NASA-CR-191327

SBIR-05.03-1100
Release Date 4/26/94

EAI® Electronic Associates, Inc. p.58
185 Monmouth Parkway, West Long Branch, N. J. 07764 (908) 229-1100

Final Report NASA Contract NAS5-30905 Construction of a Parallel Processor for Simulating Manipulators and Other Mechanical Systems.

Electronic Associates, Inc.
August 20, 1991

ABSTRACT

This report summarizes the results of NASA Contract NAS5-30905, awarded under Phase 2 of the SBIR Program, for a demonstration of the feasibility of a new high-speed parallel simulation processor, called the RTA (Real-Time Accelerator). The principal goals were met, and EAI is now proceeding with Phase 3: development of a commercial product. This product is scheduled for commercial introduction in the second quarter of 1992.

CONTENTS

George Hannauer
George Hannauer
Principal Investigator

Jack Budelman
Jack Budelman
Vice President, Product Engineering

(NASA-CR-191327) CONSTRUCTION OF A PARALLEL PROCESSOR FOR SIMULATING MANIPULATORS AND OTHER MECHANICAL SYSTEMS Final Report, Apr. 1990 - Aug. 1991 (Electronic Associates) 58 p

N94-32482

Unclas

G3/62 0010553

1. INTRODUCTION

For over 35 years, EAI has been designing and building the world's fastest simulation machines. Initially they were analog; later, hybrid. Several years ago, the company undertook a feasibility study to determine what performance could be achieved with a new simulation system, based on an analog-type architecture, but implemented digitally. The result is the RTA.

Theoretical analysis and manual simulation led to the conclusion that such a system would be ten to one hundred times faster than existing machines for the vast majority of the target simulation applications, at considerably less cost.

In 1990, under NASA Contract NAS5-30905, EAI began work on a prototype hardware system and a Continuous System Simulation Language (CSSL) compiler, to verify the design concepts. That contract effort is now complete. It meets or exceeds all its major goals. For the last several months, it has been outperforming other machines *in hardware*, as well as on paper.

The Applications Base

The starting point for the study was a set of benchmark applications that were used to evaluate the design. These included a six-degree-of-freedom missile launch simulation and a small, but very stiff, chemical kinetics application, both taken from our customer files. To these, we added several applications in robotics, furnished by NASA. These applications were chosen to be a representative sample of the ones that our customers have been running on our simulation computers for the past 35 years. They share the following characteristics:

They are all applications in Continuous Systems Simulation—mathematically speaking, initial-value problems in ordinary differential equations. They typically require high speed, but only moderate accuracy (a few percent at several hundred Hertz). They are often stiff (a wide range of eigenvalues), and usually include hardware in the loop, necessitating real-time operation. Most contain switching transients (discontinuities), so that a full set of logical, relational, and switching operations must be made available and efficiently implemented.

We have run over a dozen benchmarks on the prototype hardware, including several that were *not* in the original set. These additional applications, brought to us by potential customers after the study began, provided an important control. If the system had shown a great speed advantage for the original benchmarks, and a small advantage or none at all for the new ones, it would mean that the design had been too strongly "tailored" to the specific examples, and was not robust enough for general-purpose use. What was hoped, of course, was that the system would exhibit essentially the same competitive advantage for the new benchmarks as for the original ones, verifying that the latter had been selected well. This hope has been realized, as the results in section 3 below clearly show.

2. GOALS OF THE PHASE 2 PROJECT

The goals of phase 2 of this study are succinctly summarized in the Technical Abstract of the Phase 2 proposal summary:

> "The objective of Phase 2 is to verify the Real-Time Accelerator (RTA) concept and the timing estimates by building a prototype. EAI plans to build such a prototype and run several simulations on it, verifying both the programmability and the speed of the proposed system. It is expected that the prototype will fulfill its expectations of a tenfold-to-hundredfold speedup over conventional computers. If so, EAI plans to pursue sources of non-Government venture capital to develop it as a commercial product."

This report describes the results of the Phase 2, including the running of the benchmarks, the speed comparisons with conventional machines, and EAI's future development plans.

3. RESULTS OF THE PHASE 2 PROJECT

The major goals, as spelled out in the previous section, have been accomplished:

• The prototype RTA system has been developed and is being shipped to NASA, as per contract. Section 4 of this report provides a user's guide for the programming and operation of this prototype system.

• All three applications have been programmed and successfully run. The execution times met expectations: the RTA runs these applications ten to one hundred times faster than currently available machines.

• EAI has obtained non-government funding for the third phase of this project: the development of a commercial product.

3.1 Description of Benchmarks.

The Phase 2 proposal listed three benchmark applications to be run as part of the evaluation. All have been programmed and run, and the source files and executable files are provided on the disk shipped with the prototype system. The three benchmarks include a chemical engineering application, an aerospace application, and a robotics application:

Armstrong Cork: A small, but very stiff, chemical kinetics application, obtained from the Armstrong Cork Corporation.

Spinning Missile: A six-degree-of-freedom simulation of the boost phase of a spin-stabilized missile, obtained from the U.S. Army Missile Command, Redstone Arsenal, Huntsville, Alabama.

Robot: A four-degree-of-freedom simulation of a robotic manipulator arm, obtained from NASA, Goddard Space Flight Center, Greenbelt, Maryland., in conjunction with Dr. Roger Chen, University of Maryland.

All three of these benchmarks have been successfully run, and the resulting RTA programs are provided in both source and executable form on the disk to be delivered with the prototype system in fulfillment of the contract. In addition, seven other benchmarks, not part of the original set, have been run, and are included on the prototype disk. The benchmarks provided are listed by filename below:

SHENG: A simulation of the space shuttle main rocket engine, furnished by NASA Manned Spaceflight Center, Huntsville, Alabama. This application has a large function generation requirement—thirty functions of one variable, and three functions of two variables. It is the largest benchmark of the ten, both in terms of the number of computations required for a derivative evaluation, and in terms of frame time.

Since functions of two variables were not included in the original project plan, additional software effort (not charged to the project) was required to develop the necessary algorithms to run the shuttle engine. The macros for functions of two variables, and for unequally-spaced breakpoint search, are included in the macro libraries on the disk shipped with the prototype.

SPMIS: The Spinning Missile benchmark, described earlier.

ARM: The Armstrong Cork chemical benchmark, described earlier.

BOUNCE: A simulation of semi-elastic collisions—the classical "bouncing ball" problem. While small, this application demonstrates an important capability: the use of mode-controlled integrators.

CHAOS: A simple model of long-term weather cycles, illustrating extreme sensitivity to initial conditions, proposed by Lorentz of MIT.

VCO: A Voltage-Controlled Oscillator. This is simply an oscillator with a variable frequency, set (in this example) to sweep over a five-to-one range of frequencies. The resulting solution, displayed on an oscilloscope, exhibits very good accuracy for a real-time frequency sweep from 1 to 5 kHz., with no visible amplitude growth or decay over six cycles of oscillation. When speeded up to sweep from 2 to 10 kHz., there is a barely noticeable amplitude growth. Even over a frequency range from 4 to 20 kHz., the amplitude growth is only a few percent over six cycles. The solution compares favorably with analog results.

DPEND: A simulation of a Double Pendulum. This simulation is described in detail in the Phase 1 final report, and the description is repeated as Section 5 of the present report. It is complex enough to exemplify the inherent coupling and nonlinear characteristics of manipulator dynamics, yet simple enough to be analyzed in detail. At 76 lines of source code, it is the largest simulation ever scheduled *manually* on the RTA. The manual schedule, produced before any software was developed, corresponds very closely with the results actually obtained with the compiler's scheduling algorithm.

ROBOT: The four-degree-of-freedom robot arm, provided by NASA, Goddard as one of the original benchmark set. It is the second largest benchmark, after the shuttle engine.

HTTRA and **MUTSU**. Two nuclear reactor simulations, requiring a very wide range of values to be accurately represented.

3.2 Evaluation of RTA Performance.

The key questions to be resolved during this study were:

1) Can the RTA solve the problems and generate correct answers?
2) Can the compiler generate an efficient RTA schedule from a high-level source program, without burdening the user with the details of the machine's architecture?
3) How fast will the resulting program run on the RTA?
4) How does the RTA speed compare with other machines?

The answer to the first question is "Yes." All ten benchmarks (the original three and seven additional ones) have been run on the hardware, and the results compared with solutions obtained on other systems. The "other systems" used for comparison include several conventional digital computers running FORTRAN and CSSL, and analog/hybrid computers. In all cases, the RTA results were in substantial agreement with the results obtained by other means.

The answer to the second question is also "Yes." The programs were all written in CSSL, and compiled and run using a scheduling compiler developed on this project. The program does *not* require detailed knowledge of the details of the machine's architecture; in fact, the program is essentially a listing of the model equations. Section 5 provides an analysis of one of these benchmarks, including a complete source listing and a description of the detailed schedule.

As for speed, the RTA is, as expected, ten to 100 times faster than currently available machines. The benchmark results are summarized in the following table:

BENCH-MARK	CRITICAL PATH LIMIT (CYCLES)	RESOURCE LIMIT (CYCLES)	COMPILER SCHEDULE (CYCLES)	EFFIC-IENCY (%)	FRAME TIME (μsec.)
SHENG	220	954	1015	94.0	61.00
SPMIS	125	339	354	95.8	25.80
ARM	25	50	51	98.0	4.75
BOUNCE	20	20	22	90.9	3.20
CHAOS	25	27	28	96.4	2.85
VCO	15	18	19	94.7	2.40
DPEND	115	98	128	89.8	8.10
ROBOT	295	609	615	99.0	33.45
HTTRA	65	346	347	99.7	27.80
MUTSU	50	109	110	99.1	10.40

Table 3.1 Results of RTA Benchmark Compilation and Scheduling

This table not only gives the "bottom line" speed results (the last column) but also information about the performance of the scheduling compiler. The statistics in this table are automatically generated by the compiler and made available on the listing file.

Although in normal operation, the user is concerned only with the last column (which determines the solution speed), the other information was useful during this project for evaluating compiler performance, and thus allowing the compiler to be "tuned" for efficiency. The meaning of the terms is as follows:

The critical path limit is the number of cycles required to perform the longest path of dependent calculations in the derivative evaluation program for a given application. Since each computation in this critical path depends on the result of its predecessor, there is no way that any of these computations can be performed in parallel. Hence the derivative evaluation *must* take at least this many cycles of computer time, even if an unlimited number of processors were available.

The resource limit is, roughly, the total amount of computation to be performed divided by the number of processors available. It represents the minimum amount of time to perform the derivative evaluation with a given number of processors, even if there were no dependency constraints, so that all computations were allowed to proceed in parallel. The "total amount of computation" is calculated by an algorithm that is sophisticated enough to recognize that different computations take different amounts of time, and that different computations may require different types of processors. With a given number of processors, the derivative evaluation must take *at least* this much time, and might take longer.

Both the resource limit and the critical path limit establish lower bounds on the amount of time required for a given derivative evaluation. The critical path limit is established by concentrating on the data dependencies and ignoring resource constraints—assuming an unbounded number of processors. The resource limit ignores the data dependencies and concentrates on the amount of computation to be performed and the number of processors available. The *larger* of these two times provides an absolute lower bound—no program can perform the derivative evaluation in less time than this.

As an example, consider the Armstrong Cork benchmark. Its critical path limit, according to Table 3.1, is 25 cycles, meaning that there is at least one chain of dependent calculations that takes this long. There is assumed to be only one processor available. There are 44 operations to be performed, all of which are one-cycle operations, requiring a total of 44 cycles for computation. The scheduler adds 6 cycles of overhead to this to allow for the pipeline latency, and reports the resource limit as 50 cycles. If two processors had been available, the resource limit would have been 28 cycles (44/2 + 6). With three processors, the resource limit would have been 21 cycles (44/3 + 6, rounded up to the next higher integer).

With one processor, the computation must take at least max(25, 50) = 50 cycles. With two processors, it must take at least max(25, 28) = 28 cycles, and with three processors, it must take at least max(25, 21) = 25. If these values can, in fact, be attained, it is clear that a second processor would speed up the Armstrong Cork program by almost a factor of two, whereas a third processor would provide relatively little additional speedup, since the 25-cycle critical path limit dominates.

The actual schedule performs the derivative evaluation in 51 cycles, only one more than the theoretical limit. Thus the efficiency of the scheduler is *at least* 50/51, or about 98%. (It might, in fact, even be 100% in this example, since 51 cycles might actually be necessary. All we know *for sure* is that at least 50 are necessary.) Thus, this algorithm tends to underestimate the scheduler's efficiency somewhat. However, since the estimated efficiency is very high for all cases, there is little to be gained in improving either the accuracy of the estimate or the efficiency itself. The estimated efficiencies range from 89.8% (for the double pendulum) to 99.7% (for the HTTRA reactor). The median efficiency is 96.1%.

Multiplying the compiler-generated schedule length (in machine cycles) by 50 nanoseconds (the prototype RTA uses a 20 MHz. clock) gives the amount of time required for derivative evaluation. To this, the compiler adds the time required to update the integrators and other state variables, to get the total frame time in the last column.

3.3 Comparison of RTA Performance with Existing Machines.

The results in the last section clearly establish that the scheduling compiler is capable of generating an efficient program from a high-level source program. Both the RTA hardware and software are adequate to the task. But how fast is the RTA in comparison with other available machines? Does it live up to its promise?

Table 3-2 compares the results obtained on the Spinning Missile benchmark with results obtained on several other systems. The times given are for one frame (i.e. one complete evaluation of all the equations, using a single-pass integration method such as Euler or second-order Adams/ Bashforth). As can be seen from the table, the RTA exhibits a speed advantage ranging from about 7 to 1 (over the Cray) to more than 200 to 1 (over the Motorola).

System	Frame Time	Frames/sec.
RTA	25.8 μsec.	39,000
Cray Y-MP	175 μsec.	5,700
Intel i-860	430 μsec.	2,300
Encore 2040	1.0 msec.	1,000
Encore 32/87	1.7 msec.	600
Motorola 68040	2.0 msec.	500
Motorola 68030	5.8 msec.	171

Table 3.2. Results of the Six-DOF Missile Simulation

It should be noted that the list includes systems that are much more expensive than the RTA, as well as systems using the latest technology—in fact, several of the systems represented in this table were not available at the beginning of this project; they became available after the project was well under way. (Test results are also available for a number of slower machines, including the IBM PC/AT and the VAX 11/780, but only the six fastest machines are shown for this application).

Comparisons have also been made with other applications on several other machines, including the Inmos Transputer, the IBM Powerstation, and the SPARC 2 RISC processor. The results for all applications and all machines to date support the following conclusions:

• The prototype RTA, including its associated software developed on this project, outperforms all other machines currently available on all tested applications.

• The Cray Y-MP (or, in some cases, the X-MP) is in second place for all applications for which a Cray was available. The RTA is from two to ten times faster than the Cray (The factor of two applied to the Armstrong Cork application, which is quite small. The factor of 5 to 10 is more typical for large applications, since, as noted before, the RTA gets more efficient as the application gets larger.) Note that the Cray sells for 15 to 25 million dollars.

• The RTA is over ten times as fast as any tested machine other than the Cray.

It is clear from the above summary that the project has been successful: the RTA meets its goals of speed, power, and programmability.

4. USER'S GUIDE TO THE PROTOTYPE RTA

This section contains a description of the prototype RTA user interface. The user interacts with the RTA hardware through several pieces of software: the macro expander, the compiler, libraries, linker, and command processor. Material in this section is adapted from the original software specification as set out in the first quarterly report for the Phase 2 project. Where major differences exist between what was proposed and what was actually implemented, they are flagged.

4.1. OVERVIEW OF THE SOURCE LANGUAGE

The Phase 2 NASA SBIR project is a feasibility demonstration, not a product development project. Consequently, the compiler is a rudimentary one—prototype software for prototype hardware.

EAI is currently enhancing the compiler into a full commercial version. For the commercial version of the RTA, we will extend the prototype software with a pretranslator, performing equation reduction and other syntactic functions to simplify the user's input task. The current version of the compiler was designed to be easy to implement, and close enough to the hardware to allow full utilization of its capabilities.

The input language accepts two types of source statements: equations and directives.

4.1.1. Equations

An equation has the form
OUTPUT LIST = OPERATOR NAME(INPUT LIST)
where "OUTPUT LIST" is a list of variable names, defining the output(s) of some operation. Most operators have a single output, but multiple outputs are allowed. If there are several outputs, their names are separated by commas. Each variable in an RTA program must appear on the left hand side of exactly one source equation. This equation is called the defining equation for that variable.

"INPUT LIST" is a list of names, similar to "OUTPUT LIST," but somewhat more general. Names in the input list can be either variable names or constant names. In addition, some operators allow signed names in the input list.

"OPERATOR NAME" is the name of an operator defining the relation between the input(s) and the output(s). From the implementation point of view, there are three types of operators: hardware operators, compiler-supported operators, and macro operators. These operators are described in detail in Sections 4.2, 4.3, and 4.4, respectively, of this report.

Examples of equations include

W = SUM(X, Y), which defines W as X + Y
Q = SUM(X, -Y), which defines Q as X - Y
S, C = SCOS(THETA), which defines S as SIN(THETA), and C as COS(THETA)
Y = INTEG(YDOT, Y0), which defines Y as the integral of YDOT with initial condition Y0.

4.1.2 Directives

A directive is any source statement that doesn't fit into the equation form described in the previous section. All directives begin with an @ sign or a dollar sign[1], followed by the name of the directive. After the directive name comes a (possibly empty) list of arguments, separated by commas. The number of arguments and their meaning depend on the directive. Examples of directives include

$CON, A = 32.5, B = 97, C = -28

which defines three constants and gives them values.

$ARRAY, =, ALPHA ,11, 0, 20

which defines ALPHA as an ARRAY. There are 11 values, equally spaced, ranging from 0 to 20. A full list of directives is given in Section 4.5. of this report.

The syntax for equations and directives is chosen to make the parsing task especially simple. In particular, the first character in a statement tells the parser immediately whether the statement is a declaration or an equation. This eliminates the need for "backtracking" in the parser.

4.2. HARDWARE OPERATORS

This section lists all the operators in the initial release of the compiler which are supported directly by the hardware. It does not include every opcode of the floating-point chip, since many of those are not meaningful for simulation applications. In particular, the RTA does not use denormalized variables, wrapped variables, or unsigned integers, so all opcodes referring to these data types are unused.

Furthermore, there are a few operators that do not correspond directly to 8847 opcodes. In particular, MIN, MAX, and the various comparison operators (see 4.2.2.3 and 4.2.2.4) are supported by EAI-added hardware.

[1]The original design allowed only a dollar sign, but the compiler was modified to accept either, so as to allow it to work smoothly with the macro processor, which has a different use for the dollar sign.

Most of the hardware operators are binary (they have two inputs), but several are unary (one input). No hardware operators have more than two inputs, but there are many multi-input operators in software (see Section 4.3.2). These are reduced by the software to two-input operators[2].

4.2.1 Unary Operators

4.2.1.1. Conversion Operators

FLOAT(X) is the result of integer-to-floating conversion.
INT(X) is the integer part of X. Any fractional part is discarded. Thus INT(7.95) = 7.
NINT(X) is the nearest integer to X. Thus NINT(7.95) = 8.

On the 8847 chip, INT and NINT are implemented with the same opcode, but different rounding modes.

4.2.1.2. Negation Operators

As we will see below, there are many sign options associated with the binary operators. In most cases, one can get a negated sum or product in a single operation, so that there is comparatively little need for unary negation operators. Still, they are necessary for a few cases, and are directly supported by the hardware.

NEG(X) is -X. Floating-point negation. Only the sign bit is changed.
INEG(X) is -X. Integer negation (two's complement).
COMP(X) is the logical complement. Every bit complemented.

4.2.1.3. Other Operators

SQRT(X) is the square root of X (floating-point).
ISQRT(X) is the integer square root. Probably not useful in simulation, but included for completeness.
ABS(X) is the Absolute value of X (floating point).

Note there is no IABS operator, since the hardware does not provide this as a single operation. It can, of course, be obtained with the comparison and switch operators.

[2]Two operators—SWITCH and LIMIT, are implemented in hardware, but treated as macros by the prototype compiler to keep the software simple. The commercial version of the software will be modified to take advantage of them.

4.2.2 Binary Operators

4.2.2.1. Floating-point Arithmetic

SUM(X, Y)is X + Y
ASUM(X, Y) is |X+Y|
MUL(X, Y) is X*Y
DIV(X, Y) is X/Y

These operations allow signed inputs. Signs may be either "+" or "-" (if omitted, the sign is taken to be "+" by default). Thus SUM(X, -Y) is X - Y, and MUL(-X, Y) is (-X)*Y. For logical variables, -X means "NOT X."

4.2.2.2. Integer Arithmetic

ISUM(X,Y) is X + Y
ISUB(X,Y) is X - Y
IMUL(X,Y) is X*Y
IDIV(X,Y) is X/Y

In the above four cases, X and Y are both integers. Note no signs are allowed, because the 8847 chip does not support signed integer operations with a single instruction. If you want -X*Y (X and Y integers), you cannot simply write W = IMUL(-X,Y). But you can get the desired result by writing a separate equation, introducing a new variable:

W = INEG(Q)
Q = MUL(X,Y)

Note also that X-Y is written as ISUB(X,Y), and not ISUM(X,-Y), since signed inputs are not allowed.

4.2.2.3. Floating-point Comparison

Comparison of two numbers (either floating-point or integer) produces a logical output. Logical values are represented by 32-bit words whose bits are all equal. If the word is all 1's, it represents the value TRUE; if it's all 0's, it represents the value FALSE.

Note that this is not the same as the convention used in the C language, where *any* nonzero value is taken as TRUE. The reason for the "all bits alike" representation of logical values is that it allows the use of bitwise AND and OR operations to perform logical selection. For more details, see the discussion of the SWITCH function.

GT(X,Y)is TRUE (all 1's) if X>Y; FALSE (all 0's) otherwise.
Similarly, we have LT (less than), GE, and LE.

These operators are derived from the COMPARE operator in the floating-point chip, augmented by PALs at the chip's data output to provide the 32-bit result when the instruction is one of the comparisons.

All floating point compare operators allow signed arguments, since the floating-point chip's COMPARE operator supports them. For example, GT(X, -Y) is TRUE if and only if X > -Y.

4.2.2.4. Integer Comparison

These operators have the same names as the corresponding floating-point operators, with an I appended at the beginning. Thus the operators are IGT, ILT, IGE, and ILE. They have the obvious meanings. No signs are allowed, because the hardware does not support signed integer comparison.

4.2.2.5. Logical Operators

AND(X,Y) is the bitwise AND of the 32-bit words X and Y.
OR(X,Y) is the bitwise logical OR.
XOR(X,Y) is the bitwise exclusive OR. Each bit is true if and only if the corresponding bits in X and Y are different.

4.2.2.6. Shift Operators

SLL(X, Y) means Shift Left Logical. Y must be an integer in the range 0 to 31. X can be any type of variable: integer, logical, or floating point. The result is the value of X, shifted Y bits to the left. The vacated positions on the right are filled with 0's.

SRL(X,Y) means Shift Right Logical. Just like SLL, except that the shift is to the right. Vacated positions on the left are filled with 0's.

SRA(X,Y) means Shift Right Arithmetic. The sign bit stays put, the other bits are shifted right, and vacated positions are filled with copies of the sign.

IMPLEMENTATION NOTE: The tables in the TI manual for the 8847 chip classify these shift operators as "one-input" operators (presumably because the first argument is thought of as a "data" input and the second merely as a "control" input). This means that bit I5 in the instruction word must be set to 1 (which normally means a single input) for the shifts to work. But *two* pieces of data have to be brought to the input pins, so the compiler treats them as binary operators.

4.3. COMPILER-SUPPORTED OPERATORS

The compiler treats all operators in the previous section in essentially the same way. The input(s) are brought to the input terminals of the chip, and several cycles later (depending on the latency of the operation) the result appears at the output terminals. This section deals with those operators that require "special handling" by the compiler. They come in four categories: state operators, multi-input operators, I/O operators, and data memory operators.

4.3.1. State Operators

A state variable is a variable that is defined by specifying two things: an initial value (sometimes called the "initial condition" for the variable) and a rule about how that variable changes as the computation progresses. State variables are distinguished from algebraic variables, which are defined by explicit expressions as functions of the values of other variables.

The best-known type of state variable is the integrator output. In fact, in ACSL, the integrator is the only type of state variable allowed (a limitation that makes ACSL programming unnecessarily cumbersome). Other state variables that have proved useful in simulation include flip/flops, track/store units, counters, etc.

For each type of state variable, the compiler must allocate storage for its current value, its initial value, and other values necessary to compute the necessary changes. In the case of a flip/flop, this is the previous value; in the case of an integrator, it's the derivative (and possibly other information, depending on the integration method used).

Since each type of state variable requires different compiler code, it is desirable to keep the number of distinct types of state to a minimum, to keep the compiler simple. The current compiler design uses only three types. One of these types (the **IMPL** operator for algebraic loops) is not needed to run the benchmarks, and was not implemented on this project. It will be included in the commercial version of the compiler. All three state operators are described in this report, but the description of the **IMPL** operator omits some details in the interest of brevity.

4.3.1.1. The Integrator

The expression "Y = INTEG(Ydot, Y0)" means just what it does in ACSL or any other CSSL. Y is the integral of Ydot with initial condition Y0. For full flexibility, a more general form of the integrator with two additional arguments, is provided:
Y = INTEG(Ydot, Y0, Lreset, Vreset)

This is a resettable integrator. Lreset is the logical control for resetting it and Vreset is the value to which it is reset. At the beginning of the run (the INITIAL region of CSSL, corresponding to the IC mode of an analog computer), Y = Y0, regardless of the other inputs. During the run, Lreset controls the mode of the integrator: when Lreset is false, the integrator operates normally; when Lreset is true, the integrator is reset to the value Vreset.

This version of the integrator allows for implementation of limited and "mode-controlled" integrators, simulation of static friction, and several other operations. Some of these operations are listed in Section 4.5.

4.3.1.2. The PAST Operator

The expression "Y = PAST(Y0, X)" means that at time zero, Y = Y0, and at later times, Y = the most recent past value of X. Mathematically, Y may be thought of as an approximation to the "left-hand limit" of $X(t-\delta t)$ as δt approaches zero. In practice, δt depends on the integration algorithm, but is never more than one step, and often less.

The PAST operator is not expected to be used directly by user programs very often. It is a building block, which is used by the macro library to implement such familiar operators as flip/flops, track/store units, and transport delays.

4.3.1.3. Algebraic Loops: The IMPL Operator

Because of schedule and budget limitations, the present project does not include algebraic loop capability. However, it is important in any commercial product. To indicate that this function has not been completely forgotten, a tentative syntax and some implementation suggestions are included here. The statement "Y = IMPL(Y0, X)" means that Y is supposed to equal X; however, X may itself be defined (either directly or indirectly) in terms of Y. The IMPL statement directs the compiler to generate an iterative loop. In the "IC mode" (the initial region of the simulation) the iteration starts with the initial guess Y0. In the "OPERATE mode" (the dynamic region of the simulation), the iteration starts with the value from the previous step.

Iteration proceeds until the difference between Y and X is sufficiently small, or until a predetermined number of iterations is performed, whichever happens first. The error tolerance, as well as the maximum number of iterations, should be user-definable. This requires additional syntax, either in the form of additional arguments to the IMPL function, or additional directives. Such specification is outside the scope of this project.

4.3.2. Multi-input Operators

In simulation applications, sums of more than two terms, and products of more than two factors, are fairly common. Since the hardware adds and/or multiplies only two things at a time, it is necessary to break these operations down into two-input hardware operators.

In the interest of execution speed, this operation is performed by the scheduling compiler, not by the pretranslator. For example, the simple three-input expression "W = SUM(X, Y, Z)" could be decomposed in three different ways:

W = (X+Y)+Z. Combine X and Y, then combine the result with Z.
W = X+(Y+Z). Combine Y and Z, then combine the result with X.
W = (X+Z)+Y. Combine X and Z, then combine the result with Y.

For a four-input sum, there are 15 different ways of reducing it to two-input sums. The different methods do not all take the same amount of execution time, and, in fact, any one of the possibilities might be optimal, depending on the amount of computation needed to generate the various inputs.

For example, suppose X and Z are state variables (whose values are available at the beginning of the step) and Y is an algebraic variable that results from a long sequence of computations. In this case, the last decomposition (X+Z)+Y is better than either of the others, since it allows one of the additions to be performed immediately. With the other two, no part of the computation can be performed until Y becomes available.

Using either of the alternatives would lengthen the critical path, and increase the likelihood of "data stalling." i.e. having one or more processors idle because the necessary input values are not available.

While the hardware directly supports only two inputs for a summer or multiplier, the compiler allows up to 120 inputs (more than anyone is likely to want to write in a single input statement). The breakdown into two-input operations is performed during the scheduling process.

The operators that allow multiple inputs are the following: SUM, ASUM, MUL, ISUM, IMUL, AND, OR, and XOR. These are all commutative and associative operations.

4.3.3 Data Memory Operator (and Directive)

The Data Memory is indexed, and supports indexed fetches and stores. An indexed fetch is simply an operator, in that it deter–mines a value (on the left-hand side of an equation) which depends on the input argument on the right. The basic form is

Y = FETCH(I, A, Offset)

or simply

Y = FETCH(I, A)

where I is an integer variable, A is an array, and Offset is an integer constant. The value of Y is the value of A(Offset+I), i.e. the contents of the data memory at the address (I+Offset+address of A). What makes this different from an ordinary operator is that the second argument is an array. To the scheduler, FETCH looks like a unary operator, since I is the only dynamically changing value. The constant sum (Address of A + Offset) is represented by the address in the Data Memory instruction word, and I is in an index register.

Note that I and Offset are not interchangeable. Of the three arguments, only the first may be variable. If Offset is omitted, it is taken as zero, so that FETCH(I, A) means A(I), i.e. the contents of the data memory at the address (I + Address of A).

In addition to the FETCH operator, there is the STORE directive. This has the form of a directive, not an operator, because there is no equal sign and no output value (see Section 4.5). The form is

$STORE, X, I, A, Offset

which means that the value of X is to be stored at the location A(I+Offset), i.e. the location whose address is I + (address of A) + Offset.

It is not expected that the user will use FETCH or STORE directly in source code very often. However, they are used by system macros to support function generation, transport delay, and data logging.

4.3.4 I/O Operator (and Directive)

I/O, like memory, requires one operator and one directive. The operator is IN:

```
Y = IN(17)
```

means that Y is the value that is available on input channel #17. What this means in detail is very installation-dependent. It might be coming from an ADC or from some digital sensor, or from another digital device. (On the prototype, no real-time input is included, so the IN operator is not needed).

The directive is OUT:

```
$OUT, X, 12
```

directs the compiler to feed the value of X to output channel #12. Again, what is actually connected to each output is installation-dependent. The NASA project includes four DAC channels for display purposes, so that the $OUT directive can have values of 0, 1, 2, or 3 for its second argument.

4.4. MACRO OPERATORS

Macros work like subroutines, except that instead of generating one copy of a program and linking to it, the compiler generates a complete copy of the macro program each time the macro is called. Not only does this eliminate subroutine linkage overhead, but it also allows for overlapped execution, since the macro expansion is just a collection of hardware operations which can be run in parallel with other operations in the simulation model.

The original plan for the prototype was for macro expansion to be done manually, to avoid the effort of developing a macro expander on this project. The macro library is in text form, and the programmer wanting to use a macro would copy its entire expansion into the source file and use a text editor to substitute names to prevent clashes and connect the expanded macro with the rest of the simulation model. This is tedious, but adequate for demonstrating the hardware feasibility.

However, during the project, a macro expander became available, which had been developed on another project. This was incorporated into the prototype software, which both saved development effort and allowed a much larger range of benchmark applications to be programmed.

The following list of macro operators includes all that are needed for running the benchmarks on this project, and several that are not. All the macros originally planned have been implemented, as well as a few others (notably the function of two variables and the unequally-spaced breakpoint search).

4.4.1 Flip/Flops

A flip/flop has two logical inputs S and R, and one logical output Y. There are three forms of flip/flop:

Y = SFLOP(Y0, S, R)
Y = RFLOP(Y0, S, R)
Y = TFLOP(Y0, S, R)

In the INITIAL region, Y = Y0. In the DYNAMIC region, S and R may be thought of as "command" inputs which tell the flip/flop what to do next. If S and R are both FALSE, then Y keeps its old value. If S is true and R is false, then Y is SET, i.e. its new value is TRUE. If S is false and R is TRUE, then Y is RESET (its new value is FALSE).

The three types of flip/flop are distinguished by what they do in the fourth case (S and R both true). In this case, the SFLOP Sets, the RFLOP Resets, and the TFLOP Triggers or Toggles (its output changes). All three types of flip/flop are implemented with the PAST operator. Section 4.6 shows the implementation in more detail.

4.4.2. Switches

The basic switch operator has three arguments:
W = SWITCH(L, X, Y)
In this statement, L must be a logical variable, and X and Y can be of any type, but they must be the same type. If L is true, then W is X; otherwise, it is Y. To remember the order of the arguments, think of this equation as "If L, then X, else Y."

In the special case where the third argument is zero, the SWITCH operator may be replaced by the hardware AND operator. This is why the values "all 0's" and "all 1's" were chosen for defining logical values. The SWITCH operator is implemented in the macro library as
SWITCH(L, X, Y) = OR(AND(L, X), AND(-L, Y)).
Remember that the AND operator accepts signed inputs, and -L means NOT L. The calculation of AND(-L, Y) is a single hardware operation.

4.4.3. Mathematical Functions

The usual math functions available in most programming languages are supported. A slight change is the fact that individual SIN and COS functions are replaced by the SCOS macro that delivers both values. This is because the algorithm contains much computation that is common to both; it can generate both in only slightly more time than required to generate just one.

The math functions consist of the following:

SCOS (sine and cosine; input in radians)
ALOG (natural log)
EXP (exponential function)
ATAN2 (polar resolver)
MAX, MIN (Maximum and minimum of two real variables)
IMAX, IMIN (Maximum and minimum of two integer variables)

All of these except ALOG, EXP, and ATAN2 were needed for the benchmarks, and were implemented on this project[3]. The others will be implemented in the commercial version.

4.4.4. Function Generation

Function generation requires the capability of breakpoint search and interpolation. The original benchmarks required only functions of one variable, with equally spaced breakpoints, and the original specification called for implementing only that capability. However, once the system was working, outside customer interest led us to run additional benchmarks, beyond those in the original plan, and some of these required functions of more than one variable. Accordingly, we created the software to support variable breakpoint spacing and functions of more than one variable. The macro library contains routines for functions of up to four variables. The commercial version of the RTA will support functions of up to eight variables, with either equally spaced or unequally spaced breakpoints.

4.4.4.1. Breakpoint Search

Two operators are defined for breakpoint search: BPSE (Break Point Search with Equal spacing) and BPSU (Break Point Search with Unequal spacing). Originally, only the first of these was to be provided, but in order to run additional benchmarks, the second was added as well.

In the statement
I, DEL = BPSE(X, B) or I, DEL = BPSU(X, B)
X is a floating-point variable, and B is an array of breakpoints, defined by the ARRAY directive described in the next section. The array elements should be strictly increasing. The first output, I, is an index defined by the following inequalities:

[3]The MAX, MIN, IMAX, and IMIN functions are implemented in hardware, not as macros.

If	$X < B(1)$,	then $I = 0$
If	$B(1) \leq X < B(2)$,	then $I = 1$
If	$B(2) \leq X < B(3)$,	then $I = 2$
	...	
If	$B(N-2) \leq X < B(N-1)$,	then $I = N-2$
If	$X \geq B(N-1)$,	then $I = N-1$

Note that in general, we have $B(I) \leq X < B(I+1)$, except that if X is out of range, i.e. $X < B(0)$ or $X > B(N)$, the value of I is limited to the range [0, N-1], since we don't want to address an array outside its defined boundaries.

For BPSE (equal spacing), the value of I is calculated by subtraction, multiplication, and float-to-fix conversion. The IMAX and IMIN operators are used to assure that I remains always in the desired range [0, N-1], even if X gets out of range.

For BPSU, a binary search is used, to minimize the number of comparisons required. In this case, no MAX and/or MIN operators are needed to keep the value of I within range[4].

The second argument, DEL, is defined as [X-B(I)]/[B(I+1)-B(I)]. As X varies between B(I) and B(I+1), DEL varies from 0 to 1. For equally-spaced breakpoints, the calculation requires a multiplication and an integer-to-floating-point conversion. For unequally-spaced breakpoints, it also requires two fetches from an indexed array, in which the previously-calculated reciprocals of the interval lengths—values of 1/[B(I+1)-B(I)]—are stored.

4.4.4.2. Interpolation

The INTERP operator, defined as
INTERP(DEL, A, B) = A + DEL*(B-A)
can be used for the actual function generation. Note that INTERP has the value A if DEL = 0, and B if DEL = 1. This operator applies to for both single-variable and multi-variable functions. For functions of one variable, the function value is given by
FUN1V(I, DEL, FV) = INTERP(DEL, FETCH(I, FV), FETCH(I, FV, 1))
where I and DEL are obtained from the input X using the BPSE or BPSU operators defined in the previous section, and FV is the array of function values. Note that the two fetches are from adjacent locations in memory.

[4]Instead of a single BPSU macro, the actual implementation uses several macros: BPS3, BPS5, BPS9, BPS17, and BPS33. In any given application, use BPSn, where n is the smallest number ≥ the number of breakpoints in the array.

Original plans called for the FUN1V macro to be expressed in terms of the INTERP macro, but the current version of the macro expander does not support nested macros, so FUN1V is expressed in terms of basic operators. Analogous macros FUN2V, FUN3V, and FUN4V are also provided in the macro library.

4.4.5 Other Operators

The other operators in the original specification are DIFF, FDIFF, BIDIFF, BCKLSH, DBLINT, LIMIT, LIMPOS, LIMNEG, LIMINT, DELAY, and ZHOLD. All these operators are described in the SIMSTAR manual, except for FDIFF (falling-edge differentiator) and LIMPOS and LIMNEG (positive and negative limits). All these have been implemented, and are included in the macro library.

FDIFF(X) is equivalent to DIFF(-X), i.e. it produces a pulse when X becomes FALSE. LIMPOS(X) is equal to X when X is positive, and zero otherwise. LIMNEG(X) is equal to X when X is negative and zero otherwise.

4.4.6 New Operators

Several other operators, which were not included in the original specification, turned out to be useful, and were implemented. They are included in the macro library. These include INCMOD, PEAK, VALLEY, MAT4, SOLVE4, and ROTATE.

The INCMOD macro increments an integer modulo another integer. In the macro call
I = INCMOD(I0, N)
I is incremented modulo N. Initially, I = I0. On every frame, I is incremented by 1 until it reaches N-1. When I = N-1, the next increment "wraps around" to zero. This macro is useful in in maintaining a pointer to a circular buffer for transport delay and data logging.

The PEAK macro picks the maximum of its input over all time. Y = PEAK(Y0, X) means that initially, Y = Y0, and on each succeeding step, Y is replaced by MAX(Y, X). If Y0 is chosen to equal X0, as is usually the case, then Y = MAX[X(t)], the maximum being over all previous time. Y is the peak value of X. Similarly, VALLEY(Y0, X) tracks the *minimum* value of X.

MAT4 applies a 4-by-4 "matrix" to a 4-dimensional "vector." The result is a four-dimensional "vector." The arguments are actually scalars; each "vector" consists of four explicitly-named scalars, and the "matrix"consists of sixteen scalars. The format is

Y1, Y2, Y3, Y4 = MAT4(M11, M12, M13, M14, M21, ...M44, X1, X2, X3, X4.

The computation performed is

Y1 = M11*X1 + M12*X2 + M13*X3 + M14*X4
Y2 = M21*X1 + M22*X2 + M23*X3 + M24*X4
Y3 = M31*X1 + M32*X2 + M33*X3 + M34*X4
Y4 = M41*X1 + M42*X2 + M43*X3 + M44*X4

SOLVE4 is the inverse of MAT4. The relation between the X values and the Y values is the same, but the Y values are inputs and the X values are outputs. In other words, SOLVE4 solves four linear equations for four unknowns Y1 through Y4. Gaussian elimination is used. The matrix should have a strong diagonal to avoid roundoff errors. This condition is generally met in physical applications where the matrix is an inertia matrix and the task is to solve the equations for the highest derivatives. The format is

X1, X2, X3, X4 = SOLVE4(M11, M12, M13, M14, M21, ...M44, Y1, Y2, Y3, Y4.

SOLVE4 and MAT4 are used in the four-DOF robotic manipulator application, and were therefore needed on this project. They are provided in the macro library. For the commercial version, at least MAT2 through MAT8 and SOLVE2 through SOLVE8 will be provided, allowing for systems of up to eight equations.

The ROTATE macro rotates a two-dimensional vector with coordinates X, Y through an angle θ to procuce U and V. The form is

U, V = ROTATE(X, Y, S, C)

and the defining equations are

U = X*C + Y*S
V = Y*C - X*S

where S = sinθ and C = cosθ. The macro uses S and C as inputs rather than θ itself, so that multiple rotations of different vectors through the same angle do not need to repeat the sin and cos calculation. The SCOS macro can be used to generate S and C from the input θ.

4.5. COMPILER DIRECTIVES

The vast majority of source statements in a typical simulation program will consist of equations. These all have the same form, as described in Section 4.1. Any statement that does not fit into this form is by definition a compiler directive.

The syntax is different for different directives, but all begin with a dollar sign or "at sign" followed by the name of the directive. Since no equation begins with either an @ or a $ sign, the parser knows at once if a particular statement is a directive, and the next word tells it which directive. This eliminates the need for backtracking in the parser, since there is never any ambiguity as to whether a source statement is a directive or an equation.

4.5.1 Type Directives

There are three data types: Integer, Floating Point, and Logical. In typical applications, the vast majority of variables and constants are floating point. Therefore, this is the default (unlike the default assumption of "integer" for C functions or the FORTRAN initial-letter default). Hence the only type declarations necessary are INTEGER and LOGICAL. They are abbreviated by their first three letters.

For example, the declarations

```
$INT, X, Y, Z
$LOG, A, B, C
```

declare X, Y, and Z to be integers and A, B, and C to be logical. Every variable or constant that is either integer or logical should be declared before its first use; otherwise, it will be assumed real (floating-point).

4.5.2. Defining Constants: The CON Directive

The CON directive is used to tell the compiler that a certain name refers to a constant, and to give the constant a value. Constants of any data type are allowed, but if they are not real, they must be specified before the $CON directive. For example, the declaration

```
$CON, A = 5.0, B = 2.34, C = .FALSE., I = 5
```

declares values for four constants. A and B are automatically REAL, but I and C must be declared INT and LOG respectively.

4.5.3. Structure Directives

Structure declarations declare the beginning and end of structural blocks, such as the INITIAL, DYNAMIC, and TERMINAL blocks. The directives

```
$INIT
statements in initial region
$ENDINIT
```

delineate the beginning and end of the INITIAL region. This is a block of code that is simply copied to a file (the .INI file), and later compiled and linked with the rest of the setup and runtime program that runs on the host. It is executed at the beginning of each new run. On the prototype system, the INI file is generated, but not compiled and linked. The user must perform this task manually. The commercial system will provide for the automatic compiling and linking of this file.

Similarly, the directives

DERIV

statements in derivative region

$ENDDERIV

delineate the beginning and end of the derivative section, which contains the user's model. Only one derivative section is allowed for the prototype system, so these directives are not necessary.

Within the derivative section, there should be a either a $DONE directive or a $RUNTIME directive, to terminate the run. The $DONE directive has the form

$DONE, name of logical variable.

The logical variable must be defined by an equation in the derivative section, and represents a condition for terminating the run.

The $RUNTIME directive has the form $RUNTIME, real constant. For example, the directive

$RUNTIME, 10

means "run the simulation until TIME ≥ 10 seconds".

Both RUNTIME and DONE directives may be used in the same program; the run will then terminate when the DONE signal becomes true or after ten seconds, whichever happens first.

The directives

$TERM

statements in terminal region

$ENDTERM

delineate the beginning and end of the terminal section, which, like the initial section, consists of code that is simply copied to a file and later linked with the rest of the host program.

The directive

$END

(note nothing written after the "END") signifies the end of the source file. It is not really necessary, since the compiler will interpret end-of-file as meaning the end of the source program, but it serves to reassure a human reader that the "tail end" of the source file has not been inadvertently deleted.

4.5.4. I/O and Data Memory Directives

These are the STORE and OUT directives. They have already been described in Sections 4.3.3 and 4.3.4. They are directives, rather than equations, since they do not conform to the equation syntax: they have no equal sign and generate no output values.

4.5.5. The ARRAY Directive

This directive is used to tell the compiler that a specific name refers to an array, to dimension the array, and (optionally) to fill it with initial values. In simulation, arrays are used mostly for function generation, and can be either breakpoint arrays or function value arrays. Such arrays are normally "read-only," that is, they are not changed by the running program.

Applications like transport delay and function storage and playback require that arrays be updated dynamically. The FETCH operator and the STORE directive allow for both reading and writing arrays at run time.

The ARRAY directive has three forms, depending on the type of array.

4.5.5.1 Filled Arrays.

Filled arrays, used for function generation, are filled at load time with data from a file. The format is

```
$ARRAY, F, name, size
```

where the "F" means that this is a filled array, "name" is the name of the array, and "size" is a literal integer constant.

The compiler uses the name internally to identify functions which use it in the source program. It also passes the name along to the linker, which looks for a file in the current directory with the same name, and the extension .TXT. This is the data file used to fill the array at load time.

The size of the array is the number of values it contains. The corresponding data file must contain this many values, in text form, separated by spaces.

4.5.5.2. Blank Arrays.

Blank arrays are not filled from a file at load time, but may be (optionally) initialized with a value before each run. They are used for transport delay and data logging. The general form is

```
$ARRAY, B, name, size [,initial value]
```

The "B" tells the compiler that this is a blank array. The name and size have the same meaning as before, but no file of data values is needed. If the initial value is included, then the array will be filled with that value before each run. This is the case for arrays used in transport delay. If no initial value is provided, the array is not initialized before each run; this is the case for arrays used for data logging. The initial value, if provided, must be a parameter name, not a literal constant.[5]

[5]This restriction will be removed in the commercial version, which will allow either a named parameter or a literal constant.

4.5.5.3. Equally-spaced Arrays.

The directive

$ARRAY, =, name, size, min, max

declares an array of equally-spaced values, generally used as a breakpoint array for function generation. This array does not actually exist as a set of equally-spaced values in the data memory. Instead, the min, max, and size arguments are used to perform a breakpoint "search" by arithmetic calculations and float-to-fixed conversion. The min, max, and size arguments must be literal integer constants.

Examples:

$ARRAY, =, ALPHA, 11, 0, 20

This example was given in Section 4.1.2. It defines the array ALPHA to have 11 values 0, 2, 4, 6, 8, 10, 12, 14, 16, 18,and 20. Note that the number of values in the array is one more than the number of intervals. For ten intervals, each of length 2.0, we need eleven data points.

$ARRAY, F, BETA, 5

defines BETA as an array of five values (four intervals), unequally spaced. The breakpoint values will be found in a disk file named "BETA.TXT."

$ARRAY, B, GAMMA, 100, A

defines GAMMA as an array of 100 values, and generates code that copies the constant A into every array location before each run. If the last argument were omitted, the array would not be initialized.

4.5.6. The ALARM and VAR Directives

The original software plan included an ALARM directive, to allow for program-generated detection of exceptional conditions, and a VAR directive, to allow the user to rename the independent variable (the default name is "TIME"). These were not implemented on this project due to lack of time, but they will be included in the commercial version.

4.6 OVERALL SOFTWARE DATAFLOW

Figure 4.1 is an overall software block diagram, showing data flow for the entire prototype RTA programming system. This diagram shows five executable host programs: **MACRO**, **PHASE1**, **PHASE2**, **SLINK**, and **CMD**.

Each program reads one or more files and each program (except for **CMD**) writes one or more files. In normal use, the programs are executed in succession, under the control of a batch file.

The compilation and linking process creates a large number of intermediate files on the disk, as described below. To avoid confusion, it is strongly recommended that all files related to a particular simulation model (i.e.the source file and all necessary function generator data files) be placed in a *single directory* devoted to that particular simulation model. All intermediate files generated by the process will be placed in that directory. This practice prevents any directory from getting cluttered up with many files relating to different simulations. The benchmark applications provided on the prototype disk are structured in this manner.

The executable files (the compiler, linker, etc. as described below) are system files that are placed on the disk in the directory \RTA\SYSTEM. They are not associated with any particular simulation, so all simulations need to access them. This may be done very simply by putting the \RTA\SYSTEM directory in the PATH variable, so that DOS can fine the executable files.

The starting point is a source file containing the equations and directives describing the model. This source file should have a name without a DOS extension (referred to simply as "name" in the figure). All files created during compilation, linking, and execution will have this same name with different extensions. (e.g. NAME.MAC, NAME.EXP etc.).The macro expander reads the equations and looks up the operator name for each equation in a table of "built-in" RTA operators (see Sections 4.2 and 4.3). If an operator is not in that list, the macro expander scans the list of macro operators in the macro file(s). If the operator is found in one of the macro files, the macro expander copies the entire body of the macro into the source file, with appropriate name substitutions.

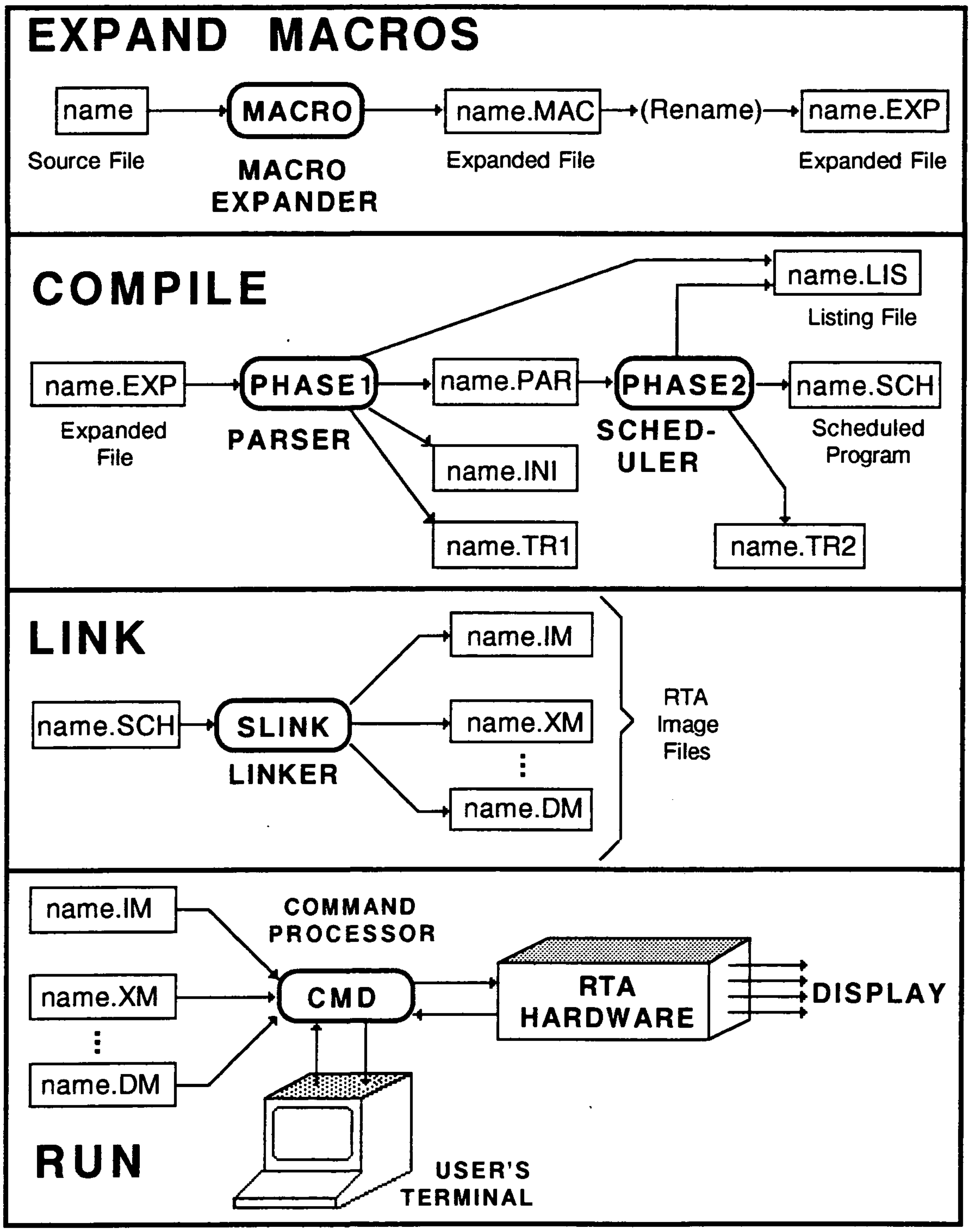


Figure 4.1 Dataflow in a Typical RTA Compilation.

To illustrate the macro expansion function, suppose the following line occurs in the source file:

TFLOP(P, P0, XPOS, XNEG)[6]

The operator TFLOP (Trigger flop or Toggle flop) is not built into the hardware or the compiler, but it is found in the macro library. The macro library contains the following definition:

```
MACRO TFLOP(Y, Y0, S, R)
    MACRO RENAME TEMP1, TEMP2, YNEXT
    TEMP1 = AND(S, -Y)
    TEMP2 = AND(-R, Y)
    YNEXT = OR(TEMP1, TEMP2)
    Y = PAST(Y0, YNEXT)
MACRO END
```

This definition corresponds to the following truth table:

S	R	TEMP1	TEMP2	Ynext
0	0	0	Y	Y
0	1	0	0	0
1	0	-Y	Y	1
1	1	-Y	0	-Y

which shows that the operator responds correctly to all possible combinations of inputs. (Remember that "-Y" means NOT Y.)

The macro expander replaces the original source line with a complete copy of the macro definition (minus the header and terminator lines). In the copying process, Y is replaced by P, Y0 by P0, S by XPOS, and R by XNEG. The dummy names TEMP1, TEMP2, and YNEXT are replaced with names generated by the macro expander. Different names are generated for each use of the TFLOP macro. As a result, the original use of TFLOP is replaced by

```
ZZ34 = AND(XPOS, -P)
ZZ35 = AND(-XNEG, P)
ZZ36 = OR(ZZ34, ZZ35)
P = PAST(P0, ZZ36)
```

Names beginning with ZZ are reserved for system use, and should not be used in the source program. In this example, the macro expander substitutes the names ZZ34, ZZ35, and ZZ36 for the

[6]The prototype software insists on this form, with the macro name first. The commercial version will also allow the more natural form, with the output first: P = TFLOP(P0, XPOS, XNEG).

dummy names TEMP1, TEMP2, and YNEXT in the macro definition. This example assumes, of course, that the last macro-assigned name was ZZ33. If the next macro encountered after this one is another use of TFLOP, this new use of the TFLOP macro will have the names ZZ37, ZZ38, and ZZ39 for its internal variables.

Thus, each equation in the file produced by the macro expander has a single operator which is directly supported either by the hardware or by the compiler. The macro expander output has the same name as the input file, but with the extension .MAC. The compiler expects an input file with the extension .EXP (EXPanded file). This file may be created by renaming the .MAC file. For programs that don't use the macro library, the user can save a step by starting with the name.EXP file and bypassing the macro expander entirely.

The scheduling compiler consists of two phases: **PHASE1** is the parser, which simply reads the source file and applies a lexical analysis and some bookkeeping chores to turn the source representation into an internal set of tables defining, for each variable, what its generating equation is (i.e., what the operands and the operator are).

The output of the parser is the file name.PAR, which is fed into **PHASE2**: the scheduler. This program contains the heuristic scheduling algorithm which decides, for each variable, when (on which clock cycle) and where (on which processor) it is to be generated. The resulting schedule is written into the name.SCH file. The compiler also generates the name.INI file, which contains the initial region (the code to be executed at the bebinning of each run). In the commercial version, this file will be compiled and linked with the rest of the program to set automatically all parameters that depend on other parameters. At present, the user must print out this file and perform the computations manually.

The other files created by the compiler are the listing file, name.LIS, and the trace files, name.TR1 and name.TR2. The listing file contains a copy of all source statements, except that blanks and comments are removed, and statement numbers added to facilitate error tracing. It also contains error messages, a brief summary of the utilization of the various operators, and the frame time. Any error messages detected in the input source file are immediately flagged, with indicators pointing directly to the offending text.

The trace files are of interest only to those who know the internal structure of the compiler (i.e. they were used for debugging the compiler itself, rather than the user's source program). Since the generation of these files takes considerable time (in many cases, more time than the actual compilation), they are made optional. By default, no trace files are generated. If a "T" flag is used on the command line invoking the PHASE1 or PHASE2 part of the compiler, the trace file will be generated. Without the flag, no trace file will be generated, and the compilation will be faster.

The linker, **SLINK**, reads the scheduler output file name.SCH, resolves memory references, and generates the executable RTA program. This program consists of several files called "image" files, since each is an image of one of the RTA memories. For each RTA memory (either program memory or data memory), there is a file specifying which locations need to be loaded, and with which values. For each module in the system, the following files are generated:

name.IM: the Instruction Memory image.
name.DM: the Data Memory image.
name.CM: the Control Memory image.
name.XM: the Index Memory image.
name.A0M, name.B0M, name.A1M., and name.B1M: the images of the four operand memories.

In a conventional computer, a variable is identified with a particular memory location, but in the RTA, the same value can reside in several locations. For example, a variable that is used as input to several computations on different modules may have its value stored in several input memories. And, of course, a constant is assumed available anywhere it is needed. To avoid the need to move constant values over the data busses at run time, duplicate copies are stored in whichever memories need the value.

Most of the program and/or data memory locations are in blocks with contiguous addresses beginning at zero, so that the information will be in the form of memory blocks consisting of sequences of data or program words. These files are used by the command interpreter **CMD**, which loads these words into the appropriate RTA memories.

CMD also interprets and executes commands to set and read various variables or constants in response to user commands typed in at the terminal. It responds to the following commands (the command name and arguments, if any, are separated by one or more spaces):

SET name F value
sets the parameter "name" to the floating-point value "value." If the parameter is integer instead of floating point, the "F" should be replaced by "I."

SHOW name F
shows the value of the variable or parameter "name" in floating-point form. Using "I"instead of "F" results in display in integer form. In the commercial version of the command interpreter, the "F" or "I" will be optional, since the type of a variable or parameter is available from the compiler and linker.

METHOD
followed by "A" or "E," sets the integration method to Adams/Bashforth (second order) or Euler

(rectangular) integration. If no letter follows "METHOD," the current method is displayed.

OPMODE R
or
OPMODE S
sets the operating mode to Repetitive or Single. The former cycles the RTA between INITIAL CONDITION and OPERATE modes until interrupted from the console: the latter makes one run and stops. Typing OPMODE without a letter simply displays the current opmode.

RUN
starts the RTA running. Depending on the current opmode (see above) the system will either make one complete run or cycle repetitively.

TAKE *filename*
tells the command processor to take its next commands from a textfile. This allows for prepared scripts of operating sequences.

WHAT *name*
accesses the .EXP file and types out all lines containing the name. During debugging, if a particular variable, say BETA, isn't behaving as expected, you can find out instantly where the variable is generated, and how it is used, by typing WHAT BETA. The response will be a listing of all source lines containing the name BETA, including the statement that generates it, and all statements that use it, but *not* including other lines containing the string BETA, such as lines containing BETA0, BETA1, or BETADOT.

HELP
provides online help.

EXIT
exits the command processor and returns to DOS.

5. ANALYSIS OF AN RTA BENCHMARK

This section illustrates the details of how an RTA program is converted step-by-step from a set of input source equations into an internal dataflow graph, or "patching diagram" and then into a final, fully scheduled executable program. The problem statement, the dataflow graphs, and the manually-generated schedule in Table 2 are adapted from the Phase 1 Final Report.

One of the key goals of this project was to verify the RTA's programmability, i.e. to show that a user could write a program without detailed knowledge of the hardware architecture, and have it compile, run, and generate correct results. Can a compiler create a schedule for a multiprocessor system from a high-level source program without need for detailed knowledge of the computer on the part of the user? We now know that the answer is "Yes."

The schedule in Table 2 was generated manually, before the scheduling compiler had been written. Seventeen months later, when the compiler was available, the compiler-generated schedule proved to be within two cycles of the manually-generated one (128 cycles versus 126). The compiler-generated schedule runs on the RTA hardware, and produces the same result as the original CSSL program run on EAI's in-house Encore 32/87.

The example shown here is the Double Pendulum (Fig. 1). This example was chosen because it is simple enough to analyze fully, and at the same time complex enough to illustrate all the important concepts. As Murray[7] says, it "exemplifies all of the inherent coupling and nonlinear characteristics of manipulator dynamics."

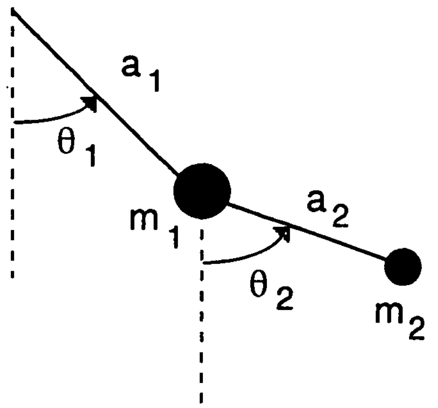


Figure 1. The Double Pendulum

The equations, taken from Murray, are as follows:

[7]Murray, J.J. "*Computational Robot Dynamics*" Ph.D Dissertation, Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA 15213, Sept. 10, 1986.

Inertial Coefficients

$d_{11} = a_1^2 m_2 + 2a_1 a_2 m_2 \cos(\theta_2) + a_2^2 m_2 + a_1^2 m_1$
$d_{12} = a_2^2 m_2 + a_1 a_2 m_2 \cos(\theta_2)$
$d_{22} = a_2^2 m_2$

Centrifugal and Coriolis Coefficients

$c_{12} = -a_1 a_2 m_2 \sin(\theta_2)$
$c_{22} = -a_1 a_2 m_2 \sin(\theta_2)$

Gravitational Coefficients

$g_1 = a_1 g m_2 \sin(\theta_1) + a_2 g m_2 \sin(\theta_1+\theta_2) + a_1 g m_1 \sin(\theta_1)$
$g_2 = a_2 g m_2 \sin(\theta_1+\theta_2)$

Torque Equations

$h_1 = c_{22}\dot{\theta}_2^2 + 2c_{12}\dot{\theta}_1\dot{\theta}_2 + g_1$
$h_2 = -c_{12}\dot{\theta}_1^2 + g_2$

Highest Derivatives (implicitly defined)

$d_{11}\ddot{\theta}_1 + d_{12}\ddot{\theta}_2 = \tau_1 - h_1$, where $\tau_1 = K_1 * \dot{\theta}_1$ to include damping
$d_{12}\ddot{\theta}_1 + d_{22}\ddot{\theta}_2 = \tau_2 - h_2$, where $\tau_2 = K_2 * \dot{\theta}_2$ to include damping

Explicit Solution for Highest Derivatives

$\Delta = d_{11}d_{22} - d_{12}^2$ (The determinant of the system)
$\ddot{\theta}_1 = (1/\Delta)[d_{22}(\tau_1 - h_1) - d_{12}(\tau_2 - h_2)]$
$\ddot{\theta}_2 = (1/\Delta)[d_{11}(\tau_2 - h_2) - d_{12}(\tau_1 - h_1)]$

A CSSL program for these equations is given in Figure 2.

```
PROGRAM

   "Double Pendulum Model. File name S.DPEND"
   "George Hannauer, July 12, 1989"
   "This two-DOF model is taken from"
   "COMPUTATIONAL ROBOT DYNAMICS, by John J. Murray."
   "PhD Dissertation, Carnegie-Mellon University."
   "Sept. 10, 1986."

   INITIAL

      CONSTANT TEND = 10, XLO = 0, XHI = 10 $ "for plotting."
      CONSTANT G = 0.980
           "Acceleration of gravity in MKS units"
      CONSTANT A1 = 2, A2 = 1, M1 = 5, M2 = 1
      CONSTANT THE10 = .0, THE1D0 = 0
      CONSTANT THE20 = 0, THE2D0 = 1

      P1 = A1*A1*M2
      P2 = A1*A2*M2
      P3 = 2*P2
      P4 = A2*A2*M2
      P5 = A1*A1*M1
      P6 = A1*G*M2
      P7 = A2*G*M2
      P8 = A1*G*M1
      P9 = P1 + P4 + P5
      P10 = P6 + P8

   END

   DYNAMIC
      DERIVATIVE

         "State variables"
            THE1D = INTEG(THE1DD, THE1D0)
            THE1 = INTEG(THE1D, THE10)
            THE2D = INTEG(THE2DD, THE2D0)
            THE2 = INTEG(THE2D, THE20)

         "Trigonometric Functions"
            S1 = SIN(THE1)
            C1 = COS(THE1)
            S2 = SIN(THE2)
            C2 = COS(THE2)

            S12 = S1*C2 + S2*C1 $ "SIN(THE1+THE2)"
```

Figure 2. CSSL Source Listing for the Double Pendulum (Sheet 1 of 2)

```
         "Inertial Coefficients"
            D11 = P9 + P3*C2
            D12 = P4 + P2*C2
            D22 = P4

         "Centrifugal and Coriolis Coefficients"
            C12 = -P2*S2
            C22 = C12

         "Gravitational Coefficients"
            G1 = P10*S1 + P7*S12
            G2 = P7*S12

         "Torque Computation"
            H1 = 2*C12*THE1D*THE2D + C22*THE2D*THE2D + G1
            H2 = -C12*THE1D*THE1D + G2

            TAU1 = -K1*THE1D
            TAU2 = -K2*THE2D

            CONSTANT K1 = 0.1, K2 = 0.1

         "Solve for highest derivatives
          (angular accelerations)"

         "Reciprocal of determinant"
            RDET = 1.0/(D11*D22 - D12*D12)

            THE1DD = RDET*(D22*(TAU1-H1) - D12*(TAU2-H2))
            THE2DD = RDET*(D11*(TAU2-H2) - D12*(TAU1-H1))

         TERMT(T.GE.TEND)
      END
   END

   TERMINAL
   END

END
```

Figure 2. CSSL Source Listing for the Double Pendulum (Sheet 2 of 2)

A few observations are worth making about this program:

• Calculations involving only constants have been moved from the DYNAMIC region to the INITIAL region, since they need to be performed only once per run. These calculations are performed on the host computer before each run and the values transferred to the RTA at the beginning of the run. Thus coefficients like $a_1^2m_2$ (A1*A1*M2 in CSSL) are given names and defined by equations written explicitly in the INITIAL region. On the commercial RTA system, this process will be completely transparent to the user, who can write, if preferred,

```
D12 = A2**2*M2 + A1*A2*M2*COS(THE1)
```

and the software will automatically generate the equivalent of

```
P2 = A1*A2*M2
P4 = A2**2*M2
```

in the INITIAL region, and

```
D12 = P4 + P2*C1
```

in the DYNAMIC region.

This task is already being performed automatically in the SIMSTAR® programming system. Machine coefficients ("pot-settings") are calculated and transmitted to the analog components before every run. There is no need for the user to break out these computations into the INITIAL region manually.

• In order to reduce the number of computations, a general decision was made to generate all trigonometric functions of *sums* and *differences* of state variables by using trigonometric identities, rather than additional uses the SIN/COS macro. Thus S1, S2, C1, and C2 are the sines and cosines of the state variables THE1 and THE2, and are calculated directly from the state variables, while he value of SIN(THE1+THE2) is calculated as S1*C2 + S2*C1, replacing a call to the SIN/COS macro by two multiplications.

This is probably a good idea in general, since a SIN/COS macro uses 20 arithmetic operations. But in this particular case, it had a peculiar consequence—it actually *lengthened* the running time slightly. This is typical of the "anomalies" that arise in multiprocessor scheduling applications. Since the general problem is NP-complete[8], it is unlikely that any efficient program will always produce optimal results. As long as the discrepancy is small (in this case, fewer than 5 cycles out of 125), the result is a program that is close enough to optimal for practical purposes.

• Since the system has only two degrees of freedom, the inertia matrix is 2 by 2, and the algebraic system is easy to solve by determinants. This is *not* typical of real applications: in practice, for systems larger than about 3 by 3, Gaussian elimination or some form of matrix factoring is preferred; the RTA macro library contains a macro for performing this operation.

The CSSL source equations are transformed by the compiler into a set of machine operations, which may be thought of as connections between virtual "components," as shown in Fig. 3.

[8]Garey, M.R. and Johnson, D.S. *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H.Freeman and Company, San Francisco, 1979.

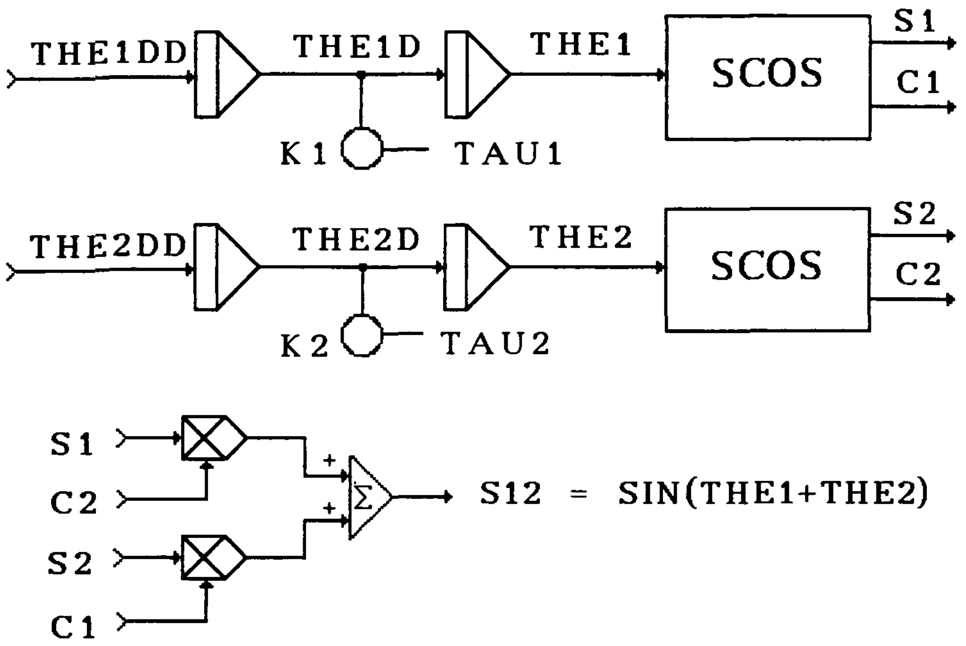
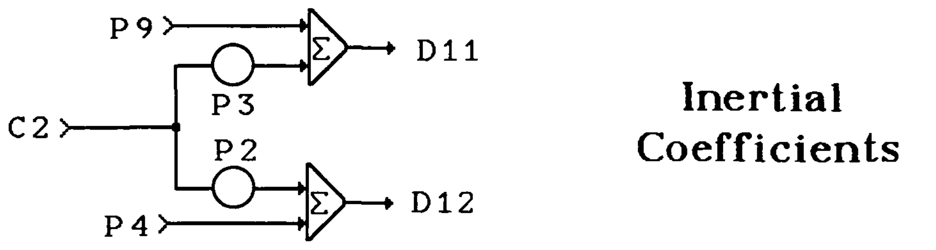
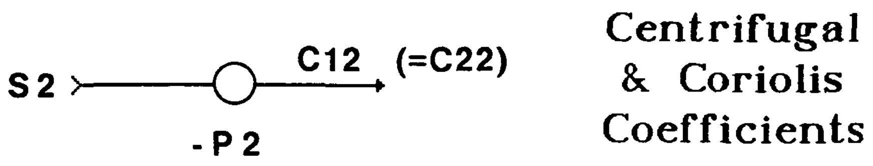


Figure 3a State Variables and Trigonometric Functions





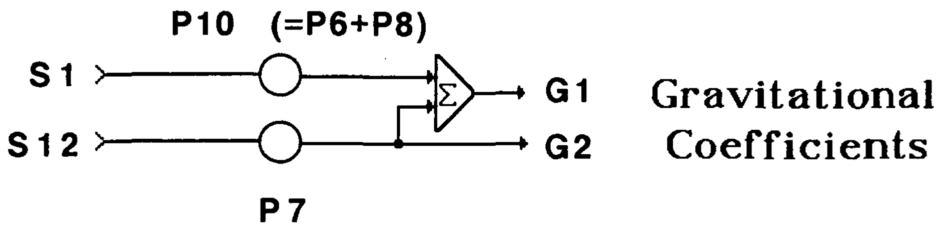


Figure 3b. Calculation of Coefficients

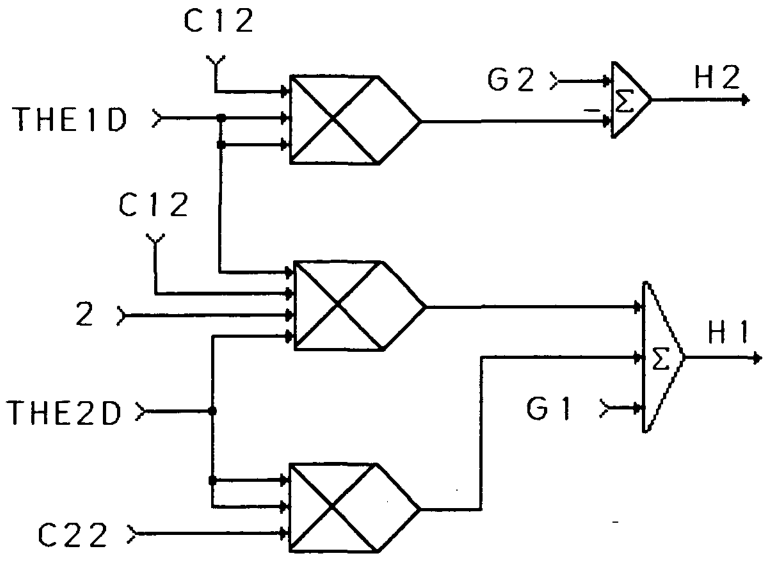


Figure 3c. Torque equations.

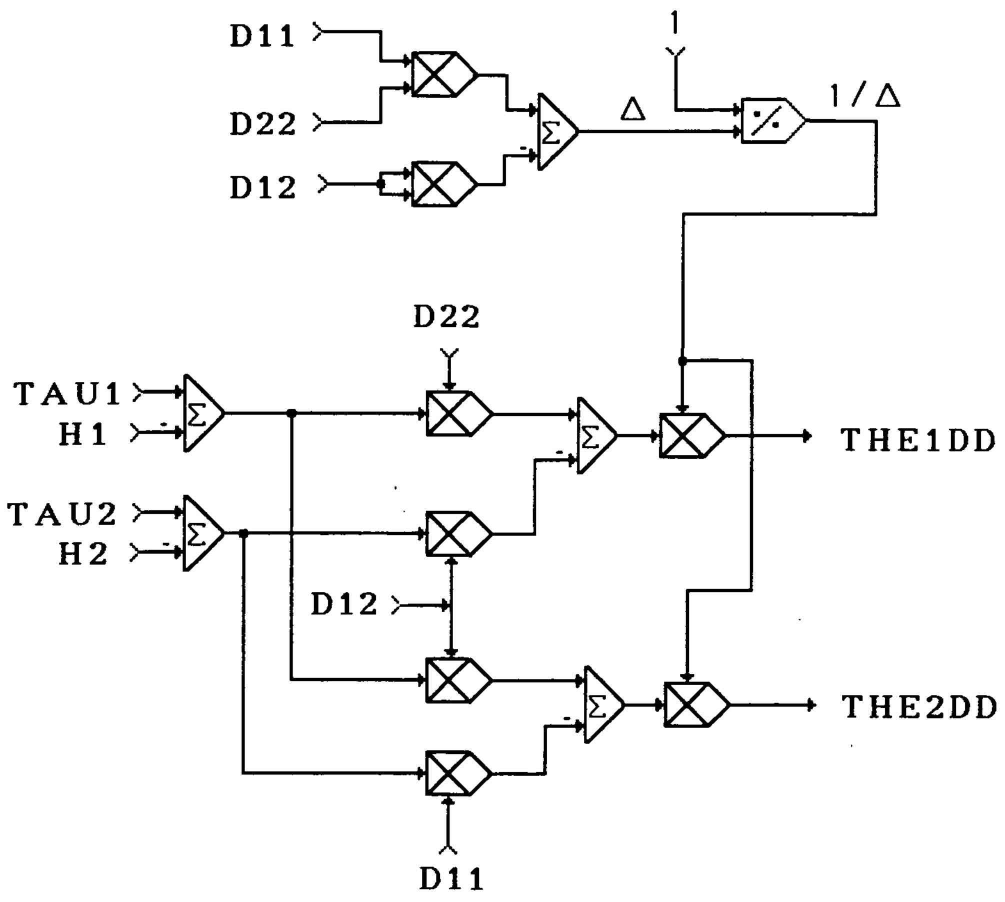


Figure 3d. Solving for the Highest Derivatives

The similarity of Figure 3 to an analog computer patching diagram is deliberate, since the RTA evolved from the analog computer. There are a few differences:

• There is no need for scaling.

• Summers *and* multipliers may have an unlimited number of inputs. Of course, the software must reduce these multi-input devices to successive applications of two-input devices: the RTA contains no multi-input arithmetic hardware. But this reduction is transparent to the user.

In addition to resolving multi-input components into single inputs, it is necessary also to replace every occurrence of a library macro (such as the sine/cosine macro in the Double Pendulum example) with a complete copy of its expansion. Every new variable defined in this process is given a name to facilitate the discussion—these auxiliary names all begin with "#", which is not a legal character for an input source name; this convention guarantees that there will not be a clash between the generated names and the user-defined input names[9]. After these transformations, the

[9]The original plan called for the use of the #, but on the prototype version, the "# "was changed to "ZZ,"

resulting dataflow graph looks like Figure 4.

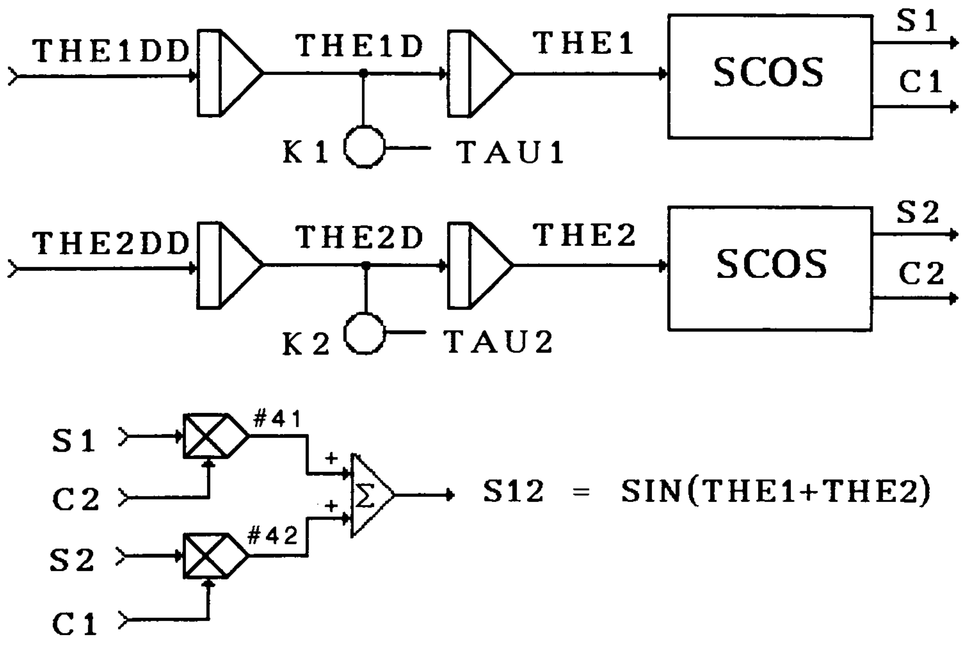


Figure 4a. State Variables and Trigonometric Functions
(With Auxiliary Variables Numbered)

Note that the replacement of the two occurrences of the sin/cos macro result in two complete copies of the sin/cos macro body being inserted into the final program—these are shown in Figures 4e and 4f. It should be remembered that the normal user will never see this macro expansion—just as the normal user of an ordinary computer doesn't see the internal working of the sine or cosine routines in the library.

to make it compatible with the pre-existing macro expander. However, the diagrams, taken from the Phase 1 Final Report, still show the "#."

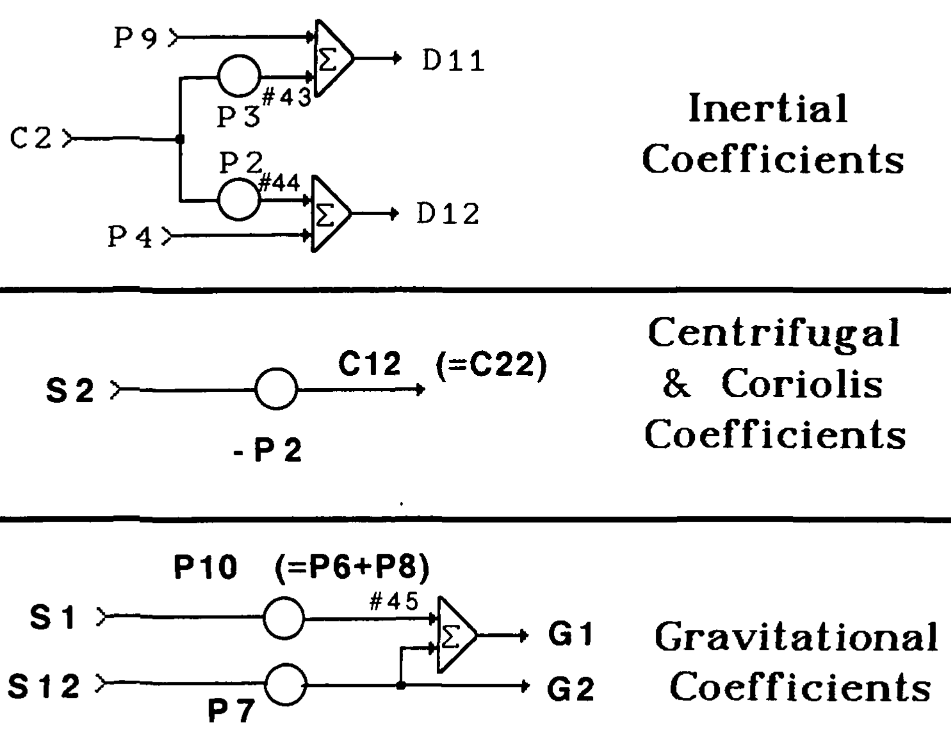


Figure 4b. Coefficient Calculation showing Auxiliary Variables

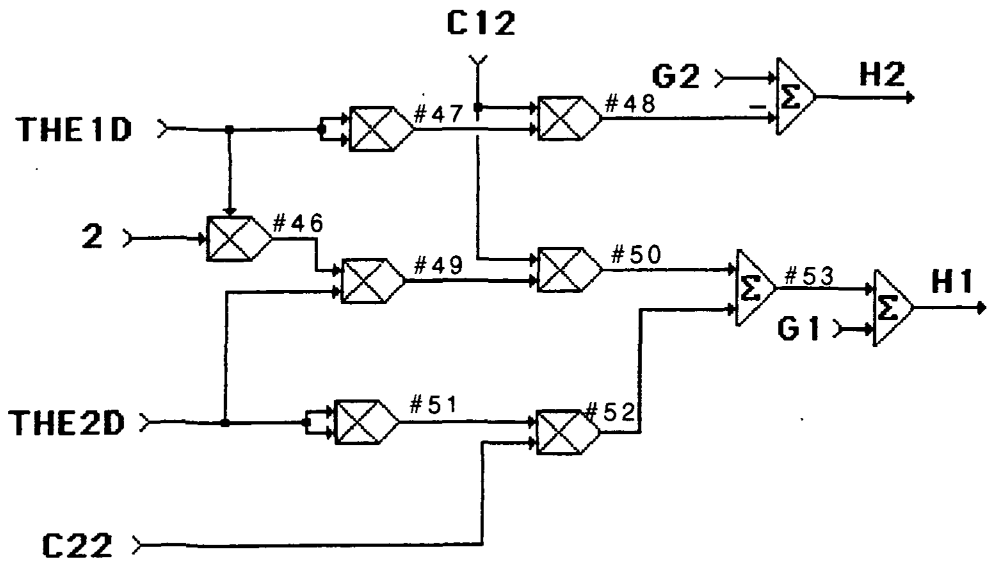


Figure 4c. Torque Equations, with Auxiliary Variables.
Compare Figure 3c.

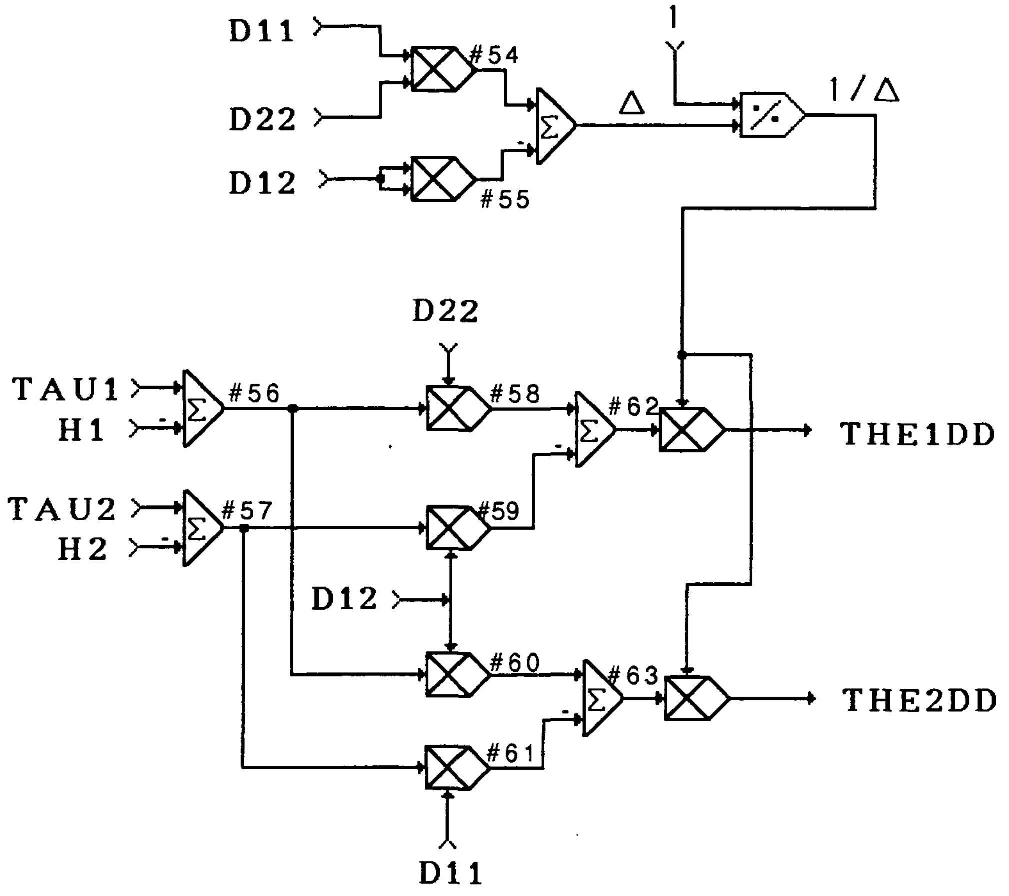


Fig. 4d. Solving for the Highest Derivatives, with Auxiliary Variables

Electronic Associates, Inc.
185 Monmouth Parkway, West Long Branch, N. J. 07764 (908) 229-1100

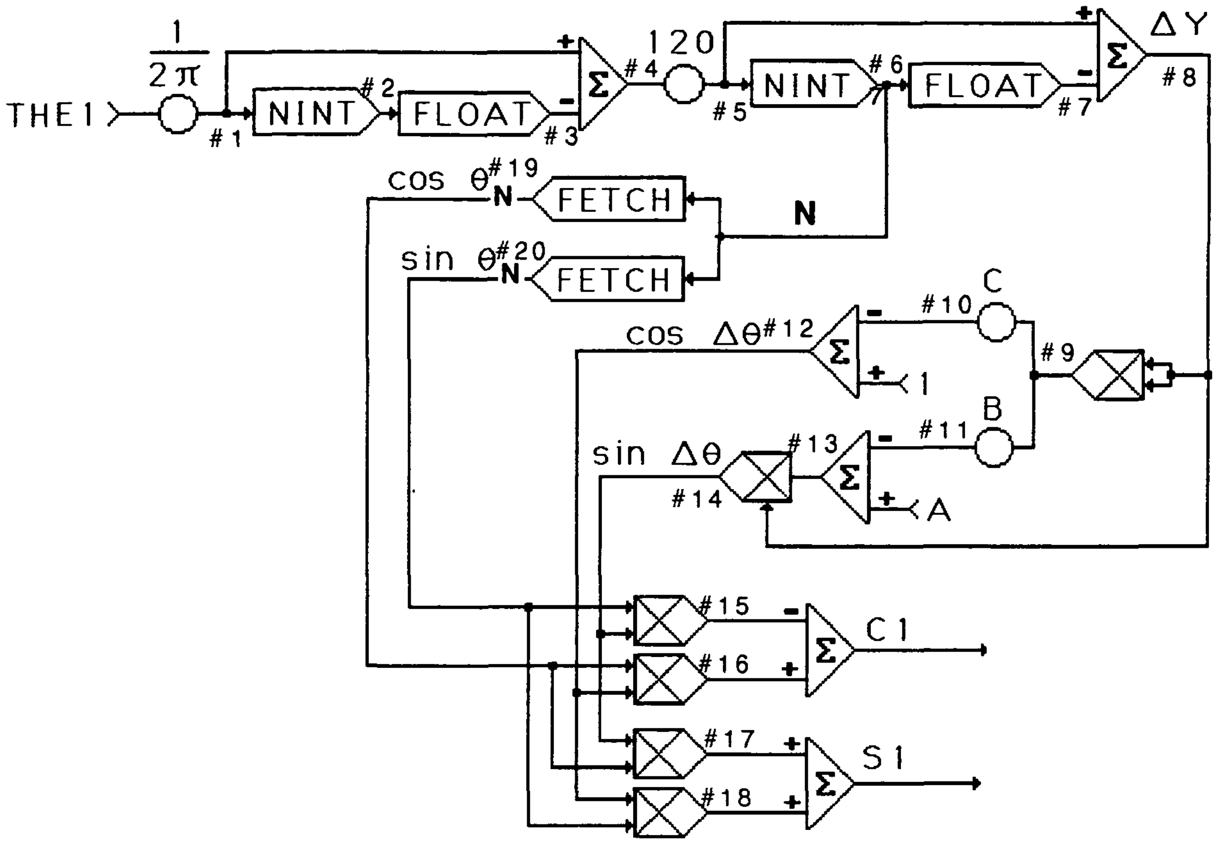


Figure 4e. Expansion of the sine/cosine macro for S1 and C1

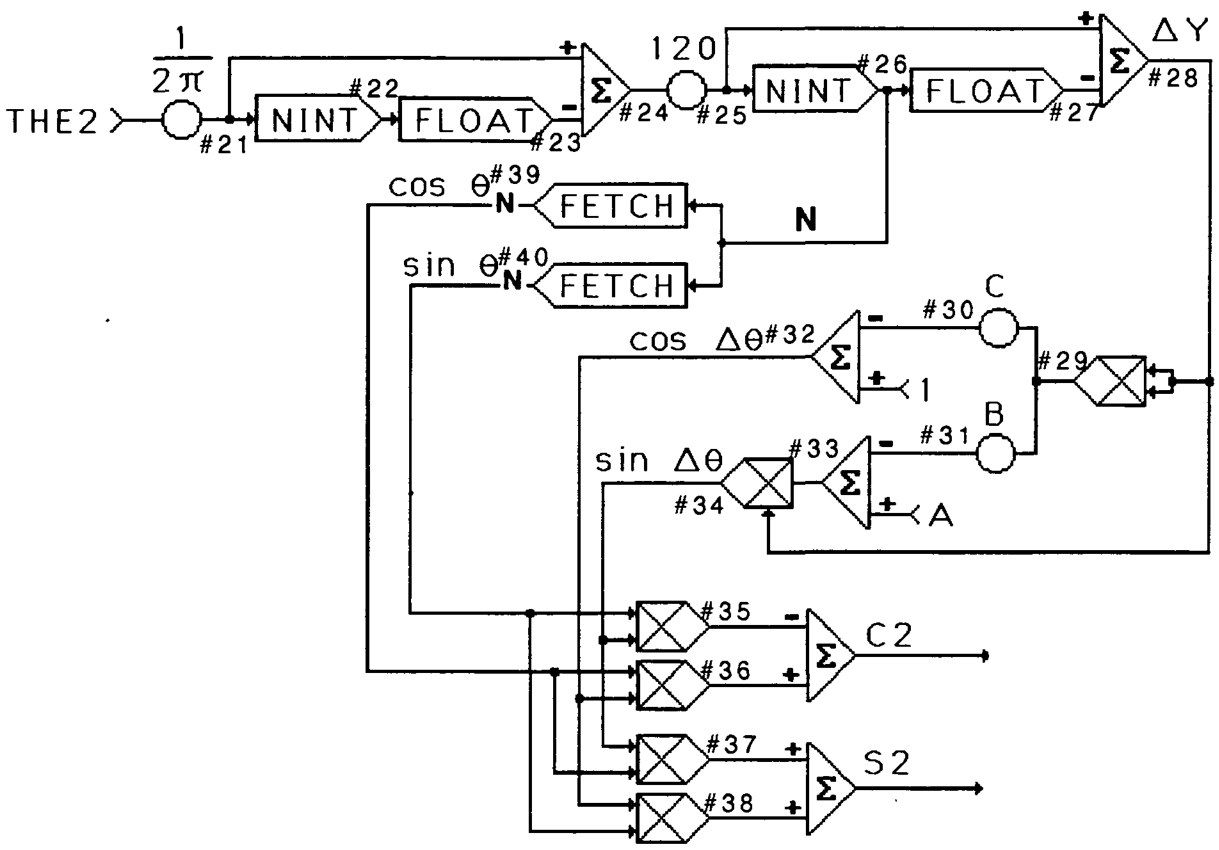


Figure 4f. Expansion of the sine/cosine macro for S2 and C2

After these macros are expanded, we get an RTA program containing 85 variables: 4 state variables, 18 algebraic variables that have explicit names from the source program, and 63 auxiliary variables created by the compiler, representing intermediate terms that the user normally never sees or worries about. From the point of view of the scheduling algorithm, these represent 4 sources of data (the integrators, which are state variables) and 81 tasks to be performed (algebraic variables to be computed).

The prototype software does not include a full CSSL compiler, but rather a simplified version, supporting the subset described in Section 4 of this report. The principal difference between the subset and the full language is in the simplified input syntax for source equations, as described in Section 4.1.1: OUTPUT LIST = OPERATOR NAME(INPUT LIST). This means that an equation such as "W = X + Y -Z" must be written "W = SUM(X, Y, -Z)" to fit the format. This restricted form of input was designed to simplify the design of the parser, so that the bulk of the project effort could be devoted to scheduling algorithms, rather than parsing algorithms. The commercial version of the compiler will contain a parser capable of handling expressions of arbitrary complexity using conventional algebraic notation.

Figure 5 shows the complete RTA program for the Double Pendulum, written in the simplified input syntax used in this project.

```
"Double Pendulum Model."
"George Hannauer, April 26, 1991"
"File name DPEND"

   INCLUDE \RTA\LIB\RTALIB

   @CON, G = 0.980 $ "Acceleration of gravity in MKS units"
   @CON, A1 = 2, A2 = 1, M1 = 5, M2 = 1
   @CON, THE10 = 0, THE1D0 = 0
   @CON, THE20 = 0, THE2D0 = 1
   @CON, P1 = 4, P2 = 2, P3 = 4, P4 = 1
   @CON, P5 = 20, P6 = 1.96, P7 = 0.98, P8 = 9.8
   @CON, P9 = 25, P10 = 11.76, D22 = 1

      "State variables"
         THE1D = INTEG(THE1DD, THE1D0)
         THE1 = INTEG(THE1D, THE10)
         THE2D = INTEG(THE2DD, THE2D0)
         THE2 = INTEG(THE2D, THE20)

      "Trigonometric Functions"
      SCOS(S1, C1, THE1)
      SCOS(S2, C2, THE2)

        "S12 = S1*C2 + S2*C1  = SIN(THE1+THE2)"
         WW1 = MUL(S1, C2)
         WW2 = MUL(S2, C1)
         S12 = SUM(WW1, WW2)

      "Inertial Coefficients"
        "D11 = P9 + P3*C2"
         WW3 = MUL(P3, C2)
         D11 = SUM(P9, WW3)
        "D12 = P4 + P2*C2"
         WW4 = MUL(P2, C2)
         D12 = SUM(P4, WW4)

      "Centrifugal and Coriolis Coefficients"
        "C12 = -P2*S2"
         C12 = MUL(-P2, S2)
         C22 = IDENT(C12)

        "Gravitational Coefficients"
        "G1 = P10*S1 + P7*S12"
         WW5 = MUL(P10, S1)
         G1 = SUM(WW5, G2)
         G2 = MUL(P7, S12)
```

Figure 5a. RTA Listing for Double Pendulum (Sheet 1 of 2)

```
"Torque Computation"
  "H1 = 2*C12*THE1D*THE2D + C22*THE2D*THE2D + G1"
   WW7 = MUL(THE1D, THE2D, C12, TWO)
   @CON, TWO = 2.0
   WW8 = MUL(C22, THE2D, THE2D)
   H1 = SUM(WW7, WW8, G1)
  "H2 = -C12*THE1D*THE1D + G2"
   WW9 = MUL(-C12, THE1D, THE1D)
   H2 = SUM(WW9, G2)

   TAU1 = MUL(-K1, THE1D)
   TAU2 = MUL(-K2, THE2D)

   @CON, K1 = 0.1, K2 = 0.1

"Solve for highest derivatives (angular accelerations)"

"Reciprocal of determinant"
  "RDET = 1.0/(D11*D22 - D12*D12)"
   WW10 = MUL(D11, D22)
   WW11 = MUL(D12, D12)
   DET = SUM(WW10, -WW11)
   RDET = DIV(ONE, DET)
   @CON, ONE = 1.0

  "THE1DD = RDET*(D22*(TAU1-H1) - D12*(TAU2-H2))"
  "THE2DD = RDET*(D11*(TAU2-H2) - D12*(TAU1-H1))"

   EPS1 = SUM(TAU1, -H1)
   EPS2 = SUM(TAU2, -H2)
   WW12 = MUL(D22, EPS1)
   WW13 = MUL(D12, EPS2)
   WW14 = SUM(WW12, -WW13)
   THE1DD = MUL(RDET, WW14)

   WW15 = MUL(D11, EPS2)
   WW16 = MUL(D12, EPS1)
   WW17 = SUM(WW15, -WW16)
   THE2DD = MUL(RDET, WW17)

   @RUNTIME, 9.5

   @OUT, TIME, 0, 10
   @OUT, THE1, 1, 1.0
   @OUT, THE2, 2, 1.0
   @OUT, RDET, 3, 6.0E-2
```

Figure 5b. RTA Listing for Double Pendulum (Sheet 2 of 2)

To illustrate the scheduling algorithm and demonstrate its results, Tables 1 and 2 give the necessary information about each variable. The EFT (Earliest Finish Time) and DST (Down-Stream Time) are calculated by the scheduling compiler for each algebraic variable. The EFT is the earliest time a variable can be calculated, assuming an unlimited number of processors and no bus conflicts. The DST is the length of the longest path "down-stream" of the variable, i.e. depending on it, either directly or indirectly through a chain of other variables.

For any variable, the *slack* is defined as

SLACK = CRITICAL PATH LENGTH - (EFT+DST)

The slack indicates how far a variable is from a critical path—the variable is on a critical path if and only if its slack is zero. Actually, the slack is not used in the scheduling algorithm; it is shown just for interest's sake; it gives one an idea of how much parallelism is available in a given problem. EFT and DST are used, of course; in particular, the scheduler decides what variable to start next, if several are ready, by choosing the one with the longest DST.

Table 1 also includes the START and FINISH times for each variable. These times were obtained from the schedule described in Table 2, which represents a step-by-step tour through the scheduling algorithm for this application.

VARIABLE	EFT	DST	SLACK	START	FINISH
THE1 [θ_1]	0	115	0	—	—
THE1D [θ_1]	0	45	70	—	—
THE2 [θ_2]	0	115	0	—	—
THE2D [θ_2]	0	40	75	—	—
S1 [SIN(θ_1)]	70	45	0	66	71
S2 [SIN(θ_2)]	70	45	0	69	74
C1 [COS(θ_1)]	70	45	0	65	70
C2 [COS(θ_2)]	70	45	0	67	72
S12 [SIN($\theta_1+\theta_2$)]	80	35	0	79	84
D11	80	23	12	78	83
D12	80	23	12	83	88
C12 (=C22)	75	35	5	75	80
G1	90	25	0	89	94
G2	85	30	0	84	89
H1	95	20	0	94	99
H2	90	20	5	90	95
TAU1	5	20	90	8	13
TAU2	5	20	90	9	14
DEL (Δ)	90	13	12	93	98
RDEL (1/Δ)	98	5	12	98	106
THE1DD ($\ddot{\theta}_1$)	115	0	0	119	124
THE2DD ($\ddot{\theta}_2$)	115	0	0	120	125

Table 1a. Variable List for the Double Pendulum (Sheet 1 of 3)

VARIABLE	EFT	DST	SLACK	START	FINISH
#1	5	110	0	0	5
#2	10	105	0	5	10
#3	15	100	0	10	15
#4	20	95	0	15	20
#5	25	90	0	20	25
#6	30	85	0	25	30
#7	35	80	0	30	35
#8	40	75	0	35	40
#9	45	70	0	40	45
#10	50	60	5	47	52
#11	50	65	0	45	50
#12	55	55	5	52	57
#13	55	60	0	50	55
#14	60	55	0	55	60
#15	65	50	0	60	65
#16	65	50	0	57	62
#17	65	50	0	61	66
#18	65	50	0	58	63
#19	33	55	27	30	33
#20	33	55	27	31	34
#21	5	110	0	1	6
#22	10	105	0	6	11
#23	15	100	0	11	16
#24	20	95	0	16	21
#25	25	90	0	21	26
#26	30	85	0	26	31
#27	35	80	0	31	36
#28	40	75	0	36	41
#29	45	70	0	41	46
#30	50	60	5	48	53
#31	50	65	0	46	51
#32	55	55	5	53	58
#33	55	60	0	51	56
#34	60	55	0	56	61
#35	65	50	0	62	67
#36	65	50	0	59	64
#37	65	50	0	63	68
#38	65	50	0	64	69
#39	33	55	27	32	35
#40	33	55	27	33	36
#41	75	40	0	72	77
#42	75	40	0	74	79
#43	75	28	12	73	78
#44	75	28	12	76	81
#45	75	30	10	71	76
#46	5	40	70	2	7

Table 1b. Variable List for the Double Pendulum (Sheet 2 of 3)

VARIABLE	EFT	DST	SLACK	START	FINISH
#47	5	30	80	4	9
#48	80	25	10	82	87
#49	10	35	70	7	12
#50	80	30	5	80	85
#51	5	35	75	3	8
#52	80	30	5	81	86
#53	85	25	5	86	91
#54	85	18	12	85	90
#55	85	18	12	88	93
#56	100	15	0	104	109
#57	95	15	5	95	100
#58	105	10	0	109	114
#59	100	10	5	105	110
#60	105	10	0	110	115
#61	100	10	5	106	111
#62	110	5	0	114	119
#63	110	5	0	115	12

Table 1c. Variable List for the Double Pendulum (Sheet 3 of 3)

Notice that the state variables all have EFT of zero; they are available at the beginning of the frame. The highest derivatives have zero DST—they are terminal variables.

Table 1 indicates that 48 out of 81 algebraic variables have zero slack; in other words, most of the variables are on a critical path. The maximum EFT and the maximum DST are the same—the length of the critical path (in this case, 115 cycles).

There are 81 algebraic variables to be calculated for each derivative evaluation. Four of these are indexed FETCH operations in the Data Memory. The other 77 are operations of the Arithmetic Unit. The Data Memory is not used very heavily in this application, which should not be too surprising, since there are only four indexed memory fetch operations, and no I/O statements.

In a one-module RTA, with a single arithmetic unit, the resource limit is 77 cycles—the application cannot be completed in less time than that. But the critical path limit is 115 cycles. Clearly, adding more modules would not provide a significant speedup; this application is heavily bound by its sequential constraints.

The actual schedule generated by the scheduling compiler is 126 cycles long. This is less than 10% above the critical-path limit, so that the scheduler is over 90% efficient on this application. Efficiency of 90% or better is typical, as shown by the results in Section 3 of this report for the ten benchmark applications run to date.

Table 2 shows how the schedule is developed line-by-line. It works by stepping through each cycle of the computation in the same order that the actual hardware will at run time. On each cycle, we maintain a list of ready variables. A variable is defined as "ready" if its inputs are available, so that the computation of the variable is ready to start as soon as a processor becomes available.

At the start, the only values available are the state variables and the constants. A variable depending only on these is ready at the start. These variables are listed at the beginning of the table.

Since most operations have a latency of five cycles, a variable that was "started" on cycle #N will normally be "finished," i.e. its value will become available, on cycle #N+5. Exception: the indexed memory fetch has a latency of 3 cycles, and the DIVIDE operation in the arithmetic unit has a latency of 8 cycles.

Since most operations have an Initiation Interval of 1, the arithmetic unit will be able to accept a new input on every cycle and produce a new output on every cycle. (Exception: division has a six-cycle latency; a division started on cycle #N will tie up the unit for 6 cycles, and the next operation (which may be another division or any other operation) may be started on cycle #N+6 (as pointed out above, the result does not emerge for two more cycles—on cycle #N+8).

The latency figures are "memory-to-memory," that is, they include time spent in bus transfers within a module. Bus transfers between modules go through the intermodule bus; this transfer takes additional machine cycles. However, since the critical path is strongly dominant here, there is no incentive to use more than one module, even if several are available.

At each step, the algorithm first looks to see whether one or more variables are becoming available at component outputs (naturally, the answer will be "no" on the first few steps, since this is a pipelined machine, but eventually, variables will start emerging).

Occasionally, a cycle will be encountered on which two or more variables appear at once. This is rare in the present program, since most operations use the arithmetic unit, and there is only one such unit in a one-module system. However, there are a few places (e.g. cycles 30 and 31) where two operations are started simultaneously—one in the arithmetic unit and one in the Data Memory. There are also a few places (e.g. cycles 35 and 36) where two variables become available on the same cycle.

In any event, the emergence of a new variable indicates that some of the computations this variable needs can be moved onto the "ready" list. If a variable, say X, becomes available, then each of the operations using X is examined:

• If the operation is a unary operation like FLOAT, SQRT, or ABS, then its variable is put onto the ready list.

• If the variable is a binary operation, like ADD, SUBTRACT, MULTIPLY, DIVIDE, AND, OR, etc., then its *other* input is checked. If it's already available, then the variable is put onto the ready list.

On any given cycle, if any variables are ready, the one with the longest DST is chosen to schedule on the processor. In case of a tie, the present scheme simply picks the first (low-numbered) operation from the list of ready operations with minimal DST.

To facilitate following the schedule, Table 2 contains a column headed "NEWLY READY," which lists each variable (with its DST in parentheses) on the cycle when it first becomes ready.

Variables initially ready, with their DSTs:
#1(110), #21(110), TAU1(20), TAU2(20), #46(40), #47(30), #51(35)

CYCLE	START	FINISH	NEWLY READY
0	#1		
1	#21		
2	#46		
3	#51		
4	#47		
5	#2	#1	#2(105)
6	#22	#21	#22(105)
7	#49	#46	#49(35)
8	TAU1	#51	
9	TAU2	#47	
10	#3	#2	#3(100)
11	#23	#22	#23(100)
12		#49	
13		TAU1	
14		TAU2	
15	#4	#3	#4(95)
16	#24	#23	#24(96)
17			
18			
19			
20	#5	#4	#5(90)
21	#25	#24	#25(90)
22			
23			
24			
25	#6	#25	#6(85)
26	#26	#25	#26(85)
27			
28			
29			
30	#7, #19		#6(50), #19(55), #20(55)
31	#27, #20		#26(85), #39(55), #40(55)
32	#39		

Table 2a. RTA Schedule for Double Pendulum (Sheet 1 of 3)

CYCLE	START	FINISH	NEWLY READY
33	#40	#19	
34		#20	
35	#8	#7, #39	#8(75)
36	#28	#27, #40	#28(75)
37			
38			
39			
40	#9	#8	#9(70)
41	#29	#28	#29(70)
42			
43			
44			
45	#11	#9	#10(60), #11(65)
46	#31	#29	#30(60), #31(65)
47	#10		
48	#30		
49			
50	#13	#11	#13(60)
51	#33	#31	#33(60)
52	#12	#10	#12(55)
53	#32	#30	#32(55)
54			
55	#14	#13	#14(55)
56	#34	#33	#34(55)
57	#16	#12	#36(50), #18(50)
58	#18	#32	#36(50), #38(50)
59	#36		
60	#15	#14	#15(50), #17(50)
61	#17	#34	#35(50), #37(50)
62	#35	#16	
63	#37	#18	
64	#38	#36	
65	C1	#15	C1(45)
66	S1	#17	S1(45)
67	C2	#35	C2(45)
68		#37	
69	S2	#38	S2(45)
70		C1	
71	#45	S1	#45(30)
72	#41	C2	#41(40), #43(28), #44(28)
73	#43		
74	#42	S2	#42(40), C22(35)
75	C22		
76	#44	#45	
77		#41	
78	D11	#43	D11(23)
79	S12	#42	S12(35)
80	#50	C22	#48(25), #50(30), #52(30)

Table 2b. RTA Schedule for Double Pendulum (Sheet 2 of 3)

CYCLE	START	FINISH	NEWLY READY
81	#52	#44	D12(23)
82	#48		
83	D12	D11	#54(18)
84	G2	S12	G2(30)
85	#54	#50	
86	#53	#52	#53(25)
87		#48	
88	#55	D12	#55(18)
89	G1	G2	H2(20), G1(25)
90	H2	#54	
91		#53	
92			
93	DEL	#55	DEL(13)
94	H1	G1	H1(20)
95	#57	H2	#57(15)
96			
97			
98	RDEL	DEL	RDEL(5)
99	----	H1	#56(15)
100	----	#57	#59(10), #61(10)
101	----		
102	----		
103	----		
104	#56		
105	#59		
106	#61	RDEL	
107			
108			
109	#58	#56	#58(10), #60(10)
110	#60	#59	
111		#61	
112			
113			
114	#62	#58	#62(5)
115	#63	#60	#63(5)
116			
117			
118			
119	THE1DD	#62	THE1DD(0)
120	THE2DD	#63	THE2DD(0)
121			
122			
123			
124		THE1DD	
125		THE2DD	**END OF PROGRAM**

Table 2c. RTA Schedule for Double Pendulum (Sheet 3 of 3)

The manually-generated schedule, from the Phase 1 final report, takes 126 cycles. The compiler-generated schedule requires 128. The cost of automatic scheduling is less than 2% in this case.

The Phase 1 final report estimated about 40 cycles more for the four integrations; the actual compiler-generated program takes 34. Hence the overall frame time is 162 cycles, as compared with the original 165-cycle estimate. However, the clock period of the prototype is 50 nanoseconds, while the original estimate was 40. Thus, the overall frame time of the prototype is 8.1 µsec, as compared with the original estimate of 6.6. Except for the slightly slower clock frequency, the speed of the system is within 2% of the value anticipated at the end of the first phase study.

6. CONCLUSIONS

The major goals of the Phase 2 project have been obtained. The feasibility of the original design concept, including both hardware and software, has been established. Both the hardware prototype and the scheduling compiler, supporting a high-level language, function essentially as anticipated.

Electronic Associates is currently proceeding, independent of U.S. Government funds, to develop the RTA into a commercial product. The software is being enhanced to include a more "friendly" user interface, additional MACRO operators, and many other features, some of which are described in Section 4 of this report. The hardware is being enhanced with I/O modules, allowing a single system to include up to 64 A/D channels, 64 D/A channels, 64 discrete logic signals in either direction, and communication with sequential processors, for running mixed parallel/procedural code.

The compiler, loader, and run-time executive programs are being re-hosted to a UNIX work-station, allowing for enhanced graphics capability and a better programming environment.

The product was announced on July 31, 1991, and first deliveries are scheduled for the third quarter of 1992.

NASA
National Aeronautics and Space Administration

Report Documentation Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
4. Title and Subtitle Construction of a Parallel Processor for Simulating Manipulators and Other Mechanical Systems		5. Report Date August, 1991 6. Performing Organization Code
7. Author(s) George Hannauer		8. Performing Organization Report No. 10. Work Unit No.
9. Performing Organization Name and Address Electronic Associates, Inc. 185 Monmouth Parkway West Long Branch, NJ 07764		11. Contract or Grant No. NAS5-30905 13. Type of Report and Period Covered Final Report April 1990/August 1991
12. Sponsoring Agency Name and Address National Aeronautics & Space Administration Washington, DC 20546-0001 Goddard Space Flight Center Greenbelt, MD		14. Sponsoring Agency Code

15. Supplementary Notes

16. Abstract

ABSTRACT

This report summarizes the results of Contract NAS5-30905, awarded under Phase 2 of the SBIR Program. The goal was to demonstrate the feasibility of a new simulation processor. The demonstration consisted of building a hardware prototype, writing a rudimentary compiler, linker, and loader, and programming and running several applications benchmarks. The goal was to determine whether this system could meet its stated objectives: computation speed 10 to 100 times faster than conventional machines, and high-level language programmability.

The objectives were met, and EAI is now proceeding with Phase 3: development of a commercial product. This product is scheduled for introduction in the second quarter of 1992.

17. Key Words (Suggested by Author(s)) Parallel Processing Simulation	18. Distribution Statement Unclassified - Unlimited

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of pages	22. Price
Unclassified	Unclassified	57	