NOTICE WARNING CONCERNING COPYRIGHT RESTRICTIONS:
The copyright law of the United States (title 17, U.S. Code) governs the making of photocopies or other reproductions of copyrighted material. Any copying of this document without permission of its author may be prohibited by law.

CMU-CS-86-159

PAST: a Program-specific and Architecture-specific Simulation Tool

M. S. Project Report

Bruce S. Siegell

June 20, 1986

Department of Electrical and Computer Engineering
Carnegie-Mellon University
Pittsburgh, PA 15213

The research was supported in part by Defense Advanced Research Projects Agency (DOD), monitored by the Air Force Avionics Laboratory under Contract F33615-84-K-1520, and Naval Electronic Systems Command under Contract N00039-85-C-0134, and in part by the Office of Naval Research under Contracts N00014-80-C-0236, NR 048-659, and N00014-85-K-0152, NR SDRJ-007.

Table of Contents

UNIVERSITY LIBRARIES
CARNEGIE-MELLON UNIVERSITY
PITTSBURGH, PENNSYLVANIA 15213

List of Figures

List of Tables

Chapter 1 Introduction

There are several levels at which computers can be modelled: the circuit level models the computer in terms of its component transistors, resistors, and capacitors; at the logic level the computer is modelled as a set of blocks characterized by boolean functions; the programming level models the computer as the programmer sees it, as a set of registers, memories, and functional units whose interactions are controlled by a computer program. At all of these levels, tools are available to aid the user in simulation. At the lowest level[1], the circuit level, Berkeley's SPICE simulator [31] is the tool most used for circuit verification and providing timing information in our environment. At the next level, a large range of logic simulators are available to test the logical correctness of designs, ranging from Bryant's MOSSIMII [8], a switch-level simulator implemented in software for modeling MOS circuits, to commercially available design systems like Daisy's Logician system [10] which verifies the logic and timing of digital designs based on specifications for the components (chips). Logic simulators have also been implemented in hardware, e.g. the ZYCAD^{TM} Logic Evaluator [32]. At the programming level, there are many tools available based on several hardware description languages. Some of the better known tools are based on descendants of the ISP notation, a functional notation developed by Bell & Newell [3]. Others are based on languages which have both structural and behavioral modeling capabilities [22] and/or are designed for multi-level simulation [2, 17, 18]. Each simulation tool is developed and optimized for a particular purpose, be it architecture evaluation, timing analysis, or just to make sure a design is correct.

One of our objectives is to be able to simulate programs running on complex high performance systems. We wish to be able to run and debug programs in parallel with the development of new systems. For this purpose, we do not need to model the structure of the computer; we care only about the model of the computer seen by the programmer. Thus, we need programming level simulators that are optimized for running and debugging programs. To meet our needs, we have found it necessary to write custom simulators because the programming level simulation tool

[1]We use the hierarchy of simulation levels described by Selvaggi [29].

available to us, the ISPS simulator, is too slow. Although a simulation ratio of 10,000 to 1 for the DEC PDP-11 being simulated with the ISPS simulator [28] might be considered tolerable, a ratio of 200,000 for the CMU Warp processing cell, which can process 10,000,000 floating point operations per second, is not. An ad-hoc simulator for the Warp processor [9] has been written which has a simulation ratio of 6,000 running the same program[2].

We believe that a tool like the ISPS simulator could be useful for simulating programs for complex architectures like that of the Warp Machine if it meets or exceeds the following requirements:

- It can execute real programs, e.g. the binary code, for the desired computer architecture.
- It provides state information about the machine it is emulating.
- It is easier to write and modify than an ad-hoc simulator.
- Its speed approximates that of a well-written ad-hoc simulator.

The last of these features is the most difficult to implement for a general simulation tool because such a general tool must be efficient for a wide range of computer architectures. Our research has attempted to overcome this difficulty, while meeting the other requirements as well.

We have designed a simulation tool, called PAST which produces simulators that run programs for the described architecture significantly faster than does the ISPS simulator. We have based our design on some of the same premises on which Atlas [2] based his research: (1) minimizing level of detail provides a speedup over the ISPS simulator; (2) reducing monitoring hooks reduces simulation time; (3) operations done at compile time are less expensive than those done at actual simulation time. However, we challenge Atlas' assertion that a translator for a hardware description language can not easily attain the simulation efficiency of a general purpose programming language translator. Our tool acts as a compiler for the ISPS language, translating ISPS descriptions into C code that can be compiled into simulators. Our new simulation tool has been named PAST, *a Program-specific and Architecture-specific Simulation Tool*, because it optionally produces simulators that are specific to a program to be simulated as well as to the architecture described further taking advantage of the third premise.

Section 2 of this report describes some of the previous research into simulators at the

[2]These are gross approximations based on execution of the simulators on a Vax 8650 running Mach/4.3/2/1 BSD Unix without file I/O. The relative order of magnitude should be correct.

programming level. Section 3 outlines the simulation model used in our approach and describes the simulators produced by PAST. In Section 4 are discussed the major issues involved in the implementation of PAST. The experiments done to determine PAST's performance are described in Section 5, along with the presentation and analysis of the results of the experiments. Finally, the results are summarized, and conclusions are drawn from the data collected.

Chapter 2
Programming Level Simulation

A programming level simulator generally consists of a user interface and a stream of code which imitates the control flow and data operations of the machine being simulated. Ideally, the user sees the simulator as a black box which takes as inputs a program, data, and user commands and produces data and state information (Figure 2-1).

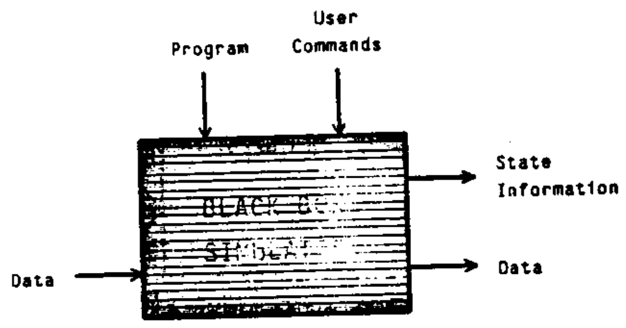

Figure 2-1: Black box model of a functional simulator.

However, there are several variations on this model, some requiring more information from the user and some less (Figure 2-2):

- The simulator can be made general enough to handle any computer architecture by making a description of the architecture an input (Figure 2-2a). We will call this type of simulator a *general* simulator. We also classify this simulator as *interpretive* because it interprets the description of the architecture.

- The architecture specifications can be contained within the simulator (Figure 2-2b). We will refer to this type of simulator as an *architecture-specific* simulator. In this case, the simulator is *compiled* rather than *interpretive* because the architecture is an intrinsic part of the simulator.

- Both the architecture and the program can be contained within the simulator program (Figure 2-2c). This implementation model has limited applications because the cost of preparing the simulator and running it on all of the data sets for the program must be less than the cost of running the program on its data sets using a more general simulator. We call this type of simulator *program-specific*.

- Even more extreme, all of the inputs including the user commands and data can be hardwired into the simulator program (Figure 2-2d). This model has the same limitations as the previous model, but it can only run on one set of data. There are cases where such a simulator might be useful, e.g. simulation of a random number generator, but there are too few cases to warrant a special tool to generate such simulators.

We can classify most functional simulators according to these four types.

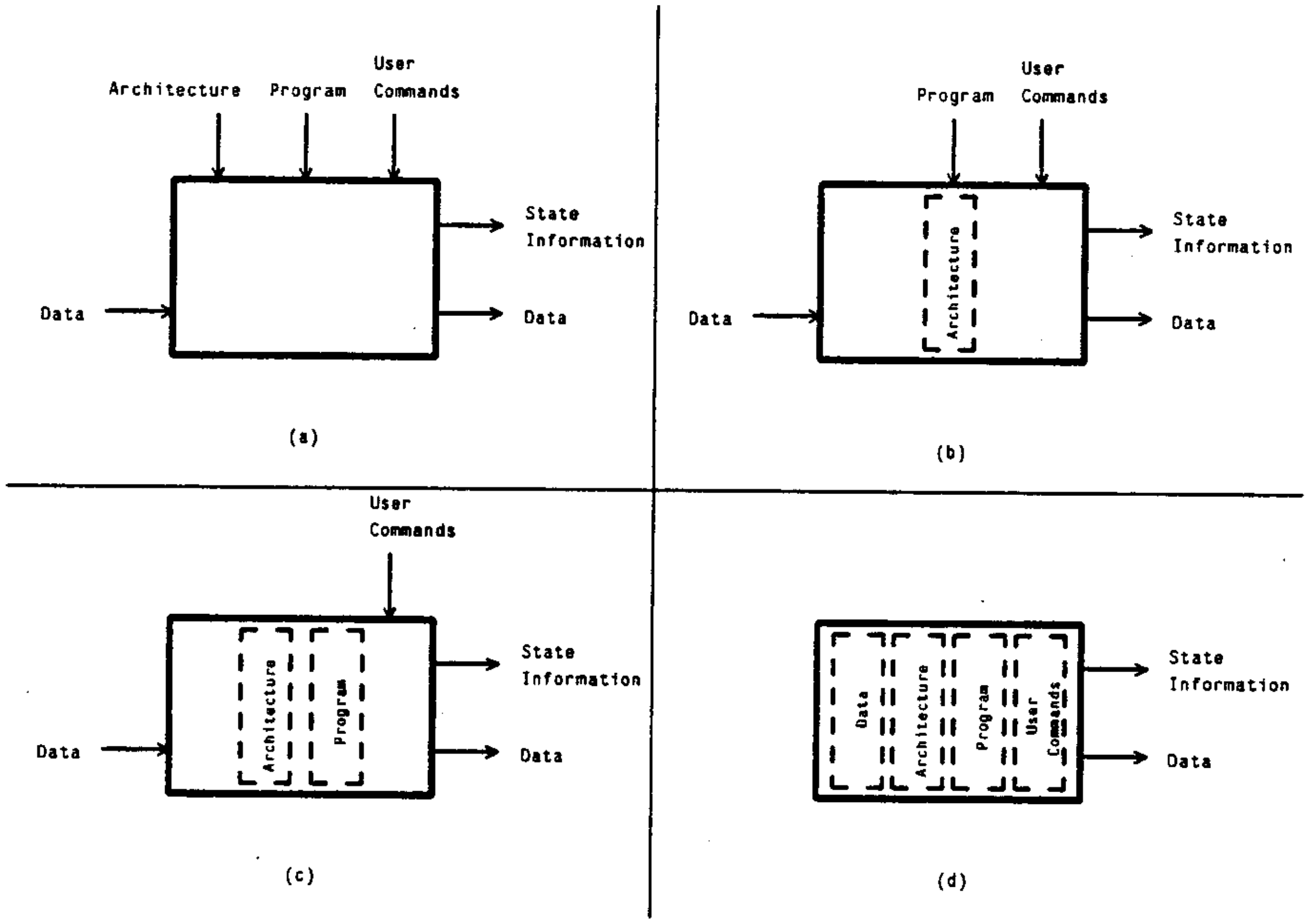

Figure 2-2: Simulator Implementation Models: (a) general; (b) architecture-specific; (c) program-specific; (d) completely specified.

In the following sections we summarize some of the research which has been done in the area of functional simulation.

2.1. The ISPS Simulator

Several versions of the ISPS simulator have been implemented at Carnegie-Mellon University. All are based on ISPS, a well-known hardware description language derived from the ISP notation introduced by Bell and Newell [7]. The original simulator was written in BLISS and runs on a DECsystem-10 [5]. This simulator has been translated into Pascal to run on Hewlett Packard 9836 workstations and most recently into C to run under Unix [28, 29]. The current work shares some code with the C version. All of the versions of the ISPS simulator fit our model of a *general* simulator as the description of the computer architecture is one of the inputs to the simulator program. Both the architecture and the programs are *interpreted* by the simulator.

The ISPS simulator does not interpret ISPS descriptions directly. The ISPS descriptions are first parsed and then converted into an intermediate code which is a set of instructions for the *Register Transfer Machine*, a hypothetical 3-address machine with variable length operands. The ISPS simulator is actually a software implementation of the *Register Transfer Machine*. Figure 2-3 shows the steps required to simulate an ISPS description on the C version of the ISPS simulator. The intermediate files shown are the GDB file, containing the parse tree information, and the RTM file containing the intermediate *Register Transfer Machine* code (see Chapter 3). The ISPS parser and the RTM code generator are described in more detail in Chapter 4.

The BLISS version of the simulator has been heavily optimized for use on the DEC-10, a 36-bit machine, replacing arithmetic and logical operations with host operations where possible, and, thus, is not portable. The HP9836 version is not portable and is reputed to be quite slow. The Unix ISPS simulator, adjusted for differences in host processor speeds, runs at a comparable speed to the BLISS version [28] and has a typical simulation ratio for a medium sized processor of 10000 to 1. The simulation ratio becomes much larger, however, for large or complicated processors (see Chapter 1).

2.2. N.mPc

N.mPc [27, 26], developed at Case Western Reserve University, is another tool which has been widely-used for functional level computer simulation. It is based on the ISP' language, also a descendent of the ISP notation. N.mPc is made up of five components which manipulate numerous files to build a sixth component, a runtime package consisting of a simulation program, a command interpreter, and a simulation memory manager (Figure 2-4). The original N.mPc system ran on a DEC PDP-11, was not portable, and had size limitations. A new more portable system called *N.2* [24, 27] resolves these problems and also includes an additional description language to specify

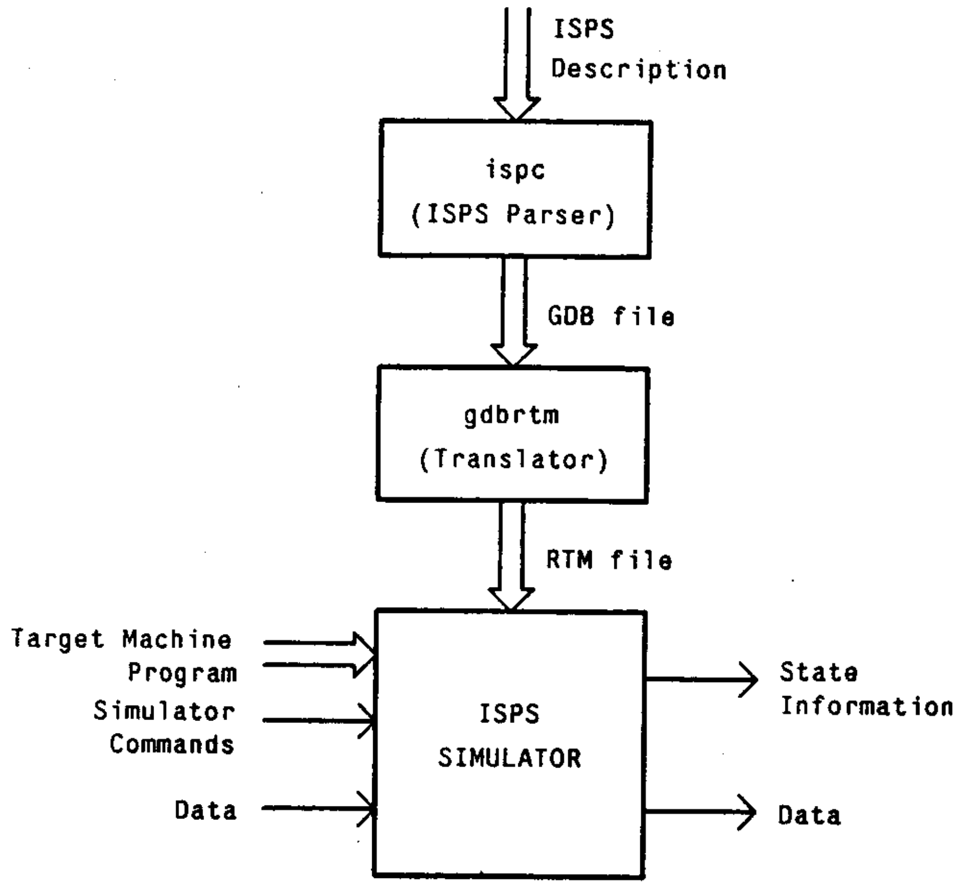

Figure 2-3: Block diagram of the C version of the ISPS simulator.

PLAs and a graphics interface for graphical specification and manipulation of topology files, display of monitoring information, etc. [27].

N.mPc offers greater capabilities for modeling structure than does the ISPS simulator, allowing a system to be specified as a collection of modules connected by ports. The ISP' language includes a WHEN statement to handle asynchronous processes and a DELAY statement which can be used to associate a delay with each register transfer for system timing.

Aside from the difference in language features, the major difference between N.mPc and the ISPS simulators is that N.mPc simulators are compiled rather than interpreted -- thus fitting into our *architecture-specific* model. However, heavy use of library functions, local states, and port communication mean that the compilation vs. interpretation tradeoff effects less than 10 percent of the execution time for the N.mPc simulators [27], thus making the simulator effectively interpretive. Because the compilation had so little effect, N.2, the successor to N.mPc is interpretive [27]. N.mPc gives simulation a ratio of approximately 1900 for the MC68000 using a programming level

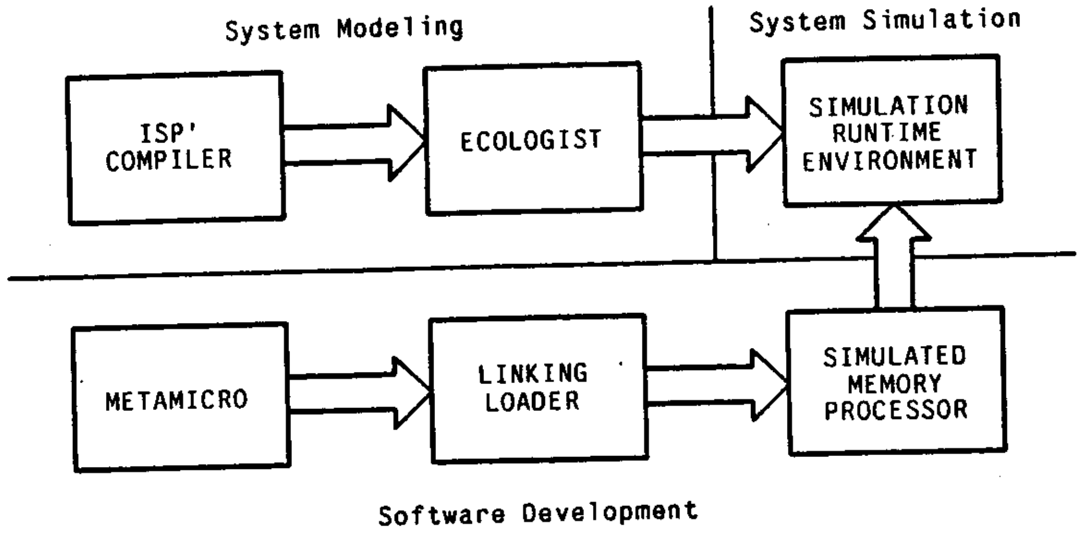

Figure 2-4: Simplified block diagram of N.mPc system [27].

model [15]. This is significantly better than the ratio for the ISPS simulator processing a description of a PDP-11, a simpler machine than the MC68000. We suspect that this better performance must be due to more efficient programming techniques and partially to use of compilation rather than interpretation.

2.3. The Register Transfer List Interpreter

Both ISPS and ISP' describe computer instruction sets in terms of their operation codes. Jack Davidson at the University of Virginia has designed a system called the *Register Transfer List Interpreter* (shown in figure 2-5) in which the instructions are specified by their mnemonic names. In his system, the machine description is a grammar which maps assembly language instructions to ISP register transfers. The language for the machine description appears to be a mix of ISPS and the BNF grammar notation. The *Machine Description Processor (MDP)* converts the machine description into C language subroutines that implement the described instructions. These routines are linked with several libraries of standard and custom routines: the MDSIM contains commonly needed routines such as 16 and 32 bit arithmetic operations; an I/O library provides the standard I/O routines; and a library provided by the user handles instructions which cannot be described using the description notation. The resulting simulation program, called the *Machine Description Interpreter* or *MDI*, reads an assembly language program and links each of the programs instructions to the appropriate C subroutines to handle the operation and operands. The simulation is then executed under control of the user as a sequence of subroutine calls. Also associated with the simulation system is a compiler system which can be used to generate the assembly language for the target machine.

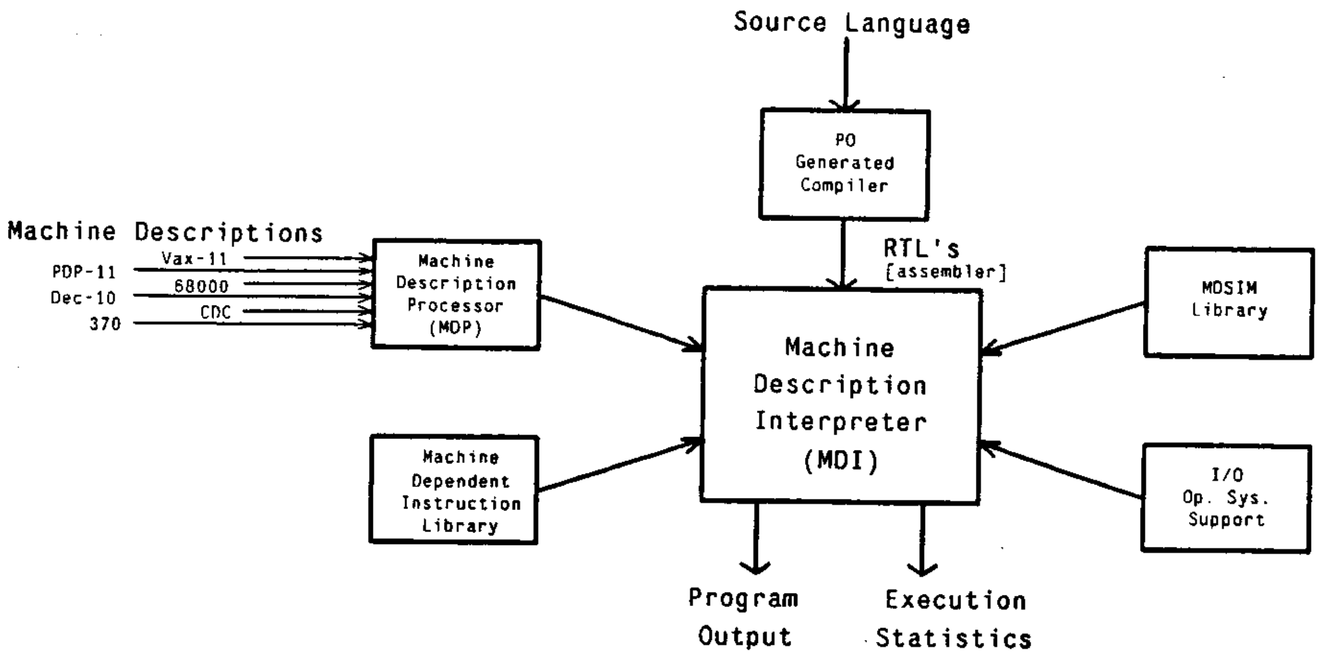

Figure 2-5: Block diagram of Register Transfer List Interpreter system [11].

The MDI is capable of simulating 500 Vax 11/780 instructions per second on a Vax 11/780. This is quite fast. Simulating the MC68000, the ISPS simulator only executed about 15 instructions per second[3]. However, the MDI simulator is limited by its description language. A microcoded machine such as the Warp cell would not be very easy to describe, and the user would probably have to write quite a few custom routines for the description to work.

2.4. The Value Trace Simulator

Thus far, there has been little research into the use of hardware to speed programming level simulation. At Carnegie-Mellon, a project has been proposed whereby a logic simulation engine such as the ZYCAD™ Logic Evaluator [32] can be used for simulation at the functional level [16]. A functional description in ISPS is first converted into a data flow graph called the Value Trace. The data flow graph is then converted into an equivalent gate network which can then be simulated using a logic simulation engine. The steps involved in simulation by this method are shown in figure 2-6.

The main advantage of this approach is that simulation is very fast: the simulation is event-driven, so only the operations which are needed are executed, and the use of hardware speeds the

[3]This figure was calculated from Schooley's data [28].

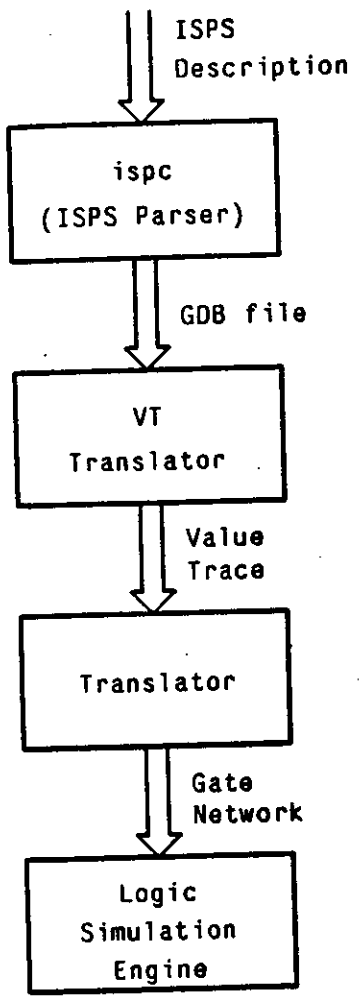

Figure 2-6: Block diagram of the Value Trace simulator [16].

evaluation of the operations. However, disadvantages of this approach include the high cost of simulation hardware and a long simulator preparation time. The long preparation time may mean that simulations must be run as batch jobs. This is all right for verifying an architecture design, but makes use of the tool for debugging programs impractical.

2.5. Summary

In this chapter, we have discussed the different programming level simulation models and have described some representative programming level simulators. Each simulation approach has advantages and disadvantages. The main tradeoff is between preparation cost and simulation cost. We need to minimize the time and user efforts involved in preparing simulators and minimize the time of simulation. We will refer back to some of the simulation techniques presented in this chapter when we describe our simulation philosophy in Chapter 3.

Chapter 3
The PAST Simulation Model

Our primary goal in the design of PAST is to be able to produce automatically functional simulators with speed comparable to that of ad-hoc simulators. The simulators should be produced from machine descriptions that are easy to write modify. Also, the simulators should have a user interface which allows the user control of the simulation and provides information about the state of the machine being simulated. In this chapter, we discuss the choices we made in designing PAST, emphasizing how we meet these goals. We attempt to predict the impact of our choices, by estimating the performance improvements of simulators produced by PAST over the performance of the ISPS simulator. We call the machine to be simulated the *target* machine and the machine on which the simulation is done the *host* machine. Our metric of performance for simulators for a target machine is the number of instructions for that machine which are simulated per unit time on the host computer. In the last section of this chapter we explain the structure of the simulators that PAST produces.

3.1. Hardware Description Language

A standard notation is needed to describe computer architectures so that a general simulation tool can process the description. Such notations are called *computer hardware description languages.* We have chosen to base PAST on the ISPS hardware description language.

For a simulation tool to run fast, it is important to choose a description language which models computers at the proper level. To execute computer programs, we just need a functional description of the architecture to be simulated. Structural information is not necessary or desirable because handling communication between structural elements is costly. In general, handling extra information has costs in either simulation time or simulator preparation time. ISPS is at the appropriate level for a simulator for program execution because it can model the register transfer operations which are required to execute programs without regard to the structure of the computer being simulated.

There are several other benefits to basing PAST on ISPS:

- ISPS is a well known and well accepted language.
- The ISPS simulator is available for performance comparison.
- ISPS descriptions have already been written for many machines.
- An ISPS parser already exists. We can use the intermediate code (RTM code) which it produces.
- The source code for the ISPS simulator is available. PAST can use much of same code.

One detriment of the ISPS language, however, is that it does not provide the user with efficient floating point operations. The user has to construct floating point routines from fixed point operations. The resulting routines are inefficient because they do not take advantages of the capabilities of the host machine. We decided that to be competitive with an ad-hoc simulator, in particular the Warp simulator, a PAST simulator must use efficient floating point routines. We have thus incorporated explicit floating point operations into the ISPS language. Our version of the language provides the same operations for floating point numbers as for the other number formats. However, these operations can only be applied to operands of the size of floating point numbers which are handled by the host machine. In our case, the host machine was a Vax, so we had the choice of 32, 64, or 128 bit floating point arithmetic [14]. We chose to base our floating point routines on the 32 bit *F*[4] type operations because the Warp machine supports 32 bit floating point arithmetic.

3.2. Implementation language

We have implemented the PAST program in the C programming language [20]. We chose C so that we could easily modify code from the C version of the ISPS simulator for use in PAST. Also, because our host systems run the Unix^{TM} operating system, C is the best-supported high level language in our environment.

For similar reasons, the code generated by PAST is in the C language also. The PAST library, a library of functions which are linked with the code generated by PAST, uses some of the same routines that PAST uses, so using a single language saved us from writing code for the same functions twice. Alternatively, we could have had PAST produce assembly language code which called the common C routines as needed. Although this would probably produce faster simulators, C is more

[4]The Vax architecture supports four floating point data types: *F* is a 32 bit data type, *D* and *G* are 64 bits, and *H* is 128 bits.

readable, easier to debug, and more portable than assembly language. Also, if we were to generate assembly language, we would have to worry about low-level compiler issues, such as register allocation and reentrancy. Instead we let the C compiler (*cc*) handle these issues (Figure 4-1). To improve the speed of our simulators, we could replace *cc* with a better C compiler.

3.3. PAST simulators are compiled

We designed PAST to produce compiled simulators because it is generally accepted that "the execution time of an interpreted program is usually slower than that of a corresponding compiled object program." [1] This is supported by performance comparisons of existing functional simulation tools. N.mPc simulators, which contain the description of computer architectures in compiled form, run several times as fast as the ISPS simulator which interprets the description of the architecture (see Chapter 2). Ad-hoc simulators, which contain the computer descriptions in a hand-compiled form, also perform much better than the ISPS simulator. PAST's simulators should perform at least as well as those produced by N.mPc because PAST simulators call library routines for only the most complicated operations while 90 percent of the N.mPc simulator execution time is spent in library routines. N.mPc also has additional overhead due to handling of structural definitions.

The ISPS simulator interprets both the description of an architecture and the code to be run on that architecture. In implementing PAST, we replaced the parts of the ISPS simulator which interpret the architecture description and program with routines that produce code to execute the architecture description. Thus we have converted the ISPS simulator from an interpreter to a compiler. The conversion of a general program into a more specialized one by use of known parameter values is known as *partial evaluation*. Research has been done on automatic partial evaluation [19], especially for converting programming language interpreters into compilers, but in the case of PAST, the partial evaluation was done by hand.

3.4. PAST's cycle is the target machine instruction cycle.

One of the most useful features of a functional simulator is the ability for the user to set breakpoints, so that he can stop the simulation at a particular location in his program and examine the state of the machine at that point. Breakpoints are typically set by specifying the value that the program counter has at the desired stopping location. The ISPS simulator defines breakpoints differently, requiring much more information from the user. In addition to specifying the program location, the user must specify the name of the program counter variable as specified in the ISPS description. The ISPS simulator stops as soon as the program counter variable is set to that value, whether the end of the instruction cycle has been reached or not. The user then has to refer to the ISPS description to determine where to set a breakpoint for the end of the instruction cycle so that he is not looking at intermediate states. The reason that ISPS makes setting breakpoints difficult is that it has no concept of an instruction cycle. Each of its cycles processes a single Register Transfer Machine instruction. The ISPS simulator steps through the RTM instructions until an RTM "STOP" operation or a user-specified stopping point is reached, with no regard for the instruction cycle of the target machine.

In PAST simulators the basic cycle is the instruction cycle for the target machine, and the RTM level is removed. Since a PAST simulator knows what instruction cycles are and knows the name of the program counter variable (assuming it was specified when creating the simulator), breakpoints can be set in the normal way, just be specifying a stopping location. The user only has to interact with the PAST simulator at the beginning of each target machine cycle so he does not need to keep track of as much information as with the ISPS simulator. Thus, it is easier to debug programs with PAST simulators than with the ISPS simulator.

Besides reducing the amount of user interaction required at run time, use of the instruction cycle as the cycle for PAST improves simulator performance. For each RTM instruction, the ISPS simulator executes at least ten C statements in addition to the statements necessary to execute the RTM operation. PAST's simulators have about the same overhead for each loop iteration, but each iteration represents several to hundreds of RTM operations. The smallest real architecture description we have encountered, for the Manchester Mark-1 computer, averaged about 13 RTM operations per instruction cycle. The ISPS description for the Warp cell, which is an architecture of only medium complexity, averaged over 350 RTM operations per simulated instruction. The loop overhead for PAST simulators becomes insignificant when compared to the overhead eliminated by changing from RTM cycles to target machine instruction cycles. On average, PAST only generates one or two C statements per RTM operation, but each of the generated statements is probably about two or three times as complicated as each of the loop overhead instructions eliminated by changing to target machine instruction cycles. A quick calculation shows that we can expect more than a 50 percent (we're being conservative here) reduction in simulation time just due to eliminating the overhead of using the RTM instruction cycle:

RTM processing cost:
  2 generated statements * 3 simple instructions/generated statement
    => 6 simple instructions
Total cost per RTM instruction:
    16 simple instructions
Overhead reduction per RTM instruction:
    10 simple instructions

Reduced cost per RTM instruction:
    6 simple instructions
Percent reduction:
    10/16 * 100% = 62.5%

It should be noted, at this point, that the time reductions that we speak of are all at simulator run time. The reductions at run time are paid for by an increase in simulator preparation time. When PAST generates a simulator, it still interprets each RTM operation to translate the instruction to C. It begins interpreting with the RTM instruction that corresponds to the ISPS instruction which the user has specified as the beginning of target machine instruction cycle. PAST follows the flow of control specified by the RTM instructions until it reaches the starting instruction again or an instruction which corresponds to the ISPS instruction specified by the user as the end of the target machine instruction cycle. The time required by PAST to interpret an RTM instruction and generate C code for that instruction may far exceed the time required by ISPS to interpret and execute the same instruction. The higher cost is acceptable, however, because the simulator preparation time for a PAST simulator is a one-time cost. PAST only needs to interpret and generate code for each instruction once: when it generates the simulator. ISPS has to interpret the same RTM instructions over and over again, for for each iteration of the target machine's instruction cycle for every program it simulates.

3.5. PAST reduces costs of Register Transfer operations.

In the preceding calculations we assumed that the cost of processing an RTM operation at simulator run time – 1 or 2 C statements – is the same for both the ISPS simulator and PAST. That is not actually the case. The run-time cost of processing RTM operations is much less for PAST-generated simulators than it is for ISPS.

Most RTM operations can be categorized into four groups [2]: control operations, data operations, arithmetic operations, and logic and shift operations. The remaining operations, masking operations, are seldom used. Previous research [2, 28] has found that 65% of executed RTM operations are control operations, 20 to 25% are data operations, and the remaining 10 to 15% are arithmetic and logic/shift operations.

3.5.1. Control Operations

Since PAST interprets the RTM code to generate C code, it can handle many of the control operations at translation time and eliminate them completely from the generated code. Several such control operations are involved in calling subroutines: CALL, PEND, PBEGIN, LOCK. An analysis by Atlas [2], found these four operations to account for 35% of the RTM instructions executed[5]. Schooley [28] came up with similar data. Schooley's data also showed three of these operations to take 8.3 percent of the simulation time using the C version of the ISPS simulator. Data for the fourth operation was not given, but we can estimate that the total percentage of simulation time taken by the four operations is about 10%.

The ISPS simulator gives the user the option of simulating parallel constructs in parallel, using a round-robin scheduling algorithm for the concurrently executing streams of RTM code, or serially by executing the streams of code one after another. The serialization option, which disables several RTM operations, is provided because the round-robin scheduling is very time consuming. Because PAST's cycle is the target machine instruction cycle, the user can never examine the streams running in parallel, so PAST always executes RTM instructions serially. (Running streams in parallel could alter the state of the machine at the end of the cycle if the target machine as described is capable of producing indeterminate results. We consider such a design a violation of good design practices.) Even if PAST were to use the round-robin scheduling to model parallel execution, there would be no cost for the scheduling during simulator run-time; some of the operations would just be reordered. Thus, PAST completely eliminates the RTM control operations which model parallelism when it translates them into C code. The run-time savings due to the elimination of these operations by PAST is minor because the serialized ISPS simulator also eliminates them.

Two other control operations which PAST does not translate into C code are the SMERGE and SJOIN operations. The SJOIN operation is a jump from the end of a conditional section of code to the point where the conditional sections rejoin, an SMERGE statement. Schooley's data show these operations to account for about 4 percent of the simulation time using the ISPS simulator.

We expect a 14 percent reduction in simulation time compared with the ISPS simulator due to elimination of unnecessary control operations from the simulators produced by PAST.

[5]This was the total for test runs of 5 different ISPS descriptions: AM2910, AM2901, I8080, PDP11, CDC6600.

3.5.2. Data Operations

Most of the data operations are transfer operations which move data from one register or memory into another. Also included in this group are several unary operations: clear, increment, decrement, etc. Schooley found two of these operations, RBYTE[6] and MOVE, to take more than 41 percent of the ISPS simulation time. Although Schooley made no measurements for the rest of the data operations, we estimate that the data operations account for more than 50% of the execution time of the ISPS simulator.

The handling of data operations probably has the greatest potential for improvement in simulation time. Because the ISPS simulator does not allocate its simulated memory on word boundaries, the ISPS simulator must access memories and registers one bit at a time. PAST does allocate memory on word boundaries so its memory accesses are simple memory transfers. Bit fields are accessed by masking and shifting operations. Since most modern computers have word sizes of 32 bits or less, we can estimate a typical memory access to be 16 bits. A PAST simulator can probably access a 16 bit memory 16 times as fast as the ISPS simulator. With such great reduction in memory access times, we estimate 40 or more percent reduction in simulation time compared to ISPS simulators due to more efficient data operations.

3.5.3. Arithmetic and Logic/Shift Operations

Because arithmetic, logic, and shift operations require transfers of operands, we can expect the same kinds of speedup for these operations as for data operations. We expect some additional speed improvements as well because all of the ISPS arithmetic, logic, and shift operations work on 128-bit data even if the operands are less than 128 bits. For logic operations, the operations are performed independently on each of the four 32-bit words required to hold the 128 bits. Arithmetic and shift operations also require data to be passed between the words making up the 128-bit operands. Thus, with the ISPS simulator, a 32-bit logic operation takes 4 times as much time as necessary, and arithmetic and shift operations take even longer. PAST uses the information known about operand lengths to minimize the number of host operations needed to produce a result. Since most computers operate on words of size 32 bits or less, PAST simulators probably perform most arithmetic, logic, and shift operations four times as fast as the ISPS simulator. We do not know what percentage of the ISPS run time is accounted for by these operations.

[6]RBYTE reads a bit field from a register.

3.5.4. Summary of expected improvement over ISPS

Optimization		% of total ISPS time remaining
none		100%
RTM Operations	**% of total ISPS time reduced**	
Data	40%	60%
Control	14%	46%
Target cycle change	**50% reduction** 50% * 46% = 23%	23%
Remaining time		23%

Table 3-1: Predicted speedup of PAST simulators over the ISPS simulator.

In Table 3-1 we have summarized the speedups that PAST should have over the ISPS simulator. We have estimated that PAST would reduce the total simulation time by more than 54 percent by optimizing the register transfer operations (14% for control operations, 40+% for data operations). In addition, using the target machine instruction cycle rather than the RTM cycle as PAST's cycle reduces the total remaining execution time by 50 percent or more. These speedups apply to both architecture-specific and program-specific simulators produced by PAST and are very conservative. The numbers only include the effects of some of the RTM statement optimizations, and we also do not know how much elimination of monitoring effects the simulation time; these factors should weigh in PAST's favor, but we can not quantize their effects. From our numerical data, we predict that simulators generated by PAST will run at least four times as fast as the ISPS simulator, but we suspect that the results could be even better.

We expect that the *program-specific* simulators will run faster than the *architecture-specific* simulators, but we do not have data from which we can make numerical predictions.

3.6. *Architecture-specific* and *Program-specific* simulators

PAST can produce either *architecture-specific* or *program-specific* simulators. Our performance estimates in the previous sections apply to both types of simulators. Our original goal in designing PAST was to produce *program-specific* simulators so that we could determine whether *program-specific* simulators show significant speed improvements over *architecture-specific* and *general* simulators[7]. We expect that *program-specific* simulators are be even faster than

[7]This is why we called our tool *PAST*

architecture-specific simulators because, in the *program-specific* case, many evaluations are done at code generation time rather than at simulation time. When we started to design the PAST program, we did not intend to produce *architecture-specific* simulators at all, but, after some design, we realized that it would take very little additional work to produce *architecture-specific* simulators as well. We added this capability to PAST so that PAST would be more general, and, also, so that we would have *architecture-specific* simulators that we could compare with the *program-specific* simulators.

The only information that PAST needs to produce an *architecture-specific* simulator, other than the ISPS description (in the form of an *RTM* file.), are the names of the ISPS entities which mark the beginning and end of the target machine's instruction cycle. (Though not essential, the name of the program counter variable should also be supplied so that breakpoints can be set.) Additional information is needed to produce a *program-specific* simulator:

- A program must be specified. The user specifies the program to PAST in the same way as to the ISPS simulator, by setting the appropriate memory locations in the simulated program memory.

- The memory range holding the program must be declared as STATIC. Otherwise PAST will not know which memory locations to treat as code and which to treat as data. When a register or memory location has been declared STATIC, PAST can replace code that accesses the memory location with the contents of the location. This replacement is what makes the generated simulator *program-specific*.

- The name of the program counter variable must be given.

- The range of valid values for the program counter must be specified. PAST loops through the list of program counter values and generates code to simulate *each* instruction of the program in much the same way it generates an entire *architecture-specific* simulator. The main difference is that before generating an instruction, the value of the program counter variable is set to one of the valid values and is marked as **known**.

Figures 3-1 and 3-2 show representative command files for generating *architecture-specific* and *program-specific* simulators, respectively[8].

[8] Lines beginning with exclamation points are comment lines.

```
! Name of ISPS entity which marks the start
! of the instruction cycle is "cstart".  The
! name of an entity marking the end of the cycle
! does not have to be specified if the cycle
! ends at the same location that it starts.
START cstart

! Name of program counter variable is "cr". This
! declaration is not necessary to generate an
! architecture-specific simulator, but for breakpoints
! to be allowed, the simulator must know which
! variable to compare the breakpoint list to.
PC cr
```

Figure 3-1: PAST command file for generating an *architecture-specific* simulator.

3.7. Standard User Interface

One of the most tedious tasks required in writing a simulator is producing a user interface. We have eliminated this problem by providing a library of standard user interface routines which are linked with the code produced by PAST to create a complete simulator. The main user interface routine, *UserInterface*, is invoked once per target machine instruction cycle. PAST inserts the call to *UserInterface* at the beginning of the cycle. This is the only call that the code generated by PAST makes to the user interface routines, but PAST does produce several routines which are called by the user interface, including the main simulation loop. The *main* routine for the simulator program is contained in the library, so the initialization process for the simulator is standardized.

The command set for PAST simulators is based on the command set for the ISPS simulator as described by Barbacci, et al. [5]. Most of the command names are identical to names of ISPS commands, but some of the commands have different meanings. A list of the currently available commands is provided below:

```
BREAK            ICONNECT        READ
! <comment>      NAMES           RUN
CYCLE            OCONNECT        SETVALUE
DBREAK           PROMPT          STEP
DUMP             QUIT            VALUE
ECHO             RADIX
```

Full descriptions of the commands are given in the *pastsim* manual entry in Appendix B.

```
! Name of ISPS entity which marks the start
! of the instruction cycle is "cstart".
START cstart

! Name of ISPS entity which marks the end
! of the instruction cycle is "cend".
END cend

! Set the memory locations to the program values.
! The quotation mark declares the number
! following it to be in hexadecimal notation.
SETVAL m["00] = "4026
SETVAL m["01] = "6023
SETVAL m["02] = "c000
            .
            .
            .
SETVAL m["19] = "4027
SETVAL m["1a] = "6027
SETVAL m["1b] = "e000

! set data locations to initial values.
SETVAL m["1c] = "00000004
SETVAL m["1d] = "00000007
SETVAL m["1e] = "0000000c
SETVAL m["1f] = "00000015
SETVAL m["20] = "0000001a
SETVAL m["21] = "00000001
SETVAL m["22] = "00000000
SETVAL m["23] = "00000000
SETVAL m["24] = "00000000
SETVAL m["25] = "00000135
SETVAL m["26] = "00000007
SETVAL m["27] = "00000000

! Name of program counter variable is "cr".
PC cr

! Initial value of program counter variable is 0.
SETVAL cr = 0

! Range of valid program counter values -> code range.
CODE "0:"1b

! Range of memory locations which do not change
! throughout program execution -> static range.
! Note that locations other than the program memory
! range can be declared as static.  This extra
! knowledge allows PAST to do more evaluations at
! translation time.
STATIC m["0:"21]
```

Figure 3-2: PAST command file for generating a *program-specific* simulator.

3.8. Standard Input/Output Routines

Also provided in the PAST library are a set of input and output routines to support the ICONNECT and OCONNECT commands. The ICONNECT command links variables to input files, and the OCONNECT command links variables to output files. Whenever an ICONNECTed variable is one of the source operands for an RTM operation, PAST generates a call to a macro which calls the appropriate input routine. Similarly, when an OCONNECTed variable is the destination for an RTM operation, PAST generates a call to a macro which calls an output routine.

The input and output routines in the PAST library are based on the Unix^{TM} standard I/O routines, but have additional code to convert numbers to the proper number format. We have also provided the option of connecting variables to BSD Unix 4.2 interprocess communication ports in the same way that variables are connected to files. We create FILE structures[9] for the IPC ports which are compatible with the structures created by the *fopen* command so that the IPC ports may be accessed using the *fprintf* and *fscanf* routines. The routines involved with interprocess communication have not yet been debugged[10], but the groundwork has been completed.

3.9. The structure of the PAST simulator

Appendix C includes samples of *architecture-specific* and *program-specific* simulators generated by PAST for a description of the Manchester Mark-1 computer. The simulators are divided into sections of code by comment headers. Below we explain the contents of each of the sections of code.

3.9.1. The program header

The program header is a comment block containing information about the simulator. It lists the names of the RTM file and the PAST command file from which the simulator was produced. Also given is the time that the simulator was produced. A **#include** line includes a file containing definitions and declarations which are common to all PAST simulators.

[9] These are the structures which are pointed to by file pointers, e.g. `FILE *fp;`.

[10] We are having difficulties with the *read* routine. It should block while waiting for input, but it does not.

3.9.2. Update and Propagate Macros

In the ISPS description, the user may define variables to be mapped onto other variables as shown in Figure 3-3. We call a variable which other variables map to a (local) *main* variable[11]. The variables which map to it are called *primary* variables. In C, the primary variables must also be mapped to the main variables. The next two sections in the simulator file define macros which do this mapping.

(a)

```
pi<15:0>,
        f<0:2> := pi<15:13>,
        s<0:12> := pi<12:0>,
```

(b)

```
#define _SET_f \
 f = (pi & 0x0000e000) >> 13;
#define _SET_s \
 s = pi & 0x00001fff;
```

(c)

```
#define _PROP_f \
 pi = (pi & 0xffff1fff) | (f << 13);
#define _PROP_s \
 pi = (pi & 0xffffe000) | s;
```

Figure 3-3: Example of mapped variables: (a) ISPS mapping; (b) update macros; (c) propagate macros.

The first of these sections contains *update* macros which are called to extract fields from main variables to update primary variables. The other section contains a complementary set of macros which we call *propagate* macros. A propagate macro is called to move the value of a primary variable into the appropriate field of its main variable after the contents of the primary variable have been changed. The macros are used to keep the main and primary variables up to date with respect to each other. PAST generates one update macro and one propagate macro for each primary variable. Main variables may have any number of macros associated with them.

The update and propagate macros are also used to implement ICONNECT and OCONNECT. If a variable is ICONNECTed, an update macro is created for it containing an input routine. If a variable is OCONNECTed, a propagate macro with an output routine is created. If an ICONNECTed or OCONNECTed variable is a primary, the input or output routine is added to the macro for updating or propagating the primary.

[11]"local" is given in parentheses because the variable to which the other variables map may also map to another variable. True *main* variables do not map to other variables.

Names of update macros begin with "_SET_" and end with the name of one of the variables declared in the ISPS description. Propagate macros are similar, but start with "_PROP_" instead. If the named variable is a memory, the macro has an address parameter.

3.9.3. Global Variables

The next three sections contain global variable declarations. The "PAST global variables" section declares the variables which are used as scratch registers in arithmetic, logical, and shift operations. The "User global variables - main" section contains declarations for the main variables declared in the ISPS description. The "User global variables - primary" section contains declarations for the primary variables.

Register variables which fit into a single integer word are declared as *unsigned ints*. Register variables with word sizes of greater than 32 bits and memory variables with word sizes of less than 32 bits are declared as arrays of *unsigned ints*. Memory variables with large word sizes are declared as two dimensional arrays.

3.9.4. Longjump Labels

The seventh section of code contains declarations for special global variables which store the stack state and program counter value for a location which is to be the destination of a long jump. Long jumps are *gotos* which can jump to any memory location which has been previously executed, even if the location is outside of the routine from which the jump was made. The destination of a long jump must be explicitly marked with a call to the *setjmp* routine, which sets the value of a corresponding longjump label. We have put the call to *setjmp* inside of a macro called LABEL.

3.9.5. The variable information table

The eighth section contains a table, called *_varinfo[]*, which lists information about each of the user variables, the variables declared in the ISPS description, so that the user interface routines can provide the user access to them. The following information is given for each variable: the name of the variable, a pointer to the memory location where the memory is stored, the size of the variable in integer words, the size of the variable in bits, the minimum and maximum valid addresses for the variable if the variable is a memory, the address increment for a memory variable, the number of the port to which the variable is ICONNECTED (if any), the number of the port to which the variable is OCONNECTED (if any), and the default number format to be used when reading and writing the variable.

This section also provides (for the user interface) the total number of variables accessible to the user, the number of the program counter variable, and the address of the program counter variable if it is a location in a memory.

3.9.6. Variable initialization

The next section contains a routine called *preset* which sets variables to the initial values specified in the PAST command file (see Figure 3-2). This routine is called by the user interface upon initialization of the simulator program and when the user gives the RESET command.

3.9.7. Channel initialization

The tenth section contains code to initialize the ports for ICONNECTed and OCONNECTed variables. The initialization may be opening a file or creating an interprocess communication connection.

3.9.8. Update and Propagate Routines

These routines give the user interface routines access to the update and propagate macros since the macros are local to the simulator file. The user interface specifies the macros by variable number.

3.9.9. The main simulation loop

The last section which is common to all of the simulators that PAST generates is the *MainLoop* routine which contains the main simulation loop of the simulator. It is this routine which contains the stream of code which models the execution of the target machine.

Because *MainLoop* must loop, the routine begins with labels that can be jumped to at the end of a cycle. Two types of labels are given: *StartLabel* is a normal C label which is jumped to by a *goto* statement at the end of the *MainLoop* routine; a longjump label is also provided (by the macro *LABEL*) so that the head of the loop can be jumped to from anywhere in the simulator program – even from outside of the *MainLoop* routine.

The first statement within the simulation loop is a call to the user interface routines. The rest of the code in the routine is dependent on whether the simulator is *architecture-specific* or *program-specific*, on the size of the description, and on user specifications.

3.9.9.1. Architecture-specific Simulators

The basic form of an *architecture-specific* simulator is a direct translation of the RTM code into C code, fully expanding all subroutines. But if the description is too big, the C compiler can not handle the long jumps which may be required[12]. We have found that much of the cause of the large amount of code being generated for *architecture-specific* simulators is the expansion of ISPS procedures in-line. We allow the user to specify which procedures are expanded[13]. If the number of calls made to a procedure in the ISPS description exceeds the amount specified by the user as allowable for in-line expansion, the procedure is made into a subroutine and the in-line expansions for that procedure are made subroutine calls instead. We have created an additional section for these subroutine definitions, labeled *Subroutines.*

3.9.9.2. Program-specific Simulators

For the *program-specific* simulators, the rest of the *MainLoop* routine is just a call to another routine called *MainSwitch.* The *MainSwitch* routine is a *switch* statement which switches on the value of the program counter variable. Each *case* of the *switch* statement represents an instruction for the target machine: it is the code generated by PAST by looping through the RTM statement table for one instruction cycle with the program counter initially set to the value of that *case.* A *switch* statement is not used directly in the *MainLoop* routine because the statement often becomes too large for the *goto* at the end of the simulation loop to jump over. Again because of the jump distance limitations, the *MainSwitch switch* statement is sometimes replaced by a statement which switches on the high bits of the program counter. The *cases* for this statement call other routines, which we call *subswitch* routines, that switch on the actual program counter value[14]. The *subswitch* routines, like the *subroutines* for the *architecture-specific* simulators are put in their own section.

[12]We have summarized our problems with the C compiler in Appendix D.

[13]This is done using the -c option with the PAST program. See Appendix A for an explanation of this option.

[14]The high bits of the program counter are gotten by doing a shift right operation. The amount shifted is set by a compiler flag in the PAST program. Another flag sets the maximum number of cases which are allowed in a *switch* statement without using *subswitch* routines. These compiler flags should be made into user-controllable parameters.

3.9.10. Summary of simulator structure

Most of the sections of the file generated by PAST contain information from either the RTM symbol table or the RTM statement table which is essential for simulating the target machine. The routines other than *MainLoop* and its auxiliary routines, contain primarily the symbol table information: variable names, mappings, sizes, etc. The statement table information is contained in *MainLoop* and its auxiliaries. A summary of the parts of the simulator file is given in table 3-1.

Simulator Section	Function	Primary source of information
Header	source files, creation date	user
Update macros	keep primary variables up to date	symbol table
Propagate macros	keep main variables up to date	symbol table
PAST global variables	temporaries used in calculations	symbol table
User global variables - main	declarations for ISPS main variables	symbol table
User global variables - primary	declarations for ISPS primary (mapped) variables	symbol table
Longjump labels	labels to store destination information for long jumps	statement table
VarInfo structure	information about the ISPS variables for the user interface	symbol table
preset routine	routine to initialize variables	user, symbol table
InitChannels routine	routine to open ports connected to ISPS variables	user, symbol table
Update and *Propagate* routines	link to Update and Propagate macros for user interface	symbol table
MainLoop routine	main simulation loop	statement table
Subswitch routines	routines to break up switch statement for program-specific simulators; auxiliary to *MainLoop* routine	statement table
Subroutines	routines to implement frequently called ISPS entities for architecture-specific simulators; auxiliary to *MainLoop* routine	statement table

3.10. Summary

This chapter has outlined the operation of the PAST simulation system and why we chose to make it work that way. We have rationalized our decisions with performance estimates, predicting better than a four times speedup over the ISPS simulator. Chapter 5 will show our estimates to be conservative, but first, in Chapter 4, our implementation of PAST is described in greater detail[15].

[15] Chapter 4 may be skipped on a first reading of the thesis. It gets quite detailed as it serves also as a maintainers reference for PAST.

Chapter 4 Implementation

PAST is actually a simulation system made up of five parts: a program which parses the ISPS description, a program which translates the parse tree into a set of tables, the PAST program which generates C code, the C compiler (*cc*) and a library of user interface and input/output routines (the PAST library). (See figure 4-1.) Most of the work that we did in designing the system was in writing the PAST program and the PAST library. The ISPS parsing program, *ispc*, and the translation program, *gdbrtm*, are part of the ISPS simulator and required few changes for use with PAST.

In this chapter, we discuss the features of *ispc* and *gdbrtm* which are important to our implementation of PAST and then we describe the workings of the PAST program. The PAST library needs no additional clarification as its routines are relatively straightforward. Our C code conforms with the specifications for the C programming language given by Kernighan and Ritchie [20] and should be portable at least between Unix™ systems.

4.1. The ISPS parser

The parser used with the PAST simulation system was written in C by Julius Thaddeus Kowalski and is called *ispc*[16]. It is based on the parser written in BLISS for the original ISPS simulation system running under TOPS-10. Both versions of the parser process ISPS descriptions and produce output in accord with the grammar described in [3]. The output of both parsers is the same, a GDB (Global Database) file which describes the parse tree for the ISPS description in a lisp-like format. (See Figure 4-2.) *ispc*, however, has a few more restrictions on its input than the original BLISS parser: *ispc* follows the ISPS grammar more closely, printing warning messages for deviations from the grammar; *ispc* does not allow ISPS macros to be as large as in the BLISS version.

Although we made a major change to the ISPS language – the addition of floating point

[16] This is the name used by Selvaggi.

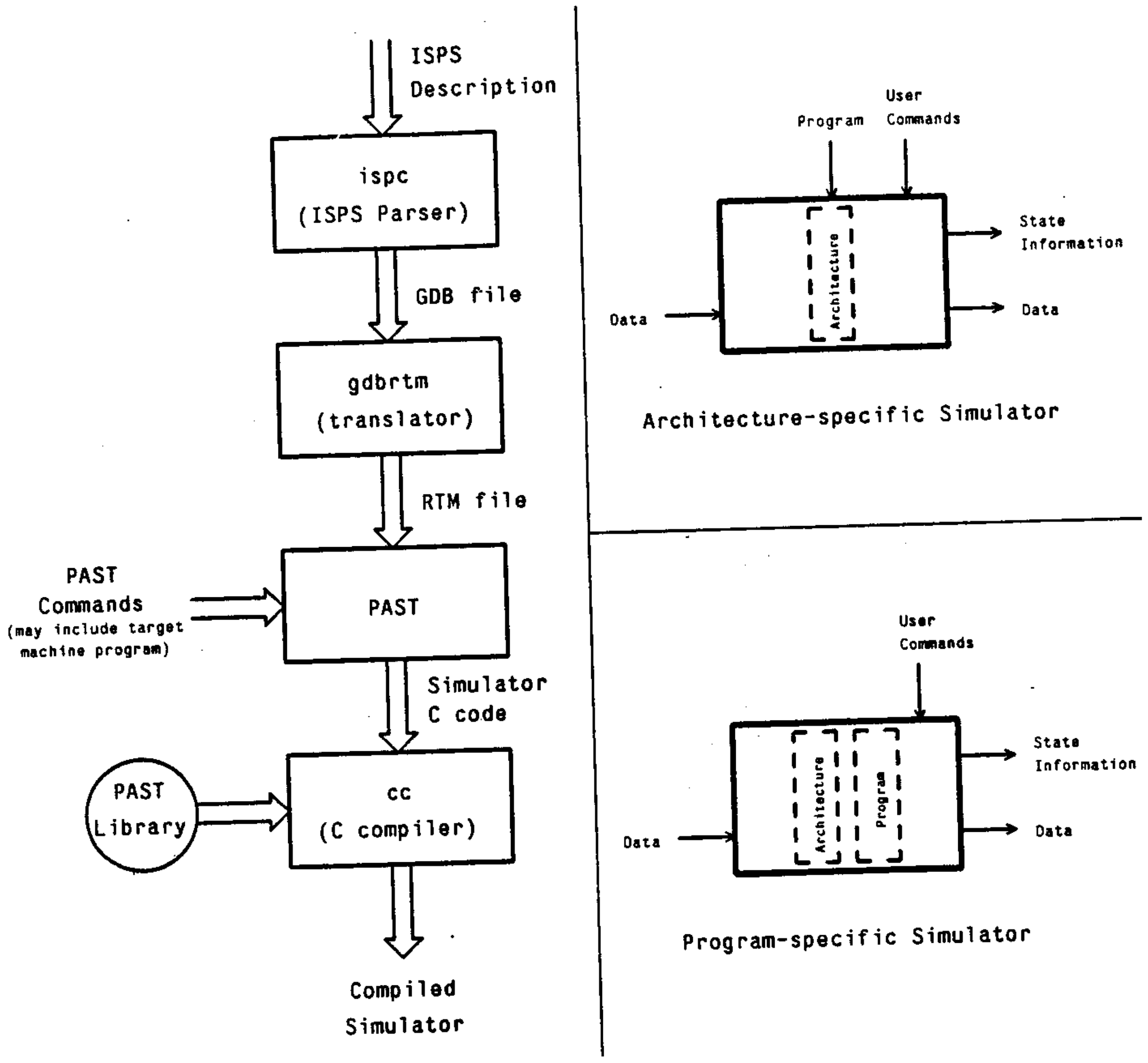

Figure 4-1: Block diagram of the PAST simulation system.

capabilities – no changes to *ispc* were necessary. The ISPS grammar includes a mechanism called a qualifier which can be used to specify additional information to be associated with entities or operators for use with application programs. The only restriction on the names of qualifiers is that they be valid identifiers; qualifiers are listed in the parse tree as they were specified in the ISPS description[17]. There is a set of predefined qualifiers that specify the number format to be used for

[17] Except that all characters are converted to a single case, e.g. upper or lower.

```
GDB:E;UNIX ISPS Compiler V2c;mark1.isp;21 May 86;16:08:17;
(ISPSDECLARATION
  (EDECLR
    (EHEAD MARK1 )
    (SECTIONLIST
      (SECTION MP.STATE (EHEAD M NIL (: 0 8191 )(: 31 0 )))
      (SECTION PC.STATE
        (EDECLRLIST
          (EHEAD CR NIL NIL (: 12 0 ))
          (EHEAD ACC NIL NIL (: 31 0 ))))
      (SECTION INSTRUCTION.FORMAT
        (EDECLRLIST
          (EHEAD PI NIL NIL (: 15 0 ))
          (EDECLR
            (EHEAD F NIL NIL (: 0 2 ))
            (EHEAD PI NIL NIL (: 15 13 )))
          (EDECLR
            (EHEAD S NIL NIL (: 0 12 ))
            (EHEAD PI NIL NIL (: 12 0 )))))
      (SECTION INSTRUCTION.EXECUTION
        (EDECLR
          (EHEAD ICYCLE NIL NIL NIL (QSET MAIN ))
          (REPEAT
            (NEXT
              (LABELLEDACTION START
              (_ (EACCESS PI )(EACCESS M NIL (EACCESS CR )(:a: 15 0 ))))
              (DECODE
                (EACCESS F )
                (NUMBEREDLIST
                  (:=n #0 (_ (EACCESS CR )(EACCESS M NIL (EACCESS S ))))
                  (:=n #1 (_ (EACCESS CR )
                       (+ (EACCESS CR )(EACCESS M NIL (EACCESS S )))))
                  (:=n #2 (_ (EACCESS ACC )(-- (EACCESS M NIL (EACCESS S )))))
                  (:=n #3 (_ (EACCESS M NIL (EACCESS S ))(EACCESS ACC )))
                  (:=n (: #4 #5 ) (_ (EACCESS ACC )
                      (- (EACCESS ACC )(EACCESS M NIL (EACCESS S )))))
                  (:=n #6
                    (IF
                      (LSS (EACCESS ACC )0 )
                      (_ (EACCESS CR )
                        (+ (EACCESS CR )1 ))))
                  (:=n #7 (EACCESS STOP (ACSET )))))
              (_ (EACCESS CR )
                (+ (EACCESS CR )1 )))))))))
```

Figure 4-2: Example of the Global Database (GDB) format.

arithmetic operations (TC, OC, SM, and US). To add floating point to the ISPS language we added another number format qualifier, which we named FP for Floating Point.

4.2. gdbrtm - the GDB to RTM translator

gdbrtm converts GDB files into *RTM* files that contain tables listing the necessary operations to simulate the described architecture on a hypothetical 3-address machine called the *Register Transfer Machine*. The *Register Transfer Machine* was designed specifically for use with ISPS and has about 130 operations sufficient for modeling machines which can be described in ISPS. The list of Register Transfer Machine operations is just one of the tables – the statement table – in the RTM file. The

RTM file also contains a symbol table, a nametable, and seven other tables listing additional information about the statements and symbols and how they are related. The contents of the 10 tables are summarized in Table 4-0 and the tables are described in detail by Barbacci et. al. [4].

Table	Name	Contents
A	*Bit-Word Sets*	Names of bits and word ranges for symbol declarations that have word or bit structures. e.g. for foo[0:127]<31:0>, the entry would be "31 0 0 127".
B	*Formals*	Lists symbols which are the formal parameters associated with ISPS procedures. (not used by PAST.)
C	*Mapping Information*	Lists primary variables which map to each main variable.
D	*Successor Vectors*	Lists choices for the conditional RTM statements (IF and DECODE) and gives statement table entry associated with each choice.
E	*Name Table*	Lists names of the symbols.
F	*NmTop*	Number of entries in name table.
G	*Temporary Registers*	Names and symbol numbers of the temporary registers allocated by gdbrtm.
H	*Symbol Table*	Information about the symbols used in the description of the target machine.
I	*Statement Table*	RTM statements and associated information.
J	*SyTop and StTop*	Numbers of entries in the Symbol and Statement tables.

Table 4-1: Tables contained in the RTM file.

The Unix version of *gdbrtm* is written in Pascal and produces the tables in a human-readable ASCII format. The BLISS version, called *GDBRTM*, produces the same tables, but as a MACRO-10 file which is assembled into object code for the DEC-10.

PAST uses a version of *gdbrtm* that we modified to handle floating point operations. The FP qualifier was added to the list of predefined qualifiers and 14 new floating point operations were added to the set of Register Transfer Machine operations. We did not have to make any structural changes to the *gdbrtm* program, but we changed array sizes and added extra variables and cases throughout the program. The code that we changed was placed within #ifdef constructs so that the changes can be easily spotted and can be removed by changing a single definition.

4.3. PAST - the code generator

The PAST program is the part of the PAST system which generates the simulator code. Its inputs are the RTM file created by *gdbrtm* and specifications from the user, provided in a file or given interactively. Its output is the simulator file described in Section 3.9.10.

We will describe the code generation as a three-phase process. In the first phase, the *initialization* phase, variables are initialized, the user specifications are processed, and symbol and statement tables are built from the information contained in the RTM file. The second phase is the *preprocessing* phase: PAST loops through the RTM statement table gathering information and modifying some of the operations to prepare for code generation. In PAST's final phase, the *code generation* phase, PAST generates C code from the information contained in the information contained in the symbol and statement tables. Before we describe PAST's three phases, we will describe the data structures which PAST uses to store the great amount of information that it handles.

4.3.1. Data Structures

PAST stores most of its data and state information in lists of structures. For the most part, these structures fall into four categories: those that make up the symbol table; those that make up the statement table; those that are dynamically created and destroyed as the RTM statements are processed; and those which store the state of the Register Transfer Machine when conditional sections of RTM code are entered. Many of the structures used in PAST are adapted from those used in the ISPS simulator. In this section we describe the major structures used in the PAST program and how they interact.

4.3.1.1. The Symbol Table

The symbol table is made up of a list of structures containing information about the individual symbols which are used in the ISPS description. The main structure describing a symbol is the Symbol structure shown in Figure 4-3. It contains the information extracted from the symbol table in the RTM file plus some additional information specific to PAST.

The information from the RTM file is described in Barbacci et. al. [4] and is stored in the following fields of the structure: SyType, SyFlags, SyDefinition, SyLabel, SyIncrement, SyBitCount (sybitcnt), SyBitWordPtr (sybwnptr), SyMapPtr, SyName (sypname), SyWordCount (sywrdcnt), and SyFather. Some of these fields are implemented as pointers to other structures so that they can be easily manipulated. The SymbolFlag (SyFlags) structure shown in Figure 4-4 holds the flags

```
struct Symbol {
    BYTE SyType;                    /* type of symbol.               */
    BYTE SyCType;                   /* LREG, REG, LMEM, MEM.         */
    SYFLAG_TYPE SyFlags;            /* boolean flags.                */
    SYMBOL_PTR  SyDefinition,       /* local parent if primary.      */
                SyMain,             /* main pointer if primary.      */
                SyFather;           /* context.                      */
    STATEMT_PTR SyLabel;            /* associated statement table    */
                                    /* entry.                        */
                                    /* also used to store address    */
    int      SyIncrement,           /* currently in temporary.       */
                                    /* number of bits in word.       */
             SyBitCount,            /* number of words.              */
             SyWordCount,           /* offset from int boundary.     */
             SyBitOffset,           /* offset from main symbol.      */
             SyWordOffset,          /* number of integers.           */
             SySize;                /* range labels.                 */
    BWV_PTR SyBitWordPtr;           /* ptr to static range list.     */
    WV_PTR SyStaticPtr;             /* pointer to list of locations  */
    WV_PTR SySetPtr;                /* which have been initialized.  */
                                    /* pointer to list of locations  */
    WV_PTR SyChangePtr;             /* which have been changed in    */
                                    /* the current level of          */
                                    /* rtmloop().                    */
                                    /* dependent symbols.            */
    SYLIST_PTR SyMapPtr;            /* symbol name.                  */
    NAME_PTR SyName;                /* simulated memory for symbol.  */
    unsigned int *SyValue;          /* number which user interface   */
    int SyNumber;                   /* references symbol by.         */

    int SyIConnect, SyOConnect;     /* port numbers to which symbol  */
                                    /* is connected.                 */
    W_PTR SyWConstant;              /* structure to store wildcard   */
                                    /* constants.                    */
    SYMBOL_PTR SyNext;              /* next symbol in symbol table.  */
};
```

Figure 4-3: Symbol structure.

provided in the RTM file and some other flags which we have defined. Some of these flags are remnants from the ISPS simulator and are not used by PAST.

```
struct SymbolFlag {
    unsigned Refactual   : 1;
    unsigned Primary     : 1;
    unsigned Secondary   : 1;
    unsigned Refformal   : 1;
    unsigned fault       : 1;
    unsigned Trace       : 1;
    unsigned Undef       : 1;
    unsigned Bitaddr     : 1;
    unsigned IConnected  : 1;
    unsigned OConnected  : 1;
    unsigned ProgCtr     : 1;   /* program counter             */
    unsigned Static      : 1;   /* static marker               */
    unsigned Set         : 1;   /* set marker                  */
    unsigned Valid       : 1;   /* valid marker                */
    unsigned AlwaysValid : 1;   /* always valid marker         */
    unsigned Change      : 1;   /* value was changed           */
    unsigned Reverse     : 1;   /* bwv was reversed.           */
    unsigned Unique      : 1;   /* name has been made unique.  */
};
```

Figure 4-4: SymbolFlag structure.

The Primary flag is used to mark a symbol which maps onto another symbol. The SyDefinition field of a primary symbol points to another symbol table entry which contains the mapping for the primary symbol onto the other symbol. The symbol which holds the mapping is called a secondary symbol and is marked with the Secondary flag; its SyDefinition field points to the structure for the symbol which the primary symbol maps to. A primary symbol may map onto another primary symbol. By following the chain of primary and secondary symbols, eventually a symbol which is not a primary or secondary is reached. This symbol we have named a *main* symbol and we have created a the SyMain field of the Symbol structure to point to the main symbol for each symbol so that it is not necessary to follow the primary/secondary chain each time access to the main symbol is required. The SyWordOffset and SyBitOffset fields of the Symbol structure are the offsets of the primary symbols from their main symbols in integer words and bits from the integer word boundary, respectively.

Several other fields were included in the Symbol structure to save computation when information must be accessed. The SySize field determined from SyBitCount is the number of integers which is needed to store the value of a symbol. The SyCType field, determined from SyType and SySize, is valid only for symbols that are variables or constants (not labels) and specifies the type of C variable which is needed to hold the value of the symbol:

- WORD - the value of the symbol fits in a single integer word. The declaration for the symbol in the generated code would be "unsigned int x;".

- LWORD - the value of the symbol requires multiple integer words for storage. The declaration would be "unsigned int x[*number of integer words*];".

- ARRAY - the symbol is a memory and each of its values fits in a single integer word. The declaration would be "unsigned int x[*memory size*];".

- LARRAY - the symbol is a memory and each of its values requires multiple integer words for storage. The declaration would be "unsigned int x[*memory size*][*number of integer words*];".

Other notable fields of the Symbol structure are SyStaticPtr, SySetPtr, and SyChangePtr which parallel the Static, Set and Change flags in the SymbolFlags structure. To explain these fields, we must first define some terms:

- A symbol or memory location is *static* if it has been declared by the user as having a constant value throughout the simulation using the STATIC command.

- A symbol is *set* if its value is currently known by PAST, e.g. a static location is always set.

- A symbol has been *changed* if its value has been set to a new value, known or otherwise. It is important to known whether a symbol's value has been changed within a conditional section of RTM code so that PAST can mark the symbol as not set, i.e. its value is not known, for the code after the conditional section.

If a symbol is static and is a register or flag, the Static flag is set. For symbols that are memories, SyStaticPtr points to a list of structures specifying the address ranges where the memory is static. In a program-specific simulator, structures listing the locations in memory containing the program code would be included in the SyStaticPtr list. The Set flag and SySetPtr field mark a symbol that is set and the Change flag and SyChangePtr field mark symbols that have been changed within the current level of the loop which processes the RTM operations.

The Valid flag is used to keep track of whether a primary symbol is up-to-date with its main symbol. When the value of a primary symbol is needed and its Valid flag is not set, the symbol is updated from its main symbol, and the Valid flag is set. Whenever a main symbol is changed, the Valid flags of all of the symbols which map to it are cleared because their values may no longer be valid. Also, when the value of a primary symbol is changed, the value is propagated back to its main symbol and all of the Valid flags for the other primaries which map to that main symbol are cleared. Main symbols are always kept up-to-date, but primaries are only updated as needed, so the Valid flag is used to mark the primaries which are valid. Non-primary symbols are always considered to be valid.

The other fields of the Symbol structure are explained adequately in Figure 4-3.

4.3.1.2. The Statement Table

Like the symbol table, the statement table is implemented as a list of structures. The Statement structures (Figure 4-5) are less complicated than the Symbol structures because they hold no dynamic state information; the statement information is static during PAST's code generation phase.

Most of the information in the Statement structure comes directly from the RTM file. Barbacci et. al. [4] describes the following fields of the structure: StFlags, StOperation, StDestination, StSource1, StSource2, StSCount, StMergeLabel, StLabel. The stslist field described by Barbacci is implemented as two fields, StSelVec and StList, in the Statement structure because different control operations require different amounts of information.

The flags associated with a Statement structure are stored in the StatementFlag structure (Figure 4-6). Most of the flags were defined for the ISPS simulator and are not used by PAST. Three of the flags were defined specifically for PAST:

```
struct Statement {
    int StOperation;
    SYMBOL_PTR  StDestination,
                StSource1,
                StSource2,
                StLabel;
    STATEMT_PTR StMergeLabel;
    int         StSCount,
                StPTime,
                StATime;
    SELECTVEC_PTR StSelVec;     /* used for SELECT ops */
    STLIST_PTR  StList;       /* used for IF,DIVERGE,BSELECT,BRANCH */
    STFLAG_TYPE StFlags;
    STATEMT_PTR StFwdLink,
                StBackLink;
};
```

Figure 4-5: The Statement structure.

```
struct StatementFlag {
    unsigned Block       : 1;    /* body was a labelled block      */
    unsigned Process     : 1;    /* PROCESS qualifier              */
    unsigned Prcedure    : 1;    /* entry point for entity body    */
    unsigned Critical    : 1;    /* CRITICAL qualifier             */
    unsigned Ptime       : 1;    /* Process/Procedure in effect    */
    unsigned Fastrtm     : 1;    /* this op/on can be in line      */
    unsigned Needsrc1    : 1;    /* operation access SRC1          */
    unsigned Needsrc2    : 1;    /* operation access SRC2          */
    unsigned Notime      : 1;    /* This RTM op/on takes 0 time    */
    unsigned Break       : 1;    /* Break flag                     */
    unsigned Trace       : 1;    /* Trace flag                     */
    unsigned Lock        : 1;    /* used to lock param.passing     */
    unsigned Ignored     : 1;    /* serialize joins                */
    unsigned Opaque      : 1;    /* disables read/write/ex tally   */
    unsigned SelectIf    : 1;    /* use if statements to           */
                                 /* represent a SELECT operation   */
    unsigned Routine     : 1;    /* if a called entity, expand     */
                                 /* the code inline (FALSE) or     */
                                 /* replace with a subroutine.     */
    unsigned Label       : 1;    /* the statement is jumped to     */
                                 /* by a LEAVE, TERMINATE,         */
                                 /* RESTART, or RESUME operation.*/
};
```

Figure 4-6: The StatementFlag structure.

- The SelectIf flag is used to mark SELECT operations – the RTM operations which implement ISPS DECODE statements – which are to be implemented using *if* statements rather than *switch* statements.

- The Expand flag marks a called entity which is to be expanded in-line rather than being implemented as a function call. (In-line expansion is discussed in sections 3.9.9.1 and 4.3.4.6.)

- The Label flag marks the destination statement for the jump operations LEAVE, TERMINATE, RESTART and RESUME. These jump operations are implemented with the *longjmp* function so a *setjmp* function call must be inserted at the destination of the jump.

The StFwdLink and StBckLink fields of the Statement structure connect the structures in a doubly linked list.

4.3.1.3. Dynamic Structures

We call *dynamic* the structures which are changed with every RTM statement or which are frequently created and destroyed. Two important structures fall under this category.

The RTM program counter is a pointer to the RTM statement currently being processed. A list of **ringstructure** structures implements a stack which stores the values of the RTM program counter for several contexts. The top structure in the stack holds the current RTM program counter value and changes every RTM cycle. The only other important field of the **ringstructure** structure stores the context in which the program counter is defined, e.g. the ISPS entity in which it is located.

For convenience and efficiency we have defined a structure, called the **codestruct** structure, for storing information about the operands associated with an RTM statement (Figure 4-7). At the beginning of an RTM statement requiring source operands, the *getsource* function is called for each operand to create a **codestruct**. Also, the *getdest* function is called to create a **codestruct** containing information about the destination operand. Much of the information in the **codestruct** structure for an operand is the same as the information contained in the Symbol structure for the operand, but fewer pointer dereferences are needed to access the information. Also, there are only three types of **codestruct** structures, in contrast to the ten types of Symbol structures, and these three types provide information that is more relevant to code generation than is provided by Symbol types. The *getsource* function determines whether the value of a symbol is known, and if so, replaces the variable with its value in the returned **codestruct** structure; the type for a symbol of known value is CONSTANT. If the value of a symbol is not known, it is typed REGISTER if it is a single word or MEMORY if the symbol needs an address to determine its value. For a destination operand, it is usually not important whether the value of the symbol is known[18], so *getdest* only returns structures with REGISTER and MEMORY types. Another benefit of the **codestruct** structure is that it creates a template which can be used with *printf* when printing the sources and destinations. The template contains the name of the symbol and spaces for the necessary indices for accessing its contents (e.g. if it is a memory); if the indices are known, they are included in the appropriate locations in the template.

[18]The exception is for the operation which sets a bit field of the destination operand, in which case it may be important to know whether the adjacent bits of the destination operand are known. The *putbyte* routine which implements this operation deals with the Symbol structures directly.

```
struct codestruct {
    int          CodeType;
    int          CodeSize;
    int          CodeBitCount;
    int          CodeOffset;
    int          CodeIndex;
    CODEFLAG_TYPE        CodeFlags;
    unsigned int         *CodeValue;
    char         *CodeString;
    SYMBOL_PTR   syptr;
    CODE_PTR     next;
};
```

Figure 4-7: The CodeStruct structure.

4.3.1.4. State Structures

For program-specific simulators, it is essential that the information known about the contents of the variables is not lost after a conditional section of RTM code (e.g. the code within an ISPS DECODE or IF statement) because if certain information is lost, a general simulator may be produced for each instruction of the program; if a general simulator is produced for each instruction, the simulator preparation time increases greatly, and the execution time of the simulators is increased as well. We thus save the entire known state of the Register Transfer Machine before each conditional section is entered and restore the state, adjusted for the changes which occurred in the conditional section, after the conditional section is exited. We store the state information in the State structure shown in Figure 4-8.

```
struct State {
    unsigned int **flags;         /* valid, set, and change flags */
                                  /* for all symbols.             */
    MEMCHUNK_PTR memchunk;        /* memory state.  saved memory  */
                                  /* contents.                    */
    WVLIST_PTR memstate;          /* list of memory locations     */
                                  /* where contents were known.   */
    WVLIST_PTR memchange;         /* list of memory locations     */
                                  /* whose contents have been     */
                                  /* changed at the current       */
                                  /* rtmloop() level.             */
};
```

Figure 4-8: The State structure.

The state information is stored in the state structure as four lists. The first list, the flags field, stores the *set*, *valid*, and *change* flags for all of the symbols. They are stored as pointers to unsigned integers because the saved *set* flags double as pointers to the values which the *set* symbols had before state was saved. (If the pointers are non-zero, then the corresponding flag was set; otherwise the flag was not set.)

The second field of the State structure points to a list of structures which store the values of the memory locations which were known when state was saved. The memory values are stored in blocks as large as there were contiguous sections of known value; the structure which stores these memory chunks is called the **MemChunk** structure and is shown in Figure 4-9.

```
/* structure for temporary storage of a section of memory.          */
/* used for saving machine state before entering a conditional      */
/* section of rtm code.                                             */

struct MemChunk {
    int size;                    /* amount saved in this chunk.     */
    unsigned int *value;         /* saved value of the memory.      */
    unsigned int *location;      /* where the values came from.     */
    struct MemChunk *next;       /* next memchunk structure.        */
};
```

Figure 4-9: The MemChunk structure.

The *memstate* and *memchange* fields of the **State** structure store the addresses of the memory ranges which had been marked as set and changed, respectively, before state was saved. The range information is stored in lists of structures, called **WordVectorList** structures, each of which contains the pointer to the symbol to which the range information pertains and a pointer to a list of structures – **WordVector** structures – containing pairs of numbers defining set or changed ranges. The **WordVectorList** and **WordVector** structures are shown in Figure 4-10.

```
/* structure for storing pairs of numbers, or single numbers        */
/* in a linked list.                                                */

struct WordVector {
    int LeftWord;
    int RightWord;
    struct WordVector *next;
};

/* structure for storing lists of number pairs associated with      */
/* particular symbols.  Each structure stores a list for a          */
/* particular symbol.                                               */

struct WordVectorList {
    SYMBOL_PTR syptr;               /* the symbol with which this     */
                                    /* list is associated.            */
    WV_PTR wvptr;                   /* list of number pairs which     */
                                    /* go with current symbol.        */
    struct WordVectorList *next;/* next wordvectorlist struct.        */
};
```

Figure 4-10: The WordVector and WordVectorList structures.

Note that there is no list containing validity information for memories. That is because PAST does not allow primaries to be memories. When one memory maps to another, the primary memory is converted by past into a register which maps to the main memory. The mapping function in this case is a little more complicated than a register to register mapping would be, but duplication of entire memories is prevented.

4.3.2. Initialization

PAST's initialization phase is very similar to the initialization of the ISPS simulator because both programs must process the RTM file. In fact, most of PAST's initialization code for building the symbol and statement tables from the RTM file was adapted from ISPS simulator code. Other functions performed in the initialization of PAST are setting variables to initial and default values, parsing the command line arguments to PAST, allocating memory to hold the values of the RTM symbols, and processing commands given by the user.

The mainline code for PAST is shown in Figure 4-11. The initialization phase covers the routines from the start of the *main* routine up to and including the *ConnectChannels* routine. Below we describe the steps in PAST's initialization phase in the order in which they occur.

4.3.2.1. Default parameters

The *defaults* routine sets up the initial file pointers for the input and output streams and sets variables that may be affected by the arguments to PAST to their default values.

4.3.2.2. Argument parsing

getargs processes the command line arguments given to PAST when invoked by the user. The valid arguments to PAST are described in Appendix A.

4.3.2.3. Command files

PAST keeps the file pointers to all open command files on a stack. The *pushreader* routine in *main* initializes the stack to the pointer to the initial command file (which defaults to *stdin*).

4.3.2.4. Variable initialization

The *initialize* routine initializes variables which do not depend on the inputs to PAST. It also initializes lists to NULL pointers.

4.3.2.5. Reading the RTM file

The *InitTables* routine builds the symbol and statement tables from the information contained in the RTM file. The name of the RTM file is one of the few required arguments to PAST, and if the name is not given in the command line arguments, the user is prompted for it by the *InitTables* routine. Once the name of the RTM file is known, *InitTables* calls another routine called *BuildTables* which procedes to build the tables. *InitTables* and almost all of the routines in its call tree were adapted from code from the ISPS simulator; most of these routines were only changed slightly.

```
main(argc, argv)
int argc;
char **argv;
{
    /* set up the default parameters.                                  */

    defaults();

    /* process the arguments to the program.  Set actual             */
    /* parameters.                                                     */

    getargs(argc, argv);

    /* initialize input file pointer.                                  */

    pushreader(ifp, OFF, 1);    /* rd from tty, NOECHO, level 1 */

    /* initialize variables and structures.                            */

    initialize();

    /* read in the rtm file and build the Symbol and Statement        */
    /* tables.                                                         */

    InitTables(rtmfilename);

    /* allocate simulated memory.                                      */

    allocate();

    /* get past commands from a file or from the user.                */

    getpst();

    /* determine whether simulator is to be general or program-      */
    /* specific.  If the PC variable and a code range have            */
    /* been specified, then produce a specific simulator.            */
    /* Otherwise produce a general one.                               */

    if (pc_ptr != NULL && code_ptr != NULL)
        specific = TRUE;
    else
        specific = FALSE;

    /* connect and number the input and output channels.              */

    ConnectChannels();

    /* preoptimize the statement table.                               */

    Preprocess();

    /* translate the RTM code and memory contents to C code.         */

    translate();

    /* close files and exit.                                           */

    Exit(EXITDONE);
}
```

Figure 4-11: PAST mainline code.

4.3.2.6. Memory Allocation

Although PAST is a simulator generation program and not a simulator, it still must keep track of values of ISPS symbols so that it can precalculate results and select conditional options when enough values are known. The *allocate* routine allocates memory for storage of the symbol values.

One of the most significant differences between PAST[19] and the ISPS simulator is in the way that they allocate memory. PAST allocates memory for each ISPS memory word or register on an integer word boundary. When the ISPS simulator allocates simulated memory for an ISPS memory, the first memory word is aligned with an integer word boundary, but the rest of the memory words are bit-aligned, e.g the least significant bit of a memory word is directly adjacent to the most significant bit of the previous memory word whether the ISPS memory had 32 bit words or not. The ISPS simulator's memory allocation scheme is more efficient in use of memory than the PAST memory allocation, but the ISPS scheme makes accessing the simulated memories more complicated. It is probably because of this greater complexity that the ISPS simulator accesses simulated memories and registers one bit at a time rather than using coarser grain masking operations as PAST does.

PAST allocates memories for primary variables in much the same way as the ISPS simulator does; the memory pointer for the primary pointer is set to the appropriate word of the corresponding main variable and the offset of the primary from the word boundary for the main variable is recorded. This differs from the allocation scheme for the primary variables in the simulators generated by PAST, in which a primary variable is a copy of the corresponding field of the main variable (see Section 4.3.1.1).

4.3.2.7. PAST Commands

The *getpst* routine is the user interface to the PAST program. The PAST user interface is based on the user interface to the ISPS simulator, but has a different set of commands. The PAST command set is listed below and is described in detail in Appendix A.

```
CODE            OCONNECT        START
END             PC              STATIC
ICONNECT        QUIT
MAKE            SETVALUE
```

The *getpst* routine processes commands supplied by the user either from a command file or interactively. *getpst* first tries to open a command file with the same root name as the RTM file and

[19] The simulators generated by PAST allocate memory in the same way that PAST does.

the extension ".pst", but if it cannot find the command file, it enters a mode in which the user can enter commands interactively. The only command mandated by PAST is the START command which tells PAST where in the ISPS description the target machine cycle begins. If the command file is found by PAST but does not contain a START command, PAST will enter its interactive mode and request a START command from the user. Either way the interactive mode is entered, PAST will not leave the interactive mode until a START command is supplied.

If only the START command is supplied by the user, PAST will make an architecture-specific simulator. If the program counter and a code range are supplied, using the PC and CODE commands, respectively, PAST will make a program-specific simulator[20]. The next section in the *main* code checks whether the program counter and a code range have been specified and sets the flag for the simulator type (*specific*) accordingly.

4.3.2.8. Input and output connections

The *ConnectChannels* routine moves information about input and output connections from a list created by the user interface routines (for the ICONNECT and OCONNECT commands) into the symbol table. The information is not put into the symbol table directly to permit the user to change his mind about variable connections while in the interactive mode.

4.3.3. Preprocessing

The preprocessing phase of PAST is made up of a single function – *Preprocess* – which loops through the statement table collecting information about and making alterations to the statements. The preprocessing phase could include preoptimizations, but, currently, *Preprocess* only does replacements which are essential to the code generation phase. *Preprocess* is implemented as a loop that steps through the statement table. Inside the loop is a *switch* statement that switches on the RTM operation of the current statement. The following paragraphs describe the functions associated with each of the RTM operations which requires special preprocessing.

The CALL operation One of the conditions which PAST considers the end of a target machine cycle is reaching a return statement (PEND) for an ISPS entity which PAST did not call previously. We permit the user to specify any labelled ISPS entity as the starting point of a target machine cycle,

[20]Although PAST does not require more information, the user should also specify that the range of memory holding the code is static, using the STATIC command, and should specify the values of the program memory using the SETVALUE command. Otherwise, PAST will assume that the program memory is dynamic and does not have known value, and PAST will produce an architecture-specific simulator for each value in the range specified by the CODE command

including labelled statements defined within subroutines. However, *gdbrtm* makes any labelled entity into a subroutine. Thus, if the labelled entity is a single statement within a subroutine, PAST will treat that statement as the entire target machine cycle. To prevent this from happening, *Preprocess* makes sure that the start label points to a single RTM statement rather than a subroutine by expanding the subroutine marked by the start label in-line. This is implemented by searching for the CALL statement which calls the entity with the name specified by the user as the starting label. That CALL statement is replaced by the contents of the called entity. Similarly the CALL to the entity named by the user as the end of the target cycle is replaced by the contents of the entity that it calls.

Another preprocessing function association associated with the CALL statement is the counting of the the number of times each ISPS subroutine is called. The call count for each subroutine is needed to determine whether the code generation phase should expand the routine in-line or make it a C function. (See sections 3.9.9.1 and 4.3.4.6.)

4.3.3.1. The CONNECT operation

In the ISPS language, parameters to functions are usually passed by value. The values are copied into formal parameter variables local to the ISPS function and changes to the formal-parameter variables within the function do not effect the original variables, the actual parameters. However, the ISPS language includes a qualifier – REF – which also lets the user specify that a variable should be passed by reference. *gdbrtm* creates a special RTM statement called CONNECT which marks a variable as passed by reference. The way that we implement the CONNECT operation is by replacing it with operations that move the actual parameter value into the formal parameter before the function call and adding operations that move the formal parameter value back into the actual parameter after the function has returned.

4.3.3.2. The NEG2C operation

NEG2C is the mnemonic for twos complement negate. In the simulators generated by PAST, all variables are implemented as unsigned integers. A standard way to do a twos complement negate with unsigned numbers is to invert the bits of the number and then add one. We implement NEG2C by replacing it with the two operations NOT and INCR (not and increment).

4.3.3.3. LEAVE, RESTART, TERMINATE, RESUME

LEAVE, RESTART, TERMINATE, and RESUME are all special types of *goto* operations which jump out of ISPS entities (subroutines). Because we sometimes implement ISPS entities as C functions, we can not use the normal C *goto* statement to implement these jumps. Instead we use a C routine called *longjmp* to make the jumps. This routine requires that the destination of the jump was marked with another routine called *setjmp* which must have been executed previously to record the stack state. The destination must be marked before the jump is made. *Preprocess* searches for LEAVE, RESTART, TERMINATE, and RESUME and marks the RTM statements that are their destinations as requiring *setjmp* routines. (See Section 3.9.4.)

4.3.4. Code Generation

The code generation phase of PAST parallels the structure of the simulator produced by PAST. First generated is the program header; then macro definitions, variable declarations, and initialization routines; and, finally, the main simulation loop is generated. The sections which contain symbol table information (see Figure 3-1) are generated by looping through the elements of the symbol table and printing the necessary information. These symbol table sections have been described in Chapter 4, and their generation is reasonably straightforward. The remaining sections – the *MainLoop* routine and its auxilary routines – are produced by following the flow of control of the RTM statements and generating C code to represent each RTM statement processed. In the next few paragraphs we will describe some of the details of the generation of code for *MainLoop* and its auxiliaries.

4.3.4.1. Simulation loop framework

The generation of architecture-specific and program-specific simulators differs primarily in the framework of the simulation loop. To produce an architecture-specific simulator, PAST generates the frame described in Section 3.9.9.1, and then calls *rtmloop* which steps through the RTM statements in the statement table generating C code until one target cycle has been completed. A program-specific simulator is produced by generating the frame described in Section 3.9.9.2 and calling *rtmloop* for each statement of the target program to be simulated. The code generated by *rtmloop* for program-specific simulators only differs from that generated for architecture-specific simulators because the values of more ISPS variables are known before entering *rtmloop* for the program-specific simulators, e.g. the value of the program counter and the values of the memory locations storing the program code.

4.3.4.2. Variable state

As mentioned in Section 4.3.1.1, PAST keeps track of whether ISPS variables are static, set, changed at the current *rtmloop* level, and valid. PAST depends on this information so that it can produce efficient code. If a variable is static (always set) or set, PAST can replace the variable with its value when using it in calculations; this allows PAST to precompute results, saving time when the generated simulator is running. Knowing whether a variable is valid saves PAST from having to generate code to extract primaries from main variables whenever the primaries are accessed, reducing both simulator generation time and simulator run time. The changed flag is used to keep track of variables which have been changed within conditional sections of RTM code so that in the code following the conditional sections, PAST will not treat unknown variables as known and vice versa. To keep the variable state information valid, a routine called *MarkSet* is called every time a variable is changed to update the flags for the variable. It is very important that the state information is not lost for program-specific simulators because lost information can make the simulators grow to unmanageable sizes.

4.3.4.3. Flow of control

As mentioned in Section 3.4, PAST's cycle is the target machine instruction cycle. The cycle begins at a point in the ISPS description specified by the user as the start of the cycle and ends at another point in the description specified by the user (or at the starting point if the description actually contains a cycle). These points, which must be labeled in the ISPS description, are represented as labeled RTM statements in the statement table. Processing for either an architecture-specific simulator or a cycle of a program-specific simulator begins at the RTM statement labeled as the starting point. The flow of control from the starting point is specified by the RTM operations; *rtmloop* interprets the RTM statements starting from the specified starting statement. Certain RTM statements explicitly specify the next statement to jump to; otherwise, control is passed from one RTM statement to the next statement in the linked list making up the statement table. Processing for a target cycle (the entire architecture-specific simulator or a single cycle of a program-specific simulator) ends when one or more of the following conditions is true:

- The RTM statement labeled as the end of the target cycle is reached.
- The RTM statement labeled as the start of the cycle is reached again. This may occur if the ISPS description actually contains a cycle.
- An unmatched PEND statement is reached. An unmatched PEND is a return from a subroutine which was not called.
- A RTM STOP statement is reached.

- A control statement which cannot properly be handled is reached. Currently, this case only occurs in program-specific simulators for the RTM *goto* operations RESTART, RESUME, TERMINATE, and LEAVE. These operations are difficult to handle because the destinations for the *gotos* must be labeled in the generated C code.

Because PAST follows the flow of control of the RTM statements, the code that it generates is already properly ordered, so it is not necessary to generate C code to represent most of the control operations. Among the operations in this category are CALL, PEND, PBEGIN, LOCK, SMERGE, PMERGE. (Some of these were mentioned in Section 3.5.1.) To follow the flow of control of the RTM statements, PAST must imitate the actions which the Register Transfer Machine would take. For example, to imitate the CALL operation, PAST calls another level of the *rtmloop* routine; the PEND operation marking the end of a subroutine is implemented as a return from the *rtmloop* routine.

Not all of the RTM control operations can be imitated at code generation time. The conditional control operations, IF and SELECT[21], must sometimes be imitated in the generated code because they depend on values of the ISPS variables.

4.3.4.4. Conditional control operations

Conditional control operations depend on the values of ISPS variables. The handling of a conditional operation depends on whether the value of the condition variable is known or not. If the value of the variable is known, PAST selects the correct branch of the conditional operation and generates the code for that branch by calling another level of the *rtmloop* routine[22]. If the value is not known, PAST generates conditional C code and calls *rtmloop* to produce code for each of the branches of the conditional RTM operation.

PAST generates a C *if* statement to represent an RTM IF statement with an unknown switching value. An RTM SELECT statement is represented by either a C *switch* statement or by cascaded *if* statements depending on how many cases the SELECT has and how many values of the switching variable fall into each case. We have tuned PAST's selection of either *switch* or *if* statements for the version of the C compiler that we have available; the parameters for deciding whether to use a *switch* statement or *if* statements are set with #define statements in the PAST code.

[21]There are two other conditional control operations – BSELECT and BRANCH – but these are just special cases of the SELECT operation.

[22]Note that this call is made from *rtmloop* so *rtmloop* is a recursive routine.

There are additional complications to generating conditional C statements from conditional RTM statements. PAST must insure that when it generates the code for each case of a conditional statement, it starts with the same information about the state of the ISPS variables (see Section 4.3.1.1), and that after code has been generated for all cases of the conditional statement, important information has not been lost. The state of PAST's symbol table may be changed by processing the branch cases of the conditional, so PAST can not process the cases sequentially without restoring the state of the symbol table between each case. Thus, before processing any cases of the conditional RTM statement, PAST saves the state of the symbol table in a State structure (see Section 4.3.1.4); then, before processing each of the cases, PAST restores the state of the symbol table to the saved values. To insure that PAST does not lose important information – e.g. the value of a variable holding the current instruction for a program-specific simulator – PAST also keeps track of the cumulative changes to the symbol table made in processing the cases. The cumulative changes are saved in a State structure. After all of the cases of the conditional RTM statement have been processed, all of the variables which were marked as changed in the cumulative State structure are marked as not known in the symbol table. If PAST did not keep track of the cumulative changes, it would have to assume that all variables were changed, and would lose much information.

The routines which save and restore variable state before and after entering conditional sections of RTM code are the most costly of PAST's routines at PAST run-time. Each of these routines must loop through the entire RTM symbol table reading and writing flags, and copying values. For memory variables, the routines must also process lists of address ranges and copy large regions of memory. This can be quite wasteful because many conditional statements only effect a small number of the ISPS variables. In future versions of PAST, the cost of these routines should be reduced. One possible way to do this is to look ahead into the branches of a conditional statement before processing it to determine which variables are affected by the conditional statement; then it would only be necessary to save the states of the affected variables.

4.3.4.5. Data, arithmetic, logic, and shift operations

We chose to declare all of the C variables representing the ISPS variables as unsigned integers (or arrays of unsigned integers) because unsigned integers were the easiest C types from which we could construct the standard ISPS data types (Unsigned, Twos complement, Ones complement, Sign Magnitude). For the standard ISPS data types all of the data-related operations are constructed from unsigned C operations. Floating point data is also stored in unsigned integers but is casted into floating point numbers when doing calculations so that the simulators produced by PAST can take advantage of the host machine's floating point operations.

Our goal with the data, arithmetic, logic, and shift operations was to generate as few instructions as possible to implement the instructions. PAST attempts to do as many computations at code generation time as are possible from the known data. There are three cases which can occur for an RTM data operation: all of the source data values are known; only some of the source data values are known; none of the source data values are known. For the case where all source values are known, PAST precomputes the result and just generates code to set the destination variable to the result. When none of the source values are known, PAST generates code which will do the necessary operations to implement the RTM operation. When only some of the source data values are known, PAST tries to minimize the number and complexity of C statements generated, but in the worst case produces the same code as when none of the source values are known except that values are substituted in for the known variables. An example of a case where PAST can compute a result from only partial knowledge of the source operands is a multiplication of an unknown value by zero; PAST just generates code to set the destination variable to zero.

Because PAST must handle so many cases, the PAST code for handling some of the RTM data-related operations gets quite complicated. In future additions to and versions of PAST, attempts should be made to generate efficient code in a more structured fashion.

We found it very difficult to write code to generate efficient C code to implement some of the more complicated operations, such as the shift and multiply operations. We decided instead to implement the more complicated operations as calls to general subroutines. The general subroutines are contained in the PAST library. Use of subroutines may effect simulator performance slightly, but assuming that the general subroutines are efficient and that the complicated operations require many C instructions to implement, the overhead due to the subroutine calls should be relatively small. Later, some of the subroutines can be replaced by macros.

4.3.4.6. Problems with simulator size

A side result of our research was the discovery of many limitations of the *cc* compiler[23]. Most of these limitations had to do with branching distances and were found because the simulators PAST produced for certain cases were very large. Because of the limitations we were prompted to determine why some of our simulators are so large.

For *program-specific* simulators, the reason is obvious: the size of a *program-specific* simulator

[23]We have described the problems that we found in Appendix D.

is proportional to the size of the program which it simulates. To reduce the size of the *switch* statement that selects the instruction to execute, we replaced this *switch* with one that calls *subswitch* routines (see Section 3.9.9.2) which contain smaller *switch* statements.

The size problems with the *architecture-specific* simulators were more unexpected. We had assumed, as did Selvaggi, that "almost all declared [ISPS] procedures are called only once" [29], and expanded all of the ISPS entities in-line. Although the assumption itself seems to be true, it does not take into account the few procedures which are called more than once. For a description of the MC68000[24], these few procedures turned out to be procedures for accessing memories and registers and were thus called many times; one routine was called 84 times. The resulting simulator was over 2 Megabytes of code and would not compile. To reduce the size of *architecture-specific* simulators we added an option of using subroutines instead of in-line expansion: the user can specify that ISPS entities called more than a given number of times should be replaced with subroutine calls rather than be expanded in-line. Table 4-2 shows the effect on generated code size of replacing in-line expansion with subroutine calls, using the MC68000 example.

Max. no. of calls expanded in-line	Resulting code size (bytes)	Max. no. of calls expanded in-line	Resulting code size (bytes)
0	186729	15	410408
1	189653	18	469600
2	200699	25	524041
3	203411	26	545933
4	211462	33	567637
6	213088	34	642200
7	223969	46	646471
8	250370	48	771049
14	251786	84	2849628

Table 4-2: Effect of expanding subroutines in-line for MC68000 description.

The size problems due to in-line expansion do not occur for all ISPS descriptions. The Mark-1 description was a single procedure so it could not be effected by in-line expansion. The PDP-8 description only had one procedure which was called multiple times, but the size was not effected significantly by in-line expansion. The Warp description also had just one multiply-called procedure, and, again, the effect one size due to in-line expansion was not considerable. In-line expansion did

[24]We did not use the MC68000 in our speed measurements because PAST can not yet handle all of functions which the MC68000 description requires.

have a significant effect on the size of the MC6502 simulator generated by PAST (Table 4-3), though not as dramatic as for the MC68000 description.

0.if 499<150 .nr 83 150

Max. no. of calls expanded in-line	Resulting code size (bytes)	Max. no. of calls expanded in-line	Resulting code size (bytes)
0	39200	6	60539
1	35460	7	60988
2	46820	8	87179
3	50255	26	90120
4	51204		

Table 4-3: Effect of expanding subroutines in-line for MC6502 description.

4.3.4.7. Current status

PAST can generate code for most of the RTM operations, but does not yet handle the Ones complement and Sign Magnitude operations. Also lacking are the twos complement multiply operation and the unsigned and twos complement division operations. Because these operations are quite complex, we think that they should be implemented as general subroutines.

We had great difficulties implementing the RTM *goto* operations RESTART, RESUME, LEAVE and TERMINATE for architecture-specific simulators, and still may not handle them properly in all cases. Implementing these operations for the program-specific simulators is even more difficult, so we directed our efforts instead to implementing more common operations. These operations should be implemented properly, but it may require a major restructuring of the code generation part of the PAST program.

Due to an oversight, PAST cannot currently handle primaries which map onto more than one word of a memory (see Figure 4-12). This is one reason why we did not use the MC68000 description as one of our test cases.

```
M\Main.Memory[0:PmemSize]<7:0>,
WM\Word.Size.Memory[0:PmemSize]<15:0> {increment:2} := M[0:PmemSize]<7:0>,
```

Figure 4-12: Example of mapping which PAST cannot handle (from MC68000 description).

PAST should eventually be able to handle any ISPS description that the ISPS simulator can handle (and also ones with data sizes greater than 128 bits) at least for the architecture-specific case. With the program-specific case, code size problems may make simulator compilation impossible for some descriptions.

4.4. Summary

In this chapter we have described the programs which make up the PAST system. The information in this chapter can supplemented with the comments contained in the program code and the code itself; we have tried to produce well-commented and readable code. In the next chapter we will describe the experiments which we performed to validate and evaluate the PAST system.

Chapter 5 Results

Since our primary goal was to produce fast simulators, our primary measure of PAST's performance was the speed of execution of the simulators it produced. We have compared the speeds of *architecture-specific* and *program-specific* simulators produced by PAST with the speed of the ISPS simulator and in one case the speed of an ad-hoc simulator. We also measured the preparation times for the PAST-generated simulators so that we could determine whether the extra preparation time for the simulators is significant. In this chapter we describe the experiments which we performed, but first we will say a little bit about the architectures and programs that we simulated.

5.1. Test cases

Our test cases were descriptions of real architectures ranging in size from a machine with 7 opcodes (6 operations) to a horizontally microcoded architecture with microwords 112 bits wide. In Table 5-1 we have presented some statistics about the ISPS descriptions. The first two columns list the sizes of the RTM symbol and statement tables respectively. The third column is the best measure of architecture complexity because it lists the average number of RTM operations which have to be processed for each instruction cycle. Note that the Warp description has significantly more RTM operations per cycle than the MC6502 description even though the MC6502 description has a larger statement table.

Machine Description	# RTM Symbols	# RTM Statements	RTM Statements executed per cycle
Mark-1	54	69	13.6
PDP-8	152	265	35.3
MC6502	288	799	41.3
Warp	455	734	354.0

Table 5-1: ISPS description complexity.

We simulated one program for each of the described architectures. The programs that we chose to simulate exercise a representative set of the machine instructions while having enough instruction cycles so that we could get accurate measurements. Time constraints prevented us from preparing more programs; however, we believe that the programs that we chose do not have atypical instruction mixes so our results should not be too inaccurate.

5.1.1. Manchester Mark-1

The Manchester University Mark-1 Computer, one of the first computers, has a very simple architecture, with a single accumulator, a 32-bit data path and a 16-bit instruction word with a 3-bit fixed-size operation code. Though there are seven possible opcodes, there are only 6 operations. Barbacci and Siewiorek [6], from which we obtained the Mark-1 description, describes the architecture in greater detail. Because of its simplicity, the Mark-1 description was the first for which we could produce a working simulator.

Our test program for the Mark-1 was a multiplication program which used all six of the operations. The algorithm we used was repeated addition because it allowed us to vary the number of instructions executed just by varying the data values; by varying the number of instructions we could make the execution time of the program great enough to be measured using the Unix *time* facility. It would have been very difficult to write a program to implement a shift and add algorithm because the only arithmetic operations the Mark-1 had were subtract and negate.

We have presented the Mark-1 as a complete example in Appendix C.

5.1.2. Digital PDP-8

The PDP-8 is more complicated than the Mark-1, having a 12-bit instruction word and a 12-bit datapath, but many more operations. Again, we obtained the ISPS description from Barbacci and Siewiorek [6], which gives a good overview of the architecture.

We obtained our test program for the PDP-8 from an article by Nestor and Thomas [23]. It is a multiplication program based on the repeated addition algorithm.

5.1.3. Motorola MC6502

We used the MC6502 as a test architecture because it has been used a lot by researchers in CMU's ECE department. We used an ISPS description of the architecture written by Jayanth Rajan. The MC6502 has a 8-bit instruction words and 8-bit data, but instructions can have a variable number of words. This is significant in PAST's program-specific mode because PAST assumes fixed length instructions; unless the user explicitly declares the address of the first word of each instruction, PAST will generate extra code for the words outside of the assumed fixed length.

The test program that we used was adapted from a double-precision multiplication program written by David Geiger. This program used a rotate and add algorithm so we could not vary the execution time by varying the data. Instead, we controlled execution time by putting the multiplication code within a loop for which we could control the number of iterations.

5.1.4. CMU Warp Cell

The idea for PAST was prompted by the large development times for simulators for machines which we are building at CMU. The Warp Systolic Array Machine, a special-purpose machine centered around an array of high-speed floating point processing cells, and several related projects have required functional simulators so that the architectures could be tested and programs could be written in parallel with hardware design. Each of these simulators took several weeks to write. It was hoped that PAST could be used to shorten the development time of future simulators.

We have written an ISPS description of the Warp cell in order to compare the simulators produced by PAST with the ad-hoc simulator written for the Warp machine and to determine whether a PAST-like tool could really be used in our design environment. The ad-hoc simulator has a slight disadvantage in the comparison because, though we only use it to simulate one Warp cell, it is capable of simulating the entire Warp array – an interface unit and ten Warp cells. The PAST simulators based on our ISPS description can only simulate a single Warp cell and thus do not have the overhead of checking which cells are to be simulated.

Our goal of simulating the Warp machine has had some major influences on the design of the PAST simulation system. For ISPS-based simulators to have reasonable speed in simulating Warp programs, we added floating-point capabilities to the ISPS language. We also added the groundwork for interprocess communication between PAST simulators with Warp in mind.

The Warp description, though small, is much more complicated than the other test descriptions because Warp is horizontally microcoded and its datapath is centered around a crossbar. (See Figure 5-1.) Also, the Warp machine has a major 200ns cycle made up of two 100ns cycles. Each 100ns cycle has a 112 bit instruction word. We modeled the Warp cell at the 200ns level, so we had to handle two of these instruction words for each instruction cycle. The horizontal microcoding and crossbar cause almost all parts of the machine to be exercised every instruction cycle which made proper sequencing of events very difficult. Sequencing information was not well documented; the most useful reference was the ad-hoc Warp simulator code because the timing problems had already been handled, but we still had to correct the timing by trial and error. Having read the Warp simulator code, it is easy to see the benefits of writing in a language such as ISPS. The ISPS description was more readable and more compact.

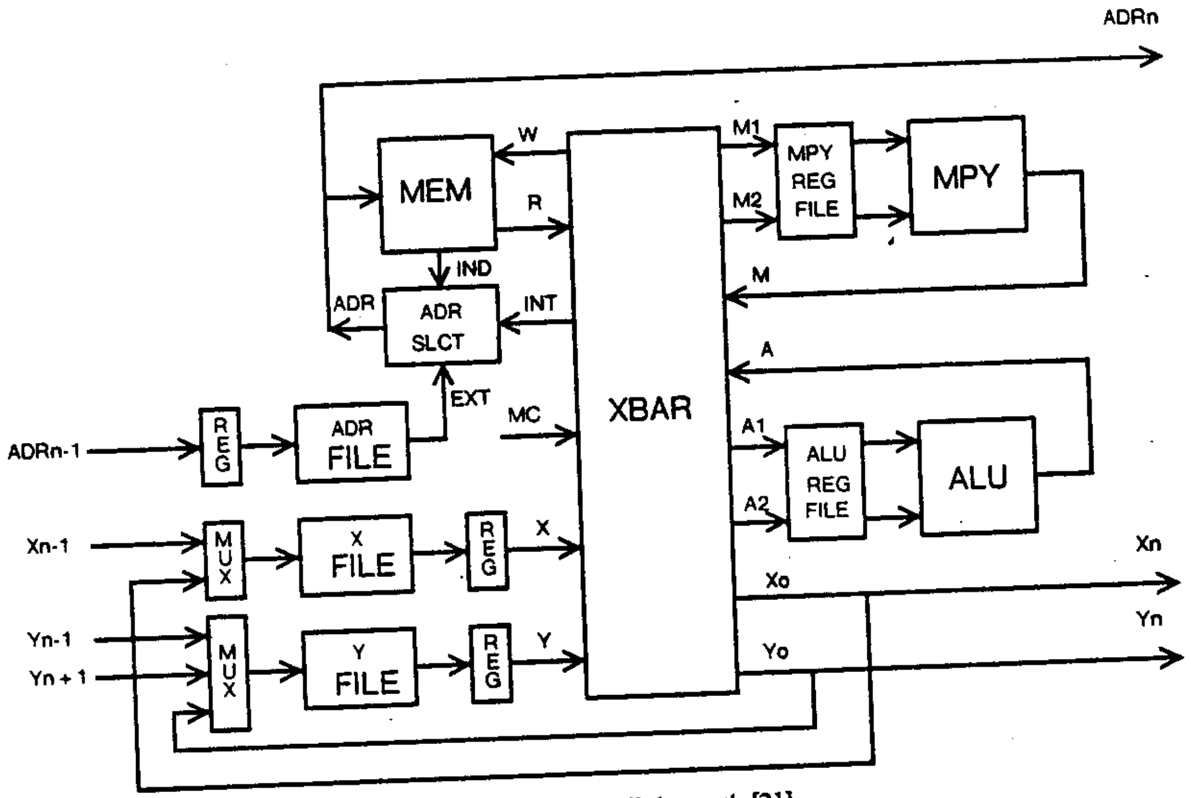

Figure 5-1: Warp cell datapath [21].

The program used with the Warp simulators was one which computes the Mandelbrot set[25]. This program was chosen because it has been run on the real hardware, it can run on a single Warp cell, its outputs can be easily verified, and, as with other test programs, its execution time can be varied easily by changing a few parameters.

[25]The algorithm for the program is the one presented in Scientific American in August, 1985 [12].

The Warp simulator is our only test case which took advantage of the I/O capabilities of PAST. Because a Warp cell is supposed to communicate with other processors, it required external inputs and outputs. The communication scheme used in the Warp machine mandated that inputs be supplied every cycle and that the simulator produce outputs every cycle. The Mandelbrot program with the parameters that we chose took about 110,500 cycles to finish so the input and output files were very large (each was 2,200,000 bytes). We found that the I/O routines dominated the execution time of both the PAST simulators and the ad-hoc Warp simulator.

5.2. Timing measurements

We made our timing measurements using the Unix™ *time* facility on a Vax 8650 computer running Mach/4.3/2/1 BSD Unix[26]. We used the real time value returned by the *time* program; *time* rounds this value to the nearest second[27]. We adjusted the problem sizes so that the smallest time values were at least a few seconds to insure that the speed differences between the simulators would be measurable. All of the measurements were done with the process being measured running as the only active process on the machine[28]. We verified that our simulation time measurements were reasonably accurate by repeating each measurement at least six times. The results were very consistent, almost all within 1 second or 5 percent of the average value[29]. The simulations with the most variations were those which used file I/O (the Warp simulations). Simulator preparation time measurements were repeated at least three times for each of the *architecture-specific* and *program-specific* simulators. Each time value presented in the tables which follow is the average of the measurements for the corresponding test case unless otherwise noted.

[26]Mach/4.3/2/1 BSD Unix is essentially 4.3 BSD Unix with some additions. Mach is an operating system/environment being developed at CMU. /2/1 are local additions to 4.3 Unix to maintain compatibility with previous CMU versions of Unix (CMU Unix 4.2 and 4.1 respectively).

[27]The "rounding" may actually be truncation. Sometimes the total user and system time indicated that the real time should have been 1 second greater. The discrepancy is probably due to real time being measured independently from user and system times.

[28]We verified that the process was the only one running by making sure there were no other users (actually, no users at all) on the machine. Also, the *at* server was not working, so we knew that no batch jobs were started.

[29]The greatest deviation was 14.8 percent, but was within 1 second of the average for that test case.

5.3. Simulator speed

For each of the four ISPS descriptions we produced *architecture-specific* and *program-specific* simulators using PAST and measured the execution times of these simulators running their respective test programs. We also simulated the same programs and descriptions with the ISPS simulator. Finally, to see how PAST simulators perform compared with an ad-hoc simulator, we ran our test program for the Warp cell on the ad-hoc Warp simulator [9]. Two versions of each of the Warp simulators (except for the ISPS simulator) were run – one with I/O and one without – because we found I/O routines to dominate over 90 percent of the simulation time when it was present, and our timing measurements are more concerned with processing speed. The execution times of the simulators are shown in Table 5-2. The values given are averages for several runs, rounded to the nearest second. The same data is presented in terms of cycles per second in Table 5-3.

Machine Description	Cycles Executed	ISPS Simulator (sec.)	PAST *architecture-specific* (sec.)	PAST *program-specific* (sec.)	Ad-Hoc (sec.)
Mark-1	851967	1741	24	74	-
PDP-8	98360	295	7	6	-
MC6502	72682	270	14	7	-
*Warp**	110500	4475**	39	22	128
Warp	110500	-	589	609	527

* Warp simulators running with no I/O and no real data.
** Estimated from ~10 minutes of running time.

Table 5-2: Simulator execution times (seconds).

Machine Description	ISPS Simulator (cy./sec.)	PAST *architecture-specific* (cy./sec.)	PAST *program-specific* (cy./sec.)	Ad-Hoc (cy./sec.)
Mark-1	489.4	35498.6	11513.1	-
PDP-8	333.4	14051.4	16393.3	-
MC6502	269.2	5197.6	10383.1	-
*Warp**	24.7**	2833.3	5022.7	863.3
Warp	-	187.6	181.4	209.7

* Warp simulators running with no I/O and no real data.
** Estimated from ~10 minutes of running time.

Table 5-3: Simulator speeds (cycles/second).

In Table 5-4 we present the factors by which PAST simulators speed up simulation compared with the ISPS simulator. PAST simulators perform quite well, achieving simulation speeds ranging from 19 to 200 times as fast as those of the ISPS simulator running the same simulations. This is much better than the conservative 4 times speedup that we predicted in Section 3.5. It should be remembered, however, that our predictions did not take all of the RTM operation optimizations into account and did not include any speedup due to reducing monitoring capabilities.

Machine Description	Dataword Size (bits)	PAST *architecture-specific*	PAST *program-specific*
Mark-1	32	72	24
PDP-8	12	42	48
MC6502	8	19	39
*Warp**	32	114	200

* Warp simulators running with no I/O and no real data.

Table 5-4: PAST speed improvement factor over ISPS.

The data confirms that the data operations are indeed a major cause of the slowness of the ISPS simulator, because the speedup improves as data word size increases. (The second column of Table 5-4 lists the sizes of the data words for the test architectures.) Since ISPS accesses data one bit at a time, total access time is proportional to the size of the data word. The difference in speedups of the Mark-1 and Warp *architecture-specific* simulators is most likely due to the fact that the Warp cell is much more complicated than the Mark-1, e.g. the description of the Warp cell is made up of many procedures so it has many more control operations that can be eliminated by PAST (Section 3.5) than the Mark-1 description, a single procedure.

The data for the *program-specific* simulators appears to conflict with the hypothesis that greater use of known information improves simulator performance: some of the *program-specific* simulators are slower than the corresponding *architecture-specific* simulators. However, the apparently anomalous data can be rationalized. The Mark-1 *program-specific* simulator which we used for measurements was a version which implemented the *MainSwitch* routine as a *switch* statement calling *subswitch* routines[30] (see Section 3.9.9.2). Thus, there were two levels of function calls for each cycle that were not present in the *architecture-specific* simulator. Because the Mark-1 description is a very simple one and function calls are expensive[31], the extra function calls have a great effect on the total

[30] This is not the same version shown in Appendix C.

[31] The *cc* compiler implements C function calls using the Vax CALLS instruction, which pushes everything onto the stack but the kitchen sink.

simulation time. We must attribute the unexpected results for the Warp simulators with I/O to either the I/O or to some effect of the greater simulator size due to function calls to the I/O routines because the simulation times without I/O fit our predictions.

The most gratifying result of this research is the data for the Warp simulators running a real program with real data. The simulators generated by PAST ran within 20 percent of the speed of the ad-hoc Warp simulator. This is quite reasonable for an automatically generated program, indicating that our research was successful. The PAST Warp simulators without I/O actually ran faster than the ad-hoc simulator. Admittedly, there is additional overhead in the custom Warp simulator because it can simulate the entire Warp system – 10 Warp cells and an interface unit – and we may not have removed all I/O related routines from the Warp simulator. Even so, these results are quite promising.

5.4. Total simulation time

Throughout this report, we have stressed that our goal was to speed up simulation by producing simulators that run fast. However, there is another major time cost involved in simulation: the time required to prepare the simulators. Simulator preparation time includes the time spent by the simulator writer in designing and debugging the simulator and the time spent by the computer converting the designer's representation into executable code. For ad-hoc simulators, the majority of the preparation time is the time spent by the simulator writer in designing the simulator and writing and debugging the simulator code; the computer's only task is compilation. Simulation tools, such as ISPS and PAST, reduce the time spent in design and coding by providing a standard easy-to-use notation for describing the architecture to be simulated. However, the role of the computer is much greater when simulation tools are used.

The preparation phase for simulation using the ISPS simulator has several steps. The first step is writing the description of the architecture; hopefully, the ISPS notation makes writing the description easy for the designer, thus minimizing the designer's time and efforts. The rest of the preparation is done by the computer: the ISPS description is parsed (by *ispc*) and is converted into an RTM file (by *gdbrtm*). An additional part of the preparation – loading the contents of the RTM file into structures – is incurred by the ISPS simulator at run time. We have included this time in the simulation time measurements.

In preparing PAST simulators, the same steps are taken as for the ISPS simulator, with two added steps. After the RTM file has been created, it is read by the PAST program and converted into

C code. This C code is then compiled by *cc* into executable code. These extra steps increase the time involved in preparation significantly. However, it is these steps that reduce the simulation time compared with the ISPS simulator. Thus, there is a tradeoff between simulator preparation time and simulation time.

Machine Description	ISPS Simulator (sec.)	PAST *architecture-specific* (sec.)	PAST *program-specific* (sec.)
Mark-1	4	10	11
PDP-8	4	15	14
MC6502	11	26	46
Warp	23	53	391

Table 5-5: Simulator preparation time.

For all of our test cases, the increases in simulation speed due to creating *architecture-specific* and *program-specific* simulators outweighed the increase in preparation time (Table 5-5) when compared with the ISPS simulator. This is because preparation of the simulators is a one-time cost, while the simulation-time savings increase linearly with the run time. If either type of PAST simulator is not run long enough, the extra preparation time may exceed the simulation-time savings. This is not a great problem for *architecture-specific* simulators because run-time savings are cumulative and, presumedly, at least a few programs will be simulated for the architecture. For *program-specific* simulators, it is a single program which must be run frequently and/or for a long time. It is possible to estimate the number of cycles where the total cost of simulation, including both preparation time and simulation time, with the ISPS simulator equals the total cost of simulation with a PAST simulator. This is the break-even point -- the number of cycles where the additional time investment in using PAST rather than ISPS is paid back by the speedup in simulation. We have determined the breakeven points for our four test cases, for both the *architecture-specific* and the *program-specific* simulators (Table 5-6). Note that break-even points involving *program-specific* simulators only apply to the programs simulated and only characterize programs of the same size running on the same architectures.

We have found that for *architecture-specific* simulators, the preparation time increases with the complexity of the architecture being simulated. For *program-specific* simulators, the preparation time increases with architecture complexity and is also proportional to the size of the program being simulated. Thus, for programs exceeding a certain size, *program-specific* simulators take longer to prepare than *architecture-specific* simulators. However, *program-specific* simulators generally (or at

Machine Description	Equation to be satisfied (solution = b)	Break-even solution (target cycles)
Mark-1	4 + b/489.4 = 10 + b/35498.6	2977
PDP-8	4 + b/333.4 = 15 + b/14051.4	3757
MC6502	11 + b/269.2 = 26 + b/5197.6	4259
*Warp**	23 + b/24.7 = 53 + b/2833.3	748

(a) Break even for ISPS and PAST *architecture-specific* simulators.

Machine Description	Equation to be satisfied (solution = b)	Break-even solution (target cycles)
Mark-1	4 + b/489.4 = 11 + b/11513.1	3578
PDP-8	4 + b/333.4 = 14 + b/16393.3	3403
MC6502	11 + b/269.2 = 46 + b/10383.1	9673
*Warp**	23 + b/24.7 = 391 + b/5022.7	9135

(b) Break even for ISPS and PAST *program-specific* simulators.

Table 5-6: Number of target cycles when PAST simulator cost equals ISPS cost.

least we think they should) run faster than *architecture-specific* simulators. Again, there is a tradeoff, and, again, we can calculate a break-even point. Table 5-7 shows the break-even points for our test cases.

Machine Description	Equation to be satisfied (solution = b)	Break-even solution (target cycles)
Mark-1	10 + b/35498.6 = 11 + b/11513.1	-17039 (never)
PDP-8	15 + b/14051.4 = 14 + b/16393.3	-98360 (immediately)
MC6502	26 + b/5197.6 = 46 + b/10383.1	208147
*Warp**	53 + b/2833.3 = 391 + b/5022.7	2196956

Table 5-7: Number of target cycles when *program-specific* cost equals *architecture-specific* cost.

Note that negative breakeven points can have different meanings. For the Mark-1 case, both preparation and simulation are more costly with the *program-specific* simulator than with the *architecture-specific* one; it is impossible to break even. In the PDP-8 case, preparation of the *program-specific* simulator took less time than preparation of the *architecture-specific* simulator (the PDP-8 program was a very short one) so using a *program-specific* simulator had an immediate payoff, and because the *program-specific* simulator is the faster simulator, the payoff continues to increase as the simulator is used. It may also possible to produce a *program-specific* simulator which has a smaller preparation time, but runs slower than the corresponding *architecture-specific* simulator; in this case, the breakeven point is a positive number of cycles, but the benefits of the smaller preparation time decrease with use of the simulator.

If we were to calculate breakeven points for a wide variety of architectures with programs of varying sizes, we could this and additional information to aid in determining which types of simulators to use for particular architectures and programs. However, there are several parameters which vary between problems:

- ISPS description size – if a description is small, PAST can create a small *architecture-specific* simulator by not expanding procedures in-line. This would make the preparation time short, but may not produce the fastest simulator.

- Architecture complexity – If an architecture is complex (in the sense of having many RTM statements per target cycle) the code for each cycle for a *program-specific* simulator can be very large, making the preparation time for long programs very large. Large code for individual cycles may also cause problems in compilation (see Appendix D). It is probably more practical to use *architecture-specific* simulators for very complex architectures.

- Program size – Large programs make the preparation time for *program-specific* simulators large and thus may make using *program-specific* simulators impractical. However, the increased preparation time may be compensated for if the program is run frequently and/or for very many cycles. For small programs, *program-specific* simulators may take less time to prepare than *architecture-specific* simulators and *program-specific* simulators may be the best option in that case.

- Cycles simulated – in most cases, the pay back of using *architecture-specific* simulators compared with using the ISPS simulator and of using *program-specific* simulators compared with using *architecture-specific* simulators increases with the number of cycles simulated. For an architecture which is simulated very little, the ISPS simulator may be the best choice. For a program which is simulated very little, *architecture-specific* simulators are more cost-effective than *program-specific* simulators.

In summary, the ISPS simulator is the best choice for architectures which are simulated for very few cycles; *architecture-specific* simulators are the most cost-effective choice in most other cases; and *program-specific* simulators are only the best choice for very short programs and/or when a program is run for a very large number of cycles.

5.5. Simulation ratios

The measure which many researchers have used to gauge the performance of their simulators is the *simulation ratio* – the time to simulate a program divided by the actual time it would take to run the program on the computer being simulated. We find this ratio to be rather ambiguous. The simulation ratio depends on the processor on which the simulation is run. We ran simulations on both a Vax 8650 and on a Vax 11/785[32] and found the simulations ran about 4 times as fast on the 8650. There is a similar difference between running on a Vax 11/785 and a Vax 11/780. Thus, just by varying the host machine, we can change the simulation ratio by a factor of 16. If the architecture that we were simulating was the Vax architecture, we could change the ratio by another factor of 16 just by stating that we were simulating a Vax 11/780 rather than a Vax 8650. Another example is the 10,000 to 1 simulation ratio given in Chapter 1 for the PDP-11. We believe that this ratio was calculated for a simulation on a Vax 11/780, so for comparison with our results measured on a Vax 8650, the ratio may be off by more than an order of magnitude. Thus, a simulation ratio given by itself can be very misleading.

In spite of the confusion which simulation ratios can cause, we have estimated them for the test cases that we simulated. To calculate the simulation ratios we need to know the speeds of the processors in instructions per second because that is how we measured the speeds of the simulators. The speeds of the Mark-1 [30] and of the Warp processing cell are known to be correct because they were presented in seconds per instruction[33]. However, the speeds of the MC6502 and the PDP-8 may not be very accurate because the speeds given for them were cycle time [25][34] and add time [13], respectively. We assumed that cycles and additions were the same as instructions in our calculations. The estimated simulation ratios are presented in Table 5-8 for simulations run on a Vax 8650. Note that a simulation ratio less than one means that the simulator ran faster than the actual machine.

[32]We have only presented results for the simulations on the Vax 8650 in this report unless noted.

[33]Actually, the speed of the Warp cell was given as a cycle time, but one microinstruction is processed per cycle.

[34]We assume that the processor is run with a 2 MHz clock. No instruction takes less than two cycles, so the speed is 1 instruction per microsecond or slower.

Machine Description	Actual Machine Speed	ISPS Simulator	PAST architecture-specific	PAST program-specific	Ad-Hoc
Mark-1	833.3 (instr./sec.)	1.703	0.023	0.072	-
PDP-8	333333.3 (add time)	999.8	23.72	20.33	-
MC6502	1000000 (cy./sec.)	3715	192.4	96.31	-
*Warp**	5000000 (instr./sec.)	202429	1765	995.5	5792
Warp	5000000 (instr./sec.)	-	26652	27563	23844

* Warp simulators running with no I/O and no real data.

Table 5-8: Estimated simulation ratios for PAST simulators.

5.6. Summary

In this chapter, we have presented measurements which characterize the simulators generated by PAST and contrast them with the ISPS simulator. We have also shown the effects of in-line expansion of subroutines on simulator size. In addition to showing the traits of the PAST and ISPS simulators, the data which we collected can be generalized and can be used to aid in choosing appropriate simulation methods for particular problems.

Chapter 6 Conclusions

We have developed a tool called PAST which automatically generates functional simulators from ISPS descriptions. PAST has two modes of operation:

- *Architecture-specific* mode: an *architecture-specific* simulator for a computer description is generated. This simulator can execute any code written to run on the described computer.

- *Program-specific* mode: PAST is given both the computer description and code for a specific program and produces a simulator that executes only the given program.

PAST generates simulator code which is linked with a library of common routines to produce a complete simulator. The PAST library includes user interface routines that allow the user to interact with the simulator before each simulated instruction cycle for the target machine. In addition, this library includes a set of input and output routines that provide the PAST simulators with a uniform way of reading from files, writing to files, or communicating with other simulators via BSD Unix 4.2 IPC ports. The interprocess communication, when fully implemented, will let PAST support simulation of multiprocessor systems such as the CMU Warp array. PAST, the PAST library, and the simulators generated by PAST are all written in the C programming language and are thus portable among systems that support C. However, the IPC port feature will only function under systems which support Unix 4.2 IPC ports.

PAST produces compiled simulators from the same inputs (with slightly different specifications) as the interpretive ISPS simulator. The structure of PAST is similar to that of the C version of the ISPS simulator, but the loop which interprets and evaluates Register Transfer Machine (RTM) instructions in the ISPS simulator has been replaced with a a loop which interprets the RTM instructions and generates optimized C code. The C code is compiled and linked with the PAST library to create an executable simulator. PAST has to deal with a tradeoff of simulator size, which effects the preparation time of the simulator and simulator speed. Also, limitations of the C compiler dictate that the code produced by PAST cannot be too large. When simulators grow too large, in-line expansion has to be replaced with calls to subroutines, and large switch statements must be broken up. These concessions slow down the simulators generated by PAST, but the PAST simulators are still much faster than the ISPS simulator.

The *architecture-specific* simulators produced by PAST simulate instructions faster than the ISPS simulator for several reasons:

- The loop overhead associated with processing RTM instructions in the ISPS simulator is not included in the simulators generated by PAST (Section 3.4).
- PAST does not generate C code for some of the RTM control operations, eliminating their costs from the simulation time.
- PAST makes better use of the information contained in the ISPS description to create code that is optimized for the operations and operand sizes of the described architecture.
- The simulators produced by PAST access memories and registers in a more efficient way than does the ISPS simulator.

These factors combine to let the *architecture-specific* simulators generated by PAST run 19 to 144 times as fast as the ISPS simulator for our test cases. Our initial analysis of PAST's optimizations over the ISPS simulator makes us fairly confident that PAST will always produce simulators at least four times as fast as the ISPS simulator for descriptions of reasonable size (at least as big as the description of the Mark-1).

Program-specific simulators produced by PAST showed as much as 50 percent speed improvement over the *architecture-specific* simulators and as much as a 200 times speed improvement over the ISPS simulator, but also showed a 67 percent degradation in speed for one case. The speed improvement is due to use of even more known information than the *architecture-specific* simulators have available: with the program known, PAST can precompute many values and often reduce a conditional statement to a single case. There is a fixed loop overhead for program-specific simulators which is incurred once for each instruction cycle of the target machine being simulated. This overhead becomes dominant when very simple descriptions are simulated, and this accounts for the anomalous case when a *program-specific* simulator was slower than the corresponding *architecture-specific* simulator. The *program-specific* simulators all ran faster than the ISPS simulator due to the same reasons as described for the *architecture-specific* simulators and due to the greater information known when they were generated.

The cost to generate or prepare a simulator depends on the method used. We have determined break-even points which can be used to decide which type of PAST simulator or the ISPS simulator is most cost-effective for a particular application. One simulation method breaks even with another when its total preparations and run time equal those of the other simulation method. *Architecture-specific* simulators are almost always more cost-effective than the ISPS simulator. *Program-specific* simulators are usually more cost-effective than the ISPS simulator, but are only more cost-effective than *architecture-specific* simulators when the programs to be simulated are simulated many times or run for many target machine cycles. The ISPS simulator is only the most cost-effective choice when the architecture is just going to be simulated for a few cycles.

In comparing simulator performance, the effect of input and output routines on simulator speed cannot be neglected. When a simulator uses much input and output, the I/O can be very costly and can dominate the simulation time.

As PAST is a prototype tool, there are still more features which can be implemented and other improvements which can be made. We have implemented most of the operations of the Register Transfer Machine, but there are still several operations which should be implemented to make PAST as general as the ISPS simulator, notably the ones complement and sign magnitude operations. With tuning, the generation time for simulators could be reduced, and it may also be possible to speed up the generated simulators. Overall, though, we are reasonably satisfied with PAST's performance. The successful simulation of a real computer system, the Warp cell, in reasonable time makes us confident that our approach was successful.

We saved much time and effort by using some of the existing ISPS software, and also maintained compatibility with the ISPS simulator by doing so. Although the PAST simulators have minimal monitoring capabilities, the user can still simulate the same ISPS descriptions with the ISPS simulator if extensive monitoring is desired (assuming that the description does not use the new floating point operations).

If another version of PAST is written, it might be interesting to have it generate code in a language which supports concurrency, such as Concurrent C, so that it can simulate some of the parallel constructs supported by ISPS more accurately.

In summary, we have successfully met the requirements that we established for a useful functional simulation tool:

- The simulators produced by PAST can execute real programs. This was demonstrated for four different computer architectures.

- PAST simulators provide the user with state information through use of a standard interface. This interface is implemented as a library of routines which can be linked with the C code that PAST produces.

- The architecture is described to the PAST simulation system as an ISPS description. The ISPS language has been designed so that descriptions of architectures are structured, and thus easy to read, write and modify. The description of the Warp processing cell written in ISPS is much more concise than the custom simulator written in C, and the writing of the ISPS code took much less time than writing the custom simulator.

- The speed of the Warp simulators generated by PAST was within 20 percent of the speed of the ad-hoc Warp simulator when I/O was included. Without I/O the PAST simulators were faster than the ad-hoc simulator.

Appendix A
Manual Entry for PAST

NAME

past – create a simulator from an ISPS description.

SYNOPSIS

past [*rtmfile*] [-o *outfile*] [-e *errfile*] [-c FORCE| *number*]

DESCRIPTION

Past translates an *.rtm* file created from an *ISPS* description (by *ispsp*(1) and *gdbrtm*(1)) and an optional command file into C code with routines to simulate the described architecture. The C file is compiled and linked with user interface routines (in the library *libpast.a*) using the *cc* compiler to create an executable simulator.

By default, output is routed to stdout. If no rtm file is specified, past prompts the user.

Command line arguments can be specified in any order. The following are the valid arguments for **past:**

−o *outfile* routes the output of **past** to the file *outfile*. If **−o** is not specified, the output is sent to *stdout*.

−e *errfile* routes the error output of **past** to the file *errfile*. If **−e** is not specified, error messages are sent to *stderr*.

−c FORCE|*number*
is used to specify the maximum number of calls an ISPS entity can have and still have its code expanded in-line. If FORCE is specified, all ISPS entities will be expanded; if a *number* is given, only ISPS entities called fewer than the specified *number* of times will be expanded in-line. The default case sets the number of times an entity can be expanded to one.

−n is used to turn off saving of state information when entering conditional sections of the RTM code. This saves code production time, but may make the code produced a little longer and the speed of simulation a little slower. This argument only affects general simulators; it is ignored for program-specific simulators.

rtmfile is the name of the file from which the symbol and statement tables for the ISPS description are to be read. If the given filename has no extension, the extension defaults to **.rtm**. The name of the past command file is determined by adding the extension .pst to the root name of the *rtmfile*.

If no command file is found, **past** enters an interactive mode in which the user can enter commands. The only mandatory command is the one which specifies the starting statement for the simulation loop: START <entity-name>. If a program-specific simulator is desired, the CODE and PC commands must also be given and the program memory should be set using the SETVALUE command. The commands are described below:

CODE *range* specifies the range of program counter values which are valid for the current program. The range is given as number pairs separated by commas (e.g. 1:5,8:10 or 1:10). This command is mandatory for generating program-specific simulators.

END *label* specifies the point in the ISPS description which is the end of the instruction cycle for the machine being simulated. The point in the description must be labeled with a unique name.

ICONNECT *streamname, streamtype, variablelist*
connects a list of variables from the ISPS description to an input stream which can be either a file, f, or an IPC port, p. The input connections can be disabled in the simulator produced by **past**, but new input connections can not be made. Thus, all variables which might need to be connected to input streams during simulation should be ICONNECTed when running **past**.

MAKE causes past to leave its interactive mode and to try to make a simulator from the information provided. If the START command has not been given, **past** ignores the MAKE command and remains in its interactive mode.

OCONNECT *streamname, streamtype, variablelist*
connects a list of variables from the ISPS description to an output stream which can be either a file, f, or an IPC port, **p**. The output connections can be disabled in the simulator produced by **past**, but new output connections can not be made. Thus, all variables which might need to be connected to output streams during simulation should be OCONNECTed when running **past**.

PC *variable address*
specifies the ISPS variable which represents the program counter for the machine to be simulated. If the variable is a memory, the address for the program counter should also be specified. This command is mandatory for generating program-specific simulators, and its use is also suggested when generating architecture-specific simulators so that breakpoints can be set when running the simulator produced.

QUIT exits from the **past** *program without making a simulator.*

SETVALUE *variable value*

SETVALUE *memoryvariable address value*
initializes an ISPS variable to the specified value. For memory variables, the memory address must be specified in addition to a value to which the address should be set. The initial values of variables specified to past will be the initial values of the corresponding variables in the simulators produced by **past**.

START *label* specifies the point in the ISPS description which is the start of the instruction cycle for the machine being simulated. The point in the description must be labeled with a unique name.

STATIC *variable*

STATIC *memoryvariable[range]*

is used to declare that the value of a variable will remain static throughout the simultation. past uses this information to precalculate results and to predetermine which branch of conditional statements to take so that it does not need to generate code for all of the branches. For a program-specific simulator, the memory range holding the program code should be declared as static; otherwise, past will generate an entire architecture-specific simulator for each instruction of the program.

FILES

/usr/bss/bin/past	executable past code
file.rtm	file containing the symbol and statement table information for the ISPS description.
file.pst	past command file.
/usr/bss/lib/libpast.a	user interface library.

BUGS

past is not yet fully implemented.

SEE ALSO

"The ISPS Computer Description Language"
"The ISPS Simulator Manual"
pastsim(1), isps(1), gdbrtm(1), gdbsim(1)

HISTORY

20-Jun-86 Bruce Siegell (bss) at Carnegie-Mellon University
Created.

Appendix B
Manual entry for simulators produced by PAST

NAME

pastsim – simulator produced by the **past** program.

SYNOPSIS

pastsim [options]

DESCRIPTION

The name **pastsim** represents any simulator produced by the **past** program. **pastsim** simulates an architecture described in ISPS. The standard user interface linked with the code produced by **past** provides the user with commands for setting breakpoints, single-stepping, and examining and setting the ISPS variables which represent the registers of the machine being simulated.

By default, output is routed to stdout. Input can come from command files or from stdin. Command file names can be listed on the invocation line for the simulator, but should not be preceded by a '-'. **pastsim** also accepts several other arguments (options) which may override the commands in the command file. Most arguments are processed from left to right. The **–r** and **–s** options are executed after all other options except for **–q**. **–q** is executed after all other options.

The valid commands for running **pastsim** interactively are described below:

! *text* Lines beginning with a exclamation point are treated as comment lines and are ignored. There must be a blank space between the exclamation point and any following text.

BREAKPOINT *number*
sets a breakpoint at the specified value of the program counter. The program counter value is checked against the breakpoint list at the beginning of the instruction loop. The breakpoint list is displayed whenever a BREAKPOINT or DBREAKPOINT command is issued.

CYCLE displays the number of target machine cycles which have been executed since the simulator was invoked.

DBREAKPOINT *number*
removes the specified breakpoint from the breakpoint list. The breakpoint list is desplayed whenever a BREAKPOINT or DBREAKPOINT command is issued.

DUMP *filename*
prints the values of all non-zero valued variables into the specified file. For each non-zero variable, a SETVALUE command line is printed. The dump file can be read by the simulator to restore the variable values to the state they were in when the DUMP command was issued.

ECHO on|off turns the echoing of commands from command lines **on** or **off**. At program initialization, echoing is set to **on**.

ICONNECT *streamname, streamtype, variablelist*
connects a set of variables to the specified input stream. The stream can either be a **file** or an IPC **port**. The only variables which can be ICONNECTed are the ones which were specified as ICONNECTed when PAST created the simulator. When variables are ICONNECTed, their old connections are closed and the new connection starts at the beginning of the specified stream.

NAMES *string*
displays the variable or variables whose names begin with the given string. If the string matches weveral variables, the user is asked whether to list all matching variables.

ICONNECT *streamname, streamtype, variablelist*
connects a set of variables to the specified output stream. The stream can either be a **file** or an IPC **port**. The only variables which can be OCONNECTed are the ones which were specified as OCONNECTed when PAST created the simulator. When variables are OCONNECTed, their old connections are closed and the new connection starts at the beginning of the specified stream.

PROMPT *variable*
changes the variable displayed as the prompt. The default prompt variable is the program counter if it was specified to PAST. Only the least significant integer word of the prompt variable is displayed in the prompt.

QUIT is the only way to exit the debugger from the interactive mode other than a "kill -9" initiated from outside the simulator. All other signals are trapped and return to the user interface.

RADIX binary|octal|decimal|hexadecimal|floatingpoint
specifies the default number format to be used in the reading and writing of values and addresses of variables. The format can be overrided by prefixing values with characters which specify their formats: ' for **binary**, # for **octal**, % for **decimal**, " for **hexadecimal**, and ^ for **floatingpoint**. At program initialization, the default number format is set to **hexadecimal**.

READ *filename*
reads a list of simulator commands from the specified file and executes them sequentially. The commands are echoed only if echoing is set to **on**.

RESET clears all variables and then resets the values of the variables whose values were specified to PAST and sets the cycle count back to zero.

RUN begins the simulation. Simulation continues until the program is interrupted or a breakpoint is reached.

SETVALUE *variable value*

SETVALUE *memoryvariable address value*
sets the specified variable to the given value. If the variable is a memory, the address into the memory must be specified. The value and the memory address must be given in the current number format as specified by the RADIX command or they must be preceded by a character specifying another number format.

STEP *number* begins the simulation. Simulation continues until the specified number of cycles has been executed, a breakpoint is reached, or the program is interrupted.

VALUE *variable*
VALUE *memoryvariable address* displays the value of the specified variable in the current number format as specified by the RADIX command. If the variable is a memory, the address into the memory must be specified. The memory address must be given in the current number format as specified by the RADIX command or must be preceded by a character specifying another number format.

OPTIONS

−b *breakpoint*
sets a breakpoint. *A breakpoint is defined to be a value of the program counter variable where the simulator is to stop running.*

−B *breakpoint*
clears a breakpoint. The specified breakpoint is deleted from the breakpoint list.

−e turns echoing of commands from command files ON.

−E turns echoing of commands from command files OFF.

−o *outputfile* routes output, which by default is deposited on stdout, to the file <*outputfile*>.

−q tells the simulator to quit after execution of all other command line arguments.

−r tells the simulator to run after processing of all other command line arguments. The simulator will run until a breakpoint is reached or until stopped by an interrupt from the user. Subsequent −s arguments or STEP commands from command files can override **−r**.

−R *radix* sets the default radix to the one specified. Valid radices are BINARY, OCTAL, DECIMAL, HEXADECIMAL, and FLOATINGPOINT.

−s *number* tells the simulator to run for the specified number of steps after processing of all other command line arguments. Subsequent −s or **−r** arguments or STEP or RUN commands from command files can override −s.

FILES

pastsim	the simulator generated by **past**.
cmdfile	a file containing commands.

BUGS

The **decimal** and **octal** radices are not yet implemented.

SEE ALSO

past(1)

HISTORY

20-Jun-86 Bruce S. Siegell (bss) at Carnegie-Mellon University.
Created.

Appendix C
Mark-1 Example

C.1. Architecture-specific simulator

The architecture-specific Mark-1 simulator was prepared from the ISPS description of the Mark-1 as follows:

```
/usr/bss/bin/ispc mark1
/usr/bss/bin/past mark1 -o mark1.c -p mark1.pst
cc -c mark1.c
cc -o mark1 mark1.o -lpast
```

ispc is a command file which calls the ISPS parser and the *gdbrtm* translator.

The following sections show all of the files involved in generating an architecture-specific simulator.

C.1.1. ISPS description - mark1.isp

```
mark1 :=
    begin

** mp.state **

m[0:8191]<31:0>

** pc.state **

cr<12:0>,       ! control register
acc<31:0>       ! accumulator

** instruction.format **

pi<15:0>,       ! present instruction
    f<0:2> := pi<15:13>,        ! function
    s<0:12> := pi<12:0>         ! address

** instruction.execution **

icycle {main} :=                ! instruction cycle
    begin
    REPEAT
        begin
start:= pi = m[cr]<15:0> next
        DECODE f =>
            begin
            #0 := cr = m[s],
            #1 := cr = cr + m[s],
            #2 := acc = -m[s],
            #3 := m[s] = acc,
            #4:#5 :=
                acc = acc - m[s],
            #6 := IF acc LSS 0 => cr = cr + 1,
            #7 := stop()
            end next
        cr = cr + 1,
        end
    end
end
```

C.1.2. GDB file - mark1.gdb

```
GDB:E;UNIX ISPS Compiler V2c;mark1.isp;21 May 86;16:08:17;
(ISPSDECLARATION
   (EDECLR
      (EHEAD MARK1 )
      (SECTIONLIST
         (SECTION
            MP.STATE
            (EHEAD M NIL (: 0 8191 )(: 31 0 )))
         (SECTION
            PC.STATE
            (EDECLRLIST
               (EHEAD CR NIL NIL (: 12 0 ))
               (EHEAD ACC NIL NIL (: 31 0 ))))
         (SECTION
            INSTRUCTION.FORMAT
            (EDECLRLIST
               (EHEAD PI NIL NIL (: 15 0 ))
               (EDECLR
                  (EHEAD F NIL NIL (: 0 2 ))
                  (EHEAD PI NIL NIL (: 15 13 )))
               (EDECLR
                  (EHEAD S NIL NIL (: 0 12 ))
                  (EHEAD PI NIL NIL (: 12 0 )))))
         (SECTION
            INSTRUCTION.EXECUTION
            (EDECLR
               (EHEAD ICYCLE NIL NIL NIL (QSET MAIN ))
               (REPEAT
                  (NEXT
                     (LABELLEDACTION START
                     (_
                        (EACCESS PI )(EACCESS M NIL (EACCESS CR )(:a: 15 0 ))))
                     (DECODE
                        (EACCESS F )
                        (NUMBEREDLIST
                           (:=n
                              #0
                              (_
                                 (EACCESS CR )(EACCESS M NIL (EACCESS S ))))
                           (:=n
                              #1
                              (_
                                 (EACCESS CR )
                                 (+ (EACCESS CR )(EACCESS M NIL (EACCESS S )))))
                           (:=n
                              #2
                              (_
                                 (EACCESS ACC )(-- (EACCESS M NIL (EACCESS S )))))
                           (:=n
                              #3
                              (_
                                 (EACCESS M NIL (EACCESS S ))(EACCESS ACC )))
                           (:=n
                              (: #4 #5 )
                              (_
                                 (EACCESS ACC )
                                 (- (EACCESS ACC )(EACCESS M NIL (EACCESS S )))))
```

```
          (:=n
             #6
             (IF
                (LSS (EACCESS ACC )0 )
                (_
                   (EACCESS CR )
                   (+ (EACCESS CR )1 ))))
          (:=n
             #7 (EACCESS STOP (ACSET )))))
    (_
       (EACCESS CR )
       (+ (EACCESS CR )1 )))))))))
```

C.1.3. RTM file - mark1.rtm

```
A
 3 15 13 0 0
 4 12 0 0 0
 9 17 0 0 0
 11 127 0 0 0
 12 127 0 0 0
 13 127 0 0 0
 15 127 0 0 0
 16 7 0 0 0
 17 127 0 0 0
 18 7 0 0 0
 19 127 0 0 0
 20 0 0 0 0
 22 0 0 0 0
 23 127 0 0 0
 24 7 0 0 0
 25 127 0 0 0
 26 127 0 0 0
 27 7 0 0 0
 28 127 0 0 0
 29 127 0 0 0
 30 17 0 0 0
 31 127 0 0 0
 32 127 0 0 0
 33 7 0 0 0
 34 15 0 0 0
 36 0 2 0 0
 37 31 0 0 0
 38 31 0 0 8191
 39 0 12 0 0
 40 31 0 0 0
 41 31 0 0 0
 43 0 0 0 0
 44 12 0 0 0

B
 10 9
 12 11
 16 15
 18 17
 20 19
 22 21
 25 23 24
 28 26 27
 31 29 30
 33 32
```

```
C
 34 39 36

D
 15 1 16 46 0 1 18 47 0 1 21 48 0 1 24 49 0 2 26 50 51 1 29 52 0 1 34 53 0
 30 32 31

E
 'ACC' 41
 'C.O.P.' 32
 'COUNT.ONE' 33
 'CR' 44
 'D.P.' 9
 'DELAY' 10
 'F' 36
 'F.O.P.' 15
 'FIRST.ONE' 16
 'I.R.P.' 21
 'ICYCLE' 5
 'IS.RUNNING' 22
 'L.O.P.' 17
 'LAST.ONE' 18
 'M' 38
 'M.L.P.1' 23
 'M.L.P.2' 24
 'M.R.P.1' 26
 'M.R.P.2' 27
 'MARK1' 2
 'MASK.LEFT' 25
 'MASK.RIGHT' 28
 'NO.OP' 8
 'P.P.' 19
 'PARITY' 20
 'PI' 34
 'PRELUDE' 1
 'S' 39
 'START' 6
 'STOP' 7
 'T.W.P.1' 29
 'T.W.P.2' 30
 'TIME.WAIT' 31
 'UNDEFINED' 13
 'UNPREDICTABLE' 14
 'W.P.' 11
 'WAIT' 12

F
 36

G
37 /.T00037/
40 /.T00040/
43 /.T00043/

H
 0 0 0 8 0 0 0 0
 4 0 0 1 0 0 0 0
 4 0 0 4 0 0 0 1
 2 4 34 0 0 3 0 0
 2 4 34 0 0 13 0 0
 4 0 0 7 0 0 0 2
 4 0 0 10 0 0 0 5
 4 0 0 50 0 0 0 1
 4 0 0 40 0 0 0 1
 2 0 0 0 0 18 0 10
```

MARK-1 EXAMPLE

```
4 0 0 42 0 0 0 1
2 0 0 0 0 128 0 12
2 0 0 44 0 128 0 1
2 0 0 46 0 128 0 1
4 0 0 48 0 0 0 1
2 0 0 0 0 128 0 16
2 0 0 52 0 8 0 1
2 0 0 0 0 128 0 18
2 0 0 54 0 8 0 1
2 0 0 0 0 128 0 20
6 0 0 56 0 1 0 1
4 8 0 0 0 0 0 22
6 0 0 58 0 1 0 1
2 0 0 0 0 128 0 25
2 0 0 0 0 8 0 25
2 0 0 60 0 128 0 1
2 0 0 0 0 128 0 28
2 0 0 0 0 8 0 28
2 0 0 62 0 128 0 1
2 0 0 0 0 128 0 31
2 0 0 0 0 18 0 31
2 0 0 64 0 128 0 1
2 0 0 0 0 128 0 33
2 0 0 66 0 8 0 1
2 0 0 0 0 16 0 2
5 0 0 0 0 16 0 0 983040
2 2 3 0 0 3 0 2
7 0 0 0 0 32 0 0 37
1 0 0 0 1 32 8192 2
2 2 4 0 0 13 0 2
7 0 0 0 0 32 0 0 40
2 0 0 0 0 32 0 2
3 0 0 0 0 2 0 0 0
7 0 0 0 0 1 0 0 43
2 0 0 0 0 13 0 2
3 0 0 0 0 2 0 0 1
3 0 0 0 0 3 0 0 0
3 0 0 0 0 3 0 0 1
3 0 0 0 0 3 0 0 2
3 0 0 0 0 3 0 0 3
3 0 0 0 0 3 0 0 4
3 0 0 0 0 3 0 0 5
3 0 0 0 0 3 0 0 6
3 0 0 0 0 3 0 0 7

I
 32 226 0 0 0 0 4 2 0 0
 36 208 0 0 0 0 68 1 0 0
 32 210 0 0 0 0 4 2 0 0
 32 221 0 0 0 0 68 0 0 0
 42 208 0 0 0 0 39 2 0 0
 32 210 0 0 0 0 7 5 0 0
 32 221 0 0 0 0 39 0 0 0
 36 208 0 0 0 0 38 5 0 0
 32 219 0 0 0 0 37 0 0 0
 32 221 0 0 0 0 14 0 0 0
 33 208 0 0 0 0 13 6 0 0
 160 19 37 38 44 0 0 0 0 0
 224 21 34 37 35 0 0 0 0 0
 32 209 0 0 0 0 10 6 0 0
 32 210 0 0 0 0 10 6 0 0
 96 193 0 36 0 7 35 0 0 0
 160 19 44 38 39 0 0 0 0 0
 32 220 0 0 0 0 35 0 0 0
 160 19 37 38 39 0 0 0 0 0
```

MARK-1 EXAMPLE

```
224 65 44 44 37 0 0 0 0 0
32 220 0 0 0 0 35 0 0 0
160 19 37 38 39 0 0 0 0 0
96 64 41 37 0 0 0 0 0 0
32 220 0 0 0 0 35 0 0 0
224 20 38 39 41 0 0 0 0 0
32 220 0 0 0 0 35 0 0 0
160 19 40 38 39 0 0 0 0 0
224 66 41 41 40 0 0 0 0 0
32 220 0 0 0 0 35 0 0 0
224 75 43 41 42 0 0 0 0 0
96 192 0 43 0 2 32 0 0 0
224 65 44 44 45 0 0 0 0 0
32 219 0 0 0 0 30 0 0 0
32 220 0 0 0 0 35 0 0 0
32 227 0 0 0 0 0 0 0 0
32 219 0 0 0 0 15 0 0 0
224 65 44 44 45 0 0 0 0 0
32 221 0 0 0 0 8 0 0 0
32 209 0 0 0 0 7 5 0 0
32 209 0 0 0 0 4 2 0 0
36 208 0 0 0 0 41 8 0 0
32 209 0 0 0 0 40 8 0 0
36 208 0 0 0 0 43 10 0 0
32 209 0 0 0 0 42 10 0 0
36 208 0 0 0 0 45 12 0 0
32 209 0 0 0 0 44 12 0 0
36 208 0 0 0 0 47 13 0 0
32 209 0 0 0 0 46 13 0 0
36 208 0 0 0 0 49 14 0 0
32 209 0 0 0 0 48 14 0 0
36 208 0 0 0 0 51 7 0 0
32 209 0 0 0 0 50 7 0 0
36 208 0 0 0 0 53 16 0 0
32 209 0 0 0 0 52 16 0 0
36 208 0 0 0 0 55 18 0 0
32 209 0 0 0 0 54 18 0 0
36 208 0 0 0 0 57 20 0 0
32 209 0 0 0 0 56 20 0 0
36 208 0 0 0 0 59 22 0 0
32 209 0 0 0 0 58 22 0 0
36 208 0 0 0 0 61 25 0 0
32 209 0 0 0 0 60 25 0 0
36 208 0 0 0 0 63 28 0 0
32 209 0 0 0 0 62 28 0 0
36 208 0 0 0 0 65 31 0 0
32 209 0 0 0 0 64 31 0 0
36 208 0 0 0 0 67 33 0 0
32 209 0 0 0 0 66 33 0 0
32 209 0 0 0 0 1 1 0 0

J
 53 13 68
```

C.1.4. PAST command file - mark1.pst

```
! start of target cycle
start start

! program counter variable
pc cr
```

C.1.5. The simulator - mark1.c

```
/* ISPS PAST V1.0 at Wed May 21 16:08:36 1986                              */
/*                                                                         */
/*      RTM file:                                                          */
/*              mark1.rtm                                                  */
/*      PST file:                                                          */
/*              mark1.pst                                                  */

#include "pastsim.h"

/*************************************************************************/
/*                                                                       */
/*      macros to update primary and iconnected variables                */
/*                                                                       */
/*************************************************************************/

#define _SET_f \
 f = (pi & 0x0000e000) >> 13;
#define _SET_s \
 s = pi & 0x00001fff;

/************** end of update macros.                     **************/

/*************************************************************************/
/*                                                                       */
/*      macros to propagate primary and oconnected variables             */
/*                                                                       */
/*************************************************************************/

#define _PROP_f \
 pi = (pi & 0xffff1fff) | (f << 13);
#define _PROP_s \
 pi = (pi & 0xffffe000) | s;

/************** end of propagate macros.                  **************/

/*************************************************************************/
/*                                                                       */
/*      PAST global variables                                            */
/*                                                                       */
/*************************************************************************/

/* loop counter variable.                                                */

int _i;

/* variables which hold mask bit and word boundaries.                    */

int _mlw, _mlb, _mhw, _mhb;
```

```
/* borrow and carry variables for subtraction and addition.             */

unsigned int _borrow, _carry;

/* fill variable for shift operations.                                  */

unsigned int _fill;

/* variables to hold sign bits of signed variables.                     */

unsigned int _sign1, _sign2;

/* temporary multiprecision variables.                                  */

unsigned int _temp[16];
unsigned int _temp1[8], _temp2[8];
unsigned int _temps1[8], _temps2[8];
unsigned int _temp3;

/* input/output port information.                                       */

FILE *_port[1];
char _portname[1][MAXNAME];
int _porttype[1];

/************** end of PAST global variables.            ***************/

/**********************************************************************/
/*                                                                    */
/*      User global variables - main                                  */
/*                                                                    */
/**********************************************************************/

unsigned int d_p_;
unsigned int w_p_[4];
unsigned int wait[4];
unsigned int undefined[4];
unsigned int f_o_p_[4];
unsigned int first_one;
unsigned int l_o_p_[4];
unsigned int last_one;
unsigned int p_p_[4];
unsigned int parity;
unsigned int is_running;
unsigned int m_l_p_1[4];
unsigned int m_l_p_2;
unsigned int mask_left[4];
unsigned int m_r_p_1[4];
unsigned int m_r_p_2;
unsigned int mask_right[4];
unsigned int t_w_p_1[4];
unsigned int t_w_p_2;
unsigned int time_wait[4];
unsigned int c_o_p_[4];
unsigned int count_one;
unsigned int pi;
unsigned int _t00037;
unsigned int m[8192];
unsigned int _t00040;
unsigned int acc;
unsigned int _t00043;
unsigned int cr;

/************** end of User global variables - main.     ***************/
```

```
/*********************************************************************/
/*                                                                   */
/*      User global variables - primary                              */
/*                                                                   */
/*********************************************************************/

unsigned int f;
unsigned int s;

/************** end of User global variables - primary.  **************/

/*********************************************************************/
/*                                                                   */
/*      labels for setjmp()/longjmp()                                */
/*                                                                   */
/*********************************************************************/

LABELTYPE Jstart;

/************** end of labels for setjmp()/longjmp().    **************/

/*********************************************************************/
/*                                                                   */
/*      _varinfo[] - user variable information                       */
/*                                                                   */
/*********************************************************************/

/* {name, variable, type, size, bc, left, right, incr, IC, OC, radix}   */

struct VarInfo _varinfo[] = {
 /*   0 */ { "d.p.", &d_p_, 1, 1, 18, 0, 0, 1, -1, -1, 16 },
 /*   1 */ { "w.p.", w_p_, 2, 4, 128, 0, 0, 1, -1, -1, 16 },
 /*   2 */ { "wait", wait, 2, 4, 128, 0, 0, 1, -1, -1, 16 },
 /*   3 */ { "undefined", undefined, 2, 4, 128, 0, 0, 1, -1, -1, 16 },
 /*   4 */ { "f.o.p.", f_o_p_, 2, 4, 128, 0, 0, 1, -1, -1, 16 },
 /*   5 */ { "first.one", &first_one, 1, 1, 8, 0, 0, 1, -1, -1, 16 },
 /*   6 */ { "l.o.p.", l_o_p_, 2, 4, 128, 0, 0, 1, -1, -1, 16 },
 /*   7 */ { "last.one", &last_one, 1, 1, 8, 0, 0, 1, -1, -1, 16 },
 /*   8 */ { "p.p.", p_p_, 2, 4, 128, 0, 0, 1, -1, -1, 16 },
 /*   9 */ { "parity", &parity, 1, 1, 1, 0, 0, 1, -1, -1, 16 },
 /*  10 */ { "is.running", &is_running, 1, 1, 1, 0, 0, 1, -1, -1, 16 },
 /*  11 */ { "m.l.p.1", m_l_p_1, 2, 4, 128, 0, 0, 1, -1, -1, 16 },
 /*  12 */ { "m.l.p.2", &m_l_p_2, 1, 1, 8, 0, 0, 1, -1, -1, 16 },
 /*  13 */ { "mask.left", mask_left, 2, 4, 128, 0, 0, 1, -1, -1, 16 },
 /*  14 */ { "m.r.p.1", m_r_p_1, 2, 4, 128, 0, 0, 1, -1, -1, 16 },
 /*  15 */ { "m.r.p.2", &m_r_p_2, 1, 1, 8, 0, 0, 1, -1, -1, 16 },
 /*  16 */ { "mask.right", mask_right, 2, 4, 128, 0, 0, 1, -1, -1, 16 },
 /*  17 */ { "t.w.p.1", t_w_p_1, 2, 4, 128, 0, 0, 1, -1, -1, 16 },
 /*  18 */ { "t.w.p.2", &t_w_p_2, 1, 1, 18, 0, 0, 1, -1, -1, 16 },
 /*  19 */ { "time.wait", time_wait, 2, 4, 128, 0, 0, 1, -1, -1, 16 },
 /*  20 */ { "c.o.p.", c_o_p_, 2, 4, 128, 0, 0, 1, -1, -1, 16 },
 /*  21 */ { "count.one", &count_one, 1, 1, 8, 0, 0, 1, -1, -1, 16 },
 /*  22 */ { "pi", &pi, 1, 1, 16, 0, 0, 1, -1, -1, 16 },
 /*  23 */ { "f", &f, 1, 1, 3, 0, 0, 1, -1, -1, 16 },
 /*  24 */ { "m", m, 3, 1, 32, 0, 8191, 1, -1, -1, 16 },
 /*  25 */ { "s", &s, 1, 1, 13, 0, 0, 1, -1, -1, 16 },
 /*  26 */ { "acc", &acc, 1, 1, 32, 0, 0, 1, -1, -1, 16 },
 /*  27 */ { "cr", &cr, 1, 1, 13, 0, 0, 1, -1, -1, 16 },
};      /* _varinfo[] */
```

```
/* number of variables in _varinfo[] array.                          */

int _varcount = 28;

/* program counter variable number and address.                      */

int _pcindex = 27;      /* program counter is cr. */
int _pcaddr = 0;        /* address into pc variable. */

/************** end of _varinfo[].                    **************/

/*******************************************************************/
/*                                                                 */
/*      preset() - initialize memories.                            */
/*                                                                 */
/*******************************************************************/

preset()
{
}

/************** end of preset().                      **************/

/*******************************************************************/
/*                                                                 */
/*      InitChannels() - opens default input/output channels       */
/*                                                                 */
/*******************************************************************/

InitChannels()
{
}       /* InitChannels() */

/************** end of InitChannels().                **************/

/*******************************************************************/
/*                                                                 */
/*      Update()/Propagate() - update/propagate primary variables  */
/*                                                                 */
/*******************************************************************/

Update(_vindex, _addr)
int _vindex;
unsigned int _addr;
{
 switch(_vindex) {
  case 23:      _SET_f; break;
  case 26:      _SET_s; break;
  default:      break;
 }      /* switch(_vindex) */
}       /* Update(_vindex) */

Propagate(_vindex, _addr)
int _vindex;
unsigned int _addr;
{
 switch(_vindex) {
  case 23:      _PROP_f;        break;
  case 26:      _PROP_s;        break;
  default:      break;
 }      /* switch(_vindex) */
}       /* Propagate(_vindex) */

/************** end of Update()/Propagate().          **************/
```

```
/**********************************************************************/
/*                                                                    */
/*      MainLoop() - main loop for general simulator                  */
/*                                                                    */
/**********************************************************************/

MainLoop()
{
 LABEL(Jstart, EndLabel);
StartLabel:
 UserInterface();
 _t00037 = m[cr];
 pi = (_t00037 & 0x0000ffff);
 _SET_f;
 switch (f) {
  case 0x00000000:
   _SET_s;
   cr = m[s];
   break;
  case 0x00000001:
   _SET_s;
   _t00037 = m[s];
   _temps1[0] = cr;
   _fill = (_temps1[0] & 0x00001000) ? 0xffffffff : 0;
   _temps1[0] |= _fill & 0xffffe000;
   cr += _t00037;
   cr &= 0x00001fff;
   break;
  case 0x00000002:
   _SET_s;
   _t00037 = m[s];
   acc = ~_t00037;
   acc++;
   break;
  case 0x00000003:
   _SET_s;
   m[s] = acc;
   break;
  case 0x00000004:
  case 0x00000005:
   _SET_s;
   _t00040 = m[s];
   acc -= _t00040;
   break;
  case 0x00000006:
   _t00043 = (_sign1 = acc & 0x80000000, _sign2 = 0,
     ((_sign1 && !_sign2)
     || (_sign1 && (0x00000000 < acc))
     || (!_sign2 && (acc < 0x00000000))));
   if (_t00043) {
    cr += 0x00000001;
    cr &= 0x00001fff;
   }
   break;
  case 0x00000007:
   stop();
   break;
 }      /* switch() */
 cr += 0x00000001;
 cr &= 0x00001fff;
 goto StartLabel;
EndLabel:       ;
}       /* MainLoop() */

/************** end of MainLoop().                    **************/
```

```
/*********************************************************************/
/*                                                                  */
/*      Subroutines                                                 */
/*                                                                  */
/*********************************************************************/

/************** end of Subroutines.                   **************/

/*********************************************************************/
/*                                                                  */
/*      Statistics                                                  */
/*                                                                  */
/*********************************************************************/

/* Total RTM operations processed:  32                              */
/*                                                                  */
/* Finished at Wed May 21 16:08:39 1986.                            */
```

C.1.6. Running the architecture-specific Mark-1 simulator

Below is a sample run of the architecture-specific Mark-1 simulator generated by PAST.

```
Script started on Thu Jun 26 17:09:31 1986

% mark1 -E mult.psim
cr=00000000> value ?
Must match one of these:
        d.p.            last.one        m.r.p.1         count.one
        w.p.            p.p.            m.r.p.2         pi
        wait            parity          mask.right      f
        undefined       is.running      t.w.p.1         m
        f.o.p.          m.l.p.1         t.w.p.2         s
        first.one       m.l.p.2         time.wait       acc
        l.o.p.          mask.left       c.o.p.          cr
cr=00000000> value m 25
m[37] = "00000135
cr=00000000> value m 26
m[38] = "00000007
cr=00000000> value m 27
m[39] = "00000000
cr=00000000> set m 25 321
cr=00000000> set m 26 12
cr=00000000> value m 25
m[37] = "00000321
cr=00000000> value m 26
m[38] = "00000012
cr=00000000> step 100
cr=0000000e> value m 25
m[37] = "00000321
cr=0000000e> value m 26
m[38] = "00000012
cr=0000000e> value m 27
m[39] = "000015e7
cr=0000000e> run
Exiting simulation loop - simulation ended normally.
cr=0000001b> value m 27
m[39] = "00003852
cr=0000001b> quit
Cycles executed:  246.
% exit
%
```

C.2. Program-specific simulator

A Mark-1 simulator specific to the multiplication program that we used as our test case was prepared from the ISPS description as follows:

```
/usr/bss/bin/ispc mark1
/usr/bss/bin/past mark1 -o mult.c -p mult.pst
cc -c mult.c
cc -o mult mult.o -lpast
```

The following sections show the files involved in generating the program-specific simulator that differ from those involved in generating the architecture-specific simulator. The ISPS description, GDB file, and RTM file are the same for both architecture-specific and program-specific simulators. Also, the same sequence of commands can be run on both types of simulators with identical results.

C.2.1. Multiplication program

```
                        ;
                        ; Program to multiply two numbers by repeated additions.
                        ;
                        ;       Source operands are in X and Y.
                        ;       Result ends up in R.
                        ;

                        ; if (y < 0) {
                        ;     nflag = TRUE;
                        ;     y = -y;
                        ; }

00: 40 26                       ldn Y           ; A <- -Y
01: 60 23                       sto N3
02: c0 00                       cmp             ; if A lss 0 (y > 0) then L2 else L1
03: 00 1c                       jmp L1
04: 00 1d                       jmp L2
05: 60 22               L1:     sto N2          ; nflag = TRUE;
06: 40 23                       ldn N3          ; y = -y;
07: 60 23                       sto N3

                        ; for (i = y - 1; i >= 0; i--)
                        ;     r = r + x;

08: 40 23               L2:     ldn N3          ; A <- YTemp
09: 80 21                       sub N1          ; A <- YTemp - 1
0a: c0 00                       cmp             ; if A lss 0 then L4 else L3
0b: 00 1e                       jmp L3
0c: 00 1f                       jmp L4
0d: 60 24               L3:     sto N4          ; YTemp = A = YTemp - 1
0e: 40 24                       ldn N4
0f: 60 23                       sto N3
```

```
                        ;
                        ;       R = R + X
                        ;

10: 40 27                       ldn R           ; A <- -R
11: 80 25                       sub X           ; A <- -R - X
12: 60 27                       sto R           ; R = -R
13: 40 27                       ldn R
14: 60 27                       sto R

15: 00 1d                       jmp L2

                        ;
                        ; if (nflag == TRUE)
                        ;     R = -R;
                        ;

16: 40 22               L4:     ldn N2
17: c0 00                       cmp
18: 00 20                       jmp L5
19: 40 27                       ldn R
1a: 60 27                       sto R

1b: e0 00               L5:     stop            ; end of program

                        ; label values are jump location - 1 because the PC
                        ; is incremented after the jump.

1c: 00000004            labels: L1
1d: 00000007                    L2
1e: 0000000c                    L3
1f: 00000015                    L4
20: 0000001a                    L5
21: 00000001            N1:     1
22: 00000000            N2:     negative flag
23: 00000000            N3:     Ytemp
24: 00000000            N4:     not used
25: 00000135            X:      source          ; 0x0135 * 0x0007
26: 00000007            Y:      source
27: 00000000            R:      result
```

C.2.2. PAST command file - mult.pst

```
radix hex
setval m[00] = 4026
setval m[01] = 6023
setval m[02] = c000
setval m[03] = 001c
setval m[04] = 001d
setval m[05] = 6022
setval m[06] = 4023
setval m[07] = 6023
setval m[08] = 4023
setval m[09] = 8021
setval m[0a] = c000
setval m[0b] = 001e
setval m[0c] = 001f
setval m[0d] = 6024
setval m[0e] = 4024
setval m[0f] = 6023
setval m[10] = 4027
setval m[11] = 8025
setval m[12] = 6027
setval m[13] = 4027
setval m[14] = 6027
setval m[15] = 001d
setval m[16] = 4022
setval m[17] = c000
setval m[18] = 0020
setval m[19] = 4027
setval m[1a] = 6027
setval m[1b] = e000
setval m[1c] = 00000004
setval m[1d] = 00000007
setval m[1e] = 0000000c
setval m[1f] = 00000015
setval m[20] = 0000001a
setval m[21] = 00000001
setval m[22] = 00000000
setval m[23] = 00000000
setval m[24] = 00000000
setval m[25] = 00000135
setval m[26] = 00000007
setval m[27] = 00000000
set cr = 0

! set the program counter variable to be cr.
pc cr

! code range.
code 0:1b

! static range.
static m[0:21]

! start of cycle.
start start
```

C.2.3. The simulator - mult.c

```
/* ISPS PAST V1.0 at Wed May 21 17:28:51 1986                           */
/*                                                                      */
/*      RTM file:                                                       */
/*              mark1.rtm                                               */
/*      PST file:                                                       */
/*              mult.pst                                                */

                                .
                                .
                                .

                /* the initial definition and        */
                /* declaration sections are the same */
                /* for both architecture-specific and */
                /* program-specific simulators.  See  */
                /* the architecture-specific simulator. */

                                .
                                .
                                .

/************** end of _varinfo[].                      **************/

/**********************************************************************/
/*                                                                    */
/*      preset() - initialize registers and memories.                 */
/*                                                                    */
/**********************************************************************/

preset()
{
 m[0] = 0x00004026;
 m[1] = 0x00006023;
 m[2] = 0x0000c000;
 m[3] = 0x0000001c;
 m[4] = 0x0000001d;
 m[5] = 0x00005022;
 m[6] = 0x00004023;
 m[7] = 0x00006023;
 m[8] = 0x00004023;
 m[9] = 0x00008021;
 m[10] = 0x0000c000;
 m[11] = 0x0000001e;
 m[12] = 0x0000001f;
 m[13] = 0x00006024;
 m[14] = 0x00004024;
 m[15] = 0x00006023;
 m[16] = 0x00004027;
 m[17] = 0x00008025;
 m[18] = 0x00006027;
 m[19] = 0x00004027;
 m[20] = 0x00006027;
 m[21] = 0x0000001d;
 m[22] = 0x00004022;
 m[23] = 0x0000c000;
 m[24] = 0x00000020;
 m[25] = 0x00004027;
 m[26] = 0x00006027;
 m[27] = 0x0000e000;
 m[28] = 0x00000004;
 m[29] = 0x00000007;
 m[30] = 0x0000000c;
 m[31] = 0x00000015;
 m[32] = 0x0000001a;
 m[33] = 0x00000001;
 m[34] = 0x00000000;
 m[35] = 0x00000000;
 m[36] = 0x00000000;
 m[37] = 0x00000135;
 m[38] = 0x00000007;
 m[39] = 0x00000000;
 cr = 0x00000000;
}

/************** end of preset().                               **************/

/***************************************************************************/
/*                                                                         */
/*      InitChannels() - opens default input/output channels               */
/*                                                                         */
/***************************************************************************/

InitChannels()
{
}       /* InitChannels() */

/************** end of InitChannels().                         **************/

/***************************************************************************/
/*                                                                         */
/*      Update()/Propagate() - update/propagate primary variables          */
/*                                                                         */
/***************************************************************************/

Update(_vindex, _addr)
int _vindex;
unsigned int _addr;
{
 switch(_vindex) {
  case  23:     _SET_f; break;
  case  26:     _SET_s; break;
  default:      break;
 }      /* switch(_vindex) */
}       /* Update(_vindex) */

Propagate(_vindex, _addr)
int _vindex;
unsigned int _addr;
{
 switch(_vindex) {
  case  23:     _PROP_f;        break;
  case  26:     _PROP_s;        break;
  default:      break;
 }      /* switch(_vindex) */
}       /* Propagate(_vindex) */

/************** end of Update()/Propagate().                   **************/
```

```
/***********************************************************************/
/*                                                                     */
/*      MainLoop() - main loop for program-specific simulator          */
/*                                                                     */
/***********************************************************************/

MainLoop()
{
 while(1) {
  UserInterface(30);
  MainSwitch();
 }      /* while(1) */
}       /* MainLoop() */

MainSwitch()
{
 switch(cr) {
  case 0:
   _t00037 = 0x00004026;
   pi = 0x00004026;
   _t00037 = m[38];
   acc = ~_t00037;
   acc++;
   cr = 0x00000001;
   break;
  case 1:
   _t00037 = 0x00006023;
   pi = 0x00006023;
   m[35] = acc;
   cr = 0x00000002;
   break;
  case 2:
   _t00037 = 0x0000c000;
   pi = 0x0000c000;
   _t00043 = (_sign1 = acc & 0x80000000, _sign2 = 0,
     ((_sign1 && !_sign2)
     || (_sign1 && (0x00000000 < acc))
     || (!_sign2 && (acc < 0x00000000))));
   if (_t00043) {
    cr = 0x00000003;
   }
   cr += 0x00000001;
   cr &= 0x00001fff;
   break;
  case 3:
   _t00037 = 0x0000001c;
   pi = 0x0000001c;
   cr = 0x00000004;
   cr = 0x00000005;
   break;
  case 4:
   _t00037 = 0x0000001d;
   pi = 0x0000001d;
   cr = 0x00000007;
   cr = 0x00000008;
   break;
  case 5:
   _t00037 = 0x00006022;
   pi = 0x00006022;
   m[34] = acc;
   cr = 0x00000006;
   break;
```

```
case 6:
 _t00037 = 0x00004023;
 pi = 0x00004023;
 _t00037 = m[35];
 acc = ~_t00037;
 acc++;
 cr = 0x00000007;
 break;
case 7:
 _t00037 = 0x00006023;
 pi = 0x00006023;
 m[35] = acc;
 cr = 0x00000008;
 break;
case 8:
 _t00037 = 0x00004023;
 pi = 0x00004023;
 _t00037 = m[35];
 acc = ~_t00037;
 acc++;
 cr = 0x00000009;
 break;
case 9:
 _t00037 = 0x00008021;
 pi = 0x00008021;
 _t00040 = 0x00000001;
 acc -= 0x00000001;
 cr = 0x0000000a;
 break;
case 10:
 _t00037 = 0x0000c000;
 pi = 0x0000c000;
 _t00043 = (_sign1 = acc & 0x80000000, _sign2 = 0,
    ((_sign1 && !_sign2)
    || (_sign1 && (0x00000000 < acc))
    || (!_sign2 && (acc < 0x00000000))));
 if (_t00043) {
  cr = 0x0000000b;
 }
 cr += 0x00000001;
 cr &= 0x00001fff;
 break;
case 11:
 _t00037 = 0x0000001e;
 pi = 0x0000001e;
 cr = 0x0000000c;
 cr = 0x0000000d;
 break;
case 12:
 _t00037 = 0x0000001f;
 pi = 0x0000001f;
 cr = 0x00000015;
 cr = 0x00000016;
 break;
case 13:
 _t00037 = 0x00006024;
 pi = 0x00006024;
 m[36] = acc;
 cr = 0x0000000e;
 break;
```

```
case 14:
 _t00037 = 0x00004024;
 pi = 0x00004024;
 _t00037 = m[36];
 acc = ~_t00037;
 acc++;
 cr = 0x0000000f;
 break;
case 15:
 _t00037 = 0x00006023;
 pi = 0x00006023;
 m[35] = acc;
 cr = 0x00000010;
 break;
case 16:
 _t00037 = 0x00004027;
 pi = 0x00004027;
 _t00037 = m[39];
 acc = ~_t00037;
 acc++;
 cr = 0x00000011;
 break;
case 17:
 _t00037 = 0x00008025;
 pi = 0x00008025;
 _t00040 = m[37];
 acc -= _t00040;
 cr = 0x00000012;
 break;
case 18:
 _t00037 = 0x00006027;
 pi = 0x00006027;
 m[39] = acc;
 cr = 0x00000013;
 break;
case 19:
 _t00037 = 0x00004027;
 pi = 0x00004027;
 _t00037 = m[39];
 acc = ~_t00037;
 acc++;
 cr = 0x00000014;
 break;
case 20:
 _t00037 = 0x00006027;
 pi = 0x00006027;
 m[39] = acc;
 cr = 0x00000015;
 break;
case 21:
 _t00037 = 0x0000001d;
 pi = 0x0000001d;
 cr = 0x00000007;
 cr = 0x00000008;
 break;
case 22:
 _t00037 = 0x00004022;
 pi = 0x00004022;
 _t00037 = m[34];
 acc = ~_t00037;
 acc++;
 cr = 0x00000017;
 break;
```

```
  case 23:
   _t00037 = 0x0000c000;
   pi = 0x0000c000;
   _t00043 = (_sign1 = acc & 0x80000000, _sign2 = 0,
     ((_sign1 && !_sign2)
     || (_sign1 && (0x00000000 < acc))
     || (!_sign2 && (acc < 0x00000000))));
   if (_t00043) {
    cr = 0x00000018;
   }
   cr += 0x00000001;
   cr &= 0x00001fff;
   break;
  case 24:
   _t00037 = 0x00000020;
   pi = 0x00000020;
   cr = 0x0000001a;
   cr = 0x0000001b;
   break;
  case 25:
   _t00037 = 0x00004027;
   pi = 0x00004027;
   _t00037 = m[39];
   acc = ~_t00037;
   acc++;
   cr = 0x0000001a;
   break;
  case 26:
   _t00037 = 0x00006027;
   pi = 0x00006027;
   m[39] = acc;
   cr = 0x0000001b;
   break;
  case 27:
   _t00037 = 0x0000e000;
   pi = 0x0000e000;
   stop();
   break;
  default:
   fprintf(stderr,
    "cr = %d:  the program counter is outside of the specified code range.\n",
    cr);
   longjmp(JStart, PCOUTOFRANGE);
   break;
 }       /* switch(cr) */
}       /* MainSwitch() */

/************** end of MainLoop().                            **************/

/**********************************************************************/
/*                                                                    */
/*      Statistics                                                    */
/*                                                                    */
/**********************************************************************/

/* Total RTM operations processed:  274                               */
/*                                                                    */
/* Finished at Wed May 21 17:28:56 1986.                              */
```

Appendix D
Problems with the C compiler

We encountered numerous problems with the *cc* C compiler when we tried to compile simulators produced by PAST. To circumvent some of the compiler's problems, we had to determine what its limits were. We wrote several C programs to test the limits, and have summarized our findings in Table D-1. Some of these limits are probably quite easy to vary if one has access to the source code for the *cc* compiler, but we do not have access to that code.

Limited parameter	Limit	Error Message
Number of distinct cases in a switch statement	499	compiler error: switch table overflow
Number of cases which can drop through to same code in a switch statement	137	yacc stack overflow
Number of C functions	2999	compiler error: symbol table full
Jump size	?	brw: Branch too far: try -J flag (assembler error)

Table D-1: Limitations of the *cc* C compiler.

References

[1] Alfred V. Aho, Jeffrey D. Ullman.
Principles of Compiler Design.
Addison-Wesley Publishing Company, Reading, Massachusetts, 1984.

[2] Benjamin Atlas.
Mixed Level Functional Specification: A Modeling Methodology for Computer System Simulation.
PhD thesis, Carnegie-Mellon University, August, 1985.
SRC Research Report CMUCAD-85-64.

[3] Mario R. Barbacci, Gary E. Barnes, Roderic G. Cattell, Daniel P. Siewiorek.
The ISPS Computer Description Language.
Manual, Departments of Computer Science and Electrical Engineering Carnegie-Mellon University, August, 1979.

[4] Mario R. Barbacci.
The Register Transfer Machine.
Technical Report, Computer Science Department Carnegie-Mellon University, September, 1979.

[5] Mario Barbacci, Andrew W. Nagle, J. Duane Northcutt.
An ISPS Simulator.
Manual, Departments of Computer Science and Electrical Engineering Carnegie-Mellon University, January, 1980.

[6] Mario R. Barbacci, Daniel P. Siewiorek.
The Design and Analysis of Instruction Set Processors.
McGraw-Hill Book Company, New York, 1982.

[7] C. Gordon Bell, Allen Newell.
Computer Structures: Reading and Examples.
McGraw-Hill, Inc., 1971.

[8] Randy Bryant, Mike Schuster, Doug Whiting.
MOSSIM II: A Switch-Level Simulator for MOS LSI - User's Manual
10 January 85 edition, Carnegie-Mellon University, 1985.

[9] C. H. Chang, Y. Shintani, P. J. Lieu.
Warp System Simulator User's Manual
Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213, 1985.

[10] Daisy Systems Corporation.
LOGICIAN User's Guide
Daisy Systems Corporation, Mountain View, California 94039, 1985.

[11] Jack W. Davidson.
Fast Interpretation of Instruction Sets: Implementation and Applications.
In C. J. Koomen and T. Moto-oka (editor), *Computer Hardware Description Languages and their applications*, pages 179-191. Kyoto, Japan, August, 1985.

[12] A. K. Dewdney.
A computer microscope zooms in for a look at the most complex object in mathematics.
Scientific American :16-24, August, 1985.

[13] *Digital Logic Handbook*
Digital Equipment Corporation, 1968.

[14] *VAX Architecture Handbook*
Digital Equipment Corporation, 1981.

[15] Roy L. Druian.
Functional Models for VLSI Design.
In *Twentieth Design Automation Conference Proceedings*, pages 506-514. IEEE, 1983.

[16] David John Geiger.
A New Look at Algorithmic Simulation.
PhD thesis, Carnegie-Mellon University, June, 85.
Ph.D. Thesis Proposal, June 5, 1986.

[17] Dwight Hill, Willem vanCleemput.
SABLE: A Tool for Generating Structured, Multi-level Simulations.
In *Sixteenth Design Automation Conference Proceedings*. IEEE, 1979.

[18] Beatriz Infante, Mark Bales, Ed Lock.
MADL: A Language for Describing Mixed Behavior and Structure.
In T. Uehara and M. Barbacci (editor), *Computer Hardware Description Languages and their Applications*, pages 115-126. 1983.

[19] Neil D. Jones.
Towards Automating the Transformation of Programming Language Specifications into Compilers.
unpublished , January, 1986.
Unpublished draft dated January 1986.

[20] Brian W. Kernighan and Dennis M. Ritchie.
The C Programming Language.
Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1978.

[21] H. T. Kung, Onat Menzilcioglu.
Design Specifications for the CMU Warp Processor
Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA 15213, August 21, 1984.

[22] Karl J. Lieberherr.
Toward a Standard Hardware Description Language.
IEEE Design & Test :55-62, February, 1985.

[23] J. A. Nestor, D. E. Thomas.
Defining and Implementing a Multilevel Design Representation.
In *Nineteenth Design Automation Conference Proceedings*, pages 740-746. IEEE, 1982.

[24] Greg M. Ordy, Charles W. Rose.
The N.2 System.
In *Twentieth Design Automation Converence Proceedings*, pages 520-526. 1983.

[25] Adam Osborne, Gerry Kane.
Osborne 4 & 8-bit Microprocessor Handbook.
Osborne/McGraw Hill, Berkeley, CA, 1981.

[26] Charles W. Rose, Greg M. Ordy, Frederic I. Parke.
N.mPC: A Retrospective.
In *Twentieth Design Automation Conference Proceedings*, pages 497-505. IEEE, 1983.

[27] Charles W. Rose, Greg M. Ordy, Paul J. Drongowski.
N.mPc: A Study in University-Industry Technology Transfer.
IEEE Design & Test of Computers 1(1):44-56, February, 1984.

[28] Jay W. Schooley.
Translation and Instrumentation Of An ISPS Simulator.
Master's thesis, Carnegie-Mellon University, August, 1985.
SRC Report Number CMUCAD-85-62.

[29] Richard J. Selvaggi.
A Parallel ISPS Simulator for a Multiprocessor.
Master's thesis, SRC-CMU Center for Computer-Aided Design, Department of Electrical and Computer Engineering, Carnegie-Mellon University, February, 1986.

[30] Daniel P. Siewiorek, C. Gordon Bell, Allen Newell.
Computer Structures: Principles and Examples.
McGraw-Hill Book Company, New York, 1982.

[31] A. Vladimirescu, Kaihe Zhang, A. R. Newton, D. O. Pederson, A. Sangiovanni-Vincentelli.
SPICE Version 2G User's Guide
Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, CA, 94720, 1981.

[32] *ZYCAD Intermediate Form Reference Manual, Volume 1: Logic Evaluator*
ZYCAD Corporation, 1985.