pass N - 1 to N
  extru  r27, 00, 18, r17     ;extract BYC
  extru  r27, 08, 18, r18     ;extract TOB
  dep  r17, 0F, 18, r11       ;pass BYC to FROMA
  dep  r18, 17, 18, r11       ;pass TOB to TOB
  extru  r27, 10, 18, r17     ;extract FROMA
  add  r28, r0, r12           ;pass static link
  dep  r17, 1F, 18, r11       ;pass FROMA to BYC
  add  r16, r0, r13           ;pass z.ep
  bl   TOWERS, r14            ;call recursively and save z.ip in r14
  nop
;
;retrieve parent's algorithm contour and return
T_EXIT:
  ldw  0010(2,r29), r17
  bv  r30(r17)
  nop
;
TOWERS  END
```

```
;*******************************************
; shortest.asm -- assembly code of The Shortest Path   *
;*******************************************
;
MAIN  algcon
;
;--- data ---
;r17: base of array A
;r18: base of array B
;r19: base of array L
; r1: integer I
; r2: integer NODE
;
;--- code ---
  alloc,0  7F5, r16     ;allocate MAIN's exec. contour
;store the register-spilling information
  stw  r0, 0000(2,r16)
  stw  r0, 0004(2,r16)
  addi 089, r0, r20
  addi 08A, r0, r22
  dep  r20, 07, 18, r21
  dep  r22, 0F, 18, r21
  dep  r0, 1F, 10, r21
  stw  r21, 0008(2,r16)
  addi 088, r0, r20
  addi 085, r0, r22
  dep  r20, 07, 18, r21
  dep  r22, 0F, 18, r21
  addi 086, r0, r20
  addi 087, r0, r22
  dep  r20, 17, 18, r21
  dep  r22, 1F, 18, r21
  stw  r21, 000C(2,r16)
;
  ldil l%MAIN, r20
  ldo  r%MAIN(r20), r21
  stw  r21, 0010(2,r16)  ;store antecedent link
  alloc,1  7C0, r17      ;allocate array A in heap
  alloc,1  7F8, r18      ;allocate array B in heap
  stw  r18, 0018(2,r16)  ;store B's base offset in MAIN'
  alloc,1  7F8, r19      ;allocate array L in heap
  alloc,1  7F8, r3       ;allocate set V in heap
  alloc,1  7F8, r4       ;allocate set S in heap
;
;initialize set V
  addi  001, r0, r20        ;r20 = 1, initial value of I
  addi  000, r0, r21        ;r21 = 0, index of V
LOADV:
  stwx,S  r20, r21(1,r3)        ;store I in set V
```

```
  addi  001, r21, r21          ;increment the index of set V
  comibf,>  08, r20, LOADV     ;if I < 8 then
  addi  001, r20, r20          ;increment I and go back to LOADV
;input prompt
  addi 069, r0, r20     ;'i'
  stb  r20, 0001(r0)
  addi 06E, r0, r20     ;'n'
  stb  r20, 0001(r0)
  addi 070, r0, r20     ;'p'
  stb  r20, 0001(r0)
  addi 075, r0, r20     ;'u'
  stb  r20, 0001(r0)
  addi 074, r0, r20     ;'t'
  stb  r20, 0001(r0)
  addi 020, r0, r20     ;' '
  stb  r20, 0001(r0)
  addi 036, r0, r20     ;'6'
  stb  r20, 0001(r0)
  addi 034, r0, r20     ;'4'
  stb  r20, 0001(r0)
  addi 020, r0, r20     ;' '
  stb  r20, 0001(r0)
  addi 06E, r0, r20     ;'n'
  stb  r20, 0001(r0)
  addi 075, r0, r20     ;'u'
  stb  r20, 0001(r0)
  addi 06D, r0, r20     ;'m'
  stb  r20, 0001(r0)
  addi 062, r0, r20     ;'b'
  stb  r20, 0001(r0)
  addi 065, r0, r20     ;'e'
  stb  r20, 0001(r0)
  addi 072, r0, r20     ;'r'
  stb  r20, 0001(r0)
  addi 073, r0, r20     ;'s'
  stb  r20, 0001(r0)
  addi 03A, r0, r20     ;':'
  stb  r20, 0001(r0)
  addi 00D, r0, r20     ;CR
  stb  r20, 0001(r0)
  addi 00A, r0, r20     ;LF
  stb  r20, 0001(r0)
;
;pass parameters and call READINDATA
  add  r17, r0, r10            ;pass A's base offset
  add  r16, r0, r11            ;pass stat. link and z.ep
  ble  READINDATA, r12   ;call READINDATA and save z.ip in r12
  nop
;
```

```
  addi  001, r0, r2   ;NODE = 1
;[ for NODE = 1 to 8 do ]
M_LOOP1:
;pass parameters and call DIJKSTRA
  add  r17, r0, r10        ;pass A's base offset
  add  r19, r0, r11        ;pass L's base offset
  add  r18, r0, r12        ;pass B's base offset
  addi  000, r2, r13       ;pass NODE
  add  r16, r0, r14        ;pass stat. link and z.ep
  ble  DIJKSTRA, r15   ;call DIJKSTRA and save z.ip in r15
  nop
;
  addi  001, r0, r1   ;I = 1
;[ for I = 1 to 8 do ]
M_LOOP2:
  addi  04C, r0, r20    ;r20 = 'L'
  stb  r20, 0001(r0)    ;output 'L' onto screen
  addi  045, r0, r20    ;r20 = 'E'
  stb  r20, 0001(r0)    ;output 'E' onto screen
  addi  04E, r0, r20    ;r20 = 'N'
  stb  r20, 0001(r0)    ;output 'N' onto screen
  addi  047, r0, r20    ;r20 = 'G'
  stb  r20, 0001(r0)    ;output 'G' onto screen
  addi  054, r0, r20    ;r20 = 'T'
  stb  r20, 0001(r0)    ;output 'T' onto screen
  addi  048, r0, r20    ;r20 = 'H'
  stb  r20, 0001(r0)    ;output 'H' onto screen
  addi  020, r0, r20    ;r20 = ' '
  stb  r20, 0001(r0)    ;output ' ' onto screen
  addi  03D, r0, r20    ;r20 = '='
  stb  r20, 0001(r0)    ;output '=' onto screen
;
  addi  7FF, r1, r20          ;r20 = I - 1
  ldwx,S  r20(1,r19), r21    ;load L[I] into r21
;
;output the rightmost 4 hex digits of the value L[I]
L_DIGIT1:
  extru  r21, 0C, 1C, r20   ;the 1st hex digit of L[I]
  comibt,<=  0A, r20, L_NONDIGIT1
  nop
  addi  030, r20, r20              ;convert '0' to '9' into ASCII #
  comibt,=  00, r0, L_DIGIT2  ;delayed-branch to next digit
  stb  r20, 0001(r0)               ;output the hex digit
L_NONDIGIT1:
  addi  037, r20, r20       ;convert 'A' to 'F' into ASCII #
  stb  r20, 0001(r0)        ;output the hex digit
;
```

```
L_DIGIT2:
  extru  r21, 08, 1C, r20   ;the 2nd hex digit of L[I]
  comibt,<= 0A, r20, L_NONDIGIT2
  nop
  addi  030, r20, r20              ;convert '0' to '9' into ASCII #
  comibt,= 00, r0, L_DIGIT3  ;delayed-branch to next digit
  stb  r20, 0001(r0)               ;output the hex digit
L_NONDIGIT2:
  addi  037, r20, r20       ;convert 'A' to 'F' into ASCII #
  stb  r20, 0001(r0)        ;output the hex digit
;
L_DIGIT3:
  extru  r21, 04, 1C, r20   ;the 3rd hex digit of L[I]
  comibt,<= 0A, r20, L_NONDIGIT3
  nop
  addi  030, r20, r20              ;convert '0' to '9' into ASCII #
  comibt,= 00, r0, L_DIGIT4  ;delayed branch to next digit
  stb  r20, 0001(r0)               ;output the hex digit
L_NONDIGIT3:
  addi  037, r20, r20       ;convert 'A' to 'F' into ASCII #
  stb  r20, 0001(r0)        ;output the hex digit
;
L_DIGIT4:
  extru  r21, 00, 1C, r20   ;the 4th hex digit of L[I]
  comibt,<= 0A, r20, L_NONDIGIT4
  nop
  addi  030, r20, r20           ;convert '0' to '9' into ASCII #
  comibt,= 00, r0, L_EXIT   ;delayed branch to exit
  stb  r20, 0001(r0)               ;output the hex digit
L_NONDIGIT4:
  addi  037, r20, r20       ;convert 'A' to 'F' into ASCII #
  stb  r20, 0001(r0)        ;output the hex digit
;
L_EXIT:
  addi  020, r0, r20     ;r20 = ' '
  stb  r20, 0001(r0)     ;output ' ' onto screen
  addi  046, r0, r20     ;r20 = 'F'
  stb  r20, 0001(r0)     ;output 'F' onto screen
  addi  052, r0, r20     ;r20 = 'R'
  stb  r20, 0001(r0)     ;output 'R' onto screen
  addi  04F, r0, r20     ;r20 = 'O'
  stb  r20, 0001(r0)     ;output 'O' onto screen
  addi  04D, r0, r20     ;r20 = 'M'
  stb  r20, 0001(r0)     ;output 'M' onto screen
  addi  020, r0, r20     ;r20 = ' '
  stb  r20, 0001(r0)     ;output ' ' onto screen
  addi  04E, r0, r20     ;r20 = 'N'
  stb  r20, 0001(r0)     ;output 'N' onto screen
  addi  04F, r0, r20     ;r20 = 'O'
```

```
  stb  r20, 0001(r0)     ;output 'O' onto screen
  addi  044, r0, r20     ;r20 = 'D'
  stb  r20, 0001(r0)     ;output 'D' onto screen
  addi  045, r0, r20     ;r20 = 'E'
  stb  r20, 0001(r0)     ;output 'E' onto screen
  addi  020, r0, r20     ;r20 = ' '
  stb  r20, 0001(r0)     ;output ' ' onto screen
;
;output the rightmost two hex digits of NODE in r2
N_DIGIT1:
  extru  r2, 04, 1C, r20   ;the 1st hex digit of NODE
  comibt,<= 0A, r20, N_NONDIGIT1
  nop
  addi  030, r20, r20              ;convert '0' to '9' into ASCII #
  comibt,= 00, r0, N_DIGIT2  ;delayed branch to next digit
  stb  r20, 0001(r0)               ;output the hex digit
N_NONDIGIT1:
  addi  037, r20, r20        ;convert 'A'to 'F' into ASCII #
  stb  r20, 0001(r0)         ;output the hex digit
;
N_DIGIT2:
  extru  r2, 00, 1C, r20   ;the 2nd hex digit of NODE
  comibt,<= 0A, r20, N_NONDIGIT2
  nop
  addi  030, r20, r20           ;convert '0' to '9' into ASCII #
  comibt,= 00, r0, N_EXIT  ;and go to N_EXIT
  stb  r20, 0001(r0)            ;output the hex digit
N_NONDIGIT2:
  addi  037, r20, r20        ;convert 'A' to 'F' into ASCII #
  stb  r20, 0001(r0)         ;output the hex digit
;
N_EXIT:
;pass parameters and call PRINTPATH
  addi  000, r2, r10         ;pass NODE
  addi  000, r1, r11         ;pass I
  add  r16, r0, r12          ;pass stat. link
  add  r16, r0, r13          ;pass z.ep
  ble  PRINTPATH, r14   ;call PRINTPATH and save z.ip in r14
  nop
;
  addi  00D, r0, r20      ;r20 = CR
  stb  r20, 0001(r0)      ;output CR onto screen
  addi  00A, r0, r20      ;r20 = LF
  stb  r20, 0001(r0)      ;output LF onto screen
  comibf,> 08, r1, M_LOOP2  ;if I < 8 then
  addi  001, r1, r1     ;increment I and go to M_LOOP2
  comibf,> 08, r2, M_LOOP1  ;if NODE < 8 then
  addi  001, r2, r2     ;increment NODE and go to M_LOOP1
  ret
```

```
  nop
;
MAIN  END


READINDATA  algcon
;--- data ---
;r26: base of array A as input argument
;r17: integer I
;r18: integer J
;
;--- code ---
;z.ep in r27
;z.ip in r28
;
   addi  001, r0, r17        ;I = 1
;[ for I = 1 to 8 do ]
R_LOOP1:
   addi  001, r0, r18        ;J = 1
;[ for J = 1 to 8 do ]
R_LOOP2:
   addi  000, r0, r19        ;r19 as accumulator
INPUTC:
   ldb  0000(r0), r20        ;input a character to r20
   addi  020, r0, r21        ;r21 = ASCII # of space
   addi  041, r0, r22        ;r22 = 41 hex
   comibt,=  0A, r20, _CRSP   ;line feed?
   nop
   comibt,=  0D, r20, _CRSP   ;carriage return?
   nop
   combt,=  r21, r20, _CRSP   ;space?
   nop
   combt,<=  r22, r20, ALPHA  ;character belongs to 'A' to 'F'
   nop
   comibt,=  00, r0, ACCUM  ;delayed branch to ACCUM
   addi  7D0, r20, r20          ;convert '0' to '9' into hex value
ALPHA:
   addi  7C9, r20, r20          ;convert 'A' to 'F' into hex value
ACCUM:
   shd  r19, r0, 03, r19          ;r19 <- r19 * 16
   comibt,=  00, r0, INPUTC  ;delayed branch to INPUTC
   add  r20, r19, r19           ;r19 <- r19 + r20
_CRSP:
;put the entered value into A[I,J]
   addi  7FF, r17, r20       ;r20 = I - 1
   addi  7FF, r18, r21       ;r21 = J - 1
   sh3add  r20, r21, r20     ;r20 = (I - 1)*8 + (J - 1)
   stwx,S  r19, r20(1,r26)   ;store the entered value in A[I,J]
   comibf,>  08, r18, R_LOOP2 ;if J < 8 then increment J
```

```
  addi  001, r18, r18              ;and go back to R_LOOP2
  comibf,>  08, r17, R_LOOP1 ;if I < 8 then increment I
  addi  001, r17, r17              ;and go back to R_LOOP1
;
;retrieve parent's algorithm contour and return
  ldw  0010(2,r27), r19
  bv  r28(r19)
  nop
;
READINDATA  END


DIJKSTRA   algcon
;--- data ---
;r26: base of array A as input argument
;r27: base of array L as input argument
;r28: base of array B as input argument
;r29: integer FROM as input argument
;r17: integer I
;r18: integer J
;r19: integer K
;r20: integer M
;r21: integer MIN
;
;--- code ---
;z.ep in r30
;z.ip in r31
;
  addi  001, r0, r17      ;I = 1
;[ for I = 1 to 8 do ]
D_LOOP1:
  addi  7FF, r17, r24          ;r24 = I - 1
  addi  7FF, r29, r25          ;r25 = FROM - 1
  sh3add  r25, r24, r25        ;r25 = (FROM-1)*8+(I-1)
  ldwx,S  r25(1,r26), r25      ;r25 = A[FROM, I]
  stwx,S  r25, r24(1,r27)      ;L[I] = A[FROM, I]
  stwx,S  r29, r24(1,r28)      ;B[I] = FROM
  comibf,>  08, r17, D_LOOP1  ;if I < 8 then
  addi  001, r17, r17     ;increment I and go back to D_LOOP1
;initialize set S
  stw  r29, 0000(1,r4)  ;store FROM in set S
  addi  000, r0, r24      ;r24 serves as the top-of-set index to S
;
  addi  001, r0, r20     ;M = 1
;[ for M = 1 to 8 do ]
D_LOOP2:
  ldil  00001F, r21
  ldo  07FF(r21), r21   ;MIN = X'FFFF
  addi  001, r0, r17      ;I =1
```

```
;[ for I = 1 to 8 do ]
D_LOOP3:
;if (I in V-S) and (MIN > L[I]) then  K = I and MIN = L[I]
  add  r24, r0, r22      ;copy the index
;I in V-S ?
I_VMINUSS:
  ldwx,S  r22(1,r4), r25     ;load the top element of S into r25
  combt,=  r17, r25, D_EXITIF1  ;I is in S, quit testing
  nop
  addi  7FF, r22, r22     ;decrement the index
  comibt,<=  00, r22, I_VMINUSS   ;go for next element in S
  nop
;
;MIN > L[I] ?
  addi  7FF, r17, r22                ;r22 = I - 1
  ldwx,S  r22(1,r27), r25            ;r25 = L[I]
  combt,<=  r21, r25, D_EXITIF1  ;MIN <= L[I], quit testing
  nop
  add  r17, r0, r19          ;K = I
  add  r25, r0, r21          ;MIN = L[I]
;
D_EXITIF1:
  comibf,>  08, r17, D_LOOP3    ;if I < 8 then
  addi  001, r17, r17     ;increment I and go to D_LOOP3
;
  addi  001, r24, r24        ;increment top-of-stack index
  addi  001, r0, r18         ;J = 1
  stwx,S  r19, r24(1,r4)     ;store K as the top-of-set element in S
;
;[ for J = 1 to 8 do ]
D_LOOP4:
;
;if (J in V-S) and (L[K] + A[K,J] < L[J]) then
;L[J] = L[K] + A[K,J] and B[J] = K
  add  r24, r0, r22          ;copy the index
;J in V-S ?
J_VMINUSS:
  ldwx,S  r22(1,r4), r25  ;load the topmost element of S into r25
  combt,=  r18, r25, D_EXITIF2  ;J is in S, quit testing
  nop
  addi  7FF, r22, r22                ;decrement index
  comibt,<=  00, r22, J_VMINUSS ;check next element in S
  nop
;
;L[K] + A[K,J] < L[J] ?
  addi  7FF, r19, r15        ;r15 = K - 1
  addi  7FF, r18, r14        ;r14 = J - 1
  ldwx,S  r15(1,r27), r22    ;r22 = L[K]
  sh3add  r15, r14, r15      ;r15 = (K - 1)*8 + (J - 1)
```

```
  ldwx,S  r15(1,r26), r25     ;r25 = A[K, J]
  add  r22, r25, r22          ;r22 = L[K] + A[K,J]
  ldwx,S  r14(1,r27), r25     ;r25 = L[J]
;if L[K] + A[K,J] >= L[J] then quit testing
  combf,>=  r22, r25, D_EXITIF2
  nop
  stwx,S  r22, r14(1,r27)     ;L[J] = L[K] + A[K,J]
  stwx,S  r19, r14(1,r28)     ;B[J] = K
;
D_EXITIF2:
  comibf,>  08, r18, D_LOOP4  ;if J < 8 then increment J
  addi  001, r18, r18         ;and go to D_LOOP4
  comibf,>  08, r20, D_LOOP2  ;if M < 8 then increment M
  addi  001, r20, r20         ;and go to D_LOOP2
;
;retrieve parent's algorithm contour and return
  ldw  0010(2,r30), r24
  bv  r31(r24)
  nop
;
DIJKSTRA  END


PRINTPATH  algcon
;--- data ---
;r26: integer I as input argument
;r27: integer J as input argument
;
;--- code ---
  alloc,0  7F2, r16   ;allocate PRINTPATH's exec. contour
;store the register-spilling information
  addi 085, r0, r17
  addi 086, r0, r19
  dep  r17, 07, 18, r18
  dep  r19, 0F, 18, r18
  addi 087, r0, r17
  dep  r0, 1F, 18, r18
  dep  r17, 17, 18, r18
  stw  r18, 0000(2,r16)
  dep  r0, 0F, 10, r18
  addi 088, r0, r17
  addi 089, r0, r19
  dep  r17, 17, 18, r18
  dep  r19, 1F, 18, r18
  stw  r18, 0004(2,r16)
  stw  r0, 0008(2,r16)
  addi 08A, r0, r17
  addi 08B, r0, r19
  dep  r17, 07, 18, r18
```

```
  dep  r19, 0F, 18, r18
  addi 08C, r0, r17
  addi 08D, r0, r19
  dep  r17, 17, 18, r18
  dep  r19, 1F, 18, r18
  stw  r18, 000C(2,r16)
;
;store antecedent link
  ldil l%PRINTPATH, r17
  ldo  r%PRINTPATH(r17), r18
  stw  r18, 0010(2,r16)
;static link in r28
;z.ep in r29
;z.ip in r30
  ldw  0018(2,r28), r17        ;load array B's base offset in r17
  addi  7FF, r27, r18          ;r18 = J - 1
  ldwx,S  r18(1,r17), r18      ;r18 = B[J]
  combt,=  r18, r26, P_EXIT  ;if B[J] = I then go to P_EXIT
  nop
  addi  000, r26, r10       ;pass I (dummy parameter I)
  addi  000, r18, r11       ;pass B[J] (dummy parameter J)
  add  r28, r0, r12         ;pass stat. link
  add  r16, r0, r13         ;pass z.ep
  bl  PRINTPATH, r14    ;recursive call and save z.ip in r14
  nop
P_EXIT:
  addi  020, r0, r19        ;r19 = ' '
  stb  r19, 0001(r0)        ;output ' ' onto screen
  addi  054, r0, r19        ;r19 = 'T'
  stb  r19, 0001(r0)        ;output 'T' onto screen
  addi  04F, r0, r19        ;r19 = 'O'
  stb  r19, 0001(r0)        ;output 'O' onto screen
  addi  020, r0, r19        ;r19 = ' '
  stb  r19, 0001(r0)        ;output ' ' onto screen
;output the rightmost two hex digits of J
J_DIGIT1:
  extru  r27, 04, 1C, r19    ;the 1st hex digit of J
  comibt,<=  0A, r19, J_NONDIGIT1
  nop
  addi  030, r19, r19            ;convert '0' to '9' into ASCII code
  comibt,=  00, r0, J_DIGIT2 ;delayed branch to next digit
  stb  r19, 0001(r0)             ;output the hex digit
J_NONDIGIT1:
  addi  037, r19, r19     ;convert 'A' to 'F' into ASCII code
  stb  r19, 0001(r0)      ;output the hex digit
J_DIGIT2:
  extru  r27, 00, 1C, r19    ;the 2nd hex digit of J
  comibt,<=  0A, r19, J_NONDIGIT2
  nop
```

```
  addi  030, r19, r19          ;convert '0' to '9' into ASCII code
  comibt,=  00, r0, J_EXIT  ;and go to J_EXIT
  stb  r19, 0001(r0)           ;output the hex digit
J_NONDIGIT2:
  addi  037, r19, r19     ;convert 'A' to 'F' into ASCII code
  stb  r19, 0001(r0)      ;output the hex digit
J_EXIT:
  ldw  0010(2,r29), r19    ;load return algorithm-contour into r19
  bv  r30(r19)             ;return
  nop
;
PRINTPATH  END
```

APPENDIX C

THE SIMULATOR-GENERATED PROFILES

1. The Simulation Results of The Towers of Hanoi

*** simulator-generated profile ***

test program: /v/ying/thesis/code/hanoi.asm

Clocks:	5832662 cycles
# of Instructions:	4521892
# of Saved CPU Slots:	65473
# of Branches:	229377
# of NOPs:	262146
# of Bus Cycles (src1):	2752487
# of Bus Cycles (src2):	720837
# of Bus Cycles (result):	2752487
# of Bus Cycles (external):	1114110
# of Loads:	79852
# of Stores:	333813
# of Register-Spills:	6133
# of Register-Restores:	6133
# of Window Overflows:	1023
# of Window Underflows:	1023

****
* Register Usage *
****

Global Registers (r0--r9): 1638330 8 0 0 0 0 0 0 0 0
Local Registers (r16--r31) of Window 0--7:
w0: 197640 403238 461082 131587 0 0 0 0
0 0 66304 99712 33152 65792 65792 0
w1: 2056 20046 4626 1028 0 0 0 0
0 0 1542 2827 771 514 514 0
w2: 4112 40092 9252 2056 0 0 0 0
0 0 3084 5654 1542 1028 1028 0
w3: 8224 80184 18504 4112 0 0 0 0
0 0 6168 11308 3084 2056 2056 0
w4: 16448 160368 37008 8224 0 0 0 0
0 0 12336 22616 6168 4112 4112 0
w5: 32896 320736 74016 16448 0 0 0 0
0 0 24672 45232 12336 8224 8224 0
w6: 65792 641472 148032 32896 0 0 0 0
0 0 49344 90464 24672 16448 16448 0
w7: 131584 1282944 296064 65792 0 0 0 0
0 0 98688 180928 49344 32896 32896 0

TABLE II

THE INSTRUCTION DISTRIBUTION TABLE
OF THE TOWERS OF HANOI

Instruction	count	percentage
add	131072	2.90%
addc	0	0.00%
and	0	0.00%
or	0	0.00%
sub	0	0.00%
xor	0	0.00%
sh2add	0	0.00%
sh3add	0	0.00%
addi	1245177	27.54%
shd	1	0.00%
extru	327672	7.25%
extrs	0	0.00%
dep	786422	17.39%
depi	0	0.00%
ldw	65535	1.45%
stw	327680	7.25%
ldo	65536	1.45%
ldb	2	0.00%
stb	851957	18.84%
ldwx	0	0.00%
stwx	0	0.00%
ldil	65536	1.45%
bl	65534	1.45%
ble	1	0.00%
bv	65535	1.45%
movb	0	0.00%
movib	0	0.00%
combt	65537	1.45%
combf	0	0.00%
comibt	131012	2.90%
comibf	0	0.00%
addbt	0	0.00%
addbf	0	0.00%
addibt	0	0.00%
addibf	0	0.00%
bb	0	0.00%
alloc	65536	1.45%
nop	262146	5.80%
ret	1	0.00%

2. The Simulation Results of The Shortest Path

*** simulator-generated profile ***

test program: /v/ying/thesis/code/shortest.asm

Clocks:	59507 cycles
# of Instructions:	42553
# of Saved CPU Slots:	2112
# of Branches:	5879
# of NOPs:	8674
# of Bus Cycles (src1):	24039
# of Bus Cycles (src2):	18421
# of Bus Cycles (result):	14793
# of Bus Cycles (external):	12436
# of Loads:	4769
# of Stores:	966
# of Register-Spills:	0
# of Register-Restores:	0
# of Window Overflows:	0
# of Window Underflows:	0

****
* Register Usage *
****

Global Registers (r0--r9): 6440 328 225 9 3617 0 0 0 0 0
Local Registers (r16--r31) of Window 0--7:

w0: 144 10 10 73 4815 356 6 0
0 0 0 0 0 0 0 0
w1: 420 4801 5260 2942 1992 1084 14856 0
1496 8688 469 934 282 272 144 16
w2: 232 504 736 1008 0 0 0 0
0 0 88 144 88 72 664 840
w3: 100 224 324 448 0 0 0 0
0 0 36 64 36 32 32 0
w4: 24 56 80 112 0 0 0 0
0 0 8 16 8 8 8 0
w5: 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
w6: 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
w7: 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0

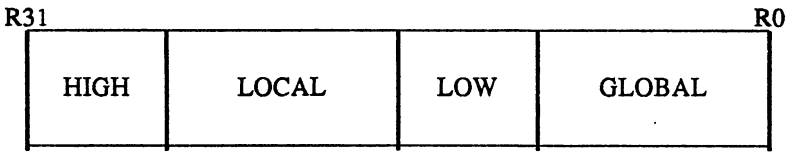
TABLE III

THE INSTRUCTION DISTRIBUTION TABLE
OF THE SHORTEST PATH

Instruction	count	percentage
add	1820	4.28%
addc	0	0.00%
and	0	0.00%
or	0	0.00%
sub	0	0.00%
xor	0	0.00%
sh2add	0	0.00%
sh3add	296	0.70%
addi	9939	23.36%
shd	162	0.38%
extru	624	1.47%
extrs	0	0.00%
dep	1327	3.12%
depi	0	0.00%
ldw	249	0.59%
stw	614	1.44%
ldo	185	0.43%
ldb	226	0.53%
stb	2467	5.80%
ldwx	4520	10.62%
stwx	352	0.83%
ldil	185	0.43%
bl	56	0.13%
ble	73	0.17%
bv	129	0.30%
movb	0	0.00%
movib	0	0.00%
combt	4268	10.03%
combf	168	0.39%
comibt	4788	11.25%
comibf	1304	3.06%
addbt	0	0.00%
addbf	0	0.00%
addibt	0	0.00%
addibf	0	0.00%
bb	0	0.00%
alloc	126	0.30%
nop	8674	20.38%
ret	1	0.00%

APPENDIX D

FIGURES



HIGH: [R31..R26]
LOCAL: [R25..R16]
LOW: [R15..R10]
GLOBAL: [R9..R0]

Figure 1. A Register Window

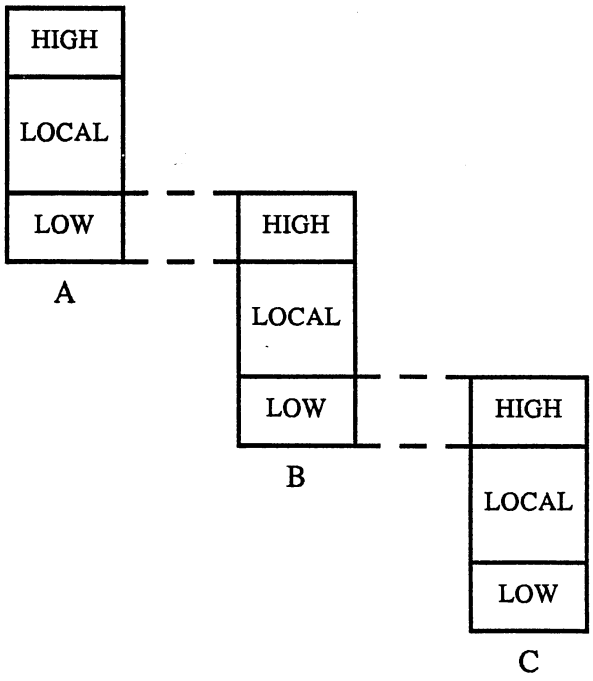


Figure 2. The Overlapped Register Windows of Nested Procedure Calls

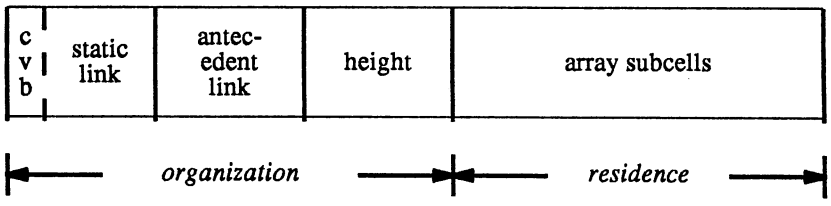


Figure 3. The Generic Format of a Contour Cell

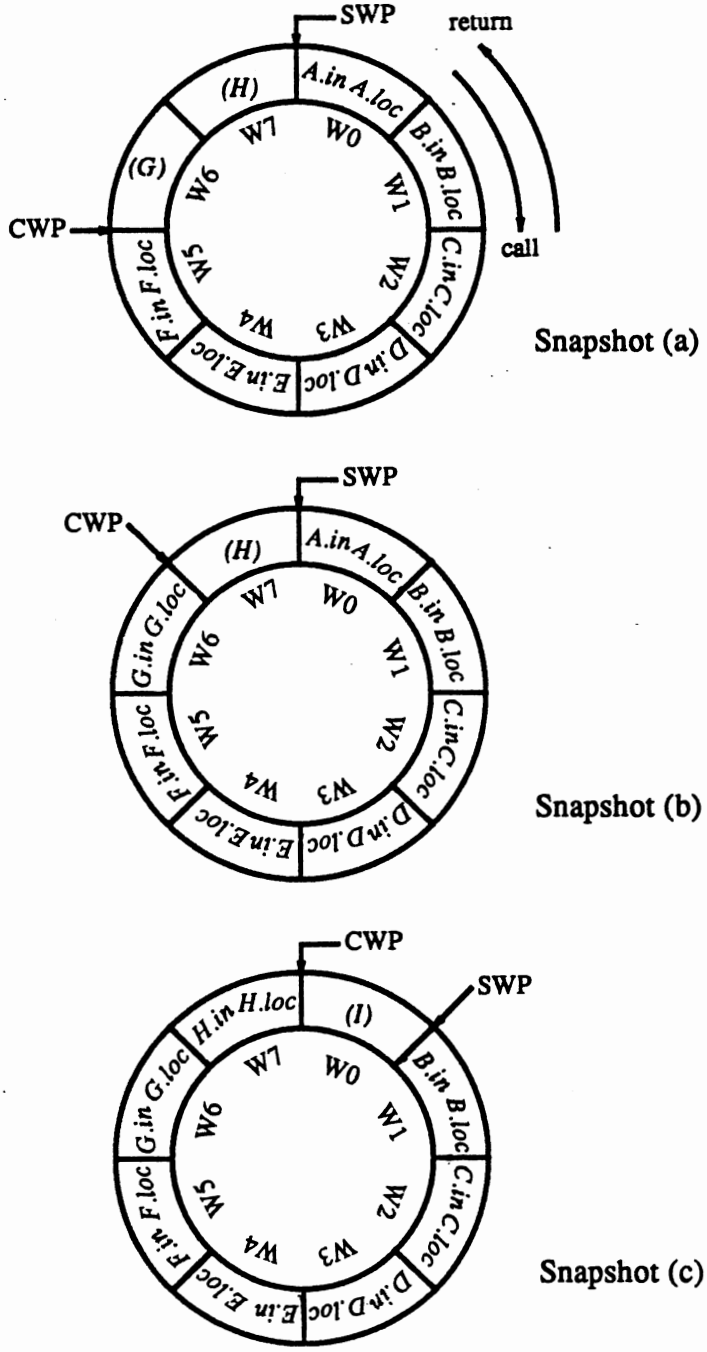
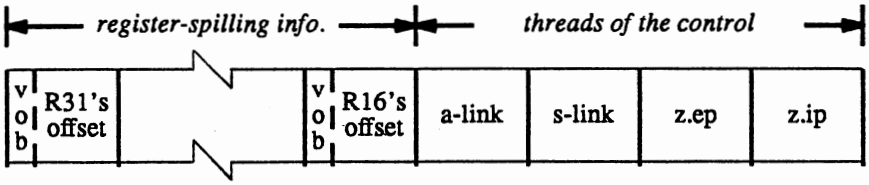
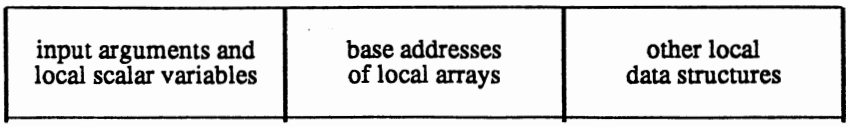


Figure 4. The Circular Buffer Organization of the Multi-Windowing Register Set



(a) The Control Part



(b) The Data Part

Figure 5. The Organization of an Execution Contour

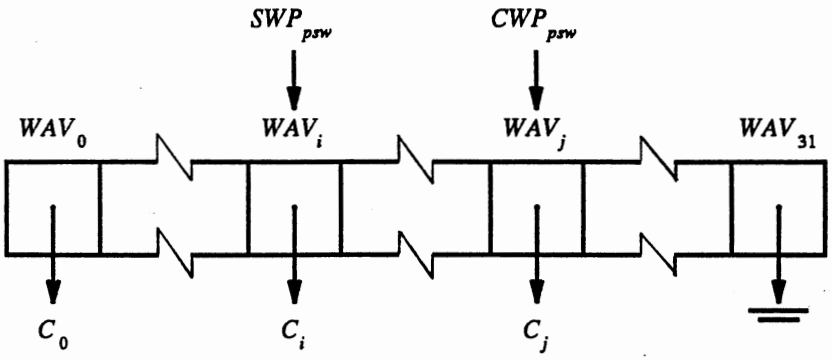


Figure 6. The Window Activation Vector

```
PROGRAM MAIN (INPUT, OUTPUT);
 VAR
  P, Q : INTEGER;

 PROCEDURE BB (PAR1, PAR2 : INTEGER);
  VAR
   P, R, S : INTEGER;

  PROCEDURE CC (VAR PAR1 : INTEGER);
   VAR
    X, Y : INTEGER;
   BEGIN
    .
    .
    .
   END; {CC}

  BEGIN
   .
   .
   .
  END; {BB}

 FUNCTION DD (PAR1 : INTEGER) : REAL;
  VAR
   Z : INTEGER;
  BEGIN
   .
   .
   .
  END; {DD}

BEGIN
 .
 .
 .
END. {MAIN}
```

Figure 7. The Block Structure of a Pascal Program

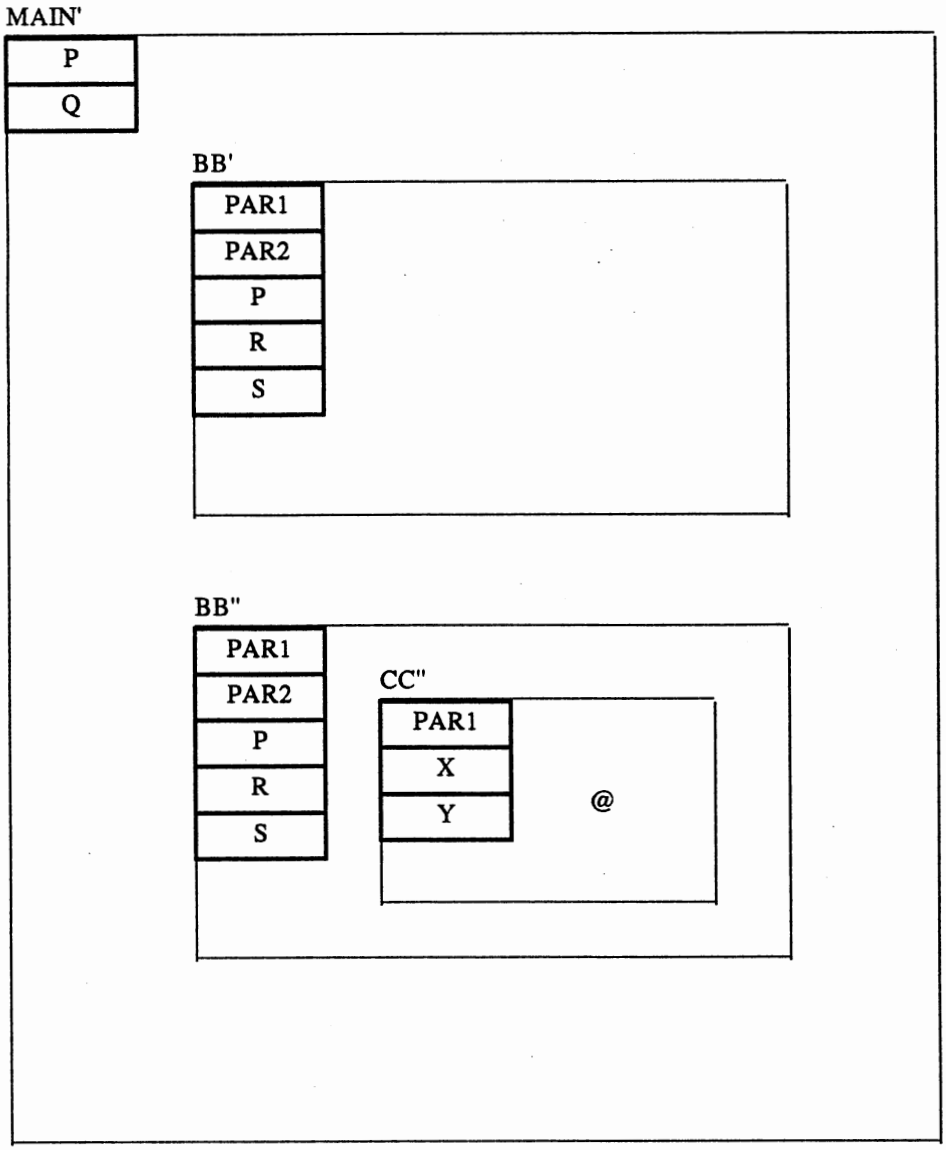


Figure 8. The Topographic Contour-Map of a Snapshot During the Execution of the Program in Figure 7

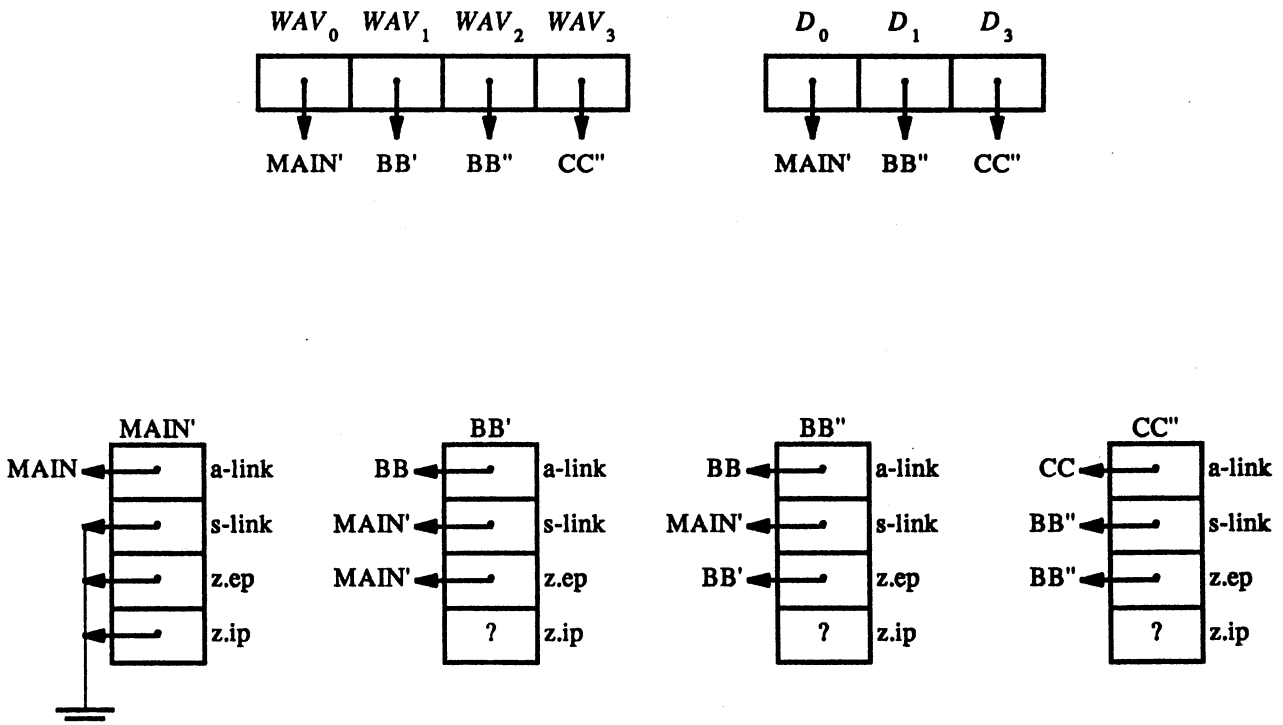


Figure 9. The Control Structure of the Snapshot Shown in Figure 8

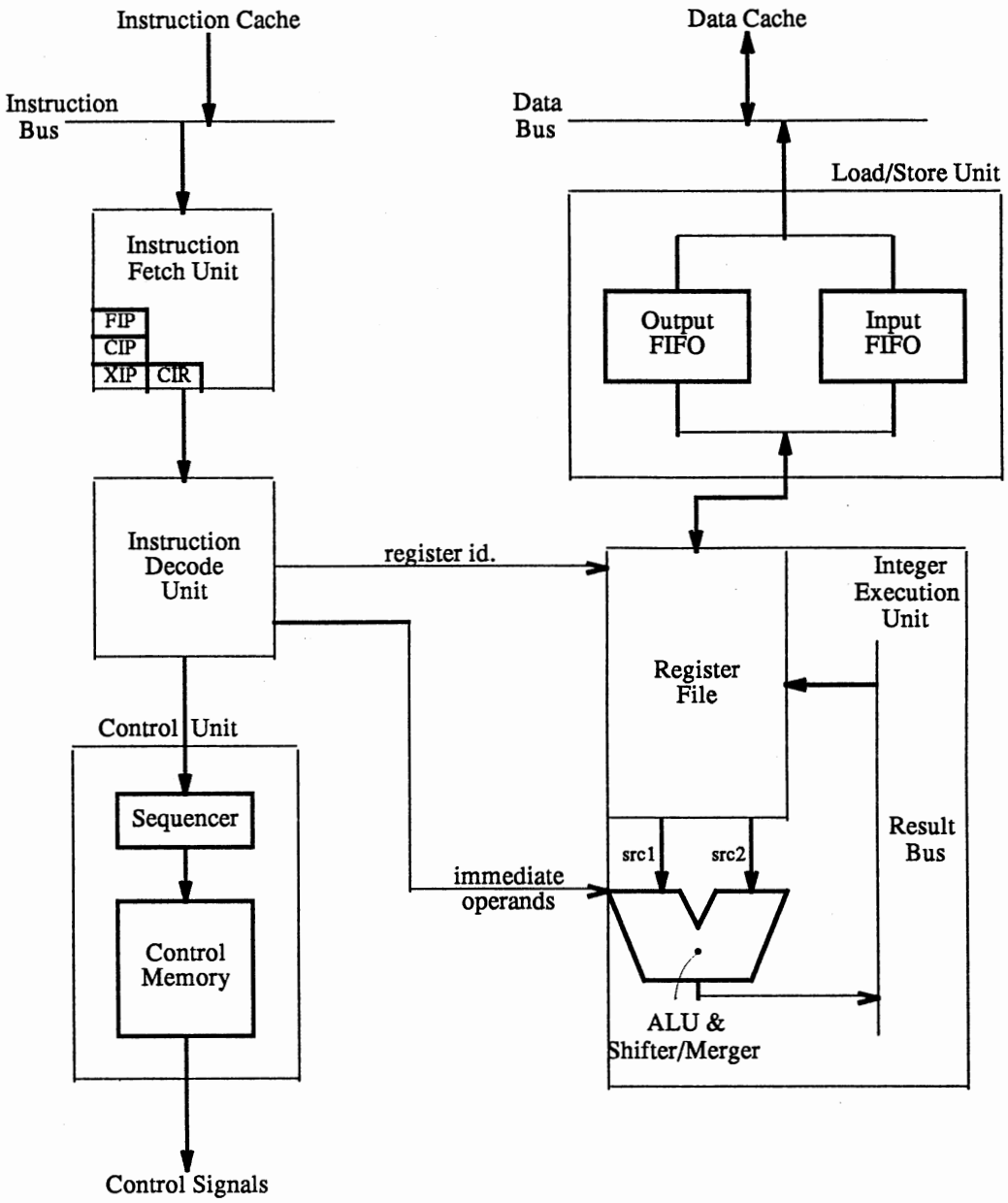


Figure 10. The Block Daigram of HMA's Processor Design

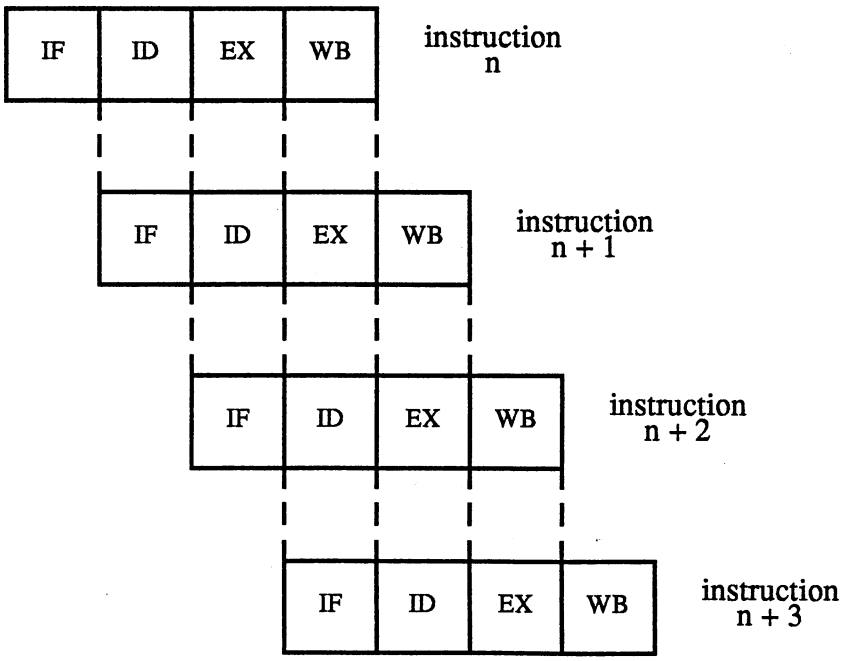


Figure 11. Instruction Pipelining

```
.
.
add  r1, r2, r3  ;r1 + r2 → r3
and  r3, r4, r4  ;r3 & r4 → r4
.
.
```

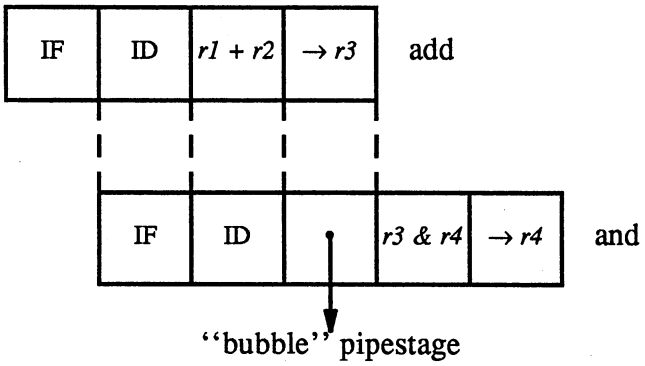


Figure 12. Data Dependency and Pipeline "Bubble"

```
L1:
   .
   .
   ldw  0(1,r6), r1       ;load a word into r1
   sub  r2, r1, r3        ;r3 ← r2 – r1
   addi 1, r7, r7         ;increment r7 by 1
   comibt,= 0, r0, L1     ;jump to L1
   .
   .
```

(a) Before Optimization

```
L1:
   .
   .
   ldw  0(1,r6), r1       ;load a word into r1
   addi 1, r7, r7         ;increment r7 by 1
   comibt,= 0, r0, L1     ;jump to L1
   sub  r2, r1, r3        ;r3 ← r2 – r1
   .
   .
```

(b) After Optimization

Figure 13. The Delayed Branch

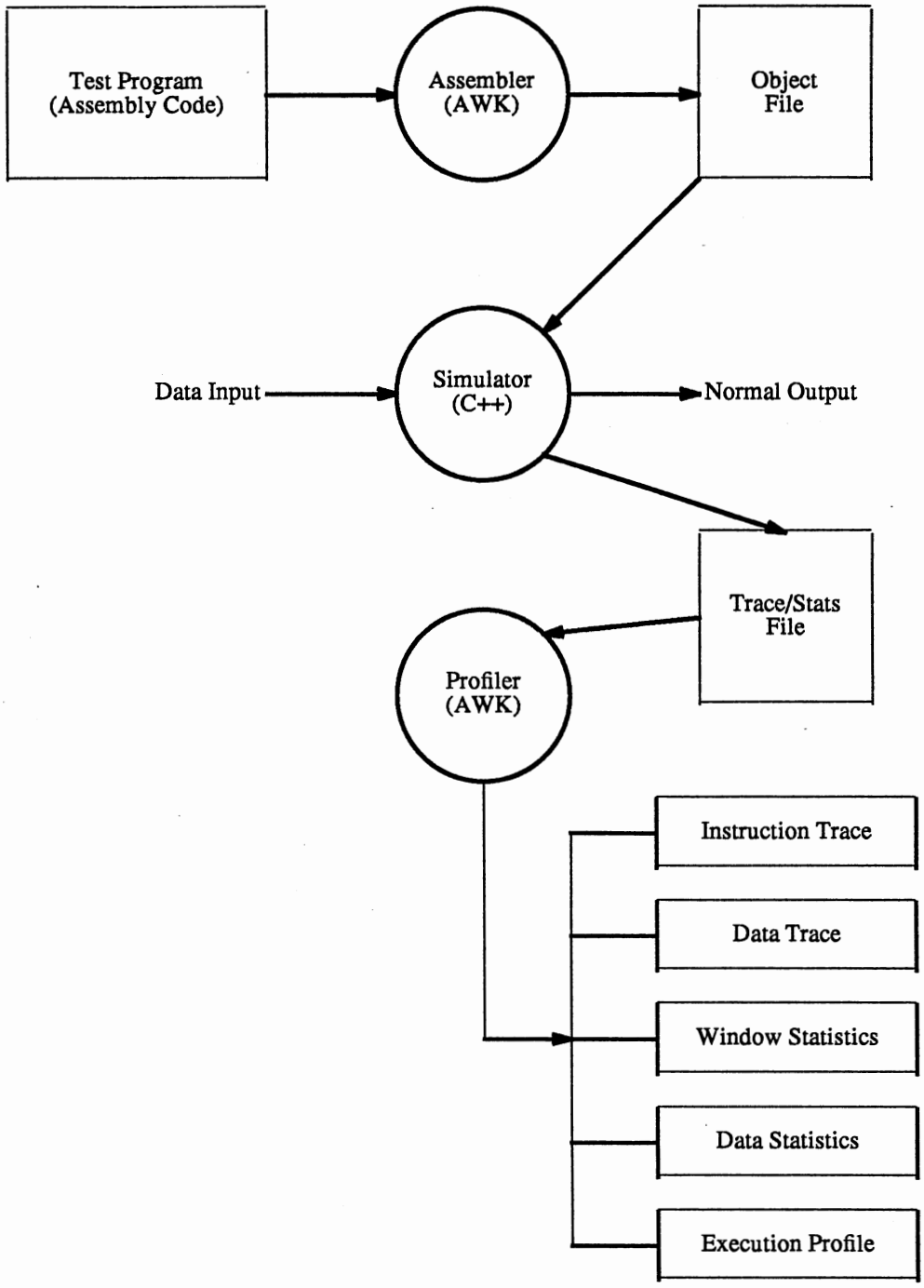


Figure 14. The Flowchart of the Simulation Project

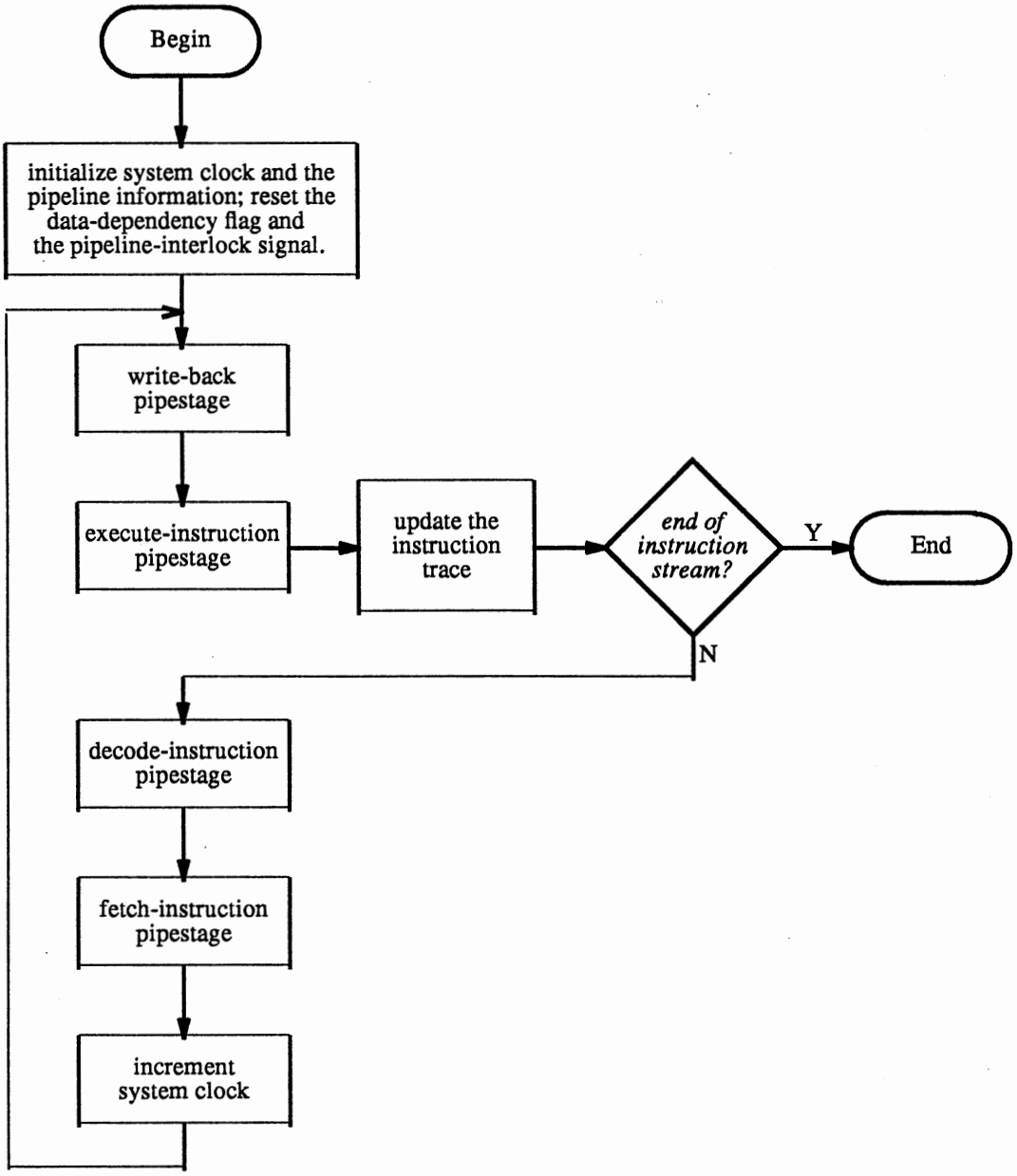


Figure 15. The Flowchart of the Instruction Pipelining

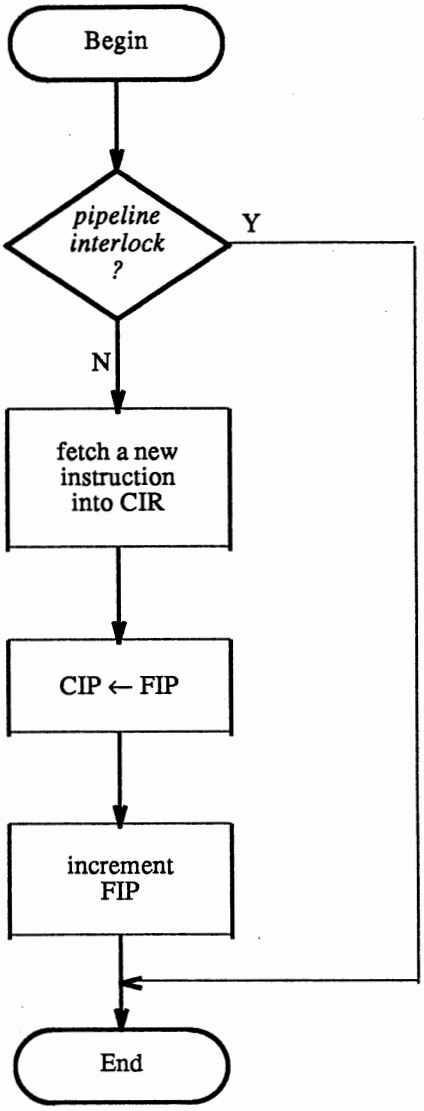


Figure 16. The Flowchart of the Fetch-Instruction Pipestage

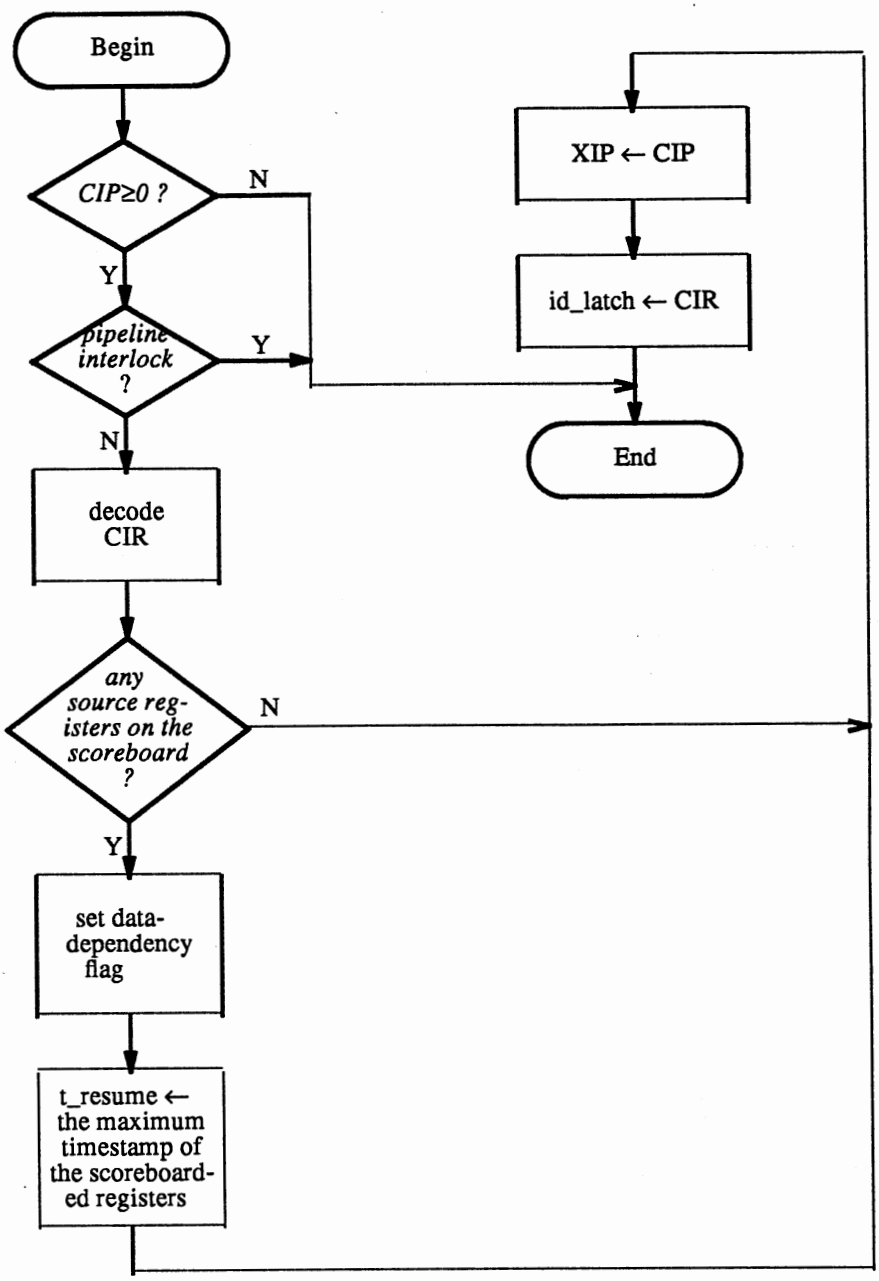


Figure 17. The Flowchart of the Decode-Instruction Pipestage

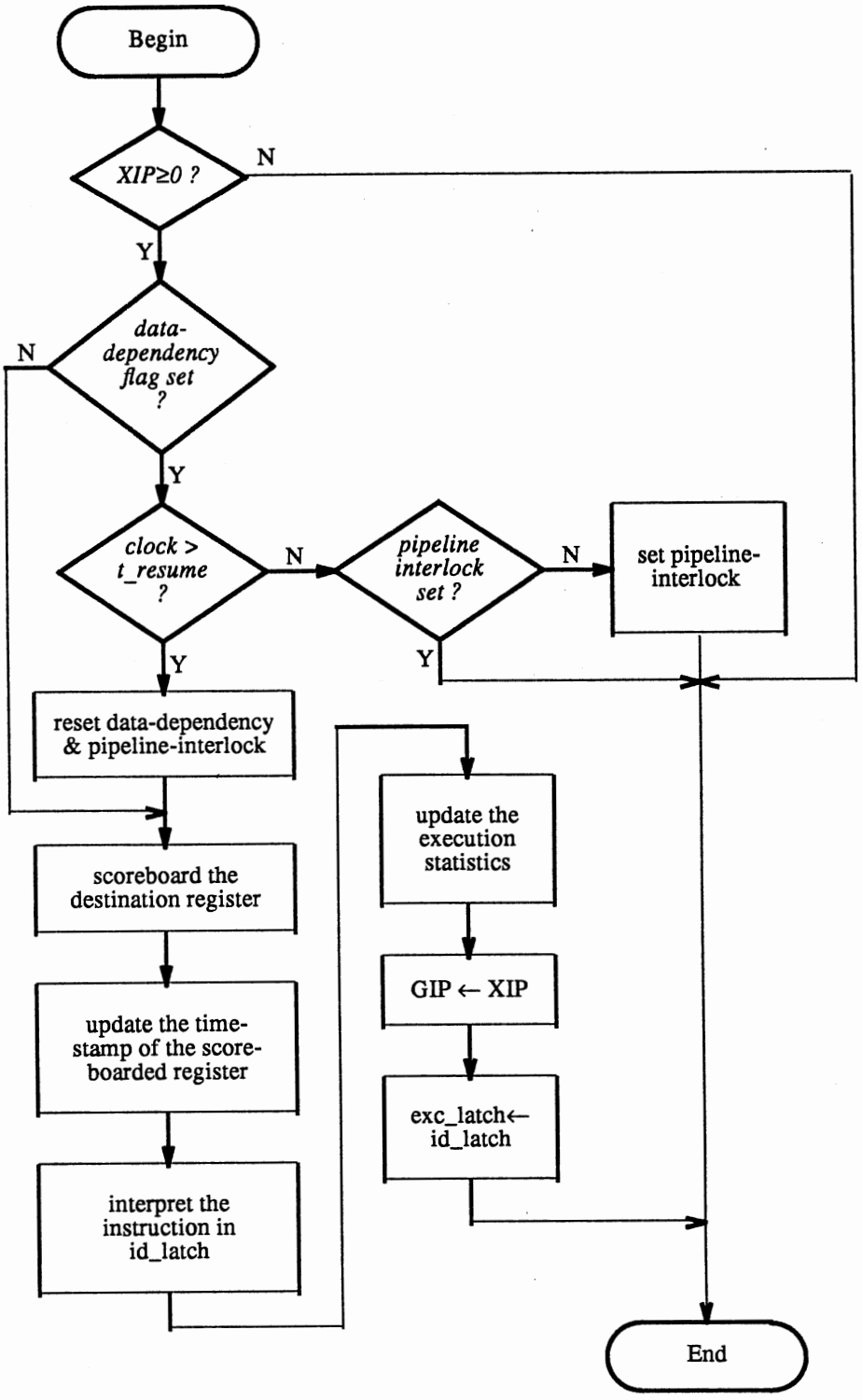


Figure 18. The Flowchart of the Execute-Instruction Pipestage

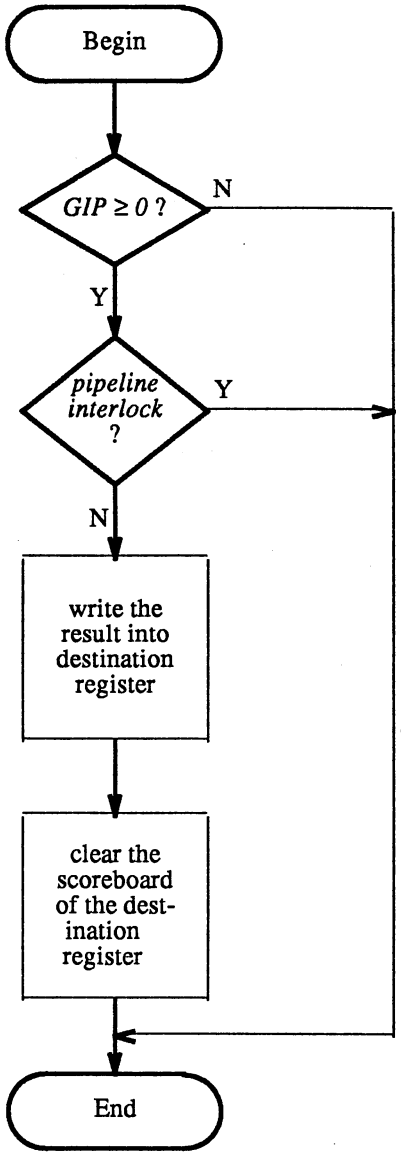


Figure 19. The Flowchart of the Write-Back Pipestage

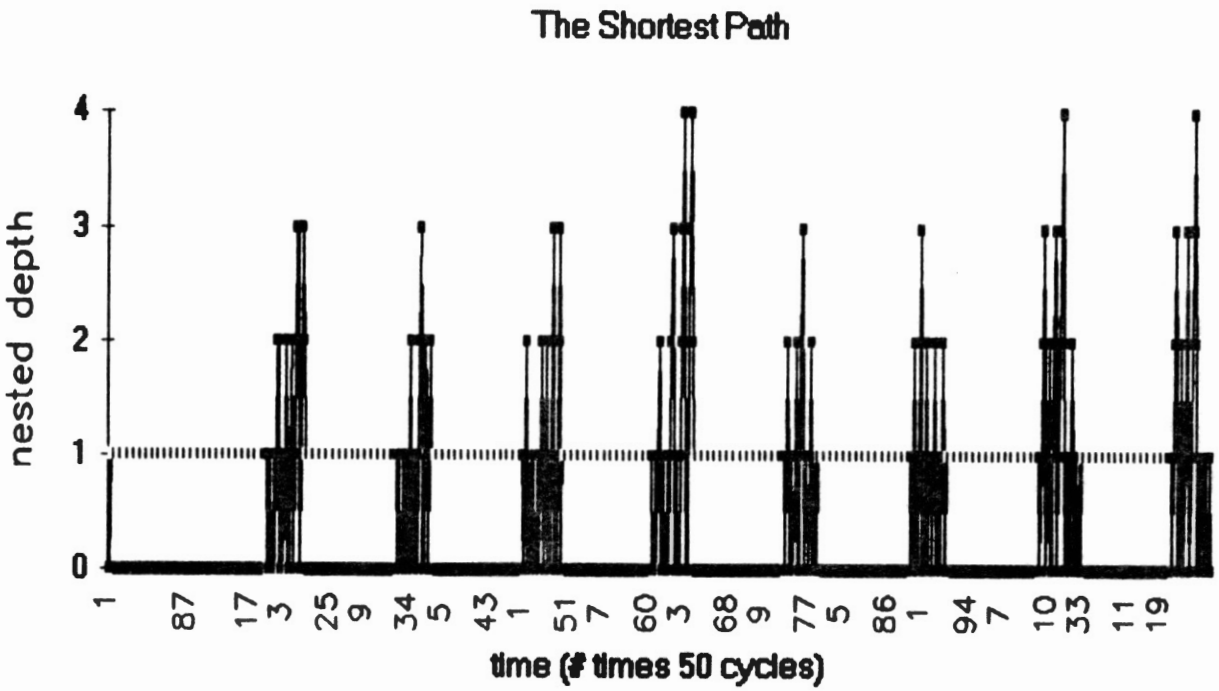


Figure 20. The Nested Procedure-Calling Depth During the Execution of The Shortest Path

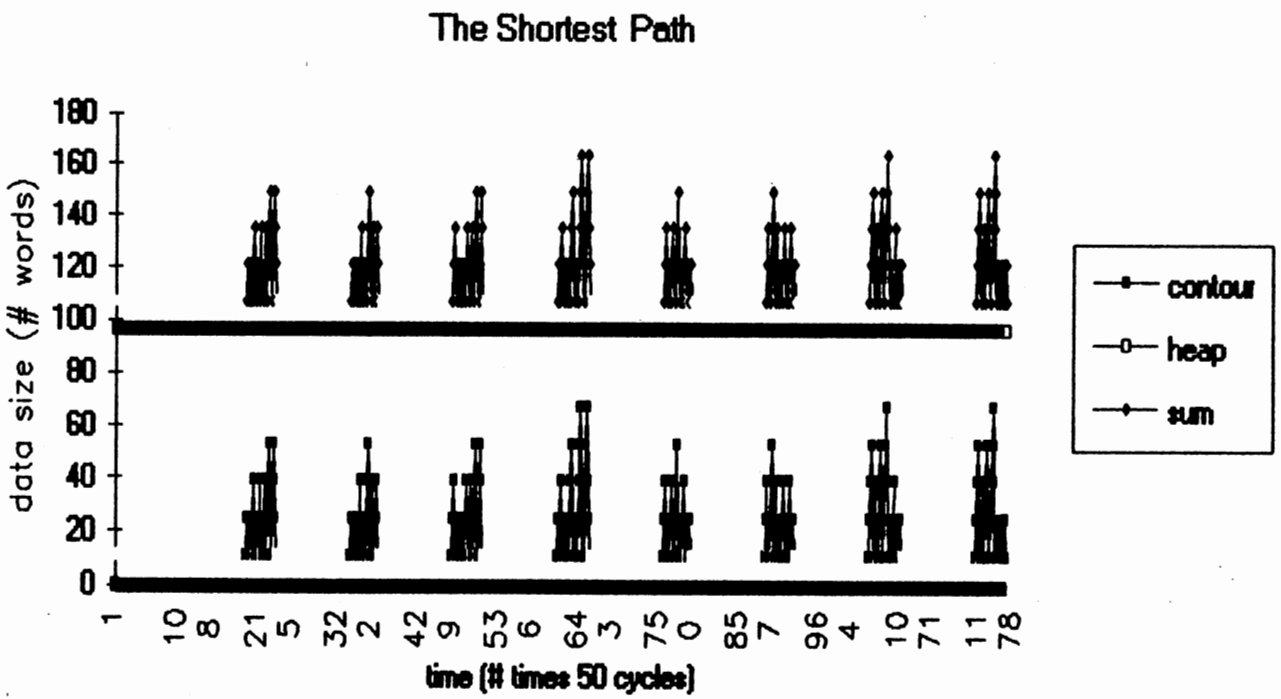


Figure 21. The Distribution of the Data Size During the Execution of The Shortest Path

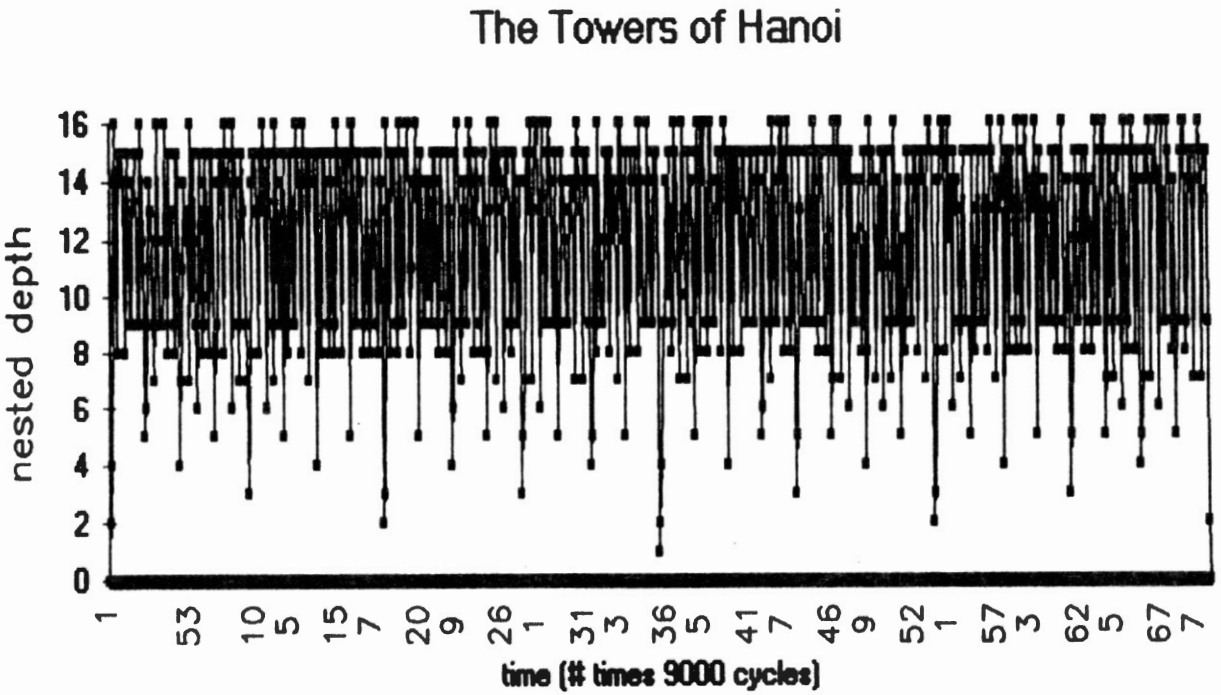


Figure 22. The Nested Procedure-Calling Depth During the Execution of The Towers of Hanoi

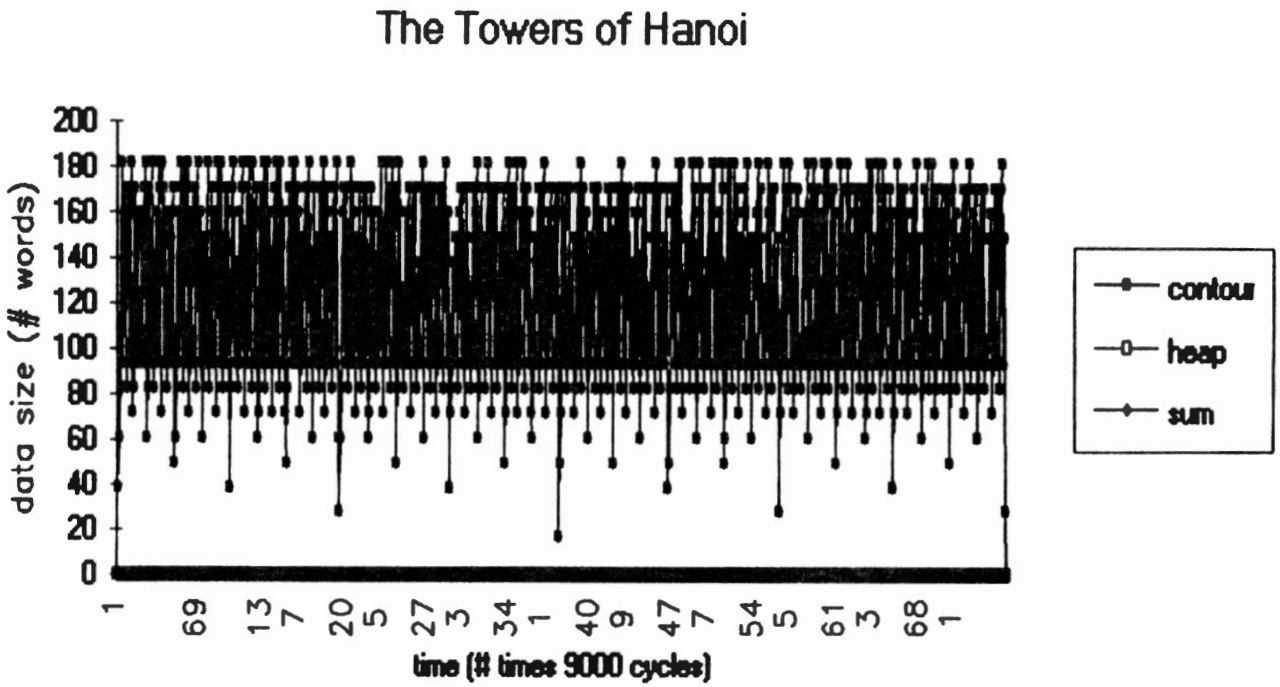


Figure 23. The Distribution of the Data Size During the Execution of The Towers of Hanoi

APPENDIX E

GLOSSARY OF TERMS

Cache Hit-Ratio: The probability for the cache to exactly contain the requested memory reference.

Data Dependency: An phenomenon in processing the data flow of sequential instructions which comprises the following events†:

(1) Read-After-Write Dependency — when a source operand of an instruction is the target operand which is overwritten by any instructions preceding it.

(2) Write-After-Read Dependency — when the target operand which an instruction overwrites is referenced by any instructions preceding it as a source operand.

(3) Write-After-Write Dependency — when the target operand which an instruction overwrites is also overwritten by other instruction(s).

Instruction Pipeline: A structure of partitioning the process of an instruction execution into multiple stages in order to exploit the instruction-level parallelism.

Memory Bandwidth: The average amount of information transferred from/to the memory per second.

On-Chip Storage: The storage which resides on the processor chip.

Pipeline Interlock: A ''hazard'' state of the instruction pipeline due to data dependencies.

Register Scoreboard: A hardware which contains flags indicating the availability of the registers. It is used to detect data dependencies relating to references to registers.

---

† Please see Section 3.3.4 of [HwBr84] for a formal description of the data dependency.

VITA

Hsu-Ku B. Ying

Candidate for the Degree of

Master of Science

Thesis: AN APPROACH TO APPLYING THE CONTOUR MODEL TO A HYPOTHETICAL MULTIPLE-REGISTER-WINDOW ARCHITECTURE FOR THE BLOCK-STRUCTURED PROCESS

Major Field: Computer Science

Biographical:

Personal Data: Born in Kaohsiung, Taiwan, October 15, 1960, the son of Ping-Han and Lan-Hsing Ying.

Education: Graduated from Tsai-Hsing High School, Taipei, Taiwan, in June 1978; received Bachelor of Science Degree in Civil Engineering from Chung-Yuan Christian University, Chungli, Taiwan, in June 1983; completed requirements for the Master of Science Degree in Computer Science at Oklahoma State University, Stillwater, Oklahoma, in May 1992.

Professional Experience: Reserve officer, the Survey Corps of the R. O. C. army, February 1984 to August 1985; civil engineer, Chien-Hwa Engineering Consultants, Inc., November 1985 to December 1986.